COMP 512
Rice University
Spring 2015

Overview 4

Global & Interprocedural Optimization

Citation numbers refer to entries in the EaC2e bibliography.

Interprocedural Optimization

Global optimization finds and removes many inefficiencies.
It misses others.

- Some opportunities arise from procedure linkages
  - ♦ Parameter binding, branches, register save/restore code
  - ♦ Costs involved in indirect calls (*function parameters, methods in OOLs*)
- Some single-procedure opportunities are disrupted by calls
  - ♦ Stops propagation of constants (*parameters & globals*)
  - ♦ Register save & restore (*tracking values through memory is hard*)

Interprocedural optimization tries to eliminate _some_ of those inefficiencies.

- Interprocedural analysis
  - ♦ Analyze all the code that is available & use the results to improve the code
- Interprocedural transformations
  - ♦ Transformations that involve code in two or more procedures

Where Are We?

Last Lecture

- Safety of global optimization often formulated as a data-flow analysis problem
- Long discussion—*very long*—on theory behind iterative data-flow analysis

Today

- Another global optimization
  - ♦ Block placement
- Two interprocedural optimizations
  - ♦ Procedure placement (interprocedural analog of block placement)
  - ♦ Inline substitution

Profile-Guided Code Positioning: The Motivation

Examples within HP

- Early **PA-RISC** tests showed **CPI** of 3, with 1/3 of that due to I-cache misses
  - ♦ Subsequent hardware improvements reduced that, but authors claim that the impact of I-cache misses on CPI was still substantial
- Pascal compiler
  - ♦ Moved frequently executed blocks to top of procedure
  - ♦ 40% reduction in <u>instruction cache misses</u>
  - ♦ 5% improvement in compiler's running time
- Fortran compiler
  - ♦ Rearranged object files before linking
  - ♦ Attempt to improve locality on calls
  - ♦ 20% system throughput improvement

So, they believed ...

Karl Pettis and Robert C. Hansen, "Profile Guided Code Positioning," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* 1990, pages 16-27.

Code Placement

Pettis & Hansen looked at two distinct problems

Block Placement (*Global optimization*)

- Find blocks on the hot path, bring them together, & use fall-through paths
  - ♦ Fall-through branches are, generally, faster than taken branches
- Rarely executed code can decrease instruction-cache utilization
  - ♦ Always bad to fetch operations that do not execute

Procedure Placement (*Whole-program optimization*)

- If A calls B, would like A & B in adjacent locations
  - ♦ On same page means smaller working set
  - ♦ Adjacent locations limit I-cache conflicts
- Unfortunately, many procedures might call B (& A)
  - ♦ A common & critical problem in interprocedural optimization
- This is an issue for the linker

Block placement precedes procedure placement, both at compile-time & in this lecture.

Block Placement

(*The "global" part of the problem*)

Targets branches with unequal execution frequencies

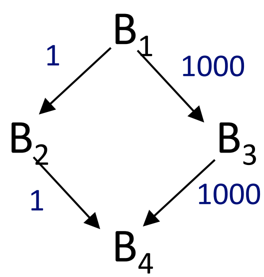
- Make likely case the "fall through" case
- Move unlikely case out-of-line & out-of-sight

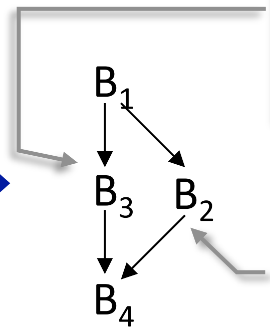
Potential benefits

- Longer branch-free code sequences (*local optimizations, such as* **LVN**)
- More executed operations per cache line
- Denser instruction stream ⇒ fewer cache misses
- Moving rarely executed code ⇒ denser page use & fewer page faults

Block Placement

Moving infrequently executed code

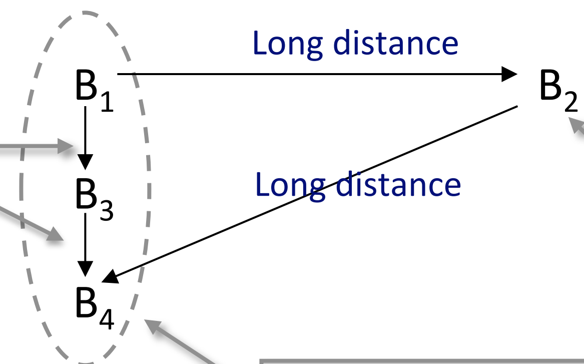




Likely path gets fall through (cheap) case

Unlikely path gets an extra branch

Would like this to become



These branches goes away

In another page, …

Denser instruction stream

Block Placement

Overview

1. Build chains of frequently executed paths (*similar to traces*)
   - ♦ 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.

Profiling: See COMP 412 lecture 39, slide 15ff, or EaC2e, pp. 452-453

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 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***

See example in § 8.6.2 in EaC2e

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

Procedure Placement (*The "interprocedural" part of the problem*)

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 (*advantage on* **PA-RISC**)
  - ♦ Direct mapped I-cache ⇒ A & B unlikely to overlap in I-cache

As much as 80 to 98% reduction in executed long branches

Ignored indirect calls (through a pointer)

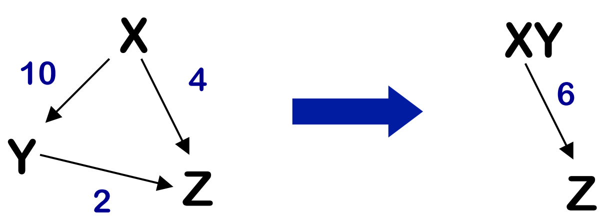
Profiling the Call Graph

- 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

Procedure Placement

Computing an order

- Combine all edges from A to B (*make the multi-graph into a graph*)
- Select highest weight edge, say X→Y
  - Combine X & Y, along with their common edges, X→Z & Y→Z
  - Place X next to Y
- Repeat until graph cannot be reduced further



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

The algorithm is straight forward

Experimental Results

They evaluated several codes on three PA-RISC Models

	Language	*SLOCs*	*Obj Bytes*
Othello	Pascal	1,133	82,139
Simulator	C	21,261	323,168
Pascal	Pascal & C	312,500	2,225,814

	825	835	840
VMIPS	9.8	14.8	8.7
RAM	8MB	40 MB	24 MB
Cache			
Unified/Split	Unified	Unified	Split
Size	16 KB	128 KB	64 KB/64 KB
Associativity	1	2	1/1
Line Size	32 b	23 B	16 B/16 b
Lines/Way	256	2048	4096/4096
Clean Miss Penalty	27	27	7/7
Dirty Miss Penalty	27	27	14/—

Experimental Results

They evaluated several codes on three PA-RISC Models

835	*Procedure*	*Block*	*Block+Fluff*	*Block+Proc*	*All Three*
Othello	0.0	1.6	1.6	2.1	2.1
Simulator	0.0	4.5	5.5	4.5	5.5
Pascal	1.9	5.6	6.9	6.9	7.6

840	*Procedure*	*Block*	*Block+Fluff*	*Block+Proc*	*All Three*
Othello	0.9	4.7	4.4	4.2	4.0
Simulator	9.9	0.6	13.8	13.9	14.9
Pascal	4.3	1.8	9.2	9.3	9.8

825	*Procedure*	*Block*	*Block+Fluff*	*Block+Proc*	*All Three*
Othello	7.2	7.9	8.7	8.7	10.2
Simulator	8.2	16.2	13.9	20.3	26.0

Percentage improvements over baseline full optimization

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 varied, but were significant
  - ♦ 2% to 10% on the PA-RISC; better results on x86 systems
- This technique paid off handsomely on early 1990s PCs
  - ♦ Slow page faults, pages based on 386 segment registers (*rather than hw pagesize*)
  - ♦ Microsoft insiders suggested it was one of the most important optimization for codes like Office (*Word, PowerPoint, Excel*)

Inline Substitution

The oldest interprocedural optimization [Ershov 1966]
The canonical interprocedural optimization

Replace a procedure call with the body of the called procedure

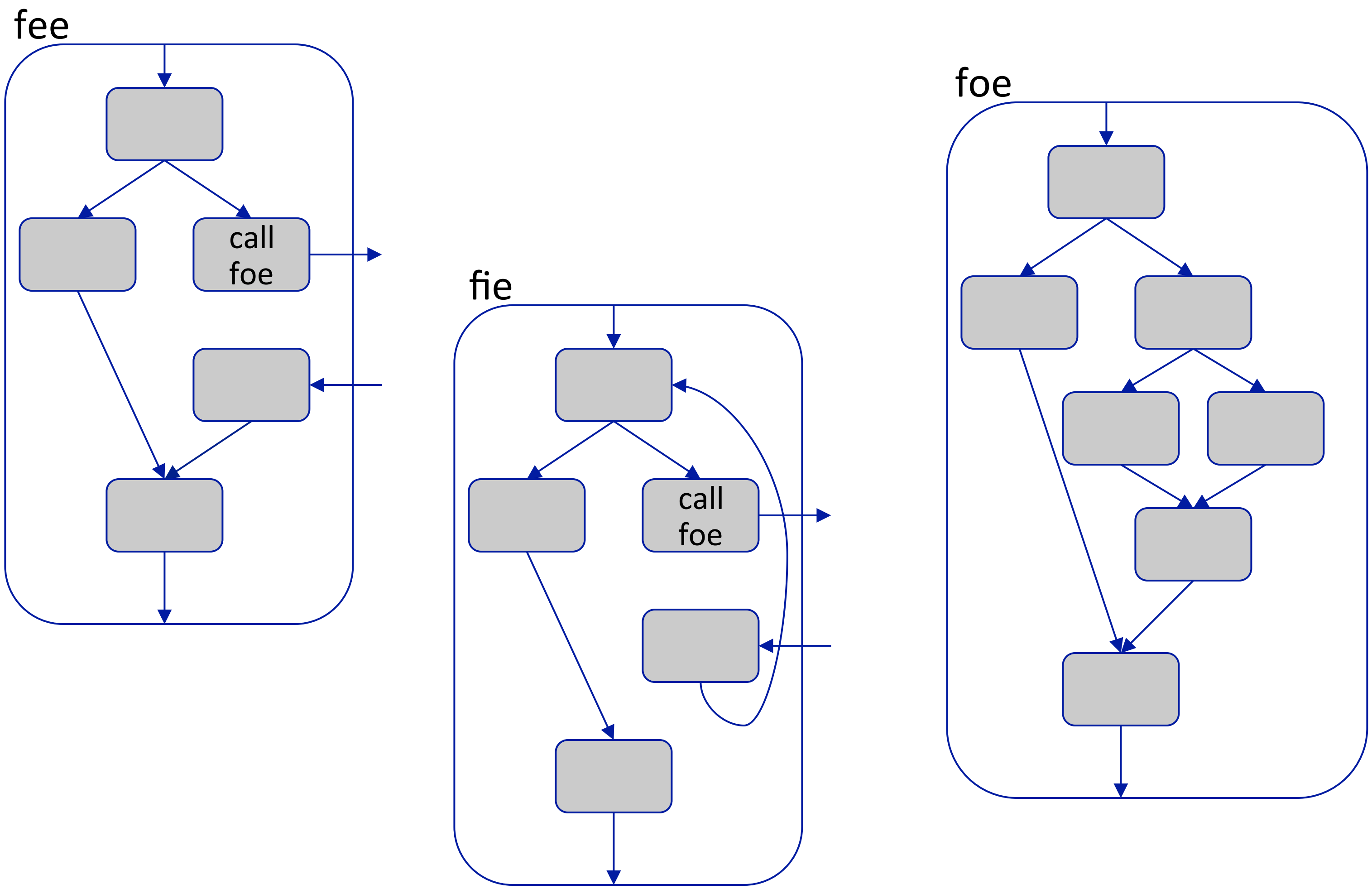
- Textual substitution to create effects of parameter binding
- Private copy of code can be tailored to call site's context
  - ♦ Constants, unambiguous pointers, aliases, …
- Eliminates disruption of procedure call
  - ♦ Register save & restore
  - ♦ Disruption of call & return
- Eliminates benefits of procedure call
  - ♦ Call resets state of register allocator
  - ♦ Procedure abstraction keeps name space small
- Usually assumed to be profitable, although studies disagree …
  - ♦ Some authors report major degradation from code-cache blowout
  - ♦ Those studies are dated; today's processor may have better code caches

Inline substitution plays an important role in optimizing OOLS, due to their relatively high ratio of call overhead to useful work *and* the difficulty of converting virtual calls into direct calls.

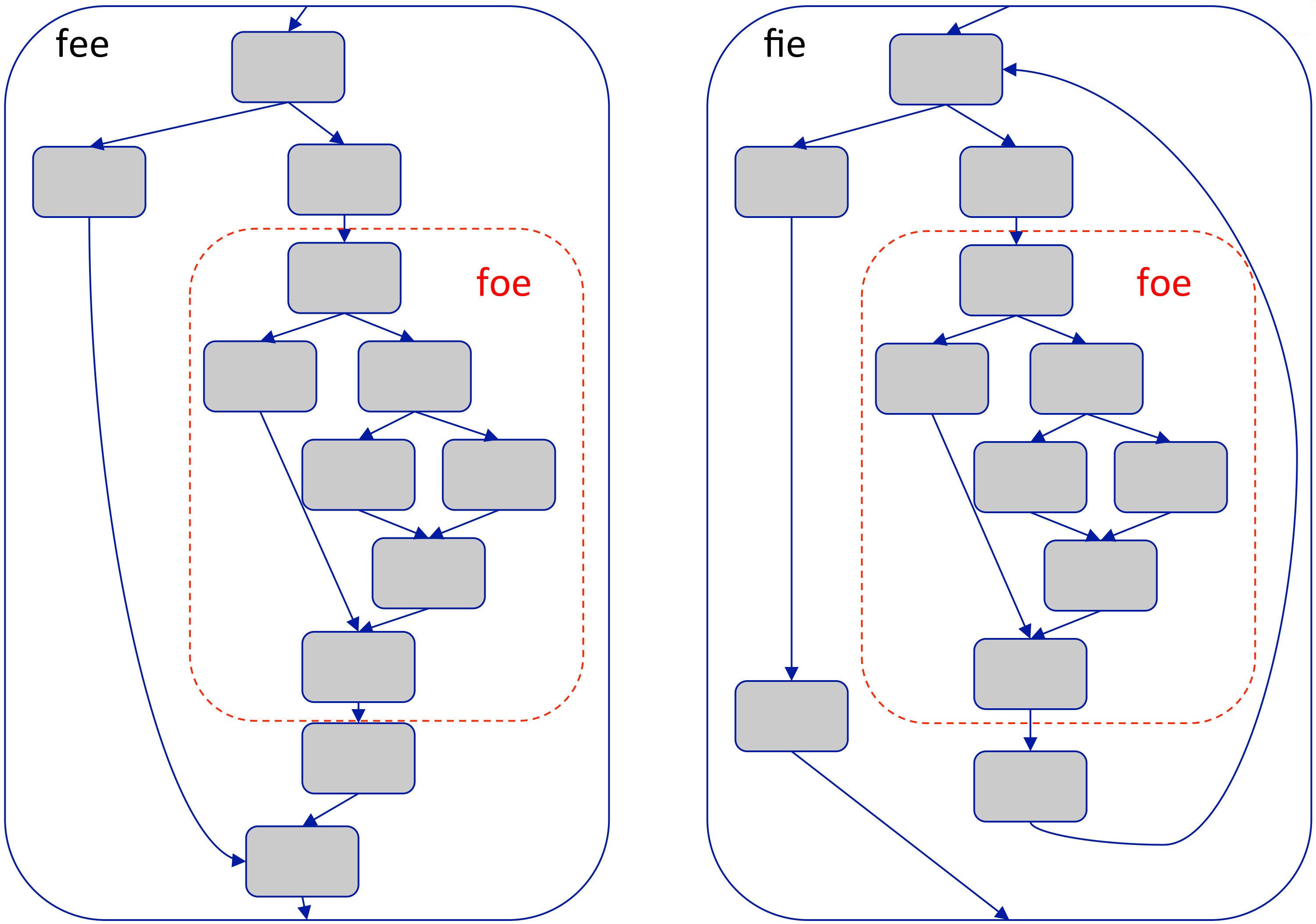
Inlining one call can reveal the relevant class for another …

Inline Substitution

Example



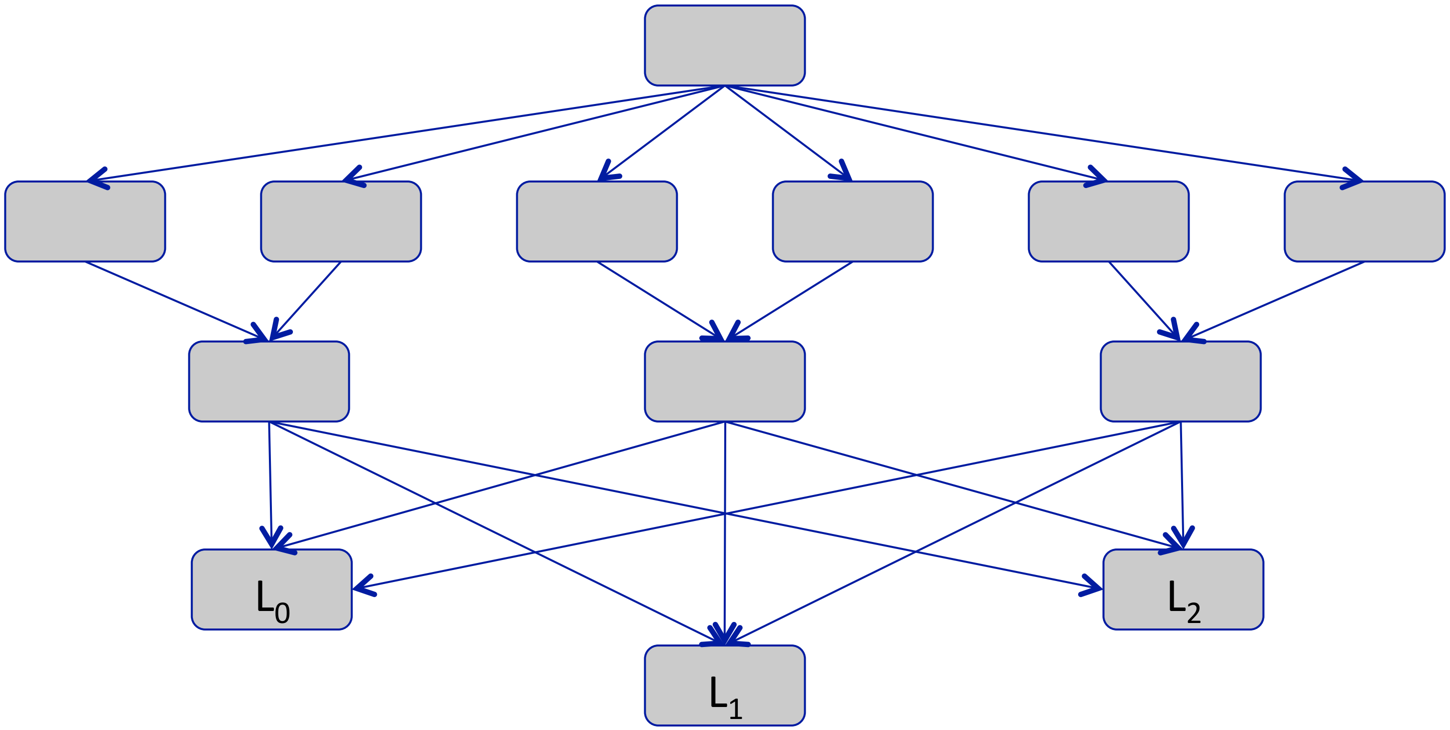
Inline Substitution — After inlining foe



Potential for exponential growth …

Inline Substitution

Potential for exponential growth



Complete inlining would create 6 copies of each of the "leaf" routines, L_0, L_1, L_2

It would also create a call graph with no merge points.

Inline Substitution

The transformation is easy

- Rewrite the call site with the callee's body
- Rewrite formal parameter names with actual parameter names

Safety

- As long as the IR can express the result, it should be safe
- Semantics does not address the number of copies of a procedure in the executable code

Profitability

- The obvious profit comes from eliminating call overhead
- The complications arise from changes in how the code optimizes

Opportunity

- Most implementations traverse the (partial) call graph & look at each edge

Inline Substitution

The transformation is easy

- Rewrite the call site with the callee's body
- Rewrite formal parameter names with actual parameter names

The decision procedure is hard (*quite hard*)

- At a given call site, profitability depends on the extent to which the callee can be tailored to the specific context
  - ♦ Performance can improve or degrade
- Resource constraints limit the amount of inlining
  - ♦ Experience suggests register demand is important
  - ♦ Code size (whole program & current procedure) play a role
    - → *Excessive code growth leads to excessive compilation time*
- Each decision affects profitability & resource use of other call sites

Inline Substitution

Choosing which call sites to inline is hard

- Performance of transformed code is hard to predict
  - ♦ *Recall the story from the introductory lecture …*
- Decisions interact
  - ♦ Inlining A into B changes B's properties
  - ♦ Inlining A into B might make B a leaf
- Can't even name the call sites
  - ♦ Inlining destroys some & creates others
- Some decisions look easy, others look hard
  - ♦ Inline procedure smaller than linkage or called from one place
  - ♦ Don't inline large procedure or calls in critical loops

Existing compilers use heuristics, such as **ORC**'s temperature

ORC eventually became Open64

an edge $E_i(p,q)$ (*i.e.* a call site in function p which calls function q in the call graph).[1]

$$temperature_{E_i(p,q)} = \frac{cycle\_ratio_{E_i(p,q)}}{size\_ratio_q} \qquad (1)$$

where:

$$cycle\_ratio_{E_i(p,q)} = \frac{freq_{E_i(p,q)}}{freq_q} \times \frac{cycle\_count_q}{Total\_cycle\_count} \qquad (2)$$

$freq_{E_i(p,q)}$ is the frequency of the edge $E_i(p,q)$ and $freq_q$ is the overall execution frequency of function q in the training execution.

$Total\_cycle\_count$ is the estimated total execution time of the application:

$$Total\_cycle\_count = \sum_{k \in PUset} cycle\_count_k \qquad (3)$$

$PUset$ is the set of all program units (*i.e.* functions) in the program, $cycle\_count_q$ is the estimated number of cycles spent on function q.

$$cycle\_count_q = \sum_{i \in stmts_q} freq_i \qquad (4)$$

where $stmts_q$ is the set of all statements of function q, $freq_i$ is the frequency of execution of statement i in the training run.

Furthermore, the overall frequency of execution of the callee q is computed by:

$$freq_q = \sum_{k \in callers_q} freq_{E_i(k,q)} \qquad (5)$$

where $callers_q$ is the set of all functions that contain a call to q.

Essentially, $cycle\_ratio$ is the contribution of a call graph edge to the execution time of the whole application. A function's cycle count is the execution time spent in that function, including all its invocations. $(\frac{freq_{E_i(p,q)}}{freq_q} * cycle\_count_q)$ is the number of cycles contributed by the callee q invoked by the edge $E_i(p,q)$. Thus, $cycle\_ratio_{E_i(p,q)}$ is the contribution of the cycles resulting from the call site $E_i(p,q)$ to the application's total cycle count. The larger the $cycle\_ratio_{E_i(p,q)}$ is, the more important the call graph edge.

$$size\_ratio_q = \frac{size_q}{Total\_application\_size} \qquad (6)$$

$Total\_application\_size$ is the estimated size of the application. It is the sum of the estimated sizes of all the functions in the application. $size_q$, the estimated size of the function q, is computed by:

[1] Because function p may call q at different call sites, the pair (p,q) does not define an unique call site. Thus, we add the subscript i to uniquely identify the i^{th} call site from p to q.

ORC's Heuristic

Compute a "temperature" for each call site

- Complicated computation
- Single number to characterize each site
- Inline sites that are hotter than some threshold
- Tuning implies choosing the threshold

Explanation actually goes on for another half page

From "To Inline or Not to Inline? Enhanced Inlining Decisions" by Zhao & Amaral

Inline Substitution

What have we learned?

- Inline substitution cures many inefficiencies that can arise at a call site
  - ♦ Eliminates overhead
  - ♦ Allows context-specific tailoring
  - ♦ Eliminates disruption to analysis in both caller and callee

- Inline substitution can cause its own problems
  - ♦ Unlimited compilation times (*ignoring the MIPS story*)
  - ♦ Performance degradation
  - ♦ Significant code growth

And, there are other consequences of inline substitution …

Interprocedural Optimization

Complications of interprocedural optimization

- Compiler needs access to all the code being improved
  - ♦ Conflicts with separate compilation
  - ♦ Alternatives: whole program compile, link-time optimization, or some system where the compiler can "see" source code
- Resulting object code depends in subtle ways on rest of code
  - ♦ Safety based on data-flow facts in the rest of the code
  - ♦ Remote changes can invalidate optimization decisions
  - ♦ Must recompile all code after each edit, or analyze the dependences to reduce the amount of recompilation

Easiest route is to perform interprocedural optimization at runtime

Decision Procedures

Of course, the hard part is deciding what to do …

- Decision for one call affects behavior at other sites
- Difficult to predict effects
  - ♦ Demand for registers can cause increased spilling
  - ♦ Inlined code can have much larger name space (analysis)
  - ♦ Quality of global optimization may fall with procedure size
- MIPSPro computes a quantitative score
  - ♦ Gives a yes or no answer based on potential and size
- Some decisions are obvious
  - ♦ Inline small procedures (< linkage size)
  - ♦ Inline procedures called only once (leaf procedures)
- Still room for experimental work
  - ♦ See Cooper, Hall, & Torczon [99] or Davidson & Holler or Waterman 2008

Jack W. Davidson and Anne M. Holler, "A Study of a C Function Inliner", S—P &E, 18(8), August 1988, pages 775-790.
Keith D. Cooper, Timothy J. Harvey, and Todd Waterman, "An Adaptive Strategy for Inline Substitution", CC '08/ETAPS '08, March, 2008.