Effective Compiler Support for Predicated Execution Using the Hyperblock

Scott A. Mahlke David C. Lin* William Y. Chen Richard E. Hank Roger A. Bringmann

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

Abstract

Predicated execution is an effective technique for dealing with conditional branches in application programs. However, there are several problems associated with conventional compiler support for predicated execution. First, all paths of control are combined into a single path regardless of their execution frequency and size with conventional if-conversion techniques. Second, speculative execution is difficult to combine with predicated execution. In this paper, we propose the use of a new structure, referred to as the hyperblock, to overcome these problems. The hyperblock is an efficient structure to utilize predicated execution for both compile-time optimization and scheduling. Preliminary experimental results show that the hyperblock is highly effective for a wide range of superscalar and VLIW processors.

1 Introduction

Superscalar and VLIW processors can potentially provide large performance improvements over their scalar predecessors by providing multiple data paths and function units. In order to effectively utilize the resources, superscalar and VLIW compilers must expose increasing amounts of instruction level parallelism (ILP). Typically, global optimization and scheduling techniques are utilized by the compiler to find sufficient ILP. A common problem all global optimization and scheduling strategies must resolve is conditional branches in the target application. 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 source operand, referred to as the predicate. When the predicate has value T, the instruction is executed normally and when the predicate has value F, the instruction is treated as a *no_op*. With predicated execution support provided in the architecture, the compiler can eliminate many of the conditional branches in an application.

The process of eliminating conditional branches from a program to utilize predicated execution support is referred to as *if-conversion* [1] [2] [3]. If-conversion was initially proposed to assist automatic vectorization techniques for loops with conditional branches. If-conversion basically replaces conditional branches in the code with comparison instructions which set a predicate. Instructions control dependent on the branch are then converted to predicated instructions dependent on the value of the corresponding predicate. In this manner, control dependences are converted to data dependences in the code. If-conversion can eliminate all non-loop backward branches from a program.

Predicated execution support has been used effectively for scheduling both numeric and non-numeric applications. For numeric code, overlapping the execution of multiple loop iterations using software pipeline scheduling can achieve high-performance on superscalar and VLIW processors [4] [5] [6]. With the ability to remove branches with predicated execution support, more compact schedules and reduced code expansion are achieved with software pipelining. Software pipelining taking advantage of predicated execution support is productized in the Cydra 5 compiler [7] [8]. For non-numeric applications, decision tree scheduling utilizes guarded instructions to achieve large performance improvements on deeply pipelined processors as well as multiple-instruction-issue processors [9]. Guarded instructions allow concurrent execution along multiple paths of control and execution of instructions before the branches they depend on may be resolved.

There are two problems, though, associated with utilizing conventional compiler support for predicated execution. First, if-conversion combines all execution paths in a region (typically an inner loop body) into a single block. Therefore, instructions from the entire region must be examined each time a particular path through the region is entered. When all execution paths are approximately the same size and have the same frequency, this method is very effective. However, the size and frequency of different execution paths typically varies in an inner loop. Infrequently executed paths and execution paths with comparatively larger number of instructions often limit performance of the resultant predicated block. Also, execution paths with subroutine calls or unresolvable memory accesses can restrict optimization and scheduling within the predicated block.

The second problem is that speculative execution does not fit in conveniently with predicated execution. Speculative or eager execution refers to the execution of an instruction before it is certain its execution is required. With predicated instructions, speculative execution refers to the execution of an instruction before its predicate is calculated. Speculative execution is an important source of ILP for superscalar and VLIW processors by allowing long latency instructions to be initiated much earlier in the schedule.

*David Lin is now with Amdahl Corporation, Sunnyvale, CA.

In this paper, we propose the use of a structure, referred to as the *hyperblock*, to overcome these two problems. 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. Hyperblocks are formed using a modified version of if-conversion. Basic blocks are included in a hyperblock based on their execution frequency, size, and instruction characteristics. Speculative execution is provided by performing predicate promotion within a hyperblock. Superscalar optimization, scheduling, and register allocation may also be effectively applied to the resultant hyperblocks.

The remainder of this paper consists of four sections. In Section 2, the architecture support we utilize for predicated execution is discussed. Section 3 presents the hyperblock and its associated transformations. In Section 4, a preliminary evaluation on the effectiveness of the hyperblock is given. Finally, some concluding remarks are offered in Section 5.

2 Support for Predicated Execution

An architecture supporting predicated execution must be able to conditionally nullify the side effects of selected instructions. The condition for nullification, the predicate, is stored in a predicate register file and is specified via an additional source operand added to each instruction. The content of the specified predicate register is used to squash the instruction within the processor pipeline. The architecture chosen for modification to allow predicated execution, the IMPACT architecture model, is a statically scheduled, multiple instruction issue machine supported by the IMPACT-I compiler [10]. The IMPACT architecture model modifications for predicated execution are based upon that of the Cydra 5 system [7]. Our proposed architectural modifications serve to reduce the dependence chain for setting predicates and to increase the number of instructions allowed to modify the predicate register file. This section will present the implementation of predicated execution in the Cydra 5, and discuss the implications that the proposed modifications to the IMPACT architecture model will have on the architecture itself, instruction set, and instruction scheduling.

2.1 Support in the Cydra 5 System

The Cydra 5 system is a VLIW, multiprocessor system utilizing a directed-dataflow architecture. Each Cydra 5 instruction word contains 7 operations, each of which may be individually predicated. An additional source operand added to each operation specifies a predicate located within the predicate register file. The predicate register file is an array of 128 boolean (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. Essentially, operations whose predicates are '0' are converted to *no_ops* prior to entering the execution stage of the pipeline. The predicate specified by an operation must thus be known by the time the operation leaves the operand fetch stage.

(a)

```
for (i=0; i<100; i++)
    if (A[i] ≤ 50)
        j = j+2;
    else
        j = j+1;
```

(b)

```
    mov  r1,0
    mov  r2,0
    ld   r3,addr(A)
L1:
    ld   r4,mem(r3+r2)
    bgt  r4,50,L2
    add  r5,r5,2
    jump L3
L2:
    add  r5,r5,1
L3:
    add  r1,r1,1
    add  r2,r2,4
    blt  r1,100,L1
```

(c)

```
    mov  r1,0
    mov  r2,0
    ld   r3,addr(A)
L1:
    ld   r4,mem(r3+r2)
    gt   r6,r4,50
    stuff p1,r6
    stuff_bar p2,r6
    add  r5,r5,2 if p2
    add  r5,r5,1 if p1
    add  r1,r1,1
    add  r2,r2,4
    blt  r1,100,L1
```

Figure 1: Example of if-then-else predication, (a) source code segment, (b) assembly code segment, (c) assembly code segment after predication.

The content of a predicate register may only be modified by one of 3 operations: *stuff*, *stuff_bar*, or *brtop*. The *stuff* operation takes as operands a destination predicate register and a boolean value, as well as, a source predicate register as described above. The boolean value is typically produced using a comparison operation. If the predicate value is '1', the destination predicate register is assigned the boolean value, otherwise the operation is squashed. The *stuff_bar* operation functions in the same manner, except the destination predicate register is set to the inverse of the boolean value when the predicate value is '1'. The *brtop* operation is used for loop control and sets the predicate controlling the next iteration by comparing the contents of a loop iteration counter to the loop bound.

Figures 1a and 1b show a simple for-loop containing an if-then-else conditional and its corresponding assembly code. To set the mutually exclusive predicates for the different execution paths shown in this example, requires 3 instructions, as shown in Figure 1c. First, a comparison must be performed, followed by a *stuff* to set the predicate register for the true path (predicated on **p1** in Figure 1c) and a *stuff_bar* to set the predicate register for the false path (predicated on **p2** in Figure 1c). This results in a minimum dependence distance of 2 from the comparison to the first possible reference of the predicate being set.

2.2 Support in the IMPACT Architecture

The proposed modifications to the Cydra 5 method for the IMPACT architecture model seek to reduce the number of instructions[1] required to set a predicate and reduce the dependence length from the setting of a predicate to its first use. Figure 2 shows a basic model of a superscalar processor

[1] In this context, instruction refers to a superscalar instruction, as opposed to a VLIW instruction as in Cydra 5.

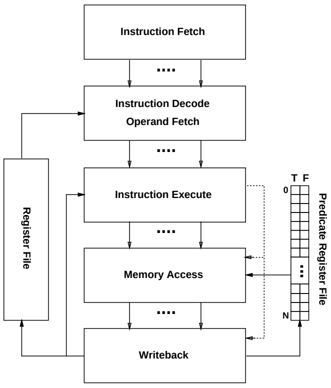

Figure 2: Pipeline model with predicated execution.

pipeline with the addition of a predicate register file. The fourth stage of the pipeline, **Memory Access**, in addition to initiating memory access, is used to access the predicate register specified by each instruction. This is passed to the **Writeback** stage which determines if the result of the instruction is to be written to either register file. Thus, rather than squashing an instruction prior to execution as in the Cydra 5 system, an instruction is not squashed until the **Writeback** stage. The dashed arrow in Figure 2 will be described later in this section.

The proposed predicate register file is an Nx2 array of boolean (1-bit) registers. For each of the N possible predicates there is a bit to hold the true value and a bit to hold the false value of that predicate. Each pair of bits associated with a predicate register may take on one of three combinations: false/false, true/false, or false/true. The false/false combination is necessary for nested conditionals in which instructions from both sides of a branch may not require execution. An instruction is then able to specify whether it is to be predicated on the true value of the predicate or its false value. This requires the addition of $log_2(N) + 1$ bits to each instruction.

In the IMPACT model, predicate registers may be modified by a number of instructions. Both bits of a specified predicate register may be simultaneously set to '0' by a *pred_clear* instruction. New instructions for integer, unsigned, float, and double comparison are added, whose destination register is a register within the predicate register file. The T field of the destination predicate register is set to the result of the compare and the F field is set to the inverse result of the compare. This allows the setting of mutually exclusive predicates for if-then-else conditionals in one instruction. By performing the comparison and setting of both predicates in one instruction, the previous code example reduces to that shown in Figure 3. The true path of the comparison is predicated on **p1_T** and the false path is predicated on **p1_F**. In addition, *pred_ld* and *pred_st* instructions are provided to allow the register allocator to save and restore individual predicate registers around a function call. In all, 25 instructions were added to the IMPACT architecture to support predicated execution.

The ability of comparison instructions to set mutually exclusive predicates in the same cycle coupled with the fact

```
    mov   r1,0
    mov   r2,0
    ld    r3,addr(A)
L1:
    ld    r4,mem(r3+r2)
    pred_gt  p1,r4,50
    add   r5,r5,2 if p1_F
    add   r5,r5,1 if p1_T
    add   r1,r1,1
    add   r2,r2,4
    blt   r1,100,L1
```

Figure 3: Example of if-then-else predication in the IMPACT model.

that instructions are not squashed until the **Writeback** stage, reduces the dependence distance from comparison to first use from 2 to 1. By adding additional hardware to the **Instruction Execute** stage that allows the result of a predicate comparison operation to be forwarded to the **Memory Access** and **Writeback** stages (the dashed arrow in Figure 2), the dependence distance is reducible to 0. This may be accomplished by scheduling a predicate comparison operation and an operation referencing the predicate defined by the comparison in the same cycle. Note that throughout this section stuff and comparison instructions are assumed to take one cycle to execute. In general, for stuffs taking i cycles and comparisons taking j cycles, the dependence distance is reduced from $i + j$ to $j - 1$ by combining the IMPACT predicate model with predicate forwarding logic in the pipeline.

3 The Hyperblock

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 enter the hyperblock only at this point. The motivation behind hyperblocks is to group many basic blocks from different control flow paths into a single manageable block for compiler optimization and scheduling. However, all basic blocks to which control may flow are not included in the hyperblock. Rather, some basic blocks are systematically excluded from the hyperblock to allow more effective optimization and scheduling of those basic blocks in the hyperblock.

A similar structure to the hyperblock is the superblock. A superblock is a block of instructions such that control may only enter from the top, but may exit from one or more locations [11]. But unlike the hyperblock, the instructions within each superblock are not predicated instructions. Thus, a superblock contains only instructions from one path of control. Hyperblocks, on the other hand, combine basic blocks from multiple paths of control. Thus, for programs without heavily biased branches, hyperblocks provide a more flexible framework for compile-time transformations.

In this section, hyperblock block selection, hyperblock formation, generation of control flow information within hyperblocks, hyperblock-specific optimization, and extensions of conventional compiler techniques to hyperblocks are discussed.

3.1 Hyperblock Block Selection

The first step of hyperblock formation is deciding which basic blocks in a region to include in the hyperblock. The region of blocks to choose from typically is the the body of an inner most loop. However, other regions, including non-loop code with conditionals and outer loops containing nested loops, may be chosen. Conventional techniques for if-conversion predicate all blocks within a single-loop nest region together [13]. For hyperblocks, though, only a subset of the blocks are chosen to improve the effectiveness of compiler transformations. Also, in programs with many possible paths of execution, combining all paths into a single predicated block may produce an overall loss of performance due to limited machine resources (fetch units or function units).

To form hyperblocks, three features of each basic block in a region are examined, execution frequency, size, and instruction characteristics. Execution frequency is used to exclude paths of control which are not often executed. Removing infrequent paths reduces optimization and scheduling constraints for the frequent paths. The second feature is basic block size. Larger basic blocks should be given less priority for inclusion than smaller blocks. Larger blocks utilize many machine resources and thus may reduce the performance of the control paths through smaller blocks. Finally, the characteristics of the instructions in the basic block are considered for inclusion in the hyperblock. Basic blocks with hazardous instructions, such as procedure calls and unresolvable memory accesses, are given less priority for inclusion. Typically hazardous instructions reduce the effectiveness of optimization and scheduling for all instructions in the hyperblock.

A heuristic function which considers all three issues is shown below.

$$BSV_i = (K \times \frac{weight\_bb_i}{size\_bb_i} \times \frac{size\_main\_path_1}{weight\_main\_path_1} \times bb\_char_i)$$

The Block Selection Value (BSV) is calculated for each basic block considered for inclusion in the hyperblock. The weight and size of each basic block is normalized against that of the "main path". The main path is the most likely executed control path through the region of blocks considered for inclusion in the hyperblock. The hyperblock initially contains only blocks along the main path. The variable bb_char_i is the characteristic value of each basic block. The maximum value of bb_char_i is 1. Blocks containing hazardous instructions have bb_char_i less than 1. The variable K is a machine dependent constant to represent the issue rate of the processor. Processors with more resources can execute more instructions concurrently, and therefore are likely to take advantage of larger hyperblocks.

An example to illustrate hyperblock block selection is shown in Figure 4a. This example shows a weighted control flow graph for a program loop segment. The numbers associated with each node and arc represent the dynamic frequency each basic block is entered and each control transfer is traversed, respectively. For simplicity, this example considers only block execution frequency as the criterion for hyperblock block selection. The main path in this example is blocks A, B, D, and E. Block C is also executed frequently, so it is selected as part of the hyperblock. However, Block F is not executed frequently, and is excluded from the hyperblock.

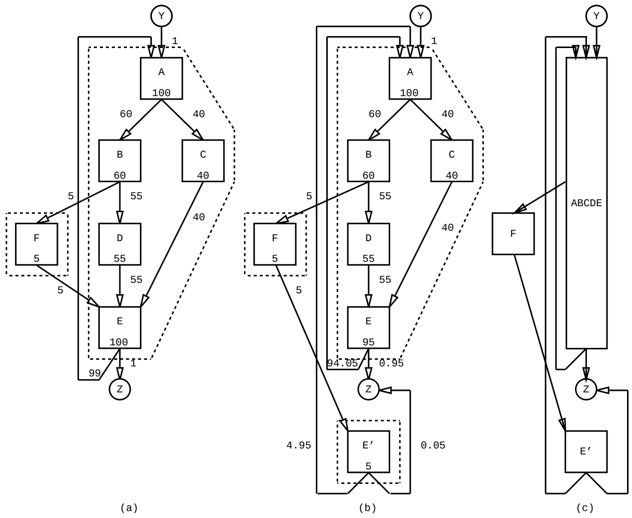

Figure 4: An example of hyperblock formation, (a) after block selection, (b) after tail duplication, (c) after if-conversion.

3.2 Hyperblock Formation

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

Condition 1 : There exist no incoming control flow arcs from outside basic blocks to the selected blocks other than to the entry block.

Condition 2 : There exist no nested inner loops inside the selected blocks.

These conditions ensure that the hyperblock is entered only from the top, and the instructions in a hyperblock are executed at most once before the hyperblock is exited. Tail duplication and loop peeling are used to transform the basic blocks selected for a hyperblock to meet the conditions. After the group of basic blocks satisfies the conditions, they may be transformed using the if-conversion algorithm described later in this section.

Tail Duplication. Tail duplication is used to remove control flow entry points into the selected blocks (other than the entry block) from blocks not selected for inclusion in the hyperblock. In order to remove this control flow, blocks which may be entered from outside the hyperblock are replicated. A tail duplication algorithm transforms the control flow graph by first marking all the flow arcs that violate Condition 1. Then all selected blocks with a direct or indirect predecessor not in the selected set of blocks are marked. Finally, all the marked blocks are duplicated and the marked flow arcs are adjusted to transfer control to the corresponding duplicate blocks. To reduce code expansion, blocks are duplicated at most one time by keeping track of the current set of duplicated blocks.

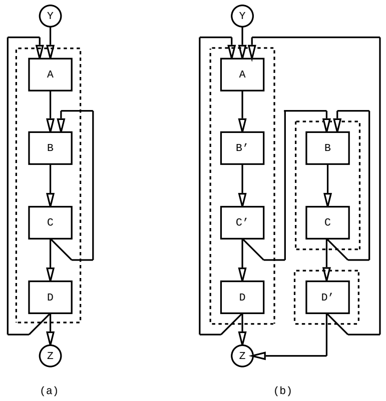

Figure 5: An example of loop peeling, (a) original flow graph, (b) after peeling one iteration of the inner loop and tail duplication.

An example to illustrate tail duplication is shown in Figure 4b. In this example block E contains a control flow entry point from a block not selected for the hyperblock (block F). Therefore, block E is duplicated and the control flow arc from F to E is adjusted to the duplicated block E. The selected blocks after tail duplication are only entered from outside blocks through the entry block, therefore Condition 1 is satisfied.

Loop Peeling. For loop nests with inner loops that iterate only a small number of times, efficient hyperblocks can be formed by including both outer and inner loops within a hyperblock. However, to satisfy Condition 2, inner loops contained within the selected blocks must be broken. Loop peeling is an efficient transformation to accomplish this task. Loop peeling unravels the first several iterations of a loop, creating a new set of code for each iteration. The peeled iterations are then included in the hyperblock, and the original loop body is excluded. A loop is peeled the average number of times it is expected to iterate based on execution profile information. The original loop body then serves to execute when the actual number of iterations exceeds the expected number.

An example illustrating loop peeling is shown in Figure 5. All the blocks have been selected for one hyperblock, however there is an inner loop consisting of blocks B and C. The inner loop is thus peeled to eliminate the backedge in the hyperblock. In this example, it assumed the loop executes an average of one iteration. Note also that tail duplication must be applied to duplicate block D after peeling is applied. After peeling and tail duplication (Figure 5b), the resultant hyperblock, blocks A, B$'$, C$'$, and D, satisfies Conditions 1 and 2.

Node Splitting. After tail duplication and loop peeling, node splitting may be applied to the set of selected blocks to eliminate dependences created by control path merges. At merge points, the execution time of all paths is typically dictated by that of the longest path. The goal of node splitting is to completely eliminate merge points with sufficient code duplication. Node splitting essentially duplicates all blocks subsequent to the merge point for each path of control entering the merge point. In this manner, the merge point is completely eliminated by creating a separate copy of the shared blocks for each path of control. Node splitting is especially effective for high-issue rate processors in control-intensive programs where control and data dependences limit the number of independent instructions.

A problem with node splitting is that it results in large amounts of code expansion. Excessive node splitting may limit performance within a hyperblock by causing many unnecessary instructions to be fetched and executed. Therefore, only selective node splitting should be performed by the compiler. A heuristic function for node splitting importance is shown below.

$$FSV_i = (K \times \frac{weight\_flow_i}{size\_flow_i} \times \frac{size\_main\_path_1}{weight\_main\_path_1} \times bb\_char_i)$$

The Flow Selection Value (FSV) is calculated for each control flow edge in the blocks selected for the hyperblock that contain two or more incoming edges, e.g., a merge point. $Weight\_flow_i$ is the execution frequency of the control flow edge. $Size\_flow_i$ is the number of instructions that are executed from the entry block to the point of the flow edge. The other parameters are the same parameters used in calculating the BSV. After the FSVs are computed, the node splitting algorithm proceeds by starting from the node with the largest differences between the FSVs associated with its incoming flow edges. Large differences among FSVs indicate highly unbalanced control flow paths. Thus, basic blocks with the largest difference should be split first. Node splitting continues until there are no more blocks with 2 or more incoming edges or no difference in FSVs above a certain threshold. Our node splitting algorithm also places an upper limit on the amount of node splitting applied to each hyperblock.

If-conversion. If-conversion replaces a set of basic blocks containing conditional control flow between the blocks with a single block of predicated instructions. Figure 4c illustrates a resultant flow graph after if-conversion is applied. In our current implementation, a variant of the RK if-conversion algorithm is utilized for hyperblock formation [3]. The RK algorithm first calculates control dependence information between all basic blocks selected for the hyperblock [12]. One predicate register is then assigned to all basic blocks with the same set of control dependences. Predicate register defining instructions are inserted into all basic blocks which are the source of the control dependences associated with a particular predicate. Next, dataflow analysis is used to determine predicates that may be used before being defined, and inserts resets (pred_clear instructions) to these predicates in the entry block of the hyperblock. Finally, conditional branches between basic blocks selected for the hyperblock are removed, and instructions are predicated based on their assigned predicate.

An example code segment illustrating hyperblock formation is shown in Figure 6. In the example, all blocks shown are selected for the hyperblock except block 5. A control entry point from block 5 to 7 is eliminated with tail duplication. If-conversion is then applied to the resultant set of selected blocks. A single predicate (p1) is required for this set of blocks. Instructions in block 2 are predicated on p1_true and instructions in block 3 are predicated on p1_false. In-

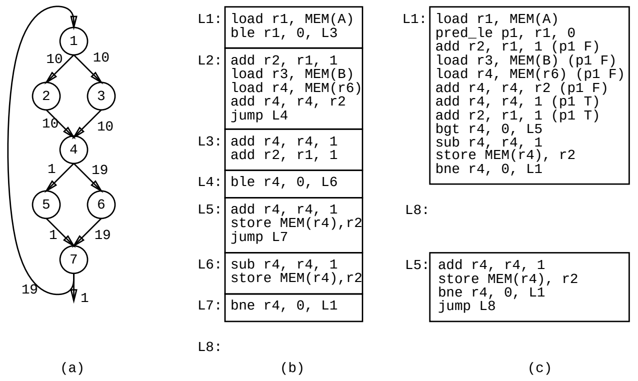

Figure 6: An example program segment for hyperblock formation, (a) original control flow graph, (b) original assembly code, (c) assembly code after hyperblock formation.

1	pred_clear	p4
2	pred_ne	p3,r2,0
3	pred_eq	p5,r2,0
4	pred_ne	p4,r0,0 if p3_T
5	pred_eq	p5,r0,0 if p3_T
6	mov	r2,r0 if p4_T
7	sub	r2,r2,r0 if p5_T
8	add	r1,r1,1

Figure 7: Example hyperblock.

structions in block 6 do not need to be predicated since block 6 is the only block in the hyperblock that may be reached from block 4. Note that hyperblock if-conversion does not remove branches associated with exits from the hyperblock. Only control transfers within the hyperblock are eliminated.

3.3 Generating Control Flow Information for a Hyperblock

Many compiler tools, including dependence analysis, data flow analysis, dominator analysis, and loop analysis, require control flow information in order to be applied. Control flow may easily be determined among basic basic blocks since instructions within a basic block are sequential and flow between basic blocks is determined by explicit branches. However, instructions within a hyperblock are not sequential, and thus require more complex analysis. For example, in Figure 7, instructions 6 and 7 demonstrate both an output dependence and a flow dependence if the predicate is not considered. These instructions, though, are predicated under mutually exclusive predicates, and therefore have no path of control between them. As a result, there is no dependence between these two instructions.

A predicate hierarchy graph (PHG) is a graphical representation of boolean equations for all of the predicates in a hyperblock. The PHG is composed of predicate and condition nodes. The '0' predicate node is used to represent the null predicate for instructions that are always executed. Conditions are added as children to their respective parent

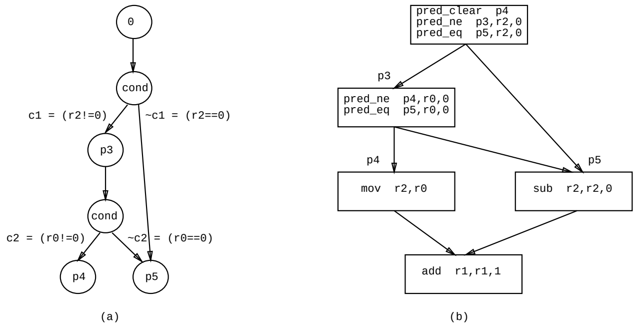

Figure 8: An example (a) predicate hierarchy graph, and (b) corresponding control flow graph.

predicate nodes. Subsequent predicates are added to their parent condition nodes. The PHG for Figure 7 is shown in Figure 8a. Instructions 2 and 3 in Figure 7 are considered the same condition in the PHG since they set complementary predicates. Thus, instruction 2 causes the creation of the top-most condition (c1) and results in the creation of a child predicate node for p3. Instruction 3 will add predicate p5 as another child predicated to condition node c1.

The goal of the PHG is to determine, based on the predicates, if two instructions can ever be executed in a single pass through the hyperblock. If they can, then there is a control flow path between these two instructions. A boolean expression is built for the predicate of each instruction to determine the condition under which the instruction is to be executed. Their corresponding expressions are ANDed together to decide if the two instructions can be executed in the same pass through the hyperblock. If the resultant function can be simplified to 0, then there can never be a control path. It is now a relatively simple matter to determine if there is a path between any two instructions. For example, in Figure 7, there is no control path between instructions 6 and 7. To show this, we must first build the equations for predicates p4 and p5. These equations are formed by ANDing together the predicates from the root predicate node down to the current predicate node. If multiple paths may flow to the same predicate, these paths are ORed together. Thus, $p4 = (c1 \cdot c2)$ since it is created by the predicates active at the first condition node (c1) and the second condition node (c2). The equation for $p5 = (\neg c1 + c1 \cdot \neg c2)$ since it may be reached by two paths. ANDing these equations results in $p4 \cdot p5 = (c1 \cdot c2) \cdot (\neg c1 + c1 \cdot \neg c2)$ which can be simplified to zero. Therefore, there is no control path between these two instructions. Figure 8b shows the complete control flow graph that is generated with the aid of the predicate hierarchy graph shown in Figure 8a.

3.4 Hyperblock-Specific Optimizations

Two optimizations specific to improving the efficiency of hyperblocks are utilized, instruction promotion and instruction merging. Each is discussed in the following section.

Instruction Promotion. Speculative execution is provided by performing instruction promotion. Promotion of a predicated instruction removes the dependence between

```
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 op(y), y < x, such that
           dest(y) = pred(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 9: Algorithm for type 1 instruction promotion.

the predicated instruction and the instruction which sets the corresponding predicate value. Therefore, instructions can be scheduled before their corresponding predicate are determined. Instruction promotion is effective by allowing long latency instructions, such as memory accesses, to be initiated early. Tirumalai *et al.* first investigated instruction promotion to enable speculative execution for software pipelined repeat-until loops [13]. In this paper, instruction promotion is extended to more general code sequences in the context of the hyperblock.

Promoted instructions execute regardless of their original predicate's value. Therefore, promoted instructions must not overwrite any register or memory location which is required for correct program execution. Also, exceptions for speculative instructions should only be reported if the speculative instruction was supposed to execute in the original code sequence. Exceptions for speculative instructions are assumed to be handled with sentinel scheduling architecture and compiler support [14]. Therefore, hyperblock instruction promotion concentrates on handling the first condition.

Three algorithms for instruction promotion are utilized to handle different types of instructions. The first algorithm, shown in Figure 9, is used for the simplest form of promotion (type 1). Type 1 instruction promotion is utilized for instructions with predicates that are not defined multiple times. When the destination of the instruction considered for promotion is not live (defined before used along all possible control paths) at the definition point of its predicate, its predicate can be promoted to that of the predicate definition instruction it is dependent upon. In this manner, each application of type 1 promotion reduces the predicate depth by one until the null predicate is reached.

An example illustrating type 1 promotion is shown in Figure 10a (the original code sequence is shown in Figure 6c). The load instruction indicated by the arrow is promoted with a type 1 promotion. Since the instruction which defines predicate p1 is not predicated, the indicated instruction is also promoted to be always executed. After promotion, the load instruction is no longer flow dependent on the predicate comparison instruction and can be scheduled in the first cycle of the hyperblock.

Type 2 instruction promotion is utilized for instructions with predicates defined multiple times. The algorithm (Figure 11) is similar to type 1 promotion except that the instruction is promoted all the way to the null predicate. A single level of promotion cannot be utilized due to the multiple definitions of the instruction's predicate each possibly predicated on differing values.

```
L1: load r1, MEM(A)
    pred_le p1, r1, 0
    add r2, r1, 1 (p1 F)
    load r3, MEM(B)
    load r4, MEM(r6) (p1 F)
    add r4, r4, r2 (p1 F)
    add r4, r4, 1 (p1 T)
    add r2, r1, 1 (p1 T)
    bgt r4, 0, L5
    sub r4, r4, 1
    store MEM(r4), r2
    bne r4, 0, L1
```
(a)

```
L1: load r1, MEM(A)
    pred_le p1, r1, 0
    add r2, r1, 1 (p1 F)
    load r3, MEM(B)
    load r5, MEM(r6)
    add r4, r5, r2 (p1 F)
    add r4, r4, 1 (p1 T)
    add r2, r1, 1 (p1 T)
    bgt r4, 0, L5
    sub r4, r4, 1
    store MEM(r4), r2
    bne r4, 0, L1
```
(b)

```
L1: load r1, MEM(A)
    pred_le p1, r1, 0
    add r2, r1, 1
    load r3, MEM(B)
    load r5, MEM(r6)
    add r4, r5, r2 (p1 F)
    add r4, r4, 1 (p1 T)
    bgt r4, 0, L5
    sub r4, r4, 1
    store MEM(r4), r2
    bne r4, 0, L1
```
(c)

Figure 10: Example of hyperblock-specific optimizations, (a) after type 1 instruction promotion, (b) after renaming instruction promotion, (c) after instruction merging.

```
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 op(y), y < x, such that
           dest(y) = pred(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(i) in {op(j), j = i + 1 ... x - 1}
           where op(i) is all ops which define pred(x) or
           ancestors to pred(x).
    then do:
      set pred(x) = ∅. } }
```

Figure 11: Algorithm for type 2 instruction promotion.

Many instructions cannot be promoted due to their destination variable being live on alternate control paths (violate condition 4 in both type 1 and type 2 promotion). Promotion can be performed, though, if the destination register of the instruction is renamed. An algorithm to perform renaming instruction promotion is shown in Figure 12. After an opportunity for renaming promotion is found, uses of the destination of the promoted instruction are updated with the renamed value. A move instruction must be inserted into the hyperblock to restore the value of the original register when the original control path is taken.

An example of renaming instruction promotion is shown in Figure 10b. The load instruction indicated by the arrow cannot be promoted with either type 1 or type 2 promotion because r4 is live at the definition point of the predicate p1 (the use of r4 on the p1_true control path causes the variable to be live at the definition of p1). However, renaming the destination of the load allows it to be promoted. The use of r4 in the subsequent add is also adjusted to the new destination (r5) to account for the renaming. Note also that the move instruction is not necessary here because r4 is immediately redefined. In normal application of this optimization, the move is inserted, and subsequently deleted by dead code elimination.

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. This technique is derived from partial redundancy elimination [15]. The goal of instruction

```
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 op(y), y > x such that src(y) = dest(x)
           and op(x) dominates op(y).
      3. dest(x) ≠ dest(j) in {op(j), j = x + 1 ... y − 1}
           for all op(y) in (2).
    then do:
      rename dest(x) to new register.
      rename all src(y) in (2) to new dest(x).
      add new move instruction, op(z), immediately following
           op(x) to move the new dest(x) to the old dest(x).
      pred(z) = pred(x).
      pred(x) = ∅. } }
```

Figure 12: Algorithm for renaming instruction promotion.

```
instruction_merging() {
  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 an identical instruction to op(y).
      4. pred(x) is the complement form of pred(y).
      5. the same definitions of src(x) reach op(x) and op(y)
      6. op(x) is placed before op(y).
    then do:
      promote op(x).
      delete op(y). } }
```

Figure 13: Algorithm for instruction merging.

merging is to remove redundant computations along multiple paths of control in the hyperblock. An algorithm to perform instruction merging is shown in Figure 13. Identical instructions with complementary predicates are first identified within a hyperblock. When the source operands definitions reaching each instruction are the same, an opportunity for instruction merging is found. Instruction merging is accomplished by performing a type 1 promotion of the lexically first instruction and eliminating the second instruction.[2] Instruction merging not only reduces the size of hyperblocks, but also allows for speculative execution of the resultant instruction.

An example of hyperblock instruction merging is shown Figure 10c. In this code segment, there are two add instructions (add r2, r1, 1) predicated on complementary predicates in the hyperblock. After instruction merging, the first add is promoted with type 1 promotion to the '0' predicate and the second add is eliminated.

3.5 Extending Conventional Compiler Techniques to use Hyperblocks

After control flow information for hyperblocks is derived (Section 3.3), conventional optimization, register allocation, and instruction scheduling techniques can be extended in a straight forward manner to work with hyperblocks. Differing from basic blocks, control flow within hyperblocks is not sequential. However, a complete control flow graph among all instructions within a hyperblock may be constructed. Therefore, compiler transformations which utilize the sequentiality inherent to basic blocks must just be modified to handle arbitrary control flow among instructions.

Hyperblocks provide additional opportunities for improvement with conventional compiler techniques. Traditional global techniques must be conservative and consider all control paths between basic blocks. Superblock techniques only consider a single path of control at a time through a loop or straight line code and thus may miss some potential optimizations that could be found across multiple paths. However, a hyperblock may contain anywhere from one to all paths of control, and therefore can resolve many of the limitations of superblock techniques and traditional global techniques.

[2]Note that instruction merging may appear to undo some of the effects of node splitting. However, instructions may only be merged when they are dependent on the same instructions for source operands (cond 5), thus node splitting is only undone for instructions it was not effective for.

4 Performance Evaluation

In this section, the effectiveness of the hyperblock is analyzed for a set of non-numeric benchmarks.

4.1 Methodology

The hyperblock techniques described in this paper have been implemented in the IMPACT-I compiler. The IMPACT-I 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 machine description file characterizes the instruction set, the microarchitecture (including the number and type of instructions that can be fetched/issued in a cycle and the instruction latencies), and the code scheduling model. For this study, the underlying microarchitecture is assumed to have register interlocking and an instruction set and latencies that are similar to the MIPS R2000. The processor is assumed to support speculative execution of all instructions except store and branch instructions. Furthermore when utilizing hyperblock techniques, the processor is assumed to support predicated execution (as described in Section 2) with an unlimited supply of predicate registers.

For each machine configuration, the execution time, assuming a 100% cache hit rate, is derived from execution-driven simulation. The benchmarks used in this experiment consist of 12 non-numeric programs, 3 from the SPEC set, *eqntott, espresso, li*, and 9 other commonly used applications, *cccp, cmp, compress, grep, lex, qsort, tbl, wc, yacc*.

4.2 Results

The performance of the hyperblock techniques are compared for 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. No limitation has been placed on the combination of instructions that can be issued in the same cycle. Performance is reported in terms

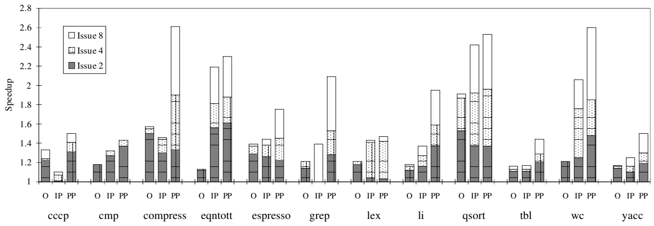

Figure 14: Performance comparison of various scheduling structures, (O) basic block, (IP) hyperblock with all execution paths, (PP) hyperblock with selected execution paths.

of speedup, the execution time for a particular configuration divided by the execution time for a base configuration. The base machine configuration for all speedup calculations has an issue rate of 1 and supports conventional basic block compiler optimization and scheduling techniques.

Figure 14 compares the performance using three structures for compile-time scheduling of superscalar processors. Note that hyperblock specific optimizations (promotion and merging) are not applied for this comparison. From the figure, it can be seen combining all paths of execution in inner loops into a hyperblock (IP) can often result in performance loss. *Cccp* and *compress* achieve lower performance for all issue rates for IP compared to basic block (O). Many of the benchmarks show performance loss with IP only for lower issue rates. This can be attributed to a large number of instructions from different paths of control filling up the available instruction slots. However, when the issue rate is increased sufficiently, this problem is alleviated. The performance with blocks selectively included in the hyperblock (PP), as discussed in Section 3.1, is generally the highest for all benchmarks and issue rates. PP provides a larger scheduling scope from which the scheduler can identify independent instructions compared to scheduling basic blocks. Several benchmarks achieve lower performance with PP compared to O for issue 2, due to a lack of available instruction slots to schedule instructions along all selected paths of execution. PP also achieves higher performance than IP for all benchmarks and issue rates. Exclusion of undesirable blocks from hyperblocks reduces conflicts when there is a lack of available instruction slots, and provides more code reordering opportunities.

Figure 15 represents the performance with and without hyperblock-specific optimizations. These optimizations consist of instruction promotion to provide for speculative execution and instruction merging to eliminate redundant computations in hyperblocks. Comparing hyperblocks with all paths of execution combined (IP and IO), an average of 6% performance gain for an 8-issue processor is achieved with hyperblock specific optimizations. For hyperblocks with selected paths of execution combined (PP and PO), an average of 11% speedup is observed for an 8-issue processor. The largest performance gains occur for *compress*, *grep*, and *lex*.

Figure 16 compares the hyperblock with the superblock.[3]

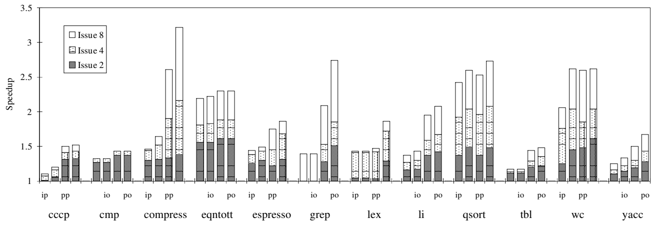

Figure 15: Effectiveness of hyperblock specific optimizations, (IP) hyperblock with all execution paths, (IO) IP with optimization, (PP) hyperblock with selected execution paths, (PO) PP with optimization.

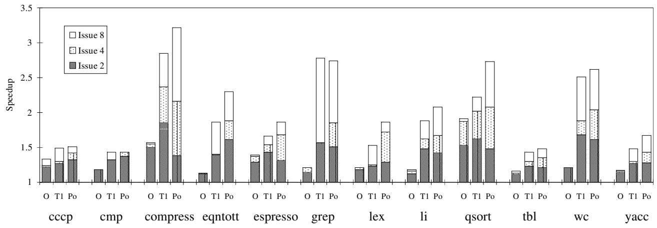

Figure 16: Performance comparison of hyperblock and superblock structure for scheduling, (O) basic block, (T1) superblock, (PO) hyperblock.

From the figure, it can be seen that both structures provide significant performance improvements over basic blocks. The superblock often performs better than the hyperblock for lower issue rates due to the lack of available instruction slots to schedule the instructions from the multiple paths of control. However, the hyperblock generally provides performance improvement for higher issue rate processors since there are a greater number of independent instructions from the multiple paths of control to fill the available processor resources.

Up to this point, an infinite supply of predicate registers has been assumed. The combined predicate register usage distribution for all 12 benchmarks is shown in Figure 17. The graph presents the number of hyperblocks which use the specified number of predicate registers. Two alternate configurations of predicate registers are compared. PR1 represents a scheme without complementary predicate registers similar to the Cydra 5. PR2 utilizes the complementary predicate register organization discussed in Section 2. From the figure, it can be seen that between 16-32 predicate registers satisfy the requirements of all benchmarks used in this study. Comparing PR1 and PR2 distributions shows that in many cases both the true and false predicates are used in the PR2 organization. The average number of predicate registers used in a hyperblock is 3.5 with PR1 and 2.0 with PR2. However, each register in the PR2 organization is equivalent to 2 registers in the PR1 organization (true and false locations), so the PR2 organization uses an average of 0.5 more predicate registers in each hyperblock. Overall though, the complementary predication organization can be efficiently utilized to reduce the overhead of setting predicate register values.

The results presented in this section represent a prelimi-

[3]Note that optimizations to increase ILP, such as loop unrolling, are not applied to either the superblock or the hyperblock for this comparison.

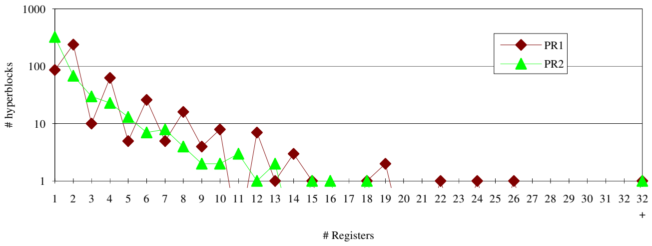

Figure 17: Predicate register usage distribution comparison, (PR1) without complementary predicate registers, (PR2) with complementary predicate registers.

nary evaluation of the hyperblock structure. The evaluation does not include compiler optimizations to increase ILP for superscalar processors, such as loop unrolling or induction variable expansion, applied to any structures. Currently, these optimizations are available for superblocks within the IMPACT-I compiler, however they have not been fully implemented for the hyperblock. To make a fair comparison, superblock ILP optimizations were disabled for this study. A complete analysis of the hyperblock, though, requires ILP optimizations be applied. In our current research, we are incorporating ILP optimizations with hyperblocks and evaluating their effectiveness.

5 Concluding Remarks

Conventional compiler support for predicated execution has two major problems: all paths of control are combined into a single path with conventional if-conversion, and speculative execution is not allowed in predicated blocks. In this paper, the hyperblock structure is introduced to overcome these problems. Hyperblocks are formed by selectively including basic blocks in the hyperblock according to their execution frequency, size, and instruction characteristics. Systematically excluding basic blocks from the hyperblocks provides additional optimization and scheduling opportunities for instructions within the hyperblock. Speculative execution is enabled by performing instruction promotion and instruction merging on the resultant hyperblocks. Preliminary experimental results show that hyperblocks can provide substantial performance gains over other structures. Hyperblocks are most effective for higher issue rate processors where there are sufficient resources to schedule instructions for multiple paths of control. However, additional superscalar optimization and scheduling techniques must be incorporated with hyperblocks to measure their full effectiveness.

Acknowledgements

The authors would like to thank Bob Rau at HP Labs along with all members of the IMPACT research group for their comments and suggestions. This research has been supported by JSEP under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Co. Ltd., Hewlett-Packard, and NASA under Contract NASA NAG 1-613 in cooperation with ICLASS.

References

[1] R. A. Towle, *Control and Data Dependence for Program Transformations.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, 1976.

[2] 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.

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

[4] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[5] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[6] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.

[7] 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.

[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. 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.

[10] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[11] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Water, 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," *To appear Journal of Supercomputing*, January 1993.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

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

[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 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

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