Techniques for Critical Path Reduction of Scalar Programs

Michael Schlansker, Vinod Kathail
Compiler and Architecture Research
HPL-95-112
December, 1995

critical path reduction, control height reduction, data height reduction, blocked control substitution, instruction level parallelism

Scalar performance on processors with instruction level parallelism (ILP) is often limited by control and data dependences. This report 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.

Internal Accession Date Only

A condensed version of this report was published in the *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, Michigan, November 1995.

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: it decreases the length of program critical paths; it improves scheduling freedom; and it yields efficient on-trace code.

This report presents a systematic approach for obtaining height-reduced and optimized code for branch-intensive scalar programs with predictable control flow. Key CPR principles are defined and used to demonstrates the benefits of CPR technology. This work extends earlier work on loops [1, 2] to the scalar case.

This report represents a starting point in understanding CPR for scalar programs. 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. Control CPR decreases the height of critical paths due to control dependences. It also reduces the amount of computation by moving rarely taken branches off-trace. Control CPR uses the concept of *fully-resolved* predicates to convert control dependences into data dependences. Data CPR techniques are adapted to expedite the calculation of fully-resolved predicates. Programs contain a mix of control and data dependences, and CPR techniques are needed for both. Techniques presented in this report unify the treatment of control and data during CPR, optimization and scheduling.

Complex actions on programs, such as CPR or scheduling, must be performed on modest sized regions selected from a program (*e.g.*, a trace [3]). This report assumes that on-trace paths can be identified for scheduling and optimization [4]. Discussion in this report is restricted to single entry code sequences like superblocks and hyperblocks [5].

This report adapts superblock scheduling to support CPR. The scheduling approach allows an operation which is not necessary on the on-trace path to naturally move off-trace. However, code moved off-trace is kept local to the scheduling region. Operations which support off-trace exits can fill in unused space within the on-trace schedule. Compiler engineering is simplified because code generation and optimization are decoupled from scheduling.

Predicated execution, as supported in PlayDoh [6], is used to reduce the coupling between the scheduling of branches and other non-speculative operations. PlayDoh's "wired-and" and "wired-or" compare operations were developed in order to provide the fast computation of n-way boolean operations required by scalar CPR.

The rest of the report 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 architectures without predicated execution. Section 9 discusses related work, and Section 10 contains concluding remarks.

2. Architectural assumptions

This report uses the HPL PlayDoh architecture [6] 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 [7, 8]. Predicated execution uses a boolean data operand to guard an operation. For example, the generic operation "r1=op(r2,r3) if p1" executes if p1 is true and is nullified if p1 is false. Omitting the predicate specifier for an operation is equivalent to unconditionally executing the operation using the constant predicate true.

PlayDoh introduces a family of compare operations which allow parallel computation of high fan-in logical operations. A compare operation appears as:

p1,p2=CMPP.<D1-action>.<D2-action>(r1<cond>r2) 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 specifier.

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

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 result is used in "normal mode" (N), or "complemented mode" (C). In complemented mode, the compare result is complemented before performing the action on the target predicate.

Figure 1 defines the action performed under each of the allowed target action specifiers. The result of an action is specified for all four combinations of input predicate and compare result. Each cell describes the result corresponding to the input combination indicate by the row, and action indicated by the column. The cell specifies one of three actions on the target predicate: set to 0, set to 1, or leave untouched (shown as "-").

pred. input	cmpp result	On result				On complement			
		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

Figure 1: Destination action specifiers for compare-to-predicate operations.

The wired-and action is used to execute high fan-in AND operations as follows: (1) initialize the result register to true; (2) execute n compare operations in parallel or in arbitrary order, each of which uses AN (or AC) action to conditionally set the result to false. After all compares have executed, the conjunction is available as the 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=1;
r=CMPP.AN(i0=0);   r=CMPP.AN(i1=0);
r=CMPP.AN(i2=0);   r=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 execute 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 = 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 in 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 the principles of control CPR. Critical paths are based on dependence constraints which are defined in the context of a scheduling model. We present CPR techniques in the context of superblock scheduling. The use of fully-resolved predicates converts branch dependences into data dependences. Data CPR is then applied to expose parallelism. A fall-through branch is introduced to provide full CPR benefits for the fall-through path. Blocked control substitution and predicate splitting are used when redundant operation count must be minimized for processors with limited ILP.

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 Figure 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.

```
<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*/
```

Figure 2: Example superblock

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

Figure 3(a) shows various types of edges to and from a branch. A branch performs two actions: it consumes a branch condition, and it transfers flow of control. As a condition consumer, 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. Scheduling strategies differ in their definition of branch dependences.

Speculative execution can be used to move operations above branches, and exceptions from speculative operations can be ignored with proper hardware support [6, 9]. However, some operations cannot be executed speculatively without disrupting program semantics. Within this discussion, operations which write values to a live-out register or operations which write to a location in memory are non-speculative. Other operations can be executed speculatively. In Figure 3(a), "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 report 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 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.

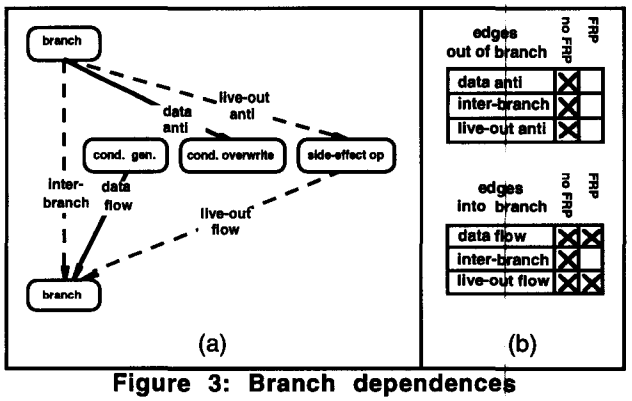


Figure 3: Branch dependences

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 Figure 2. It should branch to E2 only if the first branch falls through and its condition e2 is true. The value of e2 alone is not sufficient to decide whether the second branch takes or not. Assume that the computation for e2 is speculatively moved above the first exit branch. Also, assume that both e1 and e2 are true. The program should branch to E1 and not to E2 even though the condition (e2) for the second branch is true.

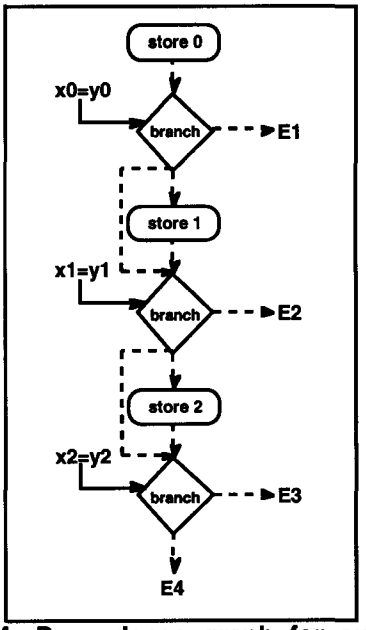


Figure 4: Dependence graph for superblock

Branches can be re-ordered using the approach described in [10]. However, the approach requires compensation code and may not help reduce the critical path. Section 9 discusses this further.

In Figure 3(b), column marked "no FRP" summarizes branch dependences for conventional superblock code. The column marked "FRP" will be discussed in Section 3.2. The relevant parts of the dependence graph for the superblock example are shown in Figure 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.

3.2 Fully-resolved predicates

Given a single entry acyclic region within a control flow graph, a **fully-resolved predicate (FRP)**[1] can be defined for every basic block within the region and for every control flow edge within or exiting the region. Each FRP is computed using a boolean-valued expression every time flow of control traverses the region. The FRP for any block (edge) is true only if the program trajectory on this entry to the region traverses that block (edge); otherwise, the FRP for that block (edge) is false.

Intuitively, the FRP for any block or edge is a conjunction of branch conditions describing the exact condition, relative to region entry, under which the block executes or the edge is traversed.

The use of an FRP allows an action to be correctly guarded using predicates and without relying on control flow. Block FRPs are used to predicate operations. Speculatively executed operations are guarded by predicates

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

other than their block FRPs (*e.g.*, true). On the other hand, non-speculative operations such as stores and live-out overwrites must be correctly guarded using their block FRPs. Edge FRPs are used to predicate branches, which are always non-speculative.

Because FRPs can guard operations without relying on control flow, they can be used to liberalize the rules of code motion. This will provide a basis for a deeper understanding of the parallel execution of programs. Optimizations developed for data can be used to simplify expressions involving both control and data, and these expressions can be transformed to decrease critical path height using data height-reduction techniques. FRPs also enable new optimizations for on-trace code sequences; *e.g.*, the motion of branches off-trace to decrease on-trace branch count.

The remainder of this section discusses FRPs in the context of superblocks which consist of a linear sequence of basic blocks. 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 Figure 5, an FRP is computed for every basic block and for every exit branch. FRPs for basic blocks are labeled 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)∧f1 and e2 =(! (x1=y1))∧f1.

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

```
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*/
```

Figure 5: Superblock code with FRPs

When FRPs are used within superblocks, some of the branch dependences as presented in Figure 3 can be relaxed. In Figure 3(b), the column marked "FRP" defines branch dependences for code using fully-resolved predicates. The dependence graph for the superblock code with FRPs is shown in Figure 6. Each two target unconditional compare operation is shown as a pair of and gates which use the condition in both true (for exit FRP), and complement (for fall-through FRP) forms. Again, for this figure, speculative execution is not considered as indicated by showing only stores within the basic blocks.

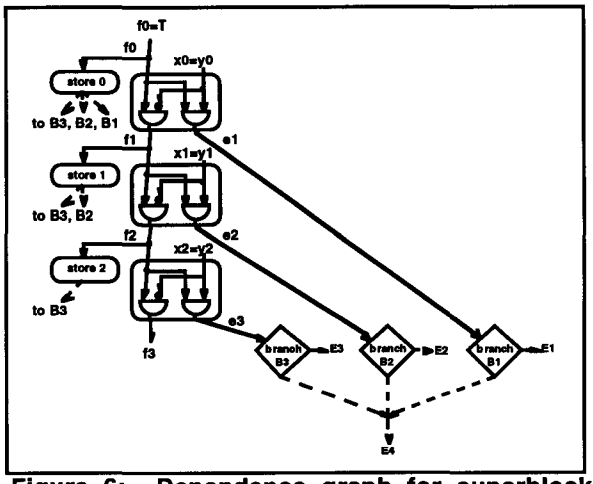


Figure 6: Dependence graph for superblock with FRPs

Data flow edges and live-out flow edges to a branch are enforced just as when predicates were not fully-resolved. FRPs eliminate data anti-dependences and live-out anti-dependences, because the use of FRPs ensures that when a branch takes, subsequent anti-dependent operations do not execute even when moved above the branch. Stores and assignments to live-outs are allowed to move above branches upon which they were anti-dependent in conventional code.

On each entry into a superblock, only a single exit branch is taken. In this case, the use of FRPs instead of conventional branch conditions ensures that branches are mutually exclusive. Thus, inter-branch edges can be eliminated, and branches can move across other branches without compensation code. Consider, for example, the exit to label E3. If the FRP for the branch to E3 is true, code at E3 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 Figure 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 condition (e.g. xi=yi in Figure 5) is now abbreviated as ci to simplify the presentation. The dependence graph for FRP computation in superblocks with full CPR is shown in Figure 8.

```
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;
```

Figure 7 Code for FRPs with full CPR

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 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. For processors with limited amounts of ILP, this is prohibitively expensive.

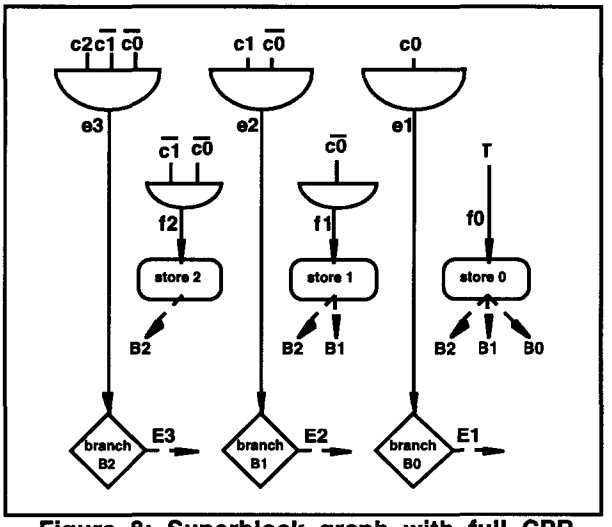


Figure 8: Superblock graph with full CPR

3.4 Blocked control substitution to compute FRPs

This report 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 [2].

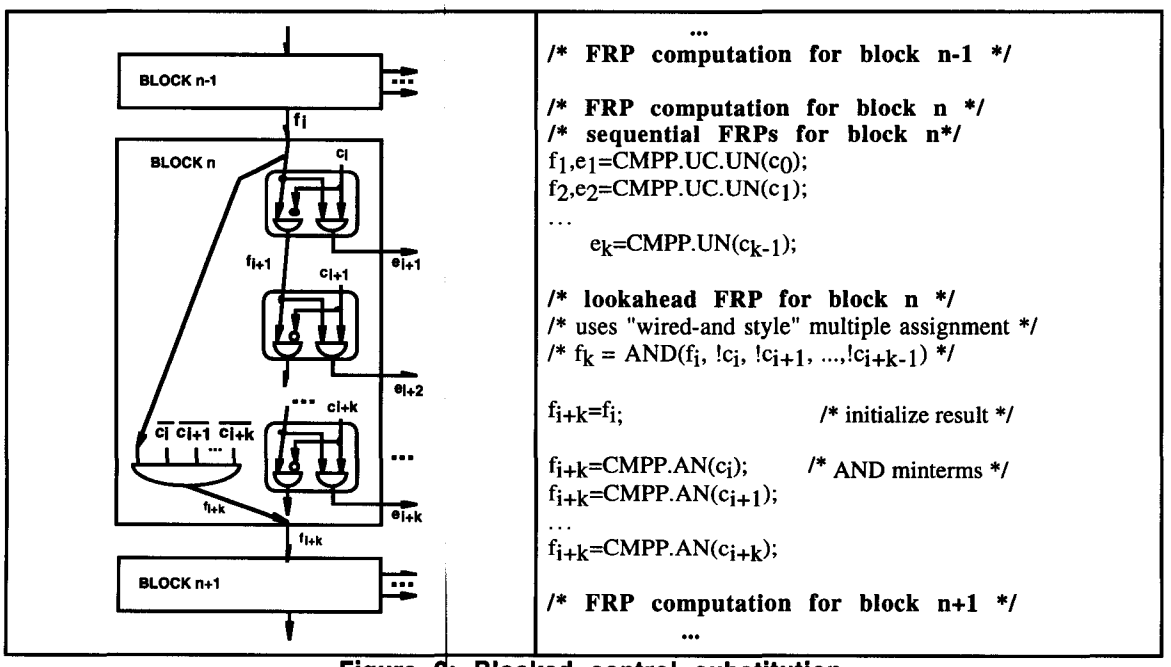


Figure 9: Blocked control substitution

Blocked control substitution is shown in Figure 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 are used to accommodate the late arrival of conditions and to simplify the interaction between code generation and scheduling. PlayDoh code to compute FRPs for blocked control substitution is shown in the right hand side of Figure 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 non-lookahead 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 using both predicate splitting (see Section 3.6) and on-trace/off-trace optimization, 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 Figure 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 Figure 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 Figure 10(a). The code is a version of the superblock in Figure 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 exit branches fail. The fall-through path accommodates live-out computations for all exits. This interferes with on-trace CPR and requires that the fall-through schedule provide time to compute all live-outs.

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*/	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*/
(a) code without fall-through branch	(b) code with fall-through branch

Figure 10: Introducing a fall-through branch

The introduction of a fully-resolved branch, called a **fall-through branch**, allows all exits to be treated identically. Table 10(b) 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 Figure 8. Now, if the live-out for the fall-through path can be calculated quickly, the fall-through branch is free to move above other branches.

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 Figure 11(a). 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 critical 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 [2]. This report 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 non-speculative 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. $q1 \vee \ldots \vee qn = p$.
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 11(b) 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 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 Figure 11(b). The lookahead FRP provides an expedited guard for store operations. Only the fall-through branch is required, and only the lookahead FRP is computed.

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

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 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.

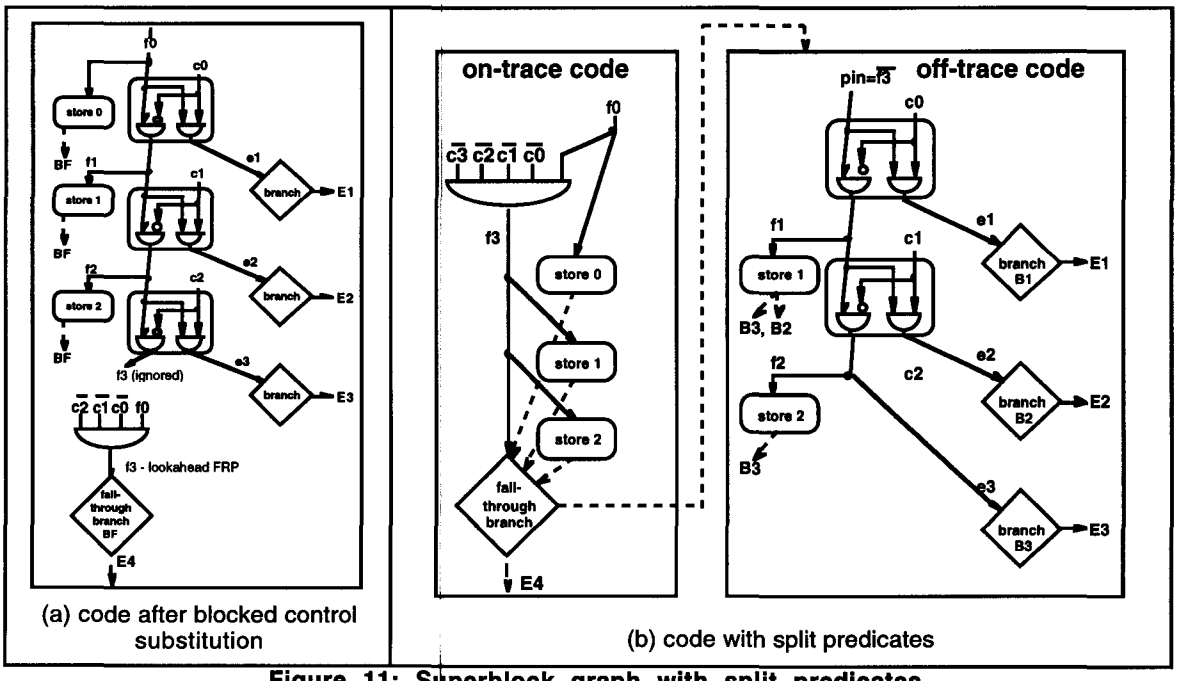


Figure 11: Superblock graph with split predicates

Predicate splitting can only be applied to stores which are separable [2] 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 [5] 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 needed on the on-trace path and can be moved out as an off-trace component.

The scheduling approach is illustrated in Figure 12. Part (a) shows a superblock selected from the control flow graph for a program. The fall-through exit E4 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 E4 has been introduced and is scheduled like any other branch.

The scheduling model is compatible with list scheduling [10]. 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 Figure 12(b) 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 Figure 12 (c).

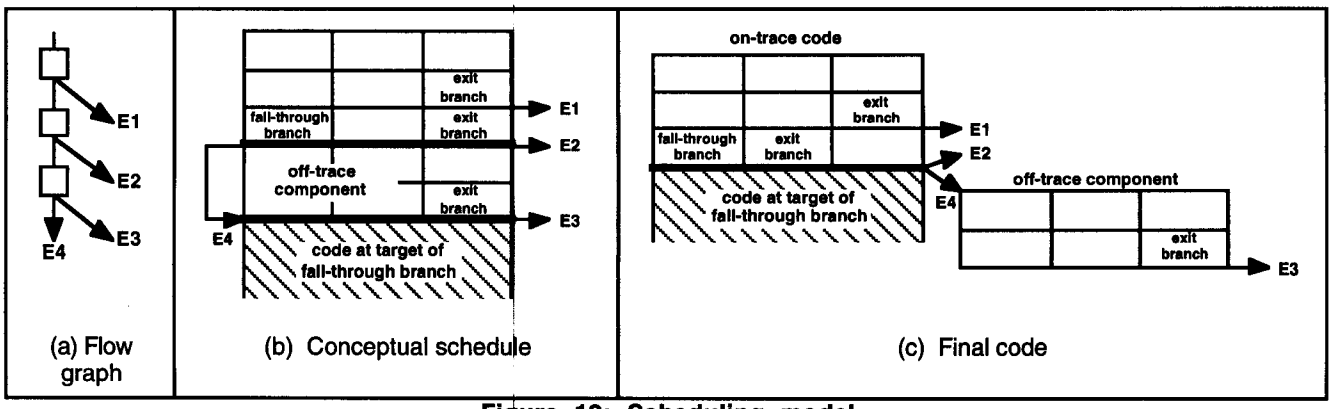


Figure 12: Scheduling model

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 exit from the superblock when a single fall-through branch is used in conjunction with a long sequence of branches.

The scheduling approach described above can accommodate multiple fall-through branches. Blocked control substitution can be 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. In general, we can insert one fall-through branch for each lookahead FRP.

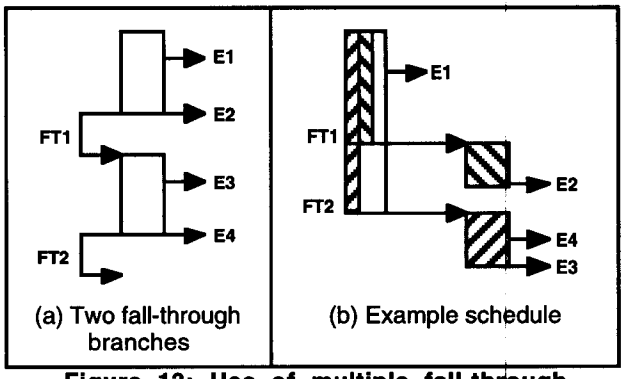


Figure 13: Use of multiple fall-through branches

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

Figure 13(b) 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 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. 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.

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.

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 above, 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. Typically, the speculative guard is available earlier than the original guard, and the transformation provides CPR. We use predicate speculation as described in [11], 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 [12, 13, 5]. 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 re-written 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 [1]. This report 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; s2=s1+t1; s3=s2+t2; s4=s3+t3;
s5=s4+t4; s6=s5+t5, s7=s6+t6; 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)) and s8=s4+((t4+t5)+(t6+ t7)).

In addition, there are six conventional expressions:

s1=s0+t0; s2=s1+t1; s3=s2+t2;

s5=s4+t4; s6=s5+t5; 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 non-lookahead 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 on 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 [12].

6. Example of Superblock CPR

This section provides an example to demonstrate CPR concepts introduced in this report. The example C++ source program is shown in Figure 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 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 14(d) 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 non-speculative 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, f3, 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.

```
void main(){
  stack q;

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

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

```
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*/
```
(c) pop and push

```
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);
```
(d) inlined sum2

Figure 14: Stack example

```
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) First sum2 invocation

```
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 FRBC */
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) Second sum2 invocation

Figure 15: CPR optimized stack example

The code for the second call to sum2 is similar to that for the first call; see Figure 15(b). 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 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 the same result. 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

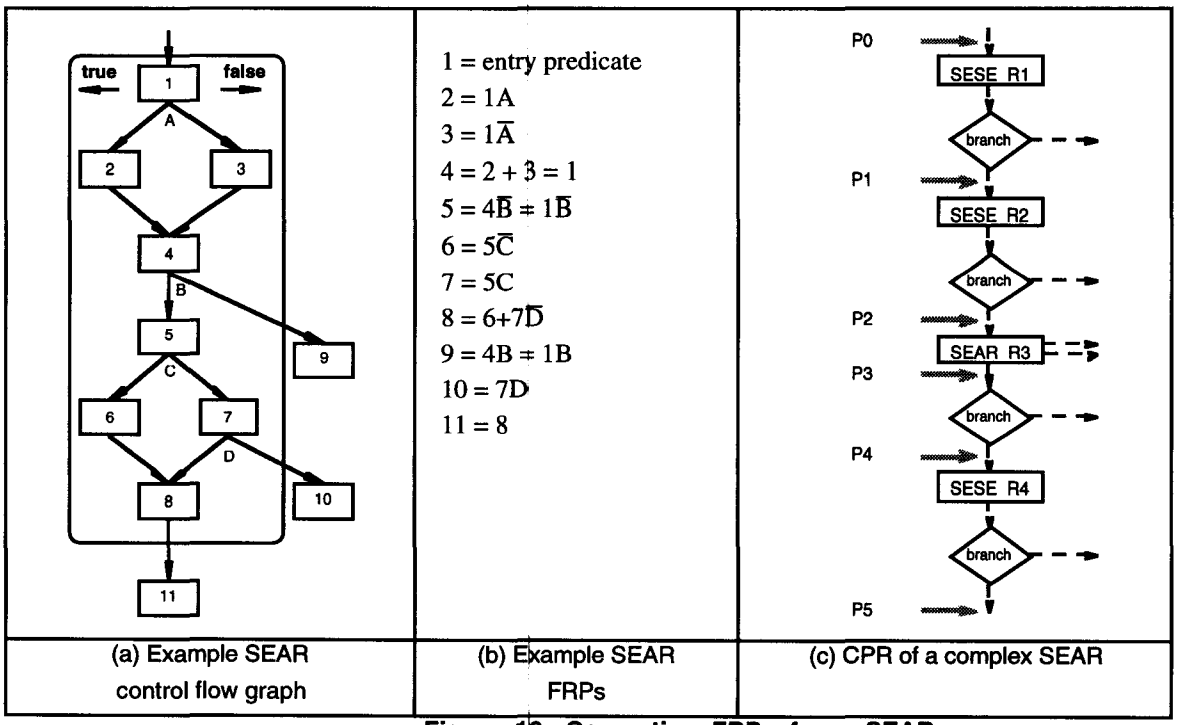
This section generalizes the CPR techniques to single entry regions with acyclic flow of control, called **Single Entry Acyclic Regions** or SEAR for short. The overall approach is as follows: compute FRPs for blocks and exits in a SEAR, use these FRPs to if-convert the SEAR so that the region is a block 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 [14]. More recent work defines efficient algorithms for computing control dependences [15]. If-conversion of SESE regions using the control dependence graph is described in [16]. If-conversion was extended to support hyperblock scheduling in [11]; 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 Figure 16 (a). Nodes corresponding to basic blocks internal to the control flow graph are numbered 1-8 with entry 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 [15].



(a) Example SEAR control flow graph | (b) Example SEAR FRPs | (c) CPR of a complex SEAR

Figure 16: Computing FRPs for a SEAR

The code to compute predicates for the example SEAR is shown in Figure 16 (b). 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. 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 & 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 Figure 16 (a), the branch within node 4 dominates the fall-through branch while the branch within node 7 doesn't.

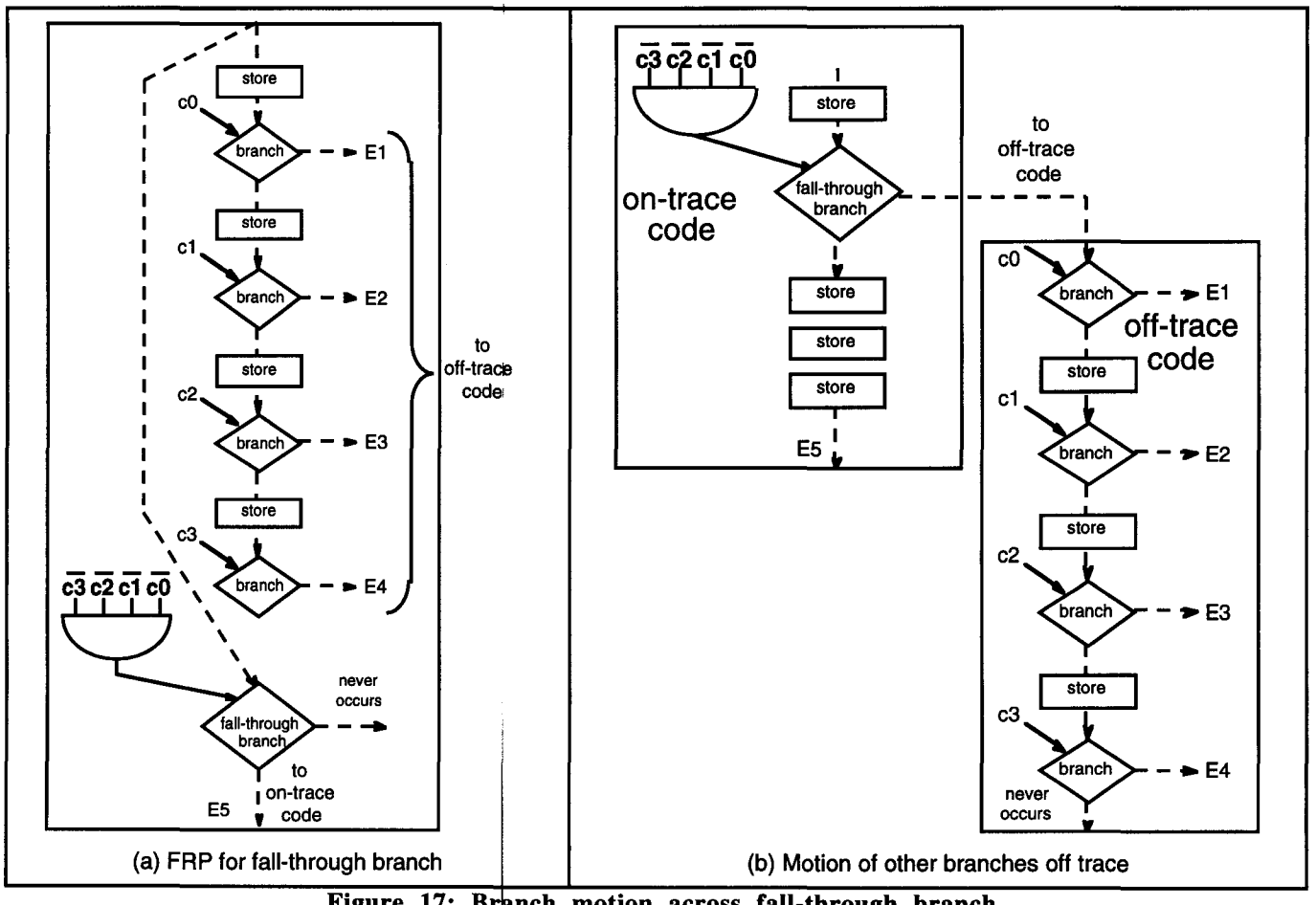


Figure 17: Branch motion across fall-through branch

Consider the abstraction of a SEAR shown in Figure 16 (c). 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, P2 can be expedited, by re-writing its expression in terms of P0 and the conditions for the first two exit dominating branches. Branches internal to the intervening SESE regions are ignored. Similarly, P5 can be expedited in terms of P3 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 report 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 Figure 17 (a), four non-fully-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 Figure 17 (b), 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 Figure 17 (a) are replicated and their on-trace components are guarded by the fall-through branch. Logical operations (ANDs) necessary for control CPR can execute within a conventional ALU.

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 [17]. Techniques such as renaming, substitution, and expression simplification have all been used to break data dependence chains [12, 13, 5]. More recent work introduced blocked back substitution for CPR of data recurrences in loops [1].

Control dependences [14] 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 [3, 10, 18-21, 9]. 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 [10, 21]. However, interchanging branches alone does not alter the number of on-trace branches as shown in Figure 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.

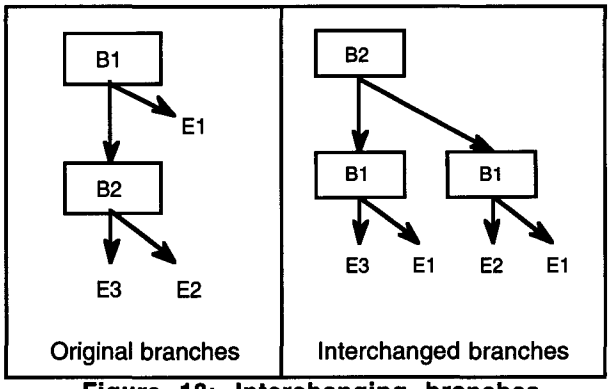


Figure 18: Interchanging branches

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 [22, 23, 10]. 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 [19, 24, 21].

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 [25, 7, 8, 20, 16, 11, 26, 6] 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 [2], and the generalization of these techniques to control CPR in scalar codes is addressed in this report.

10. Concluding remarks

CPR is a collection of techniques which increase the amount of parallelism in scalar programs. As processors provide more ILP, 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 report 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 report 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.

11. Bibliography

1. M. Schlansker and V. Kathail. Acceleration of first and higher order recurrences on processors with instruction level parallelism. In *Sixth International Workshop on Languages and Compilers for Parallel Computing,* U. Banerjee, *et al.*, Editor, Springer-Verlag, 1993, 406-429.
2. M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. *Proceedings of the 27th Annual International Symposium on Microarchitecture* (San Jose CA, 1994), 40-51.
3. J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers* C-30, 7 (1981), 478-490.
4. J. A. Fisher and S. M. Freudenberger. Predicting conditional jump directions from previous runs of a program. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Mass., 1992), 85-95.
5. W. W. Hwu, *et al.* The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing* 7, 1/2 (1993), 229-248.
6. 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 CA, 1993.
7. J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston MA, 1989), 26-38.
8. B. R. Rau, *et al.* The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. *Computer* 22, 1 (1989), 12-35.
9. S. A. Mahlke, *et al.* Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems* 11, 4 (1993), 376-408.
10. J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* (The MIT Press, Cambridge, Massachussetts, 1985).
11. S. A. Mahlke, *et al.* Effective compiler support for predicated execution using the hyperblock. *Proceedings of the 25th Annual International Symposium on Microarchitecture* (1992), 45-54.
12. G. Lowney, *et al.* The Multiflow Trace Scheduling Compiler. *The Journal of Supecomputing* 7, 1/2 (1993), 51-142.
13. J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing* 7, 1/2 (1993), 181-228.
14. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Language Systems* 9, 3 (1987), 319-349.
15. K. Pingali and G. Bilardi. APT: A Data Structure for Optimal Control Dependence Computation. *Proceedings of the Programming Languages Design and Implemenation* (La Jolla Ca., 1995).
16. J. C. H. Park and M. S. Schlansker. *On predicated execution.* Technical Report HPL-91-58, Hewlett-Packard Laboratories, Palo Alto CA, 1991.
17. D. J. Kuck. *The structure of Computers and Computations.* (Wiley, New York, 1978).
18. A. Nicolau. *Percolation scheduling: a parallel compilation technique.* Technical Report TR 85-678, Department of Computer Science, Cornell, 1985.
19. K. Ebcioglu and A. Nicolau. A *global* resource-constrained parallelization technique. *Proceedings of the 3rd International Conference on Supercomputing* (Crete, Greece, 1989), 154-163.

20. P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of loops with exits on pipelined architectures. *Proceedings of the Supercomputing '90* (1990), 200-212.

21. S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. *Proceedings of the 25th Annual International Symposium on Microarchitecture* (Portland, Oregon, 1992).

22. J. A. Fisher. 2^N-way jump microinstruction hardware and an effective instruction binding method. *Proceedings of the 13th Annual Workshop on Microprogramming* (Colorado Springs, Colorado, 1980), 64-75.

23. J. A. Fisher. Very long instruction word architectures and the ELI-512. *Proceedings of the Tenth Annual International Symposium on Computer Architecture* (Stockholm, Sweden, 1983), 140-150.

24. 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, NY, 1990.

25. P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture* (1986), 386-395.

26. 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 CA, 1992.