Interprocedural Optimization

Enough Analysis, What Can We Do?

There are interprocedural optimizations
• Choosing custom procedure linkages
• Interprocedural register allocation
• Interprocedural common subexpression elimination
• Interprocedural code motion
• Memo-function implementation
• Cross-jumping
• Procedure recognition & abstraction
**Linkage Tailoring**

Should choose the best linkage for circumstances

- Open (inlined), clone, semi-open, semi-closed, closed

---

**Linkage styles**

Traditional *closed* linkage

- Inlining eliminates the call overhead
- Inlining creates tailored private copy of callee

*Open linkage (inlined)*
**Linkage Tailoring**

**Linkage styles**

- Traditional closed linkage
- Semi-open linkage

- Move prolog & epilog across call & tailor them
- For call in loop, parts of linkage are loop invariant

*Think of this as an aggressive closed linkage*

**Procedure Cloning**

- Partition call sites by environment
- Create a location where desired facts can be true
**Linkage Tailoring**

Should choose the best linkage for circumstances
- Open, clone, semi-open, semi-closed, closed
- Estimate execution frequencies & improvements
- Assign styles to call sites

Practical approach
- Limit choices
  - standard, cloned, inlined
- Clone for better information & to specialize
  - based on idfa
- Inline for high-payoff optimizations
- Adopt an aggressive standard linkage
  - Move parameter addressing code out of callee (& out of loop)
  - Parts of prologue & epilogue that are predictable in caller

**Improving Linkages**

Attack the actual inefficiencies
- Save & restore code
  - Live across a call ⇒ target callee-saves register
  - Not live across a call ⇒ target caller-saves register
  - Leaf procedure ⇒ target caller-saves register
- Optimize actual save & restore code
  - Code space issue ⇒ use a common library routine
  - Avoid saving anything that is not needed
- Generate parameter bindings in a way that will allocate well
  - Copy actual into formal & count on coalescing (when possible)
- Return address in a register?
  - Make it the first spill (store at entry, load at exit)
- Expose pre-call & post-return sequences to LICM & GCSE
  - Expand operations before optimization
**Linkage Tailoring**

A low-level, post-compilation idea

- Compute register summaries for each procedure
  - How many registers are really used?
- Rotate assignment to move them into caller-saves registers
  - Leave callee-saves registers unused, if possible
  - Avoid saving them in the callee

Leaf routine with little demand for registers

- Adjust caller-saves/callee-saves boundary
- Remove caller’s stores for registers unused in callee

*Some authors call this interprocedural register allocation*

---

**Interprocedural Register Allocation**

Some work has been done

- Chow’s compiler for the MIPS machine, “shrink wrapping”
  - Often slowed down the code
- Wall and Goodwin at DEC SRC did link-time allocation
  - Target machine had scads of registers
  - Essentially, adjusted caller-saves & callee-saves

What about full-blown, allocate the whole thing, allocation?

- Arithmetic of costs is pretty complex
- Requires good profile or frequency information
- Need a fair basis for comparing different uses for $r_i$
- Real issue would be spilling  (*“spill everywhere” would be a disaster* )
**Interprocedural Common Subexpression Elimination**

Consider the domain carefully
- Procedures only share parameters, globals, & constants
- **No** local variables in an ICSE
- Not a very large set of expressions
  - Includes, however, parameter & global variable addresses

Possible schemes
- Create a global data area to hold ICSEs
  - Sidestep issue of register pressure
- Ellide unnecessary parameters
  - Speeding up linkages

---

**Interprocedural Code Motion**

What could we do? What could we find?
- Find and mark loop nests in the call graph
- Compute interprocedural AVAIL?
- Move code across procedure boundaries

All are difficult

Two ideas
- **Invocation invariant expressions**
  - Expression whose value is determined at point of call
  - Hoist them to the prolog, or hoist them across the call
- **Loop embedding and extraction**
  - Move control-flow for entire nest into one procedure
  - Enable optimization using intra-procedural techniques
Invocation Invariant Expressions

Do iie’s exist? (yes)

Is moving them profitable? (it should be)

Can we engineer this into a compiler? (this is tougher)

<table>
<thead>
<tr>
<th>Routine</th>
<th>size</th>
<th>add</th>
<th>sub</th>
<th>mult</th>
<th>conv</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>decomp</td>
<td>933</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>fehl</td>
<td>621</td>
<td>33</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>fib</td>
<td>57</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>fmin</td>
<td>601</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>rk45</td>
<td>191</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>rkfs</td>
<td>1502</td>
<td>23</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>seval</td>
<td>219</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>solve</td>
<td>287</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>spline</td>
<td>1105</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>svd</td>
<td>2372</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>zerosn</td>
<td>462</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

Distinct iie’s found in FMM

8.9% to 0.2% (static)
1.66% on average

Interprocedural Code Motion

Moving loops across a call

```plaintext
do i = 1 to 100
  do j = 1 to 100
    call fee(a,b,c)
    do i = 1 to 100
      do j = 1 to 100
        call fee(a,b,c,i,j)
        fee(x,y,z)
      end
      end
      end
      fee(x,y,z,i,j)
    end
  end
  end
end
```

Loop Embedding

Think of this as dual of partial inlining (partial outlining?)
Interprocedural Code Motion

Moving loops across a call

```
do i = 1 to 100
  do j = 1 to 100
    call fee(a,b,c,i,j)
  end
end

fee(x,y,z,i,j)
  do k = 1 to 100
    x(i,j) = x(i,j) + y(i,k) * z(k,j)
  end
end
```

Loop Extraction

Think of this as partial inlining

A Practical Idea for Applying Interprocedural Opt's

Code Motion, Redundancy Elimination, & Linkage Tailoring are hard to engineer into a compiler because all call sites that invoke the transformed procedure must, themselves, be transformed.

• Bookkeeping, source access, and recompilation are nightmares

Spillman had a simple solution, back in 1971

• Optimize together procedures in a single compilation unit
  > Distinct (optimized) entry points for calls from within the unit
  > Distinct (optimized) behavior for those entry points
• Since all the code for the unit is compiled together, this idea sidesteps many of the complications of whole-program opt’n
• Allows the user to govern the granularity of compilation and optimization
• Opportunities abound: static in C, private methods in an OOL

**Memo-function implementation**

**Idea**
- Find pure functions & turn them into hashed lookups

**Implementation**
- Use interprocedural analysis to identify pure functions
- Insert stub with lookup between call & evaluation

**Benefits**
- Replace evaluations with table lookup
- Potential for substantial run-time savings
- Should share table implementation with other functions

**Cross-jumping**

**Idea**
- Procedure epilogs come in two flavors
  - Returned value & no returned value
- Eliminate duplicates & save space

**Implementation**
- At start of each block, compare ops before predecessor branch
- If identical, move it across the branch
- Repeat until code stops changing

**Presents new challenges to the debugger**
**Procedure Abstraction**

**Idea**
- Recognize common instruction sequences
- Replace them with (very) cheap calls

**Experience**
- Need to abstract register names & local labels
- Use suffix trees ~ 1% slower for each 2% smaller
- Causes havoc for debugger

**Where Can IDFAO Have An Impact?**

There are few killer interprocedural optimizations
- Inlining and cloning, particularly for OO programs
- Analysis to avoid method lookup in class hierarchy
- Trace caches in dynamic systems

What about traditional Algol-like languages?
- MayMod helps single-procedure constant propagation (CP)
  > Lets SCCP pass constants across a call
- Procedure cloning on forward constants helps ICP
- Pointer analysis enables register promotion (Lecture 22)
- Inlining reduces call overhead
  > Particularly with small procedures, data abstraction languages (CLU), & OOLs (C++)

Grove & Torczon 93
Metzger & Stroud 92
Lu & Cooper 97
Scheiffler 77
OOPLSA