Register Promotion in C Programs
— improving code in the presence of pointers —

Background

Code in C & C++ often relies on pointer-based variables

- Presence of pointers makes many values ambiguous
  - Cannot keep an ambiguous value in a register
  - Cannot reason about definitions, uses, and such
  - Serious impediment to optimization
- Compilers can perform analysis to narrow the range of addresses (variables?) to which a pointer can refer
  - Points-to analysis builds a set of names for each pointer (in each block?)
  - If points-to set for p has 1 element, value is unambiguous
    → Compiler can treat it as a scalar
- How can the compiler use points-to information to improve code?
  - Indirect uses: increase precision of other analyses
  - Direct use: rewrite unambiguous values as scalars a la Carr
**Background: Scalar Replacement**

Array elements are hard for the compiler to keep in a register

- A reference, such as \( c(i) \), is typically treated as ambiguous
  - Given \( c(i) \) and \( c(j) \), can they refer to the same location?
  - Unless the compiler knows, it must be conservative
- When the compiler knows, it can rewrite the unambiguous array element reference as a local scalar variable and the allocator will keep it in a register

Steve Carr’s Scalar Replacement Transformation

- Locate patterns of consistent reuse
- Make loads and stores use temporary scalar variable
- Replace references with temporary’s name
- May need copies at end of loop to keep reused values straight
  - If reuse spans more than one iteration, need to “pipeline” it

---

*Example*

Effects

- Decreases number of loads and stores
- Keeps reused values in names that can be allocated to registers
- In essence, this exposes the reuse of \( a(i) \) to subsequent passes

---

*D. Callahan, S. Carr, and K. Kennedy, “Improving Register Allocation for Subscripted Variables”, PLDI 90.*
Rewriting Loops for Better Register Allocation

What if we are not in Fortran? What about C?

Register Promotion
- Analog of scalar replacement for ambiguous values
  - References such as \(^*p\), \(p->x\), ...
- Find the set of references that are unambiguous
- Promote such references into scalar temporaries
- Requires data-flow information on pointers + a transformation

Lu’s formulation
- Perform interprocedural analysis to disambiguate pointers
- Find loops & solve intraprocedural problem for each loop
- Rewrite code based on results of analysis

His work relied on ILOC’s memory tags

Cooper and Lu, “Register Promotion in C Programs”, PLDI 97.
Sastry and Ju, “A New Algorithm for Scalar Register Promotion Based on SSA Form”, PLDI 98.

ILOC Memory Tags

In the MSCP Compiler, certain operations had sets of “memory tags”, labels that indicated which memory locations they might reference. Specifically, loads, stores, and calls had tag sets.

- Front end generated the tags from its local analysis
- Default tag was a mangled name (source name or temp name)
- Every pass in the compiler used the Memory Tags to determine how aggressively they could treat each value

Role of points-to information
- Points-to analysis computed equivalent information to tag sets
- MSCP compiler used points-to sets to shrink its tag sets
- When we added Lu’s points-to implementation, the final code produced by the compiler got better (as expected).

Memory tags were a clean, general mechanism to handle this info
**Code Shape**

The Role of the IR

- Some loads are *explicit*; they produce unambiguous results.
  > Examples: load immediate, load from constant address or constant offset from ARP
  > Any load with a single element tag set is explicit
- Some loads are *ambiguous*; they produce ambiguous results.
  > Examples: load from address in register
  > Any load with a multi-element tag set is ambiguous

If the compiler is scrupulous about encoding knowledge about ambiguity in the IR, the compiler can use that same knowledge

---

**The Plan**

1. Perform interprocedural points-to analysis on the entire code and use the results to shrink tag sets on references & procedure calls
2. For each procedure in the code
   A. For each block, compute $B_{Explicit}$ and $B_{Ambiguous}$
   B. Identify the loop nests [Lengauer & Tarjan]
   C. For each loop†
      1) Compute the data-flow sets to find promotable values
      2) Rewrite the code so that each value in promotable value is kept in a new virtual register
      3) Promote the loads and stores into the appropriate loops

†Work one loop nest at a time, in order from innermost loop to outermost loop.
Preparing to Solve the Problem

Initial data-flow information

- \( B_{\text{Explicit}}(b) \) contains all tags referenced by an explicit memory operation in \( b \)
- \( B_{\text{Ambiguous}}(b) \) contains all tags referenced by a memory operation with multiple tags or by a procedure call in \( b \)

The compiler can compute these sets in a linear pass over the block. It can build the CFG at the same time.

To find loops, we can use the dominator-based technique described by Lengauer and Tarjan.

The Data-Flow Problem

For a loop, compute

\[
L_{\text{Explicit}}(L) = \bigcup_{b \text{ in loop } L} B_{\text{Explicit}}(b)
\]
\[
L_{\text{Ambiguous}}(L) = \bigcup_{b \text{ in loop } L} B_{\text{Ambiguous}}(b)
\]
\[
L_{\text{Promotable}}(L) = L_{\text{Explicit}}(L) - L_{\text{Ambiguous}}(L)
\]
\[
L_{\text{Lift}}(L) = \begin{cases} 
L_{\text{Promotable}}(L) & \text{if } L \text{ is an outermost loop} \\
L_{\text{Promotable}}(L) - L_{\text{Promotable}}(S) & \text{S surrounds } L
\end{cases}
\]

How does this work?

- \( L_{\text{Explicit}} \) and \( L_{\text{Ambiguous}} \) accumulate info from all the blocks in a loop
- \( L_{\text{Promotable}} \) shows which tags can be promoted in the loop
- \( L_{\text{Lift}} \) shows which tags can be loaded and stored in a given loop

This may be the simplest set of data-flow equations ever published in PLDI.
**Rewriting the Loops**

To rewrite the code

- For each tag in L_Promotable, choose a new virtual register name \( vr \) and rewrite all defs and uses inside the loop with \( vr \)
  - Some references will not need to be rewritten.
  - Detail will vary by compiler. In MSCP, values with “scalar” loads & stores were unambiguous. Analysis ignored them because they were handled appropriately without help.
- Now, \( vr \) must be loaded from the memory location before the loop and stored back after the loop.
  - \( \text{L Lift}(x) \) contains all of the tags that must be loaded and stored in loop \( x \)
  - Treat the inner loop as an atomic unit and place the loads before the inner loop and the stores after the inner loop

---

**An Improvement**

What about a pointer-based value, such as \( B[i] \)?

- Algorithm, as explained so far, rewrites scalars
- To handle arrays, Lu extended the algorithm
  - Use of \( B[i] \) in a loop, with invariant \( i \)
  - Need to recognize that \( B[i] \) computes the same address in each iteration of the loop
- Lu relied on prior application of loop invariant code motion
  - Compiler used *partial redundancy evaluation*
  - Can use any other effective form of LICM
- After LICM, the definition of the \( vr \) that holds the address of \( B \) is outside the loop, as is the definition of \( i \)
  - Add minor amount of code to recognize and promote the references in an “address-offset” form with definitions outside the loop

Plenty of room to improve this aspect of the algorithm, to achieve more Carr-like results.
**Rewriting Loops for Better Register Allocation**

**Register Promotion - Example from the paper**

```c
for (i = 0; i < DIM_X; i++) {
    B[i] = 0;
    for (j=0; j < DIM_Y; j++) {
        B[i] += A[i][j];
    }
}
```

With this simple scheme

- 0 to 16% of loads removed in test codes
- 0 to 50% of stores removed in test codes
- Other authors tried PRE-based extensions to Lu’s work

---

**Strengths & Weaknesses**

**Strengths**

- Loop-oriented algorithm relies on local notion of ambiguity
  - Scalar value, ambiguous pointer to it
  - Can keep it in a register in a loop where pointer is not used
- Easy to understand and to implement

**Weaknesses**

- Approach to proving independence is simplistic & naïve
- Stronger approaches would allow promotion of more references
Later Work

Sastry and Ju (PLDI 98)

- Algorithm designed to work directly on SSA form
  - Show how to update SSA form to reflect newly inserted defs
  - Their algorithm operates on intervals† rather than loops
- Unit of promotion is a live range inside an interval
  - A name, in Lu, may form several live ranges
- Profitability is estimated with profile information
- Show ~12% reduction in loads & stores that access scalars
  - No specific attempt to address array element values

This form of the algorithm would be a good choice for the lab.
The exposition in the paper is clean, and the algorithm is not overly complex.

† See, for example, F.E. Allen, “Control Flow Analysis,” in 1st ACM SIGPLAN Conference on Compiler Construction, 1970.

Later Work

Lo, Chow, Kennedy, Liu, and Tu (PLDI 98)

- Algorithm based on SSAPRE (Chow et al., PLDI 97)
  - Adapt SSAPRE for speculative code motion (both static & profile driven) of loads
  - Develop a dual form (static single use form) & algorithm for speculative code motion (static & profile driven) of stores
- Separate passes for loads & stores
  - Slotwise analysis & transformation
  - Run SSAPRE variants twice

This algorithm is significantly more complex than Sastry & Ju.
It has all the benefits of the SSAPRE-family of algorithms.