# Loop Fusion Amid Complex Control Flow

R Ramshankar

Dibyendu Das

AMD

# Loop Fusion

Two loops with proximity in control flow iterating over same large arrays

- Will show poor scalability
- Why? Loops on large arrays stride over memory that is too big to fit in the cache.
- Loops can be fused if dependences can be preserved, but
  - How do we deal with proximity amid complex control flows (and function calls)?

# Loop fusion with control dependence

- Build from trivial loop fusion: adjacent loops
  - Loops are typically guarded by an if (i != end) condition
  - Control dependence graph: derive from the CFG
    - If two loops have the same or almost identical control dependence

# Control dependence

If (x) { A; }

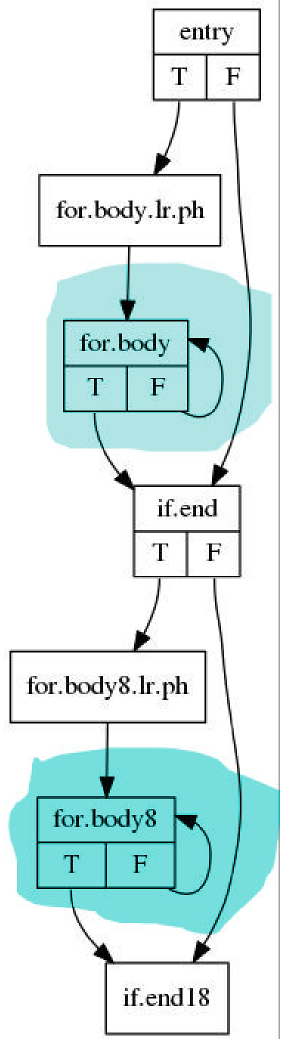
A is *control-dependent* on the block that contains the conditional branch *BR (x == true), A*

(i.e., A is *control-dependent* on the block that decides to bypass A or go to A)

- Formally, a statement *y* is said to be *control dependent* on another statement *x* if
  - (1) there exists a non-trivial path from *x* to *y* such that every statement $z \neq x$ in the path is post-dominated by *y* and
  - (2) *x* is not post-dominated by *y*
- Added the control dependence construction algorithm from Kennedy/Allen

# Generic CFG pattern containing natural loops

```
int test(int A[], long size…) {
  long i =0;
  for (i=0; i < size; i++) {
    A[i] |= (1 << a);
  }
  for (i=0; i < size; i++) {
    A[i] |= (1 << b);
  }
// …
   return 0;
}
```



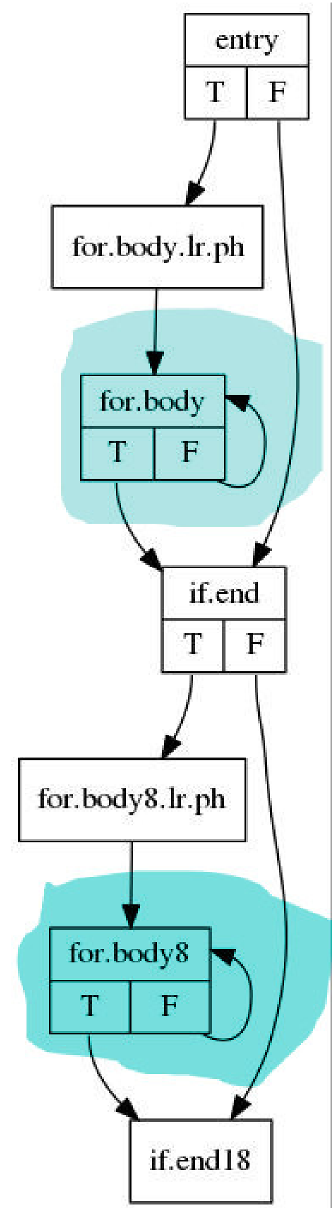

CFG for 'test' function

- *entry* leads to the first loop
  - By nature, a control dependence
- Generalize based on this standard pattern
  - Two proximal singly nested loops
    - For ex: proximal in breadth-first order
  - What if instead of the single blocks “entry”/”if.end” we have complex control flow?

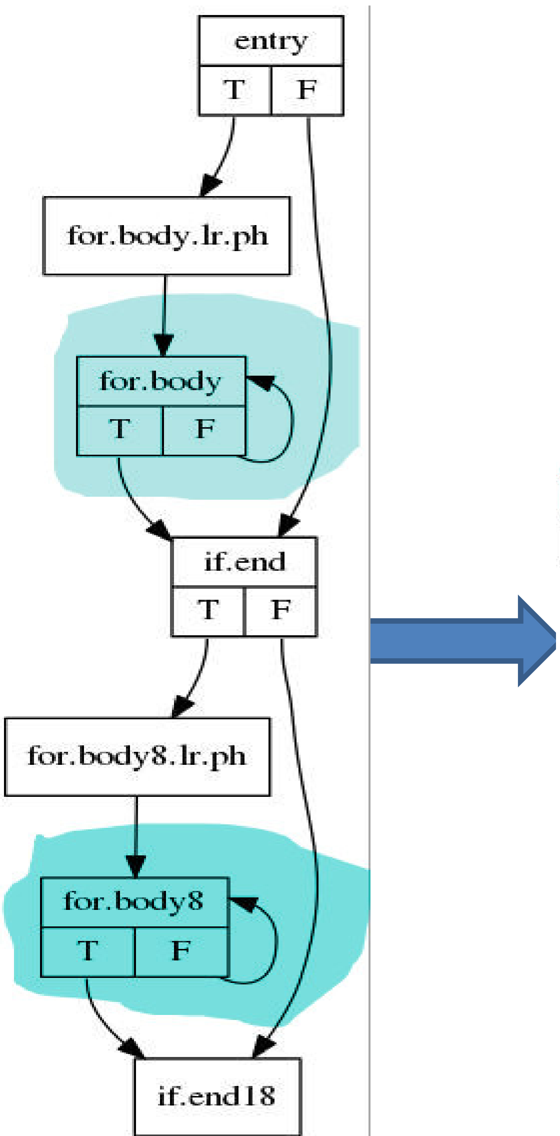
# Fusing loops despite complex control flow: slicing out paths from the CFG

```
int test(int A[], long size, int a, int b, int c,
int d, int e) {
 long i =0;
 if (a & b) {
  for (i=0; i < size; i++) {
   A[i] |= …;
  }
 }
 if (d&e) {
  for (i=0; i < size; i++) {
   A[i] |= …;
  }
 }
…
```

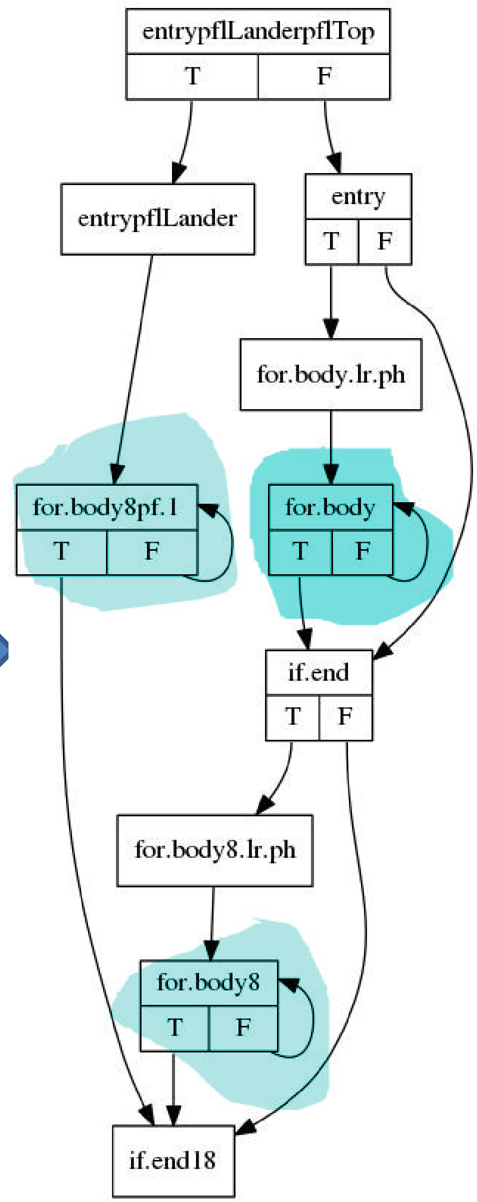
- *Suppose a&b and d&e are not mutually exclusive*
  - *Loop fusion will be of benefit*
- *entry* and *if.end* are the control-dependences
- *entry* **dominates** *if.end and if.end* **post-dominates** *entry*
- *if.end* is the single exit for first loop (could be a DAG)
- if.end18 is the first common post-dominator of the loops' exits
- Handle complex control flow by this approach: Transform the CFG by duplicating paths leading from *entry* to *if.end18*
- Use aforementioned dominance/control dependence relations



CFG for 'test' function



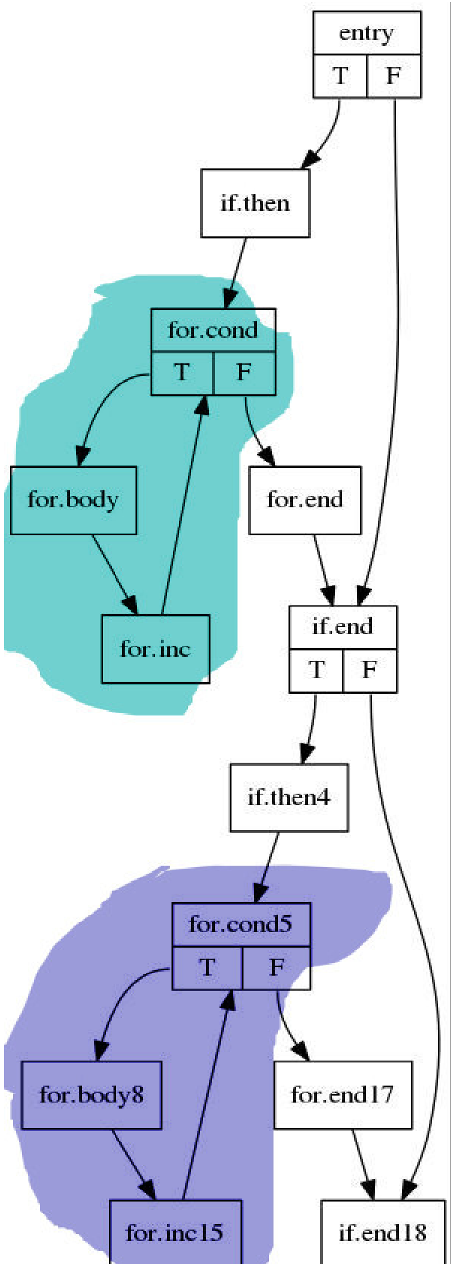
CFG for 'test' function



CFG for 'test' function

# Loop fusion

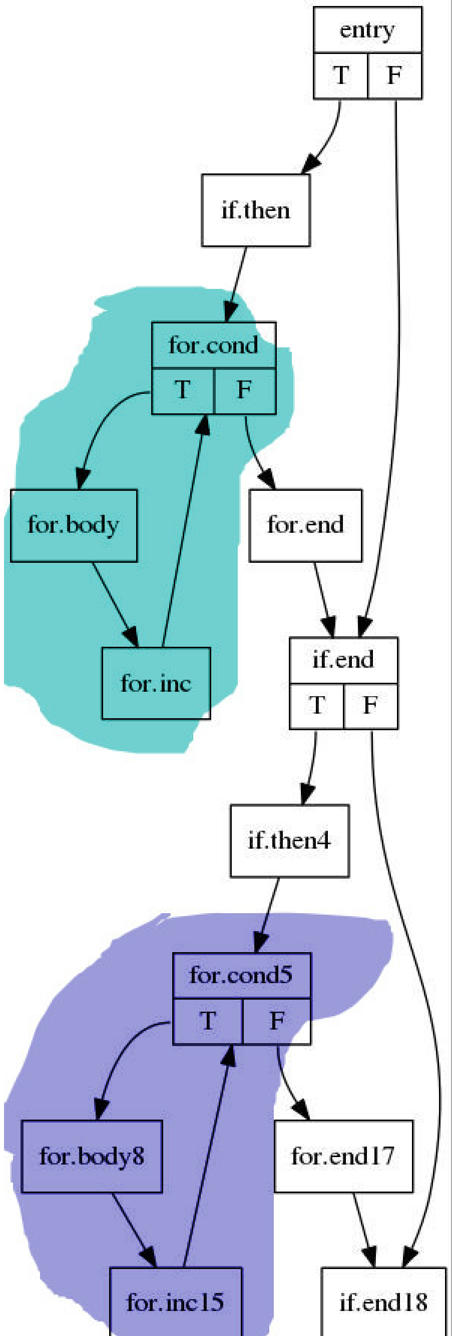
- To fuse merge *entry*, *if.end* blocks
  - Create control flow: no need for C/C++ short-circuiting
    - All conditions are anticipated at *entry*: collapse conditions with *bitwise-and*: done here in *entrypflLander*
- Fuse all the way to the common post-dominator for both loop's exits: *if.end18*
  - Preserves the CFG structure; easy recursive application of loop fusion with subsequent loops



CFG for 'test' function

# Loop fusion – control merging using closures

- We want to allow more control-dependences to be merged:
  - Create closures of the control dependence graph
    - Warshall's algorithm
  - Ensure that the newly created control flow preserves data dependences
  - Start from the common control prefix of the two loops and attempt to merge or collapse the suffices
  - Control how different the closures are using a heuristic number on the size of suffices (<5 control dependences now)



CFG for 'test' function

# Head and tail control flow strands

- *for.end* could be more than one block
  - Deal with tail control flows between the two loops
  - Likewise with *if.then:* there can be head control flows leading to the two loops
- The approach used at this time is to enumerate all paths through the head/tail control flow blocks and insert the fused loop in each path
  - Managing this with profile data should be more profitable (TBD)
  - Orthogonal approach would be code-motion(TBD)

# Fusing more than two adjacent loops

- Recursive application of fusion using a graph with edges between loop fusion candidates
  - Share a prefix control dependence closure
  - Second loop has a control dependence parent that post-dominates first loop's exit
  - Breadth-first order of the control flow graph breaks ties
    - Provides a proximity metric
    - Perhaps allows rethinking recursions until fixed point
- Walk over the graph and merge from bottom-up
- Iteratively build loop graphs and fuse, until fixed point (or a specific number of iterations)
  - Intensive optimization

# Complex control flow

- Dependences/aliases/phis/opaque-calls will prune the number of collapsed paths
- Adjacent function calls may have loops that can be fused
  - Inlining may allow some loops to be fused
  - Function unswitching (useful approach that looks for the quickly exiting function pattern)
- Inter-procedural mod-ref information provide additional alias information
  - Added metadata to carry over address non-taken global mod-ref info in load/stores for use in scalar transforms or analysis
- Inline functions in a selective manner
  - Walk over call graph SCCs and ascertain if inlining a call may allow loop fusion

# Dependence analysis

- First cut approach chooses inner-most loops that are simple (for example, loops that may be favored by the loopvectorizer)
- Need to develop a cache model that verifies to a certain degree of accuracy if loop fusion will be beneficial or not
- Exit/step SCEVs of both loops are checked to be exact matches, check for no LCD with the dependence analyzer
- Used LLVM Dependence Analyzer
  - Dependency Analyzer is said not to be robust, but was able to handle our tests

# Results (preliminary)

- Several synthetic cases demonstrate effectiveness
  - for() {} if () { for(){} } else { for () }
  - for() {} if () { for(){} }
  - for() {} for() {}
  - if() {for() {}} if() {for() {} }
  - For large arrays fusion improved performance almost exponentially
- Improves SPECCPU INT 2006
- 462.libquantum rate performance improves close to 2.5X in x86 (AMD/Intel)
  - Non-trivial control flow, inlining, unswitching, global mod-ref
  - more than 100 loop fusion steps
- POC code received favorable response from llvmdev
  - Working to address llvmdev comments
- Need to explore way for use of profile information

# Reference


- R. Allen and K. Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann 2001, ISBN 1-55860-286-0
- S. S. Muchnick, Advanced Compiler Design and Implementation. Morgan Kaufmann 1997, ISBN 1-55860-320-4
- M. Wolfe: High performance compilers for parallel computing. Addison-Wesley 1996, ISBN 0-8053-2730-4



**Trademark Attribution**

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.