Unrestricted Code Motion: A Program Representation and Transformation Algorithms Based on Future Values*

Shuhan Ding and Soner Önder

Department of Computer Science
Michigan Technological University
shding@mtu.edu, soner@mtu.edu

Abstract. We introduce the concept of *future values*. Using future values it is possible to represent programs in a new control-flow form such that on any control flow path the data-flow aspect of the computation is either traditional (i.e., definition of a value precedes its consumers), or reversed (i.e., consumers of a value precede its definition). The representation hence allows unrestricted code motion since ordering of instructions are not prohibited by the data dependencies. We present a new program representation called *Recursive Future Predicated Form* (RFPF) which implements the concept. RFPF subsumes general *if-conversion* and permits unrestricted code motion to the extent that the whole procedure can be reduced to a single block. We develop algorithms which enable instruction movement in acyclic as well as cyclic regions and give examples of various optimizations in RFPF form.

1 Introduction

Code motion is an essential tool for many compiler optimizations. By reordering instructions, a compiler can eliminate redundant computations [4,9,10], schedule instructions for faster execution [17], or enable early initiation of long latency operations, such as possible cache misses. In these optimizations, the range of code motion is limited by data and control dependencies [4,5]. Therefore, code-optimization algorithms which rely on code-motion have to make sure that control and data dependencies are not violated.

Ability to move code in a control-flow setting in an unrestricted manner would have several significant benefits. Obviously, having the necessary means to move instructions in an unrestricted manner while maintaining correct program semantics could enable the development of simpler algorithms for program optimization. More importantly however, when we permit code motion beyond the obvious limits, code-motion itself can become a very important tool for program analysis.

* This work is supported in part by a NSF CAREER award (CCR-0347592) to Soner Önder.

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 26–45, 2010.

In this paper, we first present the concept of *future values*. Future values allow a consumer instruction to be placed before the producer of its source operands. Using the concept, we develop a program representation which is referred to as *Recursive Future Predicated Form* (RFPF). RFPF is a new control-flow form such that on any control flow path the data-flow aspect of the computation is either traditional (i.e., definition of a value precedes its consumers), or reversed (i.e., consumers of a value precede its definition). When an instruction is to be hoisted above an instruction that defines its source operands, the representation updates the data-flow aspect to become reversed (i.e., *future*, meaning that the instruction will encounter the definition of its source operands in a future sequence of control-flow). If, on the other hand, the same instruction is propagated down, the representation will update the data-flow aspect to become traditional again. The representation hence allows unrestricted code motion since ordering of instructions is not prohibited by the data dependencies. Of course, for correct computation, the values still need to be produced before they can be consumed. However, with the aid of a future values based program representation, the actual time that this happens will appropriately be delayed.

RFPF is a representation built on the principle of *single-assignment* [3,7] and it subsumes general *if-conversion* [2]. In this respect, RFPF properly extends the SSA representation and covers the domain of legal transformations resulting from instruction movements. Possible transformations range from the starting SSA form where all data-flow is traditional, to a final reduction where the *entire procedure becomes a single block* through upward code motion, possibly with mixed (i.e., traditional and future) data-flow. We refer to a procedure which is reduced to a single block through code motion to be in *complete RFPF*. *Complete RFPF* expresses the program semantics without using control-flow edges except sequencing. During the upward motion of instructions, valuable information is collected and as it is shown later in the paper, this information can be used to perform several sophisticated optimizations such as *Partial Redundancy Elimination* (PRE). Such optimizations typically require program analysis followed by code motion and/or code restructuring [12,9,4].

Our contributions in this paper are as follows: (1) We introduce the novel concept of *future values* which permits a consumer instruction to be encountered before the producer of its source operand(s) in a control-flow setting; (2) Using future values, we introduce the concept of *future predicates* which permits instruction hoisting above the controlling instructions by specifying *future control flow*; (3) We introduce the concept of *instruction-level recursion*. This concept allows the loops to be represented as straight-line code and analyzed with ease. Combination of future predicates and instruction-level recursion enables predication of backward branches; (4) Using the concepts of future values, future predicates and instruction-level recursion, we develop a unified representation (RFPF) which is control-flow based, yet instructions can freely be reordered in this representation by simply comparing the instruction's predicate, source and destination variables to the neighboring instruction; (5) We illustrate that unrestricted code motion itself can be used to analyze programs for optimization opportunities.

We present a PRE example in which redundancy cannot be eliminated using code motion alone and restructuring is necessary, yet both the discovery and the optimization of the opportunity can be performed with ease; (6) We present algorithms to convert conventional programs into the RFPF. These algorithms are low in complexity and with the exception of identification of loop headers and the nesting of the loops in the program, they do not need additional external information to be represented. Instead, these algorithms operate by propagating instructions and predicates and use only the local information available at the vicinity of moved instructions; (7) We illustrate that for any graph with mixed-mode data-flow, there is a path through instruction reordering and control flow node generation to convert the future data-flow in the representation back to a traditional SSA graph, or generate code directly from the representation.

In the remainder of the paper, in Section 2, we first present the concept of future values. Section 3 through Section 6 illustrate a process through which instructions can be hoisted to convert a program into RFPF while collecting the data and control dependencies necessary to perform optimizations. For this purpose, we first illustrate how the concept can be used for instruction movement in an acyclic region in Section 3. This set of algorithms can be utilized by existing optimization algorithms that need code motion by incorporating the concept of future-values into them. Code motion in cyclic regions requires conversion of loops into instruction-level recursion. We introduce the concept of instruction-level recursion in Section 4. This section presents the idea of recursive predicates and illustrates how backward branches can be predicated. Next, in Section 5 we give an algorithm for computing recursive predicates. Combination of code motion in acyclic and cyclic regions enables the development of an algorithm that generates procedures in *complete RFPF* from a given SSA program using a series of topological traversals of the graph and instruction hoisting. Since reordering of instructions has to deal with explicit dependencies, memory dependencies pose specific challenges. We discuss the handling of code motion involving memory dependencies in Section 6. Section 7 gives examples of optimizations using the RFPF form. We discuss the conversion back into CFG in Section 8. Finally we describe the related work in Section 9 and summarize the paper in Section 10.

2 The Concept of Future Values

Any instruction ordering must respect the true data dependencies as well as the control dependencies. As a result, an instruction cannot normally be hoisted beyond an instruction which defines the hoisted instruction's source operand(s). When such a hoisting is permitted, a *future dependency* results:

Definition 1. *When instructions I and J are* true dependent *on each other and the instruction order is reversed, the true dependency becomes a* future dependency *and is marked on the source operand with the subscript* f.

Consider the statements shown in Figure 1(a). In this example, the control first encounters instruction `i1` which computes the value `x`, and then encounters the

(a) True dependence (b) Future (reversed) dependence

i_1: if($a < b$) i_1: P=($a < b$) i_2: $[P_f]x = x + 1$
i_2: x = x +1 i_2: $[P]x = x + 1$ i_1: P=($a < b$)

(c) Traditional control-flow (d) If-conversion (e) Future control-dependence

Fig. 1. The concept of Future data and control dependences

instruction `i2` which consumes the value. In Figure 1(b), the instruction `i2` has been hoisted above `i1`, and its source operand `x` has been marked to be a *future value* using the subscript `f`. If the machine buffers any instructions whose operands are future values alongside with any operand values which are not future until the producer instruction is encountered, the instructions can be executed with proper data flow between them even though the order at which the control has discovered them is reversed. Similarly, we can represent control dependencies in future form as well. Consider Figure 1(c). In this example, i_2 is control dependent on i_1. In Figure 1(d) predicate P is used to guard i_2, which represents the same control dependence. When the order of i_1 and i_2 is reversed(Figure 1(e)), predicate P becomes a future value and thus the original control dependence becomes future control dependence.

The combination of future data and control dependencies and single-assignment semantics permit unrestricted code motion. In the rest of the paper, single-assignment semantics is assumed and all the transformations maintain the single-assignment semantics. We first discuss code motion using future values in acyclic regions involving control dependencies.

3 Code Motion in Acyclic Code

For an acyclic control-flow graph $G =< s, N, E >$ such that, s is the start node, N is the set of nodes and E is the set of edges, instruction hoisting involves one of three possible cases. These are: (1) movement that does not involve control dependencies (i.e., straight-line code), (2) *splitting* (i.e., parallel move to predecessor basic blocks), and (3) *merging* (i.e., parallel move to a predecessor block that dominates the source blocks). Note that movement of a ϕ-node is a special case and normally would destroy the single-assignment property. We examine each of these cases below:

Case 1 (Basic block code motion). Consider instructions I and J. Instruction J follows instruction I in program order. If I and J are true dependent, hoisting J above I converts the true dependency to a future dependency. Alternatively, if the instructions are future dependent on each other, hoisting J above I converts the future dependency to a true dependency (Figure 1(a) and (b)).

When code motion involves control dependencies, the instruction propagation is carried out using instruction predication, instruction cloning and instruction merging. An instruction is cloned when the instruction is moved from a control independent block to a control dependent block. Cloned copies then propagate along the code motion direction into different control dependent blocks. When cloned copies of instructions arrive at the same basic block they can be merged.

Case 2 (Splitting code motion). Consider instruction I that is to be hoisted above the block that contains the instruction. For each incoming edge e_i a new block is inserted, a copy of the instruction is placed in these blocks and a ϕ-node is left in the position of the moved instruction (Figure 2).

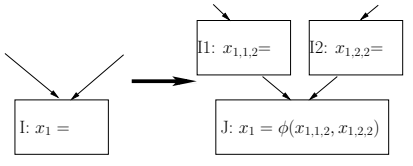


Fig. 2. Splitting code motion

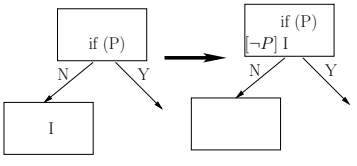


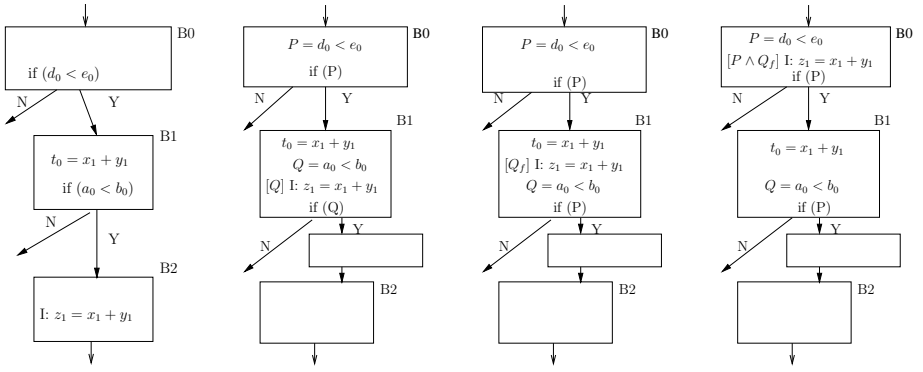
Fig. 3. Merging code motion

Note that in Figure 2, when generated copies I1 and I2 are merged back into a single instruction, the inserted ϕ-node can safely be deleted and the new instruction can be renamed back to x_1. The two new names created during the process, namely, $x_{1,1,2}$ and $x_{1,2,2}$ are eliminated as part of the merging process. In order to facilitate easy merging of clones, we adopt the naming convention $v_{i,j,k}$ where v_i is an SSA name, j is the copy version number and k is the total number of copies. Generated copies can be merged when they arrive at the immediate dominator of the origin block, and in case of reduction to a single block, all copies can be merged. We discuss these aspects of merging later in Section 3.3.

Case 3 (Merging code motion). Consider instruction I that is to be hoisted into a block where the source block is control dependent on the destination block. The instruction I is converted to a predicated instruction labeled with the controlling predicate of the edge (Figure 3).

3.1 Future Predicated Form

When a predicated instruction is hoisted above the instruction which defines its predicate, the predicate guarding the instruction becomes *future* as the predicate is also a value and the data dependence must be updated properly. Figure 4 shows a control dependent case. Instruction I is control dependent on condition $a_0 < b_0$. When the instruction I is moved from $B2$ to $B1$, it becomes predicated and is guarded by Q (Figure 4(b)). In the next step, the instruction is hoisted above the definition of Q and its predicate Q becomes future (i.e., Q_f) (Figure 4(c)).



(a) before code motion (b) after code motion (c) future predicate (d) nested predicate

Fig. 4. Code motion across control dependent regions

When a predicated instruction is hoisted further, it may cross additional control dependent regions and will acquire additional predicates. Consider Figure 4(c). Since the target instruction is already guarded by the predicate Q_f, when it moves across the branch defined by P, it becomes guarded by a nested predicate (Figure 4(d)). In terms of control flow, it means that predicate P must appear, and it will appear before Q. Similarly, if P is true, then Q must also appear since if the flow takes the true path of P the predicate Q will eventually be encountered. In other words, the conjunction operator has the short-circuit property and it is evaluated from left to right. Semantically, a nested predicate which involves future predicates is quite interesting as it defines *possible* control flow.

3.2 Elimination of ϕ-Nodes

RFPF transformations aim to generate a single block representing a given procedure. The algorithms developed for this purpose hoist instructions until all the blocks, except the start node are empty. Proper maintenance of the program semantics during this process requires the graph to be in single-assignment form. On the other hand, movement of ϕ-nodes as regular instructions is not possible and the elimination of ϕ-nodes result in the destruction of the single-assignment property. For example, elimination of the ϕ-node $x_3 = \phi(x_1, x_2)$ involves insertion of copy operations $x_3 = x_1$ and $x_3 = x_2$ across each incoming edge in that order. Such elimination creates two definitions of x_3 and the resulting graph is no longer in single-assignment form. Our solution is to delay the elimination of ϕ-nodes until the two definitions can be merged, at which time a *gating function* [13] can be used if necessary:

Definition 2. *We define the gating function* $\psi_p(a1, a2)$ *as an executable function which returns the input* $a1$ *if the predicate* p *is true and* $a2$ *otherwise.*

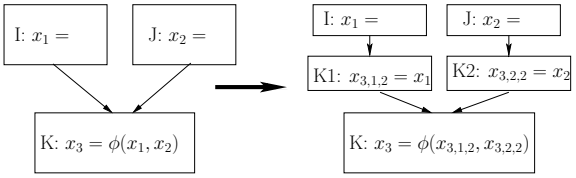


Fig. 5. ϕ-node elimination

Note that during merging, cloned copies already bring in the necessary information for computing the controlling predicate for the gating function. The merging process is enabled by transforming the ϕ-node in a manner similar to the *splitting* case described above:

Case 4 (ϕ-node elimination). Consider the elimination of the ϕ-node $x_3{=}\phi(x_1, x_2)$ (Figure 5). ϕ-node elimination can be carried out by placing copy operations $x_{3,1,2} = x_1$ and $x_{3,2,2} = x_2$ across each incoming edge in that order and updating the ϕ-node with the new definitions to become $x_3 = \phi(x_{3,1,2}, x_{3,2,2})$.

Merging of the instructions $x_{3,1,2} = x_1$ and $x_{3,2,2} = x_2$ requires the insertion of a *gating function* since the right-hand sides are different. Once the instructions are merged, the ϕ-node can be eliminated. It is important to observe that until the merging takes place and the deletion of the ϕ-node, instructions which use the ϕ-node destination x_3 can be freely hoisted by converting their dependencies to *future dependencies*.

3.3 Merging of Instructions

In general, upward instruction movement will expose all paths resulting in many copies of the same instruction guarded by different predicates. This is a desired property for optimizations that examine alternative paths such as PRE and related optimizations since partial redundancy needs to be exposed before it can be optimized. We illustrate an example of PRE optimization in Section 7. On the other hand, the code explosion that results from the movement must be controlled. RFPF representation allows copies of instructions with different predicates to be merged. Merging can be carried out between copies of instructions which result from a splitting move, as well as those created by ϕ-node elimination. As previously indicated, merging of two instructions with the same derivative destination (i.e., such as those which result from ϕ-node elimination) requires the introduction of the *gating function* ψ into the representation, whereas merging of the two copies of the same instruction can be conducted without the use of a gating function. When the merged instructions are the only copies, the resulting instruction can be renamed back to the ϕ destination. Otherwise, a new name is created for the resulting instruction, which will be merged with other copies later during the instruction propagation.

Definition 3. *Two instructions* $\gamma : x_{i,m,k} \leftarrow e1$ *and* $\delta : x_{i,n,k} \leftarrow e2$*, where* γ *and* δ *are predicate expressions, represent the single instruction* $\gamma \vee \delta : x_{i,(m,n),k} \leftarrow e1$ *if e1 and e2 are identical.*

Definition 4. *Two instructions* $\gamma : x_{i,m,k} \leftarrow e1$ *and* $\delta : x_{i,n,k} \leftarrow e2$*, where* γ *and* δ *are predicate expressions represent the single instruction* $\gamma \vee \delta : x_{i,(m,n),k} \leftarrow \psi_P(e1, e2)$ *if e1 and e2 are not identical. The predicate expression* P *is the first predicate expression in* γ *and* δ *such that* P *controls* γ *and* $\neg P$ *controls* δ*.*

Definition 5. *Instruction* $\gamma : x_{i,(p,\ldots,q),k} \leftarrow e$ *can be renamed back to* $\gamma : x_i \leftarrow e$ *if* $(p, \ldots, q)$ *contains a total of* k *version numbers.*

Theorem 1. *Copy instructions generated from a given instruction* I *during upward propagation are merged at the immediate dominator of the source node of* I*, since all generated copies will eventually arrive at the immediate dominator of the source block.*

Proof. Let node A be the immediate dominator of the source node I has originated from in the forward CFG. Assume there's one copy instruction I' which does not pass through A during the whole propagation. For this to happen, there must be a path p, which from the start node reaches I' and then reaches the source node of I. The fact that p does not pass through node A conflicts the assumption that A is the immediate dominator node of I.

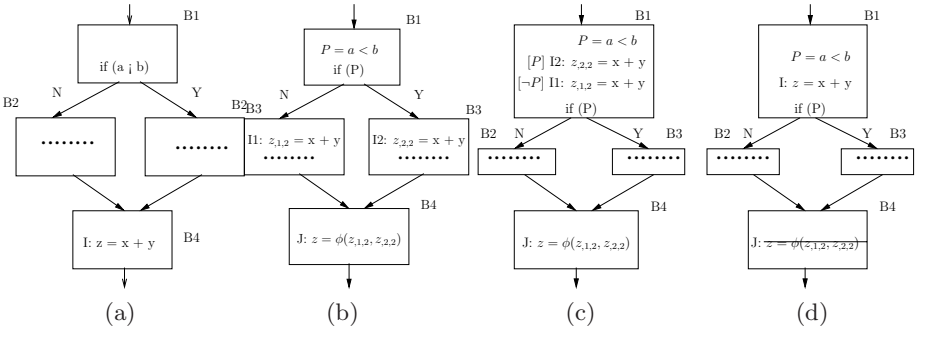


Fig. 6. Instruction propagation

Let us now see through an example how the instruction merging effectively eliminates unnecessary code duplication. Consider the CFG fragment shown in Figure 6(a). Suppose that instruction I needs to be moved to block $B1$. Further note that instruction I is control independent of the block B1. We first insert the branch condition $P = a < b$ in block $B1$. Moving of I is accomplished by applying the *splitting* transformation, followed by progression of $I1$ and $I2$ into blocks B2 and B3 respectively and the deletion of temporary nodes inserted during the movement (Figure 6(b)). Next, the instructions $I1$ and $I2$ are propagated using a *merge* move which predicates them with $\neg P$ and P respectively and places them

in block $B1$ (Figure 6(c)). At this point, using Definition 3, the two instructions can be reduced to a single instruction I without a predicate (Figure 6(d)) and the ϕ-node can be deleted. Note that the merging of the instructions and the deletion of ϕ node must be carried out at the same step to maintain single-assignment property.

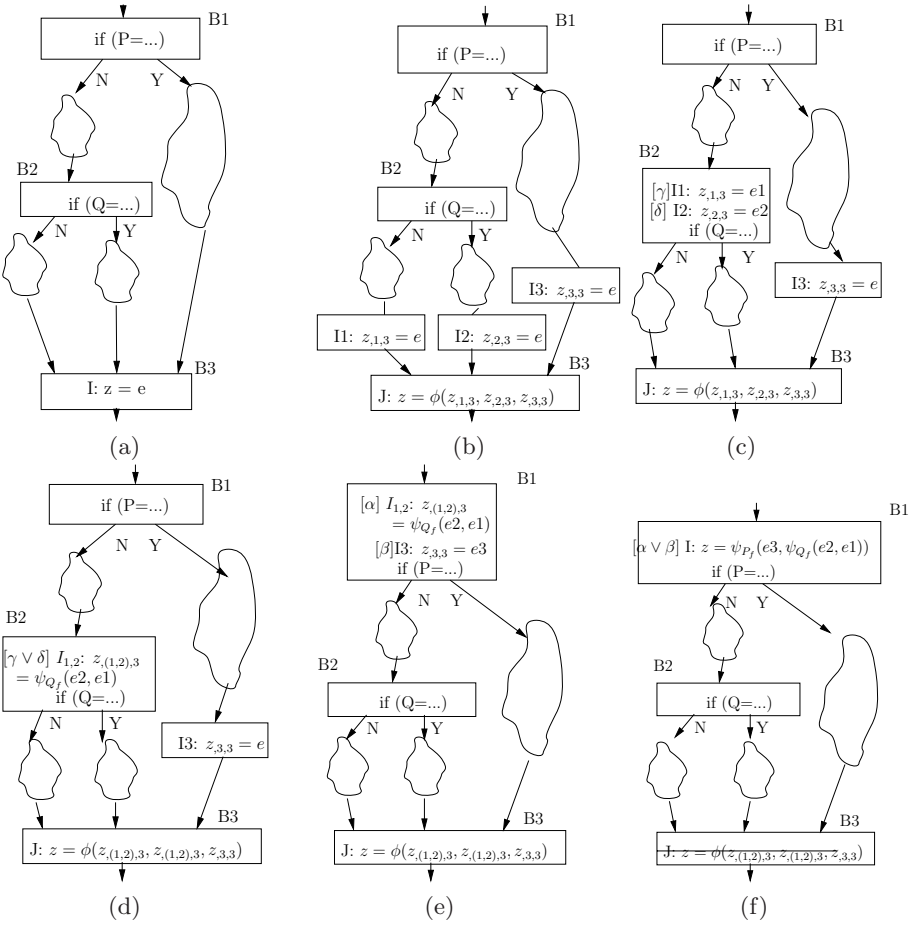


Fig. 7. Instruction merging

A detailed example which shows how the adopted naming convention facilitates instruction merging is illustrated in Figure 7(a). In this example regions represented as *clouds* are arbitrary control and dataflow regions an instruction has to pass through and cloud regions have no incoming or outgoing edges except for the explicitly indicated ones. Instruction I, which computes e is moved across block $B3$ by applying a *splitting* transformation (Figure 7(b)). Next, two of the total three copy instructions, namely, $I1$ and $I2$ converge in block $B2$ and

during propagation may acquire different expressions, namely, $e1$ and $e2$. These two instructions are merged into $I_{1,2}$ using Definition 4 (Figure 7(c)(d)).

Note that future predicate Q_f is used in the gating function for choosing between $e1$ and $e2$. At this point, checking the name of destination $z_{,(1,2),3}$, indicates that there are unmerged copies. Further instruction propagation results in the merging of $I_{1,2}$ and $I3$ in block $B1$. Applying Definition 4 and 5, all the copy instructions are reduced into a single instruction I, which is represented through a nested gating function. At this point the ϕ-node can be deleted. The final result is shown in Figure 7(f).

4 Instruction-Level Recursion

In a reducible control-flow graph, a loop region is a strongly connected region where the loop header forms the upward propagation boundary. Therefore moving instructions across the loop header requires a new approach. This approach is to convert every instruction within the loop region to an equivalent instruction that can iterate in parallel with the loop execution independently. We define an instruction that *schedules* its next iteration, a *recursive instruction.*

Conceptually, a recursive instruction appears as a function call that is spawned at the point the control visits the instruction. The instruction executes within this envelope and checks a predicate to see if it should execute in the next iteration. If the predicate is true, a recursive call is performed. Otherwise the function returns the last value it had computed. In this way, as long as the predicate which controls the loop iteration is known, any loop instruction can iterate itself and hence it can be separated from the loop structure (or pushed out of the loop region). In other words, an instruction that is hoisted above the loop header becomes a recursive instruction controlled by a special predicate called the *Recursive Predicate*:

Definition 6. *Recursive Predicate: In a loop L that has a single loop header H and a single backedge e, the predicate expression which allows control flow to reach e from H without going through e is Recursive Predicate for L.*

For loops with multiple edges we can use the disjunction of the recursive predicates computed for each edge. This follows from the observation that we can insert an empty block such that all the backedges are connected to this block and removed from the loop header and a single exit from this block becomes the single backedge for the graph. Since the controlling predicate of the newly inserted block's outgoing edge is the disjunction of the controlling predicates of all the incoming edges, such graphs can be reduced into a single backedge case described above.

Since the instruction returns only its last value, we can establish proper data dependencies with instructions outside the loop region. Note that, a recursive instruction should also include a predicate to implement the control flow within the loop body:

Definition 7. Recursive Predicated Instruction: $x_i = (R)[P]\{I : x_{i_j} = ...\}$, *where I is the instruction,* x_i *is the SSA name of the instruction's destination, j is the loop nest level, P is the predicate guarding I obtained through acyclic instruction propagation into the loop header and R is the recursive predicate the instruction iterates on.*

Note that the recursive instruction renames the destination of the original instruction by appending the loop nest level, and the function returns the original name. In Section 5.2 we revisit this renaming. From an executable semantics perspective, a recursive predicate must need to know the number of readers it is being waited by and should generate a new value after all the readers have read it.

5 Code Motion in Cyclic Code and Recursive Future Predicated Form

We follow a hierarchical approach to perform code motion in cyclic code. For this purpose, starting with the inner-most loops, we convert the loops into groups of recursive instructions, propagate them to the loop header of the immediately enclosing loop and apply the procedure repeatedly until all cyclic code is converted into recursive instruction form, eventually leading to a single block for the procedure.

5.1 ϕ-Nodes in Loop Header

Although any code movement within a given loop can be carried out using the acyclic code motion techniques, the ϕ-nodes in the loop header cannot be eliminated using the techniques developed for acyclic regions. Instead, we adopt the executable function from [13]:

Definition 8. *We define the gating function* $\mu(a^{init}, a^{iter})$ *as an executable function.* a^{init} *represents external definitions that can reach the loop header prior to the first iteration.* a^{iter} *represents internal definitions that can reach loop header from within the loop following an iteration.* a^{init} *is returned when control reaches loop header from outside of the loop.* a^{iter} *is returned in all subsequent iterations.*

5.2 Conversion of Loops into Instruction-Level Recursion

The conversion is achieved by following the following steps:

1. Identify a single-entry, multi-exit region where the entire region is dominated by an inner-most loop header.
2. Propagate all instructions except branches to the loop header using acyclic code motion discussed before.
3. Calculate the controlling predicates for the exit edges and calculate the *Recursive Predicate* using *Algorithm 1* shown in Figure 8.

Algorithm 1
for each back-edge and exit edge e do

begin
  let b be the block which e originates from
  let I be any instruction originally in block b
  let $\alpha(I)$ be the predicate expression guarding I
  if block b has a branch on predicate P then
    if e is on the true path of the branch then
      $p(e) \leftarrow \alpha(I) \wedge P$ if e is a back-edge
      $q(e) \leftarrow \alpha(I) \wedge P$ if e is an exit edge
    else
      $p(e) \leftarrow \alpha(I) \wedge \neg P$ if e is a back-edge
      $q(e) \leftarrow \alpha(I) \wedge \neg P$ if e is an exit edge
    end
  else /* e is fall through backedge */
    $p(e) \leftarrow \alpha(I)$ if e is a back-edge
  end
  if e is a back edge then
    $RP \leftarrow p(e)$
end

Fig. 8. Algorithm 1: Compute RecursivePredicate and ExitPredicate

4. Pick an unused SSA name for the *RecursivePredicate*.
5. Convert ϕ-nodes to gating function μ.
6. Insert $(RP)[T]RP = ...$ at the very beginning of loop header where RP is the SSA name picked in the previous step and it is assigned to the computed *RecursivePredicate* by converting all the predicate variables in the computed predicate to future form.
7. Convert every instruction in the header to recursive form using RP and delete the back edges and branches. The conversion involves renaming all instructions which are in the loop body such that each SSA name that is defined in the block is appended the loop nest level, starting with zero at the inner-loop and incrementing. This renaming will update any uses which are loop carried to the new name while keeping names which are defined outside the loop unchanged.

Once the above process is completed, an inner-most loop has been converted to a sequential code. We apply the above process until the entire procedure is converted into a single block.

Theorem 2. *The predicate expression controlling the backedge e can be computed correctly using Algorithm 1.*

Proof. Figure 9 that contains an arbitrary innermost loop is used to demonstrate the proof. $B1$ is the loop header, $e1$ is a backedge originating from block $B3$ which

contains instruction J. Assume a trivial instruction K is inserted in $e1$ as shown in Figure 9(b). The predicate expression controlling K, namely γ is the same as the one controlling $e1$. γ is computed by propagating instruction K to $B1$. For that purpose, K is first moved into block $B3$. K becomes $\beta : K$ in $B3$ where three cases may happen:

case1: $\beta = P$ if $e1$ is the taken edge of $B3$,
case2: $\beta = \neg P$ if $e1$ is the fall through edge of $B3$,
case3: $\beta = true$ which means K is not guarded by any predicate if $B3$ is ended with an unconditional jump.

Propagate $\beta : K$ and instruction J to $B1$. Since $\beta : K$ and J propagate from the same block, the predicates guarding these two instructions are the same when they reach $B1$. Assume J becomes $\alpha : J$ in $B1$, then K becomes $\alpha : \{\beta : K\}$. Combining nested predication yields $\gamma = \alpha \wedge \beta$.

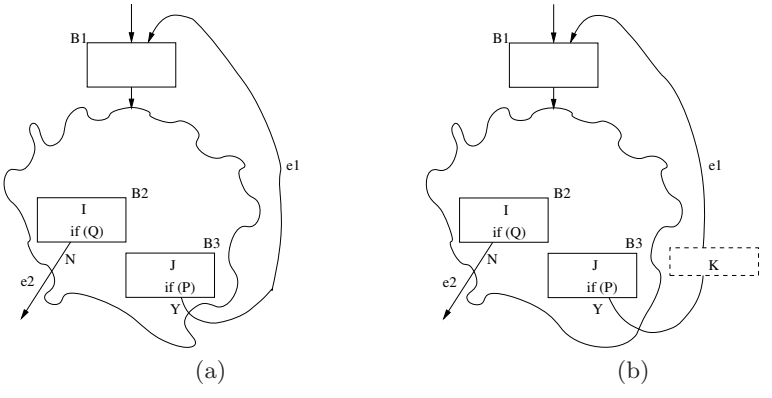
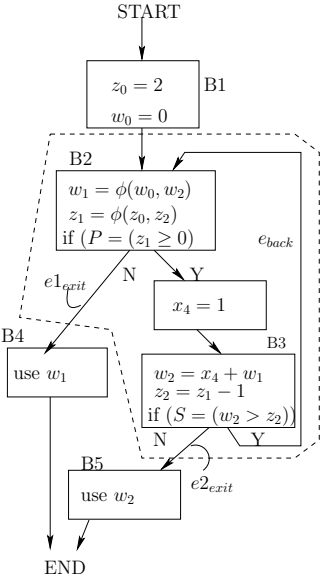


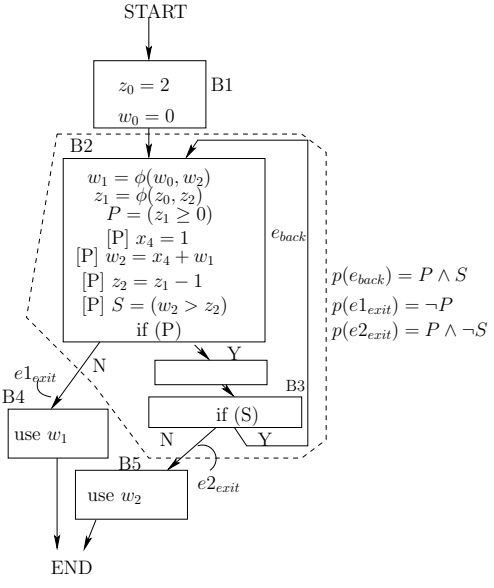
Fig. 9. Theorem 2

Note that although K may be split into multiple copies during the propagation, the last copy instruction is merged and hence the resulting instruction is renamed back to K in the loop header $B1$ if it is not merged before reaching $B1$.

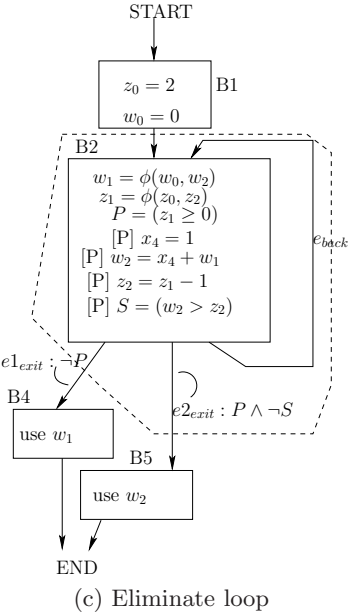
Figure 10(a) is an example that shows the steps of transforming cyclic code. The region cut out is a loop region with a single loop header $B2$. Following the algorithm, we first propagate every instruction inside the loop into the loop header(Figure 10(b)). During the instruction propagation, the necessary predicate information to compute the *RecursivePredicate* and controlling predicates for the exit edges are collected naturally, shown on the right side of Figure 10(b). Next, everything in the loop region except the loop header and the back edge is deleted.(Figure 10(c)). The result of the conversion is shown in Figure 10(d).



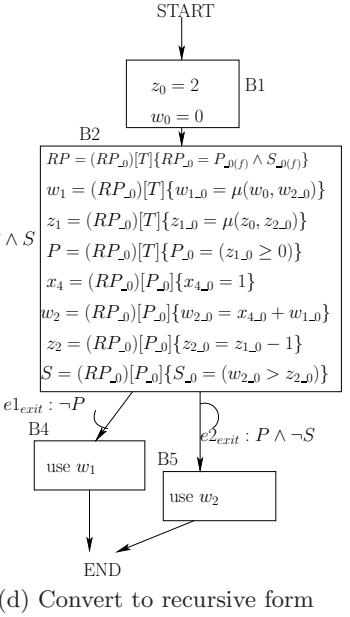
(a) A single-entry loop



(b) Apply acyclic code motion and compute *RecursivePredicate*



(c) Eliminate loop except loop header



(d) Convert to recursive form

Fig. 10. Program 1: Conversion of a cyclic program into RFPF

6 Code Motion Involving Memory Dependencies and Function Calls

Memory dependencies pose significant challenges in code motion. There are many cases a compile time analysis of memory references does not yield precise answers. Our solution is to assume dependence and enforce the original memory ordering in the program through predication. Since a series of consecutive load operations without intervening stores have no dependence on each other, RFPF allows these loads to be executed in any order once the dependence of the first load in the series is satisfied. We define the memory operations as: `MEM, @P` where MEM represents a Load/Store operation and P is a predicate whose value is set to 1 when the memory operation MEM gets executed. Any memory operation that has a dependence with MEM will be guarded by P as a predicated operation. In this way, the dependence among memory operations are converted into data dependencies explicitly. Once the memory operations are converted in this manner, they can be moved like any other instruction. Because of the predication, if a memory operation is hoisted above another which defines its controlling predicate, the controlling predicate becomes a future value (Figure 11).

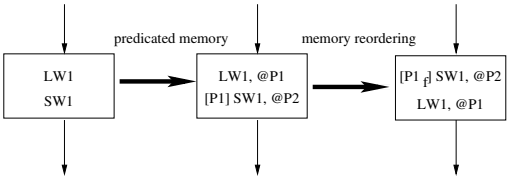


Fig. 11. Predicated memory and reordered memory

Our algorithm to rewrite memory operations is based on Cytron et al's SSA construction algorithm [7]. Since all the load/store operations can be treated as assignments to the same variable, Cytron et al's algorithm can be modified to accomplish the rewriting. Due to lack of space, we are unable to include the algorithm.

We employ a similar algorithm for handling function calls. Because of their side effects such as input/output, function calls may not be reordered without a proper analysis of the functions referenced. Therefore, we introduce a single predicate for each call instruction which is set when the call is executed. A single ϕ node is needed at merge steps to enforce the function call order on any path.

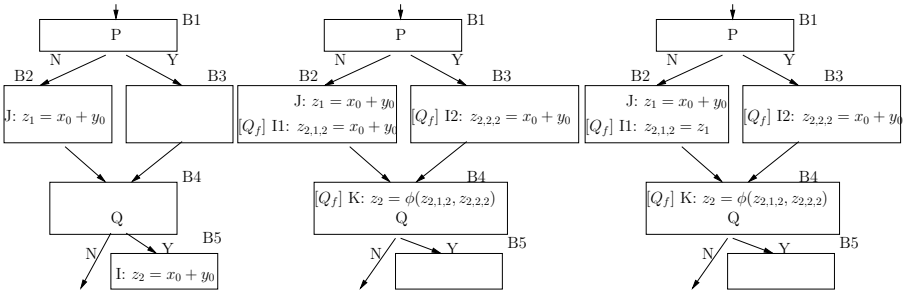
7 Optimizations Using RFPF

Many optimizations can be carried out on the *complete RFPF* and as well as during the transformation process. One of the advantages of RFPF is its ability to perform traditional optimizations while keeping the graph in single-assignment

form with minimal book keeping. We show two examples of optimizations, one which can be employed during the transformation and another after the graph is converted into full RPFP.

Case study 1. *PRE during the transformation:*

Consider Figure 12(a). There's a redundant computation of x_0+y_0 along the path (B2 B4 B5). Most PRE algorithms cannot capture this redundancy because node B4 destroys the available information for $x_0 + y_0$. On the other hand, instruction propagation and RFPF cover the case. Observe that during the instruction propagation, one of the clones, namely, ($I1$) reaches node B2(Figure 12(b)). By applying *Value numbering* [1] in the basic block, $x_0 + y_0$ in I1 is subsumed by z_1(Figure 12(c)).



(a) A PRE example (b) Code motion (c) *Value numbering*

Fig. 12. Partial redundancy elimination during the code motion

By further propagating and merging, instruction I1 and I2 are merged in B1 with the addition of the gating function ψ (Figure 13(a)) yielding the complete RFPF:

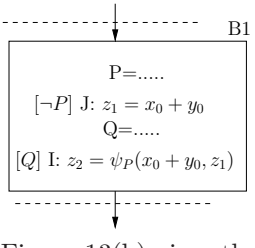
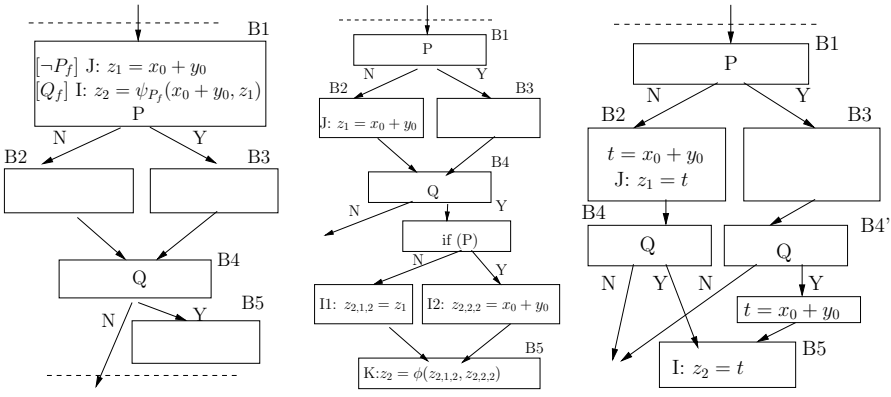


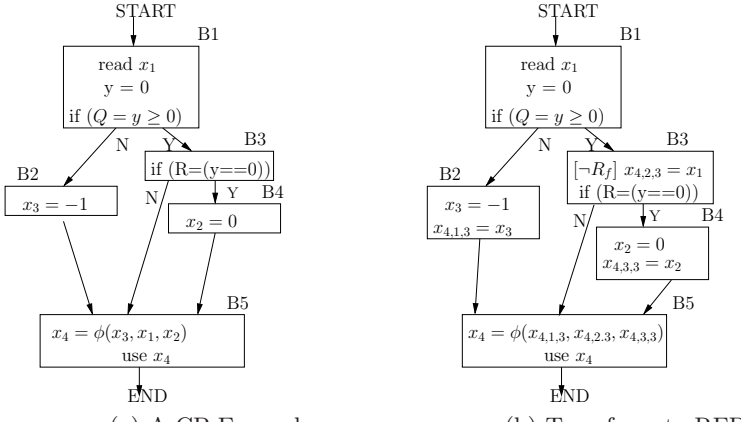
Figure 13(b) gives the result of transforming RFPF back into SSA using the algorithm in Section 8. This graph is functionally equivalent to Figure 13(c), which shows the result by using the PRE algorithm of Bodik et al. [4]. This algorithm separates the expression available path from the unavailable path by node cloning which eliminates all redundancies. As it can be seen, RFPF can perform PRE and keep the resulting representation in the SSA form.

The dependency elimination in our example is not a coincidence. By splitting instructions into copies, we naturally split the dataflow information available



(a) Instruction merging (b) RFPF to CFG (c) Bodik et al., [4]

Fig. 13. Merging and Converting Back to CFG



(a) A CP Example (b) Transform to RFPF

read x_1
$y = 0$
$Q = (y \geq 0)$
$Q : R = (y == 0)$
$Q \wedge R : x_2 = 0$
$\neg Q : x_3 = -1$
$x_4 = \psi_Q(\psi_R(x_2, x_1), x_3)$
use x_4

(c) Complete RFPF

read x_1
$y = 0$
$Q = true$
$true : R = true$
$true : x_2 = 0$
$false : x_3 = -1$
$x_4 = \psi_{true}(\psi_{true}(0, x_1), x_3)$
use $0(x_4)$

(d) Apply CCP

Fig. 14. Constant propagation on RFPF

path from unavailable path. From the perspective of the total number of the computations, RPFP yields essentially the same result. The optimality of RPFP and code motion based PRE in RFPF is yet to be studied, but its ability to catch difficult PRE cases is quite promising.

Case study 2. *Constant propagation in* complete RFPF*:*

We use another example(Figure 14(a)) to show how to do constant propagation(CP) in *complete RFPF*. As in the PRE example, constant propagation chances are caught in node B2 and B4(Figure 14(b)). Figure 14(c) and (d) shows complete RFPF of the program and the result after optimization. We use the conditional constant propagation(CCP) approach described in [18]. Note that x_4 becomes a constant in our representation because gating function ψ can be evaluated given the constant information of the predicate and the variable values.

The choice of applying various optimizations during or after the transformation has to be decided based on foreseen benefits. This is an open research problem and it's a part of our future work.

8 Algorithms for Converting RFPF Back to CFG

The inverse transformation algorithms are necessary because the existing algorithms can be applied on CFG for further optimizations and to produce machine code. In different stages of compilation, the conversion algorithms have different goals. Before scheduling, the goal is to minimize number of nodes in the resulting CFG. After scheduling, the goal is to maximize the issue rate on the resulting CFG. At the register allocation stage, the goal is to minimize live range of variables in the resulting CFG. So we must take into account different optimality criteria for different conversion stages. The basic algorithm to transform RFPF back to CFG consists of three steps:

1. Reorder RFPF in a way that no future values occur by pushing-down or moving-up instructions, which forms an initial instruction list.
2. Group instructions with identical predicates together. Such grouping reduces the multiple node insertions for a branch condition and forms the loop structures.
3. Iterate through the instruction list and insert instructions one by one into corresponding basic blocks.

The optimality of the resulting graph is dependent on how the predicate expressions are analyzed and combined. A complete inverse transformation framework is part of our future work.

9 Related Work

Intermediate program representation design has always been a very important topic for optimizing compiler research since the choice of program representation affects significantly the design and complexity of optimization algorithms. Some of the most relevant to this work are the control flow graph [1], def-use chains [1], program dependence graph [8], static single assignment(SSA) [7,3], and the

program dependence web [13]. We directly use these prior art in this paper. The SSA form as well as the gating functions that the program dependence web proposes are significant for correct translation of programs into RFPF. The dependence flow graph [14] contributed to our thinking in designing the representation.

Allen et al. proposed the idea of isomorphic control transformation(ICT) [2] which converts the control dependencies into data dependencies. This idea forms the basis of hyperblock formation in many techniques, including ours as well as others [11]. Warter et al. [17] proposes a technique which uses ICT and apply local scheduling techniques on the hyperblock and then transforms the scheduled code back to CFG representation. RFPF follows a similar, but a more comprehensive path.

Partial redundancy elimination(PRE) proposed by Morel and Renvoise [12] is a powerful optimization technique which is usually carried out using code motion [9,10]. As it is well known, code motion alone cannot completely eliminate partial redundancies. Click proposed an approach using global value numbering supported by code motion is proposed to eliminate redundancies [6]. This approach may insert extra computations along some path. Bodik et al. [4], give an algorithm based on the integration of code motion and CFG restructuring which achieves the complete removal of partial redundancies. Chow et al. [5] proposes a similar PRE algorithm for SSA yielding similar optimality to lazy code motion. The algorithm maintains its output in the same SSA form. VanDrunen and Hosking [16] present a structurally similar PRE for SSA covering more cases. Control flow obfuscate data-flow information needed by many optimization algorithms. Thakur and Govindarajan [15] proposes a framework to find out the merge region in a CFG which prevents the data-flow analysis, and restructure the CFG to make data-flow analysis more accurate. Our technique of instruction propagation and merging exposes similar opportunities.

10 Conclusion

We have presented a new approach to program representation and optimization. The most significant difference of our approach is to move instructions to collect the necessary data and control flow information, and in the process yield a representation in which compiler optimizations can be carried out. Our future work involves transformation and adaptation of state-of-the-art optimization algorithms into the new framework.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston (1986)
2. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: POPL 1983: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 177–189. ACM, New York (1983)

3. Bilardi, G., Pingali, K.: Algorithms for computing the static single assignment form. J. ACM 50(3), 375–425 (2003)
4. Bodík, R., Gupta, R., Soffa, M.L.: Complete removal of redundant expressions. In: PLDI 1998: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, pp. 1–14. ACM, New York (1998)
5. Chow, F., Chan, S., Kennedy, R., Liu, S.M., Lo, R., Tu, P.: A new algorithm for partial redundancy elimination based on ssa form. In: PLDI 1997: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, pp. 273–286. ACM, New York (1997)
6. Click, C.: Global code motion/global value numbering. SIGPLAN Not. 30(6), 246–257 (1995)
7. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
8. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
9. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: PLDI 1992: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation, pp. 224–234. ACM, New York (1992)
10. Knoop, J., Rüthing, O., Steffen, B.: Optimal code motion: theory and practice. ACM Trans. Program. Lang. Syst. 16(4), 1117–1155 (1994)
11. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture, pp. 45–54. IEEE Computer Society Press, Los Alamitos (1992)
12. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM 22(2), 96–103 (1979)
13. Ottenstein, K.J., Ballance, R.A., MacCabe, A.B.: The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. SIGPLAN Not. 25(6), 257–271 (1990)
14. Pingali, K., Beck, M., Johnson, R.C., Moudgill, M., Stodghill, P.: Dependence flow graphs: An algebraic approach to program dependencies. Tech. rep., Cornell University, Ithaca, NY, USA (1990)
15. Thakur, A., Govindarajan, R.: Comprehensive path-sensitive data-flow analysis. In: CGO 2008: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization, pp. 55–63. ACM, New York (2008)
16. VanDrunen, T., Hosking, A.L.: Anticipation-based partial redundancy elimination for static single assignment form. Softw. Pract. Exper. 34(15), 1413–1439 (2004)
17. Warter, N.J., Mahlke, S.A., Hwu, W.M.W., Rau, B.R.: Reverse if-conversion. In: PLDI 1993: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, pp. 290–299. ACM, New York (1993)
18. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13(2), 181–210 (1991)