International Journal of Parallel Programming, Vol. 25, No. 3, 1997

Techniques for Critical Path Reduction of Scalar Programs

Michael Schlansker[1] and Vinod Kathail[1]

Scalar performance on processors with instruction level parallelism (ILP) is often limited by control and data dependences. This paper describes a family of compiler techniques, called Critical Path Reduction (CPR) techniques, which reduce the length of critical paths through control and data dependences. Control CPR reduces the number of branches on the critical path and improves the performance of branch intensive codes on processors with inadequate branch throughput or excessive branch latency. Data CPR reduces the number of arithmetic operations on the critical path. Optimization and scheduling are adapted to support CPR.

KEY WORDS: Critical path reduction; control height reduction; data height reduction; blocked control substituting; instruction level parallelism.

1. INTRODUCTION

Critical paths through control and data dependences in scalar programs limit performance on processors with Instruction-Level Parallelism (ILP). Performance limits caused by critical paths in a program can be avoided using transformations which reduce the height of critical paths. Critical Path Reduction (CPR) represents a collection of techniques specifically designed to reduce dependence height in program graphs. CPR provides three main benefits when a program follows a predicted path through its control flow graph: it decreases the dependence height of program critical paths; it reduces the number of operations which must be executed; and it improves the scheduler's freedom in scheduling operations. This paper presents a systematic approach for obtaining height-reduced and optimized code for branch-intensive scalar programs with predictable control flow.

[1] Hewlett-Packard Laboratories, Palo Alto, California 94304.

0885-7458/97/0600-0147$12.50/0 © 1997 Plenum Publishing Corporation

828/25/3-1

The paper represents a starting point in understanding CPR for scalar programs. Programs contain a mix of control and data dependences, and CPR techniques are needed for both. Further work is needed to adapt these techniques to more general scalar requirements and to quantify the benefits of CPR.

Data dependences limit program performance when sequentially chained arithmetic operations are executed on processors with substantial ILP. Data CPR uses properties such as the associative property in order to re-organize code and improve performance. Data CPR must be applied with careful attention to both critical path height and operation count.

Control dependences limit performance when executing branch intensive code. There is little previous work in developing optimizations for branch intensive code which alleviate performance limits due to control dependence. This paper develops a theory of control CPR which explains how branch intensive code can be reorganized and shows the benefits using specific examples. Control CPR decreases the height of critical paths due to control dependences and reduces the amount of computation by moving rarely taken branches off-trace.

Complex actions on programs, such as CPR or scheduling, are best performed on modest sized regions selected from a program. A region consists of a subset of the control flow graph of the program which has been selected to maximize subsequent opportunities in optimization and scheduling.[1] Previous ILP compilers have identified the loop body as key region for optimization and scheduling.[2] The loop body is scheduled using software pipelining which overlaps multiple iterations of the loop. In previous work, we have shown the applicability of data and control CPR to loops.[3,4] In this work, we extend data and control CPR to the scalar case.

A "trace"[5,6] has been used as a scheduling region to enhance available parallelism in scalar programs (loops can be treated by unrolling). When program control flow branches from a basic block to a most probable subsequent block, the flow follows the trace path, or remains on-trace. Traces are selected using branch profile statistics. Newly formed traces can branch into the middle of previously formed traces. Thus, in general, traces are linear program regions with multiple entries and multiple exits.

The superblock[7] is similar to a trace except that branches are not allowed into the middle of previously formed superblocks. Each superblock has a single entry with potentially multiple exits. The single entry property for superblocks is maintained using tail duplication. Whenever a branch within a newly formed superblock jumps into the middle of a previous superblock, necessary code within the previous superblock is replicated into the new superblock. The process of tail duplication systematically

eliminates program merges from superblocks and simplifies the engineering of ILP compilers.

This paper assumes that predominate program flow paths can be identified [8] after which superblocks (or hyperblocks) are formed, optimized and scheduled. CPR techniques are presented which expedite the path from the region entry to a primary or on-trace exit. Computation required on-trace is carefully separated from computation required off-trace. After computation required to reach the primary exit is isolated, other computations which were traditionally trapped on-trace can now move off-trace.

Predicated execution, as supported in PlayDoh,[9] has been historically used to eliminate branches using if-conversion.[10,11] If-conversion by itself can reduce dependence height, for example, by allowing the parallel execution of a sequence of in-then-else constructs. Superblocks have been enhanced to allow if-conversion within a region called a hyperblock.[12]

We illustrate a rather different use of predicates. For each basic block in the control flow graph of a region, we calculate a fully-resolved predicate (FRP). Each FRP is computed using a Boolean-valued expression which is evaluated when flow of control traverses the region. Intuitively, the FRP for any block is a boolean expression of branch conditions describing the exact condition, relative to region entry, under which the block executes. The FRP for any block is true if the program's execution on this entry to the region traverses that block; otherwise, the FRP for that block is false. The computation of FRPs is accelerated using data CPR and with hardware features provided by PlayDoh.

When an operation is guarded using its FRP it can be moved upward across preceding branches within the region during scheduling. The operation continues to execute under conditions which are faithful to the original program due to the guarding action of its fully resolved predicate. The scheduler's freedom in placing operations is enhanced by allowing flexible motion of operations (including branches) across previous branches. Superblock scheduling is adapted to support CPR. Operations which are not necessary on-trace naturally move off-trace. However, code moved off-trace is kept local to the scheduling region. Operations which are required for less important (off-trace) exits can fill in unused space within the on-trace schedule. The approach simplifies compiler engineering because code generation and optimization are decoupled from scheduling.

The rest of the paper is organized as follows. Section 2 presents architectural assumptions upon which examples are based. Section 3 presents the principles of control CPR in superblocks. Section 4 presents a scheduling approach adapted to take advantage of CPR. Section 5 discusses other transformations that must be applied after CPR to get the best code quality. Section 6 provides a detailed example. Section 7 presents more

general CPR techniques for the treatment of single entry acyclic regions. Section 8 discusses CPR for processor architectures without predicated execution. Section 9 discusses related work, and Section 10 contains concluding remarks.

2. ARCHITECTURAL ASSUMPTIONS

This paper uses the HPL PlayDoh architecture[9] to explain CPR concepts. PlayDoh supports predicated execution of operations. Predication support in PlayDoh is an enhanced version of the predication capabilities provided by the Cydra 5 processor.[11, 13] Predicated execution uses a Boolean data operand to guard an operation. For example, the generic operation "$r1 = op(r2, r3)$ if $p1$" executes when $p1$ is true and is nullified when $p1$ is false. A key constraint here is that if an operation is nullified, it is as if the operation never executed. This could, for example, be carried out by executing the operation and nullifying only the write-back into the register file as well as any possible exceptions. Omitting the predicate specifier for an operation is equivalent to executing the operation using the constant predicate true (unconditionally executing the operation).

PlayDoh introduces a family of compare-to-predicate operations which are designed to efficiently support the computation of predicates. Two predicate targets can be computed within a single compare operation. The two targets are often used to compute predicates corresponding to the taken and not-taken sense of a branch allowing operations from either block subsequent to the branch to be moved prior to the branch. PlayDoh also introduces compare operations which support the parallel computation of high fan-in logical operations. Examples within this paper primarily use the unconditional and wired-and compare forms which are described later. The general form of a compare operation appears as:

$$p1, p2 = \text{CMPP}.\langle D1\text{-action}\rangle.\langle D2\text{-action}\rangle(r1 \langle \text{cond} \rangle\ r2) \text{ if } p3$$

The operation is interpreted as follows:

— $p1, p2$: target predicates set by the operation;

— CMPP: generic compare op-code;

— ⟨$D1$-action⟩, ⟨$D2$-action⟩: actions for the two targets;

— $r1, r2$: data operands to be compared;

— ⟨cond⟩: compare condition;

— $p3$: predicate input.

A single target compare is specified by omitting one target operand and one target action.

The allowed compare conditions exactly parallel those provided by the HP PA-RISC architecture. These include "=," "<," "< =," and other traditional tests on data. The Boolean result of a comparison is called its compare value. The compare value is used in combination with the target action to determine the target predicate result.

The actions allowed on each target predicate are as follows: unconditionally set (UN or UC), conditionally set (CN or CC), wired-or (ON or OC), and wired-and (AN or AC). The second character (N or C) indicates whether the compare value is used in "normal mode" (N), or "complemented mode" (C). In complemented mode, the compare value is complemented before performing an action on the target predicate.

Figure 1 provides a predicate result table for four combinations of input predicate and compare value as specified by the four rows below the horizontal double line, and eight target actions as specified by the eight columns to the right of the vertical double line. Each cell within the table specifies a result corresponding to an input combination indicated by its row, and an action indicated by its column. A cell's row is identified in the two columns to the left of the double lines which specify a choice of input predicate and compare value. A cell's column is identified immediately above the double lines where a choice of destination action is specified. The cell specifies one of three results for the target predicate: set to 0, set to 1, or leave untouched (shown as " – ").

The wired-and action (AN) is used to execute high fan-in AND operations. A careful inspection of Fig. 1 shows that when the AN specifier is used the predicate result is set to false exactly when its input predicate is true and its compare value is false. In all other cases, the result is left unchanged. A wired-and is accomplished as follows: (1) initialize the predicate result register to true; (2) execute n compare operations in parallel or in arbitrary order, each of which uses AN action (or AC if the condition is to be complemented) to conditionally set the common

input	compare	On result				On complement			
predicate	value	UN	CN	ON	AN	UC	CC	OC	AC
0	0	0	-	-	-	0	-	-	-
0	1	0	-	-	-	0	-	-	-
1	0	0	0	-	0	1	1	1	-
1	1	1	1	1	-	0	0	-	0

Fig. 1. Result table for compare-to-predicate operations.

predicate result to false. After all compares have executed, the conjunction is available in the predicate result. The "wired-or" uses a similar approach in which the result is initialized to false and then conditionally set to true.

As an example, assume that data values $i0, i1, i2$ and $i3$ are to be tested to see if all four are equal to zero, and the Boolean result is to be placed in r. The result is computed with the following code sequence:

$$r = \text{TRUE};$$
$$r = \text{CMPP.AN}(i0 = 0); \qquad r = \text{CMPP.AN}(i1 = 0);$$
$$r = \text{CMPP.AN}(i2 = 0); \qquad r = \text{CMPP.AN}(i3 = 0);$$

The first assignment initializes the result to true and precedes the wired-and compares. The predicate input for each compare is true and is omitted in the code. The wired-and compares can execute in any order or in parallel since each conditionally clears the result if the test for equality fails. When multiple compares clear the result in the same cycle, the multiported register hardware must ensure that the result is in fact cleared. We will sometimes denote such a sequence for computing AND by a high-level macro written as follows:

$$r = \text{AND}(i0 = 0,\ i1 = 0,\ i2 = 0,\ i3 = 0)$$

The use of wired-and compares provides two benefits. It allows constituent compares to be re-ordered during scheduling, and it allows the retirement of multiple terms in the conjunction within a single cycle.

PlayDoh supports multiple branches in a single cycle, but does not support dependent parallel branches; that is, when multiple branches take in the same execution cycle, the semantics is undefined. However, compare operations can be used to compute mutually exclusive branch conditions so that independent branches execute either simultaneously or in an overlapped manner.

3. CONTROL CPR IN SUPERBLOCKS

This section introduces control CPR for superblock scheduling. Each superblock has a number of unlikely (secondary) exits and a single most probable (primary) exit. Control CPR reduces the dependence height through the critical path as well as the number of operations required to reach the primary exit. The critical path's dependence height is based on individual dependences or precedence constraints which are defined in the context of a scheduling model. We begin with a discussion of conventional superblock scheduling. By introducing fully-resolved predicates, we relax

normal dependence constraints and reduce the critical path length to the primary exit. One or more fall through branches are introduced in order to allow the motion of code off trace. The improved motion of branches across branches allows code, including secondary exit branches as well as other noncritical operations, to move below the primary exit and off-trace. In addition, the use of fully-resolved predicates converts branch dependences into data dependences whose height can be reduced using data CPR. Additional techniques are presented to reduce the number of redundant operations which are introduced when using data CPR.

3.1. Branch Dependences

During scheduling, branches may impose restrictions on code-motion. The precise definition of the restrictions depends upon the scheduling strategy and the code generation schema. This section discusses the code-motion restrictions imposed by branches in superblocks.

An example superblock is shown in Fig. 2. The superblock consists of three basic blocks, each of which contains the following: some number of instructions (denoted by ⟨block n body⟩), a compare operation to calculate a branch condition, and a branch operation. All operations are within their original basic block and are executed using true predicate. The code uses PlayDoh compare operations to compute conventional branch conditions. For example, the compare operation in basic block 0 calculates the Boolean condition $x0 = y0$ and stores the result in $e1$.

The restrictions imposed by branches are defined using a dependence graph. Edges in the dependence graph describe data dependences as well as any scheduling constraints due to branches. Data dependences are conventional flow, anti, and output dependences between operations. Edges that

```
 <block 0 body> if T;          /* Basic block 0 */
 e1 =CMPP.UN(x0=y0) if T;
 branch E1 if e1;
 <block 1 body> if T;          /* Basic block 1*/
 e2 =CMPP.UN(x1=y1) if T;
 branch E2 if e2;
 <block 2 body> if T;          /* Basic block 2*/
 e3=CMPP.UN(x2=y2) if T;
 branch E3 if e3;

E4:  /* fall-through code*/
```

Fig. 2. Example Superblock.

represent scheduling constraints due to branches will be called **branch dependences**. Branch dependences will first be used to enforce the rules of code motion traditionally used in superblock scheduling.

Figure 3a shows various types of edges to and from a branch. A branch has a number of properties: it uses a branch condition, and it transfers flow of control. As a user of a branch condition, it has a flow dependence edge from an operation that generates the condition, and it may have a anti-dependence edge to an operation which over-writes the condition. These traditional data dependence edges are shown as solid edges between branches and other operations. Dotted lines represent branch dependences. Branch dependences maintain order among branches or between branches and other operations as needed to support the scheduling scheme.

Speculative execution can be used to move operations above branches, and exceptions from speculative operations can be ignored with proper hardware support.[9, 14] However, some operations cannot be executed speculatively without disrupting program semantics. Within this discussion, operations which write to a location in memory are nonspeculative. Operations which write to a register which is live out at a previous exit branch are also considered nonspeculative. Other operations can be executed speculatively. In Fig. 3a, "live-out anti" edge from a branch to a side-effecting operation ensures that the live-outs and memory are not overwritten before the branch takes.

Superblock scheduling avoids compensation code generation in order to simplify compiler engineering. This paper also assumes that compensation code is kept local to the current scheduling unit. If an operation calculates a value that is live-out at a branch, then it is not allowed to

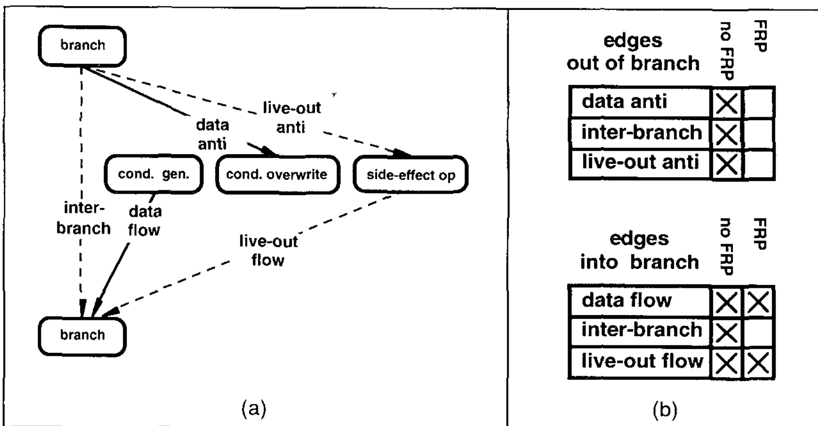


Fig. 3. Branch dependences.

move below the branch. In other words, all live-out values are calculated before exiting the region. Similarly, stores are not allowed to move below a branch. The "live-out flow" edge from a side-effecting operation to branch ensures that these conditions are satisfied.

Lastly, the "inter-branch" edge from branch to branch ensures correct branch order. Branch conditions in conventional superblock code don't take the effect of previous branches into account. Consider, for example, the second branch in Fig. 2. It should branch to $E2$ only if the first branch falls through and its compare value $(x1 = y1)$ is true. The second branch's compare value alone is not sufficient to decide whether the second branch takes. Assume that the computation of the compare value $(x1 = y1)$ is speculatively moved above the first branch. Also, assume that compare values for both the first and the second exit branch are true. The program should branch to $E1$ and not to $E2$ even though the compare value for the second branch is true. If we naively interchange branches, an incorrect program results. Branches can be re-ordered using the approach described in Ref. 15. However, the approach requires compensation code and may not help reduce the critical path. Section 9 discusses this further.

In Fig. 3b, two tables are provided which describe the necessary edges both out of and into each branch within a superblock. The rows of each table are labeled within the left hand column and indicate the type of edge under consideration. The columns of each table indicate the scheduling model under consideration with the "no FRP" scheduling model summarizing branch dependences for conventional superblock code. The column marked "FRP" will be discussed in Section 3.2. Each cell within the table is either marked with an X indicating that the dependence must be considered to generate correct code or the cell is blank indicating that the dependence can be ignored.

The relevant parts of the dependence graph for the superblock example are shown in Fig. 4. To simplify the presentation, we focus on store operations in each basic block. The dependence graph shows that branches are ordered and stores are trapped between branches when the no FRP or conventional scheduling model is employed.

3.2. Fully-Resolved Predicates

Given a single entry acyclic region within a control flow graph, a **fully-resolved predicate (FRP)**[2] can be defined for every basic block within the region and for every control flow edge within or exiting the region. The FRP for any block (edge) is true only if the program's control flow path

[2] Fully-resolved predicates were called fully-qualified predicates in Ref. 4.

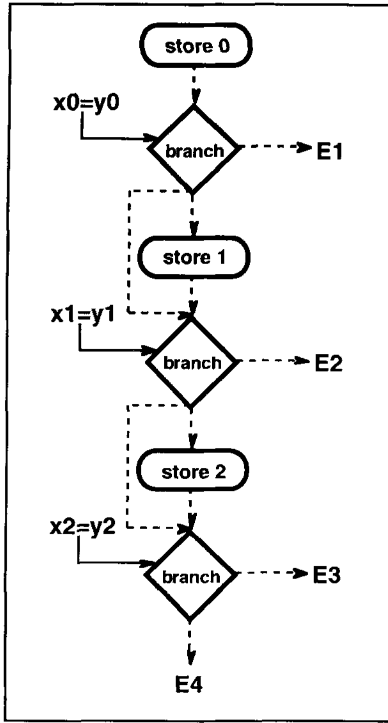


Fig. 4. Dependence graph for superblock.

on this entry to the region traverses that block (edge); otherwise, the FRP for that block (edge) is false. The use of an FRP allows an action to be correctly guarded using predicates and without relying on control flow. Because FRPs can guard operations without relying on control flow, they can be used to liberalize the rules of code motion. Block FRPs are used to predicate operations which can move upward across one or more previous branches. Speculatively executed operations are guarded by predicates other than their block FRPs (e.g., unconditionally executed when guarded using predicate true). On the other hand, nonspeculative operations such as stores and live-out overwrites are guarded using their block's FRP. Edge FRPs are used to predicate branches which are always nonspeculative and also can move upward across previous branches.

For superblocks, FRPs are defined as follows: The FRP for the entry block is defined to be true. The FRP for any current block (except the entry) is the conjunction of the FRP for the preceding block and the fall-through condition for the branch which reaches the current block. The FRP for each exit edge is the conjunction of the FRP for the block in

which the branch resides and the branch condition under which the branch is taken. Note that the FRP for each block takes into account the entire sequence of branch conditions needed to reach that block from region entry.

In Fig. 5, an FRP is computed for every basic block and for every exit branch. FRPs for basic blocks are labeled $f0$, $f1$, $f2$, $f3$, and exit FRPs are labeled $e1$, $e2$, $e3$. Each block computes FRPs for its on-trace successor block and its exit branch using a single unconditional compare operation; for example, the compare in block 1 calculates two results as follows:

$$f2 = (x1 = y1) \wedge f1 \qquad \text{and} \qquad e2 = (!(x1 = y1)) \wedge f$$

Note that the FRP for the fall-through exit ($f3$) is not used because the superblock has no code after the last branch.

When FRPs are used within superblocks, branch dependences as presented in Fig. 3 can be partially relaxed. In Fig. 3b, the column marked "FRP" defines branch dependences for code using fully-resolved predicates. Again the presence of an X indicates that the dependence must be honored. Note that anti-dependence edges out of a branch as well as inter branch dependence edges between branches can now be ignored.

The dependence graph for the superblock code with FRPs is shown in Fig. 6. Each two target compare operation is shown as a pair of and gates which use the compare value in both true (for exit FRP), and complement (for fall-through FRP) forms. Again, for Fig. 6, speculative execution is not considered as indicated by showing only stores within the basic blocks. Data flow edges and live-out flow edges to a branch are enforced just as when predicates were not fully-resolved.

The use of FRPs eliminates data anti-dependence edges and live-out anti-dependence edges, because when a branch takes, subsequent FRP

```
    f0 =true;          /* FRP for block 0 is true */
    <block 0 body>  if f0;
    f1,e1 =CMPP.UC.UN(x0=y0)  if f0;
    branch E1 if e1;
    <block 1 body>  if f1;
    f2,e2 =CMPP.UC.UN(x1=y1)  if f1;
    branch E2 if e2;
    <block 2 body>  if f2;
    f3,e3 =CMPP.UC.UN(x2=y2)  if f2;
    branch E3 if e3;

E4:    /* fall-through code*/
```

Fig. 5. Superblock code with FRPs.

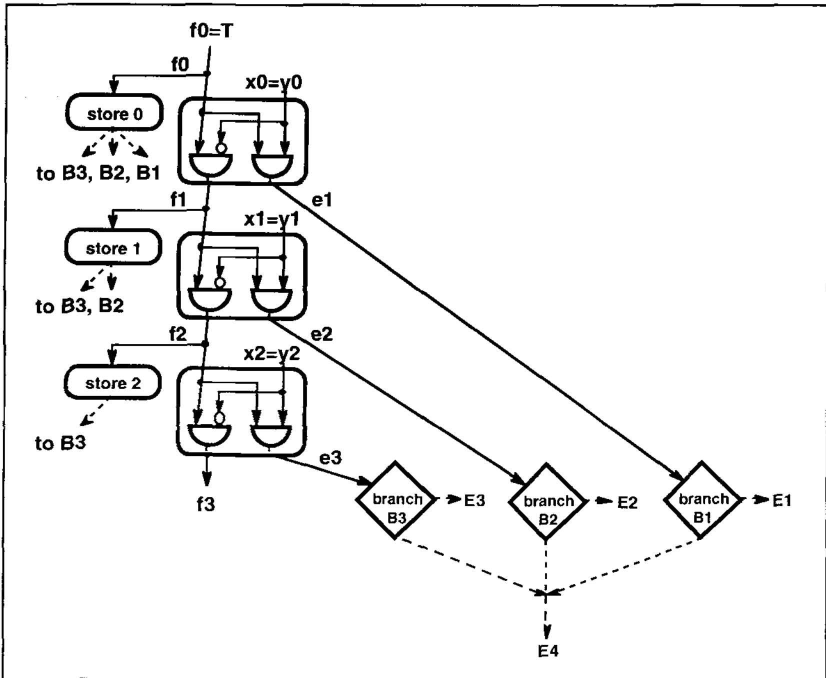


Fig. 6. Dependence graph for superblock with FRPs.

guarded anti-dependent operations do not execute even when moved above the branch. Thus, stores and assignments to live-outs are allowed to move above branches upon which they were anti-dependent in conventional code. Similarly, inter-branch dependence edges are not needed. On each entry into a superblock, only a single exit branch is taken. The use of FRPs as branch conditions ensures that branches are mutually exclusive. After, inter-branch edges are eliminated, branches can move across other branches without compensation code. Consider, for example, the exit to label *E*3. If the FRP for the branch to *E*3 is true, code at *E*3 may begin execution irrespective of previous branches. Exit branches guarded by FRPs may be scheduled simultaneously on PlayDoh because only one will take.

3.3. Fully Parallel Computation of FRPs

The use of FRPs allows the parallel execution of branches, but the computation of the FRPs themselves remains sequential. The FRP for each basic block is one AND operation removed from the previous FRP in the

sequence (see Fig. 6). The FRP computation can be performed in parallel by expressing an FRP as a multi-input AND of constituent branch conditions.

Figure 7 shows the code for computing all FRPs in parallel. FRPs corresponding to all interior basic blocks and exits are computed separately using a single wide AND macro operation. We call this the *fully parallel form* of the code. Each compare value (e.g., $xi = yi$ in Fig. 5) is now abbreviated as ci to simplify the presentation. The dependence graph for FRP computation in superblocks with full CPR is shown in Fig. 8.

The implementation of the wide AND operation varies from one processor architecture to another. In conventional architectures, a height-reduced tree of two input AND operations may be used. In PlayDoh, wired-and compares are used to further reduce the height of an FRP computation. Each AND macro operation is expanded into an initialization operation and subsequent wired-and compare operations as described earlier. Note that two-target compares allow block and exit predicates to be computed together, thus reducing the number of compares. For example, $f2$ and $e2$ can be computed together using two target compares.

The fully parallel form computes all FRPs by applying CPR separately to all paths using redundant computation. This requires $O(n^2)$ operations and is prohibitively expensive for processors with limited amounts of ILP.

3.4. Blocked Control Substitution to Compute FRPs

This paper uses an approach, called *blocked control substitution*, which reduces the amount of redundant computation. Blocked control substitution accelerates some on-trace FRPs while intervening FRPs are computed sequentially. The technique is an adaptation of the blocked back-substitution technique used for height-reduction of control recurrences in while loops.[4]

```
f0=true;
<block 0 body> if f0;
f1=!c0;
e1=c0;
branch E1 if e1;
<block 1 body> if f1;
f2=AND(!c0,!c1);
e2=AND(!c0,c1);
branch E2 if e2;
<block 2 body> if f2;
e3=AND(!c0,!c1,c2);
branch E3 if e3;
```

Fig. 7. Code for FRPs with full CPR.

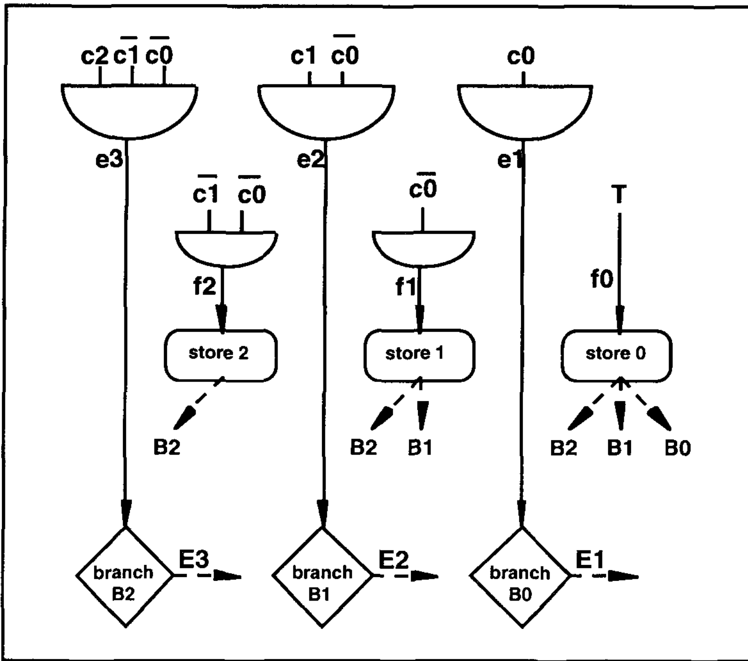


Fig. 8. Superblock graph with full CPR.

Blocked control substitution is shown in Fig. 9. After formation of a previous block, a heuristic is used to form a subsequent block by selecting a lookahead distance k. An expedited FRP, f_{i+k}, is evaluated directly from the *previous expedited FRP*, f_i, in a single wide AND operation. Intermediate FRPs are evaluated sequentially. The wide AND operation can be implemented in a number of ways; however, its implementation should minimize the path length from f_i to f_{i+k}. It can be implemented using two input AND operations by associating the tree of operations so that a single AND separates f_{i+k} from f_i. On PlayDoh, wired-and compares readily accommodate the late arrival of branch condition values and simplify interaction between code generation and scheduling. PlayDoh code to compute FRPs for blocked control substitution is shown in the right hand side of Fig. 9.

Blocked control substitution uses control CPR to expose parallelism and allows the degree of parallelism to be adjusted using the lookahead distance. When program traces are predictable, longer lookahead can be used to increase the parallelism. Blocked control substitution expedites an entire sequence of FRPs when using multiple stages of blocking. While

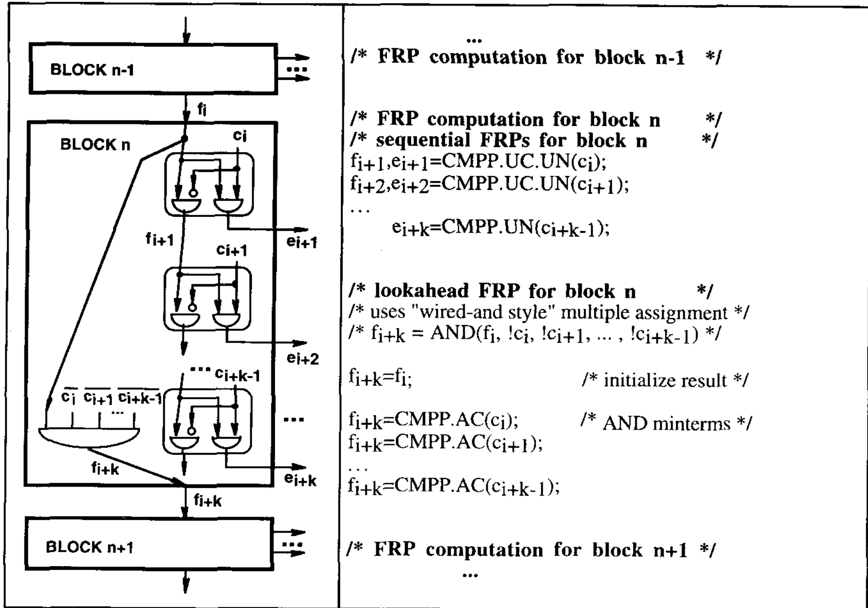


Fig. 9. Blocked control substitution.

nonlookahead FRPs are computed sequentially within each block, they benefit from CPR across previous blocks.

Sequential FRP evaluation uses n operations to traverse n branches. Fully parallel evaluation requires $O(n^2)$ operations. Blocked control substitution requires $2n$ operations or a factor of two in operation count over sequential evaluation. To expedite a superblock of length n, $n-1$ operations compute the sequential FRPs, and $n+1$ operations compute the lookahead FRP. When code is carefully organized as shown in Section 3.6, only the lookahead FRP is computed on-trace and FRP evaluation is irredundant.

3.5. On-trace CPR Using Fall-Through Branch

Consider the superblock in Fig. 5. Each execution of the superblock either takes an exit branch or falls through to the subsequent code (i.e., the code at $E4$). Up to this point, the treatment of fall-through path has differed from that of the other exits. The code at label $E4$ begins executing only after all the exit branches fall-through. In Fig. 6, this is shown by branch dependence edges from all exit branches to the code at label $E4$.

To examine the fall-through path in more detail, consider the code shown in Fig. 10a. The code is a version of the superblock in Fig. 5 in which block bodies have been replaced by assignments to l1, l2, l3, and l4. Assume that l1, l2, l3, and l4 are live-out on superblock exits $E1$, $E2$, $E3$, and $E4$, respectively. Much of the live-out computation can be done speculatively, and it may take a varying amount of time to compute each live-out. If predicates are fully-resolved, each branch can be scheduled as early as corresponding live-outs are available.

The fall-through path presents a special problem. Even when the FRP for the fall-through path (i.e., $f3$) can be quickly calculated, the fall through successor is not reached until all preceding exit branches fail. This problem arises due to the fact that in a conventional superblock with n exits (including the primary fall-through exit) there are only $n-1$ exit branches. In this situation, it can only be determined that the program reaches its final fall-through target after all $n-1$ exit branches fail to take. Thus, the fall through path accommodates live-out computations for all preceding exits. This interferes with on-trace CPR and requires that the fall-through schedule provide time to compute all live-outs.

The introduction of a fully-resolved branch, called a **fall-through branch**, allows all exits to be treated identically. Figure 10b shows the code after the introduction of a fall-through branch. The FRP for the fall-through branch ($f3$) is a conjunction of conditions which ensure that all exit branches fall-through (i.e., the superblock exits at the bottom). The evaluation of $f3$ can be expedited just as other FRPs were evaluated in Fig. 8. Now, if the live-out for the fall-through path can be calculated quickly, the fall-through branch is free to move above branches and live-out computations for preceding exits.

```
l1=...
f1,e1=CMPP.UC.UN(c0)
branch E1 if e1;
l2=...
f2,e2=CMPP.UC.UN(c1)
branch E2 if e2;
l3=...
l4=...
f3,e3=CMPP.UC.UN(c2)
branch E3 if e3;

E4: /* fall-through path*/
```

(a)

```
l1=...
f1,e1=CMPP.UC.UN(c0)
branch E1 if e1;
l2=...
f2,e2=CMPP.UC.UN(c1)
branch E2 if e2;
l3=...
l4=...
f3,e3=CMPP.UC.UN(c2)
branch E3 if e3;
branch E4 if f3;
E4: /* fall-through path*/
```

(b)

Fig. 10. Introducing a fall-through branch. (a) Code without fall-through branch; (b) code with fall-through branch.

3.6. Predicate Splitting

Using blocked-control substitution to expedite a lookahead predicate may not remove sequential FRP evaluation from the critical path. Consider the program graph of Fig. 11a. It shows the dependence graph for a superblock after blocked control substitution has been applied and the fall-through branch has been inserted. The FRP for the fall-through branch has been expedited while the computation of other FRPs remains sequential. Assume that a store operation at exit $E4$ is guarded using the fall-through branch and aliases with previous stores guarded under FRPs $f0$, $f1$, and $f2$. The store at $E4$ is trapped below previous stores which have sequentially computed FRP operands. The benefits of blocked control substitution have been thwarted and sequential FRP evaluation remains on-trace.

Predicate splitting eliminates the need for sequentially computed FRPs on the on-trace control flow path. Predicate splitting also decreases the number of required on-trace FRP evaluations. The cost to compute on-trace FRPs is reduced by evaluating only the lookahead FRP but not intervening sequential FRPs.

Predicate splitting has been used for the acceleration of control dependences in loops with conditional exits.[4] This paper adapts the technique to scalar code. Predicate splitting can be compared to the following control flow transformation. A heuristic selects lookahead branches within a superblock. After splitting, each nonspeculative operation will be positioned either before all branches or immediately after a lookahead branch. Operations which are not correctly positioned must be moved just below the next lookahead branch. As operations move down, they are copied off-trace at each branch below which they move.

Predicate splitting replaces a computation guarded by predicate p with multiple copies of the computation guarded by predicates $q1,..., qn$ provided the following conditions are satisfied:

1. $q1 \vee \cdots \vee qn = p$.
2. No more than one of $q1,..., qn$ evaluates to true.

After splitting, the effect of the multiple copies of the computation under predicates $q1,..., qn$ is the same as the effect of the original computation under p. This predicate transformation simulates the motion of a computation below a branch. The second condition can be relaxed for certain types of computations, e.g., computations that don't overwrite their input operands.

Figure 11b shows the effect of predicate splitting. As a result of the downward motion simulated by predicate splitting, each operation is split into two components: an on-trace operation guarded with the lookahead

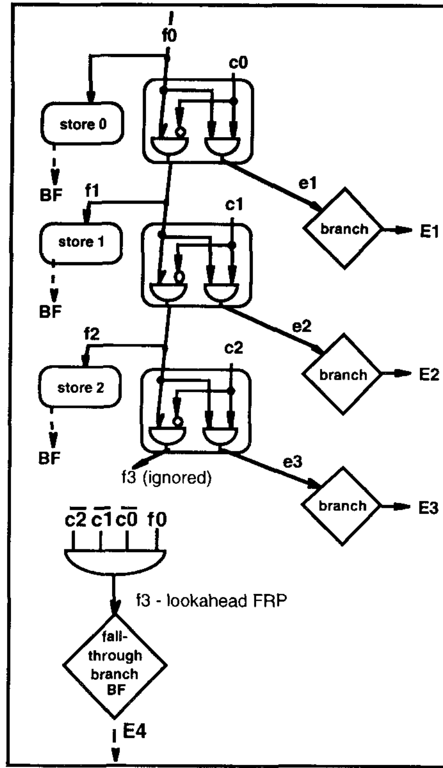
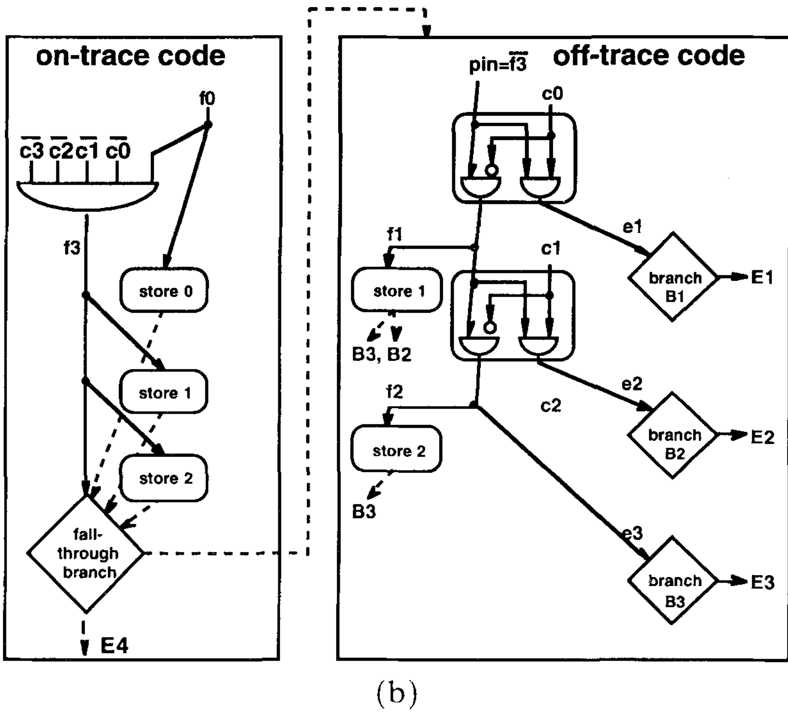


Fig. 11. Superblock graph with split predicates. (a) Code after blocked control substitution, (b) code with split predicates.

FRP, and an off-trace operation. The benefits of predicate splitting can be seen by examining the code required within the on-trace code component of Fig. 11b. The lookahead FRP provides an expedited guard for store operations as well as a branch condition for the fall-through branch. Only the fall-through branch and the corresponding lookahead FRPs must be computed on-trace, remaining FRPs and exit branches can be computed off-trace.

This paper assumes that only one of the components of a split operation may execute. Thus, off-trace operations are carefully guarded to avoid redundant execution. Two ways to accomplish this are simultaneously illustrated in Fig. 11b.

In the first approach, the complement of the lookahead FRP is used as the initial predicate for the chain of off-trace FRP conjunctions. Note



(b)

Fig. 11 *(Continued)*

that the complement of $f3$ is used as the predicate input ("pin") to the sequence of compare operations which compute off-trace FRPs. Thus, FRPs for off-trace operations are false when the lookahead predicate is true, and split operations can be moved back on-trace without redundant execution. This approach will be used to demonstrate the motion of off-trace code back on trace during scheduling as presented in Section 4.

In the second approach, off-trace operations are dependent on the fall-through branch to prevent them from moving back on-trace. This is illustrated with the branch dependence from the fall-through branch to the off-trace code. In this case, "pin" can be set to true because the branch dependence precludes redundant execution.

Predicate splitting can only be applied to stores which are separable[4] with respect to a lookahead branch (or the corresponding lookahead FRP). Consider a store within a superblock. The store is **separable** with respect to some subsequent branch if the store can be moved below that branch without violating a dependence to a load which is used to evaluate

the branch condition for any branch traversed by the store's motion. After predicate splitting, stores use a lookahead FRP. No load operation depending on these stores can be used to compute a condition needed for lookahead FRP evaluation; otherwise, there is a cycle in the computation which cannot be executed.

Blocked control substitution (see Section 3.4) requires a heuristic to select lookahead FRPs. However, lookahead FRP selection interacts with predicate splitting. The selection of lookahead FRPs should not require that predicates are split for operations which are non-separable with respect to the lookahead predicate. This leads to the following condition for selecting lookahead FRPs: Given a current lookahead FRP, select the next lookahead FRP so that predicate splitting can be applied to intervening stores. To illustrate that this is always possible, consider the limiting case where every FRP is chosen as a lookahead FRP. In this case, lookahead proceeds across only a single branch. Each lookahead FRP is calculated as the conjunction of a previous lookahead FRP and its fall-through condition. Since all FRPs are computed, every store is properly guarded by a lookahead FRP and does not need to be split. The code degenerates to un-split and irredundant code.

4. SCHEDULING FOR SUPERBLOCK CPR

This section describes a scheduling approach adapted to code produced by CPR. It is similar to superblock scheduling[7] and uses well understood list scheduling techniques.

4.1. Basic Scheduling Approach

The approach takes advantage of the scheduling freedom offered by the use of FRPs and the fall-through branch. The basic idea is this. Assuming that the fall-through branch is the most probable, its placement in the schedule divides the schedule into two parts. The code scheduled above the fall-through branch is part of the on-trace path. The code scheduled below the fall-through branch is not executed on-trace and is considered to be the off-trace code component.

The scheduling approach is illustrated in Fig. 12. Part (a) shows a superblock selected from the control flow graph for a program. The fall-through exit *E*4 is assumed to be the most probable. Part (b) shows a VLIW schedule with columns representing three function units and rows representing time proceeding top to bottom. The fall-through branch for *E*4 has been introduced and is scheduled like any other branch.

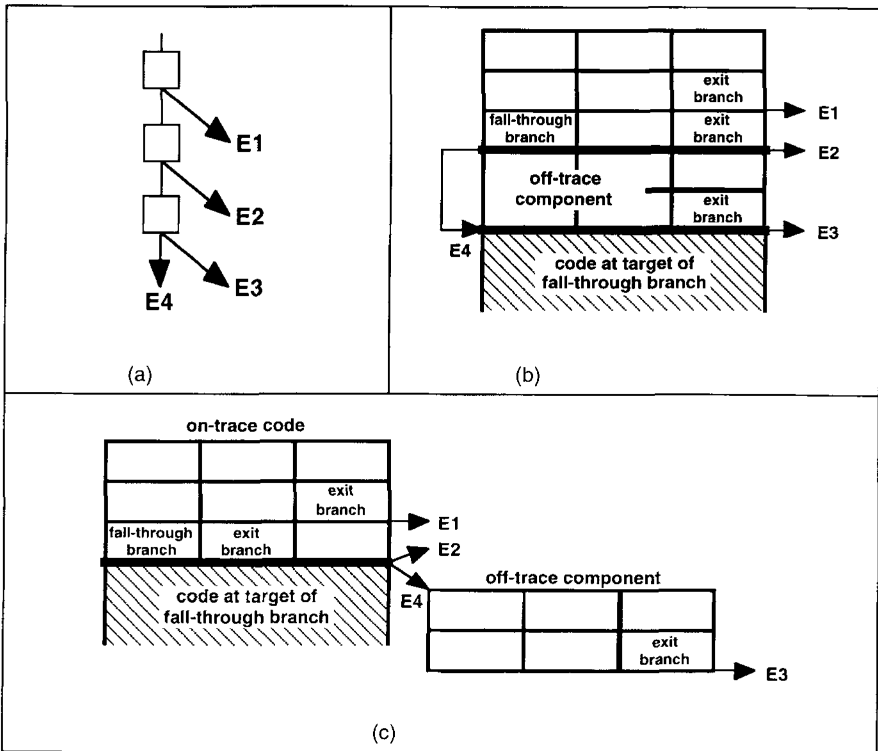


Fig. 12. Scheduling model. (a) Flow graph, (b) conceptual schedule, (c) final code.

The scheduling model is compatible with list scheduling.[15] The heuristic described here applies list scheduling separately to each exit and all operations that must precede the exit. Exits, including the fall-through branch, are scheduled in a priority order based on the exit probabilities. First, the fall-through branch and the operations needed on trace are scheduled. Then, the scheduler places the next probable exit and the operations that must precede this exit. At this time, the scheduler fills any unused spaces in the schedule for the on-trace code with the as yet unscheduled operations to support the next probable exit. This process is repeated until all exits and associated code have been scheduled. Note that the example schedule in Fig. 12b shows that the order of the branches has been interchanged. Moreover, two branches have been scheduled concurrently.

The code positioned below the branch to $E4$ but above its target, shown between the thick lines, is not part of the on-trace path. This is the off-trace component of the schedule. After scheduling, the schedule is

reorganized so that the predominant path does not branch. The FRP for the fall-through branch is negated and made to branch to the beginning of the off-trace component as shown in Fig. 12c.

4.2. Use of Multiple Fall-Through Branches

CPR requires at least one fall-through branch for the last lookahead FRP (corresponding to the on-trace exit). The use of a single fall-through branch, which gets converted to an off-trace exit after scheduling, requires that branch conditions for all intervening branches must be available to resolve the fall-through branch. This may unnecessarily delay exits from the superblock when a single fall-through branch is used in conjunction with a long sequence of exit branches.

The scheduling approach described earlier can accommodate multiple fall-through branches. Blocked control substitution is used to expedite multiple lookahead FRPs for the on-trace path. The complement of any lookahead FRP can serve as the predicate for a fall-through branch. We insert one fall-through branch for each lookahead FRP.

Figure 13 illustrates the use of multiple fall-through branches. The scheduling region in Fig. 13a contains 4 off-trace exits and two fall-through branches. Two of the off-trace exits precede the first falls through branch, and the other two are between the first and the second fall-through branch.

Figure 13b uses shading to show the allowed placement of the code needed for each exit. All code required for the two fall-through branches must remain on-trace (white region). Code for the first two off-trace exits can remain on-trace or migrate into a compensation block at the first fall-through branch. Similarly, the code for the off-trace exits $E3$ and $E4$ can

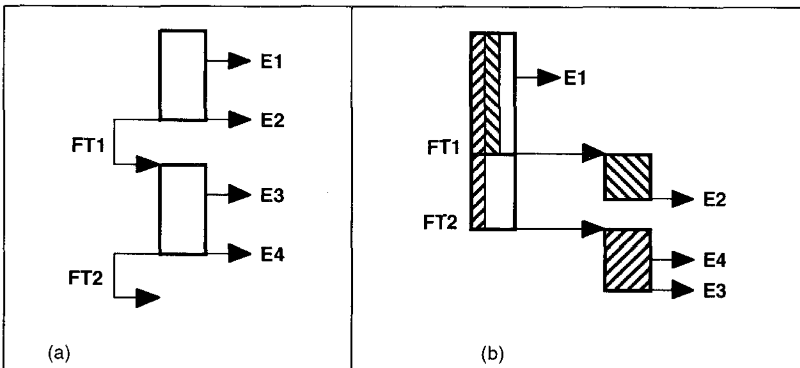


Fig. 13. Use of multiple fall-through branches. (a) Two fall-through branches, (b) example schedule.

remain on-trace or migrate into the compensation block at the second fall-through branch. In this example, on-trace code including both fall-through branches has been scheduled first. The code for exit $E1$ was placed as the scheduler filled in empty spaces after completing the on-trace schedule. All code for $E1$ landed on trace. Then, code for exits $E2$, $E3$, and $E4$ were scheduled and migrated partially off-trace. In some cases, code for all exits may fit in the on-trace schedule, and compensation blocks are empty and never visited.

The scheduling approach described in this section offers a number of advantages. First, by scheduling exits in priority order, the on-trace path is scheduled without regard for off-trace requirements. This allows a minimal length on-trace schedule. Operations scheduled to support lower priority exits naturally fill in unused space within the prior schedule of operations supporting higher priority exits. Second, all compensation code is kept local to the region of scheduling. An operation naturally moves off-trace when a fall-through branch is scheduled above it. However, it need not move to an adjacent scheduling region. It is naturally scheduled in an off-trace component of the current scheduling region. Third, the scheduling approach requires only simple interaction between code generation, optimization and scheduling. In the most general case, every code motion step during scheduling can be followed by an optimization step. This leads to very complex scheduler/optimizer interaction. By selecting lookahead FRPs and splitting predicates, CPR freezes key decisions about code motion and allows optimization to take advantage of these decisions. Scheduling proceeds without additional optimization.

5. OTHER CPR TRANSFORMATIONS

While control CPR has been discussed earlier, this section identifies a number of other techniques which must be applied to achieve the best possible performance.

5.1. Predicate Speculation

Because of the small size of basic blocks, speculation is essential for exploiting ILP. Traditionally, it has been applied in the context of branching code. In the case of predicated code, speculation is performed by substituting a speculative guard predicate for the original guard predicate for an operation. A speculative guard is a predicate for a basic block which dominates the original block. In a superblock, a speculative guard corresponds to the predicate for a block which precedes the original block.

Typically, the speculative guard is available earlier than the original guard, and the transformation provides CPR. We use predicate speculation as described in Ref. 12, which closely mirrors speculative code motion within control flow graphs.

5.2. Data CPR

ILP compilers have used data CPR to reduce the length of critical paths.[2, 6, 7] For example, consider the following sequence in which $c1$, $c2$, $c3$ are constants:

$$x = w + c1;\ y = x + c2;\ z = y + c3$$

It can be rewritten as:

$$x = w + c1;\ y = w + (c1 + c2);\ z = w + (c1 + c2 + c3)$$

The same number of operations execute after constants are folded, but now they can be executed in parallel. These techniques can be applied prior to control CPR and subsequent optimizations.

Blocked back-substitution was used to accelerate data recurrences in loops.[3] This paper introduces a similar technique for scalar code, called **blocked data substitution**. It can be applied when a chain of dependent associative operations computes a sequence of terms. Consider, for example, the following code:

$$s1 = s0 + t0; \quad s2 = s1 + t1; \quad s3 = s2 + t2; \quad s4 = s3 + t3$$

$$s5 = s4 + t4; \quad s6 = s5 + t5, \quad s7 = s6 + t6; \quad s8 = s7 + t7$$

Initially, the code executes sequentially. We could perform full data CPR for each term in the sequence using an independent height-reduced expression for each term. This, however, requires $O(n^2)$ operations.

In blocked data substitution, a heuristic selects lookahead terms in the sequence. Each lookahead term is computed from a previous lookahead term using CPR. Non-lookahead terms are computed sequentially. Assume that $s4$ and $s8$ are selected as lookahead terms. The reorganized code consists of two lookahead expressions:

$$s4 = s0 + ((t0 + t1) + (t2 + t3)) \quad \text{and} \quad s8 = s4 + ((t4 + t5) + (t6 + t7))$$

In addition, there are six conventional expressions:

$$s1 = s0 + t0; \quad s2 - s1 + t1; \quad s3 = s2 + t2$$

$$s5 = s4 + t4; \quad s6 = s5 + t5; \quad s7 = s6 + t6$$

Lookahead expressions are associated assuming that the lookahead input is critically late. Assume that L is the latency of an add operation. The critical path from $s0$ to $s8$ is reduced from $8L$ to $2L$, and the worst case path length from any input to $s8$ is reduced from $8L$ to $4L$.

Blocked data substitution provides substantial CPR with a maximum two fold increase in operation count. In some cases, the lookahead terms in the sequence (e.g., $s4$, $s8$) are the only terms that are live-out on the on-trace path. In this case, the computation of nonlookahead terms may be moved off-trace leaving no redundant code on trace.

5.3. On-Trace/Off-Trace Optimization

On-trace/off-trace optimization is an optimization framework which minimizes off-trace requirements for on-trace code. It can be viewed as an extension of blocked data and control substitution. Conceptually, on-trace/off-trace optimization replicates original code with two copies: on-trace and off-trace counterparts. Optimization is performed in multiple passes. First, CPR and optimization is applied to the on-trace code ignoring the requirements of the off-trace code. Then, the off-trace code is optimized with knowledge of the resultant on-trace code. Expressions computed on-trace need not be recomputed off-trace.

On-trace/off-trace optimization provides a viewpoint which systematically provides the lowest latency and fewest operations on trace. A number of on-trace/off-trace optimizations are used in the example discussed in Section 6. When conventional optimizations (such as copy elimination, dead code elimination, constant folding, load/store elimination) are applied first on-trace and then off-trace, improved on-trace code quality results.

Store elimination provides an important example of on-trace/off-trace optimization. Using predicate splitting, on-trace stores are moved to a lookahead FRP where they execute under a common predicate. If they overwrite a common location, redundant stores are removed, and only a final store remains on trace. The Multiflow compiler achieved a similar effect for live-out assignments. It moved them downward and into compensation blocks leaving a single assignment on trace.[6]

6. EXAMPLE OF SUPERBLOCK CPR

This section provides an example to demonstrate CPR concepts introduced in this paper. The example C++ source program is shown in Fig. 14. The main program shown in Part (a) invokes sum2 twice to add the top three stack elements. The sum2 function shown in Part (b) pops the top two stack elements, adds them, and pushes the result back on the

(a)

```
void main(){
  stack q;

/* initialize stack */
...
/*add top 3 elem.*/
  q.sum2();
  q.sum2();
... }
```

(b)

```
void
stack::sum2(){
  x = pop();
  y = pop();
  push(x+y);
  return; }
```

(c)

```
int stack::pop(){
  int r;
  if(p<=ep) goto empty;
  r=*p;
  p-=1;
  return r;
empty:
  ... }
stack::push(int a){
  if(p>= fp) goto full;
  p+=1;
  *p=a;
  return;
full:
  ... }
/*ep & fp are empty and
full pointer limits*/
```

(d)

```
p0=load(p);
c0=cmpp.un(p0<=ep);
branch empty if c0;
x0=load(p0);
p1=sub(p0,1); /*O p0=sub(p0,1)*/
store(p, p1);
c1=cmpp.un(p1<=ep);
branch empty if c1;
y0=load(p1);
p2=sub(p0,2); /*O p0=sub(p0,1)*/
store(p, p2);
v0=add(x0,y0);
c2=cmpp.un(p2>=fp);
branch full if c2;
p3=p1;          /*O p0=add(p0,1)*/
store(p, p3);
store(p3, v0);
```

Fig. 14. Stack example. (a) Main program, (b) sum2, (c) pop and push, (d) inlined sum2.

stack. Part (c) shows the relevant code for push and pop subroutines. The variables *ep* and *fp* are the low- and high-bounds for the stack. Assume that any branch to "full" or "empty" within push or pop is rare.

The scope over which analysis, optimization and scheduling are performed must be large enough to reveal significant ILP. Inlining is used to enlarge the scope. Figure 14d shows the code for sum2 after inlining push and pop and applying certain optimizations. For example, loads from *p* for the second call to pop and the call to push have been eliminated. Also, the sequential chain of assignments to *p* has been parallelized by renaming and substitution. Original code is sometimes shown in comments /***O**...*/ to help explain the inlined code. Extra copies have been left in the code to simplify presentation, assume that these will be eliminated.

Figure 15 shows the code after applying control CPR and on-trace optimizations. Stores and operations that write live-outs are nonspeculative and guarded using FRPs. Other operations can be executed speculatively. For example, loads and other speculative operations frequently execute with true (omitted) predicate.

The sequence of optimization steps and the actual placement of operations in the final schedule are not shown. To simplify presentation, the code is split into two parts, one for each call to sum2. Each part shows the on-trace code as well as the related compensation code generated by the scheduler. We assume that the heuristic for blocked control substitution and fall-through branch insertion picks FRPs that correspond to completing the first and second invocations of sum2.

Consider the optimized on-trace code for the first call to sum2. The lookahead FRP, *f*3, is expedited, and its complement is used as the

off-trace branch condition. Predicate splitting followed by redundant store elimination results in a single on-trace store to stack pointer p; the other two stores move off-trace. Careful optimization of the computation of $f3$ eliminates the $p0 <= ep$ test, since it is subsumed by the $p1 <= ep$ test. Also, note that three branches in the original code have been replaced by a single branch to off-trace code.

The code for the second call to sum2 is similar to that for the first call; see Fig. 15b. We have renamed operands with trailing % to distinguish them from the corresponding ones in the first invocation. All optimizations applied to the code for first call also apply to the code for the second call. In addition, there are new optimization opportunities. For example, the pointer $p0\%$ is equal to the previously calculated $p1$ and the value $x0\%$, which was read from memory, is the same as the value $v0$ calculated in the first call.

CPR has exposed substantial parallelism not available in the original code. The second invocation of sum2 overlaps almost entirely with the first

```
f0=true; /* entry predicate for first sum2 is true*/
p0=load(p);
p1=sub(p0, 1);
p2=sub(p0, 2);
p3 = p1;
x0=load(p0);
y0=load(p1);
v0=add(x0, y0);

/* compute lookahead FRP */
f3=f0;
                    /*O f3=cmpp.an(p0<= ep); */
f3=cmpp.an(p1<=ep);
f3=cmpp.an(p2>= fp);

/* f3 guarded code */
store(p2,v0) if f3;
store(p,p3) if f3;
branch OT1 if !f3;

/*continue on-trace with second sum2 invocation*/
 ...

OT1:     /* first compensation area */
otp=!f3; /* complement lookahead pred */
f1,e1 =cmpp.uc.un(p0<=ep) if otp;
f2,e2 =cmpp.uc.un(p1<=ep) if f1;
store(p,p1) if f1;
store(p,p2) if f2;
branch empty if e1;
branch empty if e2;
branch full if f2;
```

(a)

```
f0% = f3; /* entry predicate for second sum2 is
f3*/
p0% = p1;         /*O p0%=load(p); */
p1% = p2;         /*O p1%=sub(p0%,1); */
p2% = sub(p0, 3); /*O p2%=sub(p0%,2); */
p3% = p2;         /*O p3%=p1% */
x0% = v0;         /*O x0%=load(p0%); */
y0%=load(p1%);
v0%=add(x0%,y0%);

/* compute lookahead FRP */
f3%=F0%;
                    /*O f3%=cmpp.an(p0<= ep); */
f3%=cmpp.an(p1%<=ep);
f3%=cmpp.an(p2%>= fp);

/* f3% guarded code */
store(p2%,v0%) if f3%;
store(p,p3%) if f3%;
branch OT2 if !f3%;

/*continue on-trace with next superblock*/
 ...

OT2:  /* second compensation area */
otp% = !f3%; /*complement lookahead pred */
f1%,e1% =cmpp.uc.un(p0%<= ep) if otp%;
f2%,e2% =cmpp.uc.un(p1<= ep) if f1%;
store(p,p1%) if f1%;
store(p,p2%) if f2%;
branch empty if e1%;
branch empty if e2%;
branch full if f2%;
```

(b)

Fig. 15. CPR Optimized stack example. (a) First sum2 invocation, (b) second sum2 invocation.

invocation in spite of the presence of three branches in the original code. A single wired-and (also a single branch) separates the completion of the second invocation from the completion of the first. The use of wired-and is not necessary; properly associated two input AND operations give similar results. Additional control parallelism can be exposed using larger lookahead distance and fewer fall-through branches along the critical path.

In this example, control and data CPR provide substantial benefit to on-trace code. Control CPR has reduced six on-trace branches to two. Much of the branch resolution and branch target formation is performed only off-trace as needed. ON-trace/off-trace optimization simplifies the evaluation of on-trace compare conditions needed for the on-trace FRP. Multiple stores to p are replaced with a single on-trace store. Data height through arithmetic sequences is reduced; for example, the final value of the pointer p is evaluated in a single subtract.

7. CONTROL CPR FOR GENERAL SINGLE ENTRY ACYCLIC REGIONS

This section generalizes the CPR techniques to single entry regions with acyclic flow of control, called *Single Entry Acyclic Regions* or SEAR for short. The overall approach is as follows: compute FRPs for blocks and exits in a SEAR, use these FRPs to if-convert the SEAR so that the region is a block of predicated code with no internal control flow, and use blocked control substitution to expedite computation of certain FRPs. This section describes the computation of FRPs in a SEAR and extends blocked control substitution to SEARs.

7.1. Computing FRPs in a SEAR

FRPs for a SEAR can be computed sequentially using its control flow graph. Program branches correspond to predicate ANDs and program merges correspond to predicate ORs. However, this does not take advantage of additional parallelism that can be exposed using the control dependence graph. The concept of control dependence was defined in Ref. 16. More recent work defines efficient algorithms for computing control dependences.[17] If-conversion of single entry and single exit regions using the control dependence graph is described in Ref. 18. If-conversion was extended to support hyperblock scheduling in Ref. 12; but the approach ignores exits and does not compute fully-resolved predicates.

To illustrate the computation of FRPs for basic blocks and exits within a SEAR, consider the example in Fig. 16a. Nodes correspond to basic blocks. Nodes in the control flow graph are numbered 1–8 with entry

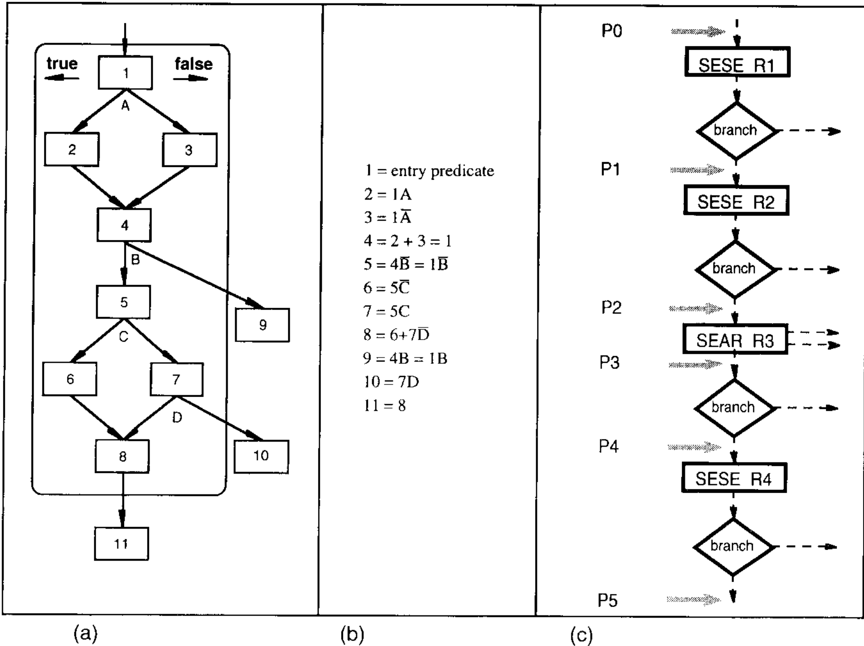


Fig. 16. Computing RRPs for a SEAR. (a) Example SEAR control flow group, (b) example SEAR FRPs, (c) CPR of a complex SEAR.

node numbered 1. We assume that an entry predicate is set to true on entry to the region. A pseudo node is introduced for each region exit; these are denoted by 9, 10, 11. Every branch has an uppercase letter corresponding to the branch condition which determines its direction of flow. By convention, branches go left on true and right on false condition.

The control dependence graph for a program defines two key concepts required to efficiently compute FRPs. First is the concept of control equivalence: two nodes x and y are control equivalent when a control flow path through the program visits x if and only if it also visits y. Second is the notion that a basic block in the control flow graph is control dependent on an edge in the control flow graph: a basic block x in the control flow graph is control dependent on the edge from basic block y to basic block z if x does not post-dominate y but x does post-dominate z.[17]

The code to compute predicates for the example SEAR is shown in Fig. 16b. Within each expression, a numeral represents the FRP corresponding to the identically numbered basic block. An upper case character corresponds to a branch condition which may be complemented. The juxtaposition of an FRP and a branch condition denotes a Boolean conjunction.

Thus, 1A denotes the conjunction of the FRP for basic block one and the branch condition labeled A. The + operation denotes Boolean or. One or more expressions is provided to compute each FRP. The first expression (after the first =), is calculated directly from control flow. For example $4=2+3$ indicates that the OR of the FRPs for blocks 2 and 3 correctly computes the FRP for block 4. A second expression (after the second =) is provided when FRP computation using control dependence differs from FRP computation using control flow. For example $4=2+3=1$ indicates that control dependence directly uses the FRP for block 1 as the FRP for block 4.

The expression for each FRP in the example is derived using the following procedure. To compute the FRP for a given node, a set of edges upon which the node is control dependent is identified. Each edge in the set provides one term in the FRP expression. An edge term is calculated using the FRP for the edge's origin node ANDed with the branch condition which traverses the edge. The FRP expression for a node is the OR of all terms for edges on which the node is control dependent. When nodes are control equivalent, multiple nodes have identical FRP expressions; for example, see nodes 4 and 1.

7.2. Blocked Control Substitution for a SEAR

The procedure described in Section 7.1 naturally parallelizes a sequence of if-then-else expressions but does not provide CPR across a sequence of exit branches. This is addressed using the following observation: if all exit branches in a SEAR dominate the fall-through exit, CPR for the fall-through FRP can be performed exactly as for superblocks. Note that in Fig. 16a, the branch within node 4 dominates the fall-through branch while the branch within node 7 does not.

Consider the abstraction of a SEAR shown in Fig. 16c. The SEAR is broken into subgraphs separated by branches which dominate the fall-through exit. These branches decompose the SEAR into a sequence of two subgraph types: single entry single exit (SESE) subgraphs (e.g., R1, R2, R4), and more general SEAR subgraphs (e. g., R3). Blocked control substitution, as in the superblock case, can be used to expedite FRPs across sequences of exit dominating branches spanning SESE regions. But, lookahead cannot be used across SEAR subgraphs because they have exit branches which do not dominate the fall-through path.

For example, *P*2 can be expedited, by re-writing its expression in terms of *P*0 and the conditions for the first two exit dominating branches. Branches internal to the intervening SESE regions are ignored. Similarly, *P*5 can be expedited in terms of *P*3 and intervening branch conditions. The

exit predicate for SEAR $R3$ labeled $P3$ has already been computed in terms of $P2$ and branch conditions internal to $R3$. Since $P3$'s computation depends on branches internal to $R3$, this approach does not expedite the computation of $P3$.

8. APPLICATION TO ARCHITECTURES WITH NO PREDICATES

Although this paper uses predicated execution, control CPR also applies to conventional architectures without predicated execution. Figure 17 shows an approach which again uses a fall-through branch. In Fig. 17a, four nonfully-resolved branches are shown above a fully-resolved fall-through branch. All off-trace exits are tested first and, if the fall-through branch is reached, it always takes and follows the path to E5. In Fig. 17b, off-trace branches are moved below the fall-through branch leaving only the fall-through branch on-trace. Like predicate splitting, stores trapped between branches in Fig. 17a are replicated and the execution of their on-trace components is guarded by the fall-through branch. Logical operations (ANDs) necessary for control CPR can execute using a conventional ALU.

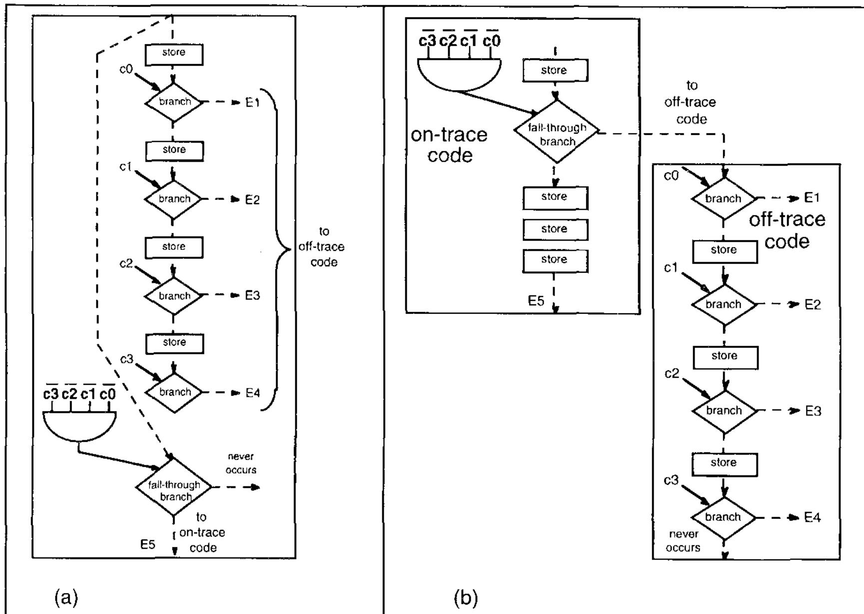


Fig. 17. Branch motion across fall-through branch. (a) FRP for fall-through branch, (b) motion of other branches off-trace.

9. RELATED WORK

There is a substantial body of work on compiler techniques and architectural features to alleviate problems caused by program dependences. Compiler techniques have been developed to reduce critical paths through data dependences. Tree height-reduction has been used to parallelize networks of arithmetic operations.[19] Techniques such as renaming, substitution, and expression simplification have all been used to break data dependence chains.[2, 6, 7] More recent work introduced blocked back substitution for CPR of data recurrences in loops.[3]

Control dependences[16] identify the relationship between a branch and the operations which depend upon its resolution. Control dependences correctly identify minimal conditions under which an operation executes without speculation. Techniques have been developed to alleviate performance limitations due to control dependences on ILP processors. The use of speculative execution is one such technique.[14, 15, 20–24] Speculative execution identifies operations whose side effects can be reversed and moves them above branches upon which they depend.

Branches also limit performance when processors have inadequate branch throughput or excessive branch latency. Compiler techniques have been developed which move branches across other branches.[15, 24] However, interchanging branches alone does not alter the number of on-trace branches as shown in Fig. 18. Performance limitations due to control dependences persist even after interchange. Each time branches are interchanged, code is reorganized requiring complex interaction between scheduling and code generation.

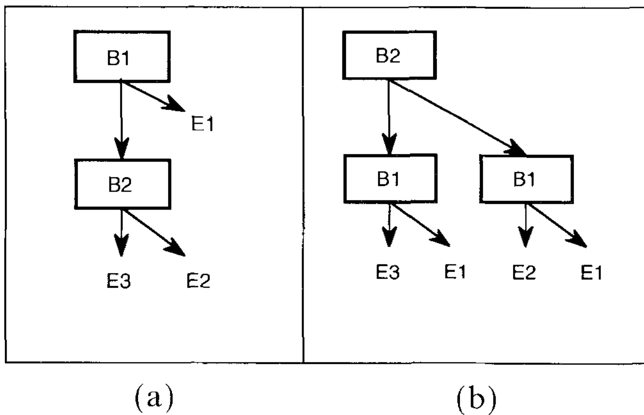


Fig. 18. Interchanging branches. (a) Original branches, (b) interchanged branches.

Architectural features have been used to reduce the dependence height of consecutive branches. The ability to retire multiple dependent branches in a single cycle reduces the height of critical paths through control dependences.[5, 15, 25] The Multiflow Trace machine used a hardware priority encoder mechanism to enforce sequential dependence among concurrently issued branches. The simplest form of multi-way branches cannot guard operations trapped between branches, but more sophisticated branch architectures have been developed to guard intervening operations.[22, 24, 26]

Prioritized multi-way branch alone may fail to eliminate bottlenecks due to control dependences for several reasons. It is unlikely that multi-way branch hardware is as fast as 2-way branch hardware. Many processor architectures require that all branch latencies be uniform. In this case, the 2-way branch latency is matched to the multi-way branch latency: Thus, simple 2-way branches are penalized by the support for multi-way branches.

Multi-way branch achieves minimum latency at the expense of branch scheduling freedom. Minimizing the critical path may force the traversal of multiple branches in a single cycle. Peak branch issue requirements may be difficult to satisfy in hardware, especially if branches can not be issued on all function units.

Predicated execution[9–13, 18, 23, 27] provides another approach to parallelize programs limited by control dependences. For example, a sequence of multiple if-then-else expressions can be parallelized with if-conversion. Control CPR for loops with exits has been demonstrated in prior work,[4] and the generalization of these techniques to control CPR in scalar codes is addressed in this paper.

10. CONCLUSIONS

CPR is a collection of techniques which increase the amount of parallelism in scalar programs. As processors provide more instruction-level parallelism, CPR techniques will become increasingly important. Compile time transformations which better tolerate data and control dependences allow us to exploit hardware implementations with deeper pipelines, wider issue, and simpler support for branches.

This paper describes transformations which reduce critical path lengths in scalar programs. Fully-resolved predicates are introduced to eliminate branch dependences. The introduction of FRPs assists in unifying CPR techniques for both control and data dependences. Critical paths which jointly traverse data and control dependences are height-reduced. The application of control CPR allows branches to move off-trace. Scheduling and optimization models suitable for use with CPR are also

described. This paper illustrates the use of CPR in the context of both superblocks and more general single entry acyclic regions. Control CPR is illustrated for architectures with and without predicated execution.

While the use of CPR transformations to enhance parallelism has been demonstrated, heuristics for the application of CPR are not yet well understood, and the benefits of CPR have yet to be quantified. The utility of CPR depends upon many factors including the nature of the application code and nature of the instruction set architecture.

REFERENCES

1. R. Hank, W. W. Hwu, and B. R. Rau, Region-Based Compilation: An Introduction and Motivation, *Proc. 28th Ann. Symp. on Microarchitecture* Ann Arbor, Michigan, pp. 158–168 (1995).
2. J. C. Dehnert and R. A. Towle, Compiling for the Cydra 5, *J. Supercomputing* 7(1/2): 181–228 (1993).
3. M. Schlansker and V. Kathail, Acceleration of First and Higher Order Recurrences on Processors with Instruction Level Parallelism, *Sixt Int'l. Workshop on Lang. Compilers for Parallel Computing*, U. Banerjee, *et al.* (Eds., Springer-Verlag, pp. 406–429 (1993).
4. M. Schlansker, V. Kathail, and S. Anik, Height Reduction of Control Recurrences for ILP Processors, *Proc. 27th Ann. Int'l. Symp. on Microarchitecture*, San Jose, California, pp. 40–51 (1994).
5. J. A. Fisher, Very Long Instruction Word Architectures and the ELI-512, *Proc. Tenth Ann. Intηl. Symp. Computer Architecture*, Stockholm, Sweden, pp. 140–150 (1983).
6. G. Lowney *et al.*, The Multiflow Trace Scheduling Compilers, *J. Supercomputing* 7(1/2):51–142 (1993).
7. W. W. Hwu, *et al.*, The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *J. Supercomputing* 7(1/2): 229–248 (1993).
8. J. A. Fisher and S. M. Freudenberger, Predicting Conditional Jump Directions from Previous Runs of a Program, *Proc. Fifth Int'l. Conf. Archit. Support for Progr. Lang. and Oper. Syst.*, Boston, Massachusetts, pp. 85–95 (1992).
9. V. Kathail, M. S. Schlansker, and B. R. Rau, HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, California (1993).
10. P. Y. T. Hsu and E. S. Davidson. Highly Concurrent Scalar Processing. *Proc. 13th Ann. Int'l. Symp. Computer Archit.*, pp. 386–395 (1986).
11. B. R. Rau *et al.*, The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions and Trade-Offs. *Computer* **22**(1):12–35 (1989).
12. S. A. Mahlke, *et al.*, Effective Compiler Support for Predicated Execution Using the Hyperblock. *Proc. 25th Ann. Int'l. Symp. Microarchitecture*, pp. 45–54 (1992).
13. J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt, Overlapped Loop Support in the Cydra 5. *Proc. Third Int'l. Conf. Archit. Support for Progr. Lang. Oper. Syst.*, Boston, Massachusetts, pp. 26–38 (1989).
14. S. A. Mahlke, *et al.*, Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution. *ACM Trans. Computer Systems* **11**(4):376–408 (1993).
15. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, Cambridge, Massachusetts, (1985).

16. J. Ferrante, K. Ottenstein, and J. Warren, The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Progr. Lang. Syst.* **9**(3):319–349 (1987).
17. K. Pingali and G. Bilardi, APT: A Data Structure for Optimal Control Dependence Computation. *Proc. Progr. Lang. Design and Implementation*, La Jolla, California (1995).
18. J. C. H. Park and M. S. Schlansker, On Predicated Execution. Technical Report HPL-91-58, Hewlett-Packard Laboratories, Palo Alto, California (1991).
19. D. J. Kuck, *The Structure of Computers and Computations*, John Wiley, New York (1978).
20. J. A. Fisher, Trace scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. Computers* **C-30**(7):478–490 (1981).
21. A. Nicolau, Percolation Scheduling: A Parallel Compilation Technique. Technical Report TR 85-678, Department of Computer Science, Cornell (1985).
22. K. Ebcioglu and A. Nicolau. A *Global* Resource-Constrained Parallelization Technique. *Proc. Third Int'l. Conf. Supercomputing*, Crete, Greece, pp. 154–163 (1989).
23. P. Tirumalai, M. Lee, and M. S. Schlansker, Parallelization of Loops with Exits on Pipelined Architectures, *Proc. Supercomputing*, pp. 200–212 (1990).
24. S.-M. Moon and K. Ebcioglu, An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors, *Proc. 25th Ann. Int'l. Symp. Microarchitecture*, Portland, Oregon (1992).
25. J. A. Fisher, 2^N-way Jump Microinstruction Hardware and an Effective Instruction Binding Method, *Proc. 13th Ann. Workshop on Microprogramming*, Colorado Springs, Colorado, pp. 64–75 (1980).
26. K. Ebcioglu and R. Groves, Some Global Compiler Optimization and Architectural Features for Improving Performance of Superscalars, Technical Report RC16145, IBM T. J. Watson Research Center, Yorktown Heights, New York (1990).
27. B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, Code Generation Schemas for Modulo Scheduled DO-Loops and WHILE-Loops. Technical Report HPL-92-47, Hewlett-Packard Laboratories, Palo Alto, California (1992).