July 22, 2003

RT0540
Computer Science 6 pages

Research Report

Translating Out of Predicated Static Single Assignment Form

Kazuaki Ishizaki and Tatsushi Inagaki

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Translating out of Predicated Static Single Assignment Form

Kazuaki Ishizaki and Tatsushi Inagaki

IBM Research, Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan
ishizaki@trl.ibm.com

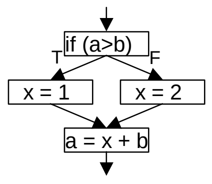
ABSTRACT
Static Single Assignment (SSA) form is an intermediate representation that allows a compiler to perform advanced optimizations to extract parallelism. Predication is an architectural feature to maximize instruction level parallelism. If a compiler uses a predicated SSA form that is a combination of an SSA form and predication as an intermediate representation, it can perform advance optimizations that are otherwise difficult to apply in order to aggressively extract a parallelism from a program. Since the code in SSA form cannot be executed directly on the existing target architecture, a translation out of the predicated SSA form must be performed, and this translation is not trivial. We propose an algorithm to translate out of predicated SSA form.

1. INTRODUCTION
Static Single Assignment (SSA) form [1] is an intermediate representation that allows a compiler to perform advanced optimizations. In the SSA form, each variable name is defined exactly once and each use refers to exactly one variable name. Thus, phi (ϕ) instructions are inserted at join points to merge multiple values into a single variable name. As a result, the compiler can aggressively extract parallelism from a program since there are only true dependencies and no anti-dependencies or output dependencies in this form. Since the phi instruction is not supported on the target architecture, all of phi instructions must be eliminated. However, translating out of SSA form is nontrivial, and thus a lot of works [1, 2, 3, 4, 5] has been done on this problem.

Recently, the *explicitly parallel instruction computing* (EPIC) architectures such as IA-64 [6] have became available. EPIC has a feature, called *predication* [7], to eliminate conditional branches and allow concurrent execution of instructions from multiple paths across basic blocks (BBs). In the predicated representation, each instruction has a Boolean source operand as its guarding predicate. By introducing predicated instructions, an acyclic control flow region can be converted to a single branch-free block with a single entry and multiple exits, called a *hyperblock* (HB) [8]. Since it translates a control flow graph into a dataflow graph, a program can be optimized in a hyperblock simply by handling the data dependencies. As a result, it is possible to perform some advanced optimizations to extract parallelism from a program.

Both the SSA form and the EPIC architecture are intended to extract a parallelism from a program. Research about combining SSA and EPIC has been published [9]. The combination is called *predicated SSA*. An example of predicated SSA form is shown in Figure 1. It introduced predicated static single assignment and how to build predicated SSA form. However, this approach could not translate out of predicated SSA form if any ϕ-instructions are still remaining. Since translating out of SSA in non-predicated code is non-trivial, predicated SSA poses challenges in translating out of it.

This paper show an algorithm for translating out of predicated SSA form that allows a compiler to perform advanced optimizations to aggressively extract parallelism.



Control flow graph

```
if (a1 > b1) {
   x1 = 1
else {
   x2 = 2
}
x3 = ϕ(x1, x2)
a2 = x3 + b1
```

SSA form

```
(p0) cmp.unc p1,p2 = (a1 > b1)
(p1) x1 = 1
(p2) x2 = 2
(p0) x3 = ϕ(x1, x2)
(p0) a2 = x3 + b1
```

Predicated SSA form

Figure 1: An example of predicated SSA

2. ALGORITHM
This section describes an algorithm for translating out of predicated SSA form. First, we describe a traditional algorithm for translating out of the SSA form. Next, we present our algorithm for translating out of predicated SSA form.

2.1 Traditional algorithm
This subsection describes traditional algorithm for translating out of SSA form.

Translating out of SSA form requires that local variable names in source operands and destination operands of a ϕ-function be renamed using the same name. In addition, anti-dependencies and output dependencies links among operands have been built if required. As described in Section 1, several algorithms are proposed. In this paper, we use an algorithm that consists of the following steps. Our algorithm only takes care of the lost copy problem [2]. The swap problem [2] is beyond the scope of this paper.

1. Rename the variable names of source operands and destination operands in *phi congruence classes* (PCCs) [3] with the same name. A phi congruence class is defined as a maximal union of true dependence chains that consist of the following two types of links. One type of link is one from the definition operand in a ϕ or non-ϕ instruction to the use operand in another ϕ instruction. The other type of the link is one from the definition operand in a ϕ or non-ϕ instruction to the use operand of another non-phi instruction, but this use operand of the non-phi instruction terminates a PCC. A PCC must include one or more ϕ instructions.

2. For each PCC, arrange the destinations of the PCC in the reverse post order considering control flow dependence, data dependencies, and other dependencies.

3. Traverse the list of destinations in forward order. For each destination, search for a dominating definition considering the control flow dependencies and the execution order in its BB. Repeat Steps 4, 5, and 6 for a dominating definition for each destination.

4. If one of the following two conditions is satisfied, a copy instruction is inserted after the dominating definition to break the cycle when S1 can be reached from S2 along the data dependency edges in Figure 2:

    1) A given definition and a use of the dominating definition exist in the same BB and the instruction for the use can be reached from the instruction for the definition.

    2) A given definition and a use of the dominating definition exist in a different BB, the use does not belong to a ϕ instruction, and the BB where the use exists can be reached from the BB where the definition exists along control flow dependences.

    An edge is added for each true dependency between a destination of the copy instruction and the use of the dominating definition. In addition, if the given destination and a use of the copy instruction exist in the same BB, an edge is added for the anti-dependency between them. In Figure 2, a copy instruction is inserted after S1, and the destination of the copy instruction is linked to the use of S3.

```
S1:  LI1 = ϕ(LI0, LI2)
     ...
S2:  LI2 = LI1 + 1
     ...
S3:  ... = LI1
```

Figure 2: An example of making a cyclic

5. If a given definition and a use of the dominating definition exist in the same BB, an edge is added for the anti-dependency between them.

6. If the dominating destination and a given destination exist in the same BB, an edge is added for the output dependency between them.

We show the detailed algorithm in Figure 3.

```
Add_PhiConguenceClass(inst, PCC, DAG[], CFG)
{
  inst                 : in        instruction;
  PCC                  : in, out   a set of sets of instructions;
  DAG[]                : in        Directed acyclic graph × # of BBs in CFG;
  src, def             : an operand;

  if (inst is visited) return;
  add inst to PCC;

  for (each use ∈ succ(defining_operand(inst))) {
    if (instr(use) ≠ ϕ) continue;
    Add_PhiConguenceClass(instr(use), PCC, DAG[]);
  }

  if (inst = ϕ) {
    for (each src ∈ source_operands(inst)) {
      for (each def ∈ pred(src)) {
        Add_PhiCConguenceClass(instr(def), PCC, DAG[]);
      }
    }
  }
}

Build_PhiConguenceClass(PCC, DAG[], CFG)
{
  PCC                  : in, out   a set of sets of instructions;
  CFG                  : in        Control Flow Graph;
  DAG[]                : in        Directed acyclic graph × # of BBs in CFG;
  bb                   : a basic block;
  inst                 : an instruction;
  insts                : a set of instructions

  for (each BB bb ∈ CFG by reverse post order) {
    for (each instruction inst ∈ DAG[bb] by forward order) {
      if (inst ≠ ϕ) continue;
      insts = {∅};
      Add_PhiConguenceClass(inst, insts, DAG[]);
      Add insts to PCC;
    }
  }

  /* sort PCC by order of definitions */
  sort PCC by PCC_compare(inst1, inst2, DAG[], CFG);
}
```

```
  PCC_compare(inst1, inst2, DAG[], CFG)
  {
    inst1, inst2                : in     an instruction;
    CFG                         : in     Control Flow Graph;
    DAG[]                       : in     Directed acyclic graph × # of BBs in CFG;

    if (BB(inst1) is prior to BB(inst2) in reverse post order) place inst1 prior to inst2 in PCC & return;
    if (BB(inst2) is prior to BB(inst1) in reverse post order) place inst2 prior to inst1 in PCC & return;

    if (inst1 is reachable to inst2) place inst1 prior to inst2 in PCC & return;
    if (inst2 is reachable to inst1) place inst2 prior to inst1 in PCC & return;
  }

TranslateOutOfSSA(CFG, DAG[])
{
    CFG                         : in       Control Flow Graph;
    DAG[]                       : in, out  Directed acyclic graph × # of BBs in CFG;
    PCC                         : a set of sets of instructions;
    insts                       : a set of instructions;
    inst, inst_use              : instruction;
    def_BB                      : BB;
    latest_def[]                : an array of BBs × # of BBs in CFG
    def, use, dominating_def    : an operand;

    // build PhiConguenceClass
    Build_PhiConguenceClass(PCC, DAG[], CFG);

    // assign new variable name to each set in PhiConguenceClass
    for (each set insts ∈ PCC) { Rename all variable names with a new variable name in insts; }

    // maintain anti dependence and output dependence
    for (each set insts ∈ PCC) {
      lastest_def[] = {∅};
      for (each instruction inst ∈ insts) {
        // find dominating definition
        def = defining_operand(inst);
        def_BB = BB(inst);
        while (def_BB is not entry BB) {
          dominating_def = latest_def[def_BB];
          if (dominating_def) break;
          def_BB = parent_in_dominator_tree(def_BB);
        } /* end while */
        latest_def[BB(def)] = def;
        if (dominating_def = ∅) continue;

        // examine each successor of the dominating defintion
        for (each use = succ(dominating_def)) {
          inst_use = instr(use);
          if (((BB(inst) = BB(inst_use)) && (inst is reachable to inst_use)) ||
              ((BB(inst) ≠ BB(inst_use)) && (inst_use ≠ ϕ) && (BB(inst) is reachable to BB(inst_use)))) {
            insert copy instruction after dominating_def;
            migrate links from use to source_operand(copy instruction);
            assign new name to defining_operand(copy instruction);
            build a true dependence edge from dominating_def to use;
            use = source_operand(copy instruction);
          }

          if (BB(inst_use) = BB(inst)) {
            build an anti dependence edge from use to def;
          }
        } /* end for use */

        if (BB(instr(dominating_def)) = BB(inst)) {
          build an output dependence edge from dominating_def to def;
        }
      } /* end for inst */
    } /* end for insts */
}
```

Figure 3: The algorithm for translating out of SSA form

2.2 Our algorithm

This subsection describes our algorithm for translating out of predicated SSA form.

In predicated code, HBs are formed that each includes one entry and multiple exits. Since each HB includes multiple paths, there are additional relationships that must be considered, such as domination and reachability.

First, we define relationships in a predicate hierarchy graph (PHG) [8]. Next, we show an algorithm for translating out of the predicated SSA form using the function.

2.2.1 Predicate Hierarchy Graph

We introduce a PHG [10], which is a directed acyclic graph of Boolean equations, where each equation describes some predications in the current HB. The PHG consists of predicate register nodes, condition nodes, and directed edges. The 'p0' predicate node is used to represent instructions that are always executed as the top node in the graph. The PHG is constructed by adding nodes while traversing the code from the top to the bottom. During the traversal, when a compare instruction is found, a condition node is added and an edge is created from the predicate node that has a guarding predicate register of the instruction. Next, two predicate nodes for the destination predicate registers are added if they do not yet exist, and two edges are created from the condition node. When an assignment instruction to the predicate register is found, an edge is created between the predicate nodes from the source predicate to the destination predicate. Finally, we add an exit node at the bottom of the graph and create the edges from every leaf predicate node to the exit node. Figure 4 shows an example of a PHG.

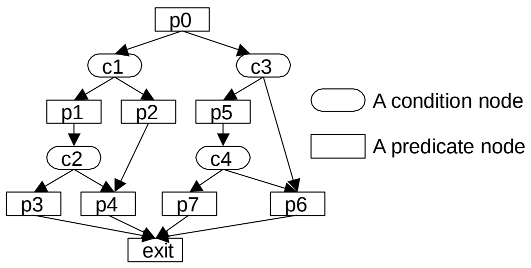


Figure 4: An example of a PHG

In the PHG, we define three properties. We say that a predicate register *p* dominates a predicate register *q* if the node *p* dominates the node *q* in the PHG. We say that a predicate register *p post-dominates* a predicate register *q* if the node *p post-dominates* the node *q* in the PHG. We say that the relationship between the predicate registers *p* and *q* is *exclusive* if there is no path between *p* and *q* in the PHG.

2.2.2 Our Algorithm

Translating out of the predicated SSA form is derived from the algorithm described in Section 2.1 as follows.

1. Rename variable names of source operands and destination operands in the PCC with the same name.

2. For each PCC, arrange the destinations of the PCC in the reverse post order considering control flow dependencies, data dependencies both for variables and predicate registers, predicate relationships, and other dependences.

3. Traverse the list of destinations in forward order. For each destination, search for a dominating definition considering control flow dependencies, predicate relationships, and the execution within an HB. There is usually only one dominating definition in an HB. If a given definition is that of a ϕ instruction (S4) and a definition (S2 or S3) reaches to a use of the ϕ instruction as in Figure 5, we handle more than one dominating definition that exists in the HB. Repeat Step 4, 5, and 6 for one or more dominating definitions for each destination.

```
S1:  (P0)   P1, P2 =       //p1 and p2 are complementary
     ...
S2:  (P1)   LI1 =
     ...
S3:  (P2)   LI2 =
     ...
S4:  (P0)   LI3 = ϕ(LI1, LI2)
```

We recognize that S2 and S3 are dominating definitions of S4.

Figure 5: An example of two dominating definitions in a HB

4. If either of the following two conditions is satisfied, a copy instruction with a predicate register of the domination definition is inserted after the dominating definition:

   1) A given definition and a use of the dominating definition exist in the same HB, the use does not belong to a ϕ instruction, the instruction of the use can be reached from the instruction of the definition, and there is not an exclusive relationship between the predicate registers of the definition and registers for the use.

   2) A given definition and a use of the dominating definition exist in a different HB, the use does not belong to a ϕ instruction, and the HB of the use can be reached from the HB of the definition.

   An edge is added for each true dependency between a destination of the copy instruction and the use of the dominating definition. In addition, if the given destination and the use of the copy instruction exist in the same HB, an edge is added for anti-dependency between them.

5. If the given destination and the use of the dominating definition exist in the same HB, and the use does not belong to a ϕ instruction, then an edge is added for the anti-dependency between them.

6. If the dominating destination and a given destination exist in the same HB, an edge is added for the output dependency between them.

```
PCC_compare(inst1, inst2, DAG[], CFG)
{
    inst1, inst2                 : in     an instruction;
    CFG                          : in     Control Flow Graph;
    DAG[]                        : in     Directed acyclic graph × # of HBs in CFG;

    if (HB(inst1) is prior to HB(inst2) in reverse post order) place inst1 prior to inst2 in PCC & return;
    if (HB(inst2) is prior to HB(inst1) in reverse post order) place inst2 prior to inst1 in PCC & return;

    if (inst1 is reachable to inst2) place inst1 prior to inst2 in PCC & return;
    if (inst2 is reachable to inst1) place inst2 prior to inst1 in PCC & return;

    // predic(inst) returns a guarding predicate register of inst.
    if (predic(inst1) dom predic(inst2)) place inst1 prior to inst2 in PCC & return;
    if (predic(inst2) dom predic(inst1)) place inst2 prior to inst1 in PCC & return;

    if (inst1 is prior to inst2) place inst1 prior to inst2 in PCC & return;
    if (inst2 is prior to inst1) place inst2 prior to inst1 in PCC & return;
}

TranslateOutOfPSSA(CFG, DAG[])
{
    CFG                          : in         Control Flow Graph;
    DAG[]                        : in, out    Directed acyclic graph × # of HBs in CFG;
    PCC                          : a set of sets of instructions;
    insts                        : a set of instructions;
    inst, inst_use               : instruction;
    def_HB                       : HB;
    def, use, dominating_def     : an operand;
    maynot_dominate              : boolean;
    i, n                         : integer;

    // build PhiConguenceClass
    Build_PhiConguenceClass(PCC, DAG[], CFG);

    // assign new variable name to each set in PhiConguenceClass
    for (each set insts ∈ PCC) { Rename all variable names with a new variable name; }

    // maintain anti dependence and output dependence
    for (each set insts ∈ PCC) {
      for (each instruction inst ∈ insts) {
        // find dominating definition
        def = defining_operand(inst);
        maynot_dominate = TRUE;
        n = index of inst in insts;
        while (maynot_dominate) {
          maynot_dominate = FALSE;
          dominating_def = {∅};
          def_HB = HB(inst);
          while (def_HB is not root) {
            for (idx = n - 1; idx >= 0; idx--) {
              pcc_def_inst = insts[idx];
              if (HB(pcc_def_inst) != def_HB) continue;
              pred_pcc_def = (HB(pcc_def_inst) = HB(inst)) ? pred(pcc_def_inst) : P0;
              if (pred_pcc_def dom predic(inst)) { dominating_def = defining_operand(pcc_def_inst); break; }
              if ((inst = ϕ) && (predic(inst) pdom pred_pcc_def)) {
                dominating_def = defining_operand(pcc_def_inst); maynot_dominate = TRUE; n = idx; break;
              }
              def_HB = parent_in_dominator_tree(def_HB);
            } /* end for */
            if (dominating_def ≠ ∅) break;
          } /* end while */
          if (dominating_def = ∅) continue;

          // examine each successor of the dominating defintion
          for (each use = succ(dominating_def)) {
            inst_use = instr(use);
            if (((HB(inst) = HB(inst_use)) && (inst_use ≠ ϕ) && (pred(inst) is reachable to pred(inst_use))) ||
                ((HB(inst) ≠ HB(inst_use)) && (inst_use ≠ ϕ) && (HB(inst) is reachable to HB(inst_use))) {
              insert copy instruction after dominating def with a predicate register of dominating def;
              migrate links from use to source_operand(copy instruction);
              assign new name to defining_operand(copy instruction);
              build an true dependence edge from dominating_def to use;
              use = source_operand(copy instruction);
            }

            if ((HB(inst_use) = HB(inst)) && (inst_use ≠ ϕ) && (pred(inst_use) is reachable to pred(inst))) {
              build an anti dependence edge from use to def;
            }
          } /* end for use */

          if (HB(instr(dominating_def)) = HB(inst)) {
            build an output dependence edge from dominating_def to def;
          }
        } /* end while */
      } /* end for inst */
    } /* end for insts */
}
```

Figure 6: An algorithm for translating out of predicated SSA form

3. CONCLUSION

We described an algorithm for translating out of predicated SSA form. It allows a compiler to perform advanced optimizations on the predicated SSA intermediate representation, such as height reduction [11, 12], which that are otherwise difficult to apply. As a result, more parallelism can be extracted from a program.

REFERENCES

[1] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficient computing static single assignment form and the control dependence graph. ACM Transaction on Programming Languages and Systems, 13(4), pp.451-490, 1991.

[2] P. Briggs, K. Cooper, T. Harvey, and T. Simpson. Practical improvements to the construction and destruction of static single assignment form, Software – Practice and Experience, 28(8), pp.859-881, 1998.

[3] V. C. Sreedhar, R. D. Ju, D. M. Gillies, and V. Santhanam. Translating Out of Static Single Assignment Form. In Proceedings of Static Analysis Symposium, LNCS 1694, pp. 194-210, Springer Verlag, 1999.

[4] A. Leung and L. George. Static Single Assignment Form for Machine Code, In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.204-214, 1999.

[5] Z. Budimlic, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, S. W. Reeves. Fast copy coalescing and live-range identification, In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.25-32, 2002.

[6] Intel Corp., IA-64 Application Developer's Architecture Guide, http://developer.intel.com/design/ia64/downloads/adag.htm.

[7] J. C. Park and M. S. Schlansker. On predicated execution, Hewlett-Packard Laboratories Technical Report, HPL-91-58, 1991.

[8] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock, In *Proceedings of the 25th International Symposium on Microarchitecture*, pp.45-54, 1992.

[9] L. Carter, B. Simon, B. Calder, L. Carter, J. Ferrante. Path Analysis and Renaming for Predicated Instruction Scheduling, International Journal of Parallel Programming, 28(6), pp 563-586, 2000.

[10] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock, In *Proceedings of the 25th International Symposium on Microarchitecture*, pp.45-54, 1992.

[11] M. S. Schlansker, S. A. Mahlke, and R. Johnson. Control CPR: A branch height reduction optimization for EPIC architectures, In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.155-168, 1999.

[12] D. I. August, J. W. Sias, J.-M. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu. The Program Decision Logic Approach to Predicated Execution, In *Proceedings of the 9th International Conference on Architecture Support for Programming Languages and Operating Systems*, pp.208-219, 2000.