In proceedings of the International Conference on Parallel Architectures and Compilation Techniques, October 1999.

Predicated Static Single Assignment

Lori Carter Beth Simon Brad Calder Larry Carter Jeanne Ferrante

Department of Computer Science and Engineering
University of California, San Diego
{lcarter,esimon,calder,carter,ferrante}@cs.ucsd.edu

Abstract

Increases in instruction level parallelism are needed to exploit the potential parallelism available in future wide issue architectures. Predicated execution is an architectural mechanism that increases instruction level parallelism by removing branches and allowing simultaneous execution of multiple paths of control, only committing instructions from the correct path. In order for the compiler to expose such parallelism, traditional compiler data-flow analysis needs to be extended to predicated code.

In this paper, we present Predicated Static Single Assignment *(PSSA) to enable aggressive predicated optimization and instruction scheduling. PSSA removes false dependences by exploiting renaming and information about the multiple control paths. We demonstrate the usefulness of PSSA for Predicated Speculation and Control Height Reduction. These two predicated code optimizations used during instruction scheduling reduce the dependence length of the critical paths through a predicated region. Our results show that using PSSA to enable speculation and control height reduction reduces execution time from 10% to 58%.*

1 Introduction

The Explicitly Parallel Instruction Computing (EPIC) architecture has been put forth as a viable architecture for achieving the *instruction level parallelism* (ILP) needed to keep increasing future processor performance [7, 15]. The Merced [1] processor being developed at Intel is an example of an EPIC architecture. An EPIC architecture issues wide instructions, similar to a VLIW architecture, where each instruction contains many operations.

One of the new features of the EPIC architecture is its support for *predicated execution* [21], where each operation is guarded by one of the predicate registers available in the architecture. An operation is committed only if the value of its guarding predicate is true.

One advantage of predicated execution is that it can eliminate hard-to-predict branches by combining both paths of a branch into a single path. Another advantage comes from using predication to combine several smaller basic blocks into one larger hyperblock [19]. This provides a larger pool from which to draw ILP for EPIC architectures.

A significant limitation to ILP is the presence of control-flow and data-flow dependences. Static Single Assignment (SSA) is an important compiler transformation used to remove false data dependences across basic block boundaries in a control flow graph [11]. Removing these false dependences reveals more ILP, allowing better performance of optimizations like instruction scheduling. Without performing SSA, the benefit of many optimizations on traditional code is limited.

Eliminating false dependences is equally important and a more complex task for predicated code, since multiple control paths are merged into a single predicated region. However, the control-flow and data-flow analysis needed to support predicated compilation is different than traditional analysis used in compilers for superscalar architectures. A sequential region of predicated code contains not only data dependences, but also *predicate dependences*. A predicate dependence exists between every operation and the definition of its guarding predicate. A chain of predicate dependences represents a unique control path through the original code.

In this paper we describe a predicate-sensitive implementation of SSA called *Predicated Static Single Assignment* (PSSA). We extend SSA to handle predicate definitions and the multiple control paths that are merged together in a single predicated region. We demonstrate that PSSA allows effective predicated scheduling by (1) eliminating false dependences along paths via renaming, (2) creating full-path predicates, and (3) providing path-sensitive data-flow analysis. We show the benefit of using PSSA to perform Predicated Speculation and Control Height Reduction during instruction scheduling. Using PSSA allows these two optimizations, when applied together, to schedule all operations at their *earliest schedulable cycle*. In our implementation, the earliest schedulable cycle takes into consideration true data dependences and load/store constraints. We conservatively assume that a load is dependent on all prior stores along a given path, and that a store is dependent on prior stores as well. In addition, we ensure that all instructions along a path leading to a branch out of the hyperblock are executed prior to exiting the hyperblock.

The paper is organized as follows. Section 2 describes predicated execution. Section 3 presents Predicated Static Single Assignment. Section 4 shows how PSSA can enable aggressive Predicated Speculation and Control Height Reduction. Section 5 reports the increased ILP and reduced execution times achieved by applying our algorithms to predicated code. Section 6 summarizes related work. Section 7 discusses using PSSA within the IA-64 framework, and Section 8 describes our future work. Finally, Section 9 summarizes the contributions of this paper.

2 Predicated Execution

Predicated execution is a feature designed to increase ILP and remove hard-to-predict branches. Machines with hardware to support predicated code include an additional set of registers called predicate registers. The process of predication replaces branches with compare operations that set predicate registers to either true or false based on the comparison in the original branch. Each operation is then associated with one of these predicate registers (the operation's *guarding predicate*). The operation will be committed only if its guarding predicate is true, except for predicates defined unconditionally. This process of replacing branches with compare operations and associating operations with a predicate defined by that compare is called *If-Conversion* [5, 21].

Our work uses the notion of a hyperblock [19]. A *hyperblock* is a predicated region of code consisting of a group of basic blocks with one entry point and possibly multiple branch points. Branches with both targets in the hyperblock are eliminated and converted to predicate definitions using if-conversion. All remaining branches have targets outside the hyperblock. Consequently, there are no cyclic control-flow or data-flow dependences within the hyperblock. The selection of basic blocks to be included in the hyperblock is based on program profiling which includes information such as execution frequency, basic block size, operation latencies, and other characteristics.

A typical code section to include in a hyperblock is one that contains a hard-to-predict (unbiased) branch [18], as shown in Figure 1. After predication, the Control Flow Graph (CFG) in Figure 1(b), which is comprised of five basic blocks, results in the predicated hyperblock shown in Figure 1(c). All operations in the hyperblock are now guarded, either by a predicate register set to the constant value of true, or by a register that can be defined as either true or false by a `cmpp` (compare and put (result) in predicate) operation. Operations guarded by the constant true, such as the operation `d=c*2` in Figure 1, will be executed and committed regardless of the path taken. Operations guarded by a predicate register, such as the operation `b=7`, will be put into the pipeline, but only committed if the value of the operation's guarding predicate (`B` for this operation) is determined to be true. In a hyperblock, a control flow path is now represented by a chain of predicate dependences.

In what follows, we describe three types of operations that can be included in a hyperblock – `cmpp` operations, the predicate OR operation, and normal (non-predicate-defining) operations.

As defined in the Trimaran System [2] (which supports EPIC computing via the Playdoh ISA [16]), guarding predicates are assigned their values via `cmpp` operations [7]. Consider the operation `B,C cmpp.un.ac a>c if A` as an example. The `cmpp` operation can define one or two predicates. This operation will define predicates `B` and `C`. The first tag (`.un`) applies to the definition of the first predicate `B` and the second tag (`.ac`) to `C`. The first character of the tag defines how the predicate is to be defined. The character `u` means that the predicate will unconditionally get a value, whether the guarding predicate (`A` in this case) is true or false. If `A` is false, then `B` is set to false. Otherwise, `A` is true and the value of `B` depends upon the evaluation of `a>c`.

The character `a` in the second tag (`.ac`) indicates that the full definition of the related predicate `C` is contingent on the value of `A`, the evaluation of `a>c`, and the prior value of `C`. If `A` is false, the value of predicate `C` does not change. If `A` is true and either `C` or `a>c` evaluate to false, the new value of `C` will be false. The second character of the tag defines whether the normal (`n`) result of the condition (`a>c`) or the complement (`c`) of the condition must be true to make the related predicate true. For a complete definition of `cmpp` statements see the Playdoh architecture specification [16].

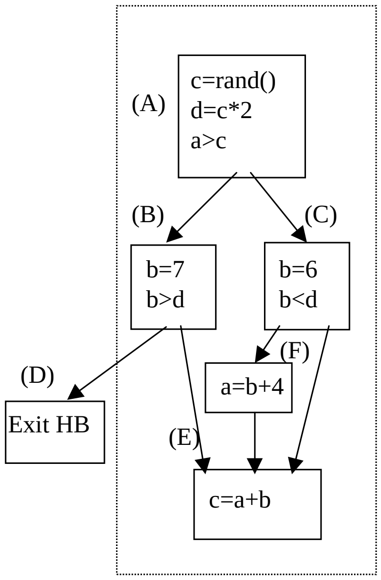
In our implementation of PSSA, we use a new OR operation currently not defined by Trimaran. The *predicate OR* operation defines block predicates by taking the logical OR of multiple predicates. For example, consider the operation `G = OR(A, B, C) if true` where `A, B` and `C` are predicates, each defining a unique path to `G`. If any one of them has the value of true, `G` will receive a value of true, otherwise `G` will be assigned false.

When scheduling, we assume that the definition of a predicate is available for use as a source for another operation or as a guard to a subsequent `cmpp` operation in the cycle following its definition. When used as a guard for all other operations, the predicate definition is available for use in the same cycle as it is defined.

We refer to all other operations, which do not define predicates, as *normal* operations. Normal operations include assignments, arithmetic operations, branches, and memory operations.

```
c=rand()
d=c*2
if a>c
    b=7
    if b>d
        exit HB
else
    b=6
    if b<d
        a=b+4
 c=a+b
```

a) Original code



b) Basic blocks included in hyperblock

```
     c=rand()            if true //c=random num
     d=c*2               if true //d=c*2
B,C cmpp.un.uc a>c       if true //if a>c B=true,
                                 //C=false, else
                                 //B=true,C=false
     b=7                 if B    //if B=true,b=7
                                 //else nullify stmt
D    cmpp.un b>d         if B    //if B=true & b>d
                                 //D=true,else false
                                 //if B=false,D=false
     branch out          if D    //if D=true, exit
                                 //hyperblock
     b=6                 if C    //if C=true, b=6
                                 //else nullify stmt
F    cmpp.un b<d         if C    //if C=true & b<d
                                 //F=true,else false
                                 //if C=false,F=false
      a=b+4              if F    //if F=true, a=b+4
                                 //else nullify stmt
      c=a+b              if true //c=a+b
```

c) Predicated hyperblock

Figure 1: Short code example showing the transformation from non-predicated code to predicated hyperblock.

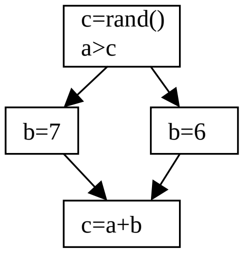
3 Predicated Static Single Assignment (PSSA)

Static Single Assignment (SSA) [11, 12] provides an efficient representation of data dependences. Code in SSA form has only true data dependences remaining, since all false data dependences have been removed [4, 28]. Removing false dependences allows more flexibility in scheduling since data independent operations can move past each other during instruction scheduling.

In non-predicated code, SSA assigns each target of an assignment operation a unique variable. At join nodes (points in the CFG where paths come together), a ϕ function is inserted to determine which of the multiple versions of a variable reaches the join. In addition, a newly renamed version of the variable is assigned using the ϕ function. This new variable is used to represent the merging of the different variable names. Figure 2 shows an example control flow graph and code in SSA form. In the assignment `b3->`ϕ`(b1,b2)`, the variable `b3` represents the reaching definition of `b` which is to be used after the join (`b1` or `b2`).

Eliminating false dependences is equally important and a more complex task for predicated code, since multiple control paths are merged. To address this problem we developed a predicate-sensitive implementation of SSA called *Predicated Static Single Assignment* (PSSA).

PSSA seeks to accomplish the same objectives as SSA for a predicated hyperblock. First, it must assign each target of an assignment operation in the hyperblock a unique variable. Second, at points in the hyperblock where multiple paths come together it must summarize under what conditions each of the multiple versions of a variable reaches that join. The second function is accomplished through the creation of full-path predicates and path-sensitive analysis.



a) Control Flow Graph

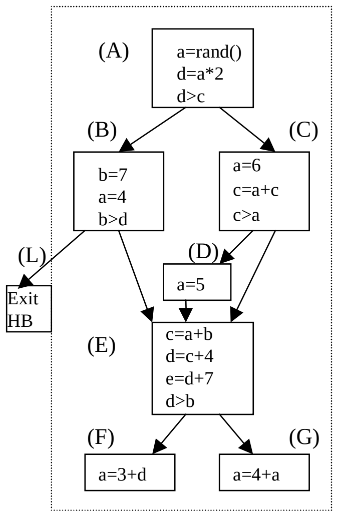
```
c1=rand()
if a1>c1
    b1=7
else
    b2=6
b3->Ø(b1,b2)
c2=a1+b3
```

b) Code in SSA form

Figure 2: Static Single Assignment

Consider the sample predicated code shown in Figure 3 using traditional hyperblock predication [19]. In this predicated example, all branches have been replaced (except the one leaving the hyperblock) with predicate-defining compare operations using if-conversion. The predicates that are defined in this example correspond to the two edges exiting each conditional branch in the CFG in Figure 3. Figure 4 shows this example after PSSA has been applied and displays a graph showing the post-PSSA dependence relationships.

The PSSA transformation has 2 phases. Hyperblocks are converted to PSSA form before optimization. After optimization, PSSA inserts clean-up code, copying renamed



a) Control Flow Graph (CFG)

```
     a=rand()          if true
     d=a*2             if true
B,C  cmpp.un.uc d>c    if true
     b=7               if B
     a=4               if B
L    cmpp.un  b>d      if B
     branch out        if L
     a=6               if C
     c=a+c             if C
D    cmpp.un c>a       if C
     a=5               if D
     c=a+b             if true
     d=c+4             if true
     e=d+7             if true
F,G  cmpp.un.uc d>b    if true
     a=3+d             if F
     a=4+a             if G
```

b) Predicated Hyperblock

Figure 3: Extended example of transformation from non-predicated CFG to predicated hyperblock

variables back to their original names and removes any unused predicate definitions.

3.1 Converting to PSSA Form

PSSA conversion takes two forms. *Control PSSA* is applied to predicate-defining operations, and *Normal PSSA* is applied to all other operations. When converting to PSSA form, each operation is processed in turn beginning at the top of the hyperblock and proceeding to the end.

When a normal operation is encountered, Normal PSSA is invoked. If the operation is an assignment, the variable defined is renamed. The third operation `d1=a0*2` in Figure 4(b) is an example. All operands are adjusted to reflect previously renamed variables (e.g. `a` becomes `a0`). If the operation is part of a join block, multiple versions of the operands may be live. The first operation (`c=a+b`) in block `E` of Figure 3(a) provides an example. In this situation, the operation will be duplicated for each path leading to the join and the correct operand versions for each path will be used in the duplicate statement as seen in Figure 4 (in the multiple definitions of `c2`). The duplicates are guarded by the full-path predicate (described in the next paragraph) associated with the path along which the operands are defined. Though there are 3 definitions of `c2`, there is only one definition of `c2` on any given path. These definitions are predicated on disjoint predicates; only one of them can possibly be true, and only one of them will be committed.

When a `cmpp` operation is processed, Control PSSA is invoked. The single `cmpp` operation that defined one or two block predicates (such as the definitions of `F` and `G` in Figure 3) is replaced by one or more `cmpp` operations, each associated with a particular path leading to that block. As can be seen in Figure 4(b) there are now three `cmpp` operations defining `FEBA` and `GEBA`, `FECA` and `GECA`, and `FEDCA` and `GEDCA`. These new predicates are called *full-path predicates* (FPPs). Each FPP definition has the appropriate operand versions for its path and each is guarded by the FPP that defined the path prior to reaching the new block. For example, the `cmpp` defining `GEBA` and `FEBA` is predicated on `EBA`.

An FPP specifies the unique path along which an operation is valid for execution, enabling PSSA to provide correct guarding predicates for the duplicate statements previously described. For example, the use of `a` in operation `a=4+a` in block G of Figure 3(a) could originate from 3 different definitions as renamed in Figure 4 (`a1, a2, a3`). It might appear that we could predicate the duplicate assignment statements as follows:

(1) `a5=4+a1 if B`
(2) `a5=4+a2 if C`
(3) `a5=4+a3 if D`

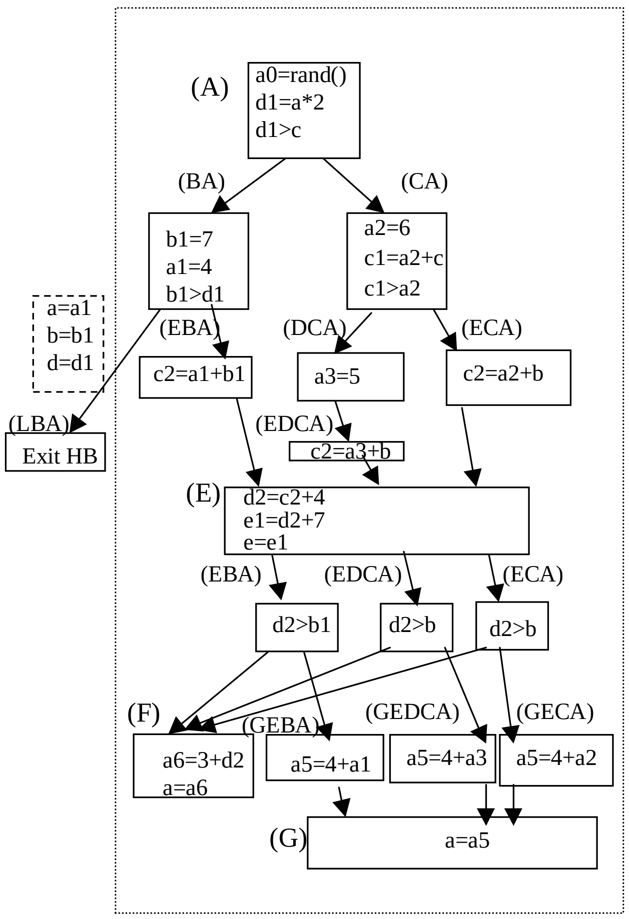
However, all three of these statements could cause erroneous execution. The first statement could cause a scheduling problem. A scheduler might assume that once the predicate B was defined and the operand `a1` was defined, all dependences had been met and `a5=4+a1 if B` could be scheduled. If the branch out of the hyperblock was taken, this would result in an incorrect modification of `a5`. Predicating the operation on `GEBA` instead of `B`, avoids this error. The operation would be executed only if block predicates `G,E,B` and `A` are true. In the case of statements (2) and (3) predicated on if `C` and if `D`, note that both `C` and `D` can be true. Without the more specific path information given by using the FPPs `GECA` and `GEDCA`, both assignments to `a5` could be made. Full path predicates are used to avoid these problems.

In addition to the `cmpp` statements added to define FPPs, `cmpp` statements are included to rename join blocks whose statements were originally predicated on true. `A` and `E` and their associated FPPs are examples. The operations in Figure 3(b) predicated on true, are predicated on `A` and `E` in the PSSA version of the code shown in Figure 4. This is necessary to maintain exact path information.

We could have used the FPPs to implement ϕ functions as in SSA. For example, instead of the 3 definitions of `a5` found in Figure 4, we could have used:

```
a7=a1 if GEBA;
a7=a2 if GECA;
a7=a3 if GEDCA;
a5=a7+4 if G;
```

However, this would add additional dependences *lengthening the schedule of the hyperblock.* Since the goal



a) PSSA dependence graph

```
                                                      cycle available
                                                      for scheduling

A             cmpp.un TRUE                  if true      1
              a0=rand()                     if A         1
              d1=a0*2                       if A         2
BA,CA         cmpp.un.uc d1>c               if A         3
B             OR (BA)                       if true      4
C             OR (CA)                       if true      4
              b1=7                          if BA        3
              a1=4                          if BA        3
LBA,EBA       cmpp.un.uc b1>d1              if BA        4
              branch out                    if LBA       4
              a2=6                          if CA        3
              c1=a2+c                       if CA        4
EDCA,ECA      cmpp.un.uc c1>a2              if CA        5
D             OR (EDCA)                     if true      6
              a3=5                          if EDCA      5
E             OR (EBA,ECA,EDCA)             if true      6
              c2=a1+b1                      if EBA       4
              c2=a2+b                       if ECA       5
              c2=a3+b                       if EDCA      6
              d2=c2+4                       if E         7
              e1=d2+7                       if E         8
FEBA,GEBA     cmpp.un.uc d2>b1              if EBA       8
FECA,GECA     cmpp.un.uc d2>b               if ECA       8
FEDCA,GEDCA   cmpp.un.uc d2>b               if EDCA      8
F             OR (FEBA,FECA,FEDCA)          if true      9
G             OR (GEBA,GECA,GEDCA)          if true      9
              a6=3+d2                       if F         9
              a5=4+a1                       if GEBA      8
              a5=4+a2                       if GECA      8
              a5=4+a3                       if GEDCA     8
              a=a5                          if G         9
              a=a6                          if F        10
              e=e1                          if E         9
```

b) PSSA transformed code

Figure 4: The PSSA dependence graph shows the flow of data and control through the PSSA-transformed code. Blocks labeled with full-path *predicates (indicated by multiple letters) contain statements that are only executed along that path. Blocks labeled with* block *predicates (single letters) contain statements that will be executed along several paths.*

of this study was to show an implementation of PSSA that could schedule each operation at its earliest cycle, we do not use any ϕ functions. For future work, we are examining using ϕ functions for non-critical paths through the hyperblock.

Block predicates are also important to the PSSA transformation. PSSA uses predicate OR statements to redefine the block predicates as the union of the FPPs associated with the paths that reach the block. PSSA does not simply duplicate every path through the hyperblock. Duplication only occurs when necessary to remove false dependences. When there is only one version of all operands reaching a statement, only one version of the statement is required. This is the case with `a6=3+d2` in Figure 4. The variable `d2` is the only version live going into node F. This statement is guarded by F, a block predicate created by taking the logical OR of FEBA, FECA and FEDCA. As long as control reaches node F, regardless of the path taken, we will execute and commit the statement `a6=3+d2`.

3.2 Post-Optimization Clean-up

After optimization is applied to code in PSSA form, a clean-up phase is run to remove unnecessary code and to assure consistent code outside of the hyperblock.

The earliest cycle PSSA implementation described in this paper generates `cmpp` statements for every path and block. These are entered into the PSSA data structure that maintains information about the relationships between the predicates they define, which provides maximum flexibility during optimization. However, some of these FPP definitions may not be used, and the corresponding `cmpp` operations will be discarded, reducing the code size significantly.

Finally, to assure correct execution following the hyperblock, PSSA inserts copy operations assigning the original variable names to all renamed definitions that are live out of the hyperblock. In Figure 4, definitions `a` and `e` are assumed to be live out of the hyperblock and so the copy operations have been inserted.

4 Hyperblock Scheduling Optimizations

In this section, we describe how PSSA enables Predicated Speculation (PSpec) and Control Height Reduction (CHR) for aggressive instruction scheduling. PSpec allows operations to be executed before their guarding predicates are determined and CHR allows the guarding predicates to be determined as soon as possible, reducing the number of operations that need to be speculated. Used together with PSSA, we demonstrate that we can schedule the code at its earliest schedulable cycle, assuming a machine with unlimited resources.

4.1 Predicated Speculation

This section describes how to perform speculation on PSSA-transformed code. In general, speculation is used to relieve constraints which control dependences place on scheduling. One can speculatively execute operations from the likely-taken path of a highly-predictable branch, by scheduling those operations before their controlling branch [17]. Similarly, Predicated Speculation (PSpec) will schedule a normal operation above the `cmpp` it is dependent upon, optimizing a hyperblock's execution time.

PSpec handles placement of the speculated predicated operation in a uniform manner. PSpec schedules a normal operation at its earliest schedulable cycle. When speculating an operation, the operation is scheduled earlier than the operation it is control dependent on, and is predicated on true. We assume that any exceptions raised by the speculated operations will be taken care of using architecture features such as poison bits [9].

4.1.1 Instruction Scheduling with Speculation

To demonstrate the usefulness of PSSA in enabling PSpec, Figure 5 shows the code from Figure 4 after the PSpec optimization has been applied. The assignments to `a1`, `a2` and `a3` are examples of speculated operations. Notice that based on dependences, they could all be scheduled at cycle one which would have been impossible without renaming.

During predicated speculation, each operation is considered sequentially, beginning with the first instruction in the hyperblock. If it is a normal, non-store operation, PSpec compares its earliest schedulable cycle with the cycle in which its guarding predicate is currently defined. If the operation can be scheduled earlier than its guarding predicate, the operation is predicated on true and scheduled at its earliest schedulable cycle.

Recall that PSSA has not performed full renaming, so further renaming may be required by PSpec. An example is the definition of `c2` in Figure 4. If we speculate any of the definitions of `c2` by predicating them on true without renaming, incorrect code can result. Consequently, we must rename the operations being speculated. The results

```
                                                        cycle available
                                                         for scheduling
A              cmpp.un TRUE                if true      1
               a0=rand()                   if A         1
               d1=a0*2                     if A         2
BA,CA          cmpp.un.uc d1>c             if A         3
B              OR (BA)                     if true      4
C              OR (CA)                     if true      4
               b1=7                        if true      1
               a1=4                        if true      1
LBA,EBA        cmpp.un.uc b1>d1            if BA        4
               branch out                  if LBA       4
               a2=6                        if true      1
               c1=a2+c                     if true      2
EDCA,ECA       cmpp.un.uc c1>a2            if CA        4
D              OR (EDCA)                   if true      5
               a3=5                        if true      1
E              OR (EBA,ECA,EDCA)           if true      5
               c2=a1+b1                    if true      2
               c3=a2+b                     if true      2
               c4=a3+b                     if true      2
               d2=c2+4                     if true      3
               d3=c3+4                     if true      3
               d4=c4+4                     if true      3
               e1=d2+7                     if EBA       4
               e1=d3+7                     if ECA       4
               e1=d4+7                     if EDCA      4
FEBA,GEBA      cmpp.un.uc d2>b1            if EBA       5
FECA,GECA      cmpp.un.uc d2>b             if ECA       5
FEDCA,GEDCA    cmpp.un.uc d2>b             if EDCA      5
F              OR (FEBA,FECA,FEDCA)        if true      6
G              OR (GEBA,GECA,GEDCA)        if true      6
               a9=3+d2                     if true      4
               a7=3+d3                     if true      4
               a8=3+d4                     if true      4
               a4=4+a1                     if true      2
               a5=4+a2                     if true      2
               a6=4+a3                     if true      2
               a=a9                        if FEBA      5
               a=a7                        if FECA      5
               a=a8                        if FEDCA     5
               a=a4                        if GEBA      5
               a=a5                        if GECA      5
               a=a6                        if GEDCA     5
```

Figure 5: Extended code example after PSpec optimization has been applied.

of applying this to the 3 definitions of `c2` (now `c2`, `c3`, and `c4`) appear in Figure 5. Speculation and renaming may require the duplication of operations using the definition being speculated, since there may now be multiple reaching definitions. When speculating `c2`, the operation `d2=c2+4` had to be duplicated and guarded on the appropriate FPP as shown in Figure 5. This is made possible, since PSSA already created all the necessary FPPs and path information.

If the guarding predicate has been defined by the operation's earliest schedulable cycle, we do not apply PSpec. It

```
PSpec(normal_op)
{
  if (normal_op.guarding_predicate not defined by
      normal_op.earliest_schedulable_cycle)
  {
      if (multiple defs of normal_op.target exist
      {
          rename(normal_op.target);
      }
      normal_op.schedule(earliest_schedulable_cycle);
      normal_op.set_predicate(true);
  }
  else
  {
      normal_op.schedule(earliest_schedulable_cycle);
  }
}
```

Figure 6: Basic PSpec Algorithm.

is again scheduled at the cycle equal to its earliest schedulable cycle, but guarded by the guarding predicate assigned by PSSA. The algorithm for PSpec instruction scheduling is shown in Figure 6.

Using PSpec, the hyperblock can now be scheduled in 6 cycles as compared to 10 cycles in Figure 4. Since PSpec is applied whenever the definition of the operation's guarding predicate occurs later than the earliest schedulable cycle of the operation, we could reduce the number of operations that need to be speculated by moving the definition of the guarding predicates earlier. The goal of the next optimization, Control Height Reduction, is to allow predicates to be defined as early as possible.

4.2 Control Height Reduction

Control Height Reduction (CHR) eases control constraints between multiple control statements. CHR allows successive control operations on the control path to be scheduled in the same cycle, effectively reducing control dependence height. For example, in the code in Figure 5, the control comparisons for `d1>c` and `b1>d1` are scheduled in cycles 3 and 4, respectively. However, the second comparison is only waiting for the definition of its guarding predicate `BA`.

To schedule it earlier, consider the PSSA dependence graph in Figure 4. The definition of EBA (defined by the condition `b1>d1`), is control dependent on the definition of BA (defined by the condition `d1>c`). We could also define EBA directly as the logical AND of the conditions `b1>d1` and `d1>c` removing the dependence on the definition of `BA`. This AND expression could also be scheduled in cycle 3.

Control Height Reduction was proposed in [24]. It was successfully used to reduce the height of control recurrences found in loops when applied to superblocks. A *superblock* is a selected trace of basic blocks through the control flow graph containing only one path of control [23]. The path defining aspects of PSSA allow our algorithm to efficiently apply CHR to predicated hyperblocks, since the full-path predicates expose all of the original separate paths throughout the hyperblock.

Schlansker et. al. [25] expanded on their previous research, applying speculation prior to attempting height reduction. Speculation can remove dependences between the branch conditions that need to be combined to accomplish

			cycle available for scheduling
A	cmpp.un TRUE	if true	1
	a0=rand()	if A	1
	d1=a0*2	if A	2
BA,CA	*cmpp.un.uc d1>c*	*if A*	*3*
B	*OR (BA)*	*if true*	*4*
C	*OR (CA)*	*if true*	*4*
	b1=7	if true	1
	a1=4	if true	1
LBA,EBA	cmpp.an.an d1>c	if A	3
LBA,EBA	cmpp.an.ac b1>d1	if A	3
	branch out	if LBA	3
	a2=6	if true	1
	c1=a2+c	if true	2
EDCA,ECA	cmpp.ac.ac d1>c	if A	3
EDCA,ECA	cmpp.an.ac c1>a2	if A	3
D	*OR (EDCA)*	*if true*	*5*
	a3=5	if true	1
E	OR (EBA,ECA,EDCA)	if true	4
	c2=a1+b1	if true	2
	c3=a2+b	if true	2
	c4=a3+b	if true	2
	d2=c2+4	if EBA	3
	d2=c3+4	if ECA	3
	d2=c4+4	if EDCA	3
	e1=d2+7	if E	4
FEBA,GEBA	cmpp.un.uc d2>b1	if EBA	4
FECA,GECA	cmpp.un.uc d2>b	if ECA	4
FEDCA,GEDCA	cmpp.un.uc d2>b	if EDCA	4
F	OR (FEBA,FECA,FEDCA)	if true	5
G	*OR (GEBA,GECA,GEDCA)*	*if true*	*5*
	a7=3+d2	if true	4
	a4=4+a1	if true	2
	a5=4+a2	if true	2
	a6=4+a3	if true	2
	a=a7	if F	5
	a=a4	if GEBA	5
	a=a5	if GECA	5
	a=a6	if GEDCA	5
	e=e1	if E	5

Figure 7: Extended example after PSpec and CHR optimizations have been applied. `Cmpp` *instructions displayed in gray define predicates that are not used after optimization. Therefore, the statements can be removed from the final code.*

the reduction. However, in that work, speculation was limited to operations that would not overwrite a live register or memory value if speculated, since they did not use renaming. In Figure 4, the `cmpp` operation defining `EDCA` and `ECA` is shown scheduled at cycle 5 due to dependences on `a2` and `c1`. PSSA allows us to apply PSpec and schedule these definitions in cycles 1 and 2 respectively, making the `cmpp` available for CHR as shown in Figure 7.

4.2.1 Instruction Scheduling with PSpec and CHR

During instruction scheduling, PSpec is performed as described in Section 4.1.1. For each *control* operation (`cmpp`), CHR is performed if possible.

Recall that the operations in Figure 4 are scheduled in the order given in the PSSA hyperblock. Like PSpec, CHR compares when the operation could be scheduled based on its earliest schedulable cycle with when it must be scheduled if it waited for its guarding predicate to be defined. If it does not need to wait on the definition of its guarding predicate, it is simply scheduled at its earliest schedulable cycle. For example, consider the definitions of `FEBA` and `GEBA` in Figure 7. The definition of `EBA` (the guarding predicate of this `cmpp` operation) is scheduled at cycle 3, but the earliest schedulable cycle of this `cmpp` operation is 4 because of its true data dependency on the definition of `d2` in cycle 3.

If the `cmpp` operation must wait for the definition of the guarding predicate it is beneficial to CHR. By ANDing the conditions of the current definition with that of its guarding predicate, we can schedule this definition earlier. If the definition of the guarding predicate involved conditions that were ANDed as well, all of the conditions must be included, so the number of `cmpp` statements needed to define the current operation increases. The `.a` tag on each of these cmpp statements indicates that all of them are required for the final definition.

Consider the operations `d1>c` and `c1>a2` in Figure 4. We control height reduce these operations in Figure 7, since they are both schedulable in cycle 3 based on our scheduling constraints. The definition of EDCA now describes the combination of `d1>c` being false AND `c1>a2` having a value of true. We implement this logical AND, using the `.ac` and `.an` qualifiers. The definition of EDCA requires that both the complement of the condition `d1>c` and the condition `c1>a2` evaluate to true for the FPP to get a value of true. If one or both of the requirements are not met, the FPP will be set to false. The compares can architecturally be performed in the same cycle [16] allowing multiple links in a control path to be defined simultaneously. The algorithm for CHR is found in Figure 8.

Using PSpec and CHR on PSSA-transformed code results in the 4 cycle schedule shown in Figure 7. One additional cycle is required to resolve renamed variables that

```
CHR(cmpp_op)
{
    if (cmpp_op.guarding_pred defined
        by cmpp_op.earliest_schedulable_cycle)
    {
        cmpp_op.schedule(cmpp_op.earliest_schedulable_cycle)
    {
/* Apply Control Height Reduction */
    else
    {
        while (more_stmts_defining(cmpp_op.guarding_pred))
        {
            next_def=next_defining_stmt(cmpp_op.guarding_pred)
            copy=duplicate(next_def)
            copy.schedule(next_def.get_scheduling_time())
            copy.predicate_on(next_def.get_guarding_pred())
            copy.set_define(cmpp_op.get_pred_defined())
            copy.set_tag_to(a)
        }
        cmpp_op.schedule(next_def.get_scheduling_time())
        cmpp_op.predicate_on(next_def.get_guarding_pred())
        cmpp_op.set_tag_to(a)
    }
}
```

Figure 8: Basic Control Height Reduction Algorithm.

are live out. Note that this last version of the code has fewer operations than the previous version in Figure 5 and the operations shown in gray can be removed in a post-pass because these operations define predicates that are never used. Using predicated speculation and control height reduction together on PSSA-transformed code allows every operation to be scheduled at its earliest schedulable cycle.

5 Results

We have implemented algorithms to perform PSSA, CHR and PSpec on hyperblocks in the Trimaran System (Version 1.00). We collect profile-based execution weights for operations in the codes and schedule operations with an assumed one-cycle latency in order to calculate execution time.

Figure 9 shows normalized execution time when applying our optimizations for several Trimaran benchmarks: `compress` (from SPECINT95), `alvinn` (from SPECFP92), `fib` and `matrix multiply (mm)` (from Trimaran), and `qsort`. The original execution times are created from the default Trimaran settings, with the exception that the architecture issue rate is set to 16. Execution time is estimated by summing together the frequency of execution of each hyperblock multiplied by the number of cycles it takes to execute the hyperblock, and a perfect memory system is assumed. The results are normalized to the original schedule generated by Trimaran for a 16 issue machine. The infinite results show the normalized execution time assuming an infinite issue architecture. The optimized results show the performance after applying PSSA, PSpec,

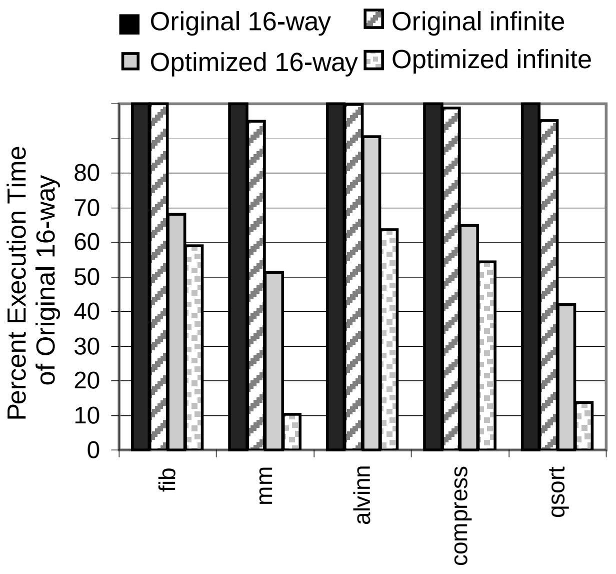


Figure 9: Executed cycles normalized to the number of cycles to execute the original code produced by Trimaran for a 16 issue machine.

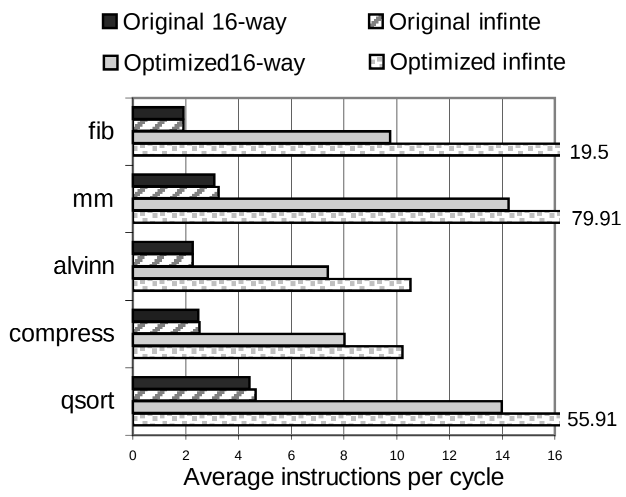


Figure 10: Weighted average number of operations scheduled per cycle for hyperblocks when using PSSA with Predicated Speculation and Control Height Reduction.

and CHR. The results show that using PSSA with PSpec and CHR results in a significant reduction in executed cycles.

Figure 10 shows the average number of operations executed per cycle for the configurations examined in Figure 9. In comparing the two graphs for the 16-way results, 3 to 4 times as many instructions are issued per cycle after applying PSSA, PSpec, and CHR, and this resulted in a reduction in execution time ranging from 10% to 58%.

The renaming required by PSSA and PSpec also significantly increases register pressure. Trimaran's ISA (Playdoh) supports 4 register files: general purpose, floating point, branch, and predicate [2] [16]. Figure 11 shows the average number of live registers for the original code and the optimized code using PSSA, PSpec and CHR. The average live register results are weighted by the frequency of hyperblock execution. For example, `matrix multiply` has on average 18 live general purpose registers in the original code, and 50 live general purpose registers after optimization. Though the increase in utilization of all these register files is notable, the weighted average utilization still remains well within the reported IA-64 register file sizes (128 general purpose, 128 floating point, 8 branch, and 64 predicate) [3].

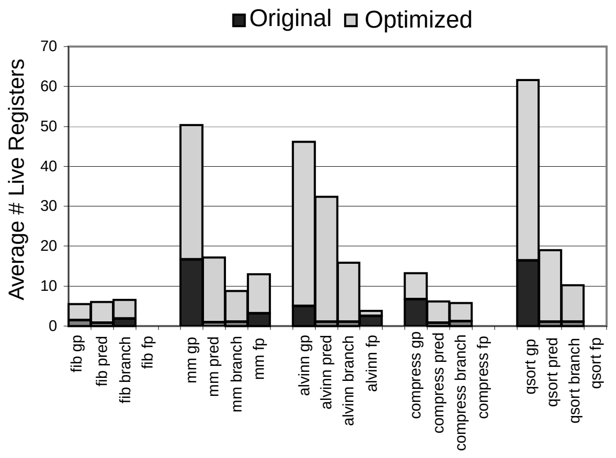


Figure 11: Weighted average register pressure in hyperblocks when using PSSA with Predicated Speculation and Control Height Reduction.

6 Related Work

Predicated execution presents challenges and prospects that researchers have addressed in a variety of ways. Mahlke et. al. [18] showed that predicated execution can be used to remove an average of 27% of the executed branches and 56% of the branch mispredictions. Tyson also found similar results and correlated the relationship between predication and branch prediction [26].

In an effort to relieve some of the difficulties related to applying compiler techniques to predicated code, Mahlke et. al. [19] defined the hyperblock as a single-entry, multiple-exit structure to help support effective predicated compilation. These hyperblocks are formed via selective if-conversion [5, 21] – a technique that replaces branches with predicate define instructions. The success of predicated execution can depend greatly on the region of the code selected to be included in the predicated hyperblock. August et. al. [8] relates the pitfalls and potentials of hyperblock formation heuristics that can be used to guide the inclusion or exclusion of paths in a hyperblock. Warter et. al. [27] explore the use of reverse-if-conversion for exposing scheduling opportunities in architectures lacking support for predicated execution as well as for re-forming hyperblocks to increase efficiency for predicated code [8, 27].

The challenges of doing data-flow and control-flow analysis on hyperblocks have also been addressed. Since hyperblocks include multiple paths of control in one block, traditional compiler techniques are often too conservative or inefficient when applied to them. Methods of predicate-sensitive analysis have been devised to make traditional optimization techniques more effective for predicated code [13, 22]. Our research has extended this predicate-sensitive analysis, as well as incorporated path-sensitive analysis for predicated code which has previously been found useful for traditional data-flow analysis [6, 10, 14]. We use this specialized information to accomplish PSSA (a predicate-sensitive form of SSA [12, 11]) which enables Predicated Speculation and Control Height Reduction for hyperblocks that have previously been examined only in the presence of the single path of control found in superblocks [23, 24, 25].

Moon and Ebcioglu [20] have implemented selective scheduling algorithms, which can schedule operations at their earliest possible cycle for non-predicated code. Our work extends theirs for predicated code, by allowing earliest possible cycle scheduling using predicated renaming with full-path predicates.

7 Implementing PSSA in IA-64

Implementing PSSA using the IA-64 ISA [3] would be straightforward with the exception of the predicate OR statement we introduced. The OR instruction can be implemented by transferring the predicate register file into a general register using the move from predicate instruction in IA-64. The general purpose masking instruction would then be used to mask all but the bits corresponding to the sources of the predicate OR instruction. A result of zero evaluates to false, and anything else evaluates to true.

IA-64 places limits on compare instructions not found in the Playdoh ISA. For example, conditions that are included in logical AND compare statements can only compare a variable to zero. Specifically, the statement `LBA, EBA cmpp.an.ac b1>d1` in Figure 7 would not be permitted. In implementing CHR, we would have to transform the prior expression into the following 2 statements:

```
temp = b1-d1;
LBA, EBA cmpp.an.ac temp>0;
```

8 Future Work

When constructing a hyperblock schedule for a specific processor implementation, resource limits will mandate how many operations can be performed in each cycle. Architectural characteristics such as issue width, resource utilization, number of available predicate registers, and number of available rename registers all need to be considered when creating an architecture-specific schedule. The goal of a hyperblock scheduler is to reduce the execution-height while taking these architectural features into consideration.

In this paper, our goal was to show that PSSA provided an efficient form of renaming and ample path information to allow all operations to be scheduled at their earliest schedulable cycle. We are currently examining different PSSA representations to reduce code duplication and the number of full-path predicates created. Since various control paths through a hyperblock may have different true data dependence heights, it may provide no advantage to speculate operations that are not on the critical path through the hyperblock. PSSA could concentrate on only the critical paths through the hyperblock (reducing code duplication), since these are the optimized paths. For non-critical paths, it may be advantageous in PSSA to implement ϕ functions combining different variable names, instead of maintaining renamed variables for each full-path in the hyperblock. At a point in the hyperblock where all paths join, copy operations could be used to return renamed definitions to original names. Path definitions could then be restarted at this point. This would reduce the amount of duplication required for a given operation to use correctly renamed variables. Our future research concentrates on these issues and creating a more efficient implementation of PSSA.

We have presented only two of the optimizations that benefit from PSSA. Many classical optimizations for traditional code would benefit from a more predicate-sensitive implementation using the information provided by PSSA, and for future work we are applying PSSA to other code optimizations.

9 Conclusions

This paper presented Predicated Static Single Assignment, a predicate-sensitive implementation of SSA, to eliminate false dependences for predicated code. We showed the benefit of using PSSA to perform Predicated Speculation and Control Height Reduction during scheduling. Predicated Speculation allows operations to be executed at the cycle of their earliest schedulable cycle, even before their guarding predicates are determined. CHR allows guarding predicates to be defined as soon as possible, reducing the amount of speculation needed.

By maintaining information about each of the control paths that exist in a hyperblock, PSSA can provide information that allows precise placement of renamed and speculated code, and allows the correct, renamed values to be propagated to subsequent operations. The renaming used by PSSA allows more aggressive speculation, as overwriting live registers and memory values is no longer a concern. In addition, PSSA supports Control Height Reduction along every control path using full-path predicates, reducing control dependence depth throughout the hyperblock.

Our experiments show that PSSA is an effective tool for optimizing predicated code. Using PSSA with PSpec and CHR results in a reduction in executed cycles ranging from 10% to 58% for a 16 issue machine.

Acknowledgements

We would like to thank the Compiler and Architecture Research Group at Hewlett Packard, University of Illinois' IMPACT Group, and New York University's ReaCT-ILP Group for providing Trimaran. We specifically appreciate the time and patience of Rodric Rabbah, Scott Mahlke, Vinod Kathail, and Richard Johnson in answering many questions regarding the Trimaran system. In addition, we would like to thank Scott Mahlke for providing useful comments on this paper. This work was supported in part by NSF CAREER grant No. CCR-9733278, a National Defense Science and Engineering Graduate Fellowship, a gift from Intel Corporation, and equipment support from Hewlett Packard.

References

[1] Intel Press Release. Merced processor and IA-64 architecture., 1998. http: //developer.intel.com/ design/processor/future/iaa64.htm, 1998.

[2] Trimaran, An Infrastructure for Research in Instruction Level Parallelism, 1998. http: //www.trimaran.org.

[3] IA-64 Application Instruction Set Architecture Guide, Revision 1.0, 1999.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques and Tools*. Addison-Wesley, 1986.

[5] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, December 1994.

[6] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. *ACM SIGPLAN Notices*, 33(5):72–84, May 1998.

[7] D. I. August, K. M. Crozier, J. W. Sias, P. R. Eaton, Q. B. Olaniran, D. A. Conners, and W. W. Hwu. The IMPACT EPIC 1.0 Architecture and Instruction Set reference manual. Technical Report IMPACT-98-04, IMPACT, University of Illinois, Feb 1998.

[8] D. I. August, W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *30th Annual Intl. Symp. on Microarchitecture,* December 1997.

[9] D. L. August, D. A. Conners, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Intl. Symp. on Computer Architecture*, July 1998.

[10] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, December 2–4, 1996.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[13] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th Annual Intl. Symp. on Microarchitecture*, pages 114–125, December 1996.

[14] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial dead code elimation using predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113, November 10–14, 1997.

[15] L. Gwennap. Intel, HP make EPIC disclosure. *Microprocessor Report*, 11(14):1–9, October 1997.

[16] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, HP Labs, Feb 1994.

[17] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[18] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual Intl. Symp. on Microarchitecture*, pages 217–227, December 1994.

[19] 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 the 25th Annual Intl. Symp. on Microarchitecture*, pages 45–54, December 1992.

[20] S. Moon and K. Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, November 1997.

[21] J. C. H. Park and M. Schlansker. On Predicated Execution. Technical Report HPL-91-58, HP Labs, May 1991.

[22] M. Schlansker and R. Johnson. Analysis techniques for predicated code. In *Proceedings of the 29th Annual Intl. Symp. on Microarchitecture*, pages 100–113, December 1996.

[23] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 57–69, November 29–December 1, 1995.

[24] M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. In *Proceedings of the 27th Annual Intl. Symp. on Microarchitecture*, pages 40–51, December 1994.

[25] M. Schlansker, S. Mahlke, and R. Johnson. Control CPR: A branch height reduction optimization for EPIC architectures. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[26] G. S. Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 196–206, November 30–December 2, 1994.

[27] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 290–299, June 1993.

[28] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.