Reverse If-Conversion

Nancy J. Warter Scott A. Mahlke Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

B. Ramakrishna Rau

Hewlett Packard Laboratories
Palo Alto, CA 94303

Abstract

In this paper we present a set of isomorphic control transformations that allow the compiler to apply local scheduling techniques to acyclic subgraphs of the control flow graph. Thus, the code motion complexities of global scheduling are eliminated. This approach relies on a new technique, Reverse If-Conversion (RIC), that transforms scheduled If-Converted code back to the control flow graph representation. This paper presents the predicate internal representation, the algorithms for RIC, and the correctness of RIC. In addition, the scheduling issues are addressed and an application to software pipelining is presented.

1 Introduction

Compilers for processors with instruction level parallelism hardware need a large pool of operations to schedule from. In processors without support for conditional execution, branches present a scheduling barrier that limits the pool of operations to the basic block. Since basic blocks tend to have only a few operations, global scheduling techniques are used to schedule operations across basic block boundaries. Global scheduling consists of two phases, inter-block code motion and local (basic block) scheduling. The engineering problem of global scheduling is to determine how to properly order these phases to generate the best schedule.

In this paper we present a set of isomorphic control transformations (ICTs) that simplify the task of global scheduling to one that looks like local scheduling. This is achieved by defining a predicate intermediate representation (predicate IR) that embodies the code motion properties and thus eliminates the need for an explicit code motion phase during scheduling. The ICTs convert from the control flow graph representation to the predicate IR and vice-versa. If-conversion, a well known technique [1][2], is used to convert an acyclic control flow graph into an enlarged basic block of predicated operations called a hyperblock [3] [4]. After scheduling, a new technique, Reverse If-Conversion (RIC), is used to convert from the scheduled hyperblock to the scheduled control flow graph. Figure 1 shows an overview of the ICT approach to global scheduling.

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 for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
ACM-SIGPLAN-PLDI-6/93/Albuquerque, N.M.
© 1993 ACM 0-89791-598-4/93/0006/0290...$1.50

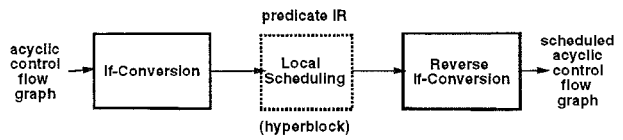


Figure 1: Overview of the Isomorphic Control Transformation (ICT) approach for global scheduling.

If-Conversion is considered an isomorphic control transformation because an operation in the original acyclic control flow graph will have the same condition for execution in the hyperblock. Likewise, RIC is isomorphic because an operation in the scheduled hyperblock will have the same condition for execution in the regenerated acyclic control flow graph. Furthermore, we assume that no operations are inserted during RIC. If an operation is inserted during RIC, it may violate the schedule. This assumption is particularly important for VLIW processors which rely on precise timing relationships.

This paper is organized as follows. In Section 2, we derive the predicate IR. If-Conversion, the scheduling issues, and Reverse If-Conversion are presented in Sections 3 through 5. In Section 6 we prove the correctness of the ICTs, and in Section 7, present an application of the ICT approach for software pipelining acyclic loop bodies. We conclude with a discussion of the strengths and limitations of scheduling under ICT as compared to other global scheduling techniques.

2 Intermediate Representation

The primary program representation is the control flow graph. The problem addressed in this paper is to define an intermediate representation for acyclic subgraphs of the control flow graph that allows the compiler to generate a globally-scheduled control flow graph by applying local scheduling techniques. Other researchers have noted the inadequacies of the control flow graph for applying compiler transformations and have proposed more powerful intermediate representations such as the Program Dependence Graph (PDG) [5][6][7][8][9]. Our intermediate representation

Rules of Code Motion

1) Identical operations from B and C merged into A (hoisting)
2) operation from A copied to both B and C
3) operation moves up from B to A (speculative)
4) operation from A copied only to B (destination is not in live-in set of C)
5) operation from F copied into C, D, and E
6) identical operations from C, D, and E merged into F
7) operation from D copied into F (predicated)

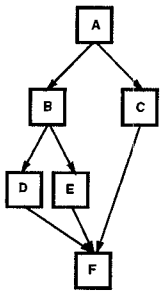
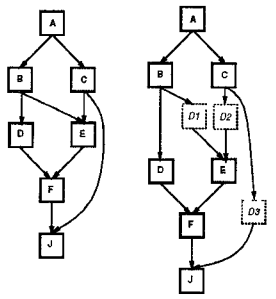


Figure 2: Rules of code motion for global scheduling.



(a) (b)

operation from	Range	Bounds	
		after branch of	before merge at
A	all	-	-
B	B, D, D1	A	E, F
C	C, D2, D3	A	E, J
D	D	B	F
E	D1, D2, E	B, C	F
F	B, D, D1, D2, E, F	E, C	J
J	all	-	-

(c)

Figure 3: Bounds on code motion for example control flow graph. (a) Example control flow graph. (b) Augmented control flow graph. (c) Code motion bounds and ranges assuming no speculative or predicated execution.

builds on the PDG concepts but is designed specifically to assist global instruction-level scheduling.

Before presenting the intermediate representation, we want to analyze the difficulty of scheduling an acyclic control flow graph.

Definition 1: An *acyclic control flow graph* is a directed acyclic graph G with a unique entry node START and a set of exit (EXIT) nodes such that each node in the graph has at most two successors. For any node N in G there exists a path from START to N and a path from N to one of the EXIT nodes.

The nodes of a control flow graph are commonly referred to as basic blocks. Applying global scheduling to an acyclic control flow graph involves two phases, code motion between basic blocks, and scheduling within basic blocks. Figure 2 illustrates the rules of code motion for global scheduling [10][11]. These rules can also be viewed as the basic steps needed for moving operations in the control flow graph. For example, an operation in basic block **F** can be moved into basic blocks **C**, **D**, and **E** by applying rule 5. The operations can then be scheduled in these basic blocks or the identical operations in **D** and **E** can be merged into **B** by applying rule 1. Again, the operations can be scheduled in these basic blocks or the identical operations in **B** and **C** can be merged into **A**. This simple example illustrates the phase ordering problem between the code motion and scheduling phases in global scheduling. It is difficult to determine when to stop code motion and schedule the operations in order to generate the best schedule.

In some code motion cases, it is necessary to create a basic block into which an operation may be copied. Consider the control flow graph in Figure 3(a). It is not possible to simply apply rule 2 to copy an operation from **B** into **D** and **E**. Because **E** has multiple predecessors, the operation from **B** cannot be placed in basic block **E**. Instead, a basic block (**D1**) must be created for the copy of the operation. Figure 3(b) shows an augmented control flow graph with all of the possible basic blocks (**D1**, **D2**, and **D3**) that could be created as a result of applying the code motion rules [10][11].

By applying the code motion rules to operations from every basic block in the control flow graph, the range of possible destination blocks can be determined for each operation. The table in Figure 3(c) presents the range assuming no speculative or predicated code motion. For example, consider an operation **b** in basic block **B**. Since we do not consider speculative execution, **b** cannot be moved to **A**. By applying rule 2, **b** can be copied into **D** and **D1**. However, since we assume no predicated execution, **b** cannot be copied from **D** to **F** or from **D1** to **E**. Thus, the range of allowable basic blocks for operation **b** is **B**, **D**, and **D1**. From this range, we can see that **b** is bounded from above by the conditional branch in **A** and bounded from below by the merges at **E** and **F**.

During local scheduling, only the precedence relations between operations are considered. Thus, if the bounds on code motion can be represented as precedence relations, the task of global scheduling can be simplified to local scheduling. The *upper bound* on code motion defines the bound for upward code motion and corresponds to *control dependence* [1]. An operation x that is bounded by a conditional branch operation y is said to be *control dependent* on y. From the scheduler's viewpoint, x cannot be scheduled until after y has been scheduled. Given the following definition of postdominance, control dependence can be defined [1][5].

Definition 2: A node X is *postdominated* by a node Y in G if every directed path from X to an EXIT node (not including X) contains Y.

Definition 3: All the operations in node Y are *control dependent* upon the conditional branch operation in node X if and only if (1) there exists a directed path P from X to Y such that every node Z in P (excluding X and Y) is postdominated by Y and (2) Y does not postdominate X.

The *lower bound* on code motion defines the bound for downward code motion. There is no existing precedence relation that can be used to define the lower bound on code motion. One problem in defining such a precedence relation is that precedence relations are between two operations, and there is no explicit merge operation. However, we can define an *implicit merge operation* to be an implied operation which proceeds the first explicit operation in every merge node. Now, the lower bound on code motion can be represented by a *control anti-dependence* relation. From a scheduler's viewpoint, a merge operation y cannot be scheduled until every operation x upon which it is control anti-dependent has been scheduled. Using the following definition for dominance [12], control anti-dependence can be defined.

Definition 4: A node X *dominates* a node Y in G if every directed path from START to Y contains X.

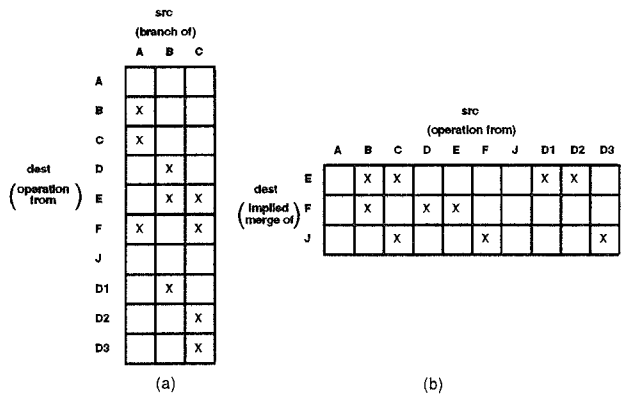


Figure 4: Control precedence relations for operations in the example control flow graph. (a) Control dependence. (b) Control anti-dependence.

Definition 5: The implied merge operation in node Y is *control anti-dependent* upon all the operations in node X if and only if (1) there exists a directed path P from X to Y such that every node Z in P (excluding Y) is dominated by X and (2) X does not dominate Y.

The matrices in Figure 4 show the non-redundant control dependences and control anti-dependences for the example control flow graph in Figure 3(b).

Given the control dependences and control anti-dependences, we need to define an IR that preserves these dependences. Since our goal is to apply local scheduling techniques, an operation is the basic unit of the IR. A predicate, which represents an operation's condition for execution, can be assigned to each operation. Every operation in the acyclic control flow subgraph becomes a *predicated operation.* To preserve control dependences, conditional branches become operations that define the predicates. These operations are referred to as *predicate define operations.* To preserve control anti-dependences, implied merge operations become operations that merge predicates, referred to as *predicate merge operations.*

It is sufficient to represent the control dependences and control anti-dependences using one set of predicates per predicate define operation and one set of predicates per predicate merge operation. However, in order to regenerate the control flow graph, the RIC algorithm must be able to determine which predicates where defined along the two paths of the conditional branch. Thus, two sets of predicates are needed for the predicate define operations, one for predicates defined along the true path and one for predicates defined along the false path of the conditional branch. Likewise, there are two types of predicates paths entering a merge node, those that have jump operations and those that do not. The predicate merge operation has two sets of predicates, those corresponding to paths being merged that require a jump, and those that do not. Table 1 shows the syntax of the predicate IR. Note that since the implicit merge operation is not an actual operation in the original control flow graph, the predicate merge operation is not predicated.

Before discussing the If-Conversion transformation, we return to the idea of augmenting the control flow graph with empty basic blocks. As discussed above, during code motion, basic blocks may need to be generated to hold copies of moved operations. Since local scheduling is applied to the predicate IR, RIC eventually generates the scheduled control flow graph and the effects of code motion will be present. To avoid inserting operations during RIC, the control flow graph is augmented with dummy blocks before If-Conversion. Given an edge $X \rightarrow Y$, if X has multiple successors and Y has multiple predecessors, a dummy node is inserted on $X \rightarrow Y$. If Y is not on the fall-through path of X, insert a jump operation in the dummy node. Algorithm DUM in Appendix A is used to insert dummy nodes.

OPERATION	SYNTAX
predicated operation	< p > op
predicate define	< p > pd [cond] {false}{true}
predicate merge	pm {no_jump}{jump}

Table 1: Predicate Intermediate Representation.

3 If-Conversion

The If-Conversion algorithm presented in this paper is based on the RK algorithm [13]. The basic RK algorithm presented in [13] decomposes the control dependences using two functions R and K. The function R(X) assign a predicate to basic block X such that any basic block that is control equivalent [9] to X is assigned the same predicate. The function K(p) specifies the condition under which a predicate p is defined. We have added two functions, $S_{true}(X)$ and $S_{false}(X)$, to merge all the predicates defined under the true and false conditions of conditional branch of X. The S algorithm is presented in Appendix A.

Figure 5 shows the predicated example control flow graph and corresponding R, K, S_{true}, and S_{false} functions. Note that basic blocks **A** and **J** are control equivalent and thus, have the same predicate **p0**. Since **A** and **J** always execute, K(**p0**) is the empty set since no conditions define their execution.

In this paper we assume that when a node has two successors, the left edge corresponds to the false condition and the right edge corresponds to the true condition of the branch. Consider the conditional branch in basic block **A**. From the control dependence matrix in Figure 4(a), we know that basic blocks **B**, **C**, and **F** are control dependent on the branch in **A**. Using the R function, the predicates of basic block **B**, **C**, and **F** are **p1**, **p2**, and **p5**, respectively. The K function is $\{\bar{\mathbf{A}}\}$ for **p1**, $\{\bar{\mathbf{A}}, \bar{\mathbf{C}}\}$ for **p5**, and $\{\mathbf{A}\}$ for **p2**. Thus, $S_{false}(\mathbf{A})$ is {**p1**,**p5**} and $S_{true}(\mathbf{A})$ is {**p2**}.

Another way of viewing control anti-dependences, is to say that an operation is reverse control dependent on a branch. **Definition 6:** All the operations in node X are *reverse control dependent* upon the implied merge operation in node Y if and only if (1) there exists a directed path P from X to Y such that every node Z in P (excluding Y) is dominated by X and (2) X does not dominate Y.

Reverse control dependence can be calculated by modifying the algorithm for calculating control dependence pre-

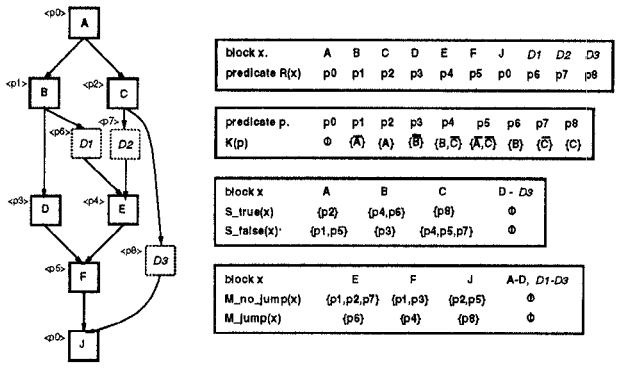


Figure 5: Solutions of R, K, S, and M for example control flow graph.

sented in [13]. The algorithm RCD in Appendix A calculates the reverse control dependences of each node in the directed graph. RCD(X) calculates the set of implied merge operations which every operation in node X is reverse control dependent upon. The algorithm M in Appendix A uses RCD(X) to determine which predicates are being merged at each predicate merge operation. Once it has been determined which predicates to merge, the predicates are divided into the no_jump and jump sets for each predicate merge operation. Given a merge node t in RCD(X) and the predicate, p, of X, p is placed in the jump set of the the predicate merge operation of t if (1) t is an immediate successor of X, and (2) t is not on the fall-through path of X. Otherwise, p is placed in the no_jump set of the predicate merge operation of t.

Figure 5 shows the merge functions M_{jump} and M_{no_jump} of the implicit predicate merge operations at nodes **E**, **F**, and **J**. Consider the implicit predicate merge operation at node **F**. From the control anti-dependence matrix in Figure 4(b), we know that operations in basic blocks **B**, **D**, and **E** are control anti-dependent on the implied merge operation of **F**. Node **E** is an immediate predecessor of **F** and is assumed to have a jump operation (e.g., **F** is not on the fall-through path) and thus its predicate is in the jump predicate set. Thus, $M_{jump}(\mathbf{F})$ is $\{\mathbf{p4}\}$ and $M_{no_jump}(\mathbf{F})$ is $\{\mathbf{p1,p3}\}$.

After If-Conversion, each operation is predicated, conditional branch operations are replaced by predicate define operations, implied merge operations are replaced by predicate merge operations, and jump operations are deleted. Figure 6(a) shows the hyperblock of the example control flow graph after If-Conversion. Note that the R function defines the predicate of each operation; the S_{true} and S_{false} functions define the true and false sets of the predicate define operations; and the M_{no_jump} and M_{jump} functions define the no_jump and jump sets of the predicate merge operations. To illustrate the scheduling and regeneration complexities, Figure 6(a) shows predicated operations for each basic block such that the lower case operations are from the corresponding upper case basic block. The condition of the predicate define operation is specified as the basic block identifier. Also, the predicate merge operations indicate from which basic block they originate. Note that since dummy nodes, **D1**, **D2**, and **D3** do not have any operations before

```
<p0> op_a1
<p0> op_a2
<p0> op_a3
<p0> op_a4
<p0> pd [A] {p1,p5}{p2}
<p1> op_b1
<p1> op_b2
<p1> op_b3
<p1> pd [B] {p3}{p4,p6}
<p2> op_c1
<p2> op_c2
<p2> op_c3
<p2> pd [C] {p4,p5,p7}{p8}
<p3> op_d1
<p3> op_d2
<p3> op_d3
pm_E {p1,p2,p7}{p6}
<p4> op_e1
<p4> op_e2
<p4> op_e3
pm_F {p1,p3}{p4}
<p5> op_f1
<p5> op_f2
<p5> op_f3
pm_J {p2,p5}{p8}
<p0> op_j1
<p0> op_j2
<p0> op_j3
<p0> op_j4
```

(a)

```
I1:   <p0> op_a1
I2:   <p0> op_a2
I3.   <p0> op_j1
I4:   <p0> pd [A] {p1,p5}{p2}
I5:   <p1> op_b1
I6.   <p0> op_j2
I7:   <p2> op_c1 / <p1> op_b2
I8:   <p1> pd [B] {p3}{p4,p6}
I9:   <p3> op_d1 / <p2> pd [C] {p4,p5,p7}{p8}
I10:  <p4> op_e1
I11:  <p5> op_f1
I12:  <p2> op_c2 / <p3> op_d2
I13:  <p4> op_e2
I14:  <p1> op_b3 / <p2> op_c3
I15:  pm_E {p1,p2,p7}{p6}
I16.  <p3> op_d3
I17:  <p5> op_f2
I18:  <p4> op_e3
I19:  pm_F {p1,p3}{p4}
I20:  <p0> op_a3
I21:  <p0> op_j3
I22.  <p5> op_f3 / pm_J {p2,p5}{p8}
I23:  <p0> op_a4
I24:  <p0> op_j4
```

(b)

Figure 6: (a) Hyperblock of example control flow graph. (b) Hyperblock after scheduling.

scheduling, there are no predicated operations corresponding to these blocks. However, their predicates are placed in the appropriate predicate define and predicate merge operations.

4 Scheduling Issues

By using the set of isomorphic control transformations, local (basic block) scheduling techniques can be used to perform global scheduling. The control dependences and control anti-dependences are used to preserve the control flow properties during scheduling. Control dependence prevents operations from being scheduled before the corresponding predicate define. In terms of the control flow graph, control dependences prevent the operations from being moved above a conditional branch. Control dependences may be removed if the processor has hardware support for speculative execution. In this case, the predicate of an operation can be promoted to the predicate of the predicate define operation and thus, the operation can be scheduled before the predicate define [3]. Control anti-dependence ensures that the predicate merge operation is scheduled before any operations that are control anti-dependent upon it. Effectively, this prevents any operations from being scheduled after a merge point in the control flow graph. Control anti-dependences may be removed if the processor has hardware support for predicated

execution [14][2].

The disadvantage of speculative and predicated execution is that the processor will fetch the operation even when the result of the operation is not used. The hyperblock formation can be used to ensure that operations are only moved along the most frequently executed paths. The hyperblock is formally defined as follows.

Definition 7: A *hyperblock* is an If-Converted region formed from an acyclic subgraph of the control flow graph which has only one entry (START) block but may have multiple exit (EXIT) blocks.

Using this definition, if some basic blocks are known to be infrequently executed, they can be excluded from the hyperblock. This allows the compiler to more effectively optimize and schedule the operations from the more frequently executed basic blocks that form the hyperblock. In order to exclude basic blocks, a technique called tail duplication is used to remove additional entry points into the hyperblock. The details of selecting basic blocks to form hyperblocks are provided in [3] [4].

Before scheduling, a hyperblock is a sequence of consecutive operations. After scheduling, a hyperblock is a sequence of consecutive instructions. For a RISC processor, an instruction has one operation. For a VLIW processor, an instruction has w operations, where w is the issue rate of the processor. In the scheduled hyperblock, there may be several operations scheduled on top of one another. Since operations with mutually exclusive predicates will be placed in different basic blocks in the scheduled control flow graph, they can be scheduled to the same resource in the same cycle. We use the Predicate Hierarchy Graph described in [3][4] to determine if two predicates are mutually exclusive.

The predicate merge operation also has special scheduling characteristics. A predicate merge is scheduled as a jump operation under the jump set predicates and as a null operation under the no_jump set predicates. Since there may be multiple predicates in the jump set, multiple jump operations can be scheduled per predicate merge. However, all predicates in the jump set are mutually exclusive and thus their operations can be scheduled to the same resource in the same cycle. A null operation does not require any resources or cycles. To schedule a predicate merge operation at a given time to a given resource, only the predicates in the jump set must be mutually exclusive to the predicates of other operations already scheduled to the resource.

Figure 6(b) shows the example hyperblock after scheduling. The schedule assumes a single-issue processor without interlocking. We present the most restrictive processor model in order to illustrate how *no-op* operations are inserted if needed. Note that the control dependences (control anti-dependences) between predicate define (merge) operations and their respective predicated operations are preserved. We assume that all other data dependences between operations are preserved. Also note that the predicate merge operation **pm_J** is scheduled in instruction **I22**. It can be scheduled in instruction **I22** since all operations that are predicated on **p2**, **p5**, and **p8** are scheduled at or earlier than **I22**. Also, **p8** and **p5** are mutually exclusive and thus both **op_f3** and **pm_J** can be scheduled in the same cycle. Note, that although a predicate merge operation is scheduled as a jump operation, it is not converted to a jump operation until RIC. Similary, a predicate define operation is scheduled as a conditional branch but is not converted to a conditional branch operation until RIC.

Before applying If-Conversion, the control flow graph was augmented with dummy nodes. During scheduling, it can be determined whether a dummy node is needed or not. If a dummy node is not needed, the scheduler should apply a jump optimization to remove unnecessary jump operations when predicate merge operations are scheduled. When a predicate merge operation is scheduled, a jump is required if any operations have been scheduled along the control path between the predicate define, pd, operation that defines a predicate p and the predicate merge operation, pm that contains p in its jump set. This can be done be checking all scheduled operations between pd and the cycle in which pm is being scheduled. If any operations have a predicate that is not mutually exclusive with the predicate p, a jump is needed. For VLIW processors, a further condition is needed. To remove a jump, pm must be scheduled in the same cycle as pd. If a jump is not needed, jump optimization is performed by deleting the predicate p from the jump set and inserting it into the no_jump set of the predicate merge operation

5 Reverse If-Conversion

After scheduling, the hyperblock represents the merged schedule for all paths of the control flow graph. The task of RIC is to generate the correct control flow paths and to place operations into the appropriate basic blocks along each path. Whereas a basic block in the original control flow graph was assigned one predicate, the basic blocks in the regenerated control flow graph have a set of predicates associated with them. The *allowable predicate set* specifies the predicates of the operations that can possibly be placed in the corresponding basic block. Intuitively, the allowable predicate set is the mechanism that accounts for the code motion phase of global scheduling.

The RIC algorithm is presented in Appendix A. The Reverse If-Conversion algorithm processes the operations in the hyperblock in a sequential manner. A leaf node set, L, is maintained that consists of the basic block nodes in which operations can be placed. Thus, at a given cycle, L contains one node from every possible execution path. Initially, L consists of the root node of the regenerated control flow graph. Each node X in L has an allowable predicate set $\rho(X)$. When an operation op in the hyperblock is processed, it is placed in every node X in L if $P(op) \in \rho(X)$. For a VLIW processor, it is necessary to insert *no-op* operations into empty slots. Since each operation slot may have multiple operations with mutually exclusive predicates, *no-op* operations are not inserted until after the entire instruction has been processed.

Let $P(pd)$ be the predicates defined by the predicate define operation pd. When a predicate define operation is encountered, it is inserted into every node X in L where $P(pd) \in \rho(X)$. Each X is then deleted from L. For each such X, two successor nodes $Succ_t$ and $Succ_f$ corresponding to the true and false path of pd are created and inserted

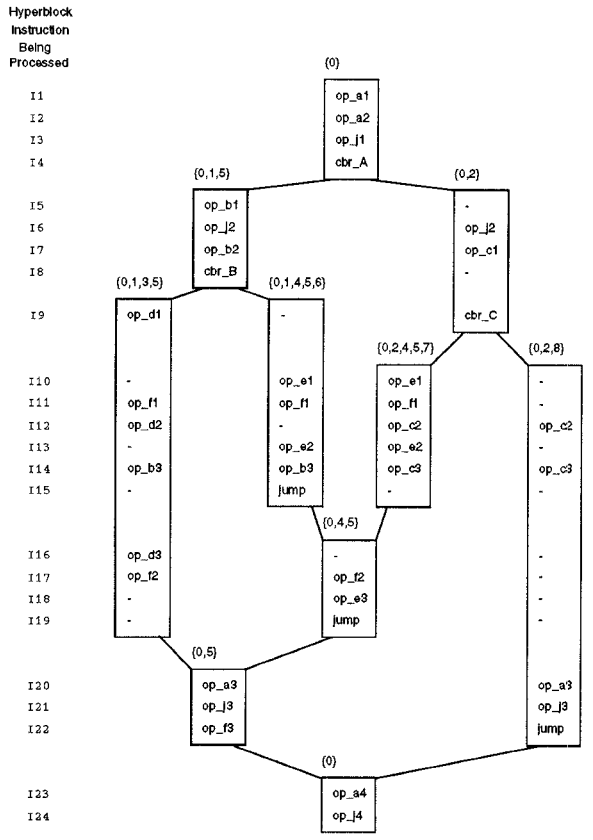


Figure 7: Control flow graph of scheduled hyperblock after RIC.

into L. The allowable predicate sets of $Succ_t$ and $Succ_f$ are,

$$\rho(Succ_t) = \rho(X) \cup true_predicates\ of\ pd, \text{ and}$$

$$\rho(Succ_f) = \rho(X) \cup false_predicates\ of\ pd.$$

When a predicate merge operation is encountered, the following is done for each node X in L. If a predicate $\rho(X)$ is in the jump or no_jump sets, the predicates specified in the jump and no_jump sets are deleted from $\rho(X)$ to create $\rho_{new}(X)$. If it was in the jump set, a jump operation is inserted into X. L is searched for a node Y with the same allowable set as $\rho(X)$. If one is found, Y is the successor of X. Otherwise, a successor is created for X with the allowable set $\rho_{new}(X)$.

Figure 7 shows the control flow graph generated from the scheduled hyperblock in Figure 6(b). The target machine in this example is a single-issue processor without interlocking. Thus, *no-op* operations are required and are represented by a dash. The allowable predicate set is indicated on top of each basic block.

6 Correctness of ICT approach

In this section we prove the correctness of the isomorphic control transform approach. First, we show that DUM preserves the control dependences of the original control flow graph G.

Lemma 6.1 *Given a directed graph, G, the augmented control flow graph created by DUM preserves the control dependence and reverse control dependence of G.*

Proof: Given a directed path P between node X and node Y of G. First consider control dependence, where Y is control dependent on X. Assume that DUM inserts a node Z in P. Since Z only has one successor and Z is in P, every path from Z to an EXIT node contains Y. Thus, Z is postdominated by Y, and hence, Y remains control dependent on X. A similar argument can be made for reverse control dependence. □

Lemma 6.2 *Assuming no speculative or predicated execution, an operation in the regenerated control flow graph is executed if and only if it would be executed in the original control flow graph.*

Proof: During If-Conversion, every operation is assigned a condition for execution (predicate). The control dependences and control anti-dependences are preserved using predicate define and predicate merge operations respectively. We assume that the scheduler does not violate these dependences.

There are three cases in which an operation can execute in the original control flow graph and not in the regenerated control flow graph: 1) the operation is moved from above to below a branch but only placed on one path of the branch, 2) the operation is moved from below to above a merge but not along every path, and 3) the operation is deleted from a path. Since the scheduler does not allow these cases, all three cases can only occur if RIC places an operation on the wrong path or fails to place an operation.

There are three cases in which an operation can execute in the regenerated control flow graph but not in the original control flow graph: 1) the operation is moved above a branch it is control dependent upon, 2) the operation is moved below a merge it is control anti-dependent upon, and 3) the operation is executed along a mutually exclusive path. The first two cases can only occur if RIC violates the dependences. The last case can only occur if RIC places an operation on the wrong path.

RIC preserves the control dependences and control anti-dependences in the following manner. During RIC, a predicate define operation is converted into a conditional branch. The predicates defined along one path of the branch are inserted into the allowable predicate set of that path. An operation is inserted only along the path that has its predicate in the allowable predicate set. Thus, an operation will not be placed before a branch upon which it is control dependent or along the wrong path of the branch. The predicate merge operation is scheduled after every operation upon which it is control anti-dependent. During RIC, paths are not merged until a predicate merge operation is encountered. Thus, an operation will not be placed after a merge. Since RIC preserves the control dependences and control anti-dependences, the predicate of an operation will be in the allowable predicate set of a block when the operation is processed. Thus, an operation cannot be deleted during RIC. □

Lemma 6.3 *Given a VLIW processor, the precise timing relationships of scheduled code in the predicated IR is preserved between any two operations in the regenerated control flow graph.*

Proof: During RIC, instructions are processed according to the schedule. Thus, instructions will be generated with the proper number of cycles between them unless operations are added or deleted. Only jump operations would need to be added during RIC. Dummy node insertion and the predicate merge operation ensure that no jump operations need to be added during RIC. After all VLIW instructions are processed, empty operation slots are filled with *no-op* operations. Thus, no operations will be inserted or deleted during RIC.□

Theorem 6.1 *Assuming the schedule is correct, the ICT approach generates a correct globally scheduled control flow graph.*

Proof: This theorem follows from Lemmas 6.1, 6.2, and 6.3.

7 Software Pipelining Application

One of the benefits of the ICT approach is that existing compiler techniques for processors with support for Predicated Execution [14] can be used for processors without PE support. One such technique is Modulo Scheduling for software pipelining developed by Rau et.al. [2][15]. By performing If-Conversion before Modulo Scheduling, loops with conditional branches can be scheduled in the same manner as those without. However, without RIC, Modulo Scheduling for processors without PE support have relied on techniques such as Hierarchical Reduction, which apply prescheduling to remove conditional branches [16]. Prescheduling limits the effectiveness of Modulo Scheduling [17][18]. Furthermore, Hierarchical Reduction can only be applied to structured loop bodies whereas the ICTs can be applied to any acyclic loop body. In this section we present the Enhanced Modulo Scheduling technique (EMS) which uses the ICTs to simplify scheduling. For further details about EMS, refer to [17].

Figure 8 shows the hyperblock after software pipelining. The target machine is a VLIW processor with two operation slots. Effectively, two iterations of the hyperblock (the schedule from Figure 6(b)) have been overlapped. Note that the operations within each instruction have been reordered to illustrate how the iterations are overlapped. Modulo Variable Expansion [16] and renaming have been applied to the predicate variables. The kernel of the software pipeline is unrolled once so that the predicate variable lifetimes of **p5** and **p8** do not overlap themselves. In the second copy of the kernel, **p5** is renamed to **p9** and **p8** is renamed to **p10**. Although no operations are predicated on **p8**, which corresponds to **D3** in Figure 5, it must be renamed to ensure that the control paths are merged correctly. Note that **op_j4** is the loop back branch. Thus, it is deleted from all but the last stage in the kernel.

Figure 9 shows the corresponding control flow graph after regeneration. The allowable predicate set is indicated for each basic block. The dashes indicate *no-op* operations.

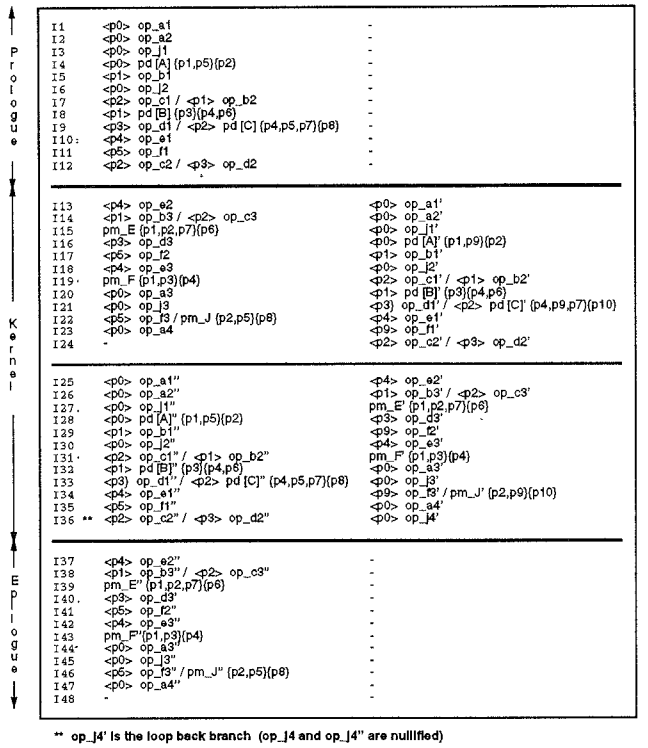


Figure 8: Hyperblock of example control flow graph after software pipelining. The hyperblock is divided to show the four stages of the pipeline.

The modified RK If-Conversion algorithm and RIC have been implemented in the IMPACT-I C compiler. The EMS technique uses these transformations to convert the acyclic loop body to and from the hyperblock which is modulo scheduled. Modulo Scheduling with Hierarchical Reduction has also been implemented. To analyze the benefits of the ICT approach, EMS and Modulo Scheduling with Hierarchical Reduction were applied to 26 *doall* loops from the Perfect benchmarks. All loops contain conditional constructs. The target machine is a VLIW processor without interlocking. The processor allows any combination of non-branch operations per cycle, but only one branch per cycle.

Due to space limitations, we summarize the results. EMS performs 18%, 17%, and 19% better than Modulo Scheduling with Hierarchical Reduction for issue rates 2, 4, and 8, respectively. The code expansion for EMS is 52%, 60%, and 105% larger than for Modulo Scheduling with Hierarchical Reduction for issue rates 2, 4, and 8, respectively. Since EMS has a tighter schedule, the conditional constructs overlap more, causing larger code expansion. These results show that the ICT approach can be used to significantly increase the performance of Modulo Scheduling when there is no special hardware support for PE. However, the effect of code expansion on performance still needs to be analyzed. A more in-depth analysis of these two techniques including a comparison with Modulo Scheduling with PE is presented in [17].

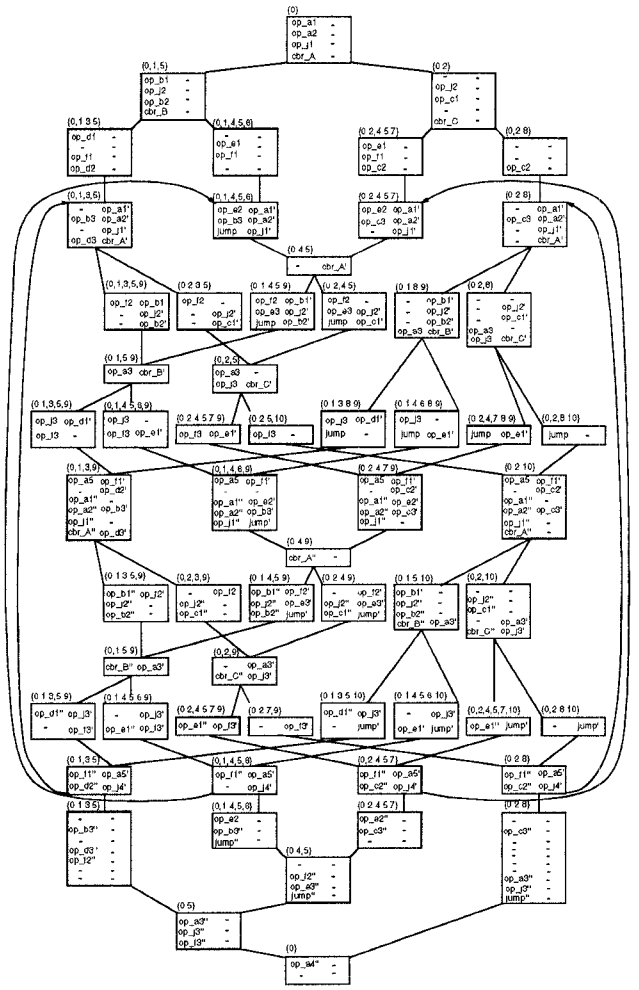


Figure 9: Control flow graph of software pipelined hyperblock after RIC.

8 Discussion

The ICT approach presented in this paper extends the concept of a predicate IR to processors without support for Predicated Execution (PE) [2][14]. Thus, in addition to applying If-Conversion [1][2][13], an inverse isomorphic control transformation, RIC, is applied to regenerate the control flow graph structure.

The ICT approach is also very similar to the the Program Dependence Graph (PDG) approach [5]. In addition to representing the data dependences, the PDG explicitly represents control dependences. Whereas the predicate IR assigns predicates to operations, the PDG assigns predicates to regions, each of which contains operations with the same execution condition. Thus, global scheduling techniques for the PDG require explicit code motion and thus need to determine the proper phase ordering between code motion and local scheduling in order to achieve the best schedule [8][9].

While the ICTs remove the phase ordering problem faced by other global acyclic scheduling techniques, we do not claim that it is possible to find the best schedule for an acyclic subgraph. Rather, we simply state that the compiler can schedule a much larger scope of operations and not have to worry about code motion. In addition to the code motion rules mentioned in Section 2, the code motion complexity incurred by moving branch operations above other branch operations [11] is eliminated. The ICT approach allows branches and merges to be scheduled in the same manner as other operations.

There are some drawbacks to the scheduling approach presented in the paper. First, an operation in the original flow graph is never replicated during scheduling. Thus, an operation is not scheduled until all control dependences are resolved. This may cause an operation to be scheduled later than necessary on one or more of the control paths. Furthermore, it is difficult to associate data flow analysis, such as live-in sets, with the operations within a hyperblock. Thus, it is currently not possible to move an operation from above to below a branch and schedule it on only one path of the branch. This is allowed when the result of the operation is in the live-in set of only one successor of the branch. These drawbacks are not necessarily drawbacks of the ICT approach to scheduling, but they do exist in the scheduling scheme presented in this paper.

Similar problems exist for cyclic scheduling. During modulo scheduling, since all paths are scheduled at once, the initiation interval is the same for all control paths through the loop [15][16][18]. While some global scheduling-based software pipelining techniques do not require a fixed initiation interval [19], we are not aware of any methods that support processors with limited resources and non-uniform latencies.

In this paper we have assumed that no operations are inserted during RIC and thus the predicate merge operations are used to specify when jump operations should be scheduled. Currently, the merge points are described in terms of the original control flow graph. In some sense, this is an arbitrary decision[1]. Consider the case where three control paths enter the merge node of a control flow graph. After scheduling, two paths may be able to be merged earlier than the third. However, since we do not merge any paths until all control anti-dependences are resolved, an early merge will not occur in our technique. It is possible to split the merge operations into individual predicate kill operations, where each predicate kill is scheduled as a jump operation. Since predicate merges, and hence predicate kills, are not predicated, there is no ordering imposed on the merge points and it is possible to merge identical control paths in any arbitrary fashion. The cost of this solution is that a larger number of jump operations will be scheduled than required. If they are not used, they are nullified with *no-op* operations. Thus, there is a tradeoff between performance and code expansion.

Alternatively, if operations can be inserted during RIC, then it is possible to merge control paths when they become the same by detecting when a predicate is no longer used in the scheduled hyperblock. In this case, a predicate merge operation is not needed.

[1]Basic block layout can be performed as a preprocessor step to make this less arbitrary.

9 Conclusion

In this paper we have presented a predicate IR that preserves control dependences and control anti-dependences. A set of isomorphic control transformations (ICTs) are used to convert an acyclic control flow graph to and from the predicate IR. If-Conversion, modified to support control anti-dependences, is used to transform acyclic control flow subgraphs into predicated hyperblocks. A Reverse If-Conversion technique was presented which transforms scheduled hyperblocks into scheduled acyclic control flow subgraphs. Using this set of transforms, the task of global scheduling is reduced to local scheduling. In this paper we have shown that the ICTs can be used to significantly improve the performance of Modulo Scheduling for processors without Predicated Execution support. We are currently investigating using the ICT methodology for other hyperblock optimization and scheduling techniques.

Acknowledgments

The authors would like to thank Grant Haab along with all members of the IMPACT research group for their comments and suggestions. Special thanks to the program committee whose comments and suggestions helped to improve the quality of this paper significantly. This research has been supported by JSEP under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Co. Ltd., Mazda Motor Co., Hewlett-Packard, and NASA under Contract NASA NAG 1-613 in cooperation with ICLASS. Scott Mahlke is also supported by a fellowship provided by the Intel Foundation.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[2] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, April 1989.

[3] 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 25th Annual International Symposium on Microarchitecture*, December 1992.

[4] D. C. Lin, "Compiler support for predicated execution in superscalar processors," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 310–349, July 1987.

[6] R. Cytron, J. Ferrante, and V. Sarkar, "Experience using control dependence in PTRAN," in *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[7] W. Baxter and I. H. R. Bauer, "The program dependence graph and vectorization," in *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, pp. 1–10, January 1989.

[8] R. Gupta and M. L. Soffa, "Region scheduling: An approach for detecting and redistributing parallelism," *IEEE Transactions on Software Engineering*, vol. 16, pp. 421–431, April 1990.

[9] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 241–255, June 1991.

[10] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[11] J. Ellis, *Bulldog: A Compiler for VLIW Architectures.* Cambridge, MA: The MIT Press, 1985.

[12] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[13] J. C. H. Park and M. Schlansker, "On Predicated Execution," Tech. Rep. HPL-91-58, Hewlett Packard Software Systems Laboratory, May 1991.

[14] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, pp. 12–35, January 1989.

[15] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[16] M. Lam, *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie Mellon University, Pittsburg, PA, 1987.

[17] N. J. Warter, J. W. Bockhaus, G. E. Haab, and K. Subramanian, "Enhanced Modulo Scheduling for loops with conditional branches," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 170–179, November 1992.

[18] N. J. Warter, D. M. Lavery, and W. W. Hwu, "The benefit of Predicated Execution for software pipelining," in *Proceedings of the 26th Hawaii International Conference on System Sciences*, vol. 1, pp. 497–506, January 1993.

[19] K. Ebcioglu and A. Nicolau, "A global resource-constrained parallelization technique," in *Proceedings of the International Conference on Supercomputing*, pp. 154–163, June 1989.

Appendix A

Algorithm DUM: Given a rooted graph G, (N, E, START), insert dummy nodes where necessary.

```
Let ∀ X ∈ N
    num_succ(X) = the number of successors of X
    num_pred(X) = the number of predecessors of X

for[X,Y] ∈ E
    if (num_succ(X) > 1) and (num_pred(Y) > 1)
        insert dummy node on edge [X,Y]
        if(Y is not the fall-through path of X)
            insert jump operation in dummy node
```

Algorithm S: Given (1) a rooted graph G, (N, E, START), (2) an ordered set of predicates, P, determined by R(n) ∀ n ∈ N, and (3) the mapping K(p) ∀ p ∈ P, compute S. For any node X that contains a conditional branch operation, $S_{true}(X)$ ($S_{false}(X)$) specifies the set of predicates defined along the true (false) path of X.

```
for p ∈ P
    for X ∈ K(p)
        if condition(X) is TRUE
            S_true(X) = S_true(X) ∪ p
        else
            S_false(X) = S_false(X) ∪ p
```

Algorithm RCD: Given a rooted graph G, (N, E, START), compute the reverse control dependence. Assume that dominators of G have been calculated.

```
Let ∀ X ∈ N
    dom(X) = {Y ∈ N: Y dominates X}
    idom(X) = the immediate dominator of X

for [X,Y] ∈ E such that X ∉ dom(Y)
    LUB = idom(Y)
    t = X
    while (t ≠ LUB)
        RCD(t) = RCD(t) ∪ Y
        t = idom(t)
```

Algorithm M: Given a rooted graph G, (N, E, START) and RCD of G, compute M. M_j specifies the set of predicates to be merged whose corresponding blocks require an explicit jump to be inserted. M_{nj} specifies those that do not.

```
Let ∀ X ∈ N
    P(X) = predicate that X is control dependent upon

for X ∈ N
    for t ∈ RCD(X)
        if t ≠ ∅
            if (t fall-through path of X) or
              (t not immediate successor of X)
                M_nj(t) = M_nj(t) ∪ P(X)
            else
                M_j(t) = M_j(t) ∪ P(X)
```

Algorithm RIC: Given hyperblock H, regenerate a correct control flow graph. For VLIW processors, insert_no-op fills each empty operation slot in an instruction with a no-op operation.

```
Let ∀ X ∈ H
    P(X) = predicate that X is control dependent upon

create root node
L = {root}
ρ(root) = {p0}
for op ∈ H in scheduled order
    for X ∈ L
        if op is predicate define operation
            if P(op) ∈ ρ(X)
                insert conditional branch operation in X
                create successor nodes Succ_t and Succ_f
                ρ(Succ_t) = ρ(X) ∪ true(op)
                ρ(Succ_f) = ρ(X) ∪ false(op)
                L = (L − X) ∪ Succ_t ∪ Succ_f
        if op is predicate merge
            for p ∈ (jump(op) ∪ no_jump(op))
                ρ_new(X) = ρ(X) − no_jump(op) − jump(op)
                if p ∈ ρ(X)
                    if p ∈ jump(op)
                        insert jump operation in X
                    for Y ∈ L
                        if ρ_new(X) ≡ ρ(Y)
                            if p ∈ jump(op)
                                Succ_t = Y
                            else
                                Succ_f = Y
                            L = L − X
                    if successor not found
                        if p ∈ jump(op)
                            create successor node Succ_t
                            ρ(Succ_t) = ρ_new(X)
                            L = (L − X) ∪ Succ_t
                        else
                            create successor node Succ_f
                            ρ(Succ_f) = ρ_new(X)
                            L = (L − X) ∪ Succ_f
        if op is predicated operation
            if P(op) ∈ ρ(X)
                insert op in X
    if target processor is VLIW and last op of instruction
        insert_no-op
```