C2 Compiler IR – Refresher

Jatin Bhateja

Agenda

- Traditional SSA IR
- C2 Sea of Nodes IR
- Debugging flags
- APX – Conditional Compare design

Traditional SSA IR

- Compiler translates the programs written in high level languages like C/C++/Java into native instructions.
- Traditional compilation pipeline has following stages

Source => [Parser] => AST => [Resolution] {Semantic Analysis, argument matching, symbol resolution, type coercion, constant folding} => HIR {SSA} => [Optimizations] => [Instruction Selection] => LIR/MIR => [Register Allocation] => [Instruction Scheduling] => [Code Gen]

Need for an IR

Q. Why do we need IR ?

A. To de-couple compiler front-end, mid-end and code-generation.

Q. Why does traditional compilers support multiple kinds of IR, like AST, HIR, MIR, LIR ?

A. To remain memory friendly i.e. each kind of IR cater to a set of passes in the compilation pipeline. E.g. AST captures lots of source level details which may not be relevant to down stream passes, low level IR is mostly composed of operands and opcode and does not need any other information.

SSA IR

- Three address code, every assignment to a variable results into creation of a new definition.
- This removes lots of complexity for Data Flow Analysis.
- At convergence points – Phi nodes are inserted which selects one out of many definitions of a variable from converging control flows.
- To optimize the placements of Phi Node compiler computes dominance frontiers.

SSA IR

- Program in traditional SSA IR is represented as a set of basic blocks.
- Each basic block has single entry and single exit.
- A block contains sequence of operations and there is on IR node for each operation.
- Terminal IR node in each block is a control transfer operation like GOTO, IF BRANCH, RETURN.

e.g

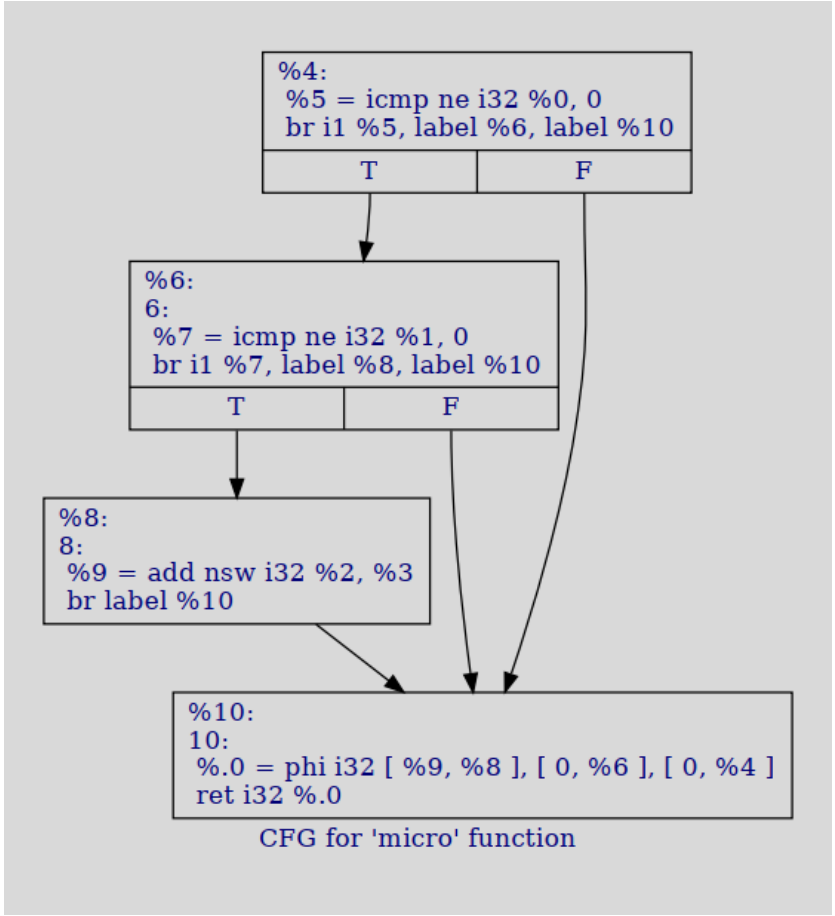
```c
int micro(int cond1, int cond2, int a, int b) {
   int res = 0;
   if (cond1 && cond2) {
      res = a + b;
   }
   return res;
}
```

```
; ModuleID = 'conditional_compare.ll'
source_filename = "conditional_compare.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @micro(i32 %0, i32 %1, i32 %2, i32 %3) #0 {
  %5 = icmp ne i32 %0, 0
  br i1 %5, label %6, label %10

6:                                                ; preds = %4
  %7 = icmp ne i32 %1, 0
  br i1 %7, label %8, label %10

8:                                                ; preds = %6
  %9 = add nsw i32 %2, %3
  br label %10

10:                                               ; preds = %8, %6, %4
  %.0 = phi i32 [ %9, %8 ], [ 0, %6 ], [ 0, %4 ]
  ret i32 %.0
}

attributes #0 = { noinline nounwind uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "frame-pointer"="all" "less-precise
-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="fal
se" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

SSA IR

- Program in IR form has a non-linear structure of a graph with vertices and edges.
- Each operation has a corresponding IR node, and a node has list of incoming and outgoing edges pointing or other IR nodes.
- In traditional compilers there is a clear separation b/w control and data flow graph.
- Vertices in a control flow graph comprises of basic blocks and edges connects to predecessors and successor blocks.
- While vertices of a data flow graph comprises of operation IR nodes and edges connects it to outputs or input to other IR nodes.

Control and Data flow graph



Analysis Passes

- Control Flow
  - Successor - During construction.
  - Predecessor - During construction.
  - Dominator – A node n dominates node m if every path from entry reaching m passes through n. Tarjan Lengauer Algorithm.
  - Post-dominator – A node n post dominates node m if every path from node m to exit passes through n.
  - Dominance Frontier – Needed to compute Phi placements.
  - Loop Detection – Based on Dominator analysis, an edge whose head dominates its tail is a back edge.

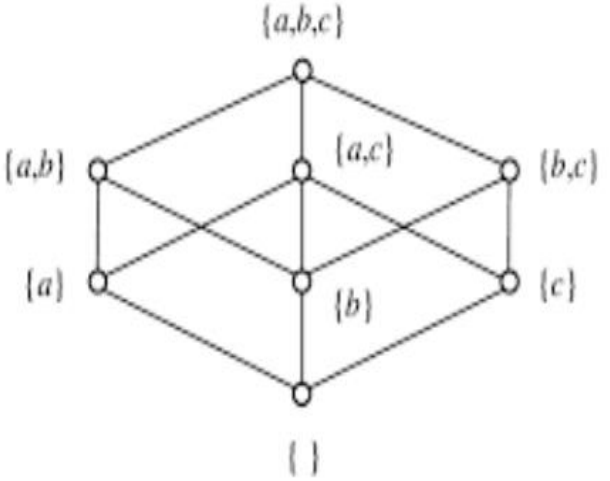
Data Flow

- Analysis extracted out of flow of data though IR graph which refines help in refining the optimizations.
- Basics
  - Data flow information is captured in a data structure called lattice.
  - Lattice holds information about the value range of its definition.
  - E.g. int a; [Lattice : hi = MAX_INT, lo = MIN_INT, while int a = 10 has hi == lo == 10.
  - In theory, lattice is a partial order relegation i.e. is reflexive, anti-symmetric and transitive.
  - Lattice set has two special values Top (all the values) and Bottom (no value) and two operations meet and join.
  - Meet b/w two lattice elements computes their greatest upper bound while join computes their lowest upper bound.

Lattice – Data flow

- Lattice is generally represented using Hasses Diagrams.
- Data Flow Analysis Algorithm

```
Initialize_lattice_corresponding_to_each_definition
while (no_change) {
    no_change |=  dfs_walk : Apply flow function over lattice
associated with inputs of each IR node.
}
// Fixed point reached; data flow analysis converged.
```



- Conditions for convergence
  - Lattice values must modify monotonically.
  - Graph should be reducible.

C2 Compiler IR

- As we discussed, traditional SSA based IR maintains clear differentiation b/w control and data flow graphs with well defined boundaries around set of instruction in the form of basic block.
- Hotspot C2 compiler and other popular runtime compilers like Graal and v8's TurboFan chose a "Sea of Nodes"[1] intermediate representation.
- It was invented by Cliff Click.

[1] https://en.wikipedia.org/wiki/Sea_of_nodes

Sea Of Nodes.

- Fundamentally its an SSA IR.
- Graph is composed of Nodes and edges.
- Node has both control and data edges.
- There is no fixed notion of basic block in SoN IR, while there are bounding box nodes like RegionNode and LoopNodes which represent a control flow node.
- RegionNode inputs are connected to control projections of IfNode , GotoNode or JumpNode.
- A graph has a unique entry node and a Return node.

SoN Graph.

- Control flow walk comprises of traversing the control edge (idx=0) of connected nodes.
- Unless needed IR nodes corresponding to an arithmetic operation has only data edges. Thus, a node is not tied to any specific basic block during optimizations, this differentiates it from tradition IR where at any given point an IR is always fixed withing a basic block.
- This also increases the window of optimization for passes like Global Value numbering which can freely share the IR nodes as long as its not tied to any specific control edge and it’s the USP of the SoN IR graph.

SoN Graph

- Since most of the nodes are floating i.e. without any controlling edge it gives the compiler a free hand to schedule them appropriately, maximizing sharing and minimize computation redundancy while still preserving program semantics.
- C2 Compilation pipeline

Bytecode => [Parsing] => SoN Graph => [Optimizations] => [Matching / BURS style instruction selector] => Mach IR => [Scheduling / PhaseCFG] => [Register Allocation] => [Peephole Optimization] => [Code Gen]

Q: Why is scheduling needed?

A. While SoN IR has floating nodes, the compiler must serialize the instruction sequence to comply with the program semantics, so after matching, the compiler creates a regular CFG with basic blocks and pins various floating point nodes to them. While it's easy to tie the IR with control edges, the tricky part is to assign a basic block to a floating node.

Q. What are the different stages in scheduling?

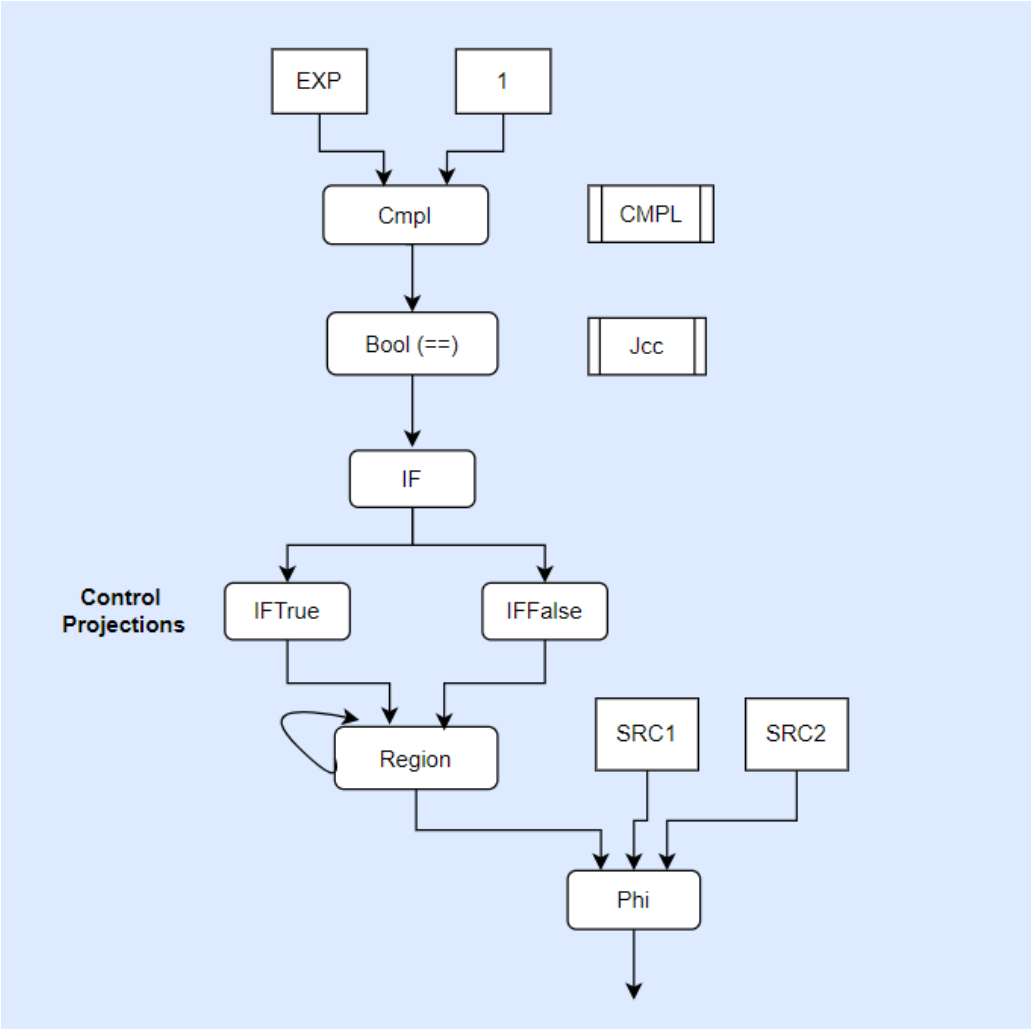
A. Eager – A node is scheduled into an earliest possible basic block of its controlling edge, all the free floating nodes are assigned to starting block. Memory IR has explicit control edges.

Late – A node is placed in the lowest common block of its user nodes. This helps in reducing the live range of the IR for efficient Register allocation.

IR Graph for IF condition.

```
if (expr == 1) {
    dst = src1;
} else {
    dst = src2;
}
```

Entry Block:-
  Terminal IR : IF <TRUE> TPATH <FALSE> FPATH
TPATH:
  dst1 = src1
  JMP END
FPATH:
  dst2 = src2
  JMP END
END:
  dst3 = PHI TPATH:DST1 FPATH:DST2



Options to dump C2 IR graph.

- -XX:CompileCommand=PrintIdealPhase,<method_name>,<PhaseName / ALL>
- -XX:CompileCommand=help list various compilation pragmas.
- -XX:+WizardMode –XX:CompileCommand=Print,<method>
- C2- Parser entry points
  - Parse::do_all_blocks
  - Parse::do_one_block
  - Parse::do_one_bytecode
- Parser uses information from a ci (compilation interface) object model rather than directly accessing the bytecodes.

APX – Condition Compare

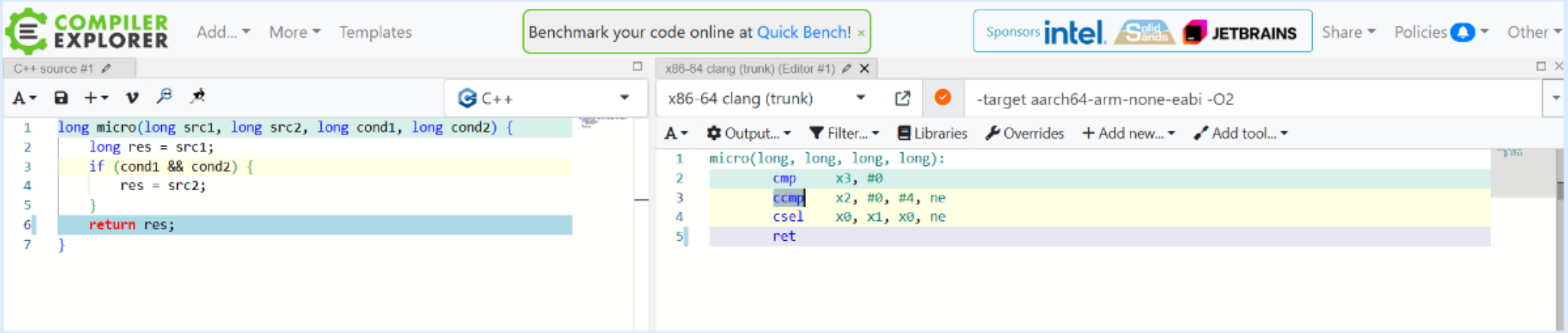
Source:-

```
res = src1;
if (c1 == 1 && c2 == 2) {
  res = src2;
}
return res;
```

```
// CCMP

IF (src_flags satisfies scc):
    dst_flags = compare(src1,src2)
ELSE:
    dst_flags = flags(evex.[of,sf,zf,cf])
```

Figure 3.8: Pseudocode for CCMP

For conditional ISA our baseline should be AARCH



C2 if branch profiling spoils the party.

```
public static int micro(int a, int b, int c, int d) {
    int res = 0;
    // CmpI => Bool => If => IfTrue / IfFalse => Region => Phi
    //     --->
    // CMoveI
    // apx : if (a == b && c == d) {
    if (a == b && c == d) {
       res = c;
    }
    return res + d;
}

public static void main(String [] args) {
   int res = 0;
   int mask = Integer.parseInt(args[0]);
   for (int i = 0; i < 98000000; i++) {
       res += micro(i, i & mask, i , i & mask);
   }
   long t1 = System.currentTimeMillis();
   for (int i = 0; i < 10000000; i++) {
       res += micro(i, i & mask, i , i & mask);
   }
   long t2 = System.currentTimeMillis();
   System.out.println("[time] "  + (t2-t1) + "ms [res] " + res);
}
```

```
This is evident when mask is 1023, we see two compilation attempts, first time compiler injects an UCT
while on second attempt it compiles both the control paths..

C2 compiler takes branch frequency into account to detect the infrequently taken control paths and
injects Uncommon Traps.

With mask value set to 2047, taken branch frequency is substantial enough to be ignored and hence both the control
paths are compiled on first compilation attempt.However, Conditional moves are not inferred despite of a diamond
control flow.

26  CmpI  === _ 10 11  [[ 27 28 ]]  !jvms: infer_cond_moves::micro @ bci:5 (line 6)
27  Bool  === _ 26  [[ 28 ]] [ne] !jvms: infer_cond_moves::micro @ bci:5 (line 6)
28  If  === 5 27 26  [[ 29 30 ]] P=0.924823, C=6784.000000 !jvms: infer_cond_moves::micro @ bci:5 (line 6)
29  IfTrue  === 28  [[ 33 ]] #1 !jvms: infer_cond_moves::micro @ bci:5 (line 6)
30  IfFalse  === 28  [[ 33 ]] #0 !jvms: infer_cond_moves::micro @ bci:5 (line 6)
33  Region  === 33 30 29  [[ 33 36 37 ]] #reducible  !jvms: infer_cond_moves::micro @ bci:11 (line 9)
36  Return  === 33 6 7 8 9 returns 37  [[ 0 ]]
37  Phi  === 33 38 13  [[ 36 ]]  #int !orig=[35] !jvms: infer_cond_moves::micro @ bci:14 (line 9)

With mask value of 4095, not only both the control flows are compiled, but also conditional moves are
inferred as expected.

26  CmpI  === _ 10 11  [[ 27 38 ]]  !jvms: infer_cond_moves::micro @ bci:5 (line 6)
27  Bool  === _ 26  [[ 38 ]] [ne] !jvms: infer_cond_moves::micro @ bci:5 (line 6)
35  AddI  === _ 13 37  [[ 36 ]]  !jvms: infer_cond_moves::micro @ bci:14 (line 9)
36  Return  === 5 6 7 8 9 returns 35  [[ 0 ]]
37  CMoveI  === _ 38 39  [[ 35 ]]  #int !orig=[34] !jvms: infer_cond_moves::micro @ bci:11 (line 9)
38  Binary  === _ 27 26  [[ 37 ]]
39  Binary  === _ 12 25  [[ 37 ]]

Two questions :-
Q1. Why do we not infer CMove with mask value 2047.
A.  Lowering the BlockLayoutMinDiamondPercentage triggers CMoveI inferencing at lower mask value.

Q2. Root cause 2x latency improvement with CMove.
A.  With lager warmup iteration and by disableing DVFS turbo scalaing fixed
    the latency gap b/w the two cases where mask was set to 2047 and 4095. Also the latency
    delta is now within +/-%5 range for differnt mask values.

Action Item :-
 - Prepare a micro for APX usecase with primary and secondary conditional expressions guarding
   the diamond shaped control structure.
```

Impact of Conditional Compare on Branch Prediction machinery

- Branch Predictor types:-

- Adaptive predictor with global and local history table.

- TAGE prediction (used in Intel processors)

```
- Pros
    - Very useful to reduce the number of branches and thus saving BTB entries.
    - Even though compiler driven short circuiting saves exeucting redundant follow up comparisons
      if does generate addtional condition logic e.g.

 source:
   if ( cond1 && cond2) {
        true_path
   }
   follow_on_path

 compiler:
     -- First level IR.
 IR:
     T1 = ICMP cond1
     BR  i1 T1 TRUE_LABEL1 , FALSE_LABEL1

 TRUE_LABEL1:
     T2 = ICMP cond2
     BR i1 T2  TRUE_LABEL2, FALSE_LABEL2

 FALSE_LABEL1:
     BR FOLLOW_ON_PATH

 FALSE_LABEL2:
     BR FOLLOW_ON_PATH

 TRUE_LABEL2:
     true_path_block
     BR FOLLOW_ON_PATH

 FOLLOW_ON_PATH:
     follow_on_path_block


     -- Branch optimization. Dead blocks with unconditional block removed and its target propagated back to its predecessor
        terminal IR.

 IR:
     T1 = ICMP cond1
     BR  i1 T1 TRUE_LABEL1 , FOLLOW_ON_PATH

 TRUE_LABEL1:
     T2 = ICMP cond2
     BR i1 T2  TRUE_LABEL2, FOLLOW_ON_PATH

 TRUE_LABEL2:
     true_path_block
     BR FOLLOW_ON_PATH

 FOLLOW_ON_PATH:
     follow_on_path_block

 Still above code sequence has 3 branching (BR) IRs. Which could potential occupy space in BTB.


 With conditional compare and test we can form a macro level IR which can absorb two conditional expressions
 there by saving entries in generally space constrained branch prediction machinery and help in improving
 misprediction ratio in real server workloads.
```

Adaptive Branch Prediction with n-bit histroy

```
// Two level 2bit adaptive predictor.
// Local patten table per branch 2^2 = 4 entries
// 2 bit saturating counter per entry.
// Branch history register size : 2 bit
// Local patten table
//  0 0     : counter 2 bit ( 00 01 10 11) -> alternates between strongly not taken (00) to strongly taken (11)
//  0 1     :
//  1 0     :
//  1 1     :
// For following branches
// with alternating branch taken pattern 010101...
// Branch history register will always hold 01 10 values.
// Thus only two rows of pattern table will be used, staurating counter
// in row corresponding to 01 entry will have 00 value
// while counter corresponding to 10 will have 11 value after
// some learning curve
// Thus after some time it will start make 100% accurate predictions.
```

TAgged GEometric history length predictor

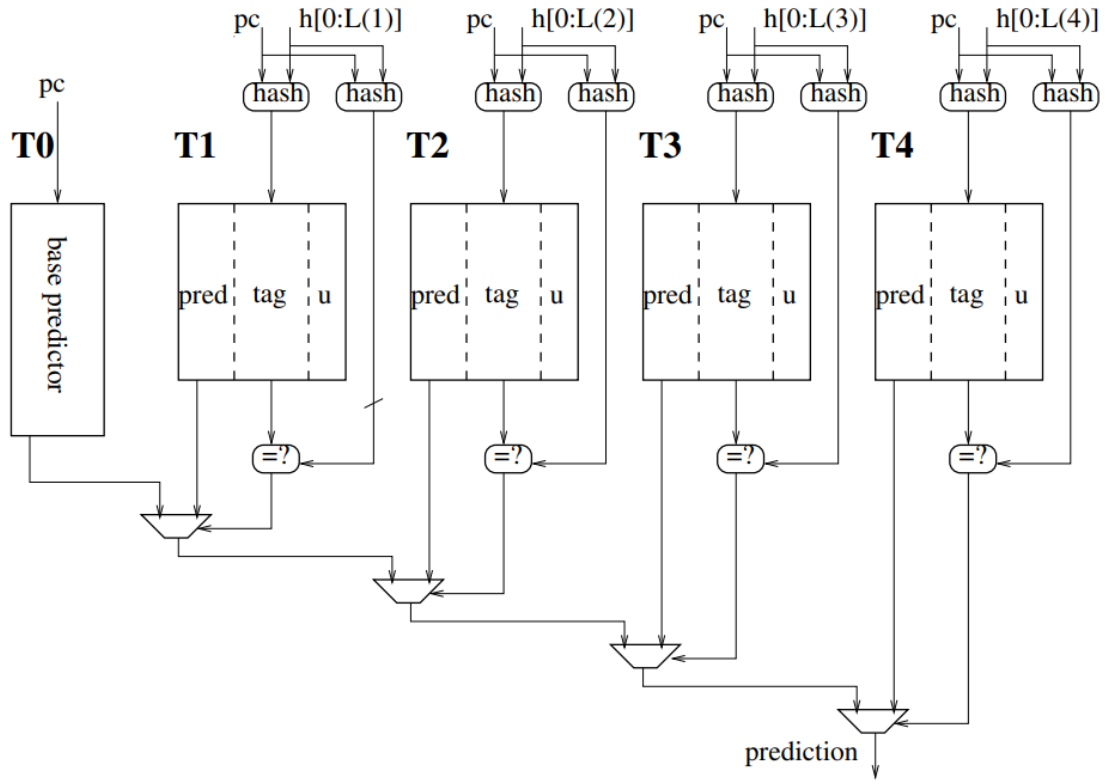


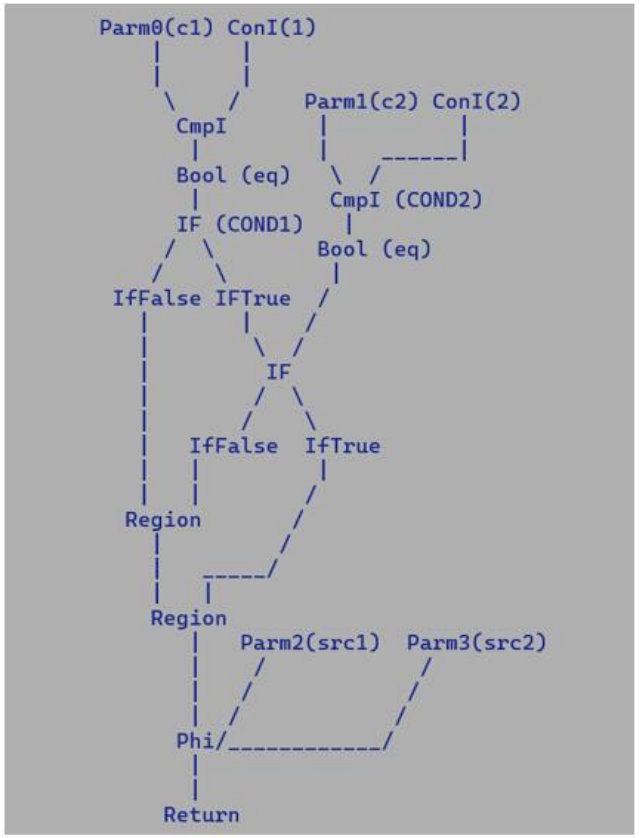
Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

TAGE prediction = function (Base Predictor[T0], Partially Tagged prediction components (T1-TM)]

SoN Graph

```
 33  CmpL  === _ 14 29  [[ 34 35 ]]  !jvms: conditional_compare::micro2 @ bci:9 (line 16)
 34  Bool  === _ 33  [[ 35 ]] [le] !jvms: conditional_compare::micro2 @ bci:9 (line 16)
 35  If  === 5 34 33  [[ 36 37 ]] P=0.651975, C=6784.000000 !jvms: conditional_compare::micro2 @ bci:9 (line 16)
 36  IfTrue  === 35  [[ 45 ]] #1 !jvms: conditional_compare::micro2 @ bci:9 (line 16)
 37  IfFalse  === 35  [[ 40 ]] #0 !jvms: conditional_compare::micro2 @ bci:9 (line 16)
 40  Region  === 40 46 37  [[ 40 50 ]] #reducible  !jvms: conditional_compare::micro2 @ bci:21 (line 17)
 43  CmpL  === _ 16 29  [[ 44 45 ]]  !jvms: conditional_compare::micro2 @ bci:18 (line 16)
 44  Bool  === _ 43  [[ 45 ]] [lt] !jvms: conditional_compare::micro2 @ bci:18 (line 16)
 45  If  === 36 44 43  [[ 46 47 ]] P=0.999774, C=4423.000000 !jvms: conditional_compare::micro2 @ bci:18 (line 16)
 46  IfTrue  === 45  [[ 40 ]] #1 !jvms: conditional_compare::micro2 @ bci:18 (line 16)
 47  IfFalse  === 45  [[ 50 ]] #0 !jvms: conditional_compare::micro2 @ bci:18 (line 16)
 50  Region  === 50 40 47  [[ 50 52 51 ]] #reducible  !jvms: conditional_compare::micro2 @ bci:24 (line 19)
 51  Phi  === 50 12 10  [[ 52 ]]  #long !jvms: conditional_compare::micro2 @ bci:24 (line 19)
 52  Return  === 50 6 7 8 9 returns 51  [[ 0 ]]
```

C2 SoN IR:-



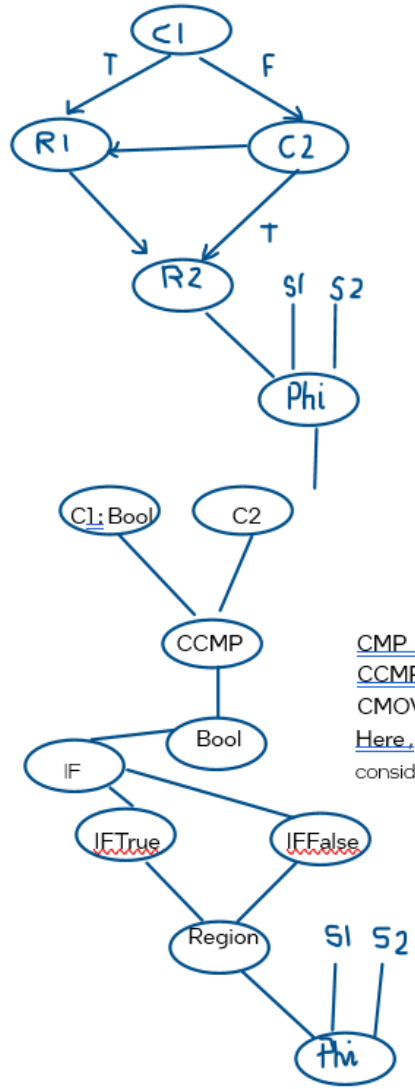
CCMP – Inferencing

Attributes per CCmpNode

DVF = EFLAGS setting when SRC.FLAGS does not comply with EVEX.SFLAGS
EVEX.SFLAGS = FLAG settings to be matched against incoming flags (SRC.FLAGS)

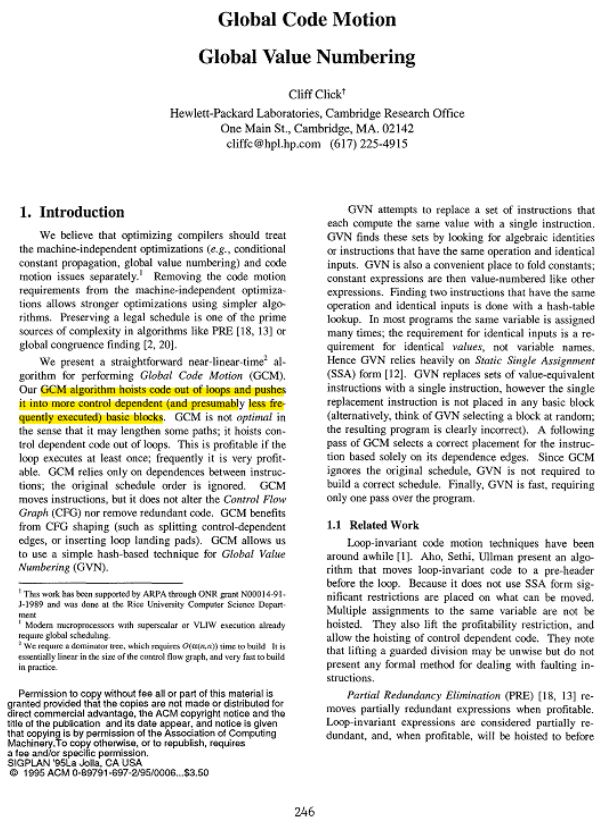
The entire graph skeleton must match.

When C2 == C1.FALSE
    EVEX.SFLAGS = NOT C1. PRED
    EVEX.DFV = C2.PRED.
ELSE
    Assert (C2 == C1.TRUE)
    EVEX.SFLAGS = C1.PRED
    EVEX.DFV = C2.PRED



CMP C1 -> Bool (NE)
CCMPNE C2 -> Bool (E)
CMOVEGT
Here , EVEX.SFLAGS = NE/E and EVEX.DFV = E
considering control flow positioning of C2 w.r.t C1.

References:



Global Code Motion

Global Value Numbering

Cliff Click[†]
Hewlett-Packard Laboratories, Cambridge Research Office
One Main St., Cambridge, MA. 02142
cliffc@hpl.hp.com (617) 225-4915

1. Introduction

We believe that optimizing compilers should treat the machine-independent optimizations (*e.g.*, conditional constant propagation, global value numbering) and code motion issues separately.[1] Removing the code motion requirements from the machine-independent optimizations allows stronger optimizations using simpler algorithms. Preserving a legal schedule is one of the prime sources of complexity in algorithms like PRE [18, 13] or global congruence finding [2, 20].

We present a straightforward near-linear-time[2] algorithm for performing *Global Code Motion* (GCM). Our GCM algorithm hoists code out of loops and pushes it into more control dependent (and presumably less frequently executed) basic blocks. GCM is not *optimal* in the sense that it may lengthen some paths; it hoists control dependent code out of loops. This is profitable if the loop executes at least once; frequently it is very profitable. GCM relies only on dependences between instructions; the original schedule order is ignored. GCM moves instructions, but it does not alter the *Control Flow Graph* (CFG) nor remove redundant code. GCM benefits from CFG shaping (such as splitting control-dependent edges, or inserting loop landing pads). GCM allows us to use a simple hash-based technique for *Global Value Numbering* (GVN).

GVN attempts to replace a set of instructions that each compute the same value with a single instruction. GVN finds these sets by looking for algebraic identities or instructions that have the same operation and identical inputs. GVN is also a convenient place to fold constants; constant expressions are then value-numbered like other expressions. Finding two instructions that have the same operation and identical inputs is done with a hash-table lookup. In most programs the same variable is assigned many times; the requirement for identical inputs is a requirement for identical *values*, not variable names. Hence GVN relies heavily on *Static Single Assignment* (SSA) form [12]. GVN replaces sets of value-equivalent instructions with a single instruction, however the single replacement instruction is not placed in any basic block (alternatively, think of GVN selecting a block at random; the resulting program is clearly incorrect). A following pass of GCM selects a correct placement for the instruction based solely on its dependence edges. Since GCM ignores the original schedule, GVN is not required to build a correct schedule. Finally, GVN is fast, requiring only one pass over the program.

1.1 Related Work

Loop-invariant code motion techniques have been around awhile [1]. Aho, Sethi, Ullman present an algorithm that moves loop-invariant code to a pre-header before the loop. Because it does not use SSA form significant restrictions are placed on what can be moved. Multiple assignments to the same variable are not be hoisted. They also lift the profitability restriction, and allow the hoisting of control dependent code. They note that lifting a guarded division may be unwise but do not present any formal method for dealing with faulting instructions.

Partial Redundancy Elimination (PRE) [18, 13] removes partially redundant expressions when profitable. Loop-invariant expressions are considered partially redundant, and, when profitable, will be hoisted to before

[†] This work has been supported by ARPA through ONR grant N00014-91-J-1989 and was done at the Rice University Computer Science Department.

[1] Modern microprocessors with superscalar or VLIW execution already require global scheduling.

[2] We require a dominator tree, which requires $O(\alpha(n,n))$ time to build. It is essentially linear in the size of the control flow graph, and very fast to build in practice.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1995 ACM 0-89791-697-2/95/0006...$3.50

SIGPLAN '95La Jolla, CA USA

Global Code Motion Global Value Numbering
https://courses.cs.washington.edu/courses/cse501/06wi/reading/click-pldi95.pdf

Semantic reasoning about the sea of nodes
https://inria.hal.science/hal-01723236/file/sea-of-nodes-hal.pdf