HEWLETT® PACKARD

# Control CPR: A Branch Height Reduction Optimization for EPIC Architectures

Michael Schlansker, Scott Mahlke, Richard Johnson
HP Laboratories Palo Alto
HPL-1999-34
February, 1999

E-mail: [schlansk,mahlke]@hpl.hp.com
rjohnson@transmeta.com

ILP, critical path reduction, compilers

The challenge of exploiting high degrees of instruction-level parallelism is often hampered by frequent branching. Both exposed branch latency and low branch throughput can restrict parallelism. *Control critical path reduction* (control CPR) is a compilation technique to address these problems. Control CPR can reduce the dependence height of critical paths through branch operations as well as decrease the number of executed branches. In this paper, we present an approach to control CPR that recognizes sequences of branches using profiling statistics. The control CPR transformation is applied to the predominant path through this sequence. Our approach, its implementation, and experimental results are presented. This work demonstrates that control CPR enhances instruction-level parallelism for a variety of application programs and improves their performance across a range of processors.

Internal Accession Date Only

© Copyright Hewlett-Packard Company 1999

# Control CPR: A Branch Height Reduction Optimization for EPIC Architectures

Michael Schlansker Scott Mahlke
Hewlett-Packard Laboratories
Palo Alto, CA 94304
{schlansk,mahlke}@hpl.hp.com

Richard Johnson
Transmeta Corporation
Santa Clara, CA 95054
rjohnson@transmeta.com

## Abstract

The challenge of exploiting high degrees of instruction-level parallelism is often hampered by frequent branching. Both exposed branch latency and low branch throughput can restrict parallelism. *Control critical path reduction* (control CPR) is a compilation technique to address these problems. Control CPR can reduce the dependence height of critical paths through branch operations as well as decrease the number of executed branches. In this paper, we present an approach to control CPR that recognizes sequences of branches using profiling statistics. The control CPR transformation is applied to the predominant path through this sequence. Our approach, its implementation, and experimental results are presented. This work demonstrates that control CPR enhances instruction-level parallelism for a variety of application programs and improves their performance across a range of processors.

## 1 Introduction

Increases in microprocessor performance are driven by both increased clock speed and the use of hardware parallelism to exploit instruction-level parallelism. Explicitly Parallel Instruction Computing (EPIC) architectures, as exemplified by Intel's recently announced IA64, represent an emerging class of processors which support higher levels of instruction-level parallelism by enabling additional parallelization to be performed at compile time. EPIC architectures use three major features to facilitate compile-time parallelization: explicit parallel issue, speculation, and predication. To fully exploit EPIC processors, compilers must also transform and schedule application programs to make more parallelism available at run time.

There is significant concern regarding the amount of available instruction-level parallelism in important applications. Applications with insufficient parallelism fail to exploit hardware parallelism and suffer substantial performance penalties on EPIC processors. Limits on application parallelism come in two basic forms: data dependences and branch dependences. Either type of dependence limits performance by requiring the sequential execution of dependent operations. Rather than accept dependences as hard limits to achieved performance, compiler researchers need to develop techniques to alleviate their limiting effects.

While traditional optimizations focus only on minimizing operation count, *critical path reduction* (CPR) is a family of techniques for transforming programs to reduce branch and data dependence height in order to enhance parallelism. CPR techniques generally face tradeoffs between the need to reduce path length versus the cost of introduced redundant operations.

In this work, we focus on control critical path reduction (control CPR) to reduce branch dependence height. We define a new control CPR technique referred to as the *irredundant consecutive branch method* (ICBM). ICBM reduces height without executing redundant operations. In fact, ICBM can greatly reduce the number of executed branches thus improving performance on processors with exposed branch latency or inadequate branch throughput. The approach is suitable for processors with substantial hardware parallelism as well as for processors with minimal hardware parallelism. While control CPR is broadly applicable to both EPIC as well as superscalar architectures, our implementation focuses on EPIC processors.

This paper makes several important contributions. First, our implementation of an approach for control CPR, ICBM, is presented. While previous papers discussed a conceptual framework for control CPR, this paper describes a working implementation of ICBM within our experimental compiler. We have extended previous control CPR techniques to treat input programs containing arbitrary uses of predicated code including both conventional if-converted code as well as other more complex uses of predicates. ICBM also generalizes previous control CPR techniques to provide more efficient treatment for predicted taken branches. This paper presents effective heuristics to utilize branch profile data to control the application of control CPR. Finally, the paper presents experimental results demonstrating the effectiveness of ICBM for enhancing performance in a variety of applications. For instance, a geometric mean speedup across all the benchmarks of 18% is observed for an EPIC processor with modest hardware resources.

## 2 Background

Researchers have developed compiler techniques to alleviate the performance limiting effects of data dependences. Tree height reduction has been used to parallelize arithmetic computations [Kuc78]. Optimization techniques such as renaming, re-

association, and expression simplification have been used to reduce data dependence height. Height reduction has also been applied to loop data recurrences [DT93] [SK93].

There has also been prior work in reducing the performance limiting effects of branch dependences. Speculative execution reduces height by moving operations above a previous guarding branch. Speculative execution has been used to accelerate program regions such as traces [LFK+93], superblocks [H+93], software pipelined loops [TLS90], and global regions [ME92]. Predicated execution uses an additional boolean operand as a guard which conditionally nullifies each operation. Operations execute to completion when their predicate is true and are nullified when their predicate is false. If-conversion using predicated execution has been used to eliminate branches associated with if-then-else constructs [AKPW83] [DT93] [MLC+92].

Compiler optimization techniques have been developed to reduce the height of critical paths threading through branches and the number of executed branches in application programs. Loop unrolling has been used to reduce the number of executed branches in counted do-loops [LFK+93]. The height and number of executed branches has also been reduced for while loops [SK95]. Optimizations for very specific code patterns have been used to eliminate conditional branches [GK92]. The number of executed conditional branches can also be reduced using code duplication [MW92] [MW95] [BGS95]. Finally, the number of executed branches can be reduced by reordering multi-way switch statements so commonly executed cases appear first [YUW98].

Scalar control CPR provides the potential for a compiler to more systematically treat a broader variety of scalar program branches [SK95]. Prior work outlines an approach for scalar control CPR which identifies new avenues for enhancing program parallelism. This paper is based on a working implementation of control CPR which provides a more detailed understanding of the issues surrounding the design and implementation of control CPR technology within a working compiler. In order to accomplish control CPR without redundant code, our approach utilizes branch profile information to expedite common program paths at the expense of rare program paths. Prior work has shown that branch profiles are relatively consistent across multiple data sets [FF92].

## 3 PlayDoh: An example EPIC architecture

PlayDoh, an EPIC architecture intended to support publicly available research, supports our experiments [KSR93]. Two central features of PlayDoh are important in our discussion and implementation of control CPR: the parallel execution model and predicated execution.

**Parallel execution.** PlayDoh exposes hardware parallelism directly to the compiler. Parallelism is exposed in two forms: simultaneous wide issue, and visible latency. While exposed parallelism is commonly seen in specialized signal and media processors, most general-purpose processors utilize a strictly sequential computational model. Explicit parallelism introduces a number of architectural issues surrounding branches. PlayDoh assumes that branch operations have one or more cycles of exposed latency. This allows the construction of simpler branch units without predictive hardware and which do not stall on a branch mis-predict. On the other hand, this places a substantial burden on the compiler to efficiently utilize the delay slots of exposed latency branches.

| input predicate | result of compare | un | uc | on | oc | an | ac |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | - | - | - | - |
| 0 | 1 | 0 | 0 | - | - | - | - |
| 1 | 0 | 0 | 1 | - | 1 | 0 | - |
| 1 | 1 | 1 | 0 | 1 | - | - | 0 |

**Table 1. Behavior of compare operations**

The execution of a taken branch does not nullify arithmetic operations within its delay slots, and PlayDoh assumes that branches are treated in a like fashion. Hardware branch priority has been previously used to nullify lower priority branches when branches are concurrently executed. This concept can be extended to nullify lower priority branches (and just branches) within the exposed delay slots of a taken branch. This extension is considered to be awkward, and difficult to implement in hardware. Instead, PlayDoh assumes that all branches are naturally pipelined and take effect at their visible latency. When multiple branches take simultaneously, execution semantics is indeterminate. To simplify overlapped branch treatment, our compiler ensures that no branch takes when it is located within a delay slot of another taken branch. Branches can be statically overlapped only when the compiler guarantees that their guarding predicates are not simultaneously true.

**Predicated execution.** The PlayDoh predicate architecture is based on that of the Cydra 5 [DT93]. Predicated execution uses boolean predicates to represent information about flow-of-control within the program. For each execution of the program, a predicate's value is true when control flow reaches a specific point in the control flow graph for the program; the predicate's value is false when control flow does not reach the specified point in the control flow graph. PlayDoh has a number of enhancements over the Cydra 5 which allow a more general and efficient computation of predicates. These enhancements include the definition of compares which compute a pair of predicates in a single operation as well as compares which streamline the evaluation of multi-input logical operations needed for control CPR. A predicate computing compare operation has the form:

```
p,q = cmpp.<x>.<y> cond(a,b) if r,
```

The compare operation is interpreted as follows: `p`, `q` are destination predicate registers; `cmpp` is the generic compare opcode; `<x>`, `<y>` are two-letter action specifiers for each compare destination; `cond(a,b)` is the comparison itself; `r` is a source predicate register.

PlayDoh action specifiers allowed for each result include the following: unconditionally set (UN or UC), wired-or (ON or OC), or wired-and (AN or AC). The first character (U, O or A) indicates the action type (”unconditional”, ”or”, ”and”), while the second character (N or C) indicates the action mode (”normal mode”, or ”complemented mode”). When an action executes in complemented mode, the compare condition is complemented before performing the action on the target predicate.

Table 1 shows the execution behavior of these compare operations in normal and complement modes. Each entry describes the result on the destination predicate; note that the destination may be assigned a value or may be left untouched (denoted as “-”).

From the table, we see that an unconditional compare operation always writes a value into its destination register. The value written is the logical and of the guarding predicate and the comparison result (or its complement). The unconditional compare is commonly used to compute predicates for taken and not-taken successor blocks after a program branch.

The wired-or operation conditionally sets its destination predicate true if both its guarding predicate and its comparison result are true. This form can be used to efficiently compute disjunctions by accumulating terms into a single predicate register that was initially cleared. Since all operations that write the same register conditionally write the same value (true), they can execute in any order. After all conditional writes have completed, a final correct value is left in the target of the wired-or operation. Wired-or writes to a common location are not treated as output dependences and are considered as unordered by the scheduler. Further, in an EPIC architecture like PlayDoh, simultaneous wired-or writes to a common register are well-defined and readily-implemented in hardware.

Similarly, the wired-and compare operation writes the value false if its guarding predicate is true and its comparison result is false. The conjunction of multiple compare conditions is computed by first setting a predicate register to true, then "accumulating" the *and*-ed terms into this predicate register, possibly in parallel. The use of the wired-and and wired-or predicates to accumulate terms in a conjunction or disjunction is used extensively in the control CPR transformation discussed in this paper.

# 4 Approach to Control CPR

This section presents an overview of control CPR and defines a specific approach called the "Irredundant Consecutive Branch Method" (ICBM). Some approaches to control CPR are redundant like full CPR [SK95] which aggressively accelerates all paths within a region at the cost of a quadratic growth in the number of compares. The use of profile data allows us to expedite some program paths at the expense of others; ICBM reduces code growth by accelerating only a single, statically predicted, program path. While the static number of operations typically increases, the dynamic number of executed operations does not. Thus, ICBM is attractive for processors with limited parallelism. Approaches that accelerate multiple paths can further improve performance for highly parallel processors or where static prediction is difficult.

## 4.1 Basic approach

Control CPR is introduced by considering a single-entry, linear sequence of operations containing one or more branches, referred to as a *superblock*. A superblock, consisting of three branches, is given in Figure 1(a). Each branch has a condition computation to determine if it is taken, $a_i < b_i$ in the figure. We assume the preferred execution path in the superblock (the *on-trace* path) is traversed by falling through each successive branch. An *off-trace* path is traversed when any of the exit branches take. Non-speculative operations are guarded by scheduling them below (and outside delay slots of) previous branches. Store operations are used in Figure 1(a) to represent generic non-speculative operations. Branches are sequentially ordered; a chain of dependences exists between branches that exposes all branch latencies.

**FRP conversion.** With traditional control flow, operations are guarded by confining them to basic blocks guarded by branches. Predicates can be used to guard operations without confining them to sequentially executed basic blocks. In prior work, if-conversion was primarily used to eliminate branches within single-entry single-exit program regions [DT93] [MLC+92]. Dependent chains of branches (e.g., those found in superblocks) were not treated and remained dependent during program scheduling. With predicates, dependent chains of branches found in superblocks may be transformed using a variant of traditional if-conversion. The program is first partitioned into single entry acyclic regions and a predicate is associated with each basic block. The predicate is used as a guard for operations within the block. The computation of these predicates and their use as guards eliminates dependences between non-speculative operations (including branches) and previous branches upon which they are dependent. These predicates are referred to as *fully-resolved predicates* (FRPs).

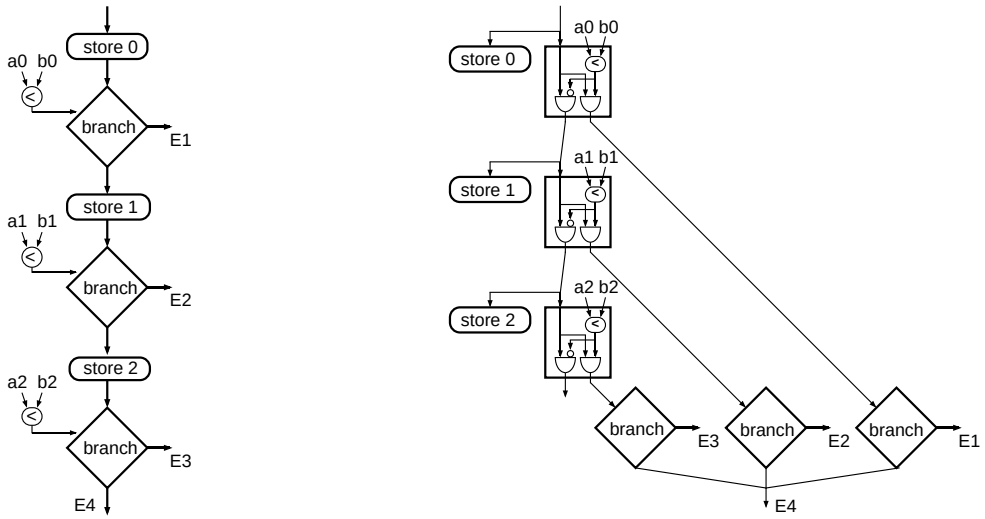
FRPs for both basic blocks and branches within a single entry acyclic region can be defined recursively as follows. The FRP for the entry block has value true. The FRP for any selected non-entry block can be computed by oring a term for each control-flow edge entering the block. The term for each edge is computed by conjoining the FRP for the block from which the edge originates with the branch condition that causes flow of control to traverse the edge and enter the selected block. The FRP for a conditional branch can be calculated as the conjunction of the FRP for the block in which the branch resides and the branch condition that causes the branch to take. Control dependence analysis and boolean expression manipulation are used to optimize FRP expressions.

Our compiler transforms regions by inserting code to compute FRPs for each basic block and for each conditional branch. Operations are guarded by referencing these FRPs as predicate operands. This process is called *FRP conversion* Chains of branch dependences are converted into chains of data dependences through operations that evaluate predicates.

The FRP-converted superblock corresponding to Figure 1(a) is shown in Figure 1(b). Each rectangle provides the functionality of a single PlayDoh compare with UC left-hand and UN right-hand outputs (see Table 1). Each compare computes both an FRP for a basic block as well as an FRP for a branch. The UC output computes a new block FRP by conjoining the previous block FRP with the complement of the branch condition. Analogously, the UN output computes the branch FRP using the branch condition itself. Block FRPs guard non-speculative operations that are no longer dependent upon prior branches.

When any branch FRP in the sequence is true, all other branch FRPs are false. As a result, the branches are mutually exclusive; they may be reordered during scheduling and they may execute in parallel. In Figure 1(b), the fall-through successor "E4" is reached after all three FRP-converted branches execute and fall-through. The conventional superblock in Figure 1(a) is limited by branch dependences, while the FRP-converted superblock in Figure 1(b) is limited by data dependence height through a sequence of compares. Boolean expression optimization can be used to height reduce data dependences through the sequences of conjunctions needed to compute FRPs for dependent branches.

**Transformation.** Figure 2 illustrates a transformation that




a) original superblock, sequential branches

b) FRP-converted superblock, independent branches

**Figure 1. FRP conversion process**

both height-reduces dependences through sequences of branches as well as height-reducing dependences through expressions that compute requisite FRPs. Note that in this Figure the branch conditions are now simply expressed as $c_i$. We begin again with the traditional superblock from Figure 1(a), which is contained inside the rectangle of Figure 2(a). We assume that the program usually falls through all three branches. The original code of Figure 2(a) is augmented with a new branch operation, referred to as a *bypass branch*. The bypass branch behaves as a composite branch that takes when any of the original branches takes and falls through when all of the original branches fall through. Compare operations are added to compute the off-trace (bypass-branch) FRP and an on-trace FRP, corresponding to falling through all of the original branches. These compares are shown as multi-input logical gates to indicate that the FRP expression can be freely re-associated to accelerate evaluation. Note that the bypass branch in Figure 2(a) is redundant, since it never takes.

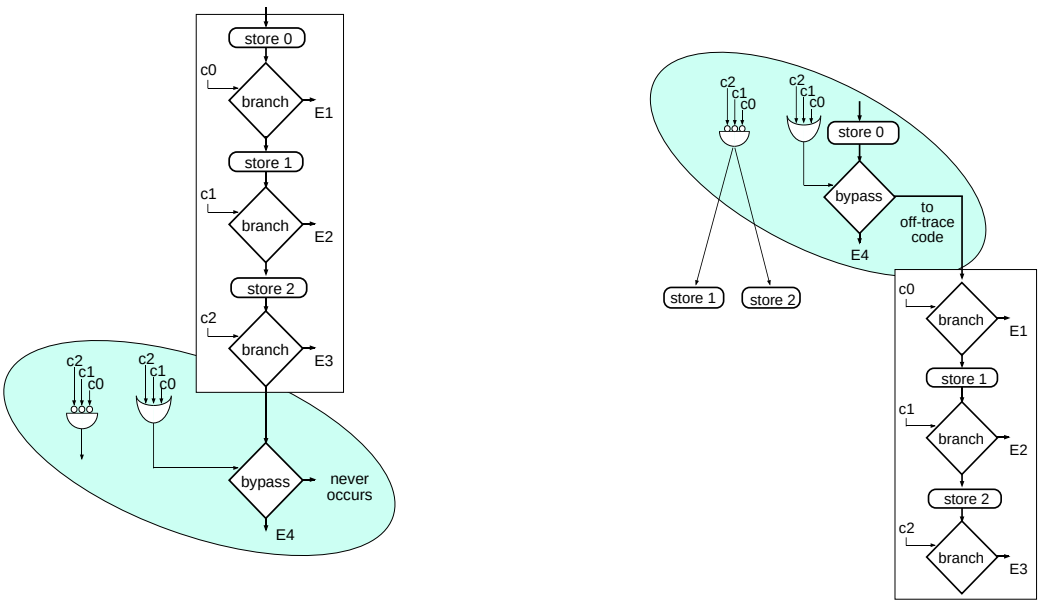
The next step of transformation is shown in Figure 2(b). Each of the original branches and any non-speculative operations dependent upon these branches are moved down across the bypass branch. Normally, code moved below a branch is replicated along both paths. However, the bypass-branch condition guarantees that if the bypass branch falls through, then none of the branches moved on trace below the bypass branch will take. After these branches are eliminated, the bypass branch is the only branch remaining on trace. Non-speculative operations (e.g. stores in the figure) that were originally trapped between branches can now be scheduled in parallel. After transformation, only a single branch remains on trace and both the on- and off-trace FRPs are computed in a height-reduced (freely re-associated) manner. Thus, control CPR reduces both branch dependence height as well as data dependence height needed to evaluate branch conditions.

There are a number of ways to implement the height-reduced computation of FRPs. On conventional processors, they can be implemented using using two-input logical operations. Height reduction uses the associative property to carefully re-organize the tree of two-input operations. The approach presented here height reduces FRP computation using the PlayDoh style wired-AND and wired-OR compares. These compares are unordered and the code motion of a static scheduler naturally re-associates FRP evaluation by accumulating input terms as they become available. Further, for wide machines, more than two terms can be combined in each machine cycle.

It is important to note that the transformation of Figure 2 can be applied either to the original superblock of Figure 1(a) or the FRP-converted superblock of Figure 1(b). This leads to a general approach for control CPR of predicated code which correctly accommodates input code of arbitrary complexity and achieves height-reduction benefits in most cases of interest including conventional and FRP-converted superblocks with embedded if-conversion. This is important because predicated execution is often introduced prior to control CPR (e.g. when if-converted intrinsics are inlined).

**Blocking.** When control CPR is uniformly applied to an entire superblock, a number of difficulties arise. First, in the context of a heuristic which requires irredundant on-trace code, it may be illegal to apply control CPR to an entire superblock. To ensure irredundant on-trace code, compares associated with exit branches that are moved off trace also move off-trace. Irredundant code is defined in more detail in the context of PlayDoh operations in Section 4.2. This motion requires that predicates computed by these compares cannot be used on trace. When redundant compares are required on trace, the code is referred to as *inseparable* and our control CPR transformation is not applied. For example, assume that the compare condition c1 in Figure 2(a) depends upon a load operation that in turn depends upon store 1. Then, the code after transformation (Figure 2(b)) is illegal because the assumed load is used to compute c1 and must execute before store 1 yet it is also dependent upon (after) the same store.

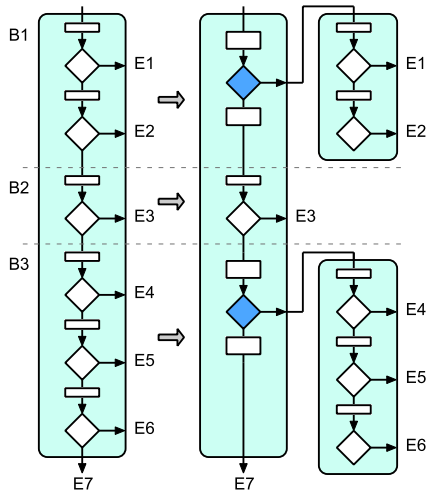
Even when correctness is not an issue, superior results are often achieved when we apply control CPR to smaller subregions. The application of control CPR can delay the execution of non-speculative operations including superblock exit branches. Exit branches are pushed below the bypass branch and out of the su-




a) insertion of bypass branch

b) final height-reduced code

**Figure 2. Control CPR schema**




a) blocked input b) transformed result

**Figure 3. Partitioning into multiple CPR blocks**

perblock. When an exit branch is taken, control CPR can introduce a performance penalty. Even when execution remains on trace, all on-trace and non-speculative operations are guarded by an on-trace FRP that is dependent upon *all* compare conditions. This can delay the execution of non-speculative operations and their dependent operations, and can compromise performance especially for long superblocks.

Blocking long superblocks into smaller subregions alleviates these problems. A *CPR block* is a linear sequence of basic blocks from the original superblock over which the control CPR is applied. Figure 3(a) illustrates a superblock prior to the application of control CPR. Dashed lines show blocking into three CPR blocks. Figure 3(b) shows the code after control CPR. Each non-trivial CPR block has been transformed into code with a single on-trace bypass branch and a compensation block, while the middle (unit length) CPR block remains unchanged. The final code is scheduled as three distinct hyperblocks, one hyperblock is scheduled for the entire on-trace region while a separate hyperblock is scheduled for each compensation block. This allows scheduling overlap between adjacent on-trace CPR blocks.

Note that when each CPR block is traversed on-trace, both on-trace and off-trace paths below this point are accelerated. Thus, the successful traversal of the first CPR block (B1) in Figure 3(b) accelerates paths to on-trace exit E7 as well as to off-trace exit E5. Thus, when CPR blocking is applied the performance of resultant code is more tolerant to unbiased branches than if control CPR were uniformly applied to the entire superblock.

## 4.2 ICBM approach

Principal goals of ICBM (Irredundant Consecutive Branch Method) are: to reduce critical-path length; and to transform code without increasing the average number of executed operations. This is accomplished in part due to the motion of branches off trace. ICBM operates on linear single-entry, multi-exit regions of code that contain predicated operations, such as superblocks and hyperblocks [H+93]. For the remainder of our discussion, we will use *hyperblock* to refer to a candidate ICBM input region.

Figure 4 shows the ICBM transformation for an input comprised of a single CPR block. The initial code is shown to the left of the arrow and transformed code is shown to the right. The symbols in the figure are as follows: squares represent the two-target compare operations, diamonds represent branches, and circles represent all other operations. The solid edges represent the control and data dependences that are necessary to explain the transformation. Conversely, the dotted edges on the right hand side of each of the shaded rectangles represent the remaining dependences that are peripheral to the transformation.

The example initial code represents a CPR block region within

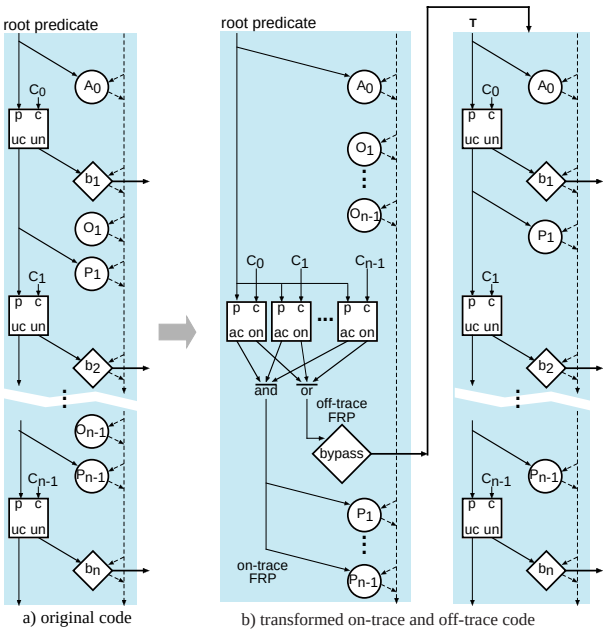



**Figure 4. Overview of the ICBM schema**

an FRP-converted superblock. The transformed code shows new operations inserted by ICBM and the code motion performed during the transformation. Now consider a basic block $i$ (other than the entry block) in the original code. The block contains a compare (with condition $C_i$), a branch ($b_{i+1}$), and sets of operations $O_i$ and $P_i$. These sets are used to precisely describe the code motion performed during ICBM. The non-compare and non-branch operations are partitioned into two sets: $O_i$ contains operations that are independent of FRPs and $P_i$ contains operations that directly or indirectly depend upon FRPs. The first basic block requires special treatment, because none of its operations are forced off trace by the motion of compares; all operations from the first block are contained in set $A_0$.

The *root predicate* holds the entry condition for a CPR block. Figure 4 represents a single CPR block taken from a sequence as in Figure 3. Each CPR block in the sequence is responsible for computing a on-trace FRP in terms of its root predicate. This on-trace FRP becomes the root predicate for the next CPR block. The root predicate for the first CPR block in a region is true.

The transformed code of Figure 4b is divided into on-trace code on the left and off-trace code on the right. On-trace code includes all operations from the first basic block ($A_0$) together with operations from subsequent basic blocks that are independent of FRPs ($O_1, \ldots, O_{n-1}$). The on-trace path also includes *lookahead compares*: operations used to compute the on-trace and off-trace FRPs. The lookahead compares implement the multi-input logical gates of Figure 2 using PlayDoh wired-and and wired-or compares. Each compare computes two results using the semantics described in Section 3. An AC term is the complement of a branch condition and'ed with the CPR block root predicate; all AC terms are wire-and'ed to form the on-trace FRP. An ON term is a branch condition and'ed with the CPR block root predicate; all ON terms are wire-or'ed to form the off-trace FRP.

The bypass branch follows the lookahead compare operations; these operations are introduced by the ICBM transformation. Finally, the added compare and branch operations are followed by operations that are dependent upon FRPs in the original code, namely operations in sets $P_1, \ldots, P_{n-1}$. Operations in these sets are replicated both on and off trace. If we consider both the on-trace and off-trace copy of operations in $P_i$, each of these operations execute under conditions in the transformed program that are identical to their execution conditions in the original program.

Note that on-trace code is said to be *irredundant* since it has fewer operations than the original code. This can be seen by first considering the sets of operations, A, O and P. Every operation within these sets in the original code appears within these sets in the on-trace code. For each compare operation in the original code, a single height-reduced compare operation appears in the on-trace code. Finally, all of the branches in the original code are replaced by a single bypass branch in the on-trace code. The net effect is to conserve the operation count except that $n$ branches in the original code are replaced by one branch in the on-trace code. It is in this sense that the code is considered irredundant (and in fact reduced operation count) for the target PlayDoh architecture.

Off-trace code consists of each of the compares in the original code and all operations that were dependent on those compares. Because the original compares are moved off trace to eliminate redundancy, operations that are dependent upon the compares also move off trace.

# 5 ICBM Implementation

The ICBM schema has been implemented within Elcor, our compiler for research in high-performance EPIC architectures. ICBM accepts general single-entry linear regions as input; common examples are conventional superblocks, FRP-converted superblocks, and hyperblocks. The ICBM transformation consists of a sequence of four phases which either analyze or transform code: 1) predicate speculation, 2) match, 3) restructure, and 4) off-trace motion. After ICBM, a pass of dead code elimination removes any unnecessary operations, such as operations that compute predicates which are not referenced.

The ICBM code modules take advantage of Elcor's family of predicate cognizant analysis tools. Classic tools for data-flow analysis and dependence edge construction have been upgraded to analyze predicated code in a conservative (with respect to correctness) yet reasonably accurate manner. Without these enhancements, the benefits of predicate-based control CPR would not be realized.

## 5.1 Predicate speculation

ICBM begins with predicate speculation. Predicate speculation serves two purposes. First, it reduces dependence height by eliminating an operation's dependence on its predicate calculation [MLC+92]. More importantly, predicate speculation eliminates dependences that would inhibit ICBM's separability condition (discussed in the next sub-section). Often, the predicate for a basic block guards operations required to compute the predicate for the next block. These dependences prevent compares from moving off trace during ICBM and constitute a separability failure. In FRP-converted code, separability systematically fails at almost every basic block. Predicate speculation removes most of these dependences allowing separability to pass more frequently.

Predicate speculation operates in two bottom-up traversals of a hyperblock. An array, called *liveness*, contains boolean expressions representing predicate conditions under which each register or memory location is live [JS96]. Initially, liveness is only available at hyperblock exit points, and is computed on the fly at each point (i.e. at each operation) during the backward traversal. All operations are candidates for promotion with the exception of compare-to-predicate operations which unconditionally compute result predicates from input predicates.

In the first pass, the guarding predicate (p) for each operation is conditionally promoted to a predicate (q) such that p implies q (or q is *larger* than p). While in principle, predicate promotion could promote a predicate to a variety of larger predicates, the only larger predicate considered here is predicate true. For each operation, the promotion occurs only if the operation will not overwrite a live register or memory value when it is promoted.

The second pass selectively demotes predicates. Demotion is the inverse of promotion: an operation's guard is demoted to a predicate that evaluates to true less frequently. The operation may be partially demoted or fully demoted (i.e. returned to its original guard). Demotion provides two benefits: first, it may reduce dependence height, and second it may demote an operations execution condition without adding dependence height. Demoting an operations predicate produces second order benefits such as reduced memory traffic, fewer cache misses, as well as improved predicate sensitive register allocation.

Demotion is demonstrated by a simple example. Consider two dependent operations initially guarded by the same predicate. During predicate speculation, assume that the first operation's predicate cannot be promoted without violating correctness while the second operation's predicate is promoted to true. However, since the second operation depends upon the first, this speculation does not reduce height. During the second pass, the predicate of the second operation is lowered to its original value. Where there is a data dependence between the these operations, demotion undoes ineffective promotion without changing dependence height. Where there is a branch dependence after the first pass, demotion reduces dependence height by replacing the branch dependence with a data dependence on the branch's guarding compare.

## 5.2 Match

Match identifies CPR blocks within a program region which is to be transformed. CPR block identification addresses two major issues: correctness and performance heuristics. Match processes hyperblocks creating a description of a transformation to be subsequently performed by the CPR transformation module. The result of match is a list of CPR blocks, where each CPR block is a subregion of the hyperblock that is to be independently transformed.

A preliminary pass within match generates a list of branches in the hyperblock in their sequential order. Next, reaching-definition analysis is performed on predicate variables. For every branch or compare operation, this analysis identifies the unique compare-to-predicate operation that computes the guarding predicate, if such an operation exists within the region. This allows compares that do not control branches (e.g. those used for conventional if-conversion in hyperblocks) to be ignored.

In a subsequent pass, match grows a list of CPR blocks to cover all branches in the hyperblock. The process is seeded with a CPR block containing only the first branch. Match grows CPR blocks by appending consecutive branches until an exit condition terminates the block. The process is re-seeded, with the next branch (not yet appended into any CPR block), and continues until all branches in the original hyperblock have been treated. Pseudo code for match is shown in Figure 5.

Match performs a number of tests including the *suitability*, *separability*, *exit-weight* and *predict-taken* tests. Each of these tests can terminate a CPR block. The suitability and separability tests guarantee that the transformation can be correctly applied, while the exit-weight and predict-taken tests are heuristics to improve performance. The correctness tests are designed to detect situations where ICBM can be applied and guaranteed to produce both correct and efficient code. Other approaches might broaden the applicability of control CPR by generalizing ICBM.

**Suitability.** The suitability test generalizes control CPR to handle input programs containing arbitrary predication. Suitability always passes when match processes simple superblocks or FRP-converted superblocks. However, in the presence of more complex uses of predicates, the suitability test ensures that CPR blocks use branch predicates and branch conditions in a fashion consistent with the production of correct code using the CPR schema of Figure 4.

Consider the following linear sequence of compare and branch operations taken from a hyperblock:

```
    pf1, pt1 = cmpp.uc.un(bc1) if pg0
    branch <exit 1> if pt1
    pf2, pt2 = cmpp.uc.un(bc2) if pg1
    branch <exit 2> if pt2
    ...
    pfn, ptn = cmpp.uc.un(bcn) if pg(n-1)
    branch <exit n> if ptn
FTEXIT:
```

The ICBM schema must generate code which computes an off-trace FRP for the bypass branch; this FRP must be true exactly when any of the branches in the CPR block takes. The $i^{th}$ branch takes exactly when $bc_i$ and $pg_{i-1}$ are both true. Any one of the exit branches takes when $((pg_0 \wedge bc_1) \vee \ldots \vee (pg_{n-1} \wedge bc_n))$. If guards and conditions for the compares which compute branch predicates are arbitrary, then we must evaluate this fully-general expression for the off-trace FRP.

The ICBM schema more efficiently addresses commonly occurring predicate usage by actually generating code for the off-trace FRP using the simpler expression $(pg_0 \wedge (bc_1 \vee bc_2 \vee \ldots \vee bc_n))$. Suitability guarantees that this simpler expression produces correct code before the control CPR transformation is applied to a given CPR block. When suitability fails, code is left unchanged over an input subregion in order to ensure correctness rather than generating the more complex fully-general expression for the off-trace FRP.

Suitability is divided into an initialization step which treats a CPR block of length one, and a growth step which decides whether the CPR block can be legally augmented with the next branch operation. The suitability test is initialized as follows. A current branch pointer is initialized to point at the first branch in a length-one CPR block. A *suitable predicate set* (SP) is initialized to the

```
Procedure ICBM_match
{
1:    final_branch_in_prev_block = 0 ;
2:    result.clear() ; // result is initially null
3:    while (TRUE) // form a list of cpr blocks
4:       first_branch = final_branch_in_prev_block + 1 ;
5:       curr_branch = first_branch ;
6:       if (curr_branch > total_number_of_branches)
7:          break;
8:       cpr_block.clear() ; // initialize a new CPR block
9:       cpr_block.append(curr_branch) ; // defines seed branch
10:      suitability_test_init(curr_branch) ;
11:      separability_test_init(curr_branch) ;
12:      exit_weight_test_init(curr_branch) ;
13:      pred_taken_flag = FALSE ;
14:      while (TRUE) // grow CPR block from seed
15:         if (pred_taken_flag)
16:            break;
17:         cand_branch = curr_branch+1 ;
18:         if (suitability_test_failure(cand_branch))
19:            break;
20:         if (separability_test_failure(cand_branch))
21:            break;
22:         if (predict_taken(cand_br))
23:            pred_taken_flag = TRUE ;
24:         if (!pred_taken_flag && exit_weight_test_failure(cand_branch))
25:            break;
26:         // passed all tests, append tuple to CPR block
27:         cpr_block.append(curr_branch) ;
28:         curr_branch = cand_branch ;
29:      endwhile
30:      result.append(cpr_block) ;
31:      final_branch_in_prev_block=curr_branch ;
32:   endwhile
33:   return result ;
}
```

**Figure 5. Match pseudo code**

empty set. For the initial branch (i.e. the $0^{th}$), if the controlling compare operation unconditionally computes its branch guard predicate (i.e. using the UN target modifier), then the compare operation's guarding predicate is added to SP. This guarding predicate is the CPR block's root predicate, as shown in Figure 4. If the compare operation also unconditionally computes a complementary fall-through predicate (i.e. using the UC modifier), then the fall-through predicate is added to SP as well.

The following three conditions form an induction hypothesis which is readily verified for an initial CPR block of length one: (1) if $pg_0$ is false then all members of SP (if any) are also false; (2) if $pg_0$ is true and no exit branch is taken, then all members of SP are true; (3) the off-trace FRP computed as $pg_0 \wedge (bc_1 \vee \ldots \vee bc_n)$ is true exactly when one of the branches in the CPR block takes.

We continue growing the current CPR block by inspecting candidate branches in order. For the current branch (i.e. the $i^{th}$), if the controlling compare operation unconditionally computes its branch-guarding predicate (i.e. using the UN target modifier) and if the compare's guarding predicate $pg_i$ is in SP, then the candidate branch can be appended to the current CPR block. Otherwise, growth of the current CPR block is terminated, and the current branch becomes the initial seed branch for a subsequent CPR block. If the compare operation also unconditionally computes a complementary fall-through predicate (i.e. using the UC modifier), then the fall-through predicate is added to SP. One can show that when a candidate branch is appended to a CPR block, all three conditions of the induction hypothesis remain true. Thus, CPR blocks formed after passing suitability have the property that their schematically generated off-trace FRP is true exactly when one of the branches in the CPR block takes.

**Separability.** The separability test is also needed to ensure the correctness of the control CPR schema. The CPR transformation shown in Figure 2 moves compares from the original code off-trace and replaces them with lookahead compares which remain on-trace. If improperly applied, the code motion required by the ICBM schema can violate dependence constraints. This might occur when a branch condition required to compute on-trace and off-trace FRPs is dependent upon a predicate which, after ICBM, is computed only off-trace. Since ICBM is designed to produce irredundant code by moving the original compares off-trace, dependences from a compare which will be moved off-trace to a lookahead compare which must remain on-trace are not allowed.

The separability test repeatedly applies the function append-successors which, for a given branch, computes a set operations which are dependent upon the compare which guards the branch. Note that in the control CPR schema, the off-trace FRP is computed as: $pg_0 \wedge (bc_1 \vee bc_2 \vee \ldots \vee bc_n)$, only the guard for the initial compare operation $(pg_0)$ is used. All guards for subsequent compares $(pg_1, \ldots, pg_{n-1})$ are ignored due to the successful application of the suitability test. Thus, when append-successors computes a set of successors from a given compare, any dependence resulting from the use of the compare's fall-through predicate as the guard of a compare which in turn guards a subsequent branch can be ignored.

The separability test is also divided into an initialization step which treats a CPR block of length one, and a subsequent separability step which decides whether the CPR block can be legally augmented with an additional basic block. During initialization, a set called *succ* is initialized to the null set, and append-successors is invoked on the initial branch, which is unconditionally included in the CPR block. After computing the appropriate set of successor operations for the compare guarding the initial branch, this set is accumulated into *succ*.

The separability step is invoked to test each subsequent candidate branch for inclusion into the current CPR block. First, the compare which guards the candidate branch is tested for membership in *succ*. If the compare is a member of *succ*, separability is violated and the candidate branch cannot join the CPR block. Otherwise, the candidate branch may be included in the CPR block and append-successors is invoked on the candidate branch thus accumulating appropriate successors of the compare guarding the branch into *succ*.

**Exit-weight and predict-taken tests.** The exit-weight and predict-taken tests are heuristics which truncate the formation of a CPR block using branch profile data which provides taken and not-taken frequencies for each branch. The exit-weight test monitors the ratio of the cumulative exit frequencies of all of the branches within the CPR block divided by the entry frequency into the CPR block. When the inclusion of a candidate branch into a CPR block causes this ratio to exceed a threshold, the candidate branch is not included and CPR block growth is terminated.

The predict-taken test identifies likely exits in the input code that are selected as the final branch in a CPR block. Such a CPR block is tagged as a likely-taken CPR block and receives special

treatment during ICBM restructure. The code generation and code motion schema for the likely-taken CPR block allows a CPR block to reach a predicted taken branch target without first branching to an off-trace compensation block and again branching from the compensation block to the final target.

The predict-taken test monitors the ratio of the candidate branch exit frequency divided by the CPR block entry frequency. When this ratio exceeds a threshold, a likely taken CPR block is formed. This test takes priority over the exit-weight test, which would otherwise truncate the CPR block. Further, when the candidate branch is identified as satisfying all conditions required for a predict-taken CPR block, growth is terminated after the candidate branch is appended to the current CPR block.

### 5.3 Restructure

Restructure performs the actual height-reducing transformation on each non-trivial CPR block identified in the previous step. This phase introduces all new operations including the lookahead compares and the bypass branch. The re-wiring of guarding predicates and branch conditions to these new predicates is also performed during this phase. Lastly, a new region known as the *compensation block* is created to hold the off-trace code after code motion. Two variations of CPR blocks are treated: first, a branch sequence in which all the branches are likely fall-through (*fall-through variation*); and second a branch sequence in which all are likely fall-through except the last which is likely-taken (*taken variation*).

Restructure is rather mechanical in nature. It consists of the following steps that are performed for each CPR block: insert on-trace and off-trace predicate computation; create compensation block; insert bypass branch; re-wire guarding predicates. These steps are explained for the fall-through variation. Afterwards, changes for the taken variation are described.

Two predicates, the on-trace and off-trace FRPs, are introduced for each CPR block. These hold the conditions that execution remains on-trace (on-trace FRP evaluates to true) or goes off-trace (bypass branch FRP evaluates to true) in the course of the entire CPR block. The on-trace FRP is computed as the conjunction of the fall-through conditions of the branches using wired-and semantics. The off-trace FRP is computed as the disjunction of the taken branch conditions using wired-or semantics. These FRPs are computed using lookahead compares inserted after each of the original branch compares. Each lookahead compare uses the same condition and source operands as the original. All of the lookahead compares target the new on-trace and off-trace FRPs using the dual target AC/ON semantics. One subtlety of the transformation is that each of the lookahead compares is not guarded by the predicate of the corresponding original compares. Rather, all are guarded by the root predicate of the CPR block. This substitution is legal due to the success of the suitability test.

The use of wired-and and wired-or predicates require that they be properly initialized. The off-trace predicate (wired-or) is simply initialized to 0. The on-trace predicate (wired-and) is initialized to the root predicate of the CPR block which is strictly greater than the wired-and result.

The next two steps create the compensation block and insert a conditional branch to that block. An empty compensation block is added to the function body. The bypass branch is added as a conditional branch to the compensation block that occurs when the off-trace FRP evaluates to true. This branch is inserted immediately after the final branch within the original CPR block.

The final step of the restructure eliminates uses of predicates computed by the original compares from operations subsequent to the bypass branch. One of the goals of the schema is to allow the original compares to move off-trace to eliminate redundancy. However, this can only be accomplished if there are no uses in the remainder of the hyperblock of the predicates they compute. It can be shown that such uses of any of the original predicates can be safely replaced by a use of the on-trace FRP.

**Taken variation.** Several small changes to restructure are necessary for the taken variation. Instead of accelerating the fall-through direction of the final branch, the taken direction is accelerated. To accomplish this, the sense of the final lookahead compare is inverted, e.g., a less-than condition in the original compare becomes a greater-than-or-equals in the new compare. The more interesting part of the variation is that a new bypass branch is not required. Rather, the last branch in the CPR block serves as the bypass branch. Its taken condition corresponds to remaining on-trace, and its fall-through corresponds to going off-trace. As a result, a new compensation block is also not required. Instead, the remainder of the hyperblock serves as the compensation block.

### 5.4 Off-trace motion

After the preceding height-reducing transformations are complete, redundant operations are moved off-trace to benefit the more likely on-trace path. This motion is performed prior to scheduling so that we can use our existing superblock/hyperblock scheduler. An alternate approach would rely on a tree-region scheduler [HBC98] to perform code motion between the height-reduced hyperblock and its associated compensation blocks.

Three passes are performed over the hyperblock region to identify the set of operations that must move off trace, and to further identify the subset of moved operations that must be split, so that a copy remains on trace. A final step performs the required operation splitting and code motion. The steps are as follows. First, identify all data dependence successors of the compare and branch operations that must be moved off-trace (set 1). Second, identify a subset of the operations in set 1 that produce a value that is also needed on-trace (set 2). These operations must be split or replicated along both paths. Stores are the most common operations that require splitting. Third, identify any of the remaining operations not in set 1 whose results are used only along the off-trace path, since their motion off-trace will benefit the on-trace path (set 3). Finally, move operations in sets 1 and 3 to compensation blocks, while replicating operations in set 2.

## 6 Code Example

The application of ICBM is illustrated in this section using a simple code example. The example chosen is the inner loop for a common string copy routine. The example source, shown in Figure 6(a), is a while loop which copies elements of string A into string B. To expose instruction-level parallelism, the loop body is unrolled four times. PlayDoh assembly code after unrolling and other traditional code optimizations is shown in Figure 6(b). Each iteration stores a current value into array $B$, loads the next value from array $A$, computes the necessary addresses, and conditionally

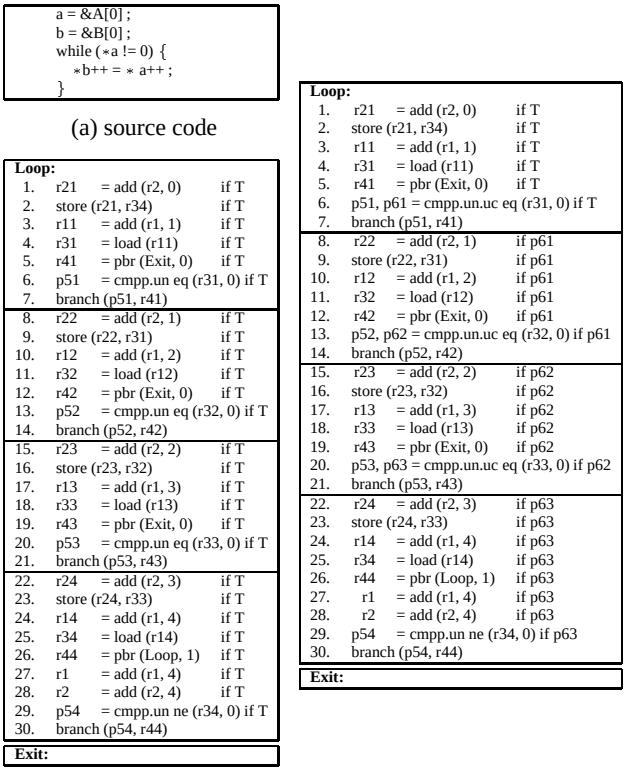



**Figure 6. Example: transformation applied to strcpy**

exits the loop after the end of *A* is reached. In PlayDoh, conditional branches are realized using three operations: a prepare-to-branch (op 5), a comparison (op 6), and a predicated control transfer (op 7). For the last iteration, two additional operations increment the array pointers by the unroll amount (four). At this point, all operations are guarded by the predicate true (denoted by `if T`).

Figure 6(c) shows the FRP-converted superblock. FRP conversion is accomplished using cmpp operations which generate two predicate outputs: the taken condition (UN output) and the fall-through condition (UC output). Operations which were dependent on the branch are now guarded by the fall-through condition to complete the transformation. For example, in Figure 6(c), op 6 has a second target, p61, that generates the fall-through condition; operations that were dependent on op 7, namely ops 8-13, are now guarded by p61. The code in Figure 6(c) represents the preferred input for the ICBM schema.

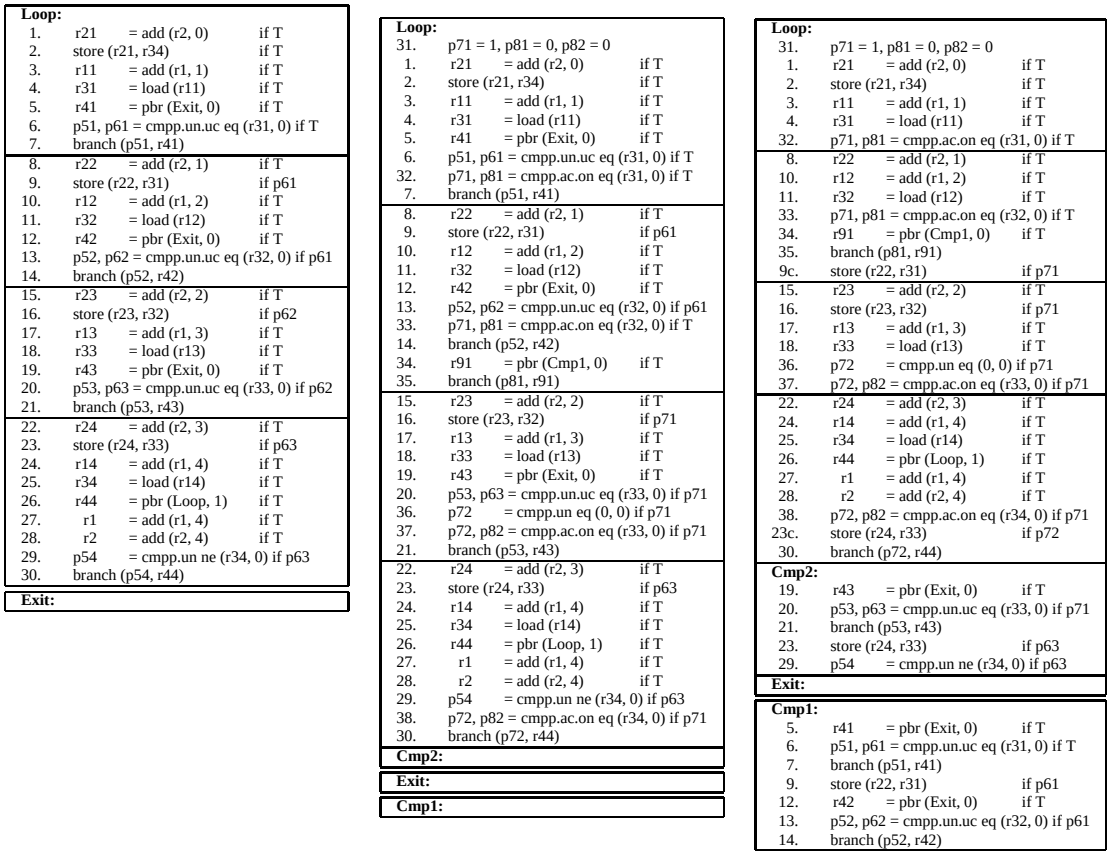
**Predicate speculation.** The first phase of the ICBM schema is predicate speculation. Predicate speculation is applied to the FRP-converted superblock in Figure 6(c). The resultant code after speculation is shown in Figure 7(a). This example is somewhat uninteresting from the viewpoint of predicate speculation. In the first pass of predicate speculation, all eligible operations in the block are promoted to true. Note that promotion of operations within an FRP-converted superblock is always legal, since promotion faithfully mirrors the original code. The second pass demotes the predicate of ops 9, 16, and 23 back to their original value. Each of these stores are dependent on a prior branch. For instance, op 16 is dependent on op 14 and speculating op 16 to true was not useful. Demotion lowers op 16's predicate to the branch fall-through predicate, p62. Not only does demotion undo a useless promotion, it also lowers dependence height by enabling the store and branch to be freely reordered.

**Match.** The next phase of the ICBM schema is to apply match to the code in Figure 7(a). Match identifies the CPR blocks or the set of subregions that will be transformed. Recall that match consists of a set of four tests (suitability, separability, exit weight and predict taken) that are conditions for terminating a CPR block. For this example, the only predicates that are used are those generated via FRP conversion and thus the suitability test succeeds across the entire block. In addition, there are no dependences that cause the separability test to fail. It is illustrative to note that if the compiler could not determine that a store and a subsequent load were independent in this example, the separability test would fail. For instance, assume there is an alias between operations 16 and 18. The match algorithm would attempt to append the compare/branch tuple 20/21 to the CPR block containing the previous two compare/branch tuples (6/7, 13/14). However, there is a chain of dependences connecting ops 13 and 20 (13 to 14 via a flow dependence, 14 to 16 via a control dependence, 16 to 18 via the assumed memory dependence, and 18 to 20 via a flow dependence). This would cause a separability violation preventing the addition of the compare/branch tuple.

The exit weight and predict taken tests are based on branch profile data. For this example, it is assumed the last branch (op 30) is predominantly taken since it is the loop back branch. Further, it is assumed the exit weight threshold is exceeded by the second branch (op 14), but its predominant direction is fall-through. As a result, the match algorithm identifies two CPR blocks in Figure 7(a). The first CPR block includes the first two branches and will use the fall-through restructure schema. The second CPR blocks includes the last two branches and will use the taken restructure schema. In the actual application of ICBM, such small CPR blocks are not typically formed. However, it is done in this example to jointly illustrate both the fall-through and taken restructure schemas.

**Restructure.** The next phase of the ICBM schema is to restructure (apply the control CPR transformation) to each of the CPR blocks. The overall result of the restructure is shown in Figure 7b. Focusing on the first CPR block, the new on-trace and off-trace predicates are p71 and p81, respectively. After each of the original cmpps that compute the the branch conditions (ops 6 and 13), the lookahead AC/ON cmpps are inserted (ops 32 and 33). The lookahead cmpps look identical to the original cmpps in terms of source operands and compare condition. The lookahead cmpps are guarded by the root predicate of the CPR block. Since this is the first CPR block in the hyperblock, the root predicate is true.

Again, focusing just on the first CPR block, op 35 is the bypass branch and the new block labeled *Cmp1* is the compensation block. The bypass branch is inserted after the last branch in the CPR block. The bypass-branch predicate is p81. Note that the PlayDoh architecture requires a prepare-to-branch operation for each branch, hence op 34 is inserted in conjunction with the bypass branch. The final phase of restructure rewires operations in the hyperblock subsequent to the last branch in the CPR block (op 14) that use predicates computed by the original cmpps. For this




(a) after predicate speculation (b) after restructure (c) after off-trace motion

**Figure 7. Example, continued: transformation applied to strcpy**

example, ops 16 and 20 are guarded by p62. The guards are modified to the new on-trace predicate, p71 to accomplish the necessary re-wiring. After this substitution is complete, there are no uses of the original predicates (p51, p61, p52, p62) outside the CPR block.

The taken variation of restructure is illustrated in the second CPR block in Figure 7(b), where op 30 is likely taken. Here, op 30 serves as the bypass branch and the compensation area denoted by the label *Cmp2* is just the tail portion of the hyperblock. The lookahead cmpps are inserted after the original cmpps using the same approach as the fall-through variation. Except, the sense of the last cmpp (op 38) is inverted from its original counterpart (op 29). The condition under which the final branch takes serves as the new on-trace predicate, p72. Note that op 36 is the initialization for the on-trace predicate, p72. This operation initializes p72 to the root predicate of the CPR block, namely p71.

**Off-trace motion.** The final phase of the ICBM schema is to apply off-trace motion to each restructured CPR block. Off-trace motion removes all the redundant operations from the on-trace path. Off-trace motion is applied to Figure 7(b). In the first CPR block, the two original branches (ops 7 and 14) and their compares (ops 6 and 13) are redundant so they will move off-trace. The store operation (op 9) must also move off-trace, since it is a successor of the first cmpp. Additionally, operations 5 and 12 are moved off-trace, since their results are not live on-trace. The final move set contains ops 5, 6, 7, 9, 12, 13, and 14. Since the store operation (op 9) is also needed on-trace, it must be split. By similar analysis, the move set for the second CPR block contains ops 19, 20, 21, 23, and 29; operation 23 is needed on-trace, so it is split. Figure 7(c) shows the final code after off-trace motion and splitting.

**Dead code elimination.** After ICBM is complete, a phase of dead code elimination is needed to remove unnecessary operations. In Figure 7(c), predicates p54 and p62 are dead; hence, dead code elimination removes op 29 and removes the second destination of op 13.

**Summary.** Figures 6 and 7 illustrate the application of the ICBM schema for control CPR to an unrolled version of a string copy loop. The initial unrolled code (Figure 6(b)) consists of 30 operations in the loop. The final code (Figure 7(c)) contains of a total of 28 operations in the loop itself and an additional 11 operations in compensation blocks. Furthermore using the operation latencies described in the next section, the transformation reduces the dependence height through the loop from 8 to 7 cycles. Overall, the ICBM transformation has both reduced the number of operations and the dependence height along the dominant execution path. The cost of this transformation was an increase in the static code size of 9 operations. The achieved height reduction in string copy experiments is substantially larger than that achieved in the example due to a larger unroll factor for the loop and a more suitable selection of CPR blocks.

| Benchmark | Seq | Nar | Med | Wid | Inf | Benchmark | Seq | Nar | Med | Wid | Inf |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 008.espresso | 1.15 | 1.04 | 1.08 | 1.14 | 1.15 | 134.perl | 1.06 | 1.05 | 1.10 | 1.12 | 1.12 |
| 022.li | 1.08 | 1.03 | 1.04 | 1.06 | 1.06 | 147.vortex | 1.12 | 1.02 | 1.08 | 1.14 | 1.14 |
| 023.eqntott | 0.85 | 0.87 | 1.10 | 1.23 | 1.23 | cccp | 1.11 | 1.10 | 1.36 | 1.50 | 1.58 |
| 026.compress | 0.95 | 1.05 | 1.15 | 1.16 | 1.17 | cmp | 1.53 | 1.25 | 1.79 | 2.87 | 3.60 |
| 056.ear | 1.09 | 1.01 | 1.12 | 1.33 | 1.52 | eqn | 1.16 | 1.06 | 1.15 | 1.24 | 1.26 |
| 072.sc | 1.16 | 1.07 | 1.16 | 1.21 | 1.23 | grep | 1.26 | 1.03 | 1.32 | 2.11 | 2.61 |
| 085.cc1 | 1.13 | 1.06 | 1.12 | 1.15 | 1.18 | lex | 1.29 | 1.08 | 1.34 | 1.97 | 2.26 |
| 099.go | 0.96 | 1.01 | 1.02 | 1.02 | 1.02 | strcpy | 1.73 | 1.27 | 1.53 | 2.76 | 4.26 |
| 124.m88ksim | 1.15 | 1.07 | 1.10 | 1.12 | 1.13 | tbl | 1.02 | 0.99 | 1.06 | 1.13 | 1.14 |
| 126.gcc | 1.02 | 1.03 | 1.06 | 1.07 | 1.07 | wc | 1.17 | 1.07 | 1.31 | 1.34 | 1.34 |
| 129.compress | 1.10 | 1.03 | 1.08 | 1.12 | 1.14 | yacc | 1.15 | 1.05 | 1.26 | 1.40 | 1.46 |
| 130.li | 1.06 | 1.06 | 1.07 | 1.07 | 1.07 | Gmean-spec95 | 1.07 | 1.04 | 1.08 | 1.10 | 1.11 |
| 132.ijpeg | 1.11 | 1.08 | 1.12 | 1.16 | 1.21 | Gmean-all | 1.13 | 1.05 | 1.18 | 1.33 | 1.41 |

**Table 2. The effectiveness of ICBM for processors with branch latency 1.**

# 7 Experiments

We now present results taken from our ICBM implementation within the Elcor compiler. Elcor and its machine description language allow us to compile for a broad range of EPIC processors. For these experiments, we restrict our discussion to a class of regular machines defined by four parameters: $I$ denoting the number of integer units, $F$ denoting the number of floating-point units, $M$ denoting the number of memory units, and $B$ denoting the number of branch units. We define the following regular EPIC processors by their $(I, F, M, B)$ tuple: *narrow* is $(2, 1, 1, 1)$; *medium* is $(4, 2, 2, 1)$; *wide* is $(8, 4, 4, 2)$, and *infinite* is $(75, 25, 25, 25)$. One additional processor is defined, the *sequential* processor issues exactly one operation of any type per cycle.

Operation latencies are as follows: simple integer - 1, simple floating point - 3, memory load - 2, memory store - 1, integer and floating point multiply - 3, and integer and floating point divide - 8. Branch latencies are 1.

The benchmarks studied consist of a set of SPEC-92 as well as SPEC-95 applications and common Unix utilities. Spec92 applications include: *008.espresso*, *022.li*, *023.eqntott*, *026.compress*, *056.ear*, *072.sc*, and *085.cc1*. Spec 95 applications include *099.go*, *124.m88ksim*, *126.gcc*, and *129.compress*, *130.li*, *132.ijpeg*, *134.perl*, *147.vortex*. Unix utilities include: *cccp*, *cmp*, *eqn*, *grep*, *lex*, *strcpy*, *tbl*, *wc*, and *yacc*.

Benchmark performance is derived using a compiler estimation approach. Code is first scheduled for each processor configuration. Then, performance is computed using static schedule lengths and profile data. The benchmark execution time is calculated as the sum across all blocks in the program of each block's schedule length weighted by its dynamic execution frequency. Benchmark performance ignores dynamic effects such as stall cycles associated with the instruction cache, data cache, or branch predictor. The measurements weight all application code. Where control CPR has not been applied, the performance of the unoptimized code is measured. Previous experience with this method has shown that it accurately determines the performance obtained via simulation of an equivalent, statically-scheduled processor where dynamic effects are ignored.

The experiments evaluate the effects of control CPR by comparing the performance of two compiled codes: baseline and height-reduced. The baseline code is optimized superblock code produced by the IMPACT compiler [H+93]. The height-reduced code is the baseline code to which FRP conversion and the ICBM schema are applied.

**Results.** The effectiveness of control CPR using ICBM is shown in Table 2. The speedup observed after applying control CPR across a spectrum of EPIC processors is presented. Speedup is calculated by dividing the execution cycles of the baseline code by that of the height-reduced code for each processor. Each data point represents the fraction of performance improvement that is observed with control CPR for each application on a particular processor. The geometric means of the performance across the SPEC-95 applications (Gmean-spec95) and across all applications (Gmean-all) are also shown.

The table shows that control CPR is highly effective for all processors. Mean speedups across all the benchmarks of 13%, 5%, 18%, 33%, and 41% are achieved for the sequential, narrow, medium, wide, and infinite processors. For the sequential processor, the speedup is largely due to the reduction in the number of operations. Branches and other operations that are only needed off-trace are removed from the main path by the transformation. At the other extreme, the largest speedups are observed for the infinite processor which fully exposes program dependence height. Four of the benchmarks, *cmp*, *grep*, *lex*, and *strcpy* achieve more than a factor of two improvement with control CPR.

Branch intensive programs with highly biased branches and separable computation of branch conditions invariably exhibit large speedups with control CPR. This is especially true on processors with substantial hardware parallelism. Such programs include *cccp*, *cmp*, *grep*, *lex*, *strcpy*, *wc* and *yacc* where speedups of 1.34 to 2.87 are achieved for the wide processor. The combination of highly biased branches and separable computation of branch conditions provides an underlying program structure where control CPR can be applied to large groups of branches. Further, programs with a large fraction of branches exhibit little parallelism due to the large number of branch dependences. The combination of these features provide an ideal environment for control CPR to be effective. Control CPR is able to substantially reduce branch dependence height in these programs which is translated directly into performance gains through increases in parallelism.

A number of effects contribute to the lower observed speedups

| Benchmark | S tot | S br | D tot | D br | Benchmark | S tot | S br | D tot | D br |
|---|---|---|---|---|---|---|---|---|---|
| 008.espresso | 1.10 | 1.06 | 0.98 | 0.39 | 134.perl | 1.01 | 1.01 | 0.97 | 0.66 |
| 022.li | 1.03 | 1.01 | 0.99 | 0.63 | 147.vortex | 1.02 | 1.01 | 0.91 | 0.62 |
| 023.eqntott | 1.11 | 1.04 | 1.04 | 0.54 | cccp | 1.10 | 1.06 | 0.88 | 0.39 |
| 026.compress | 1.14 | 1.06 | 1.06 | 0.61 | cmp | 1.08 | 1.01 | 0.71 | 0.13 |
| 056.ear | 1.06 | 1.03 | 0.94 | 0.35 | eqn | 1.03 | 1.01 | 0.91 | 0.48 |
| 072.sc | 1.05 | 1.02 | 0.92 | 0.52 | grep | 1.12 | 1.03 | 0.85 | 0.15 |
| 085.cc1 | 1.05 | 1.02 | 0.97 | 0.63 | lex | 1.12 | 1.04 | 0.83 | 0.20 |
| 099.go | 1.08 | 1.04 | 1.04 | 0.86 | strcpy | 1.16 | 1.00 | 0.61 | 0.07 |
| 124.m88ksim | 1.03 | 1.02 | 0.99 | 0.44 | tbl | 1.06 | 1.03 | 1.00 | 0.65 |
| 126.gcc | 1.05 | 1.02 | 1.01 | 0.81 | wc | 1.20 | 1.08 | 0.94 | 0.40 |
| 129.compress | 1.19 | 1.08 | 0.99 | 0.53 | yacc | 1.15 | 1.07 | 0.95 | 0.36 |
| 130.li | 1.04 | 1.02 | 1.02 | 0.66 | Gmean-spec95 | 1.06 | 1.03 | 0.98 | 0.62 |
| 132.ijpeg | 1.07 | 1.05 | 0.93 | 0.51 | Gmean-all | 1.08 | 1.03 | 0.93 | 0.42 |

**Table 3. The effect of ICBM on the static and dynamic operation counts for the medium processor.**

in application measurements. A primary cause for low speedups is difficulty in accurate identification of static program traces caused by unbiased branches. This results in either very short CPR blocks eliminating much of the benefit of control CPR; or, it results in frequent branching into off-trace compensation blocks which can reduce performance. The benchmark, *099.go* exhibits this effect most prominently as it is dominated by unbiased branches.

Another cause of low speedups, particularly for the sequential and narrow processors, is the use of a single set of CPR block selection heuristics for all the processors. The heuristics were tuned to accelerate programs for the medium processor and then directly applied for all the measured processors. The medium processor derives substantial height-reduction benefit from the formation of large CPR blocks. A side effect of large CPR blocks is that modestly important exit branches are delayed by motion off trace. For wider processors, this delay is more than made up for by the increased parallelism.

For sequential and narrow processors, large CPR blocks often cause performance problems. The increased height reduction that is achieved cannot be exploited due to the limited processor resources. Further, performance is hurt by delaying the execution of exit branches through off-trace motion. The net effect is code with poor performance for the sequential and narrow processors that becomes substantially better on processors with more hardware resources. The benchmark which most notably illustrates this behavior is *023.eqntott* where significant performance loss for the sequential and narrow machines is observed. But, the loss is converted into a substantial gain for the medium, wide, and infinite processors. The further development of distinct heuristics for each machine configuration would alleviate this problem.

Finally in Table 2, it is interesting to note that the speedups with control CPR on the narrow processor are consistently less than those on the sequential processor. This behavior is due primarily to the nature of the reduced work introduced by control CPR and the nature of the processors. For the sequential processor, each operation eliminated from the on-trace path contributes directly to performance improvement since the processor can only sustain one operation per cycle. Conversely on the narrow machine, there are dedicated units for each operation type. Thus, eliminating a branch operation may not produce any performance improvement unless the branch unit is saturated. For wider processors, this behavior disappears. With more resources, the performance gain due to eliminating operations becomes less important, while the reduction in dependence height is the dominant source of the performance gain.

Table 3 presents the effects of control CPR on both the static and dynamic operation counts for all operations as well as for only branch operations. Columns are labeled ”S tot” (static all operations), ”S br” (static branch), ”D tot” (dynamic all operations), ”D br” (dynamic branches) to indicate the nature of the measurement taken. Rows are labeled with the benchmark name. Each cell shows the ratio of the number of operations of the height-reduced code to the number of operations in the baseline code. Again the geometric means of the ratios across the SPEC-95 applications (Gmean-spec95) and across all applications (Gmean-all) are shown.

Of particular interest are the dynamic operation counts for branches. For the set of small programs dominated by predictable branches, we see that the number of executed branches is greatly reduced in the range .07 to .40 times the number of original branches. Even for the larger applications, the number of executed branches is typically less than .66 times the original number. Such results may allow computer architects designing future processors to provide less branch throughput in hardware while still achieving enhanced efficiency and performance through compiler technology.

The static total operation counts show the code expansion incurred to perform control CPR. With the exception of the small benchmarks, a less than 10% code expansion is consistently observed. As shown with the code example in Section 6, control CPR does increase static code size through the insertion of redundant code in the compensation blocks. However, the overall size is strictly controlled by applying control CPR only to the frequently executed program regions. The remainder of the code is left untreated. The net result is that the overall static code expansion is rather small for most applications.

Its important to note that these results do not represent either system-level speedups or limits to the effectiveness of this technology. Dynamic effects such as virtual memory or cache performance may dilute these speedups. Heuristics for control CPR are not very mature and have not yet been tuned for each specific processor configuration. In addition, these experiments

have not addressed the treatment of unbiased branches. Unbiased branches typically result in short CPR blocks because either an input superblock was truncated during superblock formation or, the exit-ratio test terminates CPR block growth in the match algorithm. While these experiments apply FRP conversion to linear superblocks, no traditional if-conversion has been applied. The compiler could employ traditional if-conversion to eliminate many unbiased branches and thus further improve the effectiveness of control CPR. These results do, however, indicate that control CPR techniques have the potential to produce important speedups on major applications and deserve further study.

# 8 Conclusion

In this paper, we have described the *Irredundant Consecutive Branch Method* (ICBM) schema for control critical path reduction that alleviates performance bottlenecks due to branch latency and throughput. This paper has advanced control CPR from a concept to a systematic compiler technology which automatically processes a general class programs.

Control CPR uses a number of basic tools to decrease branch dependence height. The use of fully-resolved predicates (FRPs) parallelizes branches by allowing free branch re-ordering during scheduling and by providing well-defined overlapped branch execution on processors with exposed branch latency. The use of FRPs transforms chains of branch dependences into chains of data dependences necessary to compute FRPs. These data dependence chains can be height-reduced by using the associative property and 2-input logical operations or by using PlayDoh's wired-and and wired-or compare operations.

By accelerating likely-taken paths at the expense of rarely taken paths, our implementation of control CPR (ICBM) decreases dependence height without requiring excessive operation count. In fact, ICBM systematically decreases the total number of executed operations and greatly decreases the number of executed branches for scalar programs running on our target processor. We have implemented ICBM within a working ILP compiler and we have presented experimental results to evaluate the effectiveness of ICBM on scalar programs. Our results show that ICBM provides substantial speedups for a broad range of applications and for a range of EPIC processors with varying degrees of hardware parallelism.

# Acknowledgments

The authors thank Anton Ertl for the extensive time and effort he put into improving the quality of this paper; Santosh Abraham, David August, Vinod Kathail, and Matthai Philipose for all their effort developing the Elcor hyperblock scheduler and control-flow transformation infrastructure; Rob Schreiber and the anonymous referees for their valuable comments and suggestions.

# References

[AKPW83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of POPL-10*, pp. 177–189, Jan. 1983.

[BGS95] R. Bodik, R. Gupta, and M Soffa. Interprocedural conditional branch elimination. In *Proceedings of PLDI-95*, pp. 146–158, June 1995.

[DT93] J. Dehnert and R. Towle. Compiling for the Cydra-5. *The Journal of Supercomputing*, 7(1):181–228, Jan. 1993.

[FF92] J. Fisher and S. Freudenberger. Predicting conditional jump directions from previous runs of a program. In *Proceedings of ASPLOS-V*, pp. 85–95, Oct. 1992.

[GK92] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. In *Proceedings of PLDI-92*, pp. 341–352, June 1992.

[H+93] W. W. Hwu et al. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, Jan. 1993.

[HBC98] W. A. Havanki, S. Banerjia, and T. M. Conte. Treegion scheduling for wide-issue processors. In *Proceedings of HPCA-4*, Feb. 1998.

[JS96] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of Micro-29*, pp. 100–113, Dec. 1996.

[KSR93] V. Kathail, M. Schlansker, and B. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, H.P. Laboratories, Feb. 1993.

[Kuc78] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, New York, NY, 1978.

[LFK+93] P. Lowney, S. Freudenberger, T. Karzas, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1):51–142, Jan. 1993.

[ME92] S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. In *Proceedings of Micro-25*, pp. 55–71, Dec. 1992.

[MLC+92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of Micro-25*, pp. 45–54, Dec. 1992.

[MW92] F. Mueller and D Whalley. Avoiding unconditional jumps by code replication. In *Proceedings of PLDI-92*, pp. 322–330, June 1992.

[MW95] F. Mueller and D. Whalley. Avoiding conditional branches by code replication. In *ACM SIGPLAN Notices*, pp. 272–282, 1995.

[SK93] M. Schlansker and V. Kathail. Acceleration of algebraic recurrences on processors with instruction level parallelism. In *Proceedings of LCPC-6*, pp. 406–429, 1993.

[SK95] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *Proceedings of Micro-28*, pp. 57–69, Dec. 1995.

[TLS90] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of loops with exits on pipeline processors. In *Proceedings Supercomputing '90*, pp. 200–212, Nov. 1990.

[YUW98] M. Yang, G. Uh, and D. Whalley. Interprocedural conditional branch elimination. In *Proceedings of PLDI-98*, pp. 130–141, June 1998.