Escape Analysis. Applications to ML and Java^{TM}

Bruno Blanchet
INRIA Rocquencourt
Bruno.Blanchet@inria.fr

December 2000

Overview

1. Introduction: escape analysis and applications.

2. Escape analysis
2.a Design: Common point between ML and Java
2.b Design: Differences
2.c Program transformation
2.d Complexity

3. Benchmark results

4. Conclusions

Introduction: what is escape analysis ?

A value escapes if its lifetime exceeds its static scope.

- ML: static scope of x in "$\texttt{let}\ x = M\ \texttt{in}\ N$" is N.

  Determine whether x may be returned or stored in a reference by N

- Java: static scope = the method.

  Determine whether an object may be returned, stored in a parameter or a static field by a method.

What is escape analysis ? (continued)

- Escape analysis is a particular may-alias analysis.

  It determines whether a value may be aliased with the result of an expression, with parameters and result of a method, with static fields.

- It is an abstract interpretation-based interprocedural analysis.

Applications. Summary of our benchmark results

- Stack allocation: value v does not escape
  $\Rightarrow$ v can be allocated on the stack.

  16 to 99% of data stack allocated in OCaml
  13 to 95% of data stack allocated in Java

- Synchronization elimination in Java: object o does not escape
  $\Rightarrow$ o is local to the current thread
  $\Rightarrow$ no need to synchronize calls on o.

  more than 20% of synchronizations eliminated on
  most programs, 94 and 99% on two examples

Runtime decrease up to 21% for ML (geometric mean 10%),
up to 43% for Java (geometric mean 22%).

Example: stack allocation in ML

```
let rec map f l = case l of
   [] => []
 | (a ::  l) => (f a) ::  (map f l)


let l =
   let a = [1;2;3] in
   let g x = x+1 in
      map g a
```

The same example with stack allocation

$$\texttt{map} : (\alpha \longrightarrow \underline{\beta}) \longrightarrow \underline{\alpha}\ \texttt{list} \longrightarrow \underline{\beta\ \texttt{list}}$$

```
let l =
   letstack a = [1;2;3] in
   letstack g x = x+1 in
      map g a
```

Example: stack allocation in Java

```
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);
  }
}
```

The same example with stack allocation

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

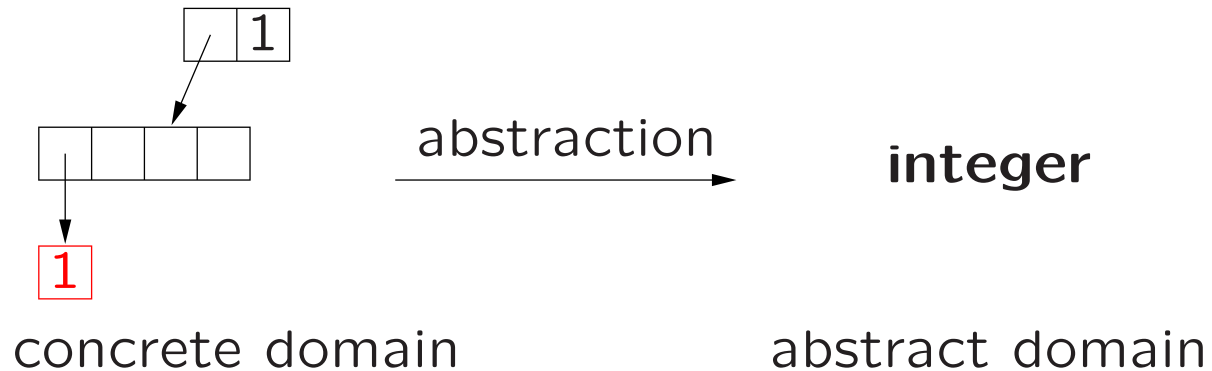
Example: synchronization elimination in Java

```
{
  java.util.Random r = new java.util.Random(RunTests.seed);
  for(int i = 0; i < length; i++)
  {
     array[i] = r.nextInt(); // synchronization
  }
}
```

`r` does not escape, so is local to the current thread, so the call to `r.nextInt()` does not need to be synchronized.

Design: common point

exact information	approximate information
What part of each value escapes ?	escape context

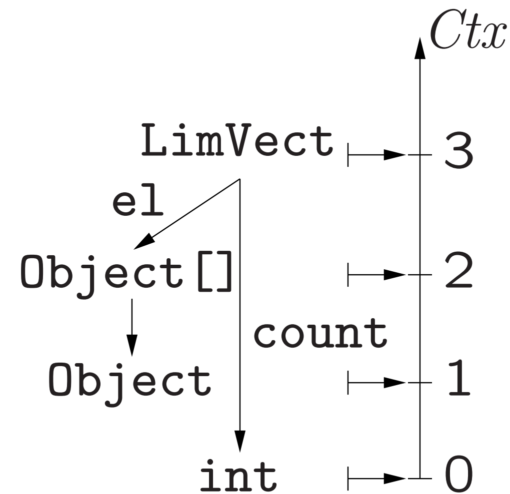


concrete domain	abstract domain

Definition of type heights

Height of a type $\tau = \top[\tau] \in \mathbb{N}$.
Main property : if τ contains τ' (as a field), $\top[\tau] \geq \top[\tau']$
If that does not contradict the preceding property,
and τ contains τ', $\top[\tau] \geq \top[\tau'] + 1$

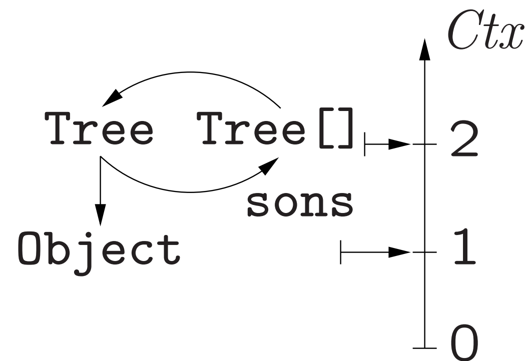
```
class LimVect {
  int count;
  Object[] el;
}
```



Definition of type heights: recursive types

Height of a type $\tau = \top[\tau] \in \mathbb{N}$.
Main property : if τ contains τ' (as a field), $\top[\tau] \geq \top[\tau']$
If that does not contradict the preceding property,
and τ contains τ', $\top[\tau] \geq \top[\tau'] + 1$

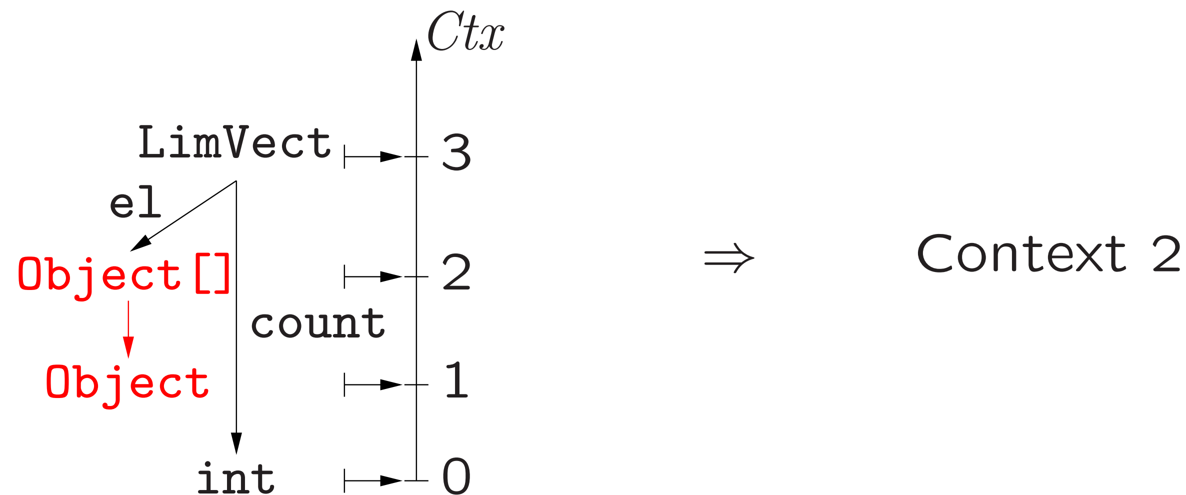
```
class Tree {
    Object element;
    Tree[] sons;
}
```



Definition of contexts

The escaping part of an object is represented by the escape context of the object.
The escape context is the height $\top[\tau] \in \mathbb{N}$ of the type τ of the escaping part.

$$Ctx = \mathbb{N}$$



The set of contexts for a value of type τ is $[0, \top[\tau]]$.

Why integers ?

- Integers provide a fast analysis.

- They also provide enough precision in practice.

  In particular, precise information for the top of the data structure.

Historically, Park and Goldberg PLDI'92, extended and improved by Deutsch POPL'97 and Blanchet POPL'98, OOPSLA'99.

Annotated types much more costly than integers,
for a small improvement on precision.

Design differences ML/Java: polymorphism

- Type variables in ML.

  The identity function `fun` $x \rightarrow x$ has type $\alpha \rightarrow \alpha$. If it is given a parameter of type τ, its result is of type τ.

  $\lambda c \in \{0, 1\}.c$ instantiated into $\lambda c \in [0, \top[\tau]].c$.

- Subtyping in Java.

  If the identity method `Object id(Object x){return x;}` is given a parameter of type τ, its result remains of type `Object`.

  $\lambda c \in \{0, 1\}.c$ instantiated into $\lambda c \in \{0, 1\}.\text{if } c \geq 1 \text{ then } \top[\tau] \text{ else } 0$.

  $\Rightarrow$ less precise than in ML (but much easier to design).

Design differences ML/Java: closures/virtual methods

- Higher-order functions in ML.
  - If f is a function, parameter of a higher-order function, we do not know which function f will be at runtime.
    We assume the worst: all parameters of f escape.
  - Types of values contained in a closure (the free variables) unknown $\Rightarrow$ difficulty when defining the height of a closure.
    A solution is ∞.
- Virtual methods in Java. Assume all the code is known.
  For each virtual call, determine a set of methods that may be called (class hierarchy analysis), and take the upper bound of the escape information of all these methods.

Design differences ML/Java: assignments

- Rare in ML.

  Approximation: the right-hand side of an assignment escapes.

- Very frequent in Java.

  A more precise analysis is necessary.

Analysis of assignments in Java: relations between escaping parts

```
void put(Object o) { this.el[count++] = o; }
static void run0() {
  LimVect local = new LimVect(4);
  local.put(new Integer(1));
  if (local.get(0) instanceof Integer) System.out.println("Integer");
}
```

The `new Integer(1)` does not escape $\Rightarrow$ do not say that when an object is stored, it always escapes.

Contexts: `local` $= 0$,
`o` $= 0$

Analysis of assignments in Java: relations between escaping parts

```
void put(Object o) { this.el[count++] = o; }
static LimVect run1() {
  LimVect local = new LimVect(4);
  local.put(new Integer(1));
  return local;
}
```

The `new Integer(1)` escapes, because it is stored in `local` and `local` escapes ⇒ The escaping part of the parameter `o` of `put` depends on the escaping part of the `this` parameter (`local`).

Contexts: $\texttt{local} = \top[\texttt{LimVect}] = 3$,
$\texttt{o} = \top[\texttt{Object}] = 1$

Context transformers

Abstract values: functions from the contexts of the result and of the parameters to the context of the considered object.

$$Val = (Ctx^n \rightarrow Ctx) \times Type^n \times Type \quad (n \in \mathbb{N})$$

Escape analysis: $L : Var \rightarrow Val$

For `LimVect.put`,

$$L(\texttt{this}) = (\{(c_0, c_1) \mapsto c_1\}, (\texttt{LimVect}, \texttt{Object}), \texttt{LimVect})$$
$$L(\texttt{o}) = (\{(c_0, c_1) \mapsto 1 \sqcap c_0\}, (\texttt{LimVect}, \texttt{Object}), \texttt{Object})$$

Program transformation features

Stack allocation:

- Reuse allocated space in loops in Java/tail recursion in ML.
- Adaptative inlining to increase stack allocation opportunities.

Synchronization elimination in Java:

- Global synchronization analysis
- Dynamic “in stack ?” test. When we allocate an object on the stack, it is thread local, so we do not synchronize on it.

Complexity

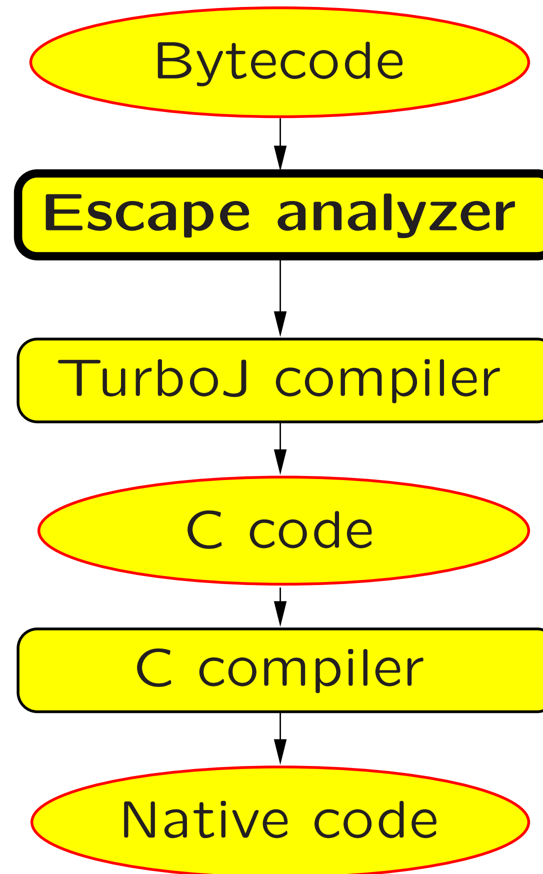
ML: $\boxed{\mathcal{O}(n \log^2 n)}$

Java:

Size of the SSA form	n
Number of parameters of a method	p
Number of parameters of a context	p'
Maximum type height	H
Number of iterations	$n_i = \mathcal{O}(np'H)$

$$\boxed{\mathcal{O}(npp'n_i)}$$

Structure of the Java compiler

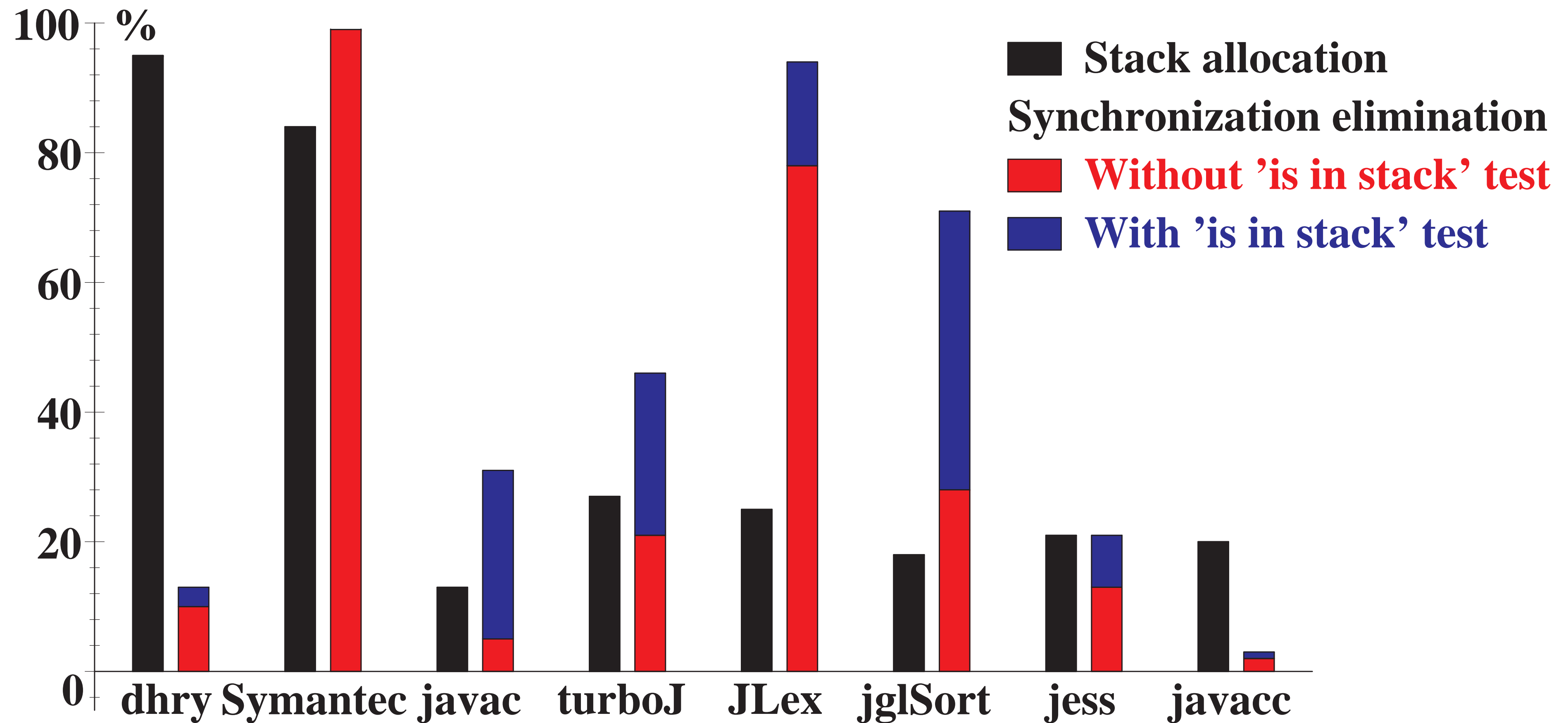


Our benchmarks

Pentium MMX 233 MHz. 128 Mb RAM, Jdk 1.1.5.

Benchmark programs		Size (kb)
dhry	Dhrystone	73
Symantec	A set of small benchmarks	76
javac	Java compiler (jdk 1.1.5) compiling jBYTE	242
turboJ	Java to C compiler from Silicomp RI	311
JLex	Lexer generator (v. 1.2) running `sample.lex`	97
jglSort	Sorting benchmarks of JGL 3.1.0	95
jess	Expert system (v. 4.1), solving `fullmab.clp`	363
javacc	Java parser generator (v. 0.8pre2)	254

Stack allocation and synchronization elimination: a precise analysis

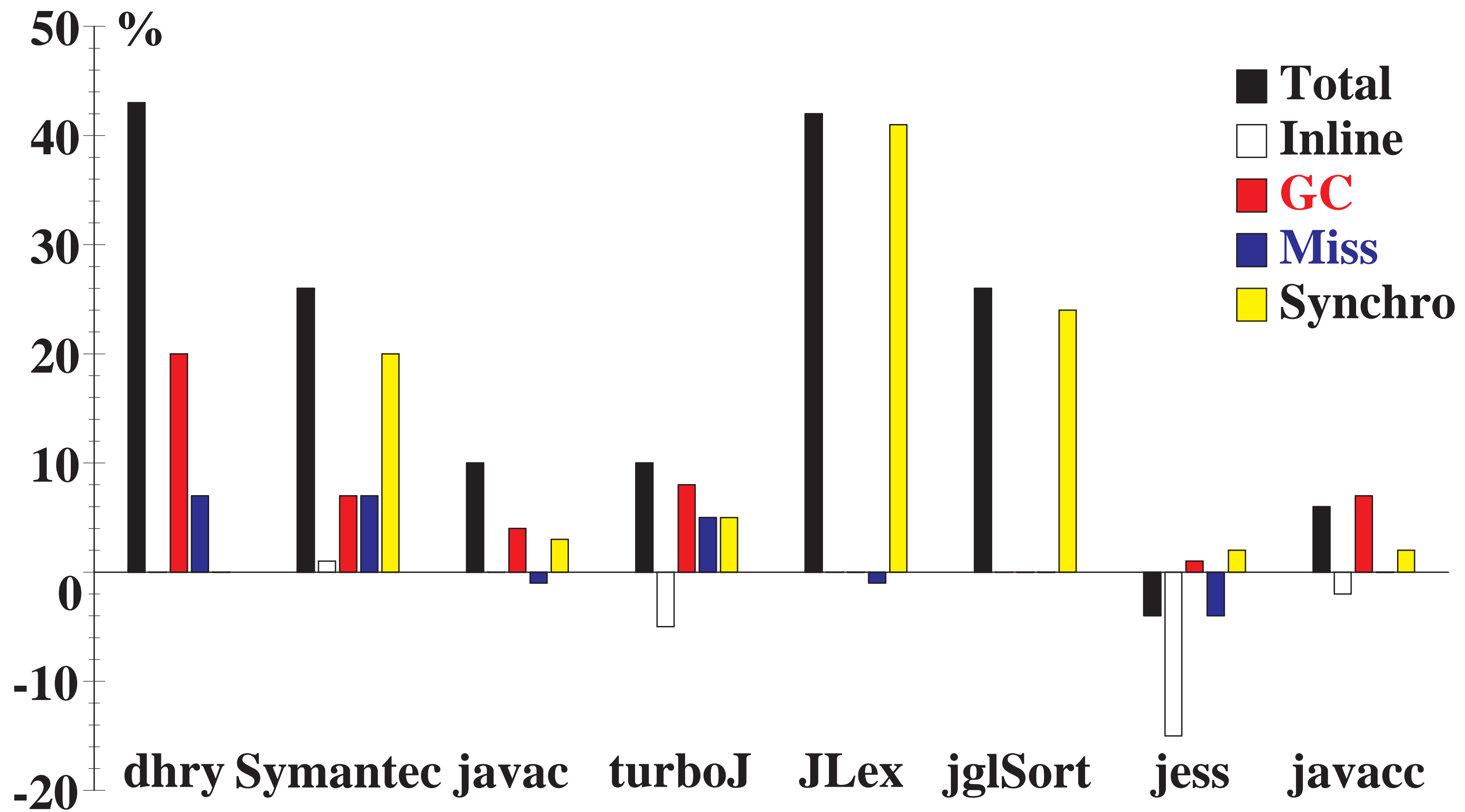


The origin of speedups

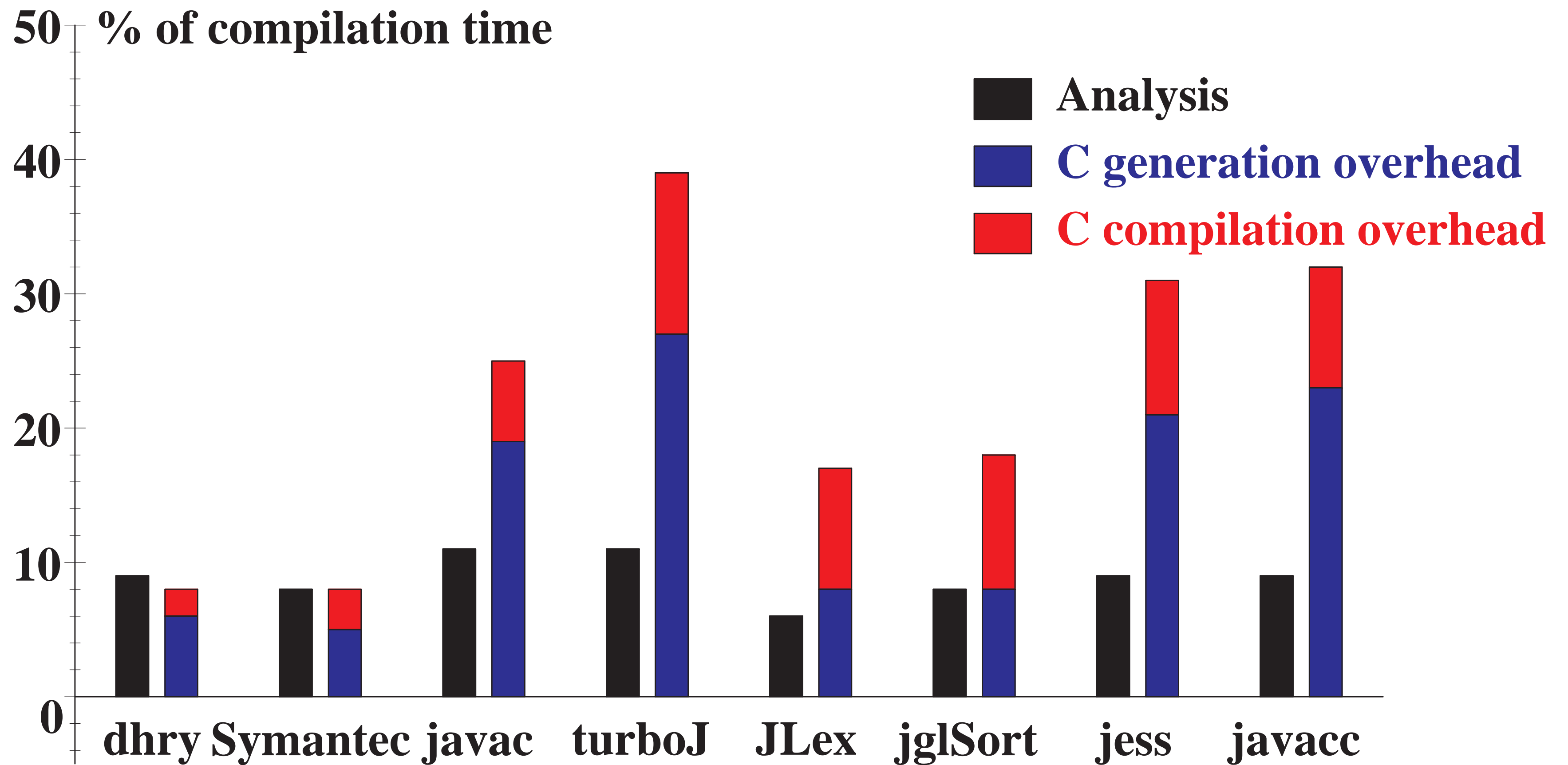
Speedups coming from stack allocation have 3 causes.

- Decrease of the GC workload (main cause with a mark and sweep GC);
- Stack allocation is faster than heap allocation (with a mark and sweep GC);
- Better data locality (main cause with a copy GC).

Runtime decrease



Analysis time: A fast analysis



Conclusion (1)

- Very reasonable cost.
  - Representing the escaping part by integers yields the best tradeoff;
  - Analyze precisely the most important features of each language;
  - Fast analyses are interesting: even if they miss a few optimization opportunities, they provide most of the expected improvement. They scale much better than more complex analyses.

Conclusion (2)

- Speedups.
  - Important speedups with a mark and sweep GC (decrease of the GC workload and allocation time);
  - Smaller speedups with a copy GC (improved data locality);
  - Important speedups coming from synchronization elimination in Java.

Future research

- Verification of cryptographic protocols
- Verification of the Java platform and of Java programs (confinement, mutability, ...)
- Verification of critical software