



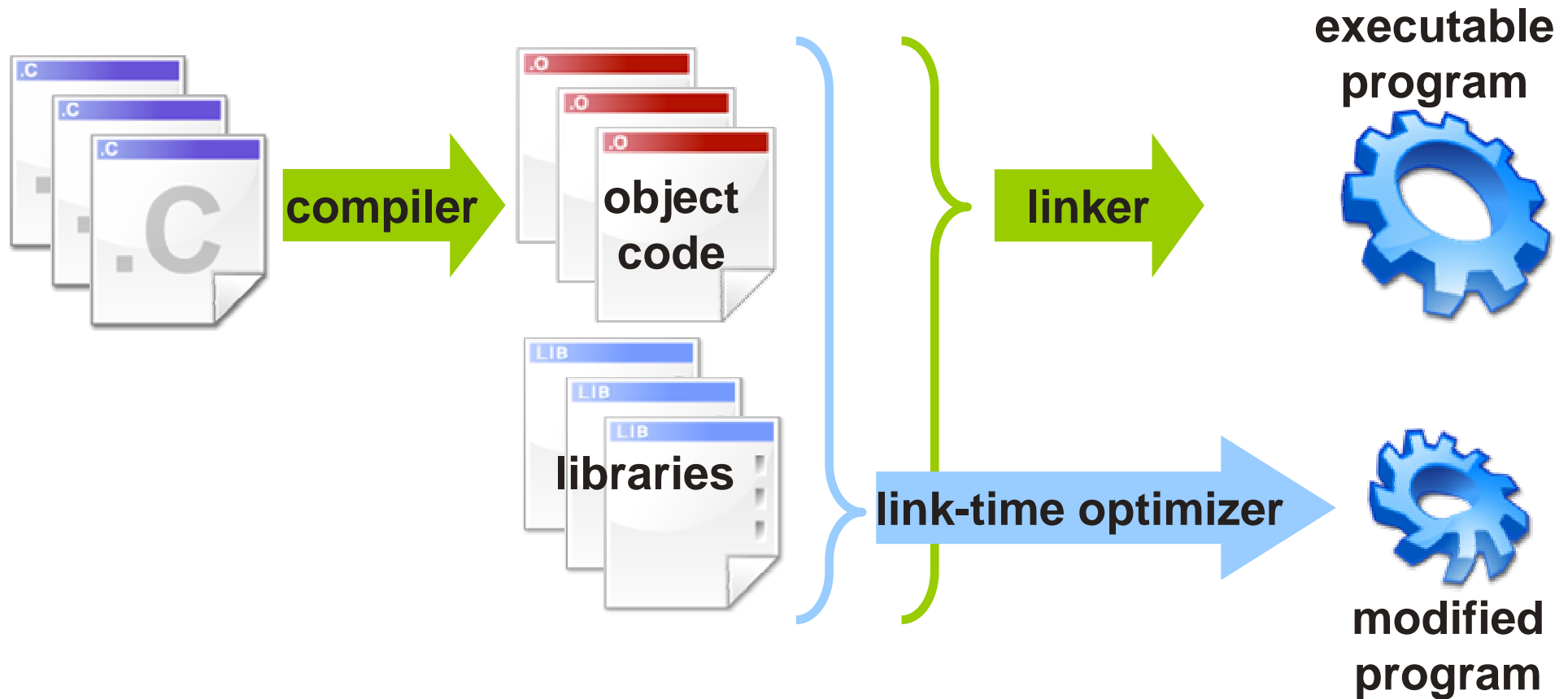
# Whole-Program Linear-Constant Analysis with Applications to Link-Time Optimization

Ludo Van Put – Dominique Chanut –  
Koen De Bosschere

Ghent University

<http://diablo.elis.ugent.be>

# Link-time optimization



# Link-time optimization

Interesting for embedded systems, e.g. ARM [De Bus et al., 2004]:

- Reduce code size (~15%)
- Improve performance (~10%)
- Reduce energy consumption (~8%)

Can we go even further?

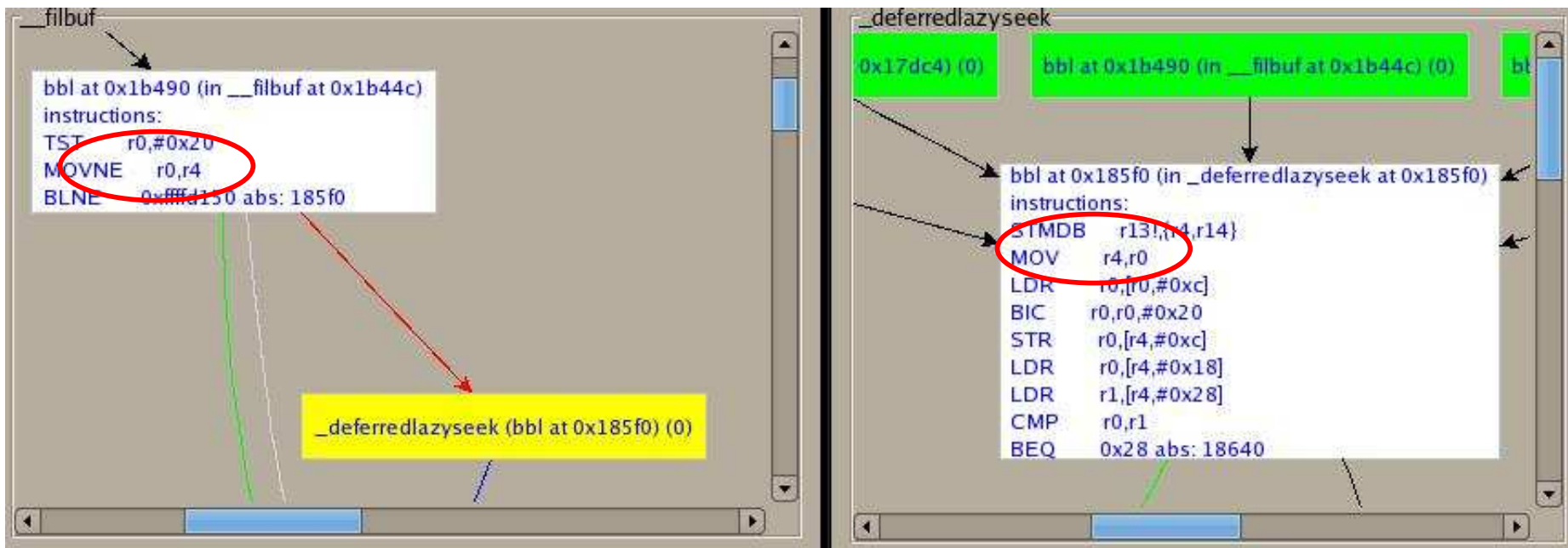


# Outline

- motivation
- theory
  - linear-constant equations & analysis
- practice
  - data structure & operations
- experience
  - ARM optimizations
- conclusion

# We could do better but...

- memory & stack: black box
- low level code: no compile time information
- need more (complex) analyses to further exploit whole-program overview



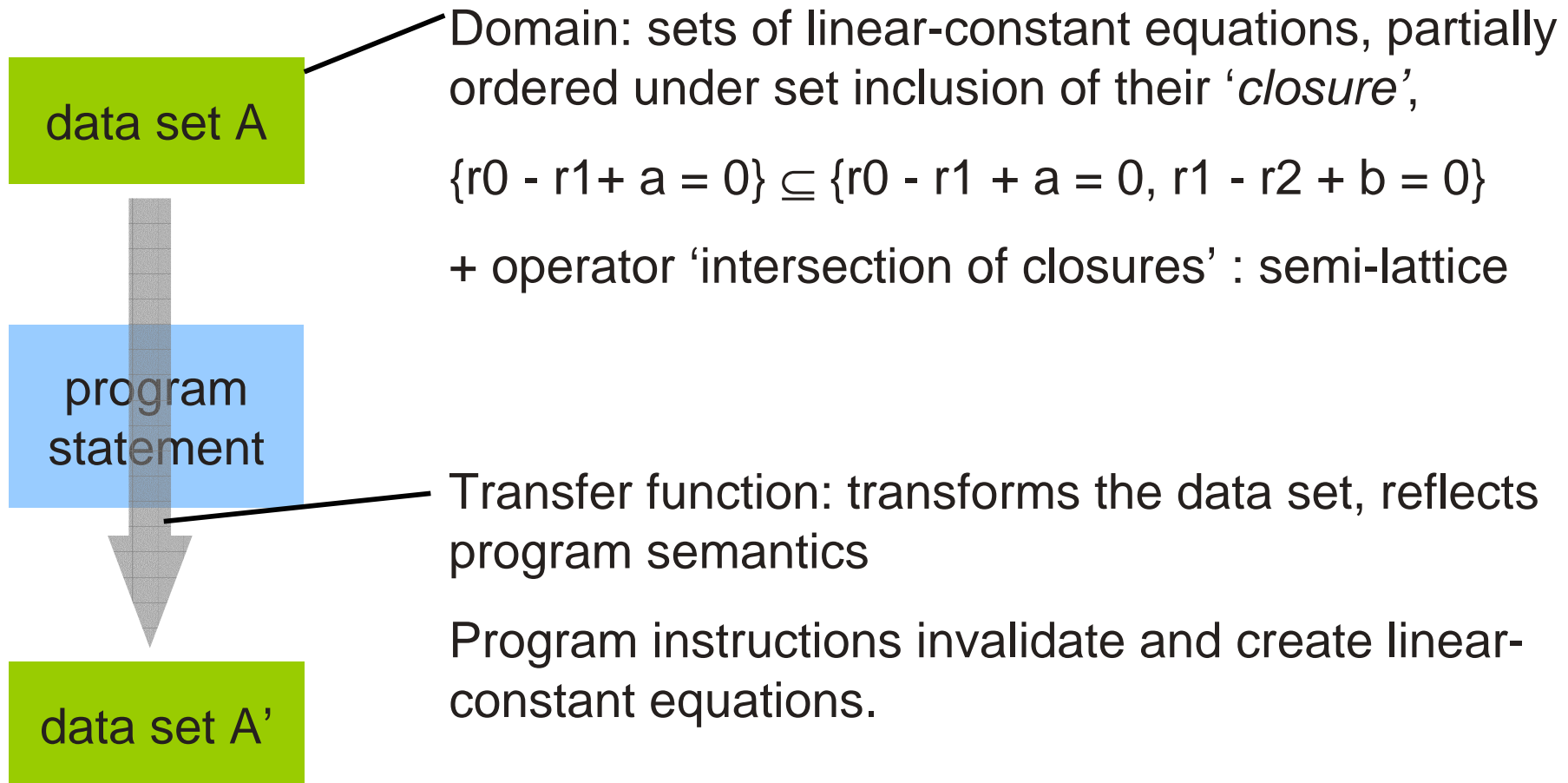
# Propagate linear expressions

- machine instructions introduce simple relations between registers: e.g.

$$\text{ADD } r0, r4, \#4 \rightarrow r0 - r4 + 4 = 0$$

- constant information we can propagate through the graph and use it
- not a new idea [Karr, 1976], but a different context: huge graph, simple instructions, fixed number of registers, explicit memory accesses & addresses
  - less general relations and less complex computations

# Dataflow analysis



# Linear-constant equations

- Here:  $y = x + c$ ,  $y$  and  $x$  registers,  $c$  integer constant
- Combining linear-constant equations:  
$$x_1 - y_1 + c_1 = 0 + x_2 - y_2 + c_2 = 0$$
  
then  $x_1 = y_2$  or  $y_1 = x_2$
- *Closure* of set: all combinations
- Restriction: underdetermined or exactly determined sets
  - by construction in straight-line code
  - by intersection of closure at merge points

# Data set representation

- assign registers unique number:  $r_0 \dots r_{n-1}$
- add virtual **zero-valued** register,  $r_\infty$  :
  - **MOV**  $r_0, \#8 \rightarrow r_0 - r_\infty - 8 = 0$
  - constant propagation
- normalize set of equations ( $r_x - r_y + c = 0$ )

$$r_0 - r_\infty - 8 = 0$$

$$r_1 - r_\infty - 4 = 0$$

$$r_3 - r_2 - 12 = 0$$



$r_0$	-	$r_1$	-	4	=	0	→	$x < y$
$r_1$	-	$r_\infty$	-	4	=	0		
$r_2$	-	$r_3$	+	12	=	0		

$\neq$        $\neq$

# What about addresses?

- Graph representation: addresses are meaningless, symbolic references instead:  $ADDR\ r2, \$reference$
- Many addressproducers at link-time: optimization opportunity
- Extend linear-constant equations:  
 $r_x - r_y + c - addr_x + addr_y = 0$
- Combination requirement:  $addr_{x1} = addr_{y2}$  or  $addr_{y1} = addr_{x2}$

$$r_0 - r_\infty - 8 = 0$$

$$r_1 - r_\infty - 4 - ref_1 = 0$$

$$r_3 - r_2 - 12 = 0$$

$$r_0 - r_1 - 4 + ref_1 = 0$$

$$r_1 - r_\infty - 4 - ref_1 = 0$$

$$r_2 - r_3 + 12 = 0$$



# Compact, fast data structure

- at most n equations: fixed-size array

$$\begin{array}{rcl} r_0 & r_1 - 4 & \text{ref}_1 \\ r_1 & r_\infty - 4 & \text{ref}_1 \\ r_2 & r_3 + 12 & \end{array}$$



Redundant! Reuse it for speedup

$$\begin{array}{rcl} 0: & r_0 & r_1 - 4 & \text{ref}_1 \\ 1: & r_0 & r_\infty - 4 & \text{ref}_1 \\ 2: & r_2 & r_3 + 12 & \\ 3: & r_2 & & \end{array}$$
Red arrows indicate the reuse of variables. An arrow points from  $r_1$  in the first equation to  $r_0$  in the second equation. Another arrow points from  $r_2$  in the third equation to  $r_2$  in the fourth equation.

# Operations

- lookup: index,  $O(1)$
- remove register: index, combination,  $O(1)$
- insert equation
  - $r_x = r_y$ : change constant  $c$ ,  $O(1)$
  - $r_x \neq r_y$ : combine until normalized,  $O(n)$
- meet operation: compute closures, intersect closures and normalize resulting set, each  $O(n^2)$

# Example applications

- copy elimination, remove redundant code
- stack analysis & dead spill code elimination

# Remove redundant code

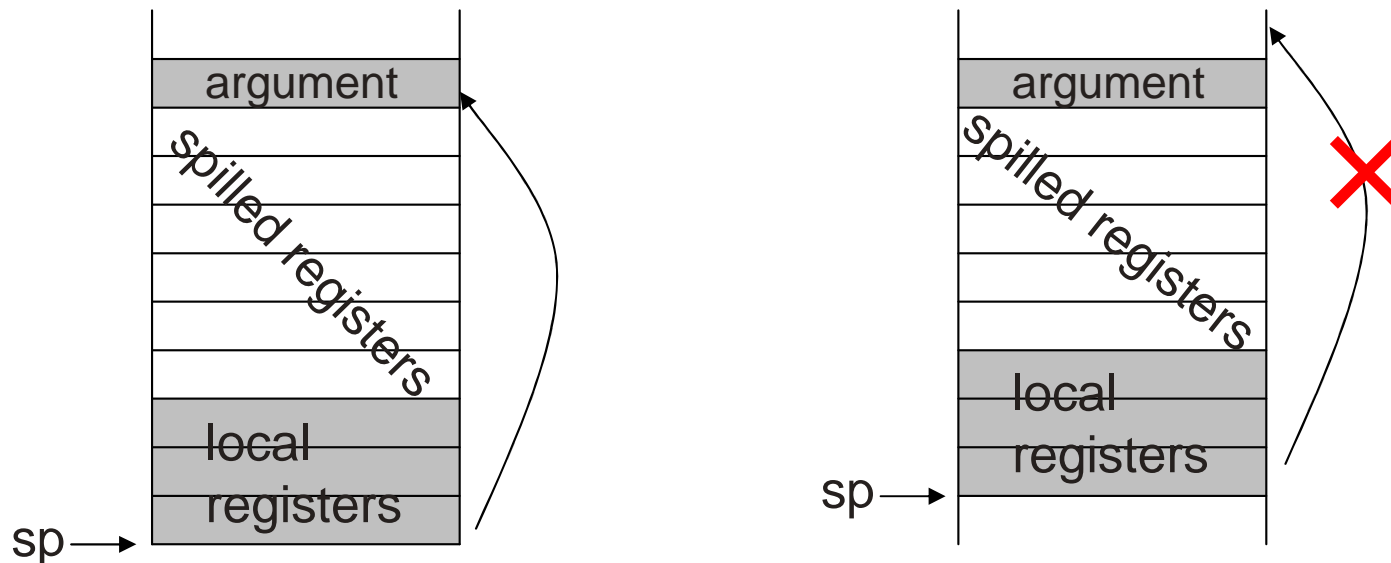
The screenshot shows a debugger window titled 'build\_tree' with a menu bar (File, View, Optimize, Info, Help) and a toolbar. A 'Nr callers' field shows '15'. The assembly code is displayed in three blocks, connected by green arrows indicating control flow. Red arrows point to specific instructions.

```
build_tree
  bbl at 0xddb8 (in build_tree at 0xda90)
  instructions:
  MOV    r14,r3
  MOV    r9,r10,LSL #3
  MOV    r8,r3
  bbl at 0xddc4 (in build_tree at 0xda90)
  instructions:
  CMP    r12,r14
  ADD    r2,r11,r12,LSL #2
  MOV    r14,r8
  BGE   0x38 abs: de10
  bbl at 0xddd4 (in build_tree at 0xda90)
  instructions:
  LDR    r5,[r2,#0x4]
  LDR    r6,[r11,r12,LSL #2]
```

Annotations: A green arrow points to the first block. Red arrows point to the first and third instructions of the first block, and the third instruction of the second block. A green arrow points from the first block to the second, and another from the second to the third. Black curved arrows point from the third block back to the second and first blocks, indicating a loop.

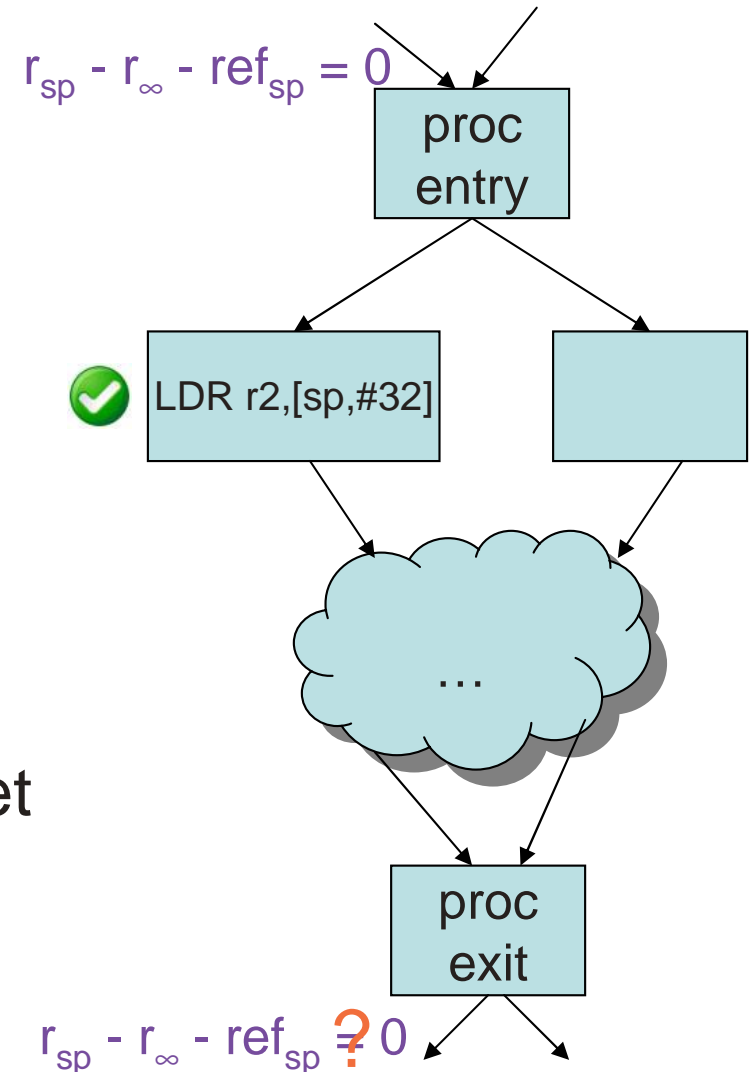
# Dead spill code

- ARM spill code: multiple-load & multiple-stores, e.g. `STMDB sp!,{r4,r5,r6,r7,ra}` saves 5 registers on the stack
- What if (one of) these registers is dead? Can we remove it from the list?
- Arguments are read using explicit offsets



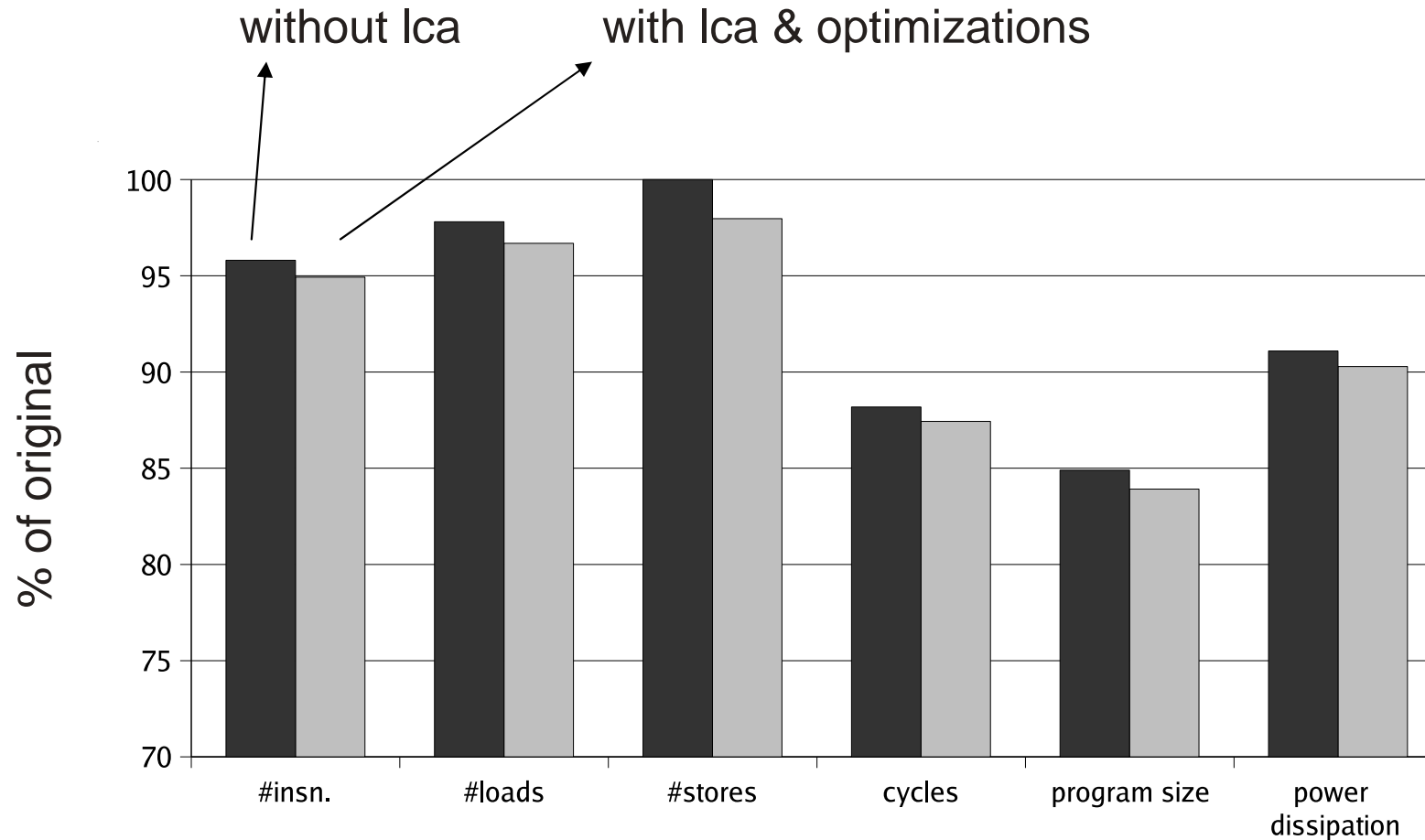
# Stack analysis

- at start of a procedure:  
 $\{r_{sp} - r_{\infty} - ref_{sp} = 0\}$
- propagate through procedure, assuming:
  - function calls restore  $r_{sp}$
  - spilled registers & local variables cannot be accessed by callee
- mark instructions that use  $r_{sp} + \text{offset}$
- remove spill code & adjust offsets where needed



# Compaction & power savings

ARM ADS 1.1 toolchain '-Os', benchmarks from De Bus et al., 2004



# Conclusion

- fast and practical link-time analysis
- enabler for new analyses and optimizations
- further reduction in code size & energy consumption
- program understanding, stack analysis

<http://diablo.elis.ugent.be>

