# Abstract interpretation

## Application to stack allocation and synchronization elimination in Java$^{TM}$

Bruno Blanchet
INRIA Rocquencourt
`Bruno.Blanchet@inria.fr`

February 28th, 2001

## Projet MOSCOVA
## MObilité, Sécurité, COncurrence, Vérification et Analyse

- Join-calculus: a new model of distributed programming
- Security
- Validation and debugging of concurrent software (Caml garbage collection, Ariane 5)

# Part I

# Abstract interpretation

# Abstract interpretation

- Static analysis: determine runtime properties of programs without executing them.

- However, exact static analysis undecidable for most properties.

- $\Rightarrow$ perform approximations, but safe approximations.

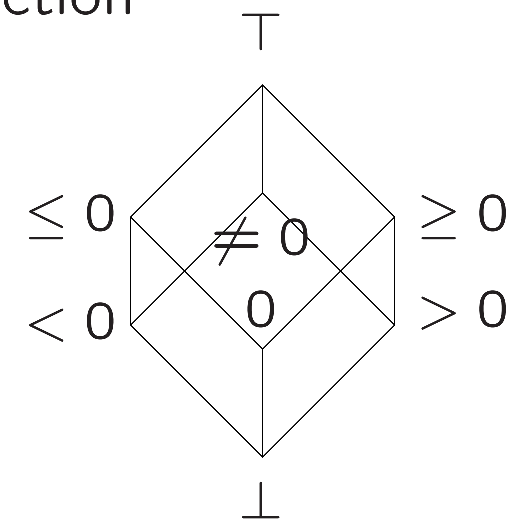
## An example: signs - first idea

Given a program, determine the sign of each variable at each program point.

| **Integer** $\longrightarrow$ | **Sign** |
|---|---|
| 2 | $> 0$ |
| -3 | $< 0$ |
| 0 | 0 |

# An example: signs - abstraction

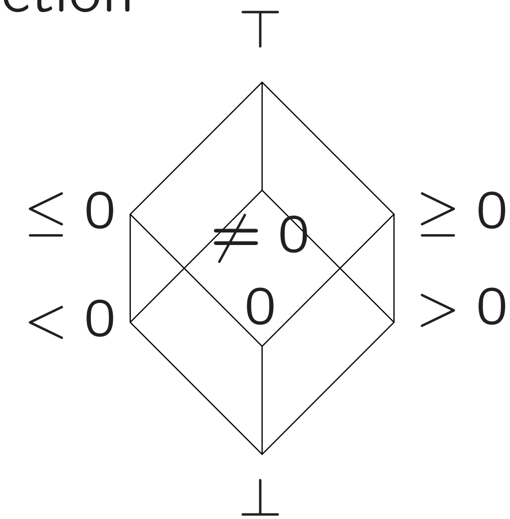
$$\textbf{Sets of integers} \underset{\text{abstraction}}{\overset{\text{concretization}}{\rightleftarrows}} \textbf{Sign}$$



## An example: signs - abstraction

$$\textbf{Sets of integers} \underset{\text{abstraction}}{\overset{\text{concretization}}{\rightleftarrows}} \textbf{Sign}$$

$\subseteq$
Lattice



## An example: signs - analysis of a program

```
x = -5;
1:
while (x < 0) {
   2:
   x++;
   3:
}
4:
```

| 1: | 2: | 3: | 4: |
|---|---|---|---|
| $< 0$ | $\perp$ | $\perp$ | $\perp$ |

## An example: signs - analysis of a program

```
x = -5;
1:
while (x < 0) {
   2:
   x++;
   3:
}
4:
```

| 1: | 2: | 3: | 4: |
|---|---|---|---|
| $< 0$ | $\perp$ | $\perp$ | $\perp$ |
| $< 0$ | $< 0$ | $\leq 0$ | $\perp$ |

## An example: signs - analysis of a program

```
x = -5;
1:
while (x < 0) {
    2:
    x++;
    3:
}
4:
```

| 1: | 2: | 3: | 4: |
|---|---|---|---|
| $< 0$ | $\perp$ | $\perp$ | $\perp$ |
| $< 0$ | $< 0$ | $\leq 0$ | $\perp$ |
| $< 0$ | $< 0$ | $\leq 0$ | $0$ |

## Historical background

- Invented by Patrick and Radhia Cousot.
- The first papers: POPL'77 and POPL'79.
- Two important papers: Journal of Logic and Computation, 1992 and PLILP'92.

## Achievements and future perspectives

- Verification of Ariane 5 software after the crash.
- Project of verification of plane softwares (Airbus A380).
- Projects of verification of Java programs, cryptographic protocols, ... Semantic watermarking...

# Part II

# Escape analysis

# Introduction: what is escape analysis ?

- Consider an object $o$ allocated in a method $m$.
  Does $o$ escape from $m$ ?
  $\Leftrightarrow$ Is $o$ still reachable after the return from $m$ ?

- Abstract interpretation interprocedural analysis.

## Applications

- Stack allocation: object $o$ does not escape from $m$
  $\Rightarrow$ $o$ can be allocated on the stack in $m$.
  13 to 95% of data stack allocated

- Synchronization elimination: 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

Speedup up to a 1.75 factor (geometric mean 1.27).

## Example

```
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);
  }
}
```

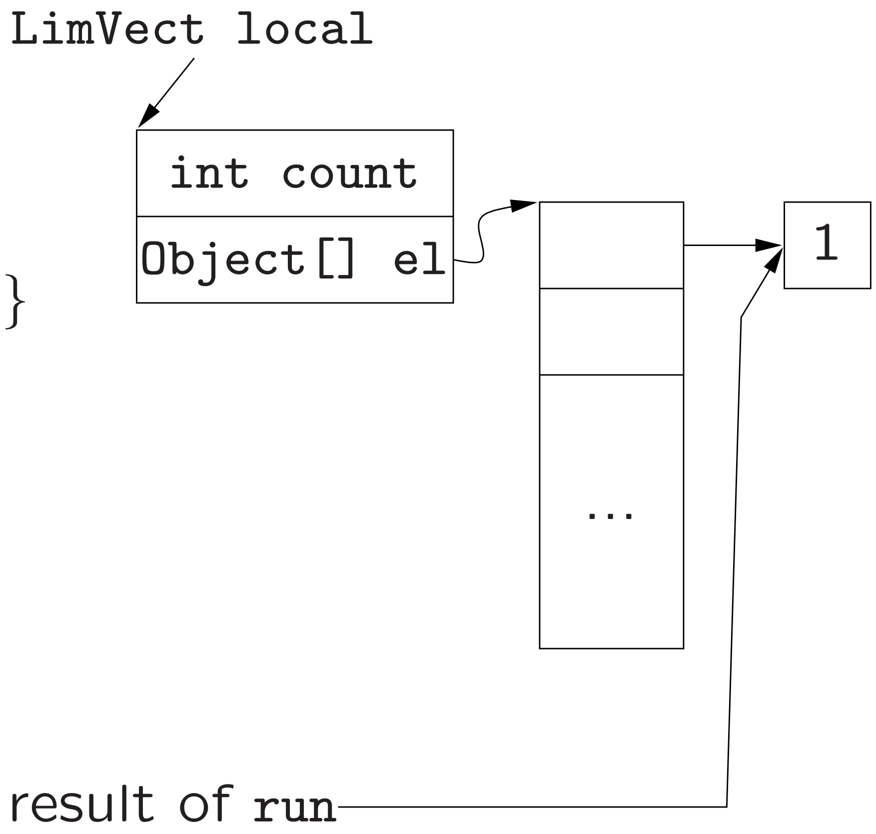
# Example

```
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);
  }
}
```

## Heap

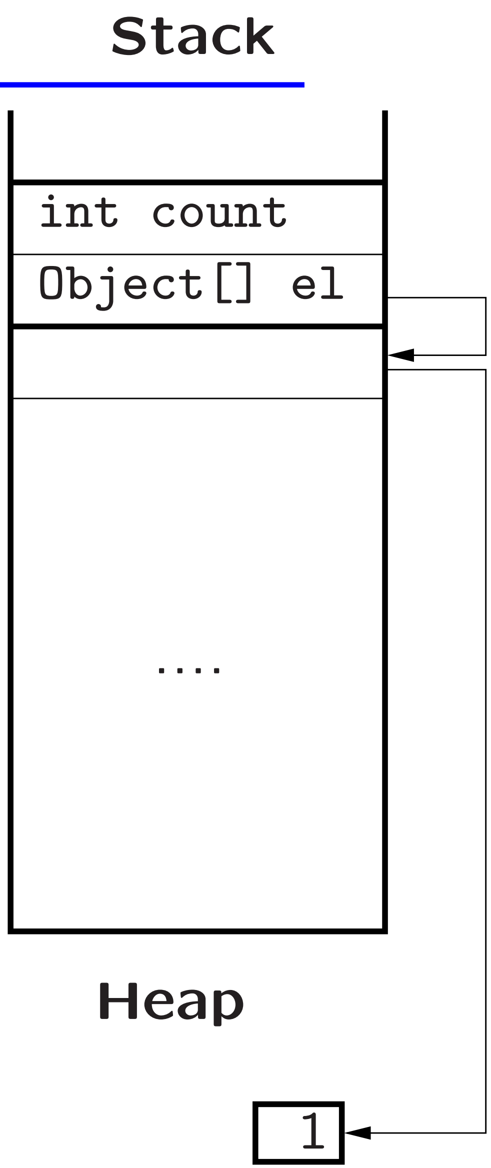


# The same example with stack allocation

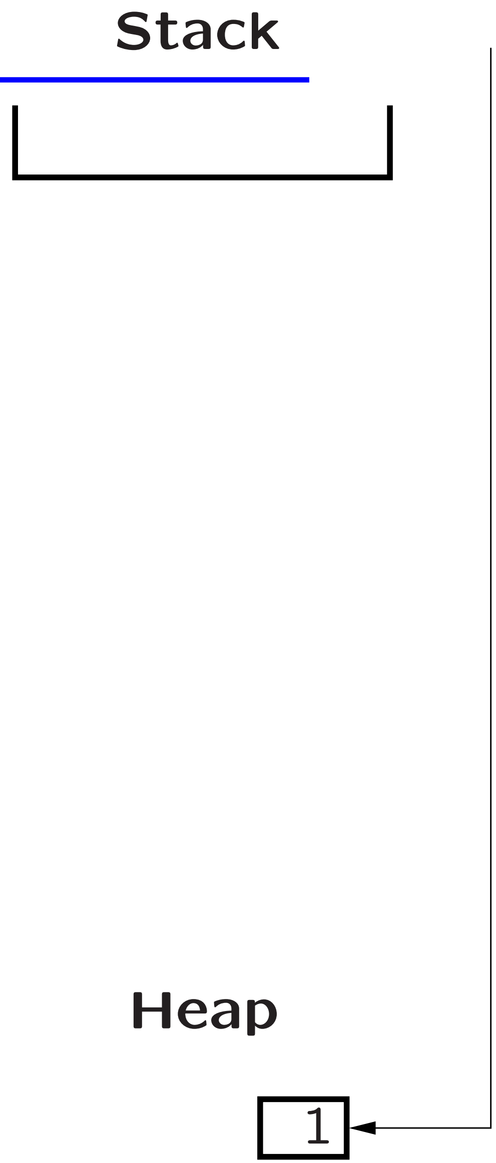
```
static Object run() {
  LimVect local = alloca_new LimVect();
  local.el = alloca_new Object[n];
  local.put(new Integer(1));
  return local.get(0);
}
```

## The same example with stack allocation

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

## Synchronization elimination example

```
{
  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.

# Abstraction

exact information

What part of each value escapes ?

approximate information

escape context


concrete domain

abstract domain

## Definition of type heights

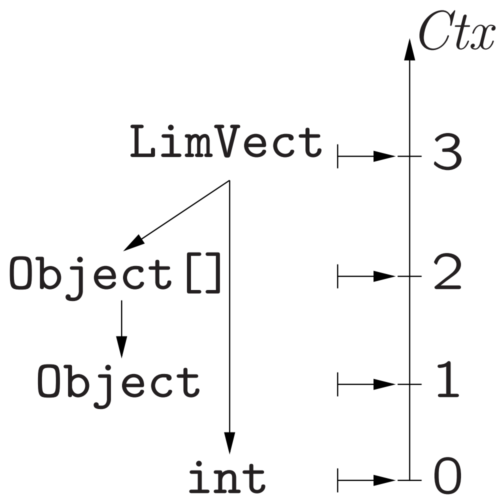
Height of a type $\tau = \top[\tau] \in \mathbb{N}$.

Main property : if $\tau$ contains $\tau'$ (as a field), $\top[\tau] \geq \top[\tau']$

If that does not contradict the preceding property,

and $\tau$ contains $\tau'$, $\top[\tau] \geq \top[\tau'] + 1$

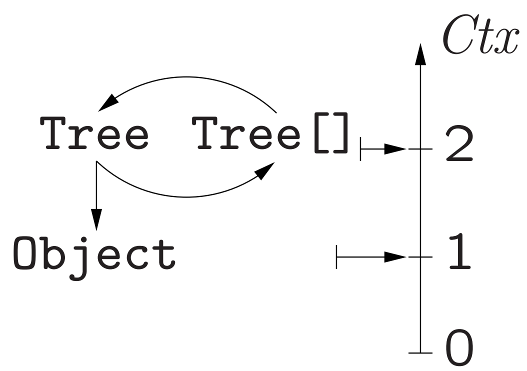
```
class LimVect {
  int count;
  Object[] el;
}
```



## Definition of type heights: recursive types

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

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

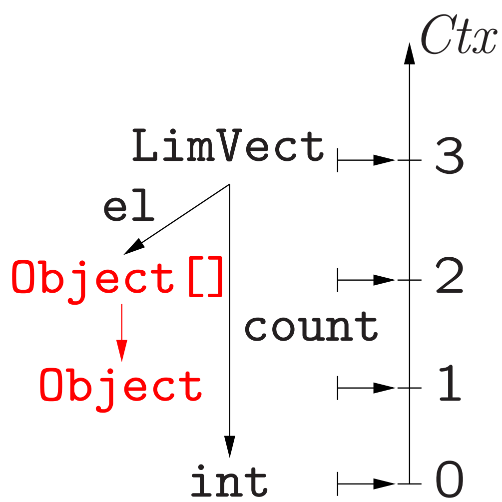


## Definition of contexts

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 $\tau$ of the escaping part.

$$\boxed{Ctx = \mathbb{N}}$$




$\Rightarrow$ Context 2

## Why integers ?

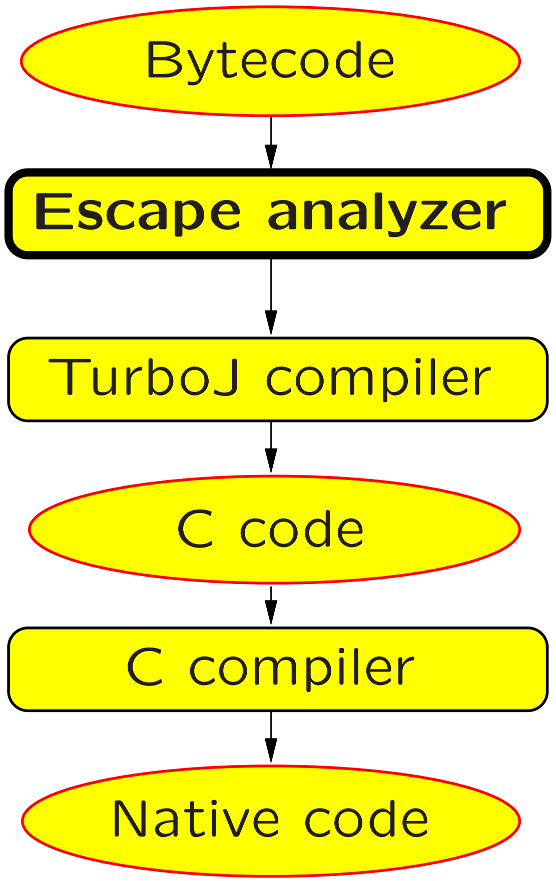
- Integers provide a fast analysis.

- 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.

## Structure of the compiler

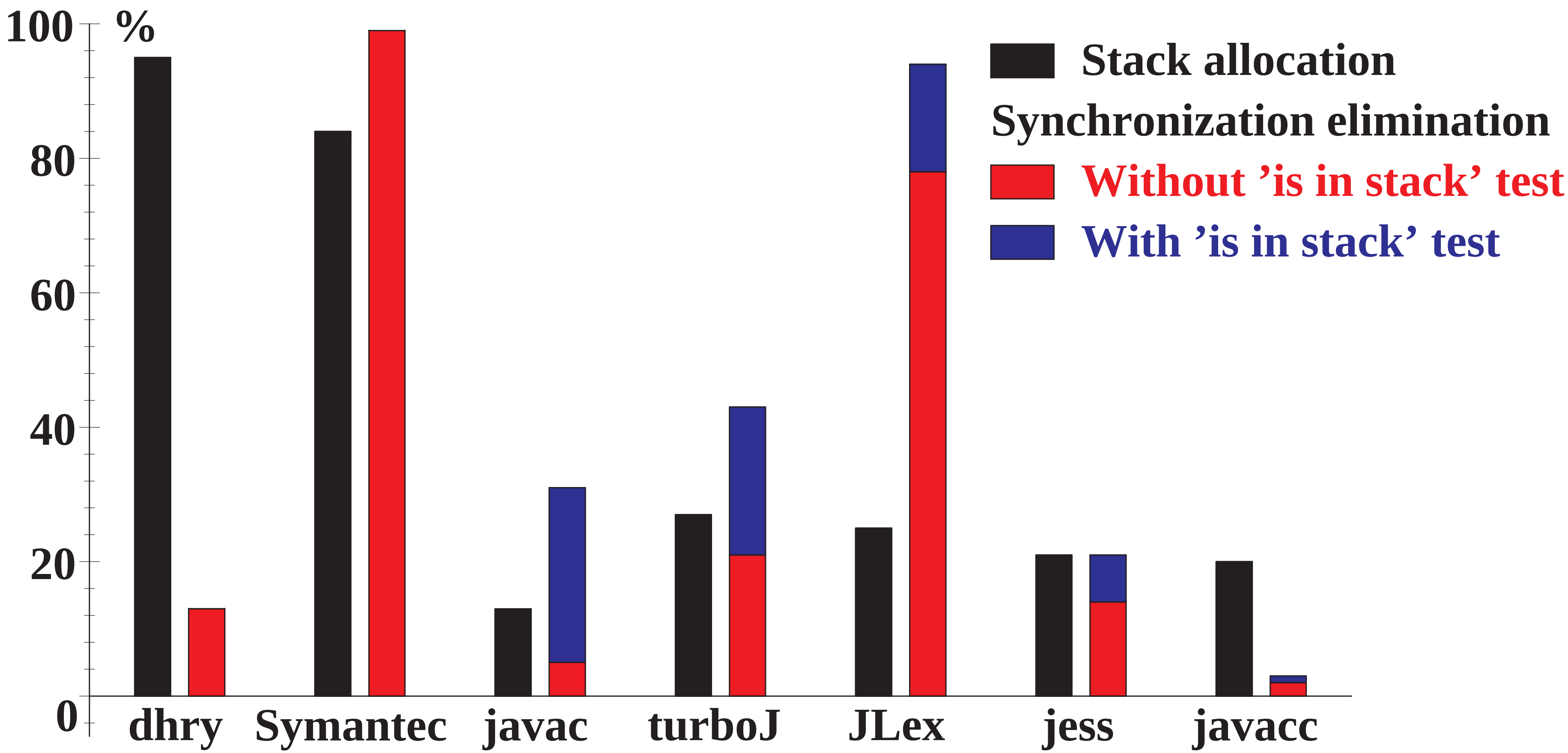


## Our benchmarks

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

| Benchmark programs | | Size (kb) |
|---|---|---|
| dhry | Dhrystone | 6 |
| Symantec | A set of small benchmarks | 19 |
| javac | Java compiler (jdk 1.1.5) compiling jBYTE | 600 |
| turboJ | Java to C compiler from Silicomp RI | 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) | 497 |

# Stack allocation and synchronization elimination: a precise analysis

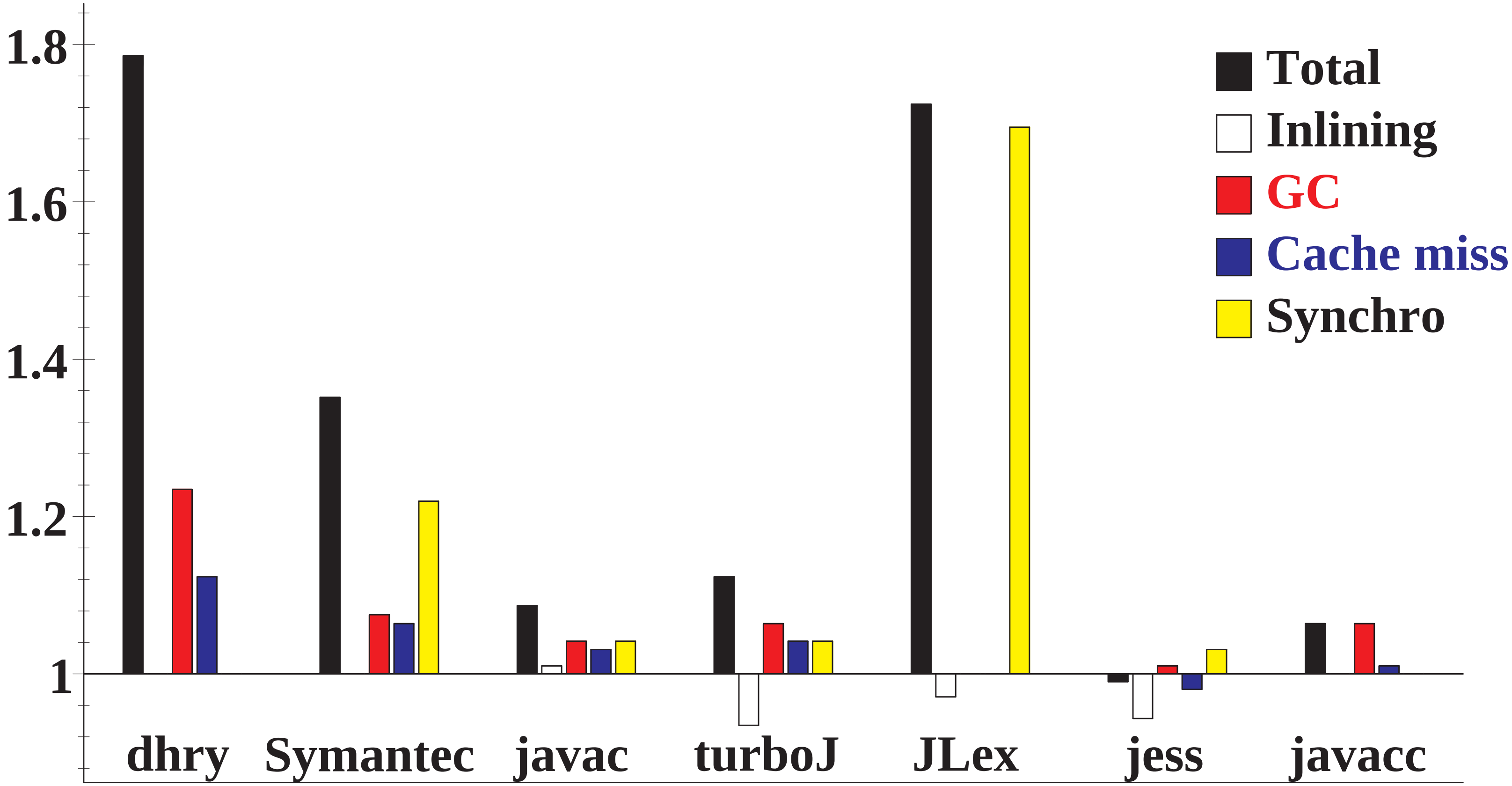


# Understanding the origin of speedups

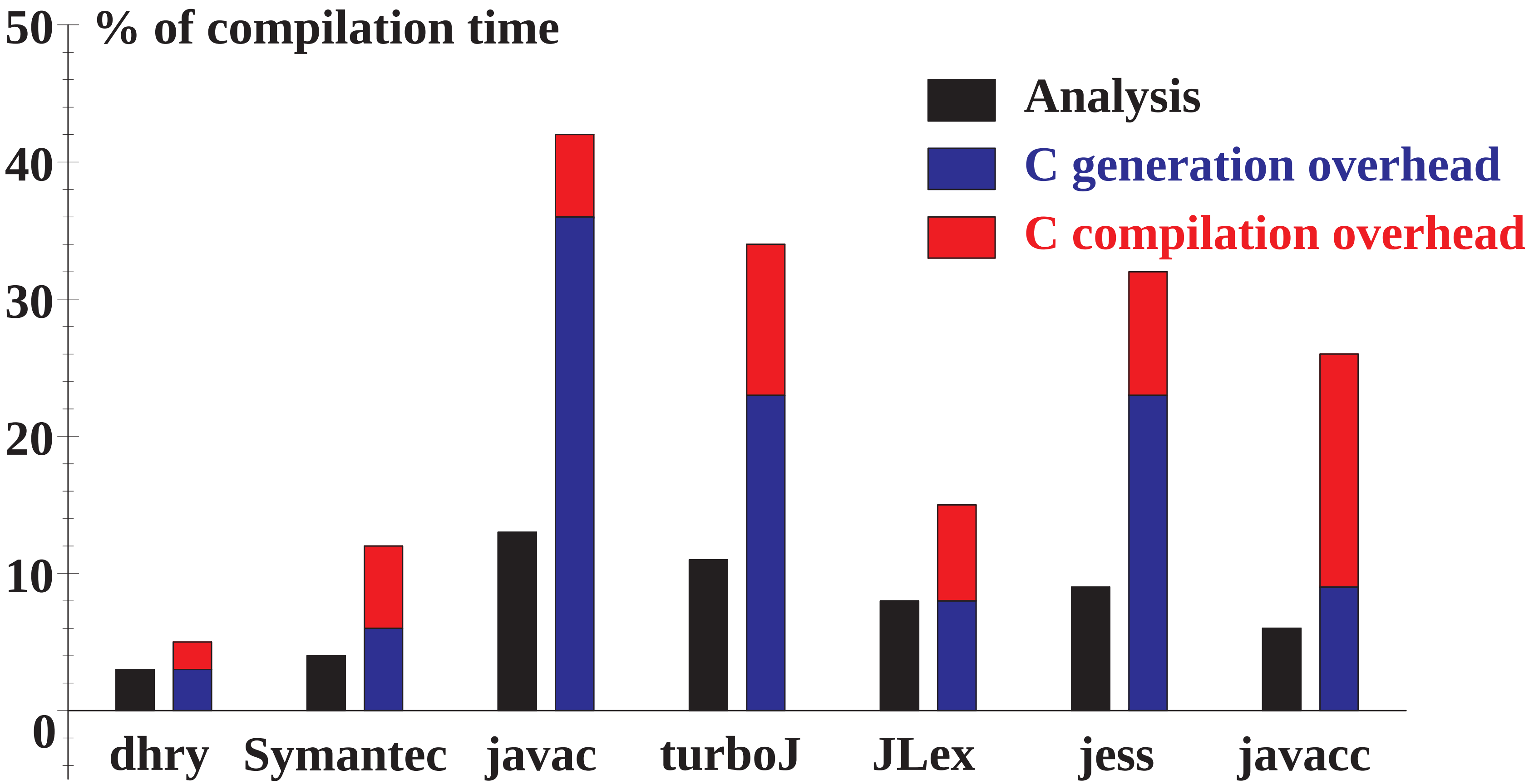
Speedup coming from stack allocation have 3 causes.

- Main cause: decrease of the GC workload;
- Stack allocation is faster than heap allocation with a mark and sweep GC;
- Better data locality.

## Speedup


1.8
1.6
1.4
1.2
1
Total
Inlining
GC
Cache miss
Synchro
dhry
Symantec
javac
turboJ
JLex
jess
javacc

## Analysis time: A fast analysis



## Conclusion

- Very reasonable cost.
- High speedups: factor 1.27 on average (geometric mean).
  - Speedup coming from a decrease of the GC workload, and to a less important extent, from a better data locality.
  - Stack allocation gives larger speedups with a mark and sweep GC.
  - Synchronization elimination gives important speedups.