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

Ludo Van Put – Dominique Chanet – Koen De Bosschere

Ghent University

http://diablo.elis.ugent.be
Link-time optimization

- Compiler
- Object code
- Libraries
- Link-time optimizer
- Linker
- Executable program
- Modified program
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
  \[ \rightarrow \] less general relations and less complex computations
Dataflow analysis

Domain: sets of linear-constant equations, partially ordered under set inclusion of their ‘closure’,
\[ \{r_0 - r_1 + a = 0\} \subseteq \{r_0 - r_1 + a = 0, r_1 - r_2 + b = 0\} \]
+ operator ‘intersection of closures’ : semi-lattice

Transfer function: transforms the data set, reflects program semantics
Program instructions invalidate and create linear-constant equations.
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: \( r_0 \ldots r_{n-1} \)
- add virtual zero-valued register, \( r_\infty \):
  - MOV r0, #8 → \( r_0 - r_\infty - 8 = 0 \)
  - constant propagation
- normalize set of equations \((r_x - r_y + c = 0)\)

\[
\begin{align*}
  r_0 - r_\infty - 8 & = 0 \\
  r_1 - r_\infty - 4 & = 0 \\
  r_3 - r_2 - 12 & = 0
\end{align*}
\]

\[
\begin{align*}
  r_0 - r_1 - 4 & = 0 \\
  r_1 - r_\infty - 4 & = 0 \\
  r_2 - r_3 + 12 & = 0
\end{align*}
\]
What about addresses?

• Graph representation: addresses are meaningless, symbolic references instead: `ADDR r2, $reference`
• Many address producers at link-time: optimization opportunity
• Extend linear-constant equations:
  \[ r_x - r_y + c - \text{addr}_x + \text{addr}_y = 0 \]
• Combination requirement: \( \text{addr}_{x1} = \text{addr}_{y2} \) or \( \text{addr}_{y1} = \text{addr}_{x2} \)

\[
\begin{align*}
    r_0 - r_\infty - 8 &= 0 \\
    r_1 - r_\infty - 4 - \text{ref}_1 &= 0 \\
    r_3 - r_2 - 12 &= 0
\end{align*}
\]

\[
\begin{align*}
    r_0 - r_1 - 4 + \text{ref}_1 &= 0 \\
    r_1 - r_\infty - 4 - \text{ref}_1 &= 0 \\
    r_2 - r_3 + 12 &= 0
\end{align*}
\]
Compact, fast data structure

- at most n equations: fixed-size array

\[ \begin{align*}
  r_0 & \quad r_1 - 4 \quad \text{ref}_1 \\
  r_1 & \quad r_\infty - 4 \quad \text{ref}_1 \\
  r_2 & \quad r_3 + 12 \\
\end{align*} \]

Redundant! Reuse it for speedup

0: \( r_0 \quad r_1 - 4 \quad \text{ref}_1 \)
1: \( r_0 \quad r_\infty - 4 \quad \text{ref}_1 \)
2: \( r_2 \quad r_3 + 12 \)
3: \( r_2 \)
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
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:
  \[ r_{sp} - r_{∞} - ref_{sp} = 0 \]
• 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

without lca with lca & optimizations

% of original

% of original

#insn. #loads #stores cycles program size power dissipation
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