The Program Decision Logic Approach to Predicated Execution

David I. August John W. Sias Jean-Michel Puiatti* Scott A. Mahlke†
Daniel A. Connors Kevin M. Crozier Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
{august, sias}@crhc.uiuc.edu, puiatti@lslsun.epfl.ch, mahlke@hpl.hp.com,
{dconnors, crozier, hwu}@crhc.uiuc.edu

Abstract

Modern compilers must expose sufficient amounts of Instruction-Level Parallelism (ILP) to achieve the promised performance increases of superscalar and VLIW processors. One of the major impediments to achieving this goal has been inefficient programmatic control flow. Historically, the compiler has translated the programmer's original control structure directly into assembly code with conditional branch instructions. Eliminating inefficiencies in handling branch instructions and exploiting ILP has been the subject of much research. However, traditional branch handling techniques cannot significantly alter the program's inherent control structure. The advent of predication as a program control representation has enabled compilers to manipulate control in a form more closely related to the underlying program logic. This work takes full advantage of the predication paradigm by abstracting the program control flow into a logical form referred to as a program decision logic network. *This network is modeled as a Boolean equation and minimized using modified versions of logic synthesis techniques. After minimization, the more efficient version of the program's original control flow is re-expressed in predicated code. Furthermore, this paper proposes extensions to the HPL PlayDoh predication model in support of more effective predicate decision logic network minimization. Finally, this paper shows the ability of the mechanisms presented to overcome limits on ILP previously imposed by rigid program control structure.*

1 Introduction

Exploiting Instruction-Level Parallelism (ILP) in the presence of branches has been the subject of much research. The two most commonly used techniques are control speculation and predication. Control speculation is most commonly performed in superscalar processors using a combination of branch prediction and dynamic scheduling [9][19][23]. Control speculation increases ILP by guessing the outcome of a branch and executing instructions along the predicted path. In this manner, control dependences are broken to execute instructions before the branch outcome is determined. Given an instruction set that supports speculative operations, control speculation can also be performed statically by an aggressive compile-time scheduler which moves instructions across branches [3][12].

Predication has become a popular instruction set architecture feature for expressing program control by conditionally executing instructions [8][16]. A compiler can employ if-conversion to convert a sequence of code containing branches into an equivalent branch-free sequence of conditionally executed instructions [2]. Predicated execution increases ILP by allowing the compiler to schedule operations from multiple paths of control for simultaneous execution.

One fundamental limitation of most previous branch handling techniques is that they do not significantly alter the program's control flow logic. As the compiler translates high-level language control constructs into assembly-level branches, it does not alter the basic control structure. Instead, techniques focused on exposing and increasing ILP within a fixed control structure are applied. With control speculation, this is obvious. Control dependences are removed to enable the motion of instructions above branches. The branches themselves are not altered. Likewise, when predication is applied by the process of if-conversion, branches are transformed into predicate computations and control dependent instructions are rendered conditional by the addition of guarding predicates. This process converts control flow and control dependences into data flow and data dependences, but preserves the original program's control structure.

Restricting a compiler to use the program's unaltered control structure is undesirable for several reasons. First, a high-level language such as C or C++ represents program control flow in an extremely sequential manner through the use of nested if-then-else statements, switch statements, and loop constructs. Each control construct is fully evaluated before proceeding to the next. This sequential computation often defines the program critical paths that constrain the available ILP. Second, programmers represent control flow for understandability or for ease of debugging rather than for efficient execution on the target architecture. As a result, software often contains redundant control constructs that are difficult to detect with traditional compiler techniques. These may involve evaluating the same conditions multiple times or evaluating con-

*Logic Systems Laboratory (DI-LSL), Swiss Federal Institute of Technology of Lausanne (EPFL), CH-1015 Lausanne, Switzerland
†Hewlett-Packard Laboratories, Hewlett-Packard, Palo Alto, CA 94304

To appear in proceedings of ISCA99

ditions that partially overlap. An effective ILP compiler should be capable of transforming the program control structure to eliminate these problems.

The ability to restructure code aggressively is a critical feature of an effective ILP compiler. The most obvious situation where aggressive transformation is regularly applied is on arithmetic expressions. Compilers often completely restructure the programmer's arithmetic computations into more parallel forms using a variety of transformations. These include expression re-association, tree height reduction [11], and blocked back substitution [17]. Although ILP compilers may aggressively restructure computation, they typically preserve the program's original control structure. This conservative approach can seriously limit the level of efficiency as well as the level of ILP achieved in branch-intensive programs.

Our Approach. Motivated by the potential of aggressive techniques for transforming arithmetic expressions, this paper introduces a new approach to optimizing program control flow. The goal of this work is to develop a systematic methodology for reformulating program control flow for more efficient execution on an ILP processor. Control expressed in branches and predicate define instructions is first extracted and represented as a *program decision logic network*. Then, a new, more efficient network is synthesized with the goals of reducing dependence height and redundancy. To accomplish the desired optimization and synthesis, the program decision logic network is modeled as a Boolean equation. Boolean minimization techniques are then applied to simplify and optimize the equation. Finally, the optimized network is re-expressed in the form of predicated assembly code. One unique feature of this approach is that all branches and predicates within a segment of code are treated jointly in a systematic manner.

This paper focuses on compiler techniques and architecture support for effective optimization of programmatic control flow. In particular, we highlight the aspects of the HPL PlayDoh predicate define instructions that are the most useful for our purposes. During the process of developing our compiler support for programmatic logic optimization, we designed a new class of predicate define instructions that extends the PlayDoh architecture to support our optimizer more effectively. We present the key idea behind this extension and show its effectiveness through simulation of compiled codes that use this extension. We observe in our experiments that programmatic logic optimization indeed results in substantial performance improvements in functions where control flow is the major impediment to exploiting ILP.

Previous Work. Previous research in the area of control flow optimization can be classified into three major categories: branch elimination, branch reordering, and control height reduction. Branch elimination techniques identify and remove those branches whose direction is known at compile-time. The simplest form of branch elimination is loop unrolling, in which instances of backedge branches are removed by replicating the body of the loop. More sophisticated techniques examine program control flow and data flow simultaneously to identify correlations among branches [5][15]. When a correlation is detected, a branch direction is determinable by the compiler along one or more paths, and the branch can be eliminated. In [15], an algorithm is developed to identify correlations and to perform the necessary code replication to remove branches within a local scope. This approach is generalized and extended to the program-level scope in [5]. The second category of control flow optimization work is branch reordering. In this work, the order in which branches are evaluated is changed to reduce the average depth traversed through a network of branches [22].

		PlayDoh types						New types			
pSRC	Comp	UT	UF	OT	OF	AT	AF	$\vee$T	$\vee$F	$\wedge$T	$\wedge$F
0	0	0	0	-	-	-	-	-	1	0	0
0	1	0	0	-	-	-	-	1	-	0	0
1	0	0	1	-	1	0	-	1	1	0	-
1	1	1	0	1	-	-	0	1	1	-	0

Table 1. Predicate definition truth table.

The final category of control flow optimization research focuses on the reduction of control dependence height. This work attempts to collapse the sequential evaluation of linear chains of branches in order to reduce the height of program critical paths [18]. In an approach analogous to a carry lookahead adder, a lookahead branch is used to calculate the taken condition of a series of branches in a parallel form. Subsequent operations dependent on any of the branches in the series need only to wait for the lookahead branch to complete. The control dependence height of the branch series is thus reduced to that of a single branch. The mechanisms introduced herein also serve to reduce control dependence height. This paper, however, introduces an approach to minimization and re-expression of control flow networks that is far more general than those proposed in previous work.

2 Architecture Support

Predicated execution, the central architectural feature examined in this work, is a mechanism that facilitates the conditional execution of individual instructions [16]. Predicates are registers that store a single bit value, representing either TRUE or FALSE. Each instruction is associated with a particular predicate, known as its guard predicate, that determines its execution. In the case when an instruction's guard predicate is TRUE, it executes normally. Conversely, when an instruction's guard predicate is FALSE, it is nullified.

The most important component of a predicate architecture is the instruction set support for computing predicates or the *predicate define instructions*. Predicate defines are inserted by the compiler to generate values for control of conditional execution. The PlayDoh predicate define instruction set provides the baseline for our work and is first summarized. Our new strategy for the generation of predicated code identifies several limitations of the PlayDoh instruction set. These limitations are described and our proposed extensions to the PlayDoh predicate define instruction set conclude the section.

Baseline Predicate Architecture. PlayDoh is a parameterized Explicitly Parallel Instruction Computing (EPIC) architecture intended to support public research on ILP architectures and compilation [10]. PlayDoh predicate define instructions generate two Boolean values using a comparison of two source operands and a source predicate. A PlayDoh predicate define instruction has the form:

$$pD_0\_type_0,\ pD_1\_type_1 = (src_0\ \mathbf{cond}\ src_1)\ \langle pSRC\rangle.$$

The instruction is interpreted as follows: pD_0 and pD_1 are the destination predicate registers; $type_0$ and $type_1$ are the predicate types of each destination; src_0 $\mathbf{cond}$ src_1 is the comparison, where cond can be *equal* ($==$), *not equal* ($!=$), *greater than* ($>$), etc.; $pSRC$ is the source predicate register. The value assigned to each destination is dependent on the predicate type. PlayDoh defines three predicate types, *unconditional* (UT or UF), *wired-or* (OT or OF), and *wired-and* (AT or AF). Each type can be in either normal mode or complement mode, as distinguished by the T or F appended to the type specifier (U, O, or A). Complement mode differs from normal mode only in that the condition evaluation is treated in the opposite logical sense.

For each destination predicate register, a predicate define instruction can either deposit a 1, deposit a 0, or leave the contents unchanged. The predicate type specifies a function of the source predicate and the result of the comparison that is applied to derive the resultant predicate. Table 1 (left-hand portion) shows the deposit rules for each of the PlayDoh predicate types in both normal and complement modes. Each entry corresponds to the result assigned to the destination predicate. Note that a "-" means that the destination is left unchanged.

As shown in the table, the unconditional types are always assigned a value. For the UT-type, the value corresponds to the logical conjunction of the source predicate and the comparison result. Conversely, the or-type and the and-type each only assign a value in one circumstance. The OT-type conditionally writes a 1 if both its source predicate and comparison result are TRUE. The or-type can be used to efficiently compute the disjunction of multiple compare conditions by accumulating terms into an initially cleared predicate register. Since the operations computing terms conditionally write the same value, they can execute in any order or even in parallel. Similarly, the and-type can be used to compute the conjunction of multiple compare conditions by accumulating terms into an initially set predicate register.

Limitations of PlayDoh. The major limitation of the PlayDoh predicate types is that logical operations can only be performed efficiently amongst compare conditions. There is no convenient way to perform arbitrary logical operations on predicate register values. While these operations could be accomplished using the PlayDoh predicate types, they often require either a large number of operations or a long sequential chain of operations, or both.

With traditional approaches to generating predicated code, these limitations are not serious, as there is little need to support logical operations amongst predicates. The Boolean minimization strategy described in the next section, however, makes extensive use of logical operations on arbitrary sets of both predicates and conditions. In this approach, intermediate predicates are calculated that contain logical subexpressions of the final predicate expressions to facilitate reuse of terms or partial terms. The intermediate predicates are then logically combined with other intermediate predicates or other compare conditions to generate the final predicate values. Without efficient support for these logical combinations, gains of the Boolean minimization approach are diluted or lost.

Predicate Define Extensions. Two new predicate types are introduced to facilitate generating efficient code using our minimization techniques. These are referred to as *disjunctive-type* ($\vee$T or $\vee$F) and *conjunctive-type* ($\wedge$T or $\wedge$F). Table 1 (right-hand portion) shows the deposit rules for the new predicate types. The $\wedge$T-type define clears the destination predicate to 0 if either the source predicate is FALSE or the comparison result is FALSE. Otherwise, the destination is left unchanged. Note that this behavior differs from that of the and-type predicate define, in that the and-type define leaves the destination unaltered when the source predicate evaluates to FALSE. The conjunctive-type thus enables the compiler easily and efficiently to form the logical conjunction of an arbitrary set of conditions and predicates.

The disjunctive-type behavior is analogous to that of the conjunctive-type. With the $\wedge$T-type define, the destination predicate is set to 1 if either the source predicate is TRUE or the comparison result is TRUE (FALSE for $\wedge$F). The disjunctive-type is thus used to compute the disjunction of an arbitrary set of predicates and compare conditions into a single predicate.

3 Overview of Compiler Techniques

This section presents a conceptual overview of the program decision logic minimization process, starting with the conversion of code to the predicated representation for subsequent optimization. In order to simplify the extraction and manipulation of control expressions, the compiler applies if-conversion and reformulation of non-branch control constructs to transform all programmatic control flow into the predicated representation. In the IMPACT compiler, this conversion is fully performed within acyclic code regions formed using *hyperblock* formation heuristics [14]. To a great extent, the ability of our control logic optimization techniques to improve performance depends on the scope of these regions, as only the control structure transformed into the predicate domain is available for subsequent optimization. In order to promote effective hyperblock formation, aggressive function inlining is performed.

An example extracted from the *UNIX* utility *wc* illustrates the application and benefit of the described techniques. Figure 1 shows the code segment before and after complete if-conversion. As shown in Figure 1(a), the code before if-conversion consists of basic blocks and conditional branches (shown in bold) which direct the flow of control through the basic blocks. As shown in Figure 1(b), the code after if-conversion consists of only a single block of sequential instructions, a hyperblock [13]. The conditional branches have been replaced with predicate define instructions (shown in bold) and the predicate registers defined have been placed as source operands on all guarded instructions in accordance with their execution conditions.

After if-conversion, control speculation is performed to increase opportunities for optimization. Control speculation is a means of breaking a control dependence by allowing an instruction to execute more frequently than is necessary. In a predicated representation, this is performed in *predicate promotion*, the process by which predicate flow dependences are broken and instructions are made to execute speculatively by changing an instruction's guard predicate to another predicate, whose expression subsumes that of the original [14]. When instructions are aggressively promoted, some predicates may no longer be utilized as guards on computation. When a predicate is no longer necessary, the program decision logic is simplified. Figure 2(a) shows the *wc* hyperblock segment after predicate promotion. Comparison with Figure 1(b) shows that four instructions (12, 13, 16, and 17) have had their predicates promoted to the TRUE predicate, denoted in the figure as the absence of a source predicate. However, no predicates were rendered completely unused by this process.

Next, the program decision logic network is constructed. Since predicates can only assume Boolean values, predicates and predicate defines can be viewed as a combinational logic circuit. To derive the Boolean function from a hyperblock, the compiler needs only to examine the predicate define instructions. Consider instructions 7 and 8 in Figure 2(a), in which the expression for $p1$ can be written as: $p1 = \overline{C}_0$ and $p2$ can be written as: $p2 = p1\overline{C}_1$,

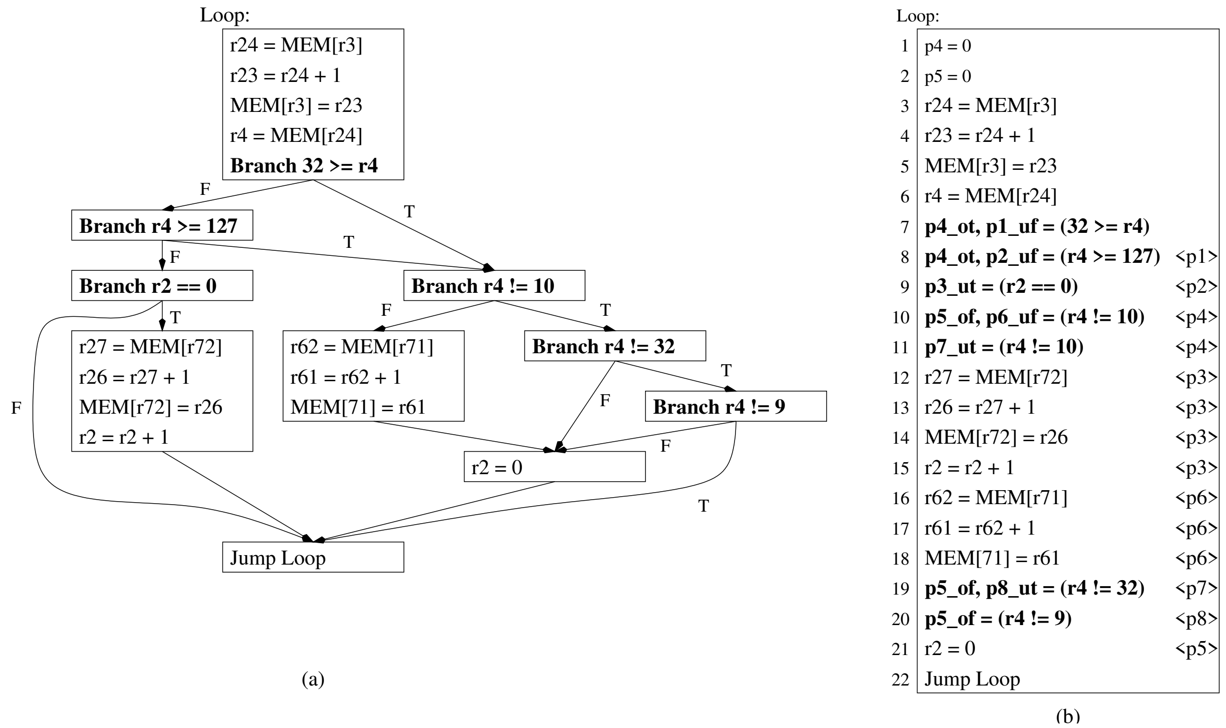

Figure 1. A portion of the inner loop of the UNIX utility wc. The control flow graph (a), and the corresponding hyperblock formed after complete if-conversion (b).

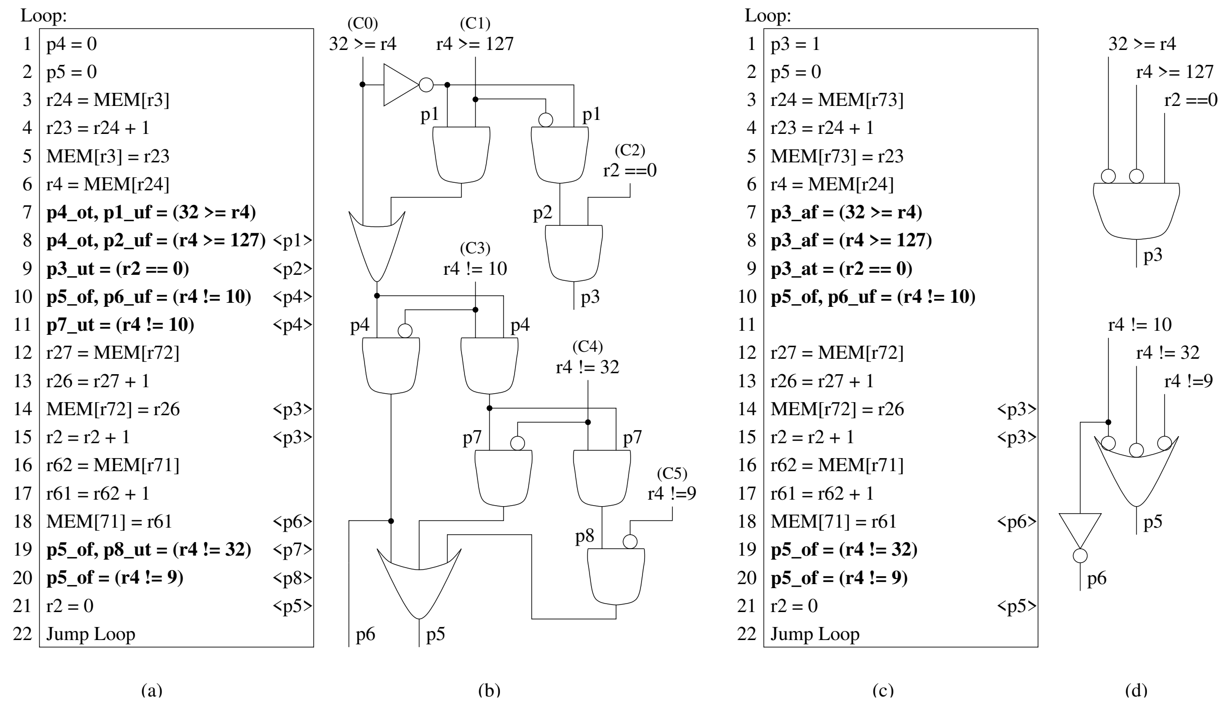

Figure 2. The wc hyperblock after speculation but before logic minimization (a) and its corresponding logic diagram (b). The hyperblock after logic minimization (c) and its corresponding logic diagram (d).

where C_0 is the condition: ($32 \geq$ r4) and C_1 is the condition: (r4 ≥ 127). The expression for $p2$, in terms of conditions, is $p2 = \overline{C}_0\overline{C}_1$. In the course of this complete back substitution, expressions based on condition variables are formulated for all predicate define instructions. The composition of all these expressions is the program decision logic network. This network can be modeled as a logic circuit that represents all the decisions made in the program. The logic circuit has conditions as its input and the predicates which control computation as its output. The multiple-output Boolean logic circuit for the *wc* code segment is shown in Figure 2(b).

Once the logic circuit has been derived, many CAD techniques can be employed to simplify the program decision logic network. In the IMPACT compiler, the derived Boolean function is represented with a Binary Decision Diagram (BDD) [1]. The BDD algorithms used are described in [6]. The *predicate BDD* contains the relationship among predicates as defined by the network of predicate define operations. The predicate BDD is used throughout the compiler as a database for queries made by optimizations when operating on predicated code. For example, one common query is to determine if one instruction executes only when another instruction has executed. This query is equivalent to the dominance relationship in the control flow domain. Here, the BDD is queried to determine if the predicate expression of one instruction subsumes the predicate expression of another. Queries to the BDD are made in IMPACT by the optimizer, the scheduler, and dataflow analysis.

For the purposes of decision logic minimization, the BDD provides a simple method by which expressions describing the hyperblock logic can be derived. The only expressions requested from the BDD are those expressions describing the *essential predicates*. Essential predicates are those predicates that guard real computation instructions (any instruction that is not a predicate define). In Figure 2(a), the essential predicates are $p3$, $p5$, and $p6$. Predicates $p1, p2, p4, p7$, and $p8$ are *non-essential predicates* as they are used only as intermediates in evaluation of the essential predicates.

The BDD maintains a canonical representation of the decision logic functions, from which a Boolean sum-of-products expression can be produced for any represented function. Note that the expression thus generated reflects the canonical nature of the BDD's internal representation, and is usually not optimal for expressions with multiple product terms. Therefore, it is necessary to optimize the derived expression before attempting to synthesize a predicate defining structure.

The expressions describing the evaluation of the essential predicates are optimized using techniques which eliminate redundant terms in the function and which re-express the Boolean function in a more parallel form. The resulting expression is reformulated back into predicate define instructions in the hyperblock. Section 4 presents the details of the Boolean logic optimizers and reformulators studied in this work. These optimizers and reformulators must balance the reduction of dependence height with the number of predicate defines that can be accommodated in the code schedule. This involves making an accurate estimate of how much time is available for computation of control functions based on the availability times of conditions and when predicates need to be consumed. These and other considerations make the design of an optimizer and a reformulator nontrivial.

Figures 2(c) and 2(d) show the reformulated hyperblock and corresponding logic circuit after the minimization process is complete. The number of logic gates in the circuit implementation is

Cycle	Instructions issued					
0	op1	op2	op3	op12	op16	
1	op4	op6	op13	op17		
2	op5	**op7**				
3	**op8**					
4	**op9**	**op10**	**op11**			
5	op14	op15	op18	**op19**		
6	**op20**					
7	op21	op22				

(a) Schedule for the hyperblock in Figure 2(a).

Cycle	Instructions issued					
0	op1	op2	op3	op12	op16	
1	op4	op6	**op9**	op13	op17	
2	op5	**op7**	**op8**	**op10**	**op19**	**op20**
3	op14	op15	op18	op21	op22	

(b) Schedule for the hyperblock in Figure 2(c).

Figure 3. Comparison of the static schedules for the wc hyperblock before and after logic minimization.

reduced from ten to three. In addition, the six-level gate network in Figure 2(b) is reduced to a single-level gate network in Figure 2(d). All non-essential predicates were also eliminated as part of this process. An example optimization performed on the logic circuit takes the form: $C_0 + C_1\overline{C}_0 \rightarrow C_0 + C_1$. An application of this optimization occurs between instructions 7 and 8 when computing $p4$.

The values of variables in the decision logic network are supplied by evaluating conditions on predicate define instructions. It is important to recognize that these variables are not necessarily independent, and that knowledge of the relationships between these variables can allow for significant further optimization of the predicate define structure. Consider the computation of $p6$ in Figure 2(a). Instruction 10 computes $p6\_uf = C_3 \ \langle p4 \rangle$. Logically, this leads to the expression $p6 = \overline{C}_3(C_0 + C_1)$, where $C_0 = (32 \geq r4)$, $C_1 = (r4 \geq 127)$, and $C_3 = (r4 \neq 10)$. Here, since $\overline{C}_3$ implies C_0 and excludes C_1, the expression for $p6$ can be simplified to $p6 = \overline{C}_3$. In our approach, the relationships between conditions are represented in a BDD, termed the *condition BDD*, which can be queried to determine if logical implications exist between conditions and, if so, what they are. The current implementation of this mechanism identifies "families" of integer register-constant comparisons which are based on the same definition of a given register. Then, within each family, a number line is created and divided into disjoint segments from which the set of register values yielding a "TRUE" evaluation for any member condition can be composed by union. Finally, the relationships between the comparisons are described in BDD form using a finite domain technique [7]. Various elements of the optimizer query this BDD to determine the inherent relationships between conditions, which are the decision network's input variables.

The overall effectiveness of the program decision logic minimization process on the *wc* example is best shown by comparing the schedules of the code before and after optimization. For illustration purposes, a six-issue processor with no restrictions on the combination of instructions that may simultaneously be issued is assumed. Furthermore, all instructions are assumed to have a latency of one cycle. Figure 3 presents the schedules for the example hyperblock before and after optimization. The instructions in bold correspond to the predicate defines in each hyperblock. The schedule for the pre-optimization hyperblock (Figure 3(a)) is relatively sparse due to the sequentiality of the predicate defines. The overall schedule length is eight cycles. The schedule after logic minimization is reduced by a factor of two. The chain of predicate define instructions in the original hyperblock is replaced by a parallel, more efficient computation in the optimized hyperblock. The reformulated hyperblock requires only a single level of predicate defines to compute the essential predicates as opposed to the five-level network used in the original code, yielding a significant increase in performance.

Once the decision component has been optimized and reformulated back into the predicated representation, further compiler transformations need to be performed. For machines without real predication support, complete reverse if-conversion must be performed [21]. For machines which support predication, partial reverse if-conversion can be employed to create the proper balance of control flow and predication for the target architecture [4].

p4_ot, p1_uf = (32>=r4)		
p4_ot, p2_uf = (r4>=127) <p1>		
p3_ut = (r2 == 0) <p2>	p5_of, p6_uf = (r4 != 10) <p4>	p7_ut = (r4 != 10) <p4>
p5_of, p8_ut = (r4 != 32) <p7>		
p5_of = (r4 != 9) <p8>		

(a) Original predicate define structure.

C_0	(32>=r4)
C_1	(r4>=127)
C_2	(r2==0)
C_3	(r4!=10)
C_4	(r4!=32)
C_5	(r4!=9)

$p3$	$\overline{C}_0\overline{C}_1C_2$
$p6$	$C_0\overline{C}_3+\overline{C}_0C_1\overline{C}_3$
$p5$	$C_0\overline{C}_3+\overline{C}_0C_1\overline{C}_3+$ $C_0C_3\overline{C}_4+\overline{C}_0C_1C_3\overline{C}_4+$ $C_0C_3C_4\overline{C}_5+\overline{C}_0C_1C_3C_4\overline{C}_5$

(b) Conditions and original predicate expressions.

$p3$	$\overline{C}_0\overline{C}_1C_2$
$p6$	$\overline{C}_3$
$p5$	$\overline{C}_3+\overline{C}_4+\overline{C}_5$

(c) Optimized predicate expressions.

p3_af = (32 >= r4)	p3_af = (r4 >= 127)	p3_at = (r2 == 0)	...
... p5_of, p6_uf = (r4 != 10)	p5_of = (r4 != 32)	p5_of = (r4 != 9)	

(d) Optimized predicate define structure.

Figure 4. Example: optimization of wc predicate network.

4 Minimization of Program Decision Logic

The previous section provided an overview of the process of program control height minimization through the optimization of the predicate define network. This section describes in detail the mechanisms by which the predicate define optimizer generates new predicate define instructions to evaluate more efficiently the program's essential predicate functions. The discussion in this section assumes that the program's decision logic has been represented by the predicate BDD and the condition BDD, and that Sum-Of-Products (SOP) expressions for the essential predicates have been extracted as described in the previous section. Once the program decision logic has been extracted, program control is optimized and re-expressed in four steps. First, sum-of-products expressions are formed to represent predicate functions in terms of program conditions. These expressions are then optimized using condition analysis and traditional Boolean logic minimization techniques. The resulting optimized expressions are then optionally factorized based on condition availability times and resource constraints. Finally, program control is re-expressed in predicate define instructions, either in a two-level network or in a multi-level network, depending on whether or not factorization was performed.

The generation of an efficient predicate define network begins with the extraction and subsequent optimization of the sums-of-products for the predicate functions. Figure 4(b) shows the expressions extracted for the essential predicates in the *wc* example, as well as the conditions to which the variables in the expressions correspond. Figure 4(a) shows the original predicate define network for reference. Since the control expressions are completely represented by the predicate BDD in terms of conditions, the non-essential predicates are eliminated from consideration. This process maps the predicate define structure, in this case five stages of predicate define instructions, into a sum-of-products which can be synthesized into a two-cycle sequence of predicate define instructions. However, this expression can exhibit a large number of redundant and constant-FALSE products, and must be refined before use in define regeneration. From Figure 4(b), two-level regeneration of the unoptimized expressions of the *wc* example would require thirteen predicate defines in the first level and six in the second, far more than the seven required in the initial network.

Optimization of predicate expressions. Predicate expressions are optimized in two steps, as indicated in Figure 5 in the description of *Simplify_funcs*. First, expressions are reduced using condition BDD information. For example, conditions which imply or exclude each other (i.e. $(\mathrm{r1} < 4)$ *implies* $(\mathrm{r1} < 5)$ and *excludes* $(\mathrm{r1} >= 7)$), can cause predicate expressions to contain redundant or constant-FALSE products, as well as redundant literals in useful products. These extraneous features are removed in this phase. One such case from the benchmark *wc* was examined in Section 3.

Once redundant and constant-FALSE products and literals have been removed from the predicate expressions, the iterative-consensus method is applied to produce a complete sum, and then to select a subset of prime implicants for a simplified two-level logic implementation [20]. Pseudo-code for this algorithm is shown in Figure 5 (*Minimize_SOP*). The heart of this iterative algorithm is the consensus-taking routine, which applies the Boolean theorem $x + \overline{x}y \rightarrow x + y$. After each pass through the product list, products subsumed (covered) by other products are removed. The iterative-consensus algorithm generates a complete sum for the input expression. Non-essential products can then be removed to generate a minimal covering sum.

In this application, the Boolean predicate expressions can be composed of a large number of variables and products (more than thirty in some instances), rendering a direct implementation of the iterative-consensus algorithm, which is exponential, intolerably slow. For this reason, when operating on large functions we apply an heuristic approximation to the iterative-consensus method. This heuristic decreases dramatically the number of intermediate products, and therefore renders the compile time reasonable. Furthermore, using this heuristic, the selection of the minimal sum-of-products expression (covering subset), also ordinarily an expensive procedure, is reduced to a linear form.

The cost of this heuristic is that the result could be suboptimal,

```
Simplify_funcs(func_list)
simplified_func_list = Empty_list();
FOREACH func IN func_list DO
    reduced_func = Reduce_using_condition_BDD(func);
    simplified_func = Minimize_SOP(reduced_func);
    List_append(simplified_func_list, simplified_func);
RETURN simplified_func_list;

Minimize_SOP(func)
product_list = func.product_list;
new_product_list = product_list;
WHILE NOT List_empty(new_product_list) DO
    new_insertion_list = Empty_list();
    FOREACH product_x IN new_product_list DO
        FOREACH product_y IN product_list DO
            consensus = Consensus(product_x, product_y);
            IF consensus THEN
                List_insert_last(new_insertion_list, consensus);
    product_list = List_append(product_list, new_insertion_list);
    new_product_list = new_insertion_list;
    product_list = Eliminate_subsumed_products(product_list);
product_list = Select_covering_subset(product_list);
RETURN product_list;

Factorize(func_list, sched)
factor_list = Empty_list();
FOREACH func_x IN func_list DO
    FOREACH func_y IN func_list BEFORE func_x DO
        IF Factor_simplifies(func_y, func_x) THEN
            IF Resource_constrained(func_x.id) THEN
                IF NOT (List_member(func_y, factor_list) THEN
                    List_insert_last(factor_list, func_y);
                func_x = Factor_SOP(func_x, func_y);
FOR cycle = sched.min_cycle TO sched.max_cycle DO
    FOREACH func IN func_list DO
        FOREACH product IN func DO
            ready_prod = Ready_product(product, cycle);
            match_prod = Match_term(ready_prod, factor_list);
            IF match_prod THEN
                ready_factor = match_prod;
            ELSE
                ready_factor = ready_prod;
                ready_factor.id = Unique_token();
                List_insert(factor_list, ready_factor);
            Factor_term(product, ready_factor);
        List_insert_last(factor_list, func_list);
    Factor_common_disjoint_subexpr(factor_list, func_list);
RETURN func_list, factor_list;

Factor_common_disjoint_subexpr(factor_list, func_list)
FOREACH func IN func_list DO
    product_factor_list = Extract_ready_products (func);
    fact_func = Find_factor(product_factor_list, func);
    IF fact_func THEN
        match_fact = Match_factor(fact_func, factor_list);
        IF NOT (match_fact) THEN
            fact_func.id = Unique_token();
            List_insert(factor_list, fact_func);
            match_fact = fact_func;
        Factor_term(func, match_fact);
```

Figure 5. Pseudo-code for performing optimization of predicate expressions

which could cause the generation of expressions with more predicate define instructions than necessary. Depending on the order in which the comparisons are made, the heuristic may eliminate some products that are necessary to generate other simpler products. To minimize this problem the heuristic includes a manipulation which sorts the products in order to reduce the likelihood of a non-optimal solution.

Figure 4(c) shows the expressions to which the essential predicates of the *wc* example are reduced in the logic optimization phase. These expressions are both less complex and more parallel than the original functions.

Two-level predicate synthesis. Following optimization of the predicate expressions, the control logic can be synthesized most intuitively as a two-level predicate define network which directly evaluates the minimized sum-of-products expression. In this approach, two levels of predicate define instructions are used for each predicate. The first level consists of and-type predicate defines of the form $p_i\_at = C_i\langle T\rangle$, where one predicate p_i is defined for each product term in the predicate expression, and T is the TRUE predicate, which always has the value 1. The second level consists of or-type predicate defines of the form $p_j\_ot = (condT)\langle p_i\rangle$, where there is one such predicate define for each product (p_i) and $condT$ is an invariant TRUE condition (*e.g.* $(0 == 0)$). Thus, a predicate expression having L literals and M products consumes $M + 1$ predicates and performs $L + M$ predicate assignments. Continuing the *wc* example in Figure 4(d), note that the two special cases of two-level predicate synthesis occur, in which the computation of functions containing a single product and functions that are disjunctions of single-literal products can be performed in a single cycle. Note also that predicates which have products in common can share intermediate predicates, allowing for some savings through reuse. In most cases, however, two-level synthesis generates an enormous number of predicate define instructions, since redundancy between products is not reduced. Furthermore, since the evaluation of such a predicate define network usually takes at least two cycles after the last condition becomes available (one for the and-level and one for the or-level), the result may also be suboptimal in latency, even when scheduled for infinite issue. Results demonstrating both these phenomena are presented in Section 6. Clearly, a more sophisticated technique is required.

Factorization. In the example of the previous section, the code sample from *wc* exhibited a large ratio of control height to computation height, and the computation was nearly completely dependent on the outcome of the decision mechanisms. Thus, it was important to compress the height of the entire decision structure as much as possible, as any reduction in the decision height improved performance. Furthermore, since the predicate conditions were strongly related, the resulting predicate define structure actually reduced the predicate and predicate define count. In many other situations, however, predicates are based on more independent conditions and the number of predicate define instructions required to generate a two-level network may be quite large. Factorization seeks to use the code's computation or datapath height to hide some portions of the decision latency which are not on the critical path. Thus, the optimizer is free to focus on reducing implementation size rather than delay when implementing these non-critical sections, saving valuable predicate registers and instruction issue resources.

The factored generation method determines how much factoring can be performed at no cost. The availability times of conditions and the time at which predicate values are needed by the computation component drive the factorizer. If parallel computation height, rather than predicate define height, is the critical path through the code segment, then it is beneficial to perform factorization instead of full expression flattening.

To measure the availability times of conditions and the time at which predicate values are needed, a special version of the code is scheduled. This version of the code has all the predicate dependences between predicate defines removed. For each condition, a predicate destination is added for each predicate whose function depends on that condition. In the resultant code, predicate define instructions are placed as early in the schedule as their condition availability will allow. Also, all uses of a predicate are placed as early as possible, but after all the conditions which may be needed to compute it. By extracting the issue time of these predicate defines and predicate uses, the amount of time the new predicate network has to compute predicates without performance penalty is ascertained. This information is then used together with the previously extracted predicate expressions in later stages of optimization.

With factorization, the goal is to form intermediate predicates as the conditions to compute them become available, and then to reuse these intermediate predicates in the computation of the essential predicates. This activity factors the optimized sum-of-products expression or its products so that the resulting define structure may take more cycles, but can reuse more intermediate predicates, thus saving predicate defines and predicate registers.

In certain cases, when resource utilization is very high and predicate functions are very complex, factorization becomes critical for performance. In some cases, generation of code which would optimally generate the predicate results on an infinitely wide machine could actually degrade performance in a real machine due to excessive width. In these situations, an additional factorization preprocessing stage is applied, in which predicates are selectively factored on subexpressions available in essential predicates generated earlier in the original code. This activity, shown in lines 2 though 8 of *Factorize* in Figure 5, has the effect of moderating the restructuring of control in cases where reordering of the predicate expressions would generate a define network too wide for the target architecture.

Figure 6 shows an example extracted from the function *cofactor* of the *008.espresso* benchmark. The minimal sum-of-products is computed for each of the final predicates, as shown in Figure 6(a). Next, with the help of condition availability and predicate use times from Figure 6(a) and 6(b), all useful predicates are factorized, and common expressions are shared. Figure 6(c) shows the result of this method. This factoring results in the reduction of the number of predicate define instructions from 37 to 13. Furthermore, the useful predicates ($p1$ and $p2$) are available a single cycle after the last condition is evaluated, sooner than would be possible using a two-level synthesis of the predicate expressions, two cycles after the last condition evaluation.

In the direct sum-of-products conversion, the computation of $p1$ and $p2$ begin respectively at cycle 5 and cycle 6, at the availability time of their latest conditions; results are available two cycles later. With the factorization method, however, predicates $p1$ and $p2$ can be evaluated in a single cycle after the availability of C_5 and C_6. Thus, in some cases, the factorization method is able to reduce predicate latency by one cycle compared to the result of the direct sum-of-products conversion.

Pred	Expression	Use Cycle
$p1$	$C_0C_2C_4\overline{C}_5+$ $C_0C_2C_3\overline{C}_5+$ $C_0C_1\overline{C}_5$	6
$p2$	$C_0C_2C_4\overline{C}_5C_6+$ $C_0C_2C_3\overline{C}_5C_6+$ $C_0C_1\overline{C}_5C_6$	7

(a) Optimized predicate expressions.

C_0	C_1	C_2	C_3	C_4	C_5	C_6
1	1	2	3	4	5	6

(b) Condition availability.

Time	Predicate expression
1	$p3\_ut = C_0$ $p4\_at = C_0$ $p4\_at = C_1$
2	$p5\_ut = C_2$ $p6\_ut = C_2\ \langle p3 \rangle$
3	$p7\_ut = C_3\ \langle p6 \rangle$
4	$p8\_ut = C_4\ \langle p6 \rangle$
5	$\mathbf{p1}\_of = C_5\ \langle p7 \rangle$ $\mathbf{p1}\_of = C_5\ \langle p8 \rangle$ $\mathbf{p1}\_of = C_5\ \langle p4 \rangle$
6	$\mathbf{p2}\_ut = C_6\ \langle p1 \rangle$

(c) Factoring with schedule time information.

Figure 6. Factorized predicate define optimization.

5 Architecture Support for Synthesis

Description of the predicate optimization in previous sections has disregarded the means by which Boolean expressions are converted back into predicate defining instructions. This section examines the instruction set considerations that evolved in supporting an effective predicate synthesis system.

Implementation of two-level predicate synthesis is straightforward in the HPL Playdoh predicate architecture. For example, in Figures 2 and 4(c), a simple sum-of-product expression is converted into a small set of predicate defines.

Synthesis of multi-level factored functions is not as simple as product-of-sums or sum-of-products expressions, but yields significant improvements in both performance and predicate define count. When an expression is factored out of one or more predicate expressions, its value is computed and stored in a predicate for later use. After factoring, expressions to be synthesized thus contain predicates as well as conditions. To illustrate the use of factoring, the example in Figure 7 is presented. In Figure 7(a), predicate $p1$ is a subexpression of $p2$. Factoring $C_1 + C_2$, or $p1$, out of $p2$ allows more sharing of predicate defines between predicate computations. As can be seen in Figure 7(b), this subexpression can be computed in cycle 1 using or-type predicate defines. The availability of this expression before the computation of $p2$ allow an efficient application of factorization. In cycle 3, the conjunction of the subexpression stored in $p1$ with the previous value of $p2$ and C_3 is required. This expression is awkward to compute using the PlayDoh predicate define semantics because the logical

Pred	Expression	Use Cycle
$p1$	$C_1 + C_2$	3
$p2$	$C_0C_1C_3 + C_0C_2C_3$	4

(a) Optimized predicate expressions.

C_0	C_1	C_2	C_3
1	2	2	3

(b) Condition availability.

Time	Predicate expression
1	$p2\_\wedge t = C_0$
2	$p1\_ot = C_1$ $p1\_ot = C_2$
3	$p2\_\wedge t = C_3 \langle p1 \rangle$

(c) Factorization with conjunctive-type predicate defines.

Time	Predicate expression
1	$p3\_at = C_0$ $p4\_at = C_0$
2	$p1\_ot = C_1$ $p1\_ot = C_2$ $p3\_at = C_1$ $p4\_at = C_2$
3	$p3\_at = C_3$ $p4\_at = C_3$
4	$p2\_ot = TRUE \langle p3 \rangle$ $p2\_ot = TRUE \langle p4 \rangle$

(d) No factorization

Time	Predicate expression
1	$p2\_at = C_0$
2	$p1\_ot = C_1$ $p1\_ot = C_2$ $p3\_af = C_1$ $p3\_af = C_2$
3	$p2\_at = C_3$ $p2\_af = TRUE \langle p3 \rangle$

(e) Factorization without conjunctive-type predicate defines.

Figure 7. Various methods of predicate expresssion regeneration.

combination of predicates is not directly supported. With the extension to the PlayDoh predicate define semantics, this expression can be computed with a single conjunctive-type predicate define. Figure 7(c) shows the final set of predicate defines used to compute the factored predicate expressions. The two expressions are computed using a total of two predicates and four predicate defines. The last predicate define conjoins $p1$ and C_3 to the previous contents of $p2$ (C_0) to finish the computation of the $p2$ expression.

Benefit of Architectural Extension. The primary use of the conjunctive-type predicate defines is to reduce the number of instructions required to compute factored expressions. This reduction is best illustrated when the generation of the predicate expressions is done without the conjunctive type. Figures 7(d) and 7(e) show two generation options that do not use the conjunctive type. In Figure 7(d), no factorization is performed and the direct sum-of-products expressions are computed. This approach requires a total of ten predicate defines, six more instructions than was required in Figure 7(c). Further, the two-level nature of the sum-of-products generation adds an extra level of dependence height. In Figure 7(e), factorization is performed, but the conjunctive-type is not used. Here, a total of seven predicate defines, three extra instrutions, is necessary. Of these, two predicate defines are needed to compute the complement of the factored expression. This is done by applying DeMorgan's theorem. Another method of complementing $p1$ could have been used, but it would have cost a cycle of latency. The third extra predicate define is used to nullify $p2$ if the complement of the factored predicate is TRUE. Note that the disjunctive-type predicate defines are analogously useful when product-of-sum expressions are used.

6 Experimental Results

The effectiveness of the Boolean minimization techniques for generating predicated code are evaluated in this section. These techniques have been implemented within the IMPACT experimental compiler framework and applied to a set of benchmarks.

Processor Model and Benchmarks. The processor modeled is an 8-issue processor with in-order execution and register interlocking. The processor has no limitation on the combination of instructions that may be issued each cycle, except that only one branch may be executed per cycle. The instruction latencies assumed match those of the HP PA-7100 microprocessor. The instruction set contains a set of non-trapping versions of all potentially excepting instructions, with the exception of branch and store instructions, to support aggressive speculative execution. The instruction set also contains support for predicated execution as described in Section 2.

The execution time for each benchmark was obtained using the IMPACT emulation-driven simulator. Some dynamic effects such as branch mispredictions, cache misses, and TLB misses were not measured. This decision was made to ensure that the experimental results highlight the effects of the techniques being evaluated. Since the reformulation of the predicate decision logic does not affect the basic nature of memory access patterns and branch histories, any change in these dynamic effects between the original and optimized codes would be spurious in nature.

The benchmarks used in this experiment consist of 13 non-numeric programs: four of the SPECINT 92 benchmarks, *008.espresso*, *022.li*, *026.compress*, *072.sc*; six of the SPECINT 95 benchmarks, *099.go*, *124.m88ksim*, *126.gcc*, *129.compress*, *130.li*, *132.ijpeg*; and three UNIX utilities, *cccp*, *lex*, *wc*.

Results. The first set of results presented compare the performance of a code set transformed with the described techniques to the performance of a baseline code set. The baseline code consists of the best code generated by the IMPACT compiler for a predicated architecture using hyperblock compilation techniques. The transformed code corresponds to the baseline hyperblock code after Boolean minimization techniques are used to restructure the

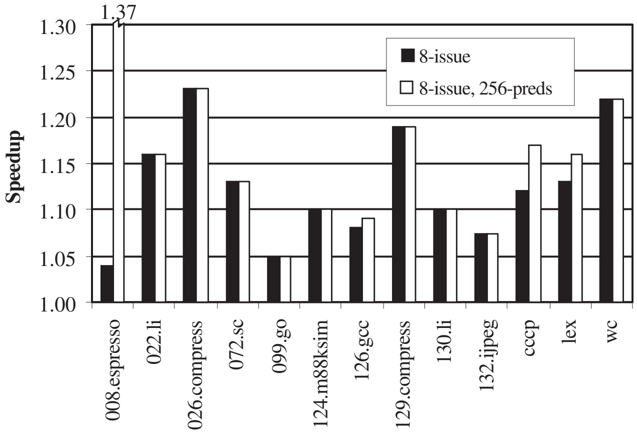

Figure 8. Speedup from minimization of program decision logic.

predicate defines, and after the code is rescheduled. Performance is derived by computing the ratio of the execution cycle count for the baseline code to that of the transformed code. The performance is examined at two levels, first at the overall benchmark level and then at the benchmark function level.

The overall benchmark speedups are presented in Figure 8. For each benchmark, two results are reported. The first is the benchmark speedup on the target architecture. The unweighted average speedup for all the benchmarks is 1.13. For some benchmarks, such as *022.li*, *026.compress*, *129.compress*, and *wc*, the program decision height was significantly limiting performance throughout the most frequently executed portions of the code; when this height is reduced by our techniques, speedups of around 1.2 are achieved.

The second result presented for each benchmark, labeled "8-issue, 256-preds," is the speedup on a hypothetical machine capable of issuing eight non-predicate-define instructions and up to 256 predicate defines per cycle. The significance of the second set of numbers is that they reflect only the dependence height of predicate defines, while eliminating their resource consumption characteristics. These results suggest a logical upper bound for gains possible with more effective factorization techniques. In most benchmarks, the optimizer produced a number of predicate defines that was appropriate for the schedule and machine model. However, in four benchmarks, *008.espresso*, *cccp*, *126.gcc*, and *lex*, the optimizer was unable to balance height reduction with resource consumption and performance was penalized. This effect was very dramatic in *008.espresso* because it is very decision height limited. Unfortunately, the excessive optimization opportunity available in *008.espresso* allowed the current minimization heuristic to be overly aggressive in reducing height. With more advanced factorization techniques, the number of predicate defines could be reduced in these instances, more closely approximating the "8-issue, 256-preds" results.

Overall, the full benchmark results are encouraging. In most cases, the benefit of our technique was limited solely by the bottleneck created by program computation height. During our experimental exploration, we observed that as optimizations which target computation height were improved, the decision logic became dominant and relative speedups improved. In particular, data and memory dependences seemed to hide much of the program decision height reduction in many important hyperblocks. As the various components of compiler technology mature, the overall effectiveness of Boolean minimization will improve.

To better understand the effect program decision logic minimization has on complete programs, we measured the performance and code size characteristics of a number of selected functions. Table 2 examines the performance of one or more functions from each of the benchmarks. These functions were chosen based on two criteria: significant program execution time and potential for optimization (e.g., the control height was significant relative to the computation height). The table compares the effectiveness of two strategies for program logic transformation: two-level predicate synthesis and factorization. For each strategy, the static number of predicate define instructions, the performance gain on an 8-issue processor with unconstrained predicate define resources (∞), and the performance gain on the 8-issue processor are reported. In addition, the static number of predicate define instructions in the code before minimization is reported.

From the table, the two-level synthesis approach shows mixed results. For the unconstrained machine, the reduction in height translates directly into large speedups. However, the unconstrained performance does not always translate into the same performance gain on the 8-issue processor. This is most pronounced in *008.espresso, essen_parts* where the 1.16 speedup is sharply reduced to 0.39. The primary reason for this behavior is the large increases in the number of predicate define instructions. The predicate defines that are created oversaturate the processor resources and result in loss of performance. Correspondingly, when the number of predicate defines is not increased by a large amount, the unconstrained performance does indeed translate directly into performance on the 8-issue processor. Clearly, factored synthesis is necessary for successful optimization of program decision logic.

As shown in the table, the factored approach yields both larger and more consistent speedups. Both methods reduce the predicate computation height, but the factored approach dramatically reduces the number of predicate defines required for the optimization. The function *126.gcc, canon_hash* provides a good example of this behavior. Both methods achieve good speedup for the unconstrained processor. However, the two-level synthesis approach requires 149 predicate defines to accomplish the improvement. For the 8-issue processor, most of the performance gain is lost due to this increase in instructions. The factored approach reduces the number of predicate defines to 116, increasing the 8-issue speedup to 1.74. The number of predicate defines is still more than the original 89. Note, however, that simply increasing the number of predicate defines from the original code is not necessarily viewed as a negative. Boolean minimization approaches do this systematically to improve performance by identifying condition subexpressions that can be computed early. This allows the final predicate to be made available as soon as possible after the final condition is ready. However, the factored approach is consistently more effective because it factors predicate expressions into multiple-level structures which are less demanding of processor resources than two-cycle evaluations. Another interesting result is that for some functions such as *update* from *072.sc* the factored synthesis method outperforms the two-level method, even at infinite issue. This is a due to the ability of the factorizer to generate expressions in one cycle rather than the two usually required by the two-level synthesis approach. The final experiment examines the effectiveness of the new predicate types (conjunctive and disjunctive, described in Section 2) in the context of Boolean minimization and justifies the need for the proposed architectural extensions. Table 3 presents the effects of the new predicate define types on the speedup for

Benchmark, Function	Original Pred. Defines	Two-Level Synthesis Pred. Defines	Two-Level Synthesis Speedup(∞)	Two-Level Synthesis Speedup(8)	Factored Synthesis Pred. Defines	Factored Synthesis Speedup(∞)	Factored Synthesis Speedup(8)
008.espresso, essen_parts	39	1293	1.29	0.39	49	1.24	1.16
022.li, xleval	48	485	1.07	0.66	80	1.10	1.10
022.li, mark	42	67	1.48	1.48	53	1.50	1.48
026.compress, compress	60	456	1.20	1.03	221	1.23	1.23
072.sc, update	141	240	1.15	1.15	159	1.23	1.23
099.go, getefflibs	98	1083	1.06	0.98	204	1.07	1.07
124.m88ksim, execute	41	47	1.12	1.12	40	1.12	1.12
124.m88ksim, goexec	176	175	1.10	1.09	155	1.09	1.08
124.m88ksim, load_data	42	54	1.30	1.30	53	1.30	1.30
124.m88ksim, loadmem	84	88	1.13	1.13	84	1.13	1.13
126.gcc, invalidate	89	202	1.27	1.24	125	1.22	1.21
126.gcc, flow_analysis	64	92	1.77	1.69	58	1.86	1.86
126.gcc, canon_hash	89	149	1.88	1.20	116	1.90	1.74
129.compress, compress	63	154	1.21	1.21	98	1.26	1.26
130.li, mark	55	148	1.15	1.14	101	1.19	1.19
132.ijpeg, forward_DCT	31	47	1.46	1.35	32	1.46	1.43
cccp, skip_if_group	157	208	1.23	1.05	190	1.32	1.24
lex, cgoto	236	330	1.31	1.10	260	1.18	1.14
wc, main	56	48	1.22	1.31	48	1.22	1.22

Table 2. Speedup and predicate define count for selected functions.

an 8-issue processor, the dynamic predicate define count, and the static predicate define count. The conjunctive and disjunctive types allow certain important logical combinations of predicates and conditions to be expressed more efficiently. For all functions except *022.li, mark* and *130.li, mark*, the performance gained from the program decision logic optimization is diminished when the proposed predicate define types are not available. Further, in six of the nineteen functions, the performance improvement is converted into a performance loss. The most dramatic example of this is *126.gcc, flow_analysis*, in which a 46% performance improvement becomes an 8% performance degradation. The lack of the new predicate define types in the target architecture also causes a code size penalty. In general, the additional predicate types allow significant reductions in both the static and dynamic predicate define counts. In one case, 74% more predicate defines are required if the new types are not available. Six functions do not exhibit this penalty. In these functions, the majority of the predicate expressions are sums of single term "products" making the conjunctive-type unnecessary for instantiating these functions.

Benchmark, Function	Speedup (8) with	Speedup (8) without	Pred. Def. Count Penalty w/o $\wedge$t/$\wedge$f dynamic	Pred. Def. Count Penalty w/o $\wedge$t/$\wedge$f static
008.espresso, essen_parts	1.16	0.96	17.2%	17.8%
022.li, xleval	1.10	1.08	35.4%	35.0%
022.li, mark	1.48	1.48	11.5%	11.3%
026.compress, compress	1.23	1.13	59.8%	60.2%
072.sc, update	1.23	0.98	4.3%	5.0%
099.go, getefflibs	1.07	1.06	17.1%	21.1%
124.m88ksim, execute	1.12	0.89	16.9%	10.0%
124.m88ksim, goexec	1.08	0.90	6.3%	6.5%
124.m88ksim, load_data	1.30	1.07	15.3%	11.3%
124.m88ksim, loadmem	1.13	1.02	74.1%	14.3%
126.gcc, invalidate	1.14	0.77	30.3%	22.4%
126.gcc, flow_analysis	1.86	0.93	0.1%	0.0%
126.gcc, canon_hash	1.74	1.60	11.4%	10.5%
129.compress, compress	1.26	1.10	53.4%	35.7%
130.li, mark	1.19	1.19	18.2%	17.8%
132.ijpeg, forward_DCT	1.43	1.33	0.0%	0.0%
cccp, skip_if_group	1.24	1.20	16.8%	14.2%
lex, cgoto	1.14	1.07	4.7%	10.8%
wc, main	1.22	1.16	4.2%	4.2%

Table 3. Effects of conjunctive-type predicate defines on speedup and instruction count.

7 Conclusion

In this paper, we have presented a new method for optimizing programmatic control flow. Our approach provides a systematic methodology for reformulating program control flow for more efficient execution on ILP processors. Control expressed through branches and predicate defines is extracted and represented as a *program decision logic network*. Boolean minimization techniques are applied to the network both to reduce dependence height and to simplify the component expressions. Redundancy is controlled by employing a schedule-sensitive factorization technique to identify intermediate logical combinations of conditions that can be shared. After optimization, the network is reformulated into predicated code.

We have also presented extensions to the HPL PlayDoh model of predication that allow more efficient computation of the predicate expressions produced by our minimization techniques, namely the *conjunctive* and *disjunctive* predicate assignment types. Experimental results show that in blocks of predicated code with significant control height, the application of logic minimization techniques together with these architectural enhancements provides substantial performance benefit. Across the benchmarks studied, program decision logic minimization provided an average overall speedup of 1.13 for an 8-issue processor. The new predicate assignment types were also shown to significantly reduce the number of predicate define instructions required. As compiler technology progresses to make more extensive and effective use of predicated code, minimization of program decision logic is likely to become an increasingly more important part of total program optimization.

Acknowledgments

The authors would like to thank all the members of the IMPACT compiler team and Professor Eduardo Sanchez for their support, comments, and suggestions. We would like to express our appreciation to Ron Barnes for his help with the execution of the final experiments. We thank Professor Farid Najm for the insight he has provided into the nature of BDDs. We would also like to thank the anonymous referees for their constructive comments. This research has been supported by the National Science Foundation (under grant CCR-9629948), Advanced Micro Devices, Hewlett-Packard, and Intel.

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Transaction on Computers*, C-27(8):509–516, June 1978.

[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*, pages 177–189, January 1983.

[3] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predication and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, June 1998.

[4] D. I. August, W. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 92–103, December 1997.

[5] R. Bodik, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 146–158, June 1997.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, August 1986.

[7] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1992.

[8] P. Y. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[9] W. W. Hwu and Y. N. Patt. HPSm, a high performance restricted data flow architecture having minimal functionality. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 297–306, June 1986.

[10] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.

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

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

[13] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 138–150, June 1995.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[15] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 55–66, June 1995.

[16] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22(1):12–35, January 1989.

[17] M. Schlansker and V. Kathail. Acceleration of first and higher order recurrences on processors with instruction level parallelism. In *Proceedings of Languages and Compilers for Parallel Computing, 6th International Workshop*, August 1993.

[18] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th International Symposium on Microarchitecture*, December 1995.

[19] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[20] J. F. Wakerly. *Digital Design: Principles and Practices*. Prentice Hall, Englewood Cliffs, NJ, 1994.

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

[22] M. Yang, G.-R. Uh, and D. B. Whalley. Improving performance by branch reordering. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 130–141, June 1998.

[23] T. Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, November 1991.