COMPILER SUPPORT FOR PREDICATED EXECUTION IN SUPERSCALAR PROCESSORS

BY

DAVID CHU LIN

B.S., University of Illinois, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support. I have learned a lot from him and I greatly enjoyed taking his classes during my undergraduate and graduate school days.

I also like to thank Scott Mahlke who helped me whenever I had a question or problem. Without his help this thesis would never have been finished.

Lastly, I like to thank my parents for encouraging me to do my best.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1
   1.1 Related Works 2
   1.2 Organization of the Thesis 5
2. OVERVIEW OF THE COMPILER 6
   2.1 Lcode Modifications 8
   2.2 Software Modifications 11
3. HARDWARE SUPPORT 12
4. HYPERBLOCK FORMATION AND OPTIMIZATION 16
   4.1 Block Selection 18
   4.2 Code Transformation 20
      4.2.1 Tail duplication 21
      4.2.2 Loop peeling 21
      4.2.3 Node splitting 24
   4.3 If-Conversion 26
      4.3.1 Dominator and post dominator 27
      4.3.2 Algorithm 1 : Association of flow arc 29
      4.3.3 Algorithm 2 : Association of basic blocks 32
      4.3.4 Algorithm 3 : RK algorithm 32
      4.3.5 Comparison of the three algorithms 35
   4.4 Predication-specific Optimizations 36
      4.4.1 Instruction promotion 36
      4.4.2 Renaming and promotion 39
      4.4.3 Instruction merging 40
   4.5 Predicate Hierarchy Graph 41
   4.6 Conventional Optimizations 44
   4.7 Scheduling 45

5. PERFORMANCE EVALUATION 46
5.1 Methodology 46
5.2 Results 48

6. CONCLUSIONS 55

REFERENCES 57

LIST OF TABLES

Table Page

5.1: Benchmarks. 47
5.2: Input data for profiling. 47
5.3: Instruction latencies. 48

LIST OF FIGURES

Figure Page

2.1: A block diagram of the compiler to support predicated execution. . . 7
3.1: A superscalar processor pipelined model with a predicate register file. 13
3.2: Example of predicate register forwarding. (a) Without forwarding. (b) With forwarding. 15
4.1: An example of hyperblock formation. (a) Original program after block selection. (b) After tail duplication. (c) After if-conversion. . . 17
4.2: A program segment before and after hyperblock formation. (a) Original control flow graph. (b) Original program segment. (c) After hyperblock formation. 18
4.3: Block selection algorithm. 20
4.4: An algorithm for tail duplication. 22
4.5: An example of loop peeling. 23
4.6: An algorithm for loop peeling. 24
4.7: An example of node splitting. 25
4.8: An algorithm for node splitting. 26
4.9: Algorithm for finding dominators. 28
4.10: A simple control flow graph. 28
4.11: Algorithm for finding post dominators. 30
4.12: Algorithm 1 for if-conversion. 31
4.13: Algorithm 2 for if-conversion. 33
4.14: RK algorithm for if-conversion. 35
4.15: Normalized execution time of the three if-conversion schemes. 36
4.16: Type 1 instruction promotion algorithm. 37
4.17: An example of code segment after each optimization. (a) Original code. (b) After instruction promotion. (c) After renaming/promotion. (d) After merging. 38
4.18: Type 2 instruction promotion algorithm. 39
4.19: Renaming and instruction promotion algorithm. 40
4.20: Type 1 instruction merging algorithm. 41

4.21: Type 2 instruction merging algorithm. 42
4.22: (a) Control flow graph and the (b) PHG graph. 43
4.23: Predicate define operations added to the blocks. 43
4.24: Algorithm for finding control path. 44
5.1: Performance comparison of basic block scheduling (O), innermost loop predication (IP), and partial predication (PP). 49
5.2: Performance comparison of inner loop predication (IP), inner loop predication with speculative execution (IO), partial predication (PP), and partial predication with speculative execution (PO). 50
5.3: Performance comparison of basic block (O), superblock (T1), and hyperblock (PO). 50
5.4: Performance of predicated execution with and without predicate register forwarding. 51
5.5: Code expansion of the different schemes. 52
5.6: Predicate register usage. 52
5.7: Normalized branch penalty of basic block, superblock, and hyperblock. 53

1. INTRODUCTION

Recently, there has been a great deal of research on superscalar and VLIW (Very Large Instruction Word) processors to increase the computational performance. This performance improvement is only possible if the program is able to utilize the additional resources in these superscalar processors. In scientific programs, sufficient parallelism between instructions can be exposed by scheduling and optimization. On the other hand, many real nonscientific applications have insufficient instruction level parallelism to be able to utilize many of the resources that exist in superscalar machines due to the large number of conditional branches, data dependencies between instructions, and small number of instructions within each basic block.

Predicated execution is an efficient method to handle conditional branches. Predicated or guarded execution refers to the conditional execution of instructions based on the value of a Boolean predicate register. When this predicate is set to true, the instruction is executed normally. When the predicate is set to false, the instruction is treated as *no_op*. Predicated execution enables the compiler to eliminate many conditional branches within the program.

In order to efficiently perform optimization and scheduling on the predicated code, a new structure called hyperblock is introduced and discussed in detail. Implementation issues regarding the hyperblock structure are presented in this thesis.

1.1 Related Works

Control and data dependencies have been studied by many researchers. Towle discussed many techniques for increasing the parallelism within a program [1]. The central idea is to transform control and data dependencies within a program into an equivalent program with the smallest execution time. Before applying the transformation technique, a data dependence graph is built. If the dependence graph is acyclic, techniques such as assignment reordering, variable renaming, statement substitution, and redundant statement removal can be used. Assignment reordering can be used to reduce the height of an expression tree, thus decreasing the execution time of that expression. Variable renaming, statement substitution, and redundant statement removal can be used to remove output and anti-dependencies and potentially increase the parallelism within the code. If the data dependence graph is cyclic, then the wavefront or recurrence method can be used to improve the code.

Towle also discussed vectorizing DO loops with if statements using the distribution algorithm. The algorithm divides loops into four types. Type 1 DO loops contain if statements which are unaffected by the execution of the loops. Type 2 DO loops require some *mode bits* to be set or reset prior to the execution of the main loop. The hardware has to check for these *mode bits* and execute the elements of the loop according to individual bit value. Type 3 DO loops require both paths of the if path to be executed. Then these computations are merged to get the correct result at the end of the loop. The last type of DO loops with if statements must be executed serially.

The Program Dependence Graph (PDG) is introduced to represent data and control dependencies in a single graph [2]. The PDG allows incremental optimization by modifying the PDG as necessary after each transformation. Optimizations which previously require both data and control dependency graphs can now be easily applied using just one combined graph. Ferrante et al. also presented an algorithm to compute the control dependencies in a control flow graph and convert the control information into the control dependency subgraph of the PDG. Even though this control dependence subgraph is an accurate representation of the control dependencies in the graph, much of the information about the original control flow information is lost. Thus, an approximate graph of the control dependencies based on hammock scheme can be built to help in reconstruction of the original flow graph. The data dependence subgraph of the PDG can be constructed by associating statements and predicates as the nodes of the graph.

Using the PDG, Park and Schlansker proposed an algorithm to convert control dependencies into data dependencies [3]. The control dependency algorithm from PDG is used to compute the dependencies between basic blocks in a program. This control dependency information can be used to remove branch statements in basic blocks. Then these blocks can be merged into a single structure that is executed sequentially when the structure is entered. This technique of converting control to data dependency is called if-conversion [1, 4]. The main problem in if-conversion is the assignment of a predicate for each basic block and the placement of the instructions to set these predicates. The predicate is used to determine if a basic block should be executed. A basic block is executed if its corresponding predicate is determined to be true. After assigning the predicates for the basic blocks, new predicate define statements are placed in the appropriate location to selectively turn on these predicates. The R-K algorithm presented by Park and Schlansker efficiently computes the control dependencies within a set of selected

basic blocks for if-conversion. Then the control dependencies are decomposed into two functions. One function assigns the predicates to the basic blocks. The other determines the placement of the predicate define statements within the basic blocks.

Predicated execution is implemented in the Cydra 5 Supercomputer, which is a VLIW multiprocessor system using directed-dataflow architecture [5]. Each instruction word contains seven operations, each of which may be individually predicated. An additional source operand added to each operation specifies a predicate located with the predicate register file. The predicate register file is an array of 128 1-bit registers. Within the processor pipeline after the operand fetch stage, the predicate specified by each operation is examined. If the content of the predicate register is '1', the instruction is allowed to proceed to the execution stage; otherwise, it is squashed. Software pipelining taking advantage of predicated execution support is implemented in the Cydra 5 compiler [8].

A technique for overlapping loops on a pipelined multiple issue processor machine is discussed in [6]. This scheme consists of if-conversion of the inner loops. With special hardware support, these predicated loops can be overlapped and executed more efficiently. A technique is presented to overlap DO loops, WHILE loops, REPEAT-UNTIL loop, and loops with multiple number of exits. Eager execution is introduced to allow an instruction to be executed before the value of the predicate associated with the instruction is known.

Another technique similar to the predicated execution scheme is discussed by Hsu and Davidson in [7]. This scheme with modest hardware support is called the guarded execution. The motivation is that with a longer instruction pipeline and also multiple instruction issue machines beyond a certain level, it becomes more difficult to achieve much of the potential performance. This is mainly due to programs with a small number of instructions in each basic block, data dependencies between instructions, and conditional branch instructions. Because of the uncertainty of the conditional branches, pipelines

with longer stages are not able to achieve much higher performance. Their solution is to use guarded store and jump operations in the branch delay slots. This allows some useful instructions to be issued while waiting for the branch to be resolved. The guarded instructions are made up of regular store or jump instructions associated with a Boolean flag, which determines if these instructions should be executed or not. Using the guarded instruction technique, performance improvement is observed even in programs with many data dependencies and large number of branches.

1.2 Organization of the Thesis

This thesis is divided into six chapters. Chapter 2 describes about the compiler modifications to support predicated execution. Chapter 3 discusses the hardware support requirements for predicated execution. Chapter 4 is the main focal point of this thesis and contains compiler techniques for hyperblock formation, optimization, and scheduling. Chapter 5 presents some performance results to evaluate the effectiveness of the hyperblock technique compared to other schemes, and Chapter 6 provides some concluding remarks.

2. OVERVIEW OF THE COMPILER

A high level diagram showing the components of the modified back-end of the IMPACT compiler to support predicated instructions is presented in Figure 2.1. *Box A* is the profile-based block selection. It takes a profiled intermediate code and selects the basic blocks for if-conversion based on the profiled weight, the instruction characteristics, and the size of the basic blocks. *Box B* consists of code transformations which alter the program to facilitate better predication. The code transformation techniques include tail duplication, node splitting, and loop peeling. *Box C* takes a set of blocks with control dependencies and converts the control dependencies within this set of blocks into data dependencies. After if-conversion, each set of blocks is predicated into one single block referred to as a hyperblock. *Box D* contains a predication-specific optimizer for increasing the instruction level parallelism within each hyperblock. The optimizations include instruction promotion, instruction promotion with renaming, and instruction merging. *Box E* is the conventional optimizer with modification to support predicated instructions. The optimizations include loop unrolling, register renaming, classical optimization, and loop optimizations. After optimization, the code is re-profiled using other input data

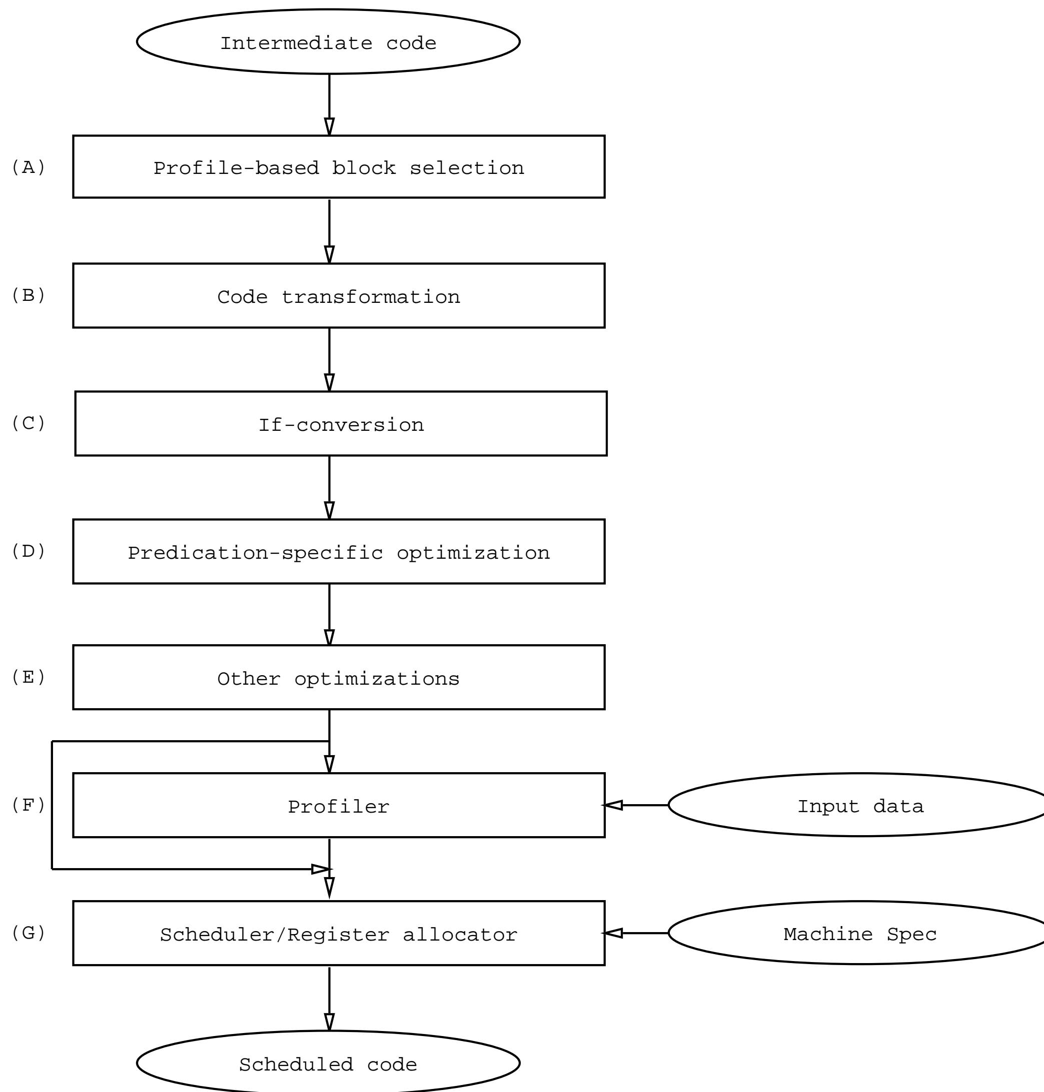


Figure 2.1: A block diagram of the compiler to support predicated execution.

shown in *box F* [1]. Then, the re-profiled, optimized, predicated intermediate code is used as input to the code scheduler and register allocator illustrated in *box G*. The final output is the optimized code scheduled for a specific architecture model.

2.1 Lcode Modifications

The intermediate instruction is based on a low-level machine-independent code called Lcode [9] with modification to support predicated instructions. Each Lcode operation is a RISC-like assembly instruction to facilitate easier code generation to different target machines. The Lcode changes consist of adding a predicate field for all existing instructions to specify a predicate register. An empty predicate field indicates that the instruction is always executed. Another change to the Lcode includes a new set of operations to define the predicate registers according to a specified condition. There is one-to-one correspondence of each predicate define operation to each regular compare and branch instruction. This allows the predication algorithm, which will be described in detail in Chapter 3, to efficiently convert branch instructions to predicate define operations. A new instruction is added to Lcode to clear the value of the specific predicate register. Other new instructions are added to Lcode to allow predicate register spilling. The last change to Lcode includes an additional predicate merge instruction. This instruction is used to assist the reverse translation algorithm to simplify the translation routine. The reverse translation is used to convert predicated instructions back to ordinary operations on architectures without predicated execution hardware support but still want to take advantage of the benefits of scheduling and optimization within the predicated code.

The modifications to the existing Lcode are listed below.

1. Predicate define instructions

[1] This stage is optional. If precise running time is not required, *box F* can be skipped.

- Format : opcode (pred dest) (src1 src2) (pred use) (ext)
- Opcode : Compare and set predicate register.
  - pred_gt, pred_lt, pred_le, pred_ge, pred_ne, pred_eq .
  - pred_gt_u, pred_lt_u, pred_le_u, pred_ge_u.
  - pred_gt_f, pred_lt_f, pred_le_f, pred_ge_f, pred_ne_f, pred_eq_f.
  - pred_gt_f2, pred_lt_f2, pred_le_f2, pred_ge_f2, pred_ne_f2, pred_eq_f2 .
- Pred dest : The predicate register to hold the comparison result.
- Operation : Compare src1 with src2 and set true and false parts of a predicate register.
- Pred use : Null field means always true. Instruction can use the true or false form of the predicate.
- Ext : List of additional information for the scheduler, register allocator, optimizer, and reverse conversion.
  - (pd #) predicate destination virtual register, used by post-pass scheduler and reverse translator after register allocation.
  - (pc #) predicate compare. Indicates the identity of the original branch for which the predicate compare was generated. Multiple predicate compare instructions with the same # are from the same branch.
  - (ppt weight) The weight of the true path of the original branch.
  - (ppf weight) The weight of the false path of the original branch.
  - (pt/pf #) The virtual number of the predicate virtual register used to predicate the execution of the current instruction. Pt means using true part and pf means false part.
- Example :
  (op 29 pred_gt ((r 5 p)) ((r 2 i)(i 5)()) (()) ((pd 5)(pc 3)(ppt 0)(ppf 0))).

2. Predicate clear instruction

- Format : pred_clear (pred dest) (()()()) () ()
- Operation : Clears both the true and false parts of the predicate register.
- Pred dest : The predicate register to be cleared.
- Example :
  (op 5 pred_clear ((r 4 p)) (()()()) (()) ()).

3. Predicate load/store instruction

- Format : opcode (pred dest) (src1 src2 src3) () ()
- Opcode : pred_ld and pred_st.

- Operation : Used by the register allocator to load and store predicate register values.
- Pred dest : The predicate register for pred_ld.
- Pred use : The source operands for pred_ld and pred_st.
- Example :
  (op 5 pred_ld ((r 4 p)) ((mac $SP i)(i 20)()) (()) ()).

4. Predicate merge instruction

   - Format : pred_merge () (()()()) (pred use) (ext)
   - Operation : Compare src1 with src2.
   - Pred use : Null field means always true. Instruction can use the true or false form of the predicate.
   - Ext : List of additional information for reverse conversion.
     - (pm #) one of the predicate registers that participates in the merge. The appearance terminates the lifetime of the virtual register and merges the true and false paths of the predicate registers into one.
     - (pt/pf #) The virtual number of the predicate virtual register used to predicate the execution of the current instruction. Pt means using true part and pf means false part.
   - Example :
     (op 36 pred_merge (()) (()()()) ((r 8 pt)) ((pm 6)(pm 7)(pt 8))).

5. Other instructions

   - Format : opcode (dest) (src1 src2 src3) (pred use) (ext)
   - Operation : Original operation if predicate is true. nop if predicate is false.
   - Pred use : Null field means always true. Instruction can use the true or false form of the predicate.
   - Ext : in addition to the other extension usages :
     - (pt/pf #) The virtual number of the predicate virtual register used to predicate the execution of the current instruction. Pt means using true part and pf means false part.
   - Example:
     (op 13 add ((r 3 i)) ((r 3 i)(i 1)()) ((r 7 pt)) ((pt 7))).

2.2 Software Modifications

New algorithms are added to the existing IMPACT-I compiler to support predicated execution. The additional modules include an algorithm to select favorable blocks for predication, if-conversion routine to convert control dependencies to data dependencies, and optimizer to allow speculative execution. Another modification includes extending the conventional optimizer to support predicated instructions. Register allocation is modified to support predicate registers, minimize the number of predicate register usages, and minimize the register spill instructions. The code scheduler is changed to support scheduling of the predicated instructions by removing excess dependencies of predicate instructions on alternate paths. After the excess dependencies are removed, the same dependence analysis routines are used to find an efficient scheduled code.

3. HARDWARE SUPPORT

In addition to compiler support for predicated execution, some special architectural support is also required. The hardware must allocate enough spaces to store the predicate values. Even though the existing general purpose registers can be used to store these values, there are two problems of using general purpose registers. First, each predicate value requires only 1 bit. Therefore, storing the value in a typical 32-bit general purpose register can be very wasteful. Second, because many of the predicate values and their comlementary values are used (representing the true and false paths of an if statement), more efficient and convenient hardware support should be used to represent the complementary values. A new predicate register file design is discussed to address these problems.

Another hardware support includes providing some new instructions to set these predicate registers. These predicate registers are used by all instructions; thus, an extension has to be added to the instruction field to specify a predicate register. And lastly, the hardware must contain some logic to nullify any side effects of the instruction if the instruction is not supposed to be executed.

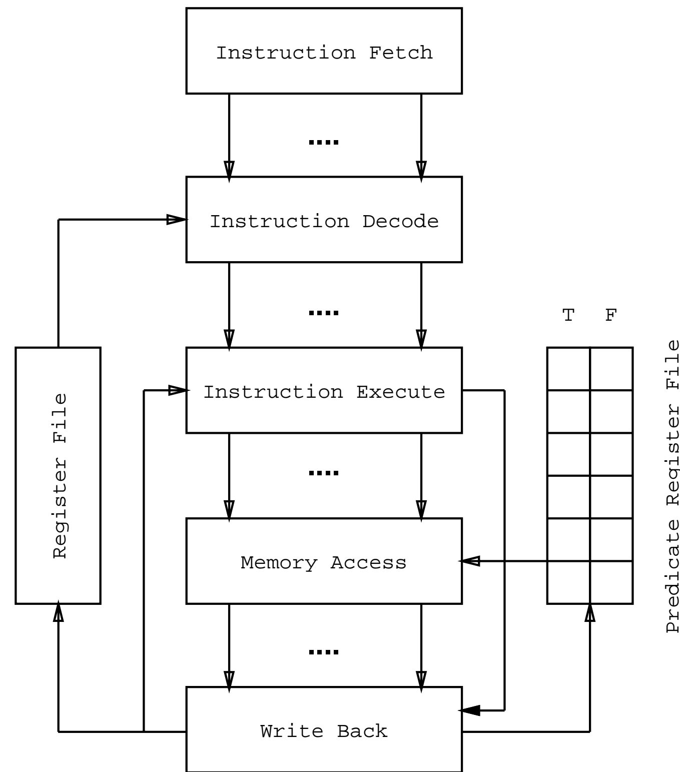


Figure 3.1: A superscalar processor pipelined model with a predicate register file.

A high-level superscalar processor pipelined model used to support predicated execution is shown in Figure 3.1. This model is a statically scheduled multiple instruction issue machine. The model consists of five pipelined stages [10]. A predicate register file is added to store the value of the predicates. The write-back stage can write the result to the predicate register file. Later, these values can be read before the memory access stage to squash out the instructions that are not supposed to be executed.

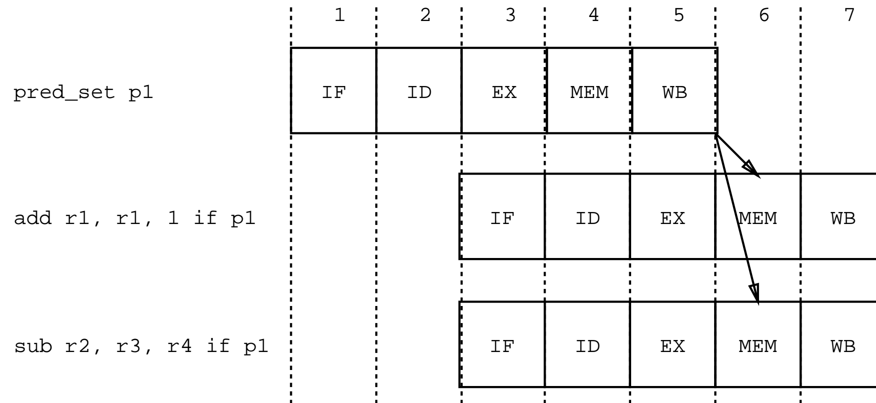
Each predicate register consists of two bit Boolean flags. These two Boolean flags are set to complementary values by any predicate set instructions. Thus, the three possible states of each predicate register are false-false, false-true, and true-false. An instruction can be predicated on the true form or false form of the predicate register.

Predicate registers may be set to false-false by the **pred_clear** instruction. Other predicate set instructions may be used to set a particular predicate register to true-false or false-true conditionally depending on the result of the comparison. For example, '**pred_gt p1, r0, 0**' instruction can set the predicate register, p1, to true-false if r0 is greater than zero. Otherwise, p1 is set to false-true.

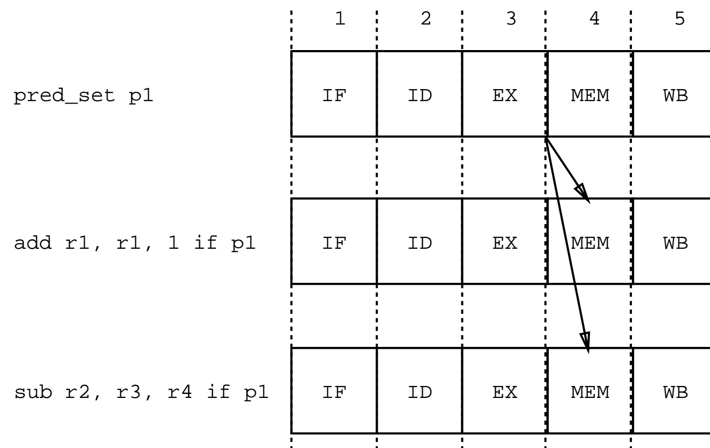
This type of predicate register file is used to reduce the number of instructions required to set a predicate register and to reduce the flow dependence length from the setting of the predicate register to its first use.

Other techniques can be used to reduce the flow dependence length even further. Figure 3.2(a) shows the data flow dependency from instruction 1 to instructions 2 and 3. In this pipelined processor example, instruction 2 or 3 can only be issued two cycles later after the predicate register is written back to the predicate register file[1]. With some simple hardware, this data dependency distance can be reduced. Figure 3.2(b) describes the execution sequence of the same three instructions on an architecture with predicate register forwarding logic. Since the result of the predicate is known at the end of the instruction execute stage, the result can be forwarded to the memory write stage of the other two instructions. This scheme allows an instruction which is flow dependent on the predicate register of a previous instruction to be issued two cycles earlier with enough resources on this processor pipelined model. The experimental result illustrating the effectiveness of the register forwarding scheme is discussed in Chapter 5.

[1]In this example, assume the instruction can only update the state of the machine during the MEM and WB stages.



(a)



(b)

Figure 3.2: Example of predicate register forwarding. (a) Without forwarding. (b) With forwarding.

4. HYPERBLOCK FORMATION AND OPTIMIZATION

A hyperblock is a set of predicated basic blocks in which control may only enter from the top, but may exit from one or more locations. A single basic block in the hyperblock is designated as the entry. Control flow may only enter the hyperblock at the entry. Figure 4.1 illustrates a segment of an original control flow graph after block selection (a), after tail duplication (b), and after if-conversion (c). Figure 4.2 illustrates a segment of assembly code before and after hyperblock formation. This example assumes L5 is not an important block and it is not predicated within the main hyperblock.

Like hyperblock, a superblock is a block of instructions such that control can only enter from the top but may exit from more than one location [11]. But unlike hyperblock, the instructions within each superblock do not contain any predicated instructions. A superblock can be seen as a special case of hyperblock where only one path of the basic blocks is combined.

One benefit of the hyperblock scheme is the flexibility to combine one or more control paths. Other schemes such as trace scheduling [12] and superblock scheme can only optimize along a single path of control. In a program where branches are not heavily biased toward any direction, schemes which only concentrate on one main path will

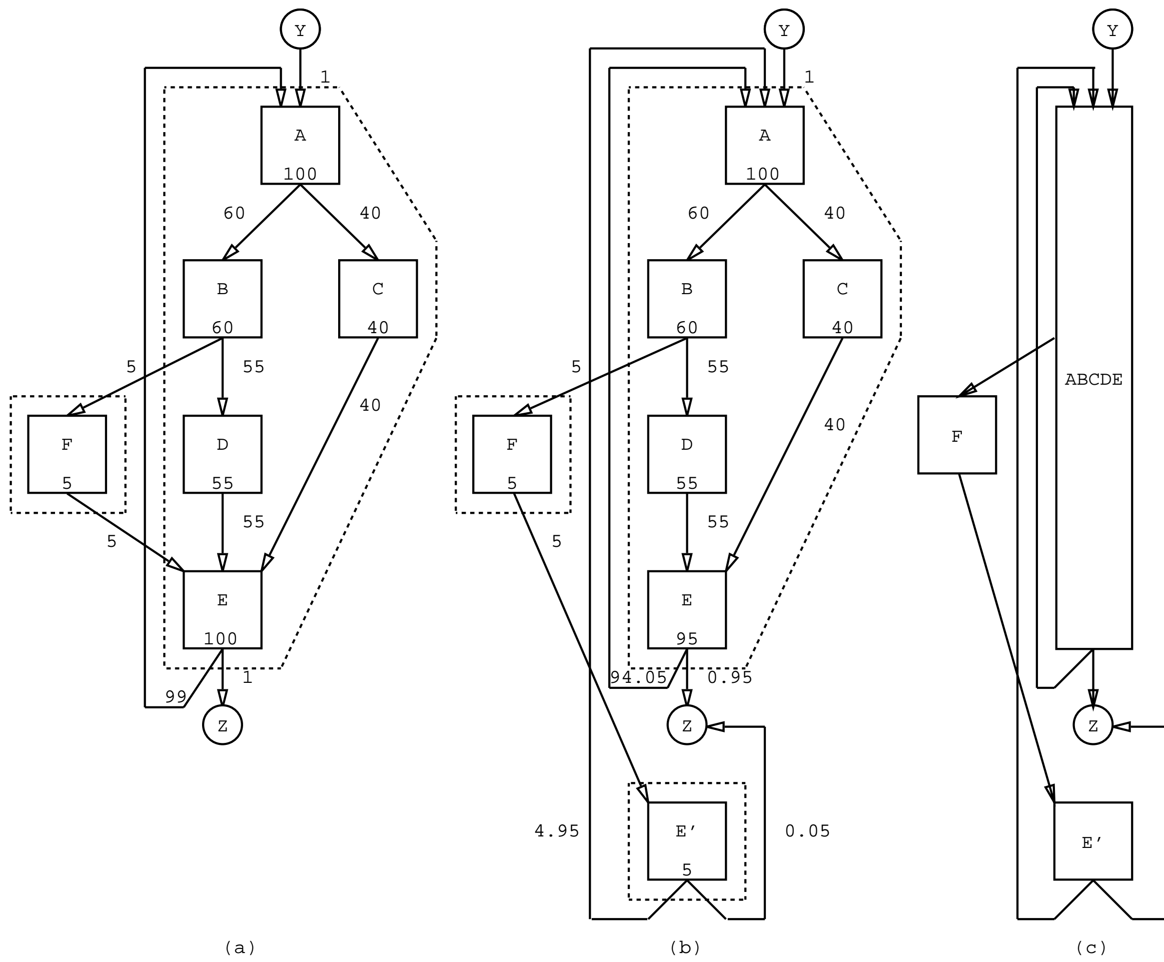


Figure 4.1: An example of hyperblock formation. (a) Original program after block selection. (b) After tail duplication. (c) After if-conversion.

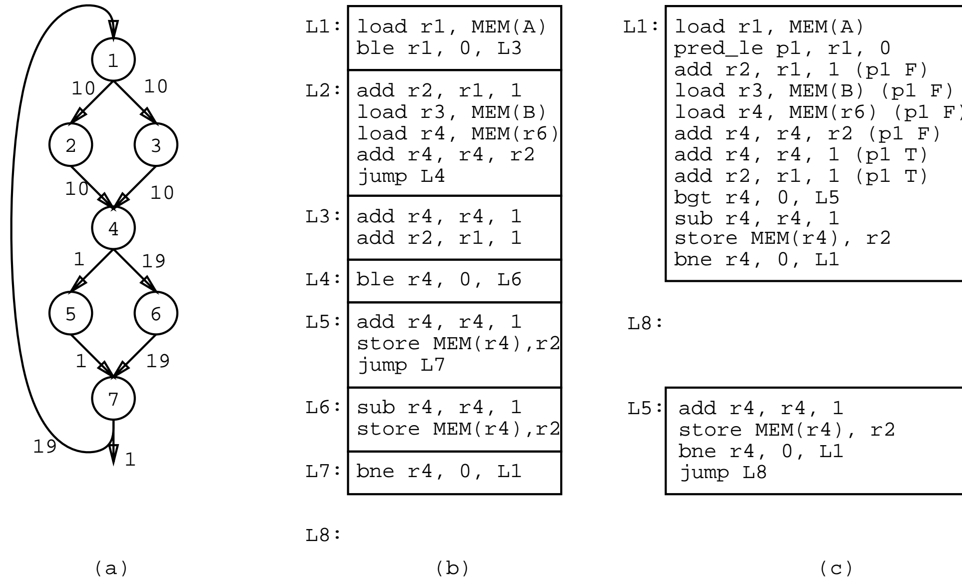


Figure 4.2: A program segment before and after hyperblock formation. (a) Original control flow graph. (b) Original program segment. (c) After hyperblock formation.

miss optimization opportunities across other paths. Hyperblock techniques, on the other hand, are structured for multiple paths. Thus, optimization and scheduling will be able to benefit more by looking across multiple numbers of important paths.

4.1 Block Selection

An important issue in hyperblock formation is deciding which blocks should be predicated together. One scheme is to predicate all blocks together in an inner-most loop [6]. With a limited number of resources (fetch units or functional units), this technique may degrade performance in a control intensive code where multiple numbers of paths are conditionally executed at the same time.

There are three important factors when deciding a basic block for inclusion into a hyperblock. The first factor is to use profile information to assist in the selection of the basic blocks for if-conversion. A profile-based algorithm selectively removes paths that

are not executed often. Figure 4.1 shows a piece of profiled control flow graph. The number associated with each basic block indicates the execution frequency of that block. In this example, the path consists of block F and is executed less frequently. Therefore, it is not included in the formation of the main hyperblock which includes blocks A, B, C, D, and E.

The second factor is to select the basic blocks based on the size of the basic blocks. Basic blocks that are larger should have lower priority for predication than smaller blocks.

Finally, the instructions within each basic block are important in deciding which block should be predicated. Procedure calls and unresolvable memory access instructions should have less priority for inclusion due to the fact that these instructions can potentially limit the optimization and scheduling.

A heuristic function used to approximate the criteria for selecting a basic block is shown below. This heuristic function takes the execution frequency, basic block size, and instruction characteristic within the basic block into consideration.

$$BSV_i = (K \cdot \frac{weight_bb_i}{size_bb_i} \cdot \frac{size_main_path_1}{weight_main_path_1} \cdot bb_char_i)$$

The block selection value, or BSV, is calculated for each basic block to indicate its importance. A larger BSV value indicates that the basic block is more important for inclusion. The weight and the size of each basic block are normalized against the main path of the basic blocks. The variable bb_char_i is the basic block characteristic of each basic block. The maximum value of bb_char_i is 1. Blocks containing procedure calls or unresolvable memory references have bb_char_i less than 1. K is a machine-dependent constant. Machines that have a higher issue rate have a higher K value to allow more basic blocks to be selected for if-conversion. An algorithm for block selection is shown in Figure 4.3.

```
block_selection()
{
    Determine the header block, bb_1. Usually the header block of a loop.
    selected_set = set of basic blocks on the main path starting from bb_1
    calculate all BSV of blocks not in selected_set
    done = false
    while (not done) {
        done = true
        for each successors, bb_i, of blocks in selected_set
            if (BSV_i > threshold and bb_i is not in selected_set) {
                selected_set = selected_set + {bb_i}
                done = false
            }
        }
    }
    return selected_set
}
```

Figure 4.3: Block selection algorithm.

4.2 Code Transformation

After the blocks are selected, two conditions must be true before the selected blocks can be if-converted and transformed into a hyperblock.

Condition 1 : No incoming arc from outside the basic blocks to blocks to be predicated other than to the entry block.

Condition 2 : No nested inner loop inside the blocks to be predicated.

These conditions ensure that the hyperblocks are entered only from the entry blocks and also that the instructions in a hyperblock are executed at most once before the hyperblock is exited. Techniques used to satisfy conditions 1 and 2 are called tail duplication and loop peeling, respectively. If a group of basic blocks satisfies the above two conditions, these blocks can be if-converted easily using the if-conversion algorithms described later.

4.2.1 Tail duplication

Tail duplication is used to transform programs to satisfy condition 1 for if-conversion. As mentioned before, condition 1 basically means that if the predicated blocks are to be executed the control must begin at the entry block. If a selected group of blocks violates condition 1, then the original program must be transformed to satisfy this condition.

The tail duplication algorithm transforms the control flow graph by first marking all of the flow edges that violate condition 1. Then all of the basic blocks that are to be predicated are marked if the flow arcs can reach them before reaching the entry block. Finally, all the marked blocks are duplicated, and the marked flow arcs now adjusted to point to the duplicated blocks which correspond to the original blocks. Figure 4.1(a) shows that the control flow from block F to block E is a side entrance to the main hyperblock and must be eliminated. Figure 4.1(b) shows the flow graph after block E is duplicated. The control from F to E now changes to duplicated block E. A tail duplication algorithm is presented in Figure 4.4.

4.2.2 Loop peeling

To satisfy condition 2, another transformation called loop peeling is used. Loop peeling peels off iterations of inner loops that are nested within the selected blocks for predication. After peeling off iterations of a loop, the outer blocks can satisfy condition 2 for if-conversion. When the inner loop blocks and the outer loop blocks are transformed into hyperblocks, loop unrolling can then be easily applied to these hyperblocks. Figure 4.5(a) shows two loops, one nested within another. After loop peeling is applied, the outer loop can be predicated together as shown in Figure 4.5(b). Loop peeling is accomplished by first detecting loops nested in the selected blocks for predication. The loop detection algorithm is described in more detail in [13]. After the blocks in the nested

```
tail_duplication(set_of_bb)
{
        Let set_of_bb be the blocks that are selected for hyperblock formation.
                and bb1 is the entry of the hyperblock.
        done = false
        while (not done) {
                done = true
                for each basic block in set_of_bb, bbi {
                        if (bbi ≠ bb1)
                                mark each bbi that has predecessor not in set_of_bb }
                for each basic block in set_of_bb, bbi {
                        if (bbi is marked) {
                                if bbi is not duplicated yet
                                        duplicate bbi
                                else
                                        use previously duplicated bbi
                                change all incoming flow arcs of basic blocks not in set_of_bb from
                                        pointing to bbi to duplicate bbi.
                                done = false
                        }
                }
        }
}
```

Figure 4.4: An algorithm for tail duplication.

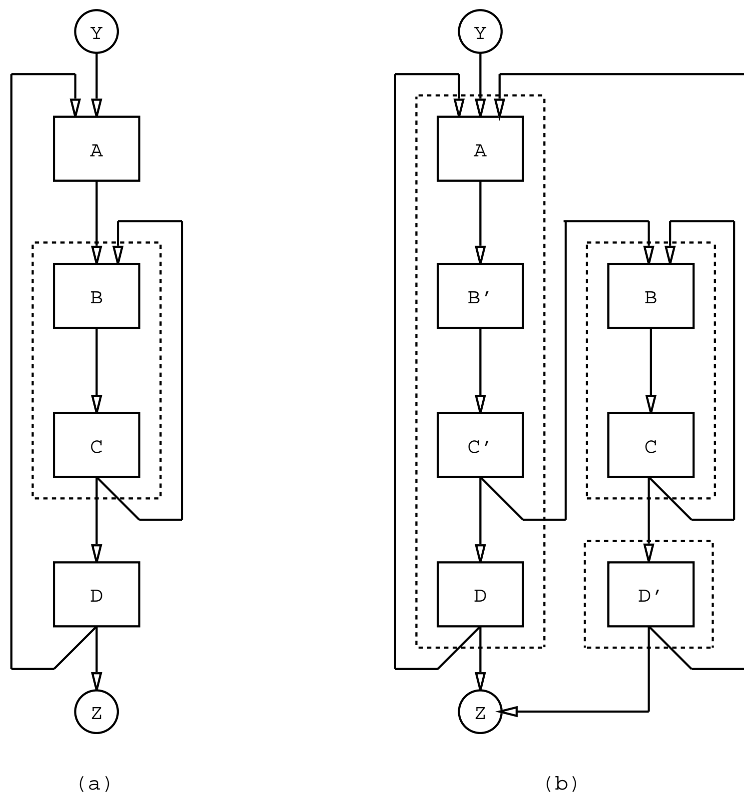


Figure 4.5: An example of loop peeling.

```
loop_peeling(set_of_bb)
{
        Let set_of_bb be the blocks that are selected for hyperblock formation.
        and bb_1 is the entry of the hyperblock.
        for each inner loop in set_of_bb, loop_i {
                k = expected number of times loop_i is executed each time through the loop
                if (k > MAX_NUM_PEELS) k = MAX_NUM_PEELS
                duplicate loop_i k times
                change the loop back arc from last duplicated loop_i to the
                        original header block of loop_i
                add the duplicated loop_i to set_of_bb
        }
}
```

Figure 4.6: An algorithm for loop peeling.

loops are detected, several iterations are peeled off. Finally, all loopback flow arcs to the original nested loops are modified to point to the target of the duplicated loops. An algorithm for loop peeling is shown in Figure 4.6.

4.2.3 Node splitting

After tail duplication and loop peeling, another technique called node splitting can be applied to a segment of the program. This technique duplicates the basic blocks that are at a merge point of two or more blocks. Node splitting is especially effective at higher issue rates in control intensive programs where control and data dependencies limit the number of independent instructions. Node splitting can also be effectively applied to a basic block where one incoming path to this block contains a longer chain of dependencies than the other incoming paths. Node splitting these blocks effectively improves the performance since the execution time will no longer be dictated by the path with a longer chain of dependencies. Figure 4.7(a) shows an example of a flow

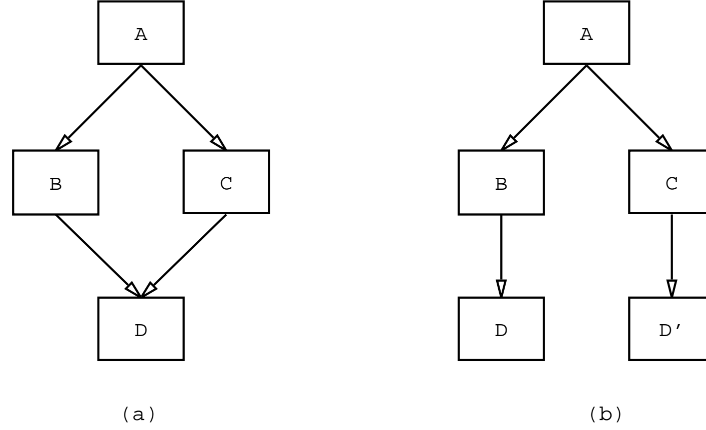


Figure 4.7: An example of node splitting.

graph before node splitting. Figure 4.7(b) shows the flow graph after the node splitting is applied to block D.

There are also disadvanges to node splitting. This technique can cause large code expansion. Heuristics need to be added to the algorithm to prevent node splitting if a basic block does not benefit from it. One heuristic function is shown below.

$$FSV_i = (K \cdot \frac{weight_flow_i}{size_flow_i} \cdot \frac{size_main_path_1}{weight_main_path_1} \cdot bb_char_i)$$

$$\Delta FSV_i = max(abs(FSV_j - FSV_k)) \text{ and } FSV_j, FSV_k \text{ target same } bb_i$$

The flow selection value, or FSV, is calculated for each flow edge in the set of blocks selected for if-conversion that contains two or more incoming edges. Larger FSV implies the importance of the flow edge. $Weight_flow_i$ is the number of times the control flow through edges traversed. $Size_flow_i$ is the number of instructions on the path from the entry block to the point where the flow edge is taken. bb_char_i and K are the same parameters used in calculating BSV for block selection.

After the FSVs are computed, the node splitting is processed starting from the basic block with highest ΔFSV_i. High ΔFSV_i indicates highly unbalanced control flow path. Thus, basic blocks with the highest difference should be node split first. The algorithm

```
node_splitting(set_of_bb)
{
        Let set_of_bb be the blocks that are selected for hyperblock formation.
                and bb_1 is the entry of the hyperblock.
        merge_set= set of blocks that has two or more predecessors excluding bb_1
        done = false
        while (merge_set is not empty and not done) {
                calculate FSV of all source edges of blocks in merge_set
                calculate Δ FSV for each block in merge_set
                bb_i = block with largest Δ FSV
                done = true
                if (weight_bb_i > threshold_1 and ΔFSV_i > threshold_2) {
                        done = false
                        num = number_of_predecessors(bb_i)
                        duplicate bb_i num − 1 times
                        modify each predecessor of original bb_i to point to different copy of bb_i
                }
                update merge_set
        }
}
```

Figure 4.8: An algorithm for node splitting.

continues until there are no more blocks with two or more incoming edges, no Δ FSV above a certain threshold, or no basic blocks with weight above a minimum threshold. An algorithm for node splitting is presented in Figure 4.8.

4.3 If-Conversion

If-conversion is a process in which conditional branches are selectively removed by converting control dependencies to data dependencies. In this section, three if-conversion algorithms for predicating a group of blocks into a hyperblock are presented. Algorithm 1 predicates blocks by associating a predicate with each conditional flow edge. Operations inside a basic block are conditionally executed when one of the predicates associated with

the flow edge into the block is set to TRUE. Algorithm 2 predicates blocks by associating a predicate directly with each basic block. Operations are conditionally executed when the predicate associated with this block is set to TRUE. Lastly, algorithm 3 also associates a predicate with basic blocks similar to algorithm 2. However, this algorithm reduces the number of predicate assignments and definitions [3].

Branch instructions can be classified into three types: forward, backedge, and exit. A forward branch has a target basic block in the same hyperblock excluding the entry basic block. A backedge branch has a target in the entry block of the same hyperblock. An exit branch has a target outside the hyperblock.

The if-conversion algorithms described in this section convert all forward branches. The backward and exit branches are not converted because if-converting these branches may cause performance degradation due to execution of extra useless predicated instructions.

4.3.1 Dominator and post dominator

Before the if-conversion algorithm can predicate a group of blocks, dominator and post-dominator information of these blocks must be computed. A basic block, bb_i, dominates another block, bb_j; if bb_j is visited, then bb_i must also be visited [13]. Figure 4.9 shows an iterative algorithm which computes the dominator information of a group of basic blocks. This algorithm assumes that bb_0 is the initial basic block of the group. In Figure 4.10, block A dominates A, B, C, and D. Block B dominates B and D. Block C and D dominate only themselves.

After dominator information is calculated, the post dominator of these basic blocks must also be computed. A basic block, bb_i, post dominates another block, bb_j; if bb_j is executed, bb_i is also executed. Before computing the post dominators of the basic blocks,

```
compute_dominator(set_of_bb)
{
    ⋃ = the set of all basic blocks in set_of_bb
    dom(bb_0) = {bb_0}
    for all basic blocks ∈ set_of_bb, bb_i {
        post_dom(bb_i) = ⋃
    }
    change = 1
    while (change ≠ 0) {
        change = 0
        for all basic blocks ∈ set_of_bb, bb_i, (i ≠ 0) {
            old = dom(bb_i)
            dom(bb_i) = ⋂(dom(bb_j)), bb_j a predecessor of bb_i
            dom(bb_i) = dom(bb_i) + {bb_i}
            diff = old − dom(bb_i)
            if (diff ≠ 0) then change = change + 1
        }
    }
}
```

Figure 4.9: Algorithm for finding dominators.

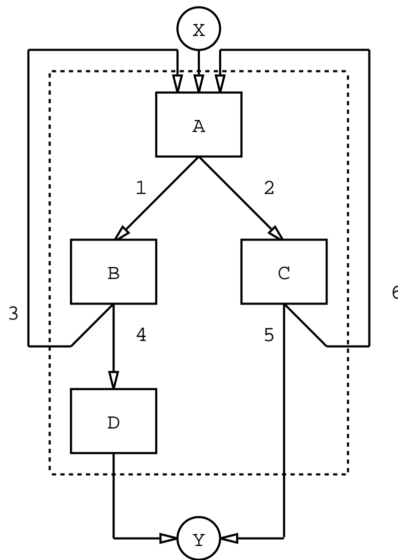


Figure 4.10: A simple control flow graph.

all exit branches and backedge branches are marked and eliminated from the computation of the post dominator information.

The algorithm similar to the iterative algorithm used to compute dominator information is shown in Figure 4.11. The exit branches and backedge branches are removed from the computation of post dominator information to facilitate better utilization of the predicate registers. For example, Figure 4.10 shows a simple control flow graph of four basic blocks within a hyperblock. In this example, blocks B and C are predicated with complementary predicate registers. Even though basic block D is control dependent on the branch condition in B, it can use the same predicate register as B if the backedge branch in B is not if-converted. After removing the exit branch 5 and backedge branches 3 and 6, the post dominator of the blocks can be calculated. In this case, D is the post dominator of D and B. A, B, and C post-dominate only themselves.

4.3.2 Algorithm 1 : Association of flow arc

This if-conversion algorithm is a very simple scheme but not very efficient in terms of predicate register usage and define operations. In this algorithm, a predicate register is used to hold the value of the computation from the conditional flow arcs. The basic blocks are if-converted according to a topological order starting from the entry basic block which is predicated with the constant TRUE predicate. Two basic blocks can share the same predicate register if they have the same control dependencies. Two basic blocks have the same control dependencies if the first block dominates the second and the second post-dominates the first.

To generate a predicate register for a particular basic block that does not have the same control dependencies with a basic block already predicated, a new predicate register is allocated. The predicate flag is set to TRUE if any one of the incoming flow arcs are

```
compute_post_dominator(set_of_bb)
{
        ⋃ = the set of all basic blocks in set_of_bb
        for all basic blocks ∈ set_of_bb, bb_i {
                post_dom(bb_i) = ⋃
        }
        mark off all exit and backedge flow arcs

        change = 1
        while (change ≠ 0) {
                change = 0
                for all basic blocks ∈ set_of_bb, bb_i, (i ≠ 0) {
                        old = dom(bb_i)
                        dom(bb_i) = ⋂(dom(bb_j)), bb_j a successor of bb_i
                                and flow arc from bb_i to bb_j is not marked off
                        dom(bb_i) = dom(bb_i) + {bb_i}
                        diff = old − dom(bb_i)
                        if (diff ≠ 0) then change = change + 1
                        }
                }
        }
}
```

Figure 4.11: Algorithm for finding post dominators.

```
predication_1(set_of_bb)
{
        Compute_dominator(set_of_bb)
        Compute_post_dominator(set_of_bb)
        Examine basic blocks in the topological order.
        Assign a predicate register to each conditional forward flow arc.
        for each basic block, bb_i {
                if (there exist bb_j such that bb_j comes before
                        bb_i in topological sort and bb_j dominates bb_i and bb_i
                        post-dominate bb_j)
                        bb_i use the same predicate register as bb_j
                else {
                        assign a new predicate register to bb_i
                        Add ORing instructions to set the predicate if any one flow arc is true
                }
                Convert forward branch in bb_i to predicate set operations
                Predicate all other instructions in bb_i with the register assigned to it
        }
        Add operations to reset predicate registers
}
```

Figure 4.12: Algorithm 1 for if-conversion.

predicated to TRUE. Thus, to enable the operations in a basic block, the algorithm generates the 'OR' instructions which set the predicate register of the basic block if one of the registers associated with the flow arcs is set.

After forward branches are converted to the predicate flag define operations, the basic block is predicated using the predicate register assigned. Also, the predicate registers that are defined within the context of another predicate register must be reset at the entry block. This prevents a predicate register from being used prior to being defined. Figure 4.12 contains a simple high level description of this algorithm.

4.3.3 Algorithm 2 : Association of basic blocks

The second algorithm for predicating a group of basic blocks works similarly to the first algorithm. But unlike the first algorithm which associates a predicate register with each flow edge, this scheme uses the flow edges to directly set the predicate register of the target basic block. This scheme is more efficient because no predicate register is needed to store the conditions of the flow edges.

Like the first algorithm, each basic block is predicated in turn whenever its predecessors are all predicated excluding the entry block, since the entry block is always predicated with constant TRUE. Two basic blocks can use the same predicate registers if the first block dominates the second block while the second post dominates the first.

Next, in a topological sort, the basic block that uses a predicate register in the earliest instance has to add predicate define operations prior to using the predicate value. Each incoming forward flow arc is converted to a predicate define operation. A conditional flow arc sets the predicate register according to its condition. An unconditional flow arc sets the predicate register to constant TRUE, since execution of the basic block that contains an unconditional flow arc implies that the target basic block is also executed. The predicate define operations always inherit the predicate value of source block of the flow arc.

At the beginning of the predicated block, predicate registers which may be used prior to define instruction are reset. Figure 4.13 shows this algorithm.

4.3.4 Algorithm 3 : RK algorithm

The third algorithm for predication is based on the work from [3]. In any if-conversion algorithm there are two main questions that must be solved. First, the algorithm must be able to assign predicates to basic blocks. And second, the algorithm must be able to

```
predication_2(set_of_bb)
{
        Compute_dominator(set_of_bb)
        Compute_post_dominator(set_of_bb)
        Examine basic blocks in the topological order.
        for each basic block, bb_i {
                if (there exist bb_j such that bb_j comes before
                        bb_i in topological sort and bb_j dominates bb_i and bb_i
                        post-dominate bb_j)
                        bb_i use the same predicate register as bb_j
                        Remove all incoming forward branches to bb_i
                else {
                        assign a new predicate register to bb_i
                        Convert all incoming forward branch to bb_i to predicate set operations
                }
                Predicate all other instructions in bb_i with the register assigned to it
        }
        Add operations to reset predicate registers
}
```

Figure 4.13: Algorithm 2 for if-conversion.

place predicate define operations. Algorithms 1 and 2 presented in the previous sections compute the assignment and the placement, although not very efficiently.

The RK algorithm, on the other hand, is an efficient algorithm which minimizes the number of predicate usages and number of define operations. This algorithm formulates two functions called R and K. Function R is responsible for assignment of predicates and function K is required for the placement of the predicate define operations. Then if-conversion can be divided into several stages: compute the post dominator information, compute the control dependencies between instructions, decompose the control dependencies into R and K functions, add some define operations for those predicates that may be used before defined, convert conditional branches, and compact the basic blocks into one single block of code.

The function R is derived by assigning one predicate register to all of the basic blocks with the same control dependencies. Function K is derived directly from the control dependency information. Each distinct set of control dependencies determines the placement of the predicate define operations of its associated predicate. On machines with the complementary predicate register file described in Chapter 2, two basic blocks with complementary control dependencies can be assigned true and false forms of the same predicate register to reduce the number of predicate register usages and the number of define operations.

The augment function of the RK algorithm uses the data flow equations and solves for the predicates which may be used before defined, and inserts resets (pred_clear instructions) to these predicates in the entry block of the hyperblock. After R and K are determined, conditional branches are removed and instructions are predicated based on their assigned predicate. Figure 4.14 shows if-conversion based on the RK algorithm.

```
predication_3(set_of_bb)
{
        Compute_post_dominator(set_of_bb)
        Compute_control_dependencies(set_of_bb)
        Decompose_control_dependencies(set_of_bb)
        Augment_K(set_of_bb)
        for each basic block, bb_i {
                use the predicate register assigned in R
                for each forward conditional flow arc use K to determine which predicates
                        are dependent on this arc. And add the predicate define operation.
                Remove all forward branches
                predicate this basic block with the assigned register
        }
}
```

Figure 4.14: RK algorithm for if-conversion.

4.3.5 Comparison of the three algorithms

Figure 4.15 shows the total cycle time for issue 2 machine using the three schemes on four benchmarks: *cmp, compress, grep,* and *wc*. The resulting speedup is normalized to the execution time of the hyperblock scheme using algorithm 1 if-conversion (ARC). The figure shows that RK outperforms both the BB and ARC algorithms. However, at issue rate 4 and higher, virtually identical performances for the three schemes are observed. One explanation is that at low issue rate, there are a fewer number of empty slots available for scheduling. Thus, since the RK algorithm produces the least number of predicate uses and define operations than the other two schemes, it's more beneficial than the other schemes. On the other hand, at higher issue rates the extra predicate define instructions produced by ARC and BB do not degrade the performance significantly since there are more empty slots available for these excess instructions.

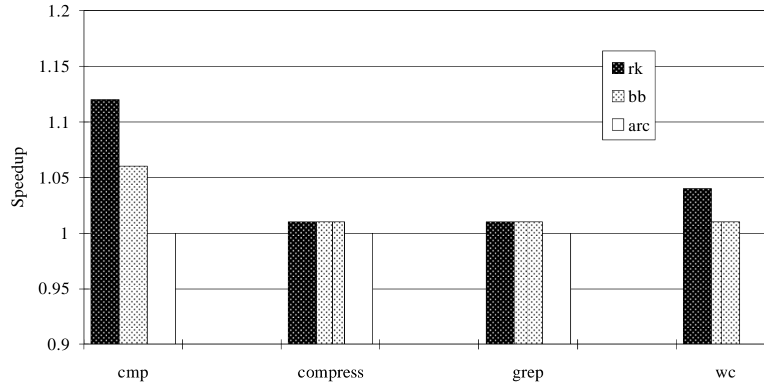


Figure 4.15: Normalized execution time of the three if-conversion schemes.

4.4 Predication-specific Optimizations

In this section, three optimization techniques specific to the predicated code are presented. Instruction promotion is used to speculatively execute instructions before the predicate value associated with the instruction is known. Renaming the register is used in conjunction with promotion to speculatively execute more instructions. Instruction merging can be used to remove redundant operations.

4.4.1 Instruction promotion

Promotion of predicated instruction removes the flow dependency between the predicated instruction and the predicate define operation which sets the predicate value. This technique enables speculative execution of instruction by removing the predicate associated with the instruction. Instructions with long latencies can be scheduled earlier with this technique. By issuing long latency instruction earlier, other instructions which depend on the completion of this instruction may also be initiated earlier. Instructions can be executed speculatively if the result value from executing this instruction later found not required does not affect the execution of subsequent instructions. Another problem of speculative execution is correct detection of exceptions. Exceptions generated

```
instruction_promotion_1()
{
        for each instruction, op(x), in the hyperblock {
                if all the following conditions are true:
                        1. op(x) is predicated.
                        2. op(x) has a destination register.
                        3. there is a unique y such that dest(y) = pred(x) in {op(y), y < x}.
                        4. dest(x) is not live at op(y).
                        5. dest(j) ≠ dest(x) in {op(j), j = y + 1 ... x − 1}.
                then do:
                        set pred(x) = pred(y)
        }
}
```

Figure 4.16: Type 1 instruction promotion algorithm.

from instructions that are not supposed to be executed should be ignored. With non-trapping hardware, instructions that are speculatively executed can use the nontrapping instruction equivalent of the normal instruction. If a precise exception is desired, sentinel scheduling can be used to accurately report the exceptions when the instructions that cause the exceptions are supposed to be executed [14]. Two algorithms for instruction promotion are utilized based on the characteristic of the predicate define operation.

Type 1 promotion operates on predicated instructions only if the predicate is defined by only one predicate define instruction. The algorithm in Figure 4.16 is used for instruction promotion. A predicated instruction can be promoted if the instruction has a destination register that is not live at the unique operation which defines the predicate used by this instruction.

Figure 4.17(b) shows an example code after instruction promotion optimization from the original code shown in Figure 4.17(a). The arrow in Figure 4.17(b) indicates the promoted instruction. Speculatively executing this promoted load instruction allows the instruction to be scheduled before the predicate define instruction. By scheduling this

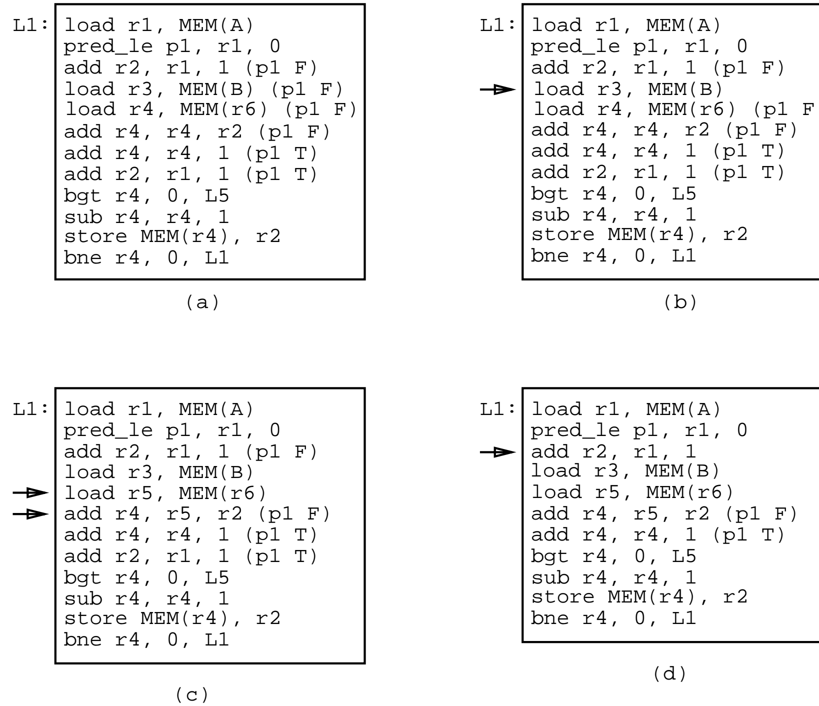


Figure 4.17: An example of code segment after each optimization. (a) Original code. (b) After instruction promotion. (c) After renaming/promotion. (d) After merging.

load instruction earlier, other instructions that use the result of the load can also be scheduled sooner. This results in a more compact code which translates into higher performance.

Predicated instruction can still be promoted if there are multiple numbers of operations which define the predicate used by an instruction. Type 2 promotion operates by promoting instruction above all the relevant predicate define operations as shown in Figure 4.18. A predicated instruction can be promoted to the constant TRUE predicate, if the destination register of this instruction is not live at any of the relevant predicate define operations.

```
instruction_promotion_2()
{
        for each instruction, op(x), in the hyperblock {
                if all the following conditions are true:
                        1. op(x) is predicated.
                        2. op(x) has a destination register.
                        3. there exists more than one y such that dest(y) = pred(x)
                                in {op(y), y < x}
                        4. dest(x) is not live at any instructions which either define pred(x)
                                or define an ancestor of pred(x) in the PHG.
                        5. dest(j) ≠ dest(x) in {op(j), j = y + 1 ... x − 1}.
                then do:
                        set pred(x) = ∅
        }
}
```

Figure 4.18: Type 2 instruction promotion algorithm.

4.4.2 Renaming and promotion

Many operations cannot be promote and speculatively executed due to the destination variable of the instruction live on the alternate paths. A solution is to rename the destination registers of this instruction and rename all subsequent source registers which can only be reached by the same value defined by this instruction. Then another instruction may have to be added to preserve the original value of the register if the path containing the renamed register is taken. An example is presented in Figure 4.17(c) which shows the program after renaming and promotion is applied to Figure 4.17(b). The destination register of the promoted instruction is renamed from R4 to R5 and the subsequent instruction which uses R4 is also renamed R5. The algorithm for renaming with promotion is shown in Figure 4.19.

```
renaming_and_promotion()
{
        for each instruction, op(x), in the hyperblock {
                if all the following conditions are true:
                        1. op(x) cannot be promoted by either type 1 or type 2.
                        2. there exists src(k) = dest(x) in {op(k), k > x}.
                                op(x) dominates op(k).
                        3. dest(x) ≠ dest(z) in {op(z), z = x + 1 ... y − 1}.
                then do:
                        Rename dest(x)
                        Rename all op(y) with src(y) = dest(y) and op(x) dominates op(y)
                        Add new instruction, op(z), to move the value from new register
                                to old register
                        pred(z) = pred(x)
                        pred(x) = ∅
        }
}
```

Figure 4.19: Renaming and instruction promotion algorithm.

4.4.3 Instruction merging

Instruction merging combines two instructions in a hyperblock with complementary predicates into a single instruction which will execute whether the predicate is true or false. Identical instructions with complementary predicates can be merged if the same definition of source register values reaches both instructions. Instructions that satisfy these conditions are merged by promoting one instruction and deleting the complementary instruction. This process of instruction merging is similar to elimination of partial redundancies discussed in [15]. An example of instruction merging is shown in Figure 4.17(d). In this example, the two ADD instructions with identical operands and complementary predicates are merged. The algorithm for instruction merging is described in Figure 4.20.

```
instruction_merging_1()
{
        for each instruction, op(x), in the hyperblock {
                if all the following conditions are true:
                        1. op(x) can be promoted with type 1 promotion
                        2. op(y) can be promoted with type 1 promotion
                        3. op(x) is identical operation as op(y)
                        4. pred(x) is complement form of pred(y)
                        5. same definition reaches source register in op(x) and op(y)
                        6. op(x) is placed before op(y)
                then do:
                        promote op(x)
                        delete op(y)
        }
}
```

Figure 4.20: Type 1 instruction merging algorithm.

Another merging optimization can be applied if there are two identical instructions with one dominating the other and there are no other operations between these two instructions which can modify the destination registers used by this pair of instructions. The optimization function is shown in Figure 4.21.

4.5 Predicate Hierarchy Graph

An important issue in the hyperblock optimization and scheduling is the reconstruction of the control flow information from the hyperblock structure. The control flow information can be used to determine if two predicated instructions can ever be executed when the hyperblock is entered each time. One method is to build a predicate hierarchy graph (PHG) similar to the RK graph. The graph is a representation of the Boolean equations under which all predicates are defined. The PHG is an acyclic graph and contains two types of nodes: the register and condition. Figure 4.22(a) shows an example of

```
instruction_merging_2()
{
        for each instruction, op(x), in the hyperblock {
                if all the following conditions are true:
                        1. op(x) dominates op(y)
                        2. op(x) is identical operation as op(y)
                        3. same definition reaches source register in op(x) and op(y)
                        4. dest(x) ≠ dest(z) in {op(z), z = x + 1 ... y − 1}
                then do:
                        delete op(y)
        }
}
```

Figure 4.21: Type 2 instruction merging algorithm.

a control flow graph. C1 to C6 are the conditions in which the flow edge is taken. P1 to P4 are the predicate registers used for this example. Figure 4.23 lists the predicate define operations that are created when the basic blocks from the above control flow graph are if-converted. Figure 4.22(b) shows the PHG graph reconstructed from the predicate definitions in Figure 4.23.

The control flow information can be determined from the PHG graph. To determine if two predicates can ever be true at the same time, the PHG is used to find the Boolean expression for each predicate. If ANDing the two Boolean expressions can be simplified to 0, the corresponding predicates are never both true for each entry into the hyperblock.

For example, with Figure 4.23, p3 and p4 can both be true each time the hyperblock is entered. To determine this, the Boolean equations for p3 and p4 are examined using the PHG from Figure 4.22(b). From the graph, $p3 = (c1 \cdot c2_bar) + (c1_bar \cdot c3)$ and $p4 = (c3 \cdot c1_bar) + (c1)$. These two predicates may be true at the same time if $p3 \cdot p4$ cannot be simplified to 0. In this case, $p3 \cdot p4 = ((c1 \cdot c2_bar) + (c1_bar \cdot c3)) \cdot ((c3 \cdot c1_bar) + (c1))$

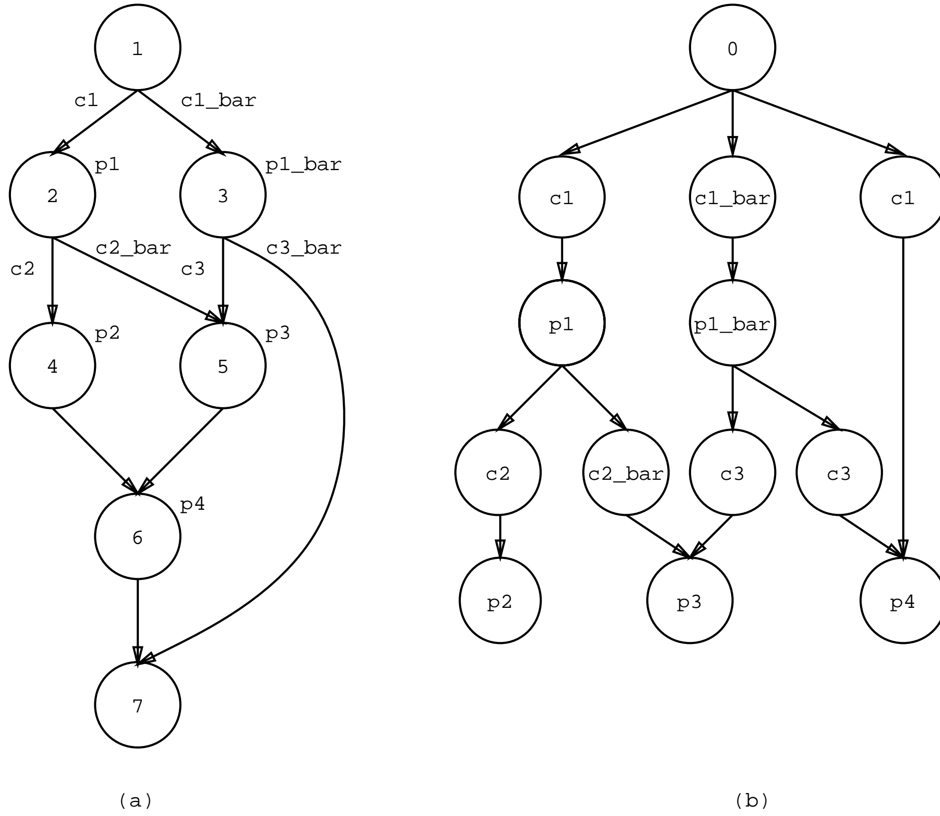


Figure 4.22: (a) Control flow graph and the (b) PHG graph.

	pred_clear p4	
1:	pred_set p1 $\leftarrow$ c1	
	pred_set p4 $\leftarrow$ c1	
2:	pred_set p2 $\leftarrow$ c2	(p1)
	pred_set p3 $\leftarrow$ c2_bar	(p1)
3:	pred_set p3 $\leftarrow$ c3	(p1_bar)
	pred_set p4 $\leftarrow$ c3	(p1_bar)

Figure 4.23: Predicate define operations added to the blocks.

```
is_control_path(p1, p2)
{
        for each term of the SOP of p1, term_i {
                for each term of the SOP of p2, term_j {
                        if ((term_i · term_j) can not be simplified to 0)
                                return false
                }
        }
        return true
}
```

Figure 4.24: Algorithm for finding control path.

cannot be simplified to 0. Thus, any two instructions predicated with p3 and p4 are on a control path.

Another example, using Figure 4.23 again, p2 and p1_bar cannot both be true at the same time. Using the PHG, $p2 = (c2 \cdot c1)$ and $p1_bar = c1_bar$. Now, $p2 \cdot p1_bar = (c2 \cdot c1) \cdot c1_bar$ which can be simplified to 0. Thus, any two instructions predicated with p2 and p1_bar are independent. An algorithm to check for the control flow information between two predicate registers is shown in Figure 4.24.

4.6 Conventional Optimizations

Other optimizations for nonpredicated code can also be extended to predicated code in the hyperblock using the control flow information. Most optimization works by increasing the instruction level parallelism within the program. Loop unrolling can be used to overlap different iterations of frequently executed loops to increase the performance of the program [16]. Register renaming is used to remove the anti, output, and control dependencies between instructions. Classical optimizations [17], [13] can also be applied to the hyperblock.

Optimization on a hyperblock is more effective than traditional global optimization techniques, because traditional techniques must be conservative and consider all control paths between basic blocks. Superblock optimization considers only a single path at a time. But, many of the early branch exits in a superblock can potentially limit a large portion of the performance improvement. The hyperblock, on the other hand, can optimize across multiple numbers of paths. Thus, there are more potential performance improvements using the hyperblocks.

4.7 Scheduling

After the hyperblock is optimized, the code must be scheduled according to the architecture specifications. The hyperblock scheduler is based on superblock scheduler. The code scheduling consists of two steps: dependence graph construction and list scheduling. The dependence graph is built by checking for flow, anti, output, and control dependencies between two instructions in each hyperblock. Dependencies between predicated instructions within a hyperblock can be removed if the predicate registers associated with the pair of instructions cannot both be true when the hyperblock is entered. Some control dependencies may be eliminated with available hardware support for speculative execution. For example, the dependencies between a branch and an instruction following the branch can be eliminated, if the hardware consists of nontrapping instructions and the destination of the instruction is not used before it is redefined when the branch is taken. After the appropriate dependencies are removed, list scheduling is used to schedule the code appropriately according to a set of parameters including instruction latencies, resource constraints, and the reduced dependence graph.

5. PERFORMANCE EVALUATION

The performance of the hyperblock formation and optimization techniques has been evaluated on a DEC 3100 workstation using the code generator for the MIPS R2000 processor [18]. The code generator is modified to convert the predicated intermediate code to the MIPS assembly code. Table 5.1 shows the benchmark programs used for the evaluation. Most benchmarks are commonly used Unix programs. The size column gives the length of each benchmark measured in numbers of lines of code. A set of input data has been selected to profile every benchmark. Table 5.2 gives a brief description of the characteristics of the selected input data sets.

5.1 Methodology

The hyperblock technique described in the previous chapter has been implemented in the IMPACT-I compiler. The compiler is a prototype optimizing compiler designed to generate efficient code for VLIW and superscalar processors. The compiler utilizes a machine description file to generate code for a parameterized superscalar processor.

The microarchitecture model uses RISC-like assembly language similar to the MIPS R2000 instruction set. Further, the architecture is assumed to have a CRAY-1 style

Table 5.1: Benchmarks.

name	*size*	*description*
cccp	4787	GNU C preprocessor
cmp	141	compare files
compress	1514	compress files
eqntott	3461	Boolean minimization
espresso	6722	Boolean minimization
grep	464	string search
lex	3316	lexical analysis program generator
li	7747	Lisp interpreter
qsort	136	quick sort program
tbl	2817	format tables for troff
wc	120	word count
yacc	2303	parsing program generator

Table 5.2: Input data for profiling.

name	*input*	*description*
cccp	20	C source files (100 - 5000 lines)
cmp	20	similar / different files
compress	20	C source files (100 - 5000 lines)
eqntott	5	Boolean equations
espresso	20	Boolean functions (original espresso benchmarks)
grep	20	C source files (100 - 5000 lines) with various search strings
lex	5	lexers for C, Lisp, Pascal, awk, and pic
li	5	Gabriel benchmarks
qsort	1	sort 1024 numbers
tbl	20	troff files (100 - 4000) lines
wc	20	C source files (100 - 5000) lines
yacc	10	grammars for C, Pascal, pic, eqn, awk, etc.

Table 5.3: Instruction latencies.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	3
Int multiply	3	FP conversion	3
Int divide	10	FP multiply	3
branch	1 / 1 slot	FP divide	10
memory load	2	memory store	1

interlocking and deterministic instruction latencies (Table 5.3). A set of nontrapping instructions is assumed to be provided by the architecture for speculative execution. The architecture also contains an infinite number of predicate registers to support predicated execution. The performance result derived from execution-driven simulation is assumed to have a 100% cache hit rate.

5.2 Results

In this section the performance of the hyperblock technique is examined for VLIW and superscalar processors with issue rates 2, 4 and 8. The issue rate is the maximum number of instructions the processor can fetch and issue per cycle. The speedup is compared to that for the base machine with issue rate of 1 using basic block scheduling. No limitation has been placed on the combination of instructions that can be issued in the same cycle.

Figure 5.1 shows the effectiveness of the partial predication by selectively excluding unimportant basic blocks from hyperblock formation. Predication-specific optimizations, promotion and merging, are not used for this comparison. Across all benchmarks, partial predication (PP) outperforms both basic block scheduling (O) and full inner-loop predication (IP). For some benchmarks, *cccp* and *compress*, IP does not do as well as basic block scheduling at any issue rates. Also, full inner loop predication of *compress,*

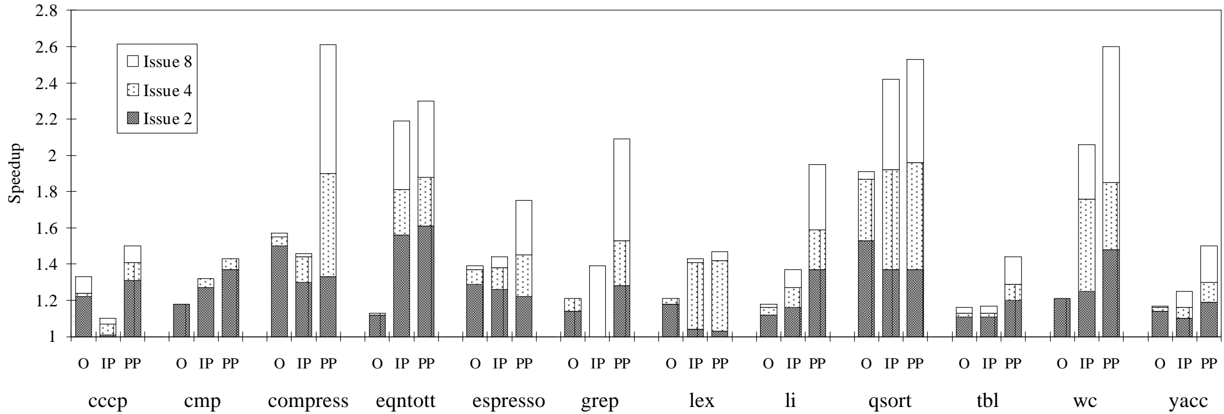


Figure 5.1: Performance comparison of basic block scheduling (O), innermost loop predication (IP), and partial predication (PP).

espresso, grep, lex, qsort, and *yacc* does not perform as well as the basic block scheme of the same benchmarks at lower issue rates. This can be attributed to the fact that IP combines all basic blocks in a loop together which fills up the available instruction slots. On the other hand, when the issue rate is increased sufficiently, this problem is alleviated.

Figure 5.2 represents the performance with and without predicate-specific optimizations. The graph indicates that performance improvement is possible by speculatively executing instructions using the instruction promotion optimization technique. An average of 11% speedup is observed for partial predication with speculative execution (PO) over partial predication (PP), and approximately 6% speedup is noted for inner-loop predication with speculative execution (IO) over inner-loop predication (IP). Thus, speculative execution is an important source of performance improvement regardless of the base scheme.

A performance comparison of the basic block, superblock, and hyperblock schemes is illustrated in Figure 5.3. At issue 8, hyperblock technique (PO) outperforms both the superblock (T1) and basic block (O) techniques in all benchmarks except *grep*. At lower

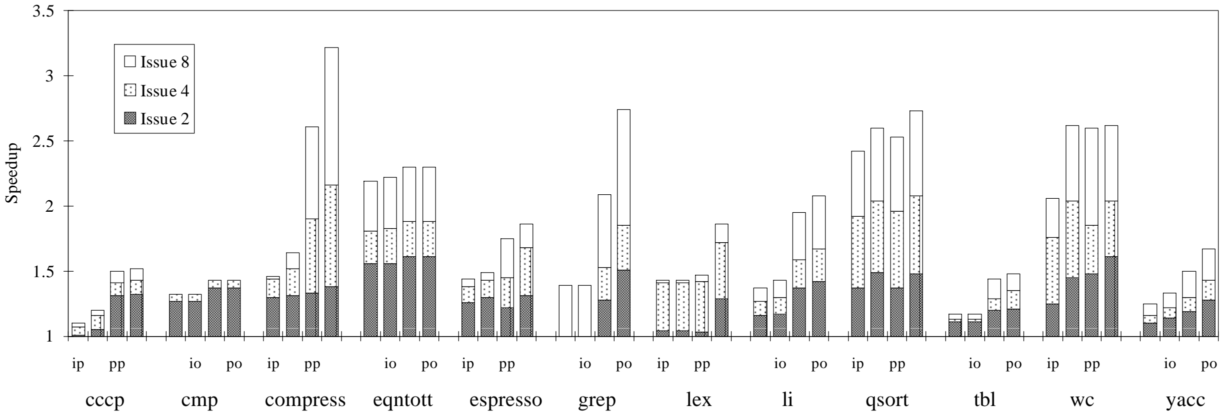


Figure 5.2: Performance comparison of inner loop predication (IP), inner loop predication with speculative execution (IO), partial predication (PP), and partial predication with speculative execution (PO).

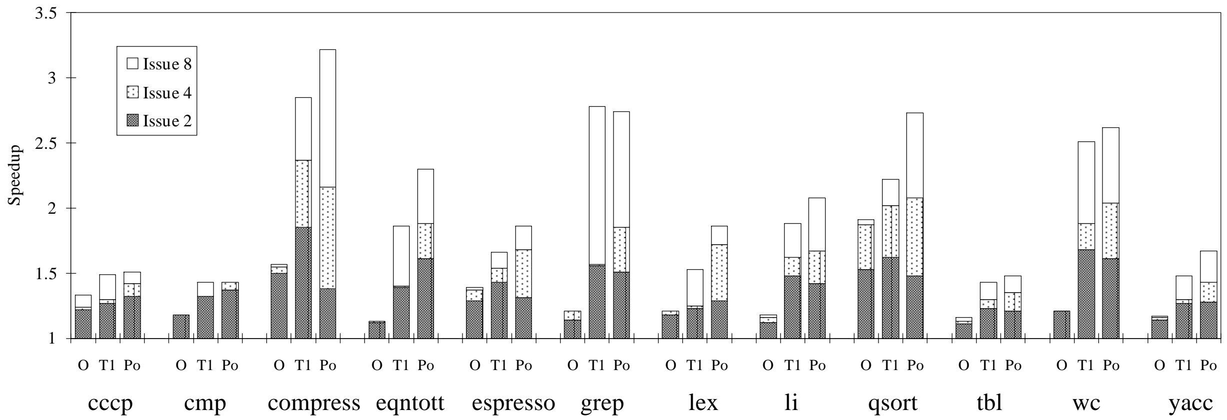


Figure 5.3: Performance comparison of basic block (O), superblock (T1), and hyperblock (PO).

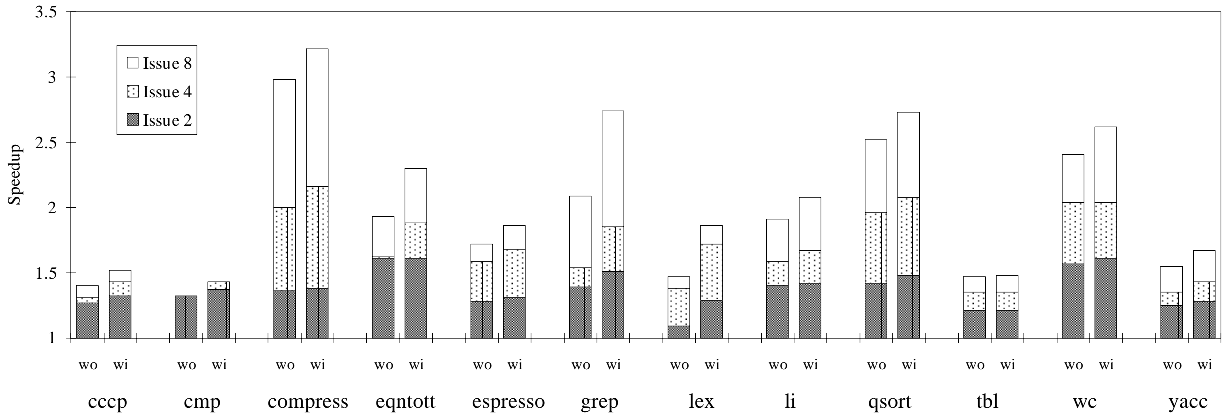


Figure 5.4: Performance of predicated execution with and without predicate register forwarding.

issue rates, however, the performance of the superblock is somewhat better than for the hyperblock. This can be expected, because the hyperblock technique examines multiple paths. Thus, there are not enough instruction slots available at the lower issue rates. When the resources are greater, hyperblock provides better performance improvement since independent instructions in the hyperblock are able to utilize more of the resources concurrently than other schemes.

The effectiveness of the predicate register forwarding scheme discussed in Chapter 2 is shown in Figure 5.4. The figure indicates that predicate register forwarding hardware can be used to increase the performance of architecture with predicated execution. Average speedups of about 12% at issue 8, 9% at issue 4, and 3% at issue 2 are observed by comparing the same machine architecture with and without the forwarding hardware support. This technique is more effective at higher issue rates because more instructions are able to use the forwarding mechanism and, thus, can be scheduled earlier and give higher performance improvement.

Another source of measurement is the code size expansion shown in Figure 5.5. This figure represents the size of code expansion normalized to program size of the original

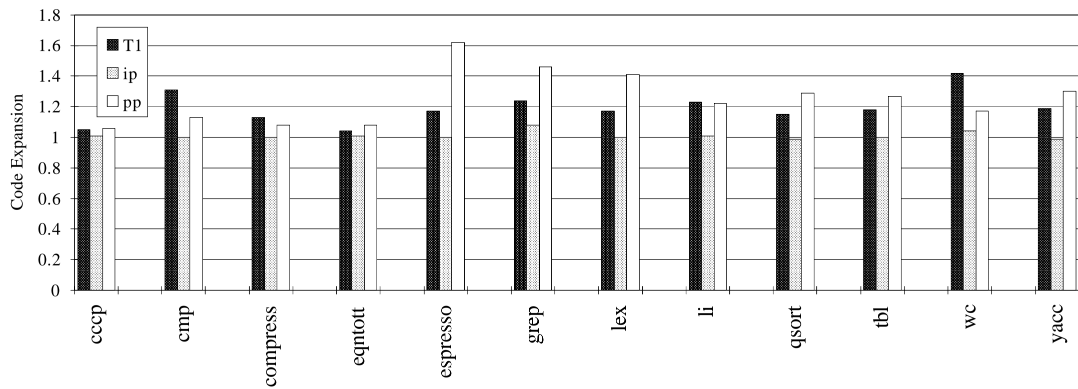


Figure 5.5: Code expansion of the different schemes.

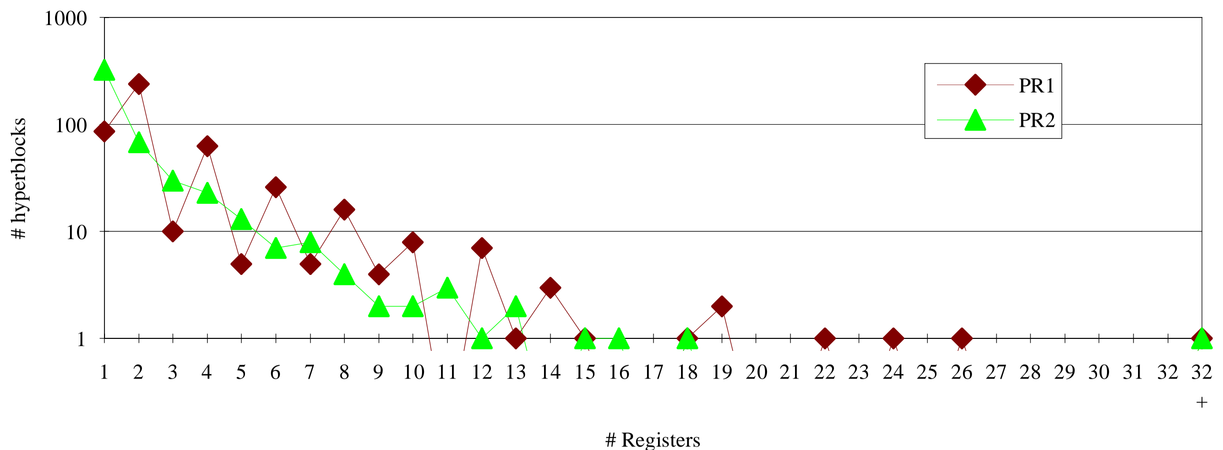


Figure 5.6: Predicate register usage.

optimized basic block code. Inner-loop predication gives the least amount of code expansion. In some cases it actually reduces the code size. This can be explained by the fact that IP is able to remove many forward branch instructions. Superblock and hyperblock, on the other hand, cause some code expansion ranging from about 110% to 160%. The main source of hyperblock expansion is the node splitting technique. For example, the three benchmarks with the highest code expansion, *espresso, grep,* and *lex*, contain a significant amount of code duplication due mostly to node splitting.

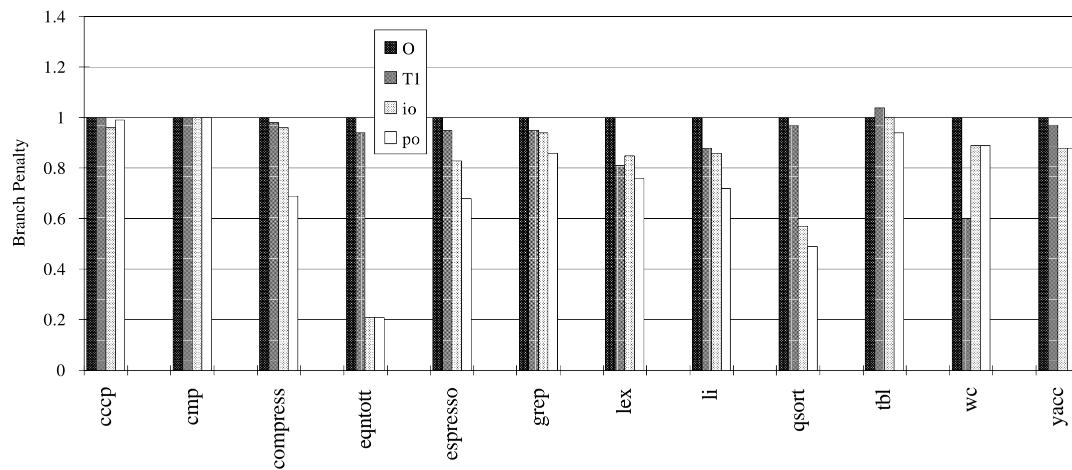


Figure 5.7: Normalized branch penalty of basic block, superblock, and hyperblock.

The predicate register usage combined from all 12 benchmarks is shown in Figure 5.6. The graph gives the number of hyperblocks versus the number of predicate register usages after register allocation. PR1 represents a scheme without complementary predicate registers similar to the one used in Cydra 5 [5]. PR2 uses the predicate register organization with the complementary forms discussed in Chapter 2. From the figure, it can be seen that that about 16-32 predicate registers satisfy the requirement of all benchmarks used in this study.

The average number of predicate registers used in a hyperblock is about 3.5 with the PR1 scheme and approximately 2 with the PR2 scheme. However, each register in the PR2 scheme is equivalent to 2 registers in the PR1 scheme (true and false form), so the PR2 scheme actually uses an average of 0.5 more predicate registers in each hyperblock. On the other hand, the complementary register scheme can be used to reduce the excess predicate set instructions which can improves the performance.

Another source of measurement is the branch penalties of the different schemes. The branch penalty is defined to be the wasted cycles introduced when a branch instruction is mispredicted in a pipelined processor. In this study, one clock cycle of penalty is assumed

for each mispredicted branch. Also, the compiler predicts the direction of all branches using profile information.

The branch penalties of the basic block (O), superblock (T1), hyperblock with inner loop predication (IP), and hyperblock with partial predication (PO) schemes are shown in Figure 5.7. The penalty cycle time is normalized to the branch penalty cycle time of the basic block scheme. The figure shows that superblock reduces the branch penalty on the average by 10%. Hyperblock schemes, on the other hand, are able to reduce the branch penalty by an average of about 25% with reductions as high as 80% for some benchmarks. The branch penalty is reduced significantly in the hyperblock scheme, because it is able to remove much of the frequently executed branches in the programs by if-conversion. From this figure, predicated execution with the hyperblock scheme is shown to be very effective for reducing the branch penalty, especially on machines with a high penalty for a mispredicted branch target.

6. CONCLUSIONS

This thesis discusses effective compiler techniques for superscalar processors with predicated execution support. A structure called hyperblock is used to represent each group of predicated basic blocks. The hyperblock representation allows a conventional optimizer and scheduler to support predicated instructions with minor software modifications.

Many algorithms are described here to support hyperblock formation and optimization. Block selection, tail duplication, loop peeling, node splitting, and if-conversion algorithms are used for the formation of hyperblocks. Hyperblock-specific optimizations include instruction promotion and merging. Promotion allows instructions to be speculatively executed and removes the flow dependency between the instruction and its predicate define operation. This creates more opportunities for compact scheduled code and improves the performance of the program. Instruction merging is used to reduce redundant instructions across different paths of control.

The profile information and instruction characteristics are used to improve the if-conversion technique by selecting basic blocks that are more beneficial for predication. Experiments have shown that these techniques can improve the program performance

significantly. More detailed work to find a good heuristic function for block selection can help to further improve the performance of predicated execution.

REFERENCES

[1] R. A. Towle, "Control and data dependence for program transformations," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1976.

[2] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "Program dependence graph and its use in optimization," *ACM Transactions on Programming Languanges*, pp. 319–348, July 1987.

[3] J. C. Park and M. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, HP Laboratories, Palo Alto, CA, May 1991.

[4] 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*, pp. 177–189, January 1983.

[5] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, pp. 12–35, January 1989.

[6] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Proceedings of Super Computing 90*, November 1990.

[7] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[8] J. C. Dehnert, P. Y. T. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 26–38, May 1989.

[9] P. P. Chang and W. W. Hwu, "The lcode language and its environments," Tech. Rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, April 1991.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture : A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

[11] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," Tech. Rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[12] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[13] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[14] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and superscalar processors," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[15] E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," *Communications of the ACM*, pp. 96–103, February 1979.

[16] S. Weiss and J. E. Smith, "A study of scalar compilation techniques for pipelined supercomputers," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 105–109, April 1989.

[17] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[18] W. Y. Chen, "An optimizing compiler code generator: A platform for risc performance analysis," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.