Comp 512
Spring 2011

Global Optimization

Live Variables, Global Block Placement

Last Lecture

- Computing dominance information
  - → Quick introduction to global data-flow analysis
    - That is *compile-time reasoning about the runtime flow of values*
    - Round-robin iterative algorithm to find MOP solution to DOM
- Using immediate dominators to improve on SVN
  - → For a node *n*, start LVN with the hash table from IDOM(*n*)
    - Includes results from each predecessor in dominator tree
  - → Use scoped hash table and SSA names to simplify algorithm
  - → Some predecessor information at each node in CFG

This Lecture

Examples of Global Analysis and Transformation

- Computing live variables
  - → Classic backwards global data-flow problem
  - → Used in SSA construction, in register allocation
- Using live information to eliminate useless stores
  - → Simple demonstration of the use of LIVE
- Single-procedure block placement algorithm (Pettis & Hansen)
  - → Arrange the blocks to maximize fall-through branches
  - → Improves code locality as a natural consequence

Computing Live Information

A value v is *live* at p if $\exists$ a path from p to some use of v along which v is not re-defined

Data-flow problems are expressed as simultaneous equations

Annotate each block with sets *LIVEOUT* and *LIVEIN*

$$LIVEOUT(b) = \cup_{s \in succ(b)} LIVEIN(s)$$

$$LIVEIN(b) = UEVAR(b) \cup (LIVEOUT(b) \cap \overline{VARKILL(b)})$$

$$LIVEOUT(n_f) = \varnothing$$

Domain of LIVEOUT is variables

§ 8.6.1 in EaC2e

where

UEVAR(b) is the set of names used in block b before being defined in b

VARKILL(b) is the set of names defined in b

Note that LIVE is a backwards data-flow problem

Computing Live Information

The compiler can solve these equations with a simple algorithm

WorkList ← { all blocks }
while (WorkList ≠ Ø)
  remove a block b from WorkList
  Compute LIVEOUT(b)
  Compute LIVEIN(b)
  if LIVEIN(b) changed
    then add *pred* (b) to WorkList

The Worklist Iterative Algorithm

Why does this work?

- LIVEOUT, LIVEIN $\subseteq 2^{\text{Names}}$
- UEVAR & VARKILL are constants for b
- Equations are monotone
- Finite # of additions to sets

⇒ will reach a fixed point !

Speed of convergence depends on the order in which blocks are “removed” & their sets recomputed

Follows from last lecture’s algorithm for DOM

The worklist should be implemented as a set so that it does not contain duplicate entries.

Using Live Information: Eliminating Unneeded Stores

Transformation: Eliminating unneeded stores

- Value in a register, have seen last definition, never again used
- The store is <u>dead</u> (*except for debugging*)
- Compiler can eliminate the store

The Plan:

- Solve for LIVEIN and LIVEOUT
- Walk through each block, bottom to top
  - → Compute local LIVE incrementally
  - → If target of STORE operation is not in LIVE, delete the STORE
- If all STOREs to a local variable are eliminated, can delete the space for it from the activation record

|LIVEOUT| = |*variables*|

Using LIVE Information: Eliminating Unneeded Stores

Safety

- If $x \notin \text{LIVE}(s)$ at some STORE s, its value is not used along any path from s to the exit node of the CFG
  - → Its value is not read and is, therefore, *dead*
- Relies on the correctness of LIVE

Profitability

- Assumes that not executing a STORE costs less than executing it

Opportunity

- Linear search, block-by-block, for STORE operations
  - → Could build a list of them while computing initial UEVAR set

Block Placement

The order of blocks in memory matters

- Bad placement can increase working set size (*TLB & page misses*)
- Fall-through and branch-taken paths differ in cost & locality

The plan

- Discover which paths execute frequently
- Rearrange blocks to keep those paths in contiguous memory

Finding hot paths

- Need execution profile information

Block Placement

Targets branches with unequal execution frequencies

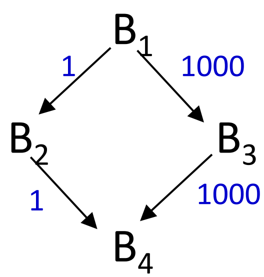
- Make likely case the “fall through” case
- Move unlikely case out-of-line & out-of-sight

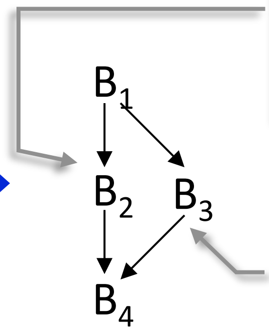
Potential benefits

- Longer branch-free code sequences
- More executed operations per cache line
- Denser instruction stream ⇒ fewer cache misses
- Moving unlikely code ⇒ denser page use & fewer page faults

Block Placement

Moving infrequently executed code

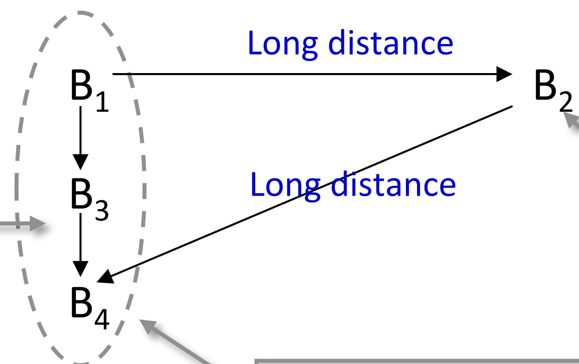




Unlikely path gets fall through (cheap) case

Likely path gets an extra branch

Would like this to become



This branch goes away

In another page, ...

Denser instruction stream

*

Block Placement

Overview

1. Build chains of frequently executed paths
   → Work from profile data
   → Edge profiles are better than node profiles
   → Combine blocks with a simple greedy algorithm
2. Lay out the code so that chains follow short forward branches

Gathering profile data

- Instrument the executable
- Statistical sampling
- Infer edge counts from performance count data

While precision is desirable, a good approximation will probably work well.

Block Placement

The Idea

- Form chains that should be placed to form straight-line code

First step: Build hot paths

EaC2e, Figure 8.16

E ⇽ |edges|

for each block b
*   make a degenerate chain, d, for b*
*   priority(d) ⇽ E*

P ⇽ 0

for each CFG edge <x,y>, x ≠ y, in decreasing frequency order
*   if x is the tail of chain a and y is the head of chain b then*
*   t ⇽ priority(a)*
*   append b onto a*
*   priority(a) ⇽ min(t,priority(b),P++)*

Point is to place targets after their sources, to make forward branches

PA-RISC predicted most forward branches as taken, backward as not taken

Block Placement

Second step: Lay out the code

t ⇽ chain headed by the CFG entry node, n_0

WorkList ⇽ { (t,priority(t)) }

while (Worklist ≠ Ø)

remove a chain c of lowest priority from WorkList

for each block x in c, in chain order

place x at the end of the executable code

for each block x in c

for each edge <x,y> where y is unplaced

t ⇽ chain containing <x,y>

if (t,priority(t)) ∉ WorkList

then add (t,priority(t)) to WorkList

Intuitions

- Entry node first
- Tries to make edge from chain *i* to chain *j* a forward branch

→ Predicted as taken on target machine
→ Edge remains only if it is lower probability choice

Going Further – Procedure Splitting

Any code that has profile count of zero (0) is "fluff"

- Move fluff into the distance
  - → It rarely executes
  - → Get more useful operations into I cache
  - → Increase effective density of I cache
- Slower execution for rarely executed code

Branch to fluff becomes short branch to long branch.

Block with long branch gets sorted to end of current procedure.

Implementation

- Create a linkage-less procedure with an invented name
- Give it a priority that the linker will sort to the code's end
- Replace original branch with a 0-profile branch to a 0-profile call
  - → Cause linkage code to move to end of procedure to maintain density

*

Block Placement

Safety

- Changing position of code, not values it computes
- Barring bugs in implementation, should be safe

Profitability

- More fall-through branches
- Where possible, more compiler-predicted branches
- Better code locality

Opportunity

- Profile data shows high-frequency edges
- Looks at all blocks and edges in transformation – **O**(*N+E*)

Many transformations have an **O**(*N+E*) component

Transformations We Have Seen

Scope	Name	Analysis	Effect
Local	LVN	Incremental	Redundancy, constants, & identities
	Balancing	LIVE info.	Enhance ILP
Regional	Superlocal VN	CFG, EBBs	Redundancy, constants, & identities
	Dominator VN	CFG, DOM info.	
Global	Dead store elim.	LIVE info.	Eliminate dead store
	Block placement	CFG, Profiles	Code locality & branch straightening
Interprocedural	*Inline subs'n* *Proc. placement*		*On Monday*