Profile-Guided Code Positioning

See also §8.6.2 & 8.7.2 in EaC2e
Final Exam, Fall 2016

- Friday December 9, 2016 @ 9AM in Herring Hall Room 100
- Three hours, closed-book, closed-notes, no electronic devices

- Covers material since midterm exam  
  (see Lectures page on web site)
  - You are responsible for the contents of the lectures in the course

In **EaC2e**, this includes material from the following chapters & sections
  - Chapter 6
  - Chapter 7
  - Chapter 8, sections .3, 8.4.1, 8.6.2, 8.7.2
  - Chapter 9, section 9.2.1
  - Chapter 10, sections 10.2, 10.6.1
  - Chapter 11
  - Chapter 12
Code Layout

The placement of program text in memory matters

- Large working sets cause excessive TLB & page misses
- Bad placement can increase instruction cache misses
  - Pettis & Hansen report that on some PA-RISC benchmarks, 1 of 3 cycles was a cache miss (single-level cache — after all, it was 1989 ..)

Random placement of code leaves these effects to chance

Remember

- Code lives in the process’ virtual address space
- It must be moved into RAM (& mapped to a physical address)
- It must be moved into the instruction cache

All before it can execute

... and page faults & cache faults take time ...
Address Space Mapping

(Earlier Lecture)

The Big Picture

Compiler’s view

virtual address spaces

On a disk

OS’ view

In RAM

Expensive transition

1980 Hardware view

Physical address space

TLB is a single word cache used by the OS to speed virtual-to-physical address translation. A processor may have > 1 level of TLB.
Of course, the “Hardware view” is no longer that simple.

Many processors now include L3 caches and L4 may be on its way.
Code Layout

The placement of program text in memory matters

- Large working sets cause excessive TLB & page misses
- Bad placement can increase instruction cache misses

Random placement of code leaves these effects to chance

Cache Misses and Page Faults on Branches (& Jumps) Are Expensive

- Can turn a 5 cycle branch into 80 cycles or 2,000 cycles
- Compiler should avoid this problem whenever it can

The plan: Compute code layouts that reduce cost due to branches & faults

- Discover execution-time paths, similar to traces
- Rearrange the code to keep those paths in contiguous memory
- Make heavy use of execution profiles
Pettis and Hansen tried this back in late 1980s

**Examples within HP**

**Pascal compiler**
- Moved frequently executed blocks to top of procedure  
  (*block locality*)
- 40% reduction in instruction cache misses
- 5% improvement in compiler’s running time

**Fortran compiler**
- Rearranged object files before linking  
  (*procedure locality*)
- Attempt to improve locality on calls
  - Put caller after, but near, the callee
- 20% system throughput improvement

**So, they believed ...**

And these experiments were done when RAM was a couple of cycles away, not 50 to 100 cycles away.
How Does Placement Improve Performance?

Most cache improvement work focuses on the data cache hierarchy

• Change iteration order of a loop
  – Increase stride-one access (consecutive locations)
  – Increase reuse of values

• Techniques make a huge difference in performance

Instruction cache improvement helps in two key ways

• Increase both use & reuse
  – Use every word that is fetched into the i-cache
    → Unused words waste time, effort, & energy
  – Keep code for loops in the i-cache

• I-Cache activity can interfere with D-Cache activity in shared levels

These effects were exaggerated by early 90’s PC cache designs

  – Small number of relatively long lines

Historically, many processors had “loop buffers” — small i-caches that held a handful of operations.
Two Major Issues

**Procedure placement**

- If A calls B, would like A & B in adjacent locations
  - On same page means smaller working set (pages)
  - Adjacent locations limit I-cache conflicts
- Unfortunately, multiple procedures may call B (& A)
  - A common & critical problem in interprocedural optimization
- This interprocedural issue should be solved in the linker
  - Linker has all of the code and all of the relationships

**Block placement**

- Same effects occur on a smaller scale
- Fall through branches create an additional incentive
- Rarely executed code fills up the cache, too!
- This intraprocedural issue can be solved in the compiler

Block placement precedes procedure placement
Procedure Placement

Simple principles
- Build the call graph
- Annotate edges with execution frequencies
- Use “closest is best” placement
  - A calls B most often ⇒ place A next to B
  - Keeps branches short
  - Direct mapped I-cache ⇒ A & B unlikely to overlap in I-cache

Profiling the call graph
- Linker inserts a stub for each call that bumps a counter
- Counters are kept in statically initialized storage (set to zero)
- Adds overhead to execution, but only in training runs

HP: Reduction in executed “long” branches by 80 to 98%

HP: Profiles ignored all indirect calls (call through a pointer) (virtual methods calls)
Procedure Placement

Computing an order

- Combine all edges from A to B
- Select highest weight edge, say X→Y
  - Combine X & Y, along with their common edges, X→Z & Y→Z
  - Place Y immediately after X
- Repeat until graph cannot be reduced further

May have disconnected subgraphs
- Indirect calls
- Must add new procedures at end
  - W→X and Y→Z with WZ & XY
  - Use weights in original graph
  - Largest weight closest

The actual algorithm is straightforward
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 code sequences without a “taken” branch
- More executed operations per cache line
- Denser instruction stream $\Rightarrow$ fewer cache misses
- Moving unlikely code $\Rightarrow$ 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

We would like this to become

This branch goes away

Denser instruction stream

In another page, in another galaxy …

COMP 412, Fall 2016
Block Placement

**Principles**

- Goal is to eliminate taken branches
  - Build up traces – acyclic single paths
- Work from profile data
  - Edge profiles are better than block profiles
- Use a greedy, bottom-up strategy to combine blocks

**Gathering profile data**

- Insert code to count edges
- Split critical edges
- Myriad details for gathering accurate whole-program edge counts
Block Placement

The Idea

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

First step: Build the chains

1. Make each block a degenerate chain & set its priority to # blocks
2. \( P \leftarrow 1 \)
3. \( \forall \) edge \( e = <x,y> \) in the CFG, in order by decreasing frequency
   if \( x \) is the tail of a chain, \( a \), and \( y \) is the head of a chain, \( b \)
   then merge \( a \) and \( b \)
   else set priority(\( y \)) to \( \min(\text{priority}(y), P++) \)

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**

\[ \text{WorkList} \leftarrow \text{chain containing the entry node, } n_0 \]

While (\text{WorkList} \neq \varnothing)

- Pick the chain \( c \) with lowest priority(\( c \)) from \text{WorkList}
- Place it next in the code
- For all edge \( <c,z> \) leaving \( c \)
  - \( z \) to \text{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

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

Branch to fluff becomes short branch to long branch.
Block with long branch gets sorted to end of current procedure.
Putting It Together

- Procedure placement is done in the linker

- Block placement is done in the optimizer
  - Allows branch elision due to fluff, other tailoring

- Speedups ranged from 2 to 10%, depending on cache size
  - On the PA-RISC; better results on x86 systems

- This technique paid off handsomely on early 1990s PCs
  - Long cache lines
  - Slow page faults
  - Microsoft insiders suggested it was most important optimization for codes like Office

(Word, PowerPoint, Excel)