Escape Analysis for Object Oriented Languages. Application to Java™

Bruno BLANCHET
INRIA Rocquencourt
Domaine de Voluceau - BP 105
78153 Le Chesnay Cedex, France
Bruno.Blanchet@inria.fr

ABSTRACT

Escape analysis [27, 14, 5] is a static analysis that determines whether the lifetime of data exceeds its static scope.

The main originality of our escape analysis is that it determines precisely the effect of assignments, which is necessary to apply it to object oriented languages with promising results, whereas previous work [27, 14, 5] applied it to functional languages and were very imprecise on assignments. Our implementation analyses the full Java™ Language.

We have applied our analysis to stack allocation and synchronization elimination. We manage to stack allocate 13% to 95% of data, eliminate more than 20% of synchronizations on most programs (94% and 99% on two examples) and get up to 44% speedup (21% on average). Our detailed experimental study on large programs shows that the improvement comes more from the decrease of the garbage collection and allocation times than from improvements on data locality [7], contrary to what happened for ML [5].

1 INTRODUCTION

Object-oriented languages such as C++ and Java often use a garbage collector (GC) to make memory management easier for the programmer. A GC is even necessary for the Java programming language, since Java has been designed to be safe, so it cannot rely on the programmer to deallocate objects when they are useless. However, garbage collecting data is time consuming, especially with a mark and sweep collector as in the JDK. Therefore stack allocation may be an interesting alternative. However, it is only possible to stack allocate data if its lifetime does not exceed its static scope. The goal of escape analysis is precisely to determine which objects can be stack allocated.

Escape analysis is an abstract interpretation-based analysis [10, 11] which we have already applied to functional languages [5]. However, object-oriented languages have specific features, which make the analysis completely different from the functional version:

- Object-oriented languages use dynamic calls, so before analyzing the code, we must first determine which methods may actually be called at each call point;
- Object-oriented languages make an intensive use of assignments, which must therefore be precisely analyzed, which much complicates our task;
- Object-oriented languages use subtyping, which must be taken into account for the representation of escape information, since it is computed from the types.

Escape analysis has two applications: an object o which does not escape from method m (i.e. whose lifetime is included in m runtime) can be stack allocated in m. Our analysis has been designed such that o is also local to the thread of m, so we need not perform synchronizations when calling a synchronized method on object o. This second application is more specific to Java.

1.1 Related Work

Escape analysis on lists has been introduced by Park and Goldberg [27] for functional languages, and Deutsch [14] has much improved the complexity of their analysis, reducing it to $\mathcal{O}(n \log^2 n)$, with exactly the same results for first-order expressions (there is an unavoidable loss of precision in the higher-order case). He has also suggested several extensions. In [5], we gave a complete implementation of escape analysis on the Caml Special Light compiler (an implementation of ML) with some extensions, and a proof of correctness as well as experimental data.

Mohnen [25, 26] describes a similar analysis, but the analyzed language is restricted to first order and does not handle imperative operations. Hughes [20] already introduces integer levels to represent the escaping part of data. He does not perform stack allocation, but keeps in memory the addresses of data to be deallocated in order to avoid using the GC. The work closest to Hughes' is [21] by Inoue, Seki and

Yagi, who only free the top of lists, but give experimental results.

McDowell [24] gives experimental evidence that there are many opportunities for stack allocation in Java programs, and suggests it would be interesting, but does not actually allocate data on the stack. [28] gives an analysis algorithm that can be applied to stack allocation in object oriented languages, but it is much more costly than ours, and no implementation is mentioned. [15] applies an escape analysis to stack allocation in Java, but considers that an object escapes as soon as it is stored in another object. Our analysis is therefore much more precise. [6] uses escape analysis for synchronization elimination, but our algorithm is more precise on assignments since we can detect objects that are transitively reachable only from local variables. [2] investigates other analyses to eliminate synchronizations, when a monitor is always protected from concurrent access by another monitor, or when the monitored object is thread local. These analyses eliminate fewer synchronizations than ours on the benchmarks. [9] uses an escape analysis based on connection graphs. They are similar to alias graphs and points-to graphs but can be easier summarized to avoid recomputing the escape information when a method is called in different escape contexts. This analysis is applied to stack allocation and synchronization elimination. It is however more costly than ours.

Alias analysis [13], reference counting [18, 19], storage use analysis [29] which is similar to [12, 17, 22, 30] can be applied to stack allocation though at a much higher cost.

Another allocation optimization has been suggested in [1, 3, 31]: region allocation. All objects are allocated in heap regions whose size is not known statically in general, but for which we know when they can be deallocated. Regions can therefore be deallocated without GC. This analysis solves a more general problem than ours, but at the cost of an increased complexity. In fact, on many programs, opportunities for stack allocation outnumber opportunities for region allocation, as noticed in [3].

[16] uses annotated types to describe escape information. The results are not as precise as ours and it only gives inference rules and no algorithm to compute annotated types.

1.2 Overview

In Section 2, we define our escape analysis for Java, and state its correctness theorem. We have done a correctness proof, but we shall not detail it here, as we have chosen to focus on the experimental side in this paper.

Section 3 describes the implementation details of our analysis. It is based on TurboJ, a Java to C compiler designed by the Silicomp Research Institute. It benefits from several extensions: we use inlining to increase the number of stack allocation opportunities. We reuse the space when possible if an allocation occurs in a loop. Our analysis is intermodular, and supports separate compilation of libraries (although precision is improved if we perform global analysis, because we have more precise information on the call graph).

Section 4 is devoted to the experimental study of the speedups. The programs benefit both from improvements on data locality and from a decrease of the GC workload.

Address	Address in a method ($\{0...65535\}$)
NVar	Number of a local variable ($\{0...65535\}$)
Class	Name of a class
Name	Name of a field or a method
SimpleType	Simple type (we restrict ourselves to `int`)
RefType	Reference type (*Class* or array of *Type*)
Type	Type (*SimpleType* $\cup$ *RefType*)
MethodType	Method type ($\mathit{Type}^* \times (\mathit{Type} \cup \{\texttt{void}\})$)
Method	Method ($\mathit{Class} \times \mathit{Name} \times \mathit{MethodType}$)
Field	Field ($\mathit{Class} \times \mathit{Name} \times \mathit{Type}$)

`putfield` <*Field*>	`iastore`
`getfield` <*Field*>	`iaload`
`putstatic` <*Field*>	`aaload`
`getstatic` <*Field*>	`aastore`
`new` <*Class*>	`areturn`
`anewarray` <*RefType*>	`ireturn`
`newarray` <*SimpleType*>	`aconst_null`
`invokevirtual` <*Method*>	`dup`
`instanceof` <*RefType*>	`iload` <*NVar*>
`checkcast` <*RefType*>	`istore` <*NVar*>
`goto` <*Address*>	`aload` <*NVar*>
(other jumps similar)	`astore` <*NVar*>

Figure 1: Java bytecode syntax

But the most important improvements come from the GC. For our benchmarks, we get up to 44% speedup (21% on average) and our analysis can be applied to the largest applications thanks to its very good measured efficiency.

1.3 Notations

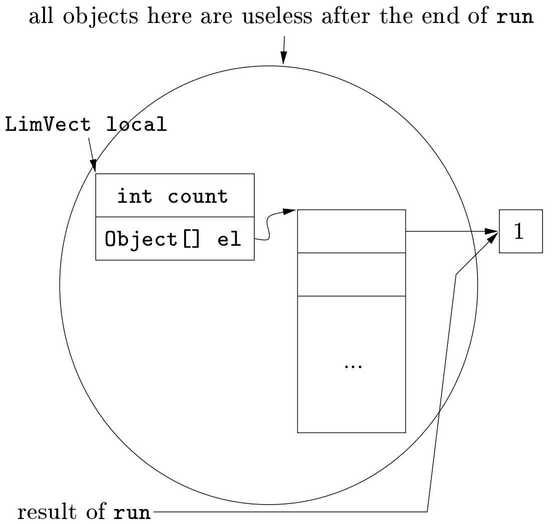
Let S^* be the set of lists whose elements are in S. The empty list is $[]$. The list of elements $p_1, \ldots, p_n$ is $[p_1, \ldots, p_n]$. The list l at the head of which p_1 has been concatenated is $p_1 : l$. The ith element of l is $l(i)$. The function which maps x to $f(x)$ is $\{x \mapsto f(x)\}$. The extension of f which maps x to y is $f[x \mapsto y]$; if f was already defined at x, the new value replaces the old one. In a lattice, the join is $\sqcup$ and the meet is $\sqcap$.

2 ESCAPE ANALYSIS

2.1 Syntax of the analyzed language

Our analysis applies to the Java bytecode. For each method, the bytecode is a sequence of instructions. These instructions will be represented by mnemonics from *The Java Virtual Machine Specification* [23]. For our study, we will restrict ourselves to a small but representative subset of the Java bytecodes, which is listed on Figure 1.

For simplicity, we do not consider `jsr` and `ret` bytecodes and exceptions here, since they raise specific problems, with no connection with escape analysis. Our implementation correctly handles these bytecodes.




```
class LimVect {
  int count = 0;
  Object[] el;

  LimVect(int n) { el = new Object[n]; }

  void put(Object o) { el[count++] = o; }

  Object get(int n) { return el[n]; }

  static Object run() {
    LimVect local = new LimVect(4);
    local.put(new Integer(1));
    return local.get(0);
  }
}
```

Figure 2: Code and structure of objects for Example 2.1

2.2 Our analysis

An object is said to escape from method m if its lifetime exceeds the runtime of method m, as illustrated by the following example.

Example 2.1 Let us consider the simple vector class defined on Figure 2 (its bytecode will be studied in the following). In this example, the `LimVect` object `local` and the `Object` array `local.el` are useless at the end of `run`. The `LimVect` object can be stack allocated. The `Object` array could be stack allocated if the `LimVect` constructor had been inlined in `run`.

The escaping part of an object will be represented by an integer. This integer is said to be the *context* associated with the object, therefore the set of contexts is: $Ctx = \mathbb{N}$. The context of an object can be defined from the type of its escaping part, as follows.

For each object or array type $\tau \in RefType$, let us define the set $Cont(\tau)$ of the types that τ contains (types of the fields for an object, type of the elements for an array). We define

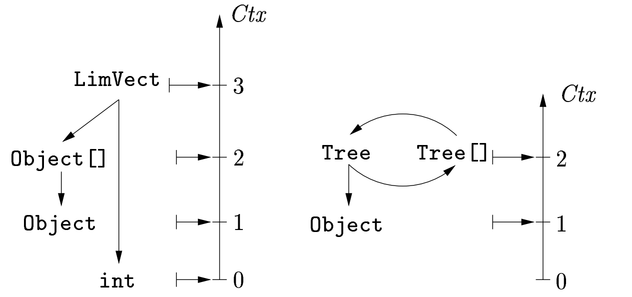


Figure 3: Type heights. An arrow $\tau \to \tau'$ means $\tau' \in Cont(\tau)$ i.e. τ has a field (or element) of type τ'.

the height $\top[\tau]$ of an object or array type τ as the smallest integer such that:

$$\top[\tau] \geq 1 \tag{1}$$

$$\text{if } \tau' \in Cont(\tau), \top[\tau'] \leq \top[\tau] \tag{2}$$

$$\text{if } \tau' \text{ is a subtype of } \tau, \text{ and } \tau \neq \texttt{Object}, \top[\tau'] \leq \top[\tau] \tag{3}$$

if that does not contradict rules (2) and (3),

$$\text{if } \tau' \in Cont(\tau), 1 + \top[\tau'] \leq \top[\tau] \tag{4}$$

Only Rule (2) is necessary for the correctness of the analysis. Rules (3) and (4) are useful the get the best possible precision. By convention, if τ is not an object or an array, $\top[\tau] = 0$, $Cont(\tau) = \emptyset$.

Example 2.2 We assume that no unmentioned subtypes of the types considered in the following examples are defined, so (3) does not apply. Let us consider for example the `LimVect` class. This case is simple: there is no cycle in the *Cont* relation (i.e. no object may contain an object of the same type), so (4) always applies, and the height of a type is 1 plus the maximum height of the types it contains. The results are represented on Figure 3.

Let us consider now the `Tree` class:

```
class Tree { Object element;
             Tree[] sons;      }
```

For `Tree`, there is a strongly connected component in the graph of the *Cont* relation, which contains `Tree[]` and `Tree` (see Figure 3) since $Cont(\texttt{Tree}) = \{\texttt{Object}, \texttt{Tree[]}\}$, and $Cont(\texttt{Tree[]}) = \{\texttt{Tree}\}$. According to (2), $\top[\texttt{Tree}] \geq \top[\texttt{Tree[]}] \geq \top[\texttt{Tree}]$, so we cannot apply (4) between `Tree[]` and `Tree`. (4) gives $\top[\texttt{Tree}] \geq \top[\texttt{Object}] + 1 = 2$, so $\top[\texttt{Tree}] = \top[\texttt{Tree[]}] = 2$.

The escaping part of an object will be represented by the height of its type (for example, since `local.el[0]` escapes in `LimVect.run`, the context for `local` will be $\top[\texttt{Object}] = 1$). We do not directly compute types, because computing integers will lead to a faster analysis and the experiments show that the analysis is still precise.

To compute the escape information, we use a *bidirectional* propagation: E is a backward analysis, whereas E_S is a forward analysis. The analyses E and E_S depend on each other. First, we know that what is read to build the result of a method escapes (for example, if a method reads a field with getfield, and returns this field as its result, then the field escapes). This is computed by analysis E, using a backward propagation (from the result to the parameters). However, because of assignments, objects may also escape because they are stored in static fields, in parameters, or in the result of the method. Backward propagation cannot take into account the fact that an object o escapes because it is stored in a parameter o' of the method for example, because at the point of the assignment, the analyzer would not know that o' is a parameter of the method. Therefore, we have to introduce a forward analysis E_S to cope with assignments (S for *store*). For example, in `LimVect.run()`, E can take into account that elements of `local.el` may be part of the result, but E_S is necessary to take into account the fact that the new `Integer` may be stored in `local.el`.

Abstract values are *contexts transformers*: they take as parameters the escape contexts of the result of the method for analysis E (except when it is `void`, in which case this parameter is omitted. This case will be omitted in the following definitions) and of the parameters for E_S. They yield the escape context associated with the concrete value. Java supports typecasts and subtyping, so the static type of the same object may not be the same during the whole runtime, and we have to remember the assumed type of objects with each context transformer. Therefore, abstract values are: $Val = \cup_{n \in \mathbb{N}}((Ctx^n \to Ctx) \times Type^n \times Type)$.

Notations for escape analysis are summarized on Figure 4. We define the following abstract operations:

- Conversion: The purpose of this operation is to convert a context computed for one type τ to another type τ'. This will enable us to apply the following abstract operations to any type, even if they are normally defined for some types only. Conversions appear explicitly when using `checkcast` to convert types (but in the analysis they are delayed until really necessary), or implicitly when using a subtype as a supertype (in method calls or field accesses). $\text{convert}(\tau, \tau') : Ctx \to Ctx$.

  Let $t[]^k$ be the type $t[]...[]$ with k times $[]$. Assume that τ is an array type with k dimensions, $\tau = t[]^k$ and t is not an array. τ' is also an array type with k' dimensions: $\tau' = t'[]^{k'}$ and t' is not an array.

  $$\text{convert}(\tau,\tau')(n) = (n \sqcap \top[\tau']) \sqcup \begin{cases} \top[t'[]^{k'-i}] & \text{if } i \geq 0 \text{ minimum such that } i \leq k, i \leq k' \\ & \quad \text{and } n = \top[t[]^{k-i}], \\ \top[t'] & \text{if } k > k' \text{ and } \top[t] \leq n \leq \top[t[]^{k-k'}], \\ 0 & \text{if } n < \top[t]. \end{cases}$$

  (For integers, $\sqcup$ is the maximum, $\sqcap$ is the minimum).

  In the particular case when τ and τ' are not array types, this reduces to: if $\top[\tau'] \leq \top[\tau]$, $\text{convert}(\tau,\tau')(n) = n \sqcap \top[\tau']$. Otherwise $\text{convert}(\tau,\tau')(n) = \text{if } n \geq \top[\tau] \text{ then } \top[\tau'] \text{ else } n$.

  For example, $\text{convert}(\texttt{LimVect}, \texttt{Object})(2) = 2 \sqcap \top[\texttt{Object}] = 1$. $\text{convert}(\texttt{Object}, \texttt{LimVect})(1) = \top[\texttt{LimVect}] = 3$.

  By convention, if τ and τ' are not objects, $\text{convert}(\tau,\tau')(0) = 0$ (0 is the only possible context since $\top[\tau] = 0$). If one and only one of τ and τ' is an object type, the conversion is not defined.

- Abstract values are ordered by the preorder relation[1] defined by $(\phi, (\tau_0, ..., \tau_j), \tau) \leq (\phi', (\tau'_0, ..., \tau'_j), \tau') \Leftrightarrow \forall n_0 \in [0, \top[\tau'_0]], ..., \forall n_j \in [0, \top[\tau'_j]]$, $\text{convert}(\tau,\tau')\ (\phi(\text{convert}(\tau'_0,\tau_0)(n_0)), ..., \text{convert}(\tau'_j,\tau_j)(n_j))) \leq \phi'(n_0, ..., n_j)$. $v_1 \leq v_2$ if and only if v_1 gives a more precise escape information than v_2.

- Upper bound: Let $v_1 = (\phi_1, (\tau_0, ..., \tau_j), \tau'_1)$, $v_2 = (\phi_2, (\tau_0, ..., \tau_j), \tau'_2)$. Then $v_1 \sqcup v_2 = ((\text{convert}(\tau'_1, \tau'_2) \circ \phi_1) \sqcup \phi_2, (\tau_0, ..., \tau_j), \tau'_2)$ if $\top[\tau_2] > \top[\tau_1]$. Otherwise, we swap the indices 1 and 2. The choice of the highest type as the type of the result aims at improving the precision of the analysis.

- Construction: Let $v = (\phi, (\tau_0, ..., \tau_j), \tau')$. If v is the abstract value associated with the f field of an object o, $\text{cons}_f(v)$ gives the abstract value associated with o. If v corresponds to the elements of the array o, $\text{cons}_A(v)$ is the abstract value associated with o.

  $\text{cons}_{(C,f,t)}(v) = (\text{convert}(\tau', t) \circ \phi, (\tau_0, ..., \tau_j), C)$. $\text{cons}_A(v) = (\phi, (\tau_0, ..., \tau_j), \tau'[])$.

  For example, let $v = (\{c \mapsto 2\}, (\texttt{int}), \texttt{Object}[])$, which means that the corresponding `Object[]` escapes. Then $\text{cons}_{(\texttt{LimVect},\texttt{el},\texttt{Object}[])}(v) = (\{c \mapsto 2\}, (\texttt{int}), \texttt{LimVect})$ which means that the `LimVect` object does not escape, only its field `Object[]`.

- Restriction: Let $v = (\phi, (\tau_0, ..., \tau_j), \tau')$. If v is the abstract value associated with an object o, $\text{cons}_f^{-1}(v)$ gives the abstract value associated with the field $o.f$. If o is an array, $\text{cons}_A^{-1}(v)$ is the abstract value associated with the elements of o.

  $\text{cons}_{(C,f,t)}^{-1}(v) = (\phi \sqcap \top[t], (\tau_0, ..., \tau_j), t)$, if $t \in Cont(\tau')$. Otherwise, $\text{cons}_{(C,f,t)}^{-1}(v) = ((\text{convert}(\tau', C) \circ \phi) \sqcap \top[t], (\tau_0, ..., \tau_j), t)$.

  $\text{cons}_A^{-1}(v) = (\phi \sqcap \top[t], (\tau_0, ..., \tau_j), t)$ if $\tau' = t[]$. Otherwise, $\text{cons}_A^{-1}(v) = ((\text{convert}(\tau', t[]) \circ \phi) \sqcap \top[t], (\tau_0, ..., \tau_j), t)$ where t can be arbitrarily chosen.

  For example, let $v = (\{c \mapsto 3\}, (\texttt{int}), \texttt{LimVect})$ which means that the `LimVect` object escapes. Then $\text{cons}_{(\texttt{LimVect},\texttt{el},\texttt{Object}[])}^{-1}(v) = (\{c \mapsto 2\}, (\texttt{int}), \texttt{Object}[])$ which means that the `Object[]` may escape.

- Composition: Let $v_k = (\phi_k, (\tau_0, ..., \tau_i), \tau'_k)$, $(\phi', (\tau''_0, ..., \tau''_j), \tau''') \circ (v_0, ..., v_j) = (\phi' \circ (\text{convert}(\tau'_0, \tau''_0) \circ \phi_0, ..., \text{convert}(\tau'_j, \tau''_j) \circ \phi_j), (\tau_0, ..., \tau_i), \tau''')$. It is the usual composition except that we have to convert contexts types when they do not correspond.

If the analyzed method has j parameters of types $\tau_1, ..., \tau_j$ and a result of type τ_0, we note $\emptyset$ all abstract values corresponding with simple types $(\{(c_0, ..., c_j) \mapsto 0\}, (\tau_0, ..., \tau_j), \tau')$ where $\tau' \in SimpleType$. Let $\top_v[(C, f, t)] = (\{(c_0, ..., c_j) \mapsto \top[t]\}, (\tau_0, ..., \tau_j), t)$. Let $\perp_v[E_S] = (\{(c_0, ..., c_j) \mapsto 0\}, (\tau_0, ..., \tau_j), t)$ where $E_S = (\phi, (\tau_0, ..., \tau_j), t)$. This is extended to stacks by $\perp_v[E_1, ..., E_n] = [\perp_v[E_1], ..., \perp_v[E_n]]$ and to local variables by $\perp_v[L] = \{n \mapsto \perp_v[L(n)]\}$. $\mathit{first} = (\{(c_0, ..., c_j,) \mapsto c_0\}, (\tau_0, ..., \tau_j), \tau_0)$. $\forall i < j, P_v(i) = (\{(c_0, ..., c_j) \mapsto c_{i+1}\}, (\tau_0, ..., \tau_j), \tau_{i+1})$.

[1] reflexive and transitive but not antisymmetric

$c \in Ctx = \mathbb{N}$	Escape contexts
$pc \in PC = Method \times Address$	Program counter
$Val = ((Ctx^n \to Ctx) \times Type^n \times Type)(n \in \mathbb{N})$	Abstract domain of a value
$\rho, \rho' \in Env = Method \to \mathbb{N} \to Val$	Environment
$\rho_S, \rho'_S \in Env_S = Method \to Val$	
$S \in Stack = (Val)^*$	Abstract stack
$L \in VarLoc = NVar \to Val$	Abstract local variables
$(S, L) \in State = Stack \times VarLoc$	Abstract state
$\mathrm{In}_S, \mathrm{Out}_S : PC \to State$	Entry and exit forward states[a]
$\mathrm{In}, \mathrm{Out} : PC \to State$	Entry and exit backward states[b]
$Idx = \{\mathrm{Loc}(n), \mathrm{Sta}(n) \mid n \in \mathbb{N}\}$	Index[c]

$E_S : PC \times Idx \to Val$ is: if $\mathrm{In}_S(pc) = (S_S, L_S)$, $E_S(pc, \mathrm{Sta}(n)) = S_S(n)$, $E_S(pc, \mathrm{Loc}(n)) = L_S(n)$
$E : PC \times Idx \to Val$ is: if $\mathrm{In}(pc) = (S, L)$, $E(pc, \mathrm{Sta}(n)) = S(n)$, $E(pc, \mathrm{Loc}(n)) = L(n)$

[a] These are the abstract states for analysis E_S, $\mathrm{In}_S(pc)$ just before the instruction at pc, $\mathrm{Out}_S(pc)$ just after.

[b] These are the abstract states for analysis E, $\mathrm{In}(pc)$ is the state after the analysis of the instruction at pc since E is backward.

[c] Indices are used to represent stack elements and local variables: $\mathrm{Loc}(n)$ stands for the n-th local variable, and $\mathrm{Sta}(n)$ for the n-th element of the stack (0 is the top of the stack). Local variables can also be represented by their name for simplicity. For example, in the method `LimVect.put(Object o)`, `this` is equivalent to $\mathrm{Loc}(0)$, `o` is equivalent to $\mathrm{Loc}(1)$.

Figure 4: Notations for escape analysis

For the `invokevirtual`(C, m, t) bytecode, we need to evaluate the worst escape information for all methods that may be called. We consider that all methods that have a correct signature (m, t) and are defined in a subclass of C may be called. We define $\rho'(m)(i) = \sqcup_{m' \text{ redefining } m}\rho(m')(i)$ and $\rho'_S(m) = \sqcup_{m' \text{ redefining } m}\rho_S(m')$.

The upper bound is defined on abstract states by taking the upper bound for each element of the abstract stack and each abstract local variable. The Java Virtual Machine specification [23, page 130] requires that when merging two operand stacks, the number of values on each stack are identical, and the types of values at corresponding places on the stacks are also identical (or both objects). Therefore, the only rules for the upper bound of elements of the stack are $\emptyset \sqcup \emptyset = \emptyset$ and the upper bound of abstract values $v_1 \sqcup v_2$. On the contrary, two corresponding local variables may contain data of different types. In this case, the local variable becomes unusable after the merge. This corresponds to adding the rules $\emptyset \sqcup c = \emptyset$, $c \sqcup \emptyset = \emptyset$ for the upper bound of local variables. With these rules, $\mathrm{In}_S(pc) = \sqcup_{pc' \text{ predecessor of } pc}\mathrm{Out}_S(pc')$.

For the backward transition $\mathrm{In}(pc) = (S, L) \Rightarrow \mathrm{Out}(pc')$ where pc' is a predecessor of pc, the value of L in pc' is in spirit the upper bound of L on all successors pc of pc', but we have to take into account that a variable may be unusable at pc and usable at pc', so, with $\mathrm{Out}_S(pc') = (S_S, L_S)$, we define $\mathrm{restore}(L, L_S)(n) = L(n)$ if $L(n) \neq \emptyset$, $\mathrm{restore}(L, L_S)(n) = \emptyset$ if $L_S(n) = \emptyset$, $\mathrm{restore}(L, L_S)(n) = \perp_v[L_S(n)]$ in all other cases (the variable was an object at pc', and has become unusable at pc). Then, $\mathrm{Out}(pc') = \sqcup_{pc \text{ successor of } pc'}(S, \mathrm{restore}(L, L_S))$ where $\mathrm{In}(pc) = (S, L)$ and $\mathrm{Out}_S(pc') = (S_S, L_S)$.

Escape analysis is summarized on Figure 5. The following theorem is the main correctness theorem of our analysis. We skip the proof because of its length. Intuitively, we assume that the parameters and result escape, and we test whether the allocated object escapes.

Theorem 2.3 (Stack allocation) *Let us consider a* `new` *in method m, at program counter pc. Let o be the object allocated by that* `new`*. Then with $E(pc, \mathrm{Sta}(0)) = (\phi, (\tau_0, ..., \tau_j), \tau')$, if $\phi(\top[\tau_0], ..., \top[\tau_j]) < \top[\tau']$, o can be stack allocated.*

Theorem 2.4 (Additivity) *Analysis E is additive: with $E(pc, i) = (\phi, (\tau_0, ..., \tau_j), \tau')$, $\forall c, c_0, ..., c_j \in Ctx, \phi(c_0, ..., c_j) \sqcup \phi(c'_0, ..., c'_j) = \phi(c_0 \sqcup c'_0, ..., c_j \sqcup c'_j)$. E_S has the same property.*

Example 2.5 Let us consider the `LimVect.get` method from Example 2.1.

$\mathrm{In}_S(pc)$	a $\mathcal{P}(pc)$	$\mathrm{In}(pc)$
$([], P_v)$	0 `aload_0`	$([], \{0 \mapsto \mathrm{cons}_f(E_2), 1 \mapsto \emptyset\})$
$([P_v(0)], P_v)$	1 `getfield` f	$([\mathrm{cons}_f(E_2)], \perp_v[P_v])$
$([E_0], P_v)$	4 `iload_1`	$([E_2], \perp_v[P_v])$
$([\emptyset : E_0], P_v)$	5 `aaload`	$([\emptyset : E_2], \perp_v[P_v])$
$([E_1], P_v)$	6 `areturn`	$([\mathit{first}], \perp_v[P_v])$

where $m =$ (`LimVect`, `get`, (`int`)`Object`) represents the `get` method, $pc = (m, a)$. The `el` field is represented by $f =$ (`LimVect`, `el`, `Object`[]). The middle column gives the analyzed bytecode.

Abstract values for this method have three parameters: c_0 corresponds to the E escaping part of the result, c_1 to the E_S escaping part of `this` of type `LimVect`, and c_2 to the E_S escaping part of `n` of type `int`. So the abstract values are of the form $(\{(c_0, c_1, c_2) \mapsto \phi(c_0, c_1, c_2)\}, (\texttt{Object}, \texttt{LimVect}, \texttt{int}), \tau)$. When analyzing another method that calls `get`, we need to know what happens if we keep only a part of the parameters or of the result, that is why c_0, c_1, c_2 are parameters and not constants.

The left column corresponds to the forward pass (analysis E_S). Each line gives the state before the execution of the corresponding instruction. We say that an object o store-escapes when, if we store an object o' in o,

$\mathcal{P}(pc)$	Forward transition $\text{In}_S(pc) \Rightarrow \text{Out}_S(pc)$	Backward transition $\text{Out}(pc) \Rightarrow \text{In}(pc)$
entry	$\Rightarrow([], P_v)$	$([], L)\Rightarrow$ **emit:** $\forall i \in \{0, ..., j-1\}, \rho(m)(i) \geq L(i)$
`aload` n	$(S_S, L_S)\Rightarrow(L_S(n) : S_S, L_S)$	$(E : S, L)\Rightarrow(S, L[n \mapsto E \sqcup L(n)])$
`iload` n	$(S_S, L_S)\Rightarrow(\emptyset : S_S, L_S)$	$(\emptyset : S, L)\Rightarrow(S, L)$
`astore` n	$(E_S : S_S, L_S)\Rightarrow(S_S, L_S[n \mapsto E_S])$	$(S, L)\Rightarrow(L(n) : S, L[n \mapsto \perp_v[L_S(n)]])$
`istore` n	$(\emptyset : S_S, L_S)\Rightarrow(S_S, L_S[n \mapsto \emptyset])$	$(S, L)\Rightarrow(\emptyset : S, L[n \mapsto \perp_v[L_S(n)]])$
`ireturn`	$(\emptyset : S_S, L_S)\Rightarrow$	$\Rightarrow(\emptyset : \perp_v[S_S], \perp_v[L_S])$
`areturn`	$(E_S : S_S, L_S)\Rightarrow$ **emit:** $\rho_S(m) \geq E_S$	$\Rightarrow(\mathit{first} : \perp_v[S_S], \perp_v[L_S])$

$\mathcal{P}(pc)$	S_S transition $\text{In}_S(pc) \Rightarrow \text{Out}_S(pc)$	S transition $\text{Out}(pc) \Rightarrow \text{In}(pc)$
`goto a`	$S_S\Rightarrow S_S$	$S\Rightarrow S$
`getfield` f	$E_S : S_S\Rightarrow\text{cons}_f^{-1}(E_S) : S_S$	$E' : S\Rightarrow\text{cons}_f(E') : S$
if f not object	$E_S : S_S\Rightarrow\emptyset : S_S$	$\emptyset : S\Rightarrow\perp_v[E_S] : S$
`putfield` f	$E_S : E'_S : S_S\Rightarrow S_S$	$S\Rightarrow\text{cons}_f^{-1}(E'_S) : \text{cons}_f(E_S) : S$
if f not object	$\emptyset : E'_S : S_S\Rightarrow S_S$	$S\Rightarrow\emptyset : \perp_v[E'_S] : S$
`getstatic` f	$S_S\Rightarrow\top_v[f] : S_S$	$E' : S\Rightarrow S$
`putstatic` f	$E_S : S_S\Rightarrow S_S$	$S\Rightarrow\top_v[f] : S$
`new` c/`aconst_null`	$S_S\Rightarrow E_S : S_S$	$E : S\Rightarrow S$ **emit:** $E_S \geq E$
`(a)newarray` t	$\emptyset : S_S\Rightarrow E_S : S_S$	$E : S\Rightarrow\emptyset : S$ **emit:** $E_S \geq E$
`dup`	$E_S : S_S\Rightarrow E_S : E_S : S_S$	$E' : E'' : S\Rightarrow E' \sqcup E'' : S$
`invokevirtual` m'	$E_{Sn} : ... : E_{S0} : S_S \Rightarrow E'_S : S_S$	$E' : S \Rightarrow \rho'(m')(i) \circ (E', E_{S0}, ..., E_{Sn}), i = n...0 : S$ **emit:** $E'_S \geq \rho'_S(m') \circ (E', E_{S0}, ..., E_{Sn})$
`iaload`	$\emptyset : E_S : S_S\Rightarrow\emptyset : S_S$	$\emptyset : S\Rightarrow\emptyset : \perp_v[E_S] : S$
`aaload`	$\emptyset : E_S : S_S\Rightarrow\text{cons}_A^{-1}(E_S) : S_S$	$E' : S\Rightarrow\emptyset : \text{cons}_A(E') : S$
`iastore`	$\emptyset : \emptyset : E_S : S_S\Rightarrow S_S$	$S\Rightarrow\emptyset : \emptyset : E : S$
`aastore`	$E_S : \emptyset : E'_S : S_S\Rightarrow S_S$	$S\Rightarrow\text{cons}_A^{-1}(E'_S) : \emptyset : \text{cons}_A(E_S) : S$
`instanceof` t	$E_S : S_S\Rightarrow\emptyset : S_S$	$\emptyset : S\Rightarrow\perp_v[E_S] : S$
`checkcast` t	$E_S : S_S\Rightarrow E_S : S_S$	$E : S\Rightarrow E : S$

The current method is m, which has j parameters of types $\tau_1, ..., \tau_j$ and a result of type τ_0. The bytecode at program counter pc is $\mathcal{P}(pc)$. A field $f = (C, f', t)$ is said not to be an object if its type t is a simple type (such as `int`). The generated equations are mentioned after the word "**emit:**". The transition for bytecodes of the second tabular leaves local variables unchanged.

Figure 5: Escape analysis.

o' escapes (this is the intuitive meaning of analysis E_S). At the beginning of the method, the local variables are initialized with the parameters, so the corresponding escape information is P_v, with $P_v(0) = (\{(c_0, c_1, c_2) \mapsto c_1\}, (\texttt{Object}, \texttt{LimVect}, \texttt{int}), \texttt{LimVect})$ which means that if the first parameter (`this`) store-escapes, then local variable 0 also store-escapes. In the same way, $P_v(1) = (\{(c_0, c_1, c_2) \mapsto c_2\}, (\texttt{Object}, \texttt{LimVect}, \texttt{int}), \texttt{int})$ (c_2 will always be 0 since it corresponds to an integer which is a simple type).

The abstract value associated with `this.el` (after the `getfield` f) is $E_0 = \text{cons}_f^{-1}(P_v(0)) = (\{(c_0, c_1, c_2) \mapsto c_1 \sqcap \top[\texttt{Object}[]]\}, (\texttt{Object}, \texttt{LimVect}, \texttt{int}), \texttt{Object}[])$ which means that if `this` store-escapes, `this.el` also store-escapes (the height is limited to $\top[\texttt{Object}[]]$ which is the height of `this.el`). Let $E_1 = \text{cons}_A^{-1}(E_0) = (\{(c_0, c_1, c_2) \mapsto c_1 \sqcap \top[\texttt{Object}]\}, (\texttt{Object}, \texttt{LimVect}, \texttt{int}), \texttt{Object})$ which means that if `this` store-escapes, an element of `this.el` also store-escapes. The equation emitted on `areturn` shows that $\rho_S(m) = E_1$: if we store an object in the result of `get`, it will also be stored in `this`, and so it escapes if `this` escapes.

The right column corresponds to the backward pass (analysis E). Each line gives the state after analysing the corresponding instruction (which is an abstraction of the concrete state before the instruction since the analysis is backward). This column should be read from bottom to top. Let $E_2 = \text{cons}_A(\mathit{first}) = (\{(c_0, c_1, c_2) \mapsto c_0\}, (\texttt{Object}, \texttt{LimVect}, \texttt{int}), \texttt{Object}[])$ which means that if the result escapes, the elements of the array `this.el` also escape (c_0 is at most $\top[\texttt{Object}]$, so only the elements may escape). The escaping part of the first parameter, `this`, is $\text{cons}_f(E_2) = (\{(c_0, c_1, c_2) \mapsto c_0\}, (\texttt{Object}, \texttt{LimVect}, \texttt{int}), \texttt{LimVect})$ which means that the elements of `this.el` (not more than the elements, since $c_0 \leq \top[\texttt{Object}]$) may escape if the result escapes.

Example 2.6 Let m be the `LimVect.put` method from Example 2.1. The result given by the analyzer is

$$
\begin{aligned}
E((m, 0), \texttt{this}) &= E((m, 0), \text{Loc}(0)) \\
&= (\{(c_0, c_1) \mapsto c_1\}, (\texttt{LimVect}, \texttt{Object}), \texttt{LimVect}) \\
E((m, 0), \texttt{o}) &= E((m, 0), \text{Loc}(1)) \\
&= (\{(c_0, c_1) \mapsto 1 \sqcap c_0\}, (\texttt{LimVect}, \texttt{Object}), \texttt{Object})
\end{aligned}
$$

The escaping part of object `o`, $E((m, 0), \texttt{o})$, is the one of the elements of array `this.el` (c_0 corresponds to the value of E_S for `this`) and if the object `o` escapes, the elements of the array `this.el` may also escape. Escape analysis is therefore able to express relations between escaping parts of the different parameters, which gives a very precise information. The fact that `o` and `this` may escape comes from the store in the array `el` (done by `aastore`). The analyzer finds that `local` can be stack allocated: $E((\texttt{LimVect.run}(), 0), \texttt{local}) = (\phi = \{c_0 \mapsto c_0\}, (\texttt{Object}), \texttt{LimVect})$ and $\phi(\top[\texttt{Object}]) = 1 < \top[\texttt{LimVect}]$.

Example 2.7 We do not take into account assignments which decrease the set of aliases, and could therefore improve the precision of the analysis, but at the cost of an increased complexity. For example,

```
aload 0
putstatic (C,f,t)
....
aconst_null
putstatic (C,f,t)
```

If operations "...." do not make the variable 0 or the static field `C.f` escape, and no other thread accesses the static field `C.f`, the variable 0 does not escape. Our analysis cannot discover this property since it believes that variable 0 escapes as soon as it analyses the first `putstatic`. Also notice that escape analysis is not control-flow sensitive (its result does not depend on the order of assignments to fields of objects). Taking such properties precisely into account would require alias analysis which is much more costly than escape analysis.

Experiments have shown that this does not prevent our analysis from giving precise results.

Example 2.8 When there is an assignment `x.f = y` or in bytecode (`x` is in variable 0, `y` in variable 1)

```
1: aload 0
2: aload 1
3: putfield (c,f,t)
```

we have: $E((m, 2), \mathrm{Sta}(0)) = \mathrm{cons}^{-1}_{(c,f,t)}(E_S((m, 2), \mathrm{Sta}(1)))$ which expresses that when `x.f` escapes, `y` escapes, and $E((m, 2), \mathrm{Sta}(1)) = \mathrm{cons}_{(c,f,t)}(E_S((m, 2), \mathrm{Sta}(0)))$ which expresses that if `y` escapes, `x.f` escapes. The first equation seems fairly natural, but the second one may be surprising at first sight. It is however necessary, as the following example shows:

```
t x = new t(); t' y = new t'(); t'' z = new t''();
C.static_field = y;
x.f = y;
x.f.f' = z;
```

When executing the assignment `x.f = y`, when `y` escapes, without this second equation, the E and E_S-escaping parts of `x` would be empty, and we would think that `z` does not escape, which is wrong. In effect, the above program stores `z` in `y` (it is equivalent to `y.f' = z`) and as `y` escapes in the static field `C.static_field`, `z` also escapes.

3 IMPLEMENTATION

Escape analysis has been implemented in the Java-to-C compiler turboJ produced by the Silicomp Research Institute. The compiler is written in Java and so is our analyzer.

3.1 Representation of contexts transformers

In the last section, we have seen that context transformers of escape analysis were of the form $((\mathit{Ctx}^* \to \mathit{Ctx}) \times \mathit{Type}^* \times \mathit{Type})$, $v = (\phi, (\tau_0, ..., \tau_j), \tau')$. Moreover, ϕ is additive (Theorem 2.4), so it can be split into $\phi(c_0, ..., c_j) = g_0(c_0) \sqcup ... \sqcup g_j(c_j)$. Now we still have to represent the monotone functions $\mathbb{N} \to \mathbb{N}$ $g_0, ..., g_j$.

We can notice that the needed elementary operations are built by composition and upper bound from constants, identity, intersection: $n \mapsto n \sqcap \top[t]$, and step: $n \mapsto$ if $n \geq \top[\tau]$ then $\top[\tau']$ else 0 (there is at most one step for conversions where array types do not intervene, several steps in general for conversions between array types). This suggests to use the general form

$$\lambda c.(\text{if } c \geq s \text{ then } i^+ \text{ else } 0) \sqcup (c \sqcap f) \sqcup i$$

Identity will be represented by $\lambda c.c \sqcap \infty$ (in practice, we use an integer larger than all type heights instead of infinity). However, with this general form, we cannot represent all monotone functions. We have to use approximate operations for upper bound, composition and conversion between array types. If $\phi_1(c) = (\text{if } c \geq s_1 \text{ then } i_1^+ \text{ else } 0) \sqcup (c \sqcap f_1) \sqcup i_1$ and $\phi_2(c) = (\text{if } c \geq s_2 \text{ then } i_2^+ \text{ else } 0) \sqcup (c \sqcap f_2) \sqcup i_2$, the approximate upper bound is

$$(\phi_1 \sqcup' \phi_2)(c) = (\text{if } c \geq s_1 \sqcap s_2 \text{ then } i_1^+ \sqcup i_2^+ \text{ else } 0) \\ \sqcup (c \sqcap (f_1 \sqcup f_2)) \sqcup (i_1 \sqcup i_2)$$

The approximate composition is

$$(\phi_1 \circ' \phi_2)(c) = \text{if } c \geq \underline{(\text{if } f_2 \geq s_1 \text{ then } s_1 \sqcap s_2 \text{ else } s_2)} \text{ then} \\ \underline{[(\text{if } f_2 \sqcup i_2^+ \geq s_1 \text{ then } i_1^+ \text{ else } 0) \sqcup (i_2^+ \sqcap f_1)]} \text{ else } 0 \sqcup (c \sqcap \\ \underline{(f_2 \sqcap f_1))} \sqcup \underline{((i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0))}$$

For type conversions from $\tau = t[]^k$ to $\tau' = t'[]^{k'}$, let $S = \{n | \mathrm{convert}(\tau, \tau')(n) > n\}$. If $S = \emptyset$, $\mathrm{convert}(\tau, \tau')(n) = n \sqcap \top[\tau']$ (as $\mathrm{convert}(\tau, \tau')(n) \geq n \sqcap \top[\tau']$ and $\mathrm{convert}(\tau, \tau')(n) \leq \top[\tau']$) which can be exactly represented. Otherwise let $s = \min S$, $i^+ = \mathrm{convert}(\tau, \tau')(\max S)$. Then $\mathrm{convert}(\tau, \tau')(n) \leq (\text{if } n \geq s \text{ then } i^+ \text{ else } 0) \sqcup (n \sqcap \top[\tau'])$ which can be exactly represented.

Finally, we can represent function ϕ by $\phi(c_0, ..., c_j) = i \sqcup (\text{if } c_0 \geq s_0 \text{ then } i_0^+ \text{ else } 0) \sqcup (c_0 \sqcap f_0) \sqcup ... \sqcup (\text{if } c_j \geq s_j \text{ then } i_j^+ \text{ else } 0) \sqcup (c_j \sqcap f_j)$. Theorem 2.3 easily extends to this slightly more approximate representation with only one step (since the computed escape information is always greater in the one step approximation than in the analysis of Section 2). Theorem 2.4 also remains true, since we can only represent additive functions.

Moreover, we use a sparse representation, where only the useful parameters appear, to save up both memory and compute time, as in general a context only depends on a few of the parameters of the method.

3.2 Computing type levels

The height of type τ, $\top[\tau]$, is computed using the rules (1) to (4) given in Section 2.2. To do this, we split the graph of the relation $\tau' \in \mathit{Cont}(\tau) \vee (\tau'$ subtype of $\tau \wedge \tau \neq \texttt{Object})$ in strongly connected components. Inside a strongly connected component, all types have the same height by rules (2) and (3). Between strongly connected components, we add one to the height, following rule (4).

Notice that the set of fields of a class is not given in a single class file. Super-classes also have to be read. We memorize the computations already done, so the total time to compute type heights is $\mathcal{O}(t)$ where t is the number of different types in the analyzed program.

3.3 The analysis algorithm

The analysis algorithm works in several passes, as follows:

For each method m, if it has not been analyzed yet, search the graph of methods that may be called by m, and compute the strongly connected components of this graph.

For each strongly connected component,

- Build the equations for each method:
  - Sort the bytecode topologically by a depth first search. In case of cycle (loop), we cannot sort topologically. In this case, we mark specially the instruction at the end of the back edge during the depth first search.
  - Compute the equations for E_S thanks to a forward pass (in the order determined above). New unknowns are introduced at the end of back edges, since we need the analysis of the current bytecode to know the state before it.
  - Compute the equations for E thanks to a backward pass.
- Solve the equations, by an iterative fixed point solver. To get a satisfying speed, we begin with splitting the dependence graph into strongly connected components, and we solve each component separately. The size of systems is therefore much reduced.
- Prepare the post-transformation, by building the structure giving for each allocation its escape information.

The post-transformation itself is done when generating the C code.

The escape information for the elements of the stack is represented by a list. For local variables, it is represented by a binary tree, each bit of the variable number indicates which branch of the tree should be visited to find the variable. This provides the maximum sharing between data useful for the analysis of one instruction and the following one, and so the fastest program and minimal memory usage.

3.4 Program transformation

3.4.1 Stack allocation

The simpler post-transformation is to replace an allocation `new`, `newarray`, `anewarray` or `multianewarray` by a stack allocation by `alloca` when the allocated object does not escape from the method in which it is allocated. By Theorem 2.3, the object can be stack allocated when $\phi(\top[\tau_0], ..., \top[\tau_j]) < \top[\tau']$, with $E(pc, \mathrm{Sta}(0)) = (\phi, (\tau_0, ..., \tau_j), \tau')$, if the allocation takes place at program counter pc. We can determine whether this condition is realized or we do not know thanks to information computed during the analysis.

3.4.2 Reuse allocated space in loops

If an allocation takes place in a loop, the allocation is done again at each iteration, which leads to increasing the stack size (this may lead to program failure because of stack overflow [8]) whereas without stack allocation, the stack size remained constant. Furthermore, data referenced from the stack are considered as always alive by the GC (the GC is conservative for the stack), which may lead to keep important quantities of useless data is the heap.

A typical example of this situation is the following:

```
String s = "";
for (int i = 0; i < 10000; i++)
  s += i + " ";
  //i.e. s=new StringBuffer(s).append(i)
  //         .append(" ").toString();
```

where all intermediate `StringBuffer`s used to compute `s += i + " "` can be stack allocated, and so all intermediate char arrays are kept uselessly by the GC until the end of the method.

To solve this problem, if an allocated object is always useless after one iteration, we do not use a new space for each allocation, but we reuse the already allocated space. To determine whether the object will still be useful at the next iteration, we consider that all live variables just before the allocation escape, and we test whether the allocated object escapes. The intuitive idea behind this criterion is the following: consider for example the `while` loop

```
while (test) { ... x:  new C() ... }
```

and assume that we unroll that loop:

```
    if (!test) goto end;
    ... x1:  new C() ...
    if (!test) goto end;
    ... x2:  new C() ...
    ...
  end:
```

If the object allocated at `x1` is not live any more at `x2`, i.e. it is not accessible from a variable live at `x2`, i.e. it does not escape when we assume that all the variables live at `x2` escape, then we can reuse at `x2` the memory space allocated at `x1`.

Technically, when analyzing a `new`, we emit, together with the equations already mentioned in the preceding section, the following equation for each variable live before the `new`, $i \in \mathit{Idx}$ being the index of this variable, and t_i the type of this variable:

$$\phi(p_0, ..., p_n) \geq p_k \sqcap \top[t_i]$$

where $E(pc, i) = (\phi, (\tau_0, ..., \tau_n), t_i)$, $n \geq k$, p_k is a new parameter which is 0 for the normal escape analysis, the one which determines whether `alloca` should be introduced, and $p_k = \infty$ when determining if the allocation at pc can reuse the space allocated at the preceding iteration. A new p_k parameter is introduced for each allocation which occurs in a loop.

3.4.3 Inlining

The algorithm can still be improved to discover more stack allocation opportunities: it may happen that an object d cannot be stack allocated in method m because it is still live at the end of m, but becomes dead at the end of method m' which has called m. In this case, we can inline m in m' and allocate d on the stack in m'. This technique can of course be extended to m'' calling m' which calls m, and d dead at the end of m'', etc.

However, we have to determine at the end of which method d will be dead. Assume that we call a method m' with j parameters at pc in method m. Then, we define

$$\begin{aligned} E_C(pc) = &(E(\mathrm{next}(pc), \mathrm{Sta}(0)),\\ &E_S(pc, \mathrm{Sta}(j-1)), ..., E_S(pc, \mathrm{Sta}(0))) \end{aligned}$$

the $(j+1)$-tuple containing the escaping part of the result and parameters of method m' in method m. $E_C(pc)$ is the transformer which converts escaping parts computed in the callee m' to escaping parts computed in the caller m, taking into account that the scope of m is larger than the scope of m'.

Theorem 3.1 (Inlining) *Assume that method m_0 calls m_1 at pc_1, which calls m_2 at pc_2, ..., which calls m_n at pc_n, and that in method m_n, at pc, an object of type τ is allocated. Then, if $\phi(\top[\tau_0], ..., \top[\tau_j]) < \top[\tau]$ where*

$$E(pc, \mathrm{Sta}(0)) \circ E_C(pc_n) \circ ... \circ E_C(pc_1) = (\phi, (\tau_0, ..., \tau_j), \tau)$$

this object can be stack allocated if we inline m_1, ..., m_n in m_0.

Moreover, we wish to inline a call to method m' only if that allows more stack allocations, i.e. if the above criterion is true for some allocations in m' or in methods m' calls, but is wrong without the inlining of the call to m'.

We look for the condition on c_0, ..., c_j such that: $\phi(c_0, ..., c_j) < \top[\tau]$. As analysis E is additive (since only additive functions can be represented by the general form of Section 3.1), this is: $g_0(c_0) \sqcup ... \sqcup g_j(c_j) < \top[\tau]$ where g_i are monotone functions $\mathbb{N} \to \mathbb{N}$. That is $g_0(c_0) < \top[\tau] \wedge ... \wedge g_j(c_j) < \top[\tau]$, i.e. $c_0 \leq c'_0$, ..., $c_j \leq c'_j$, where c'_i is the greatest integer such that $g_i(c'_i) < \top[\tau]$. As all c_i are zero or positive, this defines a right-angled parallelepiped in $\mathbb{R}^{j+1}$ which sides are contained by the coordinate axes (see Figure 6) which is called a $j+1$-cell (this parallelepiped may be empty or infinitely long in certain directions, if $\forall c_i, g_i(c_i) < \top[\tau]$). The set of solutions is represented in the implementation by the null pointer for the empty set or an array of $j+1$ integers corresponding to the sides of the $j+1$-cell.

```
static T f(Vector v1, T[] v2){
  if (v1.size() < 10) {
    T x = new T();
    v1.addElement(x);
    return x;
  } else {
    T y = new T();
    v2[0] = y;
    return y;
  }
}
```

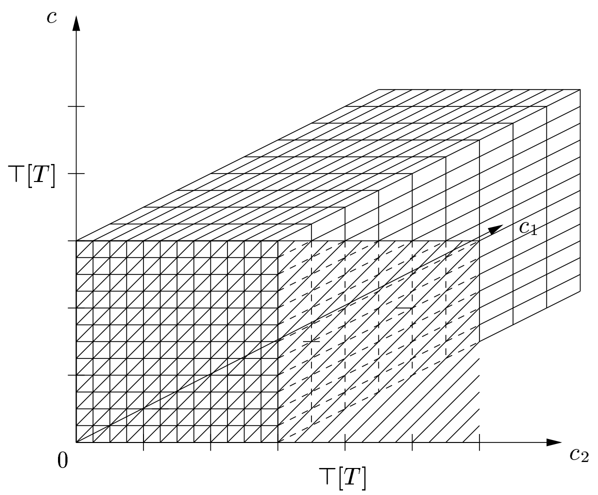


Figure 6: Example of inlining condition. The 3-cell of depth 0 corresponds to the stack allocation condition of `x`, the other one of `y`.

Practically, the sides of the $j+1$-cell can easily be computed. For the first step, we compute the condition on c_0, ..., c_j such that $\phi(c_0, ..., c_j) < \top[\tau]$ where $E(pc, \mathrm{Sta}(0)) = (\phi, (\tau_0, ..., \tau_j), \tau)$. The set of solutions is called $E(pc, \mathrm{Sta}(0))^{-1}(\top[\tau])$. For the following steps, we have to solve $(\phi_0, ..., \phi_{j'})(c_0, ..., c_j) \in S$ where S is the $j'+1$-cell computed at the preceding step, $E_C(pc) = (v_0, ..., v_{j'})$ and $v_k = (\phi_k, (\tau_0, ..., \tau_j), \tau'_k)$. The set of solutions is called $E_C(pc)^{-1}(S)$.

Example 3.2 Consider the method from Figure 6. Let c be the E-escaping part of the result, c_1 and c_2 the E_S-escaping parts of the parameters v_1 and v_2 respectively. The escaping part of `x` is $c \sqcup$ (if $c_1 > 0$ then $\top[T]$ else 0) and that of `y` is $c \sqcup c_2$. Then `x` can be stack allocated if and only if $c < \top[T]$ and $c_1 = 0$. `y` can be stack allocated if and only if $c < \top[T]$ and $c_2 < \top[T]$. This is represented on Figure 6.

Consider now the `g` method:

```
void g(Vector v1) {
```

```
  System.out.println(f(new Vector(), new T[10])
    .toString());
}
```

If the parameter of `T.toString()` does not escape, then `f` can be inlined to stack allocate `y`. `x` may be stack allocated in a caller of `g`, if $c_1 = 0$.

The above representation enables us to build easily the conditions such that a given chain of function inlining should be done. A first idea of algorithm would be to store for each method m' the list of all interesting chains of inlining. This leads to a memory space quadratic in the size of the program in the worst case, which in the case of very large projects may lead to memory overflow.

So we have to keep only a summary of all stack allocation opportunities in the methods called from m. However, allocation conditions may be represented by complex sets and we cannot guarantee that they can be represented in a small enough space. To solve this problem, we use heuristics. When these heuristics do not give the answer to the question "should we inline ?", we make a complete and exact computation, at the cost of speed. In practice, this case will be rare enough for the system to be very efficient.

We store for each method m the following information:

- m.inter_cond is the $j+1$-cell such that all stack allocations can be done if the entry escaping parts are in it (the entry escaping parts are the escaping parts of the parameters and the result of m, coming from the caller of m — E_S for the parameters, E for the result).
- m.maxvol_cond is a $j+1$-cell which corresponds to a stack allocation which cannot be done if we do not inline m, and of the greatest possible volume (so that it has the greatest probability to be realized if we inline m). The entry escaping parts of m are in m.maxvol_cond if and only if this stack allocation becomes possible if we inline m. The volume of a $j+1$-cell p will be denoted $vol(p)$.
- m.englob_cond is a $j+1$-cell which contains all $j+1$-cells corresponding to possible stack allocations that we are not sure to do without inlining m. If p_1 and p_2 are two $j+1$-cells, we call contains(p_1, p_2) the smallest $j+1$-cell which contains p_1 and p_2 (it can be computed by taking the maximum of the sides of p_1 and p_2).

Let $C = (\phi_0, ..., \phi_j)(\top[\tau'_0], ..., \top[\tau'_{j'}])$ where $E_C(pc_i) \circ ... \circ E_C(pc_1) = (v_0, ..., v_j)$ and $v_k = (\phi_k, (\tau'_0, ..., \tau'_{j'}), \tau_k)$ be the entry escaping parts of method $m = m_i$. If $C \in m$.maxvol_cond, we know that we should inline the call to m. If $C \notin m$.englob_cond, we know that it is useless to inline m. Only when $C \in m$.englob_cond $-$ m.maxvol_cond, we still do not have the answer, and we have to compute it by looking at all possible stack allocations.

Those $j+1$-cells can be computed by the algorithm of Figure 7 (the current method is m, its parameters are of types $\tau_1, ..., \tau_j$, its result is of type τ_0; the parameters of m' are of types $\tau'_1, ..., \tau'_{j'}$, its result is of type τ'_0).

Example 3.3 Let us consider again Example 3.2. `f`.englob_cond is the 3-cell $c < \top[t]$. `f`.maxvol_cond is $c < \top[T], c_2 < \top[t]$ (associated with `y`). `f`.inter_cond is $c < \top[T], c_1 = 0, c_2 < \top[t]$.

The entry context of `f` when analyzing `g` will be $c = 0, c_1 = \top[\texttt{Vector}], c_2 = 0$, so our algorithm correctly detects that `f` should be inlined, as this point is in `f`.maxvol_cond.

AddCond(pc, englob_cond, maxvol_cond, inter_cond) =
  if $(0, ..., 0) \in$ englob_cond **then**
    m.inter_cond = m.inter_cond $\cap$ inter_cond
    if $(\top[\tau_0], ..., \top[\tau_j]) \notin$ maxvol_cond
    and vol(maxvol_cond) $>$ vol(m.maxvol_cond) **then**
      m.maxvol_cond = maxvol_cond
    m.englob_cond = contains(m.englob_cond, englob_cond)
    Add (pc, englob_cond, maxvol_cond, inter_cond)
      to m.inline_info

For each call to a method m' at pc,
  if $(0, ..., 0) \in m'$.englob_cond
  and $(\top[\tau'_0], ..., \top[\tau'_{j'}]) \notin m'$.inter_cond **then**
    AddCond(pc, $E_C(pc)^{-1}$(m'.englob_cond),
      $E_C(pc)^{-1}$(m'.maxvol_cond),
      $E_C(pc)^{-1}$(m'.inter_cond))
For each allocation of an object of type τ at pc,
  $cond = E(pc, \mathrm{Sta}(0))^{-1}(\top[\tau])$
  AddCond(pc, $cond$, $cond$, $cond$)

Figure 7: Computing inlining conditions

3.4.4 Synchronization elimination

A method only needs to be synchronized if the object on which it is called can be accessed by several threads. Objects accessible from several threads are objects reachable from static fields and from `Thread` objects. Objects reachable from static fields are considered as escaping by our analysis, and the thread creation method `Thread.start()` makes its parameter escape. Therefore, if our analysis finds that an object does not escape from a method m, it can be accessed only by one thread, the thread in which it is allocated. So there is no need to synchronize methods called on this object. This is an important optimization as synchronization is a costly operation in the JDK.

We perform two transformations.

- First, assume that we call a synchronized method m_n on an object o of type τ. Assume that m_n has k parameters (including the object o on which it is called), and m_0 = `main` or `Thread.run` has j parameters of types $\tau_1, ..., \tau_j$ and a result of type τ_0. If our analysis can determine that for all call chains m_0 = `main` or `Thread.run` calls m_1 at pc_1, ..., which calls m_n at pc_n, o does not escape from m_0, i.e.:

$$\sqcup_{pc_1,...,pc_{n-1}} \phi_{pc_1,...,pc_n}(\top[\tau_0], ..., \top[\tau_j]) < \top[\tau] \text{ where}$$

$$E_S(pc_n, \mathrm{Sta}(k-1)) \circ E_C(pc_{n-1}) \circ ... \circ E_C(pc_1) =$$

$$(\phi_{pc_1,...,pc_n}, (\tau_0, ..., \tau_j), \tau)$$

  then we call a copy of the method m_n without synchronization. Choosing the largest possible scope m_0 gives the best results, since fewer objects escape.

- Second, before acquiring a lock on an object o, we test at runtime whether o is on the stack, and if it is, we skip the synchronization. This technique could be extended by marking at allocation time objects that are detected to be local to one thread, and testing this mark before acquiring a lock.

3.5 Complexity

Size of the bytecode	n
Number of local variables	l
Height of the stack	s
Number of parameters of a method	p
Number of parameters of a context	p'
Maximum type height	H

Equations building	$\mathcal{O}(n(l+s)p')$
Number of equations	$n_e = \mathcal{O}(n(l+s))$
Number of unknowns	$n_u = \mathcal{O}(n(l+s))$
Number of iterations	$n_i = \mathcal{O}(n_u p' H) = \mathcal{O}(n(l+s)p'H)$
Equations solving	$\mathcal{O}(n_e pp' n_i) = \mathcal{O}(n(l+s)pp'n_i)$
Posttransformation	$\mathcal{O}(np)$

Figure 8: Notations and results on the complexity of escape analysis

When out-of-loop stack allocation is activated, contexts transformers may have more parameters than methods, hence the distinction between p' and p. If out-of-loop stack allocation is disabled, $p' = p$.

The analysis time of a bytecode depends on the considered bytecode: an upper bound on the whole stack and local variables takes $\mathcal{O}((l+s)p')$ time, a method call (`invoke`) $\mathcal{O}(pp')$, a local variable access $\mathcal{O}(\log l)$, an upper bound, conversion or minimum on contexts $\mathcal{O}(p')$. The total complexity of equation building is therefore $\mathcal{O}(n(l+s)p')$ (because $p \leq l$ so the cost of a method call is dominated by the one of an upper bound).

The number of generated equations or unknowns is $\mathcal{O}(n(l+s))$ (an upper bound generates $\mathcal{O}(l+s)$ equations, the other operations generate a constant number of equations).

If the system contains n_u unknowns, the number of iterations n_i is less than the height of the lattice of n_u-tuples of contexts transformers, so $n_i = \mathcal{O}(n_u p' H)$. In practice, the number of iterations is very small (at most 17 in our benchmarks. 44.8% of equations are iterated at most twice, 91.3% at most 7 times, 98.4% at most 10 times. On average, each equation is iterated 3.9 times). This is possible thanks to the splitting of equations systems into strongly connected components. On the other hand, one iteration costs the compute time of equations, which is at most $\mathcal{O}(n_e pp')$. The complexity of the solving is therefore $\mathcal{O}(n_e pp' n_i)$. This complexity which sounds large is much reduced in practice by using a sparse representation for contexts transformers, which ensures that the number of parameters of a context is in fact small compared to p in most cases.

The post-treatment of a method call (when inlining is activated) or of an allocation is in $\mathcal{O}(p)$ if we can decide to inline or not with our fast algorithm. In this case, the post-treatment is dominated by the generation of equations.

Memory size decrease (%)			Size
`alloca`	No loops	All	(Mb)
dhry	95	95	5.1
Symantec	84	84	40.2
javac	13	13	6.5
turboJ	27	29	23.2
JLex	25	26	1.7
jess	21	21	1.9
javacc	20	43	7.5

Table 2: Stack allocated memory

If our fast algorithm does not give the answer, the post-treatment of a call may take time up to $\mathcal{O}(n'p)$ where n' is the number of allocations in all methods called from the program point, following any call chain. This case is very rare and can be neglected (it appends 21 times only in all the tested benchmarks, which represent more than 2 Mb of classes).

Finally, the complexity is therefore $\mathcal{O}(n(l+s)pp'n_i) = \mathcal{O}(n^2(l+s)^2 pp'^2 H)$, with n_i small in practice.

4 BENCHMARKS

Benchmarks have been conducted on a 233 MHz Pentium MMX, 128 Mb RAM, primary cache 16 kb instructions and 16 kb data, secondary cache 512 kb instructions and data, under Linux. Benchmark programs are listed on Table 1.

4.1 Results

Each program has been tested in two stack allocation configurations: in the first configuration called "No loops" all stack allocations done in a loop must reuse the same memory space. In the second configuration "All", `alloca` is allowed even in a loop. The last configuration may stack allocate more objects, but may lead to stack overflows. In the following benchmarks, inlining is allowed for methods of less than 150 bytes.

Table 2 gives the percentage of stack allocated data in each program. The last column represents the total size of allocated data in the program. Table 3 is similar but deals with number of objects instead of size. Table 4 gives the percentage of eliminated synchronizations at runtime. The left column represents the part of eliminated synchronizations that we would get without testing at runtime whether objects are in the stack or not. The middle column gives the part of eliminated synchronizations with this dynamic test. The right column is the total number of synchronizations (without elimination) at runtime.

In Dhrystone, we mainly stack allocate arrays of one integer, which are used as references to integers. Symantec is a set of small benchmarks which contains particularly a version of Dhrystone, in which we do the same optimizations. It also contains a function which uselessly allocates many nodes of a tree, which can be stack allocated. We also manage to eliminate all synchronizations corresponding to synchronized calls to a random number generator.

Benchmark programs		Size(kb)
dhry	Dhrystone	6
Symantec	Set of small benchmarks	19
javac	Java compiler (jdk 1.1.5) compiling jBYTEmark	600
turboJ	Java to C compiler from Silicomp RI compiling jBYTEmark	788
JLex	Lexer generator (v. 1.2) running `sample.lex`	89
jess	Expert system (v. 4.1), solving `fullmab.clp`	402
javacc	Java parser generator (v. 0.8pre2) generating a Java parser	497

Table 1: Benchmark programs. The size is the total size of the `.class` files, Java standard library excluded.

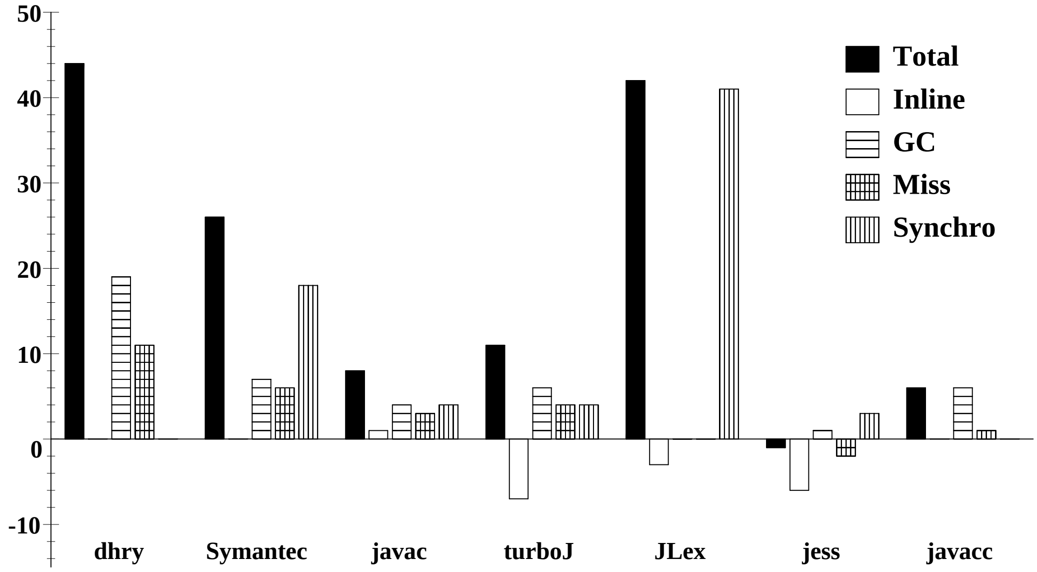


Figure 9: Speedup without `alloca` in loops.

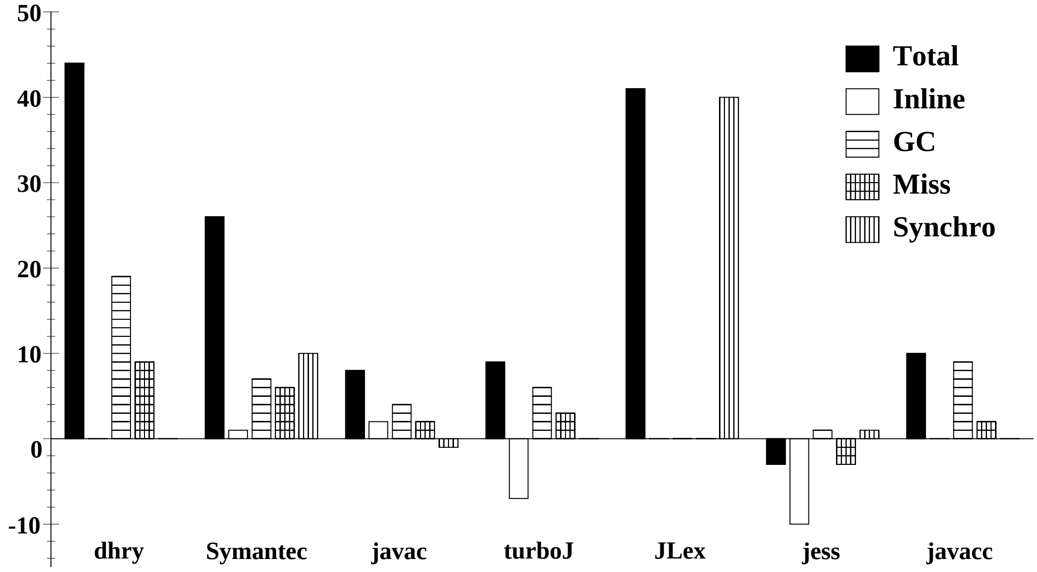


Figure 10: Speedup with all possible stack allocations, even in loops

Stack allocated objects (%)			Total number
`alloca`	No loops	All	of objects (K)
dhry	99	99	600
Symantec	99	99	1199
javac	20	20	160
turboJ	36	37	394
JLex	29	30	40
jess	25	25	43
javacc	17	21	162

Table 3: Stack allocated objects

Eliminated synchro. (%)			Thousands
	(1)	(2)	of synchro.
dhry	–	–	0
Symantec	99	99	2520
javac	5	31	430
turboJ	21	46	587
JLex	78	94	1989
jess	14	21	158
javacc	2	3	1075

(1) Eliminated synchronizations without dynamic "is in stack" test.
(2) Eliminated synchronizations with dynamic "is in stack" test.

Table 4: Eliminated synchronizations

In turboJ and javacc, we mainly stack allocate strings and arrays of characters. Our inlining algorithm is useful to allocate such data on the stack, since they are generally allocated in `StringBuffer.ensureCapacity` and `StringBuffer.toString`, and of course still live at the end of these methods (char arrays stack allocated thanks to inlining represent 4% of data for javac, 20% for turboJ, 29% for javacc). The difference between the "No loops" and the "All" configurations is only important for javacc.

Table 5 indicates the speedups. The last column is the runtime without any optimization, for reference. The mean in the bottom row is the geometric mean. Figures 9 and 10 also give the speedup percentage (Total bar), and they detail the reasons of the speedup. The GC bar gives the part of the speedup percentage that comes from a decrease of GC time. The Inline bar gives the part of the speedup that comes from inlining of methods (inlining is done as if there were stack

Speedup (%)			Time
`alloca`	No loops	All	(ms)
dhry	44	44	2752
Symantec	26	26	19051
javac	8	8	3981
turboJ	11	9	19449
JLex	42	41	3418
jess	-1	-3	2923
javacc	6	10	7922
mean	21	21	

Table 5: Speedups

allocation, but all allocations are done in the heap). The Miss bar gives the contribution of the cache misses to the speedup. This is an estimation based on a cost of 165 ns for a read miss and 70 ns for a write miss, measured experimentally. The Synchro bar gives the part of the speedup that comes from synchronization elimination. A synchronization costs 790 ns when the monitor is in the monitor cache, and 1450 ns when it is not.

The speedups correspond with what could be expected knowing the percentage of stack allocated data and of synchronization elimination. We get high speedups for dhry (thanks to stack allocation), for JLex (thanks to synchronization elimination), and for Symantec. Speedups about 10% are obtained for javac, turboJ and javacc. Inlining has negative effects for turboJ and jess. This may come from an increase of the code size, which leads to more traffic for loading the code from memory. Inlining also leads to changes in register allocation which may affect performance. This negative effect of inlining explains the slowdown that we obtain for jess. For JLex, stack allocating data gives negligible speedups, because the GC time was very small anyway.

The speedups that come from stack allocation mainly have three causes: reduced allocation time, better data locality, decrease of the GC workload. The graphs show that the part of the speedup coming from the GC is important (about half of the speedup not coming from synchronization elimination). This is logical since the JDK uses a mark and sweep garbage collector, which is not efficient for short lived data, which can precisely be stack allocated. The better data locality is responsible for about a quarter of the speedup. The rest, i.e. about a quarter comes from allocation time. Stack allocating an object just requires moving the stack pointer, whereas heap allocating it requires scanning the list of free blocks, and acquiring a lock, which is much less efficient.

For javacc, the speedup is more important in the "All" configuration, in which more allocations can be done in the stack. In the "All" configuration, the stack size increases by a factor 10 for the javacc program, whereas in the "No loops" configuration the increase is limited to only 75%. This shows that it may be useful to limit ourselves to the "No loops" configuration to avoid stack overflows (even if it does not happen in our benchmarks). The stack size increase is 129% for JLex in the "All" configuration. It is small in all other cases (at most 31%).

The left part of Table 6 gives the speedup percentage on the GC time, and in the last column the GC time for reference. The speedup on the GC time is similar to the percentage of stack allocated data. The exceptions are javac which does a single GC without stack allocation, and none with stack allocation, and turboJ and JLex for which the GC time is less reduced than what we could expect. This comes from the fact that stack allocated data are mainly short lived data which are scanned only once by the GC, and so have a smaller cost for the GC than data which cannot be stack allocated (they may be scanned in several GC cycles if they remain alive). Data allocated in turboJ and JLex live longer than in the other benchmarks.

The table about cache misses shows that stack allocation reduces the number of read misses (except for jess). But when there is only a few stack allocated objects, the number of write misses increases. This seems to be linked with the fact that the Pentium MMX has a write-through cache (This

GC speedup (%)			GC time
alloca	No loops	All	(ms)
dhry	100	100	549
Symantec	97	97	1552
javac	100	100	196
turboJ	15	15	8474
JLex	1	1	56
jess	27	24	196
javacc	29	41	1811

	Read miss			Write miss		
	Decrease (%)		Thousands	Decrease (%)		Thousands
alloca	No loops	All	of misses	No loops	All	of misses
dhry	47	47	3249	20	1	3248
Symantec	33	32	14955	51	49	11231
javac	8	8	6533	6	3	8235
turboJ	16	13	27845	3	-2	13407
JLex	1	1	2276	2	-0	2574
jess	-6	-6	5980	-3	-12	4757
javacc	7	14	10876	-2	-7	5948

Table 6: Effects on the GC and the cache.

does not happen on a Pentium II).

4.2 Analysis speed

We have measured the analysis time and compilation overhead, as a percentage of the total compilation time without stack allocation. The compilation overhead comes from inlining and synchronization elimination, which lead to more code generation. In these tests, the library is not reanalyzed when analyzing each benchmark (whereas it was in the preceding tests, but it does not make sense to compare the analysis time of a large amount of code with the compilation time of a small program). These tests have been run with TurboJ bootstrapped.

	An (%)	*Gen* (%)	*Comp* (%)	*Time* (s)
dhry	3	3	2	12.2
Symantec	4	6	2	28.4
javac	13	36	6	487.9
turboJ	11	23	11	634.8
JLex	8	8	7	67.3
jess	9	23	9	641.3
javacc	6	9	17	657.6

Time is the total compilation time without stack allocation.
An is the escape analysis time, as a percentage of *Time*.
Gen is the C code generation overhead, as a percentage of *Time*. *Gen* includes *An*.
Comp is the C compilation overhead, as a percentage of *Time*.

These results show that our analysis takes about 10% of the C compilation time, and the total compilation overhead generated by our optimizations is about 34%.

5 CONCLUSION

We have extended escape analysis from Park and Goldberg [27], Deutsch [14] and Blanchet [5] to allow a precise treatment of assignments, and to support subtyping and inheritance. Our implementation can analyze the full Java language.

Our study has shown the feasibility of escape analysis for object oriented languages. The example of Java shows that it gives high speedups (21% on average) at a reasonable analysis cost (the analysis takes about 10% of the compilation time, the total compilation overhead is about 34%).

Since Java uses a mark and sweep garbage collector, the main reasons for speedups due to stack allocation are a decrease of the GC workload and of the allocation time. Improvements on data locality also contribute to the speedups, but to a less important extent. The best results are obtained when stack allocating an object is allowed even with multiple allocations in a loop, when it actually allows more stack allocation. This solution is unsafe, as it may cause stack overflows, but it does not happen in the considered benchmarks. Inlining of small functions increases stack allocation opportunities. Synchronization elimination also gives impressive speedups, by eliminating most of the cost of synchronization (JLex, Symantec).

Acknowledgements

Many thanks to Alain Deutsch for his help during this work, to Patrick Cousot for helpful comments on a draft of this paper and to the Java team of the Silicomp Research Institute for providing TurboJ.

REFERENCES

[1] Aiken, A., Fähndrich, M., and Levien, R. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *ACM SIGPLAN Conference on Programming Language, Design and Implementation (PLDI'95)* (San Diego, California, June 1995), pp. 174–185.

[2] Aldrich, J., Chambers, C., Sirer, E. G., and Eggers, S. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Static Analysis Symposium (SAS'99)* (Sept. 1999).

[3] Birkedal, L., Tofte, M., and Vejlstrup, M. From Region Inference to von Neumann Machines via Region Representation Inference. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1996), pp. 171–183.

[4] Blanchet, B. Garbage Collection statique. DEA report, INRIA, Rocquencourt, Sept. 1996.

[5] Blanchet, B. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *ACM*

SIGACT-SIGPLAN Conference on Principles of Programming Languages (POPL'98) (San Diego, California, Jan. 1998), ACM, pp. 25–37.

[6] BOGDA, J., AND HÖLZLE, U. Removing Unnecessary Synchronization in Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)* (Nov. 1999).

[7] CARR, S., MCKINLEY, K. S., AND TSENG, C.-W. Compiler Optimizations for Improving Data Locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1994), pp. 252 – 262.

[8] CHASE, D. R. Safety considerations for storage allocation optimizations. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (22-24 June 1988), ACM Press, pp. 1 – 10.

[9] CHOI, J.-D., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. Escape Analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)* (Nov. 1999).

[10] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Languages* (Jan. 1977), pp. 238 – 252.

[11] COUSOT, P., AND COUSOT, R. Systematic Design of Program Analysis Frameworks. In *Sixth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1979), pp. 269 – 282.

[12] DEUTSCH, A. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Seventeenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1990), pp. 157 – 168.

[13] DEUTSCH, A. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation* (20-24 June 1994), ACM Press, pp. 230 – 241.

[14] DEUTSCH, A. On the Complexity of Escape Analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1997), pp. 358 – 371.

[15] GAY, D., AND STEENSGAARD, B. Stack Allocating Objects in Java. `http://research.microsoft.com/apl`.

[16] HANNAN, J. A Type-based Analysis for Stack Allocation in Functional Languages. In *Proceedings of the Second International Static Analysis Symposium (SAS '95)* (Sept. 1995), vol. 983 of *Lecture Notes in Computer Science*, Springer, pp. 172 – 188.

[17] HARRISON, W. The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation 2* (1989), 176 – 396.

[18] HEDERMAN, L. Compile Time Garbage Collection Using Reference Count Analysis. Tech. Rep. Rice COMP TR88-75, Rice University, Houston, Texas, Aug. 1988.

[19] HUDAK, P. A Semantic Model of Reference Counting and its Abstraction (Detailed Summary). In *Proceedings of the 1986 ACM Conference on LISP and functional programming* (Aug. 1986), pp. 351 – 363.

[20] HUGHES, S. Compile-Time Garbage Collection for Higher-Order Functional Languages. *J. Logic Computat. 2*, 4 (1992), 483 – 509.

[21] INOUE, K., SEKI, H., AND YAGI, H. Analysis of Functional Programs to Detect Run-Time Garbage Cells. *ACM Transactions on Programming Languages and Systems 10*, 4 (Oct. 1988), 555 – 578.

[22] JONES, N. D., AND MUCHNICK, S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Nineth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1982), pp. 66 – 74.

[23] LINDHOLM, T., AND YELLIN, F. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, 1996.

[24] MCDOWELL, C. E. Reducing garbage in Java. *ACM Sigplan Notices 33*, 9 (Sept. 1998), 84–86.

[25] MOHNEN, M. Efficient Closure Utilisation by Higher-Order Inheritance Analysis. In *Static Analysis Symposium (SAS'95)* (1995), vol. 983 of *Lecture Notes in Computer Science*, Springer, pp. 261 – 278.

[26] MOHNEN, M. Efficient Compile-Time Garbage Collection for Arbitrary Data Structure. In *Symposium on Programming Language Implementation and Logic Programming (PLILP'95)* (1995), vol. 982 of *Lecture Notes in Computer Science*, Springer, pp. 241–258.

[27] PARK, Y. G., AND GOLDBERG, B. Escape Analysis on Lists. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (17-19 July 1992), vol. 27, pp. 116 – 127.

[28] RUGGIERI, C., AND MURTAGH, T. P. Lifetime Analysis of Dynamically Allocated Objects. In *Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1988), pp. 285 – 293.

[29] SERRANO, M., AND FEELEY, M. Storage Use Analysis and its Applications. In *1996 ACM SIGPLAN International Conference on Functional Programming* (May 1996), pp. 50–61.

[30] SHIVERS, O. Control flow analysis in Scheme. In *ACM SIGPLAN Conference on Programming Language, Design and Implementation* (jun 1988), pp. 164 – 174.

[31] TOFTE, M., AND TALPIN, J.-P. A theory of Stack Allocation in Polymorphically Typed Languages. Tech. Rep. 93/15, Departement of Computer Science, Copenhagen University, 9 July 1993.