Program decision logic optimization using predication and control speculation

Wen-mei W. Hwu, David I. August, John W. Sias

Abstract— **The mainstream arrival of predication, a means other than branching of selecting instructions for execution, has required compiler architects to reformulate fundamental analyses and transformations. Traditionally, the compiler has generated branches straightforwardly to implement control flow designed by the programmer and has then performed sophisticated "global" optimizations to move and optimize code around them. In this model, the inherent tie between the control state of the program and the location of the single instruction pointer serialized runtime evaluation of control and limited the extent to which the compiler could optimize the control structure of the program (without extensive code replication). Predication provides a means of control independent of branches and instruction fetch location, freeing both compiler and architecture from these restrictions; effective compilation of predicated code, however, requires sophisticated understanding of the program's control structure. This article explores a representational technique which, through direct code analysis, maps the program's control component into a canonical database, a reduced ordered binary decision diagram (ROBDD), which fully enables the compiler to utilize and manipulate predication. This abstraction is then applied to optimize the program's control component, transforming it into a form more amenable to instruction-level parallel (ILP) execution.**

I. Introduction

The performance of statically-scheduled instruction-level parallel (ILP) architectures depends on the compiler identifying a sufficient number of independent instructions to keep the processor's multiple execution units busy. Going well beyond the sometime goal of 1 instruction per cycle, modern ILP compilers may be burdened with attempting to transform single program threads to utilize six or more issue slots per cycle. One of the primary obstacles faced by the compiler is the prevalence of branching instructions, typically comprising as many as one out of every five or six instructions in integer codes. Branches implement program control by conditionally altering the location from which subsequent instructions will be loaded, rendering them obstacles to code motion at compilation time and to parallel instruction fetch at execution time. In this model, evaluation of program decisions is serialized both logically and in practice. Complex decisions require multiple branches, multiple control redirections, and multiple predictions in hardware. Predicated execution [1][2], an architectural alternative in which the execution of individual instructions is enabled or disabled according to the value of an associated Boolean predicate, increases ILP by allowing the compiler to arrange both for parallel computation of control conditions and for parallel execution of instructions from different control paths. A compiler employs *if-conversion* to convert a sequence of code containing branches into an equivalent sequence of conditionally executed instructions [3].

Traditionally, compilers have expressed high-level language control constructs directly into assembly-level branches without fundamentally altering the program's basic control structure. Subsequent techniques have focused on exposing and enhancing ILP within that structure. Constraining the compiler to operate within 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 sequentially through the use of nested if-then-else statements, switch statements, and loop constructs. This structure, which maps easily to conditional branches, binds decision-making to the time a decision must be consumed, often contributing to critical paths that constrain available ILP. Second, programmers tend to 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. An effective ILP compiler should be capable of transforming program control to eliminate these problems.

Although if-conversion retains the basic decision-making steps present in branching code, predication presents an unique opportunity to attack control structure. Since, within a predicated region, decisions, rather than immediately changing the fetch location of the processor, instead contribute to Boolean functions which are applied directly to control individual instructions, the control structure of the program can be considered as a unit for targeted optimization, much as the control path of an integrated circuit would be optimized apart from its data path. In predicated code, this control component enables and disables various instructions in a common "data path," achieving the same result as the original program but with fewer control redirections and a higher degree of ILP. This conversion, while it does not itself optimize the program's control logic, does decouple it from the side-effects of branching and sets up regions of code within which the compiler can separate, analyze, and optimize the control component. The compiler's understanding of program control, however, has traditionally not been powerful enough to allow significant optimization of this aspect of program execution. For the compiler to take advantage of this new perspective, a more systematic approach is required.

This work presents a systematic approach to restructuring program control, which is based on logical analysis of the program's extracted control component. Predication enables the abstraction of this component, the *program decision logic*, which consists of all decision-making that could be expressed as a combinational logic circuit alongside the running program. We therefore present techniques (1) for providing the compiler with a full, abstract comprehension of the program decision logic; (2) for optimizing the program decision logic; and (3) for ensuring that subsequent direct code analysis retains all accuracy available prior to optimization. The result is a system which, among its other benefits, reduces the dependence of ILP performance on initial program control structure.

W. Hwu and J. Sias are with the University of Illinois at Urbana-Champaign. E-mail: w-hwu@uiuc.edu, sias@crhc.uiuc.edu. D. August is with Princeton University. E-mail: august@cs.princeton.edu.

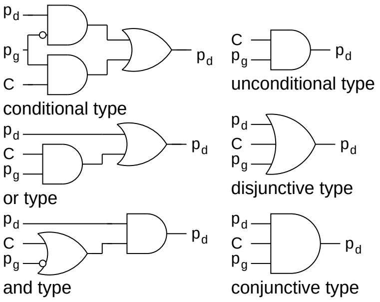


Fig. 1. Predicate definition types and their logic equivalents.

II. System overview

While predicated execution exists in a variety of strengths, we consider the most general form, "full" predication, in which the instruction set architecture provides a set of Boolean predicate registers, a set of predicate defining operations, and, on virtually every instruction, a guard predicate operand. An instruction whose guard predicate is true executes; all others are nullified.[1] Instances of predication presented in this article are in the IMPACT EPIC style [4], which subsumes the predication models of both HPL-PD [5] and Intel's Itanium family [6]. IMPACT EPIC predicate define instructions generate two Boolean values using a comparison of two source operands and a guard predicate. A predicate define instruction has the form:

$$(\mathrm{p}_g)\ \mathrm{p}_{d0}\_type_0,\ \mathrm{p}_{d1}\_type_1 = (src_0\ \mathbf{cond}\ src_1)$$

Here p_g represents the guard predicate and src_0 **cond** src_1 is the comparison, denoted C when treated as a unit, where **cond** can be *equal (==)*, *not equal (! =)*, *greater than (>)*, etc. The guard and condition are shared by the instruction's two predicate computations, which potentially write to p_{d0} and p_{d1}, the destination predicate registers. The computation *type* specifies the function used to compute new destination values given the comparison outcome, the value of p_g, and the previous value of p_d. Figure 1 shows logic-gate equivalents of the predicate define types, which indicate how the destination p_d is computed based on the type, guard predicate p_g and condition C. Four of the types, *or*, *and*, *conjunctive*, and *disjunctive*, are termed *parallel* because multiple such defines having a common type can write simultaneously to one register (due to the fact that each of these types can evoke only a single type of transition in the destination register: $0 \rightarrow 1$ or $1 \rightarrow 0$ but not both).

The original HPL-PD predicate types (*unconditional*, *conditional*, *and*, and *or*) were designed to support if-conversion rather than general Boolean computation. A thoroughly Boolean minimization strategy, however, because it strives to make better reuse of intermediate values, makes extensive use of logical operations on arbitrary sets of both predicates and conditions. Without efficient support for these logical combinations, gains of the Boolean minimization approach are diluted or lost. As an aid to optimization, therefore, the *disjunctive* type ($\vee$T or $\vee$F) and *conjunctive* type ($\wedge$T or $\wedge$F) were introduced [7]. The $\wedge$ type computes the logical *and* of previous value, source predicate and condition; the $\vee$ type likewise computes a logical *or*. These new predicate types increase the reusability of predicate subexpressions by allowing the parallel conjunction and disjunction of existing predicates. As can be noted from the logic of Figure 1, the *unconditional* and *conjunctive* types write a 0 when their guard predicates are false, forming the aforementioned exceptions to instruction squashing.

[1] As will soon be discussed, some types of predicate defines are exceptions to this rule, as they write a result of zero under a false guard predicate.

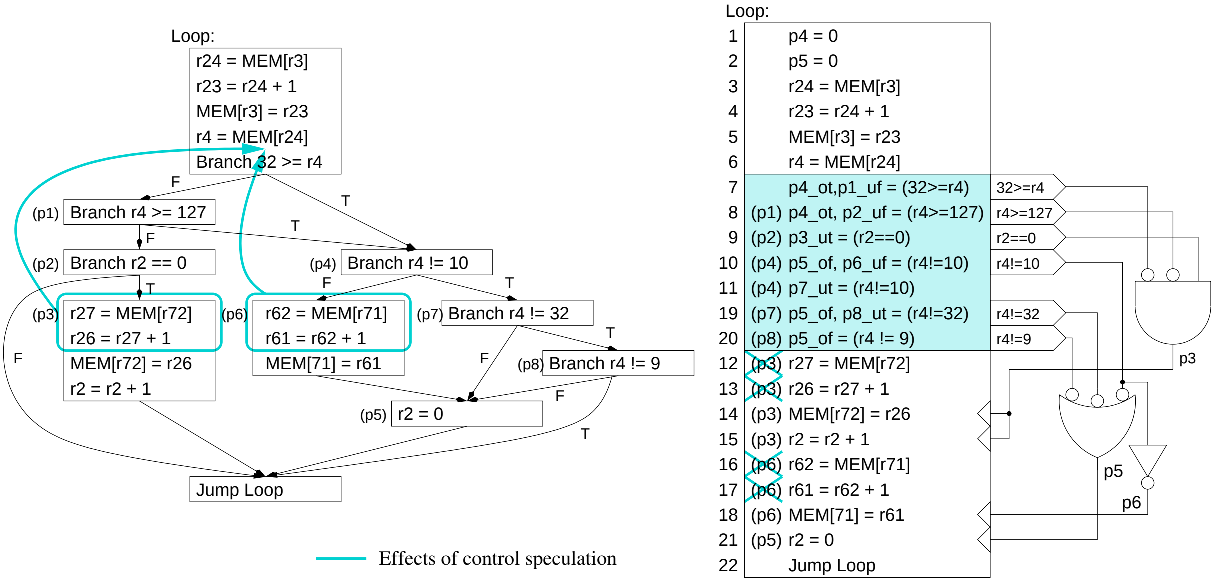
A. Conversion to the predicated form

For our purposes, the program control component is available for manipulation only within predicated regions of code, where it has been decoupled from instruction fetch redirection. If-conversion is therefore used to convert acyclic regions of branching control flow into straight-line predicated blocks. In the IMPACT compiler, this conversion is performed using *hyperblock* formation heuristics [8]. A hyperblock is a single-entry, potentially multiple-exit block containing predication. Heuristics examine the dependence height, resource usage, and profile weight of the paths through an acyclic region of code to determine which paths should be included or excluded to maximize performance. Tail duplication is performed as necessary to exclude undesirable paths. 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 2 shows the code segment before and after if-conversion. As shown in Figure 2(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. Supposing all paths are selected for inclusion, the code after if-conversion consists of only a single block of predicated instructions, a hyperblock [9], as shown in Figure 2(b). In if-conversion, an unique predicate is assigned to each set of control-equivalent blocks, as shown to the left of each basic block. *Unconditional* and *or* type predicate defining instructions are then inserted to compute these predicates, based on the conditions and order of evaluation of the program's branches. Finally, the blocks are combined into a single hyperblock and instructions are assigned guarding predicates according to their home blocks.

After if-conversion, control speculation is performed to increase opportunities for optimization. Control speculation breaks a control dependence by allowing an instruction to execute more frequently than is necessary. In a predicated representation, instructions are speculated by *predicate promotion*, the weakening of an instruction's guard predicate to one subsuming the original [8]. An instruction such as 12 in Figure 2(b) may be promoted so long as its execution under conditions excluded by its guard predicate does not alter program outcome. Here, for example, `r27` is dead when `p3` is false, so instruction 12 may be promoted fully by removing its predicate. This process improves scheduling freedom by easing predicate dependences and can additionally render some predicates unused as guards, improving opportunities for optimization of the control logic.

Figure 2(b) shows the predication we intend to optimize, along with a conceptual (reduced) view of the control logic it can be understood as implementing. The techniques described in this article extract the control logic from predicate defining instructions in an analysis phase, optimize the logic, and regenerate the logic as new, more efficient, predicate defining instruc-



(a) Original control flow graph

(b) After if−conversion and promotion, showing conceptualized program decision logic view

Fig. 2. A portion of the inner loop of the UNIX utility *wc*.

tions. While the predicate optimization technique described is but a single application of the analysis methodology we present, it is one that demonstrates well the generality and power of the abstracted representation. Using the *wc* example, we present an overview of this process.

B. Analysis techniques

The program decision logic must first be extracted into the Boolean representation before optimization can be performed. In our system, this is the function of the predicate analyzer, which applies the semantics of Figure 1 to form the logic circuit of the program's "control path." The example shown in Figure 3 demonstrates this technique on the predicate expressions of *wc*. Figure 3(a) shows the predicate defines from Figure 2(b). The Boolean circuit of Figure 3(b) is derived by applying the semantics of the predicate define types, as shown in Figure 1, to these defines. Three predicates, p3, p5, and p6, are used to guard computation other than predicate defines, and are therefore considered "essential." These are the logical values that must be generated by the new network; all other internal values can be modified arbitrarily to achieve a more efficient computation. Back-substituting into the circuit, expressions in terms of the fundamental conditions are derived: $\mathtt{p3} = C_2\overline{C_1}\ \overline{C_0}$, $\mathtt{p6} = \overline{C_3}(C_0+\overline{C_0}C_1)$, and $\mathtt{p5} = \overline{C_3}(C_0+\overline{C_0}C_1)+\overline{C_4}C_3(C_0+\overline{C_0}C_1)+\overline{C_5}C_4C_3(C_0+\overline{C_0}C_1)$. These expressions represent the original control component of the program and are the starting point for optimization.

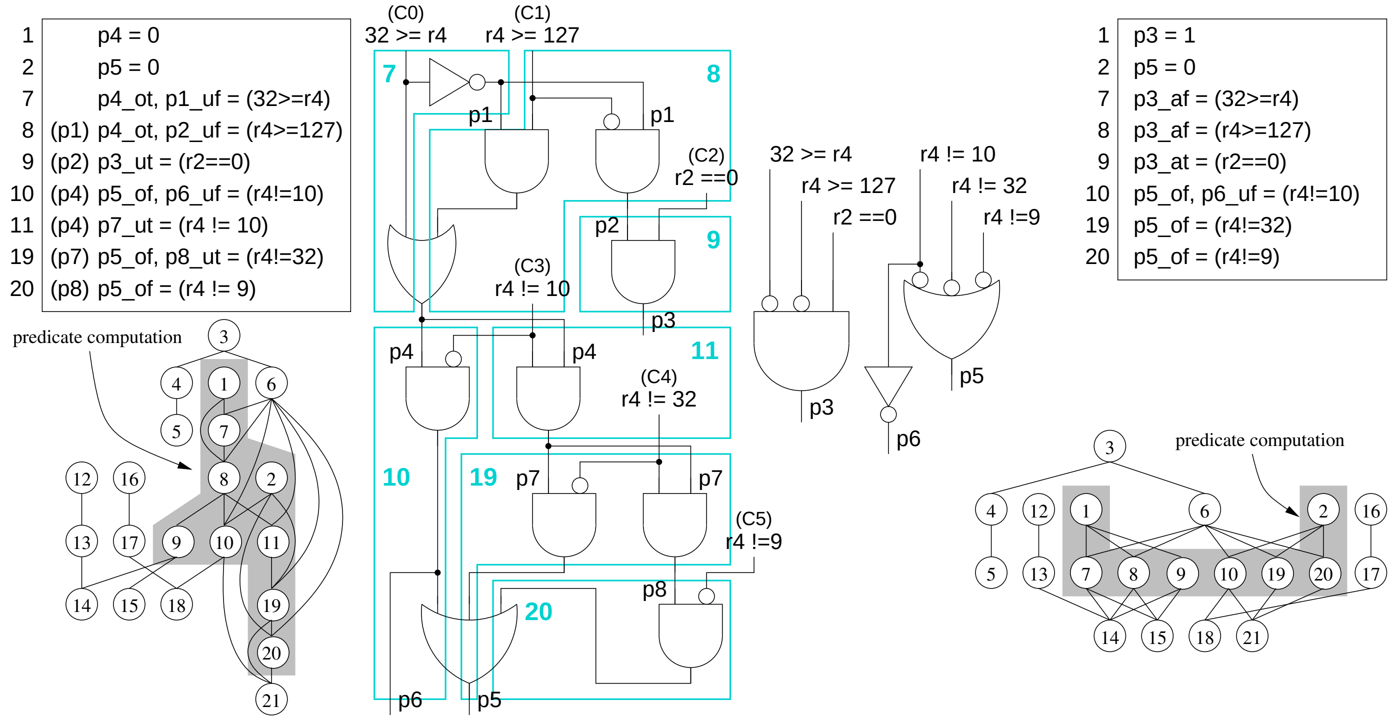
These expressions, however, are missing an element of information critical to successful optimization. The variables of these expressions are in many cases not logically independent. For example, since C_0 represents the evaluation of `(32 >= r4)` and C_3 the evaluation of `(r4 != 10)`, it is clear that $\overline{C_3}$ implies C_0 and that $\overline{C_0}$ excludes $\overline{C_3}$. This and similar relations among conditions defined based on the same source operands can be used to further reduce the decision logic, as redundant tests may be removed. Supposing the the predicate expression optimization phase gives us the expression $(\overline{C_3}+\overline{C_4}+\overline{C_5})(C_0+C_1)$ for p5, the condition database makes the observation that $C_0 = (\mathtt{32 >= r4})$ is *implied* by $\overline{C_3} = (\mathtt{r4 == 10})$, $\overline{C_4} = (\mathtt{r4 == 32})$, and $\overline{C_5} = (\mathtt{r4 == 9})$ and therefore p5 may be simplified to $\overline{C_3}+\overline{C_4}+\overline{C_5}$. In this manner, analysis of conditions allows removal of predicate computation that is redundant with inherent relationships among the condition tests.

In this work, Reduced Ordered Binary Decision Diagrams (ROBDD) [10] are used as the Boolean representational mechanism for both predicate and condition relations. Section III details this mechanism.

C. Define optimization

The expressions describing the evaluation of the essential predicates are optimized using standard Boolean minimization techniques which eliminate redundant terms and which re-express the Boolean function in a more parallel form. Optimization to minimal sum-of-products form yields the expressions: $\mathtt{p3} = C_2\overline{C_1}\ \overline{C_0}$, $\mathtt{p6} = C_0\overline{C_3}+\overline{C_0}C_1\overline{C_3}$, and $\mathtt{p5} = \overline{C_3}C_0+\overline{C_3}C_1+\overline{C_4}C_0+\overline{C_4}C_1+\overline{C_5}C_0+\overline{C_5}C_1$. As indicated previously, optimization based on condition analysis further reduces the expressions for p5 and p6 to $\overline{C_3}+\overline{C_4}+\overline{C_5}$ and $\overline{C_3}$, respectively, producing the logical form shown in Figure 3(c). The number of logic gates in the circuit implementation is reduced from ten to three. In addition, the six-level gate network is reduced to a single-level gate network.

The optimizer significantly reduces the program logic, but only in its abstracted form. To capitalize on the gains of logic optimization, the reformulator must moderate its efforts to reduce predicate dependence height based on surrounding dependences and machine resource constraints. As in many ILP optimizations, reducing dependence height often involves increas-



(a) Original defines and dependence graph (b) Original logic equivalent (c) Optimized logic (d) Optimized defines and dependence graph

Fig. 3. Overview of *wc* optimization

ing the number of instructions issued. If slack exists in the predicate define dependence chains (that is, if concurrent dependences constrain execution such that reducing the dependence height of the predicate expression does not effect a speedup), a more parallel expression of the predicate computation simply increases code size. Furthermore, if the increased instruction issue resulting from the more parallel expression exceeds the issue capabilities of the machine, performance can be hurt rather than helped. These considerations make the design of the reformulator nontrivial.

Figure 3(d) shows the reformulated predicate defining instructions. All non-essential predicates were eliminated as part of this process. The reformulator generated six parallel predicate defining instructions to replace the seven sequential predicate defining instructions originally used. The overall effectiveness of the program decision logic minimization process on the *wc* example is best shown by comparing the dependence graphs of the code before and after optimization. Figure 3(a) and (d) show the dependence graphs for the example hyperblock before and after optimization. The chain of predicate define instructions in the original hyperblock, a five-level network which dominated the critical path, is replaced by a parallel, single-cycle computation in the optimized hyperblock. The highlighted instructions correspond to the predicate defines in each hyperblock. On a six-wide, general-issue, unit-latency processor, this means a reduction in schedule height from eight cycles to four.

D. Post-optimization considerations

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 [11]. 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 [12].

Depending on the compilation model in which predicate optimization is performed, it may be necessary to perform predicate analysis subsequent to the optimization. It is desirable, therefore, to ensure that predicate analysis performed on optimized code is as accurate as analysis performed on unoptimized code. This enables incremental compilation and re-compilation without requiring the compiler to save any analysis results alongside the intermediate form. Returning to the example of Figure 3, consider the forms of p5 optimized with and without condition information. Without examining condition relations, p5 was optimized to a form that was simpler than, but still logically equivalent to, its original expression. From these expressions, it is clear that p5 $\equiv (\overline{C_3} + \overline{C_4} + \overline{C_5})(C_0 + C_1)$ and p3 $\equiv C_2\overline{C_1}\,\overline{C_0}$ are disjoint. With condition information, however, p5 was reduced to $\overline{C_3} + \overline{C_4} + \overline{C_5}$. Comparing this to the expression of p3 the disjointness is obscured. To avoid loss of accuracy due to predicate optimization, therefore, it is necessary to incorporate condition analysis seamlessly into the predicate analysis framework [13].

The following section describes the abstraction of predicate defines into the desired fully Boolean form using a code example better illustrative of that process; we shall return to the *wc* example in a subsequent section to illustrate the details of the optimization process.

III. THE PREDICATE ANALYSIS SYSTEM

Given that the program has been rendered into the predicated form, the compiler must comprehend the logical relationships among predicates to proceed with optimization. As suggested by Figure 2(b), our approach is to treat the control component in a purely Boolean sense. While it seems straightforward that the predicate definitions could be treated in this way, it necessitates selecting a particular representation for the resulting Boolean expressions. An appropriate representation must be reasonably efficient in space and construction time and must allow frequently used queries to be made efficiently. This section focuses on the design of the Predicate Analysis System and its two modes of interaction with the compiler. In the first mode, analyses of conditions and predicates are conducted separately, to aid in optimization of predicates using conditions as the logical literals. In the second mode, analysis of predicates and conditions is unified to provide accurate results after condition-aware predicate optimization. We begin by considering the important characteristics of the underlying Boolean representation.

A. *Efficient representation of program control*

Logical expressions relating predicates and conditions are provided by the Predicate Analysis System, a framework which integrates analysis of both predicates and conditions into a common representation [13]. PAS expresses the relations among predicates by defining a set of interrelated Boolean functions. An efficient implementation requires an appropriate representation. In general, a Boolean function $F(x_0, x_1, \ldots, x_n)$ can be represented in a number of forms. The most familiar of these are conjunctive-normal form (sum-of-products or CNF) and disjunctive-normal form (product-of-sums or DNF). The efficiency of queries varies significantly with the form selected. Tautology, the test if an expression is constant-true, and satisfiability, the test if an expression is constant-false, for example, are two important queries in predicate analysis. Unfortunately, such queries are NP-hard in these common Boolean representations. The root of the problem is that it is possible in DNF and CNF for two different expressions to be logically equivalent—these forms lack *canonicity*. A canonical form, one in which a given logical entity has exactly a single representation, is ideal for predicate analysis because tests for tautology, satisfiability, and identity are all rendered *constant time*. In such a form it is obvious, for example, when two predicates with different derivations end up being logically equivalent, or when a predicate works out to be constant-false. Because in a canonical form this property is always maintained, "automatically," a canonical form is ideal for the predicate relation database. The cost of using a canonical form is that the size of the representation is provably exponential in the number of variables in the worst case [10]. For all these reasons, the choice of a representation appropriate to the expressions being analyzed is critical.

B. *Reduced ordered binary decision diagrams*

The reduced-ordered binary decision diagram (ROBDD) forms the underlying Boolean representation for PAS. ROBDD are a fully canonical representation based on the *if-then-else normal form* (INF). Much work has been done in the development of efficient ROBDD implementations, mostly intended for use in the domain of Boolean logic circuit optimization [10]. BDD have also been applied in software problems, usually in the verification domain. The contribution of PAS is an effective mapping of the predicate analysis problem to this efficient representation.

INF constructs Boolean functions using the ternary if-then-else (ITE) operator, where $\mathrm{ITE}(x, Y, Z) \equiv (x \wedge Y) \vee (\overline{x} \wedge Z)$, where x is a BDD variable and Y and Z are other functions (subgraphs) in the BDD. Functions are expressed by recursive decomposition, in the form of a Shannon expansion, using the ITE operator:

$$F(x_0, \ldots, x_n) = x_n F(x_0, \ldots, x_{n-1}, 1) \vee \overline{x_n} F(x_0, \ldots, x_{n-1}, 0)$$
$$\Downarrow \mathrm{ITE}(x, Y, Z) \equiv xY \vee \overline{x}Z$$
$$F(x_0, \ldots, x_n) = \mathrm{ITE}(x_n, F_1(x_0, \ldots, x_{n-1}), F_0(x_0, \ldots, x_{n-1}))$$

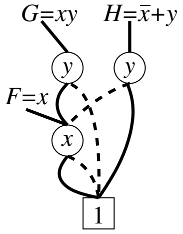
Here, considering the common graph representation, two sub-BDD, F_1 and F_0, are connected as the then- and else- decisions of an ITE node labeled with the variable x_n, and the function F is represented by a reference to this decision node. The whole graph is rooted with the logical constants 0 and 1. A system of INF expressions in which all equal subexpressions are shared is termed a *binary decision diagram* (BDD). A BDD in which all identical ITE nodes are shared, in which variables appear in the same order and at most once in any path from root to leaf, and in which no redundant tests are performed is termed a *reduced ordered binary decision diagram* (ROBDD). Such BDD are canonical: each derivation of a particular Boolean function arrives at the same graph representation; that is, any two equal expressions share the same subgraph. Certain queries are thus vastly simplified; for example, it is possible to test if two given functions are identical or opposite in constant time. This is useful especially for testing if a function evaluates to the constant 0 or the constant 1.

PAS uses the Colorado University Decision Diagram (CUDD) implementation of ROBDD [14]. CUDD implements "invert" arcs, which can be used in the place of "else" arcs to implement the alternate "if-then-invert" construct:

$$F(x_0, \ldots, x_n) = x_n F_1(x_0, \ldots, x_{n-1}) \vee \overline{x_n} \overline{F_0(x_0, \ldots, x_{n-1})}$$

Here we observe the same recursive formulation as before, but the formula inverts an existing subgraph (F_0) for use as an "else" subexpression of F without making additional nodes. Represented by an "invert" arc in the graph, this extension allows for constant-time inversion and avoids the addition of extra internal nodes when the complement of an existing subgraph is required. Now only the constant 1 is provided; evaluations to 0 are made via invert-arcs. CUDD ensures canonicity and the optimal reuse of subexpressions by imposing rules on the use of invert-arcs and by using a node hashing table called the *computed-table*, respectively[15]. Notationally, we speak of the BDD as constructed of ITE nodes, where a given ITE node possesses, in addition to its outgoing "then" arc, either an outgoing "else" arc, in which case it implements the first semantic, or an outgoing "invert" arc, in which case it implements the alternate semantic.

Initially, the BDD consists only of the node 1. An interface exists to add new variables x_i to the BDD, each of which is created as a single ITE node with a then-arc and an invert-arc to 1.



computed–table

(x,1,0)	F
(y,1,$\overline{F}$)	H
(y,F,0)	G

$I = FH$

(1) $I = \text{ITE}(F, H, 0)$
$top(F, H, 0) = y$

(2) $I = \text{ITE}(y, \text{ITE}(F|_y, H|_y, 0), \text{ITE}(F|_{\overline{y}}, H|_{\overline{y}}, 0))$
$F|_y = F,\ F|_{\overline{y}} = F,\ H|_y = 1,\ H|_{\overline{y}} = \overline{F}$

(3) $I = \text{ITE}(y, \text{ITE}(F, 1, 0), \text{ITE}(F, \overline{F}, 0))$
$\text{ITE}(\mathcal{F}, 1, 0) \equiv \mathcal{F},\ \text{ITE}(\mathcal{F}, \overline{\mathcal{F}}, 0) \equiv 0$

(4) $I = \text{ITE}(y, F, 0)$
computed-table(y, F, 0) = G

(5) $I = G$

then — else — invert

Fig. 4. BDD canonicity example: computing $I = FH$

The order of variable definition determines the subsequent order of the variables from root to leaf in each expression path. The BDD is built using the function `ite(f,g,h)`, which builds a subgraph to compute $\text{ITE}(F, G, H)$, where F, G, and H are all existing functions in the graph. The function checks to see if the requested node is a terminal case (a constant) or, through a hash, if it already exists in the graph; if so, it is returned immediately. If not, the topmost variable x_t of the existing functions F, G, and H is extracted, and then- and else- sub-BDD, which assume $x_t = 1$ and $x_t = 0$, respectively, are recursively computed. A new node containing x_t is formed, and the sub-BDD are connected to it, forming the requested function. The `ite` function automatically maintains graph canonicity [15] and operates in time proportional to the size of the resulting function graph.

A brief example illustrates the construction of the BDD. Figure 4 shows a small BDD containing three named functions representing the indicated Boolean expressions. Let us try to compute $I = FH$. Canonicity requires that this computation result in G, since $FH = x(\overline{x} + y) = x\overline{x} + xy = xy = G$. In step 1, the generation of I is requested using the ITE construct; subsequent steps detail the recursive descent which finds the resulting representation. The top variable among F, H, and 0 is y, so step 2 partitions the expression on y. F is insensitive to y, which is not its top variable, so it remains unaltered. Following the arcs from H, on the other hand, $H|_y$, "*H given y*," resolves to 1 and $H|_{\overline{y}}$ resolves to $\overline{F}$. In step 3, the indicated axioms reduce $\text{ITE}(F, 1, 0)$ to F and $\text{ITE}(F, \overline{F}, 0)$ to 0. Thus, by step 4 we have arrived at the expression $\text{ITE}(y, F, 0)$. This expression cannot be trivially reduced, so the BDD hashes into the computed-table and finds that it already exists as G. Thus the recursion is complete and I has been discovered to be identical to the existing function G, as desired.

C. Mapping predicate defines to the BDD

BDD are well known for and apply naturally to Boolean logic manipulation; the unique contribution of PAS is the mapping of the predicate analysis problem to this efficient substrate. Figure 5 shows an example hyperblock, for which the PAS BDD will be constructed. The source code in (a) is translated to the intermediate representation, if-converted and scheduled in (b). Solid lines in the figure indicate the break between cycles in the schedule. Predicate analysis is significant for a code segment such as this for reasons beyond predicate optimization. Here, for example, predicate analysis informs the scheduler that the predicates on `stmtB` (`p4`) and `stmtC` (`p2`) are mutually exclusive; thus, these instructions may be reordered freely. Since the logical relations among predicates change infrequently but are queried often during the compilation process, it is efficient to construct a database of relations from which common queries may be drawn rapidly.

The code is subjected to predicate optimization and rescheduled using a predicate analysis of the type described here (Figure 5(c)). Optimization removes the guard predicate on the define that computes `p4`. This is legal because the logical expression of the predicate guard `p3`, `(r1>-8 && r1<8)`, is implied by the condition on the instruction, `(r1=0)`. Since the unconditional type define computes the conjunction of the guard with the condition, the guard may safely be eliminated. As in the *wc* example, this has an effect on the ability of future predicate analyses to determine the relation of predicates `p2` and `p4`. In (b), the apparent derivation of `p4` from `p1`, which is non-intersecting with `p2`, indicates that $\texttt{p4} \cap \texttt{p2} = \emptyset$. In (c), however, the predicate analysis needs to examine relations among conditions themselves (and ones more complex than simple recognition of opposites) to reach the same proper conclusion. The following shows how the BDD is constructed to support this analysis.

D. Construction of the condition layer

The first step of finding relations among predicates is the definition of relations among condition evaluations. In PAS, these relations are represented together with predicate information in the BDD by providing a set of *condition nodes* [16]. PAS incorporates arbitrary relations within families of conditions based on comparing the same register values, representing, for example, the exclusivity of `(r1=1)` and `(r1=2)` while indicating that both are subsets of `(r1>0)`. A family is initially represented as a single interval containing all representable numbers. For each condition that depends on the same register value, the number line is split at the boundaries of the intervals of numbers yielding an evaluation to "true." The number line in Figure 6(a) represents the condition family of `r1` in Figure 5. The set of values causing any condition to evaluate to "true" is represented as the union of disjoint intervals. The relations among all possible outcomes on this family are represented in the BDD by creating a Boolean space, known as a *finite domain* [17], and assigning all intervals to mutually exclusive and collectively exhaustive expressions, meaning that any assignment of the variables causes exactly one of the expressions to evaluate to 1. The expressions must be mutually exclusive, as each value belongs to only one interval, and also collectively exhaustive, so that an expression such as "`(x>5) || (x<=5)`" is recognized as a tautology.

Figures 6(a) through (d) show the construction of the condition layer for the conditions of Figure 5(c). In this case, the conditions divide the number line into five discrete segments, broken between -8 and -7, -1 and 0, 0 and 1, and 7 and 8. The finite domain technique is applied, using $\lceil \log_2(5) \rceil = 3$ BDD variables to create an eight-element Boolean space $\{(v_2, v_1, v_0) \in (0|1)^3\}$. Since this case requires exactly five elements, we merge the three extra elements to neighbors, forming three two-variable expressions and two three-variable expressions to implement the finite domain. In general, representing i intervals

```
if (x > -8 && x < 8) {
   stmtA;
   if (x == 0)
      stmtB;
}
else {
   stmtC;
   if (x < 0)
      stmtD;
}
```

(a)

```
          p2_uf = (0=0)
     ------------------------
          p1_ut,p2_of = (r1>-8)
     ------------------------
(p1) p3_ut,p2_of = (r1<8)
     ------------------------
(p3) p4_ut = (r1=0)
(p2) p5_ut = (r1<0)
(p3) stmtA
(p2) stmtC
     ------------------------
(p4) stmtB
(p5) stmtD
```

(b)

```
     p1_ut,p2_uf = (0=0)
     ------------------------
     p1_at,p2_of = (r1>-8)
     p1_at,p2_of = (r1<8)
     p4_ut = (r1=0)
     p5_ut = (r1<=-8)
     ------------------------
(p1) stmtA
(p4) stmtB
(p2) stmtC
(p5) stmtD
```

(c)

Fig. 5. An example hyperblock: (a) source, (b) if-converted, (c) optimized.

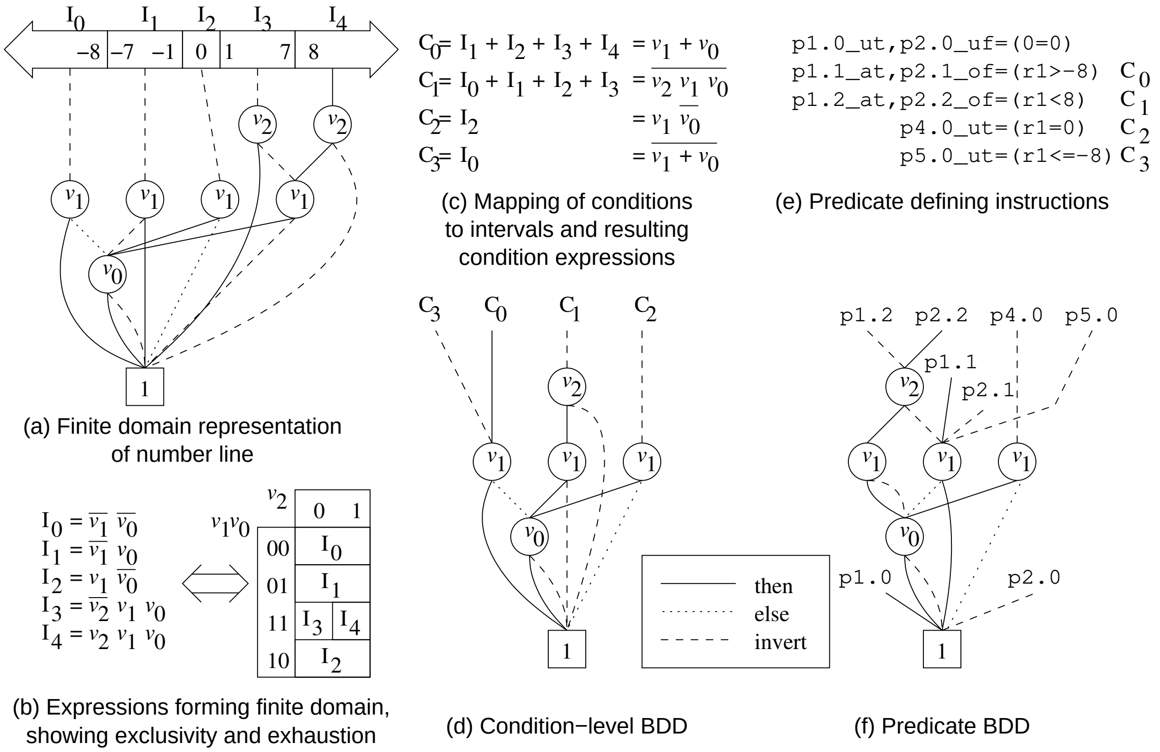


Fig. 6. Assembly of Predicate BDD for the example of Figure 5(c).

adds $n = \lceil \log_2(i) \rceil$ variables, and generates $2^n - i$ expressions in $n-1$ variables and $2i - 2^n$ expressions in n variables. This procedure creates the simplest possible finite domain structure for the given number of elements. In the resulting BDD, shown in (a), each segment of the number line is represented by a BDD node; for example, I_0 represents the expression $\overline{v_1}\ \overline{v_0}$ and thus has the equivalent (canonical) BDD expression $\overline{\mathrm{ITE}(v_1, 1, \mathrm{ITE}(v_0, 1, \overline{1}))}$ as shown. In this and all BDD expressions, the basis variables appear in a fixed order in all paths from root to leaf. The rest of the expressions represented in the BDD are shown in (b), along with a Karnaugh map showing the expressions to be mutually exclusive of each other and collectively exhaustive of the Boolean 3-space, as desired.

Applying the interval composition of the conditions (c), the interval nodes are used together with the ITE operator to compose the condition nodes shown in Figure 6(d). A condition, such as `(r1<8)`, is represented by the disjunction of the interval nodes which represent the set of values resulting in an evaluation to 1. Considering the condition C_0, `(r1>-8)`, we see that $C_0 = I_1 + I_2 + I_3 + I_4$. Thus, $C_0 = \overline{v_1}v_0 + v_1\overline{v_0} + \overline{v_2}v_1v_0 + v_2v_1v_0 = v_0 + v_1$, represented in the BDD as $\mathrm{ITE}(v_1, 1, \mathrm{ITE}(v_0, 1, \overline{1}))$. This simplified expression is computed automatically in the BDD as the disjunction representing C_0 is formed, one ITE at a time. Figure 6(d) shows the resulting BDD, in which all relationships among conditions in a family are represented. For example, the expressions for C_0 `(r1>-8)`, $v_0 + v_1$, and C_2 `(r1=0)`, $v_1\overline{v_0}$, show that C_2 implies C_0. This process is described in detail in [16].

This condition layer is used in the optimization phase to determine the relations among condition variables. In this capacity it is separate from the predicate BDD; in later analyses, however, it serves as the foundation for the predicate layer.

E. Construction of the predicate layer

The mapping of predicate defines to the BDD is somewhat more straightforward than the mapping of conditions. For pur-

TABLE I
PREDICATE DEPOSIT LOGIC.

SSA pred. def.	*ITE* Expression
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_ut = C	$n_{i.j} = C?n_g : 0$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_uf = C	$n_{i.j} = C?0 : n_g$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_ot = C	$n_{i.j} = C?(n_g?1 : n_{i.j-1}) : n_{i.j-1}$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_of = C	$n_{i.j} = C?n_{i.j-1} : (n_g?1 : n_{i.j-1})$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_at = C	$n_{i.j} = n_g?(C?n_{i.j-1} : 0) : n_{i.j-1}$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_af = C	$n_{i.j} = n_g?(C?0 : n_{i.j-1}) : n_{i.j-1}$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_ct = C	$n_{i.j} = n_g?C : n_{i.j-1}$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_cf = C	$n_{i.j} = n_g?\overline{C} : n_{i.j-1}$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_$\vee$t = C	$n_{i.j} = C?1 : (n_g?1 : n_{i.j-1})$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_$\vee$f = C	$n_{i.j} = C?(n_g?1 : n_{i.j-1}) : 1$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_$\wedge$t = C	$n_{i.j} = n_g?(C?n_{i.j-1} : 0) : 0$
($\mathtt{p}_g$) $\mathtt{p}_{i.j}$_$\wedge$f = C	$n_{i.j} = n_g?(C?0 : n_{i.j-1}) : 0$

poses of the optimizer, predicate uses are constrained to have a single forward-control-flow-reaching definition. This means that all predicate value flows are within the acyclic-rendered control flow graph (i.e. there are no flows around backedges) and that each use has a single reaching definition; that is, a single static assignment view (SSA) contains no ϕ functions[18]. In this setting, predicates are given single static assignment tags. The predicate graph is constructed in a single topological traversal of the control flow graph by adding at each predicate define a new expression according to Table I. In the table, $x\ ?\ y\ :\ z \equiv \mathrm{ITE}(x, y, z)$ and $n_{i.j}$ represents the BDD node associated with the predicate $\mathtt{p}i.j$ (j represents the SSA subscript). As indicated in the table, a new subgraph representing the defined predicate is generated from existing subgraphs representing the predicate source, the previous value of the predicate destination, and the condition. The forward-flow constraint guarantees that these expressions are available when required.

For purposes of predicate optimization, in the construction of the predicate BDD each condition is treated as an independent BDD variable; in subsequent analyses, the condition family sub-BDD composed using the condition analysis are incorporated directly into the predicate BDD.

Returning to the example of Figure 5, we construct the corresponding local relation BDD. A BDD expression has already been defined for each condition, as shown in Figure 6(d), and the predicate defines of Figure 5(c) have been converted to the SSA form shown in Figure 6(e). In the following topological traversal, predicate define instruction semantics are applied to generate the form shown in Figure 6(f), which expresses the relations among all predicate definitions. Consider the derivation of the predicate `p2`. The first assignment (with SSA name `p2.0`) is an initialization to 0. Thus `p2.0` is attached via an invert-arc to 1, as shown. `p2.1` is an *or-false* type definition with a constant-true guard predicate and condition C_0. Consulting Table I, `p2.1`=$\overline{C_0}$?`p2.0`:(1?1:`p2.0`). Since `p2.0`=0, this degenerate case results in `p2.1` being attached via an "invert" arc to the same node as C_0. Finally, by the or-false expression as before, `p2.2`=$\overline{C_1}$?`p2.1`:(1?1:`p2.1`). The two `ite` calls used to compose this expression provide the node labeled `p2.2`. Figure 6(f) shows the BDD after excess condition nodes are freed (once all predicates are computed); thus nodes such as those for C_1 no longer exist in the graph. The CUDD BDD package employs reference counting to ensure that such nodes are removed when no longer required. A more complex example with multiple comparison families would show several initially independent condition BDD (based on different variables) rooted on the same '1' node. During predicate define processing, graphs would be composed of members of the various subtrees, effectively unifying them into one predicate BDD. The resulting BDD expresses relations among all conditions and predicates.

The techniques presented are capable of faithfully representing relations among acyclically-defined predicates in the assumed SSA model. Both the extension of these techniques to general predicated codes and the size-safety of the BDD for predicate analysis have been described in other work [13].

IV. Minimization of program decision logic

With the program decision logic and condition relations abstracted into a Boolean form, optimization can proceed. Progressing from decision logic to new predicate define instructions encompasses several steps. First, sum-of-products expressions are formed to represent predicate functions in terms of program conditions. Knowledge of condition interrelation is applied to these expressions to remove redundant literals and redundant or constant-false terms. Traditional Boolean logic minimization techniques are then used to simplify the expressions to a reduced two-level form. The resulting optimized expressions are then factored into a multi-level form based on condition availability times and resource constraints. Finally, program control is re-expressed in predicate define instructions. We return in this section to the *wc* example begun in Figure 3.

The BDD maintains a canonical representation of the decision logic functions, from which Boolean sums-of-products can easily be produced. The generated expressions reflect the canonical nature of the BDD, and are usually not optimal for expressions of multiple terms. Therefore, reduction of the expressions is almost always required before re-expression becomes a realistic option.

Figure 7(a) shows the original *wc* predicate define structure. Figures 7(b) and 7(c) show the result of the analyses described in the previous section. In the figure, condition relations are expressed in disjunctive form for understanding; in the implementation these expressions remain embodied in the BDD representation. The predicate expressions shown in Figure 7(c), on the other hand, are extracted by a straightforward technique from the BDD and expressed in disjunctive form. It is these expressions that will be optimized using the condition relation database and traditional Boolean optimization techniques.

A. Condition-based optimization

Figure 7(b) shows the expressions represented in the BDD for the conditions of the *wc* example. As indicated in the previous section, condition families have been formed to relate the conditions to each other. Here two families are indicated. The first, based on comparisons to register `r4`, relates conditions C_0, C_1, C_3, C_4, and C_5 using the BDD variables v_0, v_1, and v_2. The second family, containing only the single condition C_2, requires only the single BDD variable v_3. These expressions embody all logical interrelation among the conditions. For example, the condition $\overline{C}_4$, true when `r4 == 32` clearly implies C_0, true

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

(a) Original predicate define structure

Variable	Condition	Expression
C_0	(32>=r4)	$\overline{v_2}+\overline{v_1}$
C_1	(r4>=127)	$v_2 v_1 v_0$
C_2	(r2==0)	v_3
C_3	(r4!=10)	$v_2+\overline{v_1}+\overline{v_0}$
C_4	(r4!=32)	$\overline{v_2}+v_1+\overline{v_0}$
C_5	(r4!=9)	$v_2+\overline{v_1}+v_0$

(b) Conditions and condition expressions

	Extracted	Cond. opti.	*espresso*
$p3$	$\overline{C}_0\overline{C}_1C_2$	$\overline{C}_0\overline{C}_1C_2$	$\overline{C}_0\overline{C}_1C_2$
$p6$	$C_0\overline{C}_3+\overline{C}_0C_1\overline{C}_3$	$\overline{C}_3$	$\overline{C}_3$
$p5$	$C_0\overline{C}_3+\overline{C}_0C_1\overline{C}_3+$ $C_0C_3\overline{C}_4+\overline{C}_0C_1C_3\overline{C}_4+$ $C_0C_3C_4\overline{C}_5+\overline{C}_0C_1C_3C_4\overline{C}_5$	$\overline{C}_3+$ $C_3\overline{C}_4+$ $C_3C_4\overline{C}_5$	$\overline{C}_3+\overline{C}_4+\overline{C}_5$

(c) Optimization of 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) Regenerated predicate define structure

Fig. 7. Analysis and optimization of *wc*

when `r4 >= 32`. The expressions hold the same relation to each other: $\overline{\overline{v_2}+v_1+\overline{v_0}} = v_2\overline{v_1}v_0$ implies $\overline{v_2}+\overline{v_1}$. The composition of these expressions therefore exposes constant-false and redundant minterms and literals among the predicate expressions. Three types of optimizations are performed: (1) removal of redundant literals within a term; (2) removal of redundant terms; and (3) removal of constant-false terms. Figure 7(c) shows the result of "filtering" the extracted terms using the condition relations. Consider the optimization of `p6`. Its first term, $C_0\overline{C}_3$, is reduced to $\overline{C}_3$ because of the implication between the two variables. When the expression for the second term $\overline{C}_0C_1\overline{C}_3$ is composed, this term is discovered to be constant-false and is removed: $\overline{C}_0C_1\overline{C}_3 = v_2v_1v_2v_1v_0\overline{v_2}v_1\overline{v_0} = 0$. The example's predicate expressions are thus reduced using condition information to the form shown in Figure 7(c).

The BDD lends itself naturally to this optimization process due to its strong canonicity. For example, to determine if a term is constant-false, it is necessary only to compose the term from functions available in the BDD. If the term is unsatisfiable (that is, if the literals present are mutually exclusive) the BDD will generate a reference to 0 as the formula for the term.

B. Predicate expression optimization

After condition optimization, the predicate expressions still exhibit two types of redundancy: that present in the original sequences of define instructions and that generated by the extraction of the disjunctive form from the BDD. To address both these inefficiencies, a traditional Boolean optimizer, *espresso*, is applied to the predicate expressions. The optimizer applies the iterative consensus technique to the minterms, reducing the expression to minimal two-level logic. 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 and then removes nonessential products to generate a minimal covering sum [19]. In the *wc* example, the expression for `p5` is reduced by iterative consensus. Figure 7(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.

Predicate expressions can grow very large in heavily predicated code, having in some instances in excess of thirty variables. This renders the consensus algorithm, the complexity of which is in the worst case exponential in the number of variables, intolerably slow in some cases. Fortunately, the structure of the expressions extracted from the BDD lends itself to a divide-and-conquer optimization approach which drastically reduces computation time. Terms generated from the BDD are an irredundant form, in which no two minterms overlap. Adjacent terms, however, are often easily reduced by consensus. Thus an unmanageably large *espresso* input can be piecewise-preprocessed in more manageable chunks, drastically reducing computation time without sacrificing optimization. By this method we have been able to optimize even the largest predicate expressions resulting from the benchmarks studied.

C. 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 $\texttt{p}i\texttt{\_at} = C_i$, where one predicate $\texttt{p}i$ is defined for each product term in the predicate expression. The second level consists of *or* type predicate defines of the form $(\texttt{p}i)\texttt{p}j\texttt{\_ot} = (condT)$, where there is one such predicate define for each product $(\texttt{p}i)$ 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 7(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 many cases, however, the sum-of-products form is far larger than the optimal unconstrained network. 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. These deficiencies, indicated in the results (Section VI) required development of factorization techniques.

D. Factorization

The *wc* example of the previous section exhibited a high 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 reformulation required fewer predicates and predicate defines than the original form. 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 without increasing the length of any critical path. 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 migrate 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 "slack," or the amount of time the new predicate network has to compute predicates without performance penalty, is ascertained.

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 so that the resulting define structure may take more cycles, but can reuse more intermediate predicates, thus saving predicate defines and predicate registers.

When resource utilization is very high and predicate functions are very complex, factorization becomes critical for performance. In these situations, an additional factorization preprocessing stage is applied, in which predicates are selectively factored on subexpressions of other available essential predicates. This activity 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 8 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

$$\texttt{p1} = C_0C_2C_4\overline{C}_5 + C_0C_2C_3\overline{C}_5 + C_0C_1\overline{C}_5$$
$$\texttt{p2} = C_0C_2C_4\overline{C}_5C_6 + C_0C_2C_3\overline{C}_5C_6 + C_0C_1\overline{C}_5C_6$$

(a) Optimized predicate expressions

Time	Cond. arrivals	Predicate expression
1	C_0 C_1	p3_ut $= C_0$ p4_at $= C_0$ p4_at $= C_1$
2	C_2	p5_ut $= C_2$ (p3)p6_ut $= C_2$
3	C_3	(p6)p7_ut $= C_3$
4	C_4	(p6)p8_ut $= C_4$
5	C_5	(p7)p1_of $= C_5$ (p8)p1_of $= C_5$ (p4)p1_of $= C_5$
6	C_6	(p1)p2_ut $= C_6$

(b) Factoring with schedule time information

Fig. 8. Factorized predicate define optimization.

in Figure 8(a). In the example, when the code segment is scheduled by the slack estimation technique described, predicate p1 is required in cycle 6 and p2 is required in cycle 7. The second column of Figure 8(b) indicates at which times the various conditions become available. Given this information, the factorizer generates the sequence of predicate defines shown in the final column of Figure 8(b), which takes advantage of the available slack to factor common subexpressions out of the two functions. In this example, factoring generated in 13 predicate defines a result that required 37 predicate defines in the two-level logic form. Since the dependence height of this sequence is matched to parallel dependences in the code, the factored result is guaranteed to perform at least as well as the two-level form, even on an infinitely wide machine. Furthermore, the useful predicates (p1 and p2) are available a single cycle after the last condition is evaluated, sooner than would have been possible using a two-level synthesis of the predicate expressions. Clearly, resource- and dependence-aware factorization is essential for achieving the benefit of Boolean logic minimization in real programs.

V. Architectural support for synthesis

The 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-PD predicate architecture. For example, in Figure 7(c) and (d), a simple sum-of-products expression is converted into a small set of predicate defines.

Synthesis of multi-level factored functions is not as simple as that of sum-of-products expressions, but yields significant improvements in both performance and predicate define count. Factoring involves combining available subexpressions, stored in predicates, into new predicates. After factoring, expressions to be synthesized thus contain predicates as well as conditions. Figure 9 shows an example of factoring. In Figure 9(a), pred-

p1 $= C_1 + C_2$
p2 $= C_0\, C_1\, C_3 + C_0\, C_2\, C_3$

Condition Availability	(a) Factored generation	(b) "Two–level" generation	(c) Factored generation without conjunctive types
C_0	p2_/\t= C_0	p3_at,p4_at= C_0	p2_at= C_0
$C_1\, C_2$	p1_ot= C_1 p1_ot= C_2	p3_at,p1_ot= C_1 p4_at,p1_ot= C_2	p1_ot,p3_af= C_1 p1_ot,p3_af= C_2
C_3	(p1)p2_/\t= C_3	p3_at,p4_at= C_3	p2_at= C_3 (p3)p2_af= 1
		(p3)p2_ot= 1 (p4)p2_ot= 1	

Fig. 9. Various methods of predicate expression regeneration

icate p1 is a subexpression of p2. Factoring $C_1 + C_2$, or p1, out of p2 is desirable because both C_1 and C_2 are available earlier than other variables in p2 and because it allows exploitation of redundancy between p2's two terms. As can be seen in Figure 9(b), this subexpression can be computed in cycle 1 using *or* type predicate defines. The subsequent computation of p2 can incorporate this subexpression beneficially if it can conjoin a predicate with its own previous value and the condition C_3 in cycle 3. HPL-PD semantics do not provide for the parallel conjunction of multiple predicates and conditions. The addition of the *conjunctive* type, however, provides a predicate define suited to this task. Figure 9(c) shows the final, factored predicate define formulation. The two expressions are computed using two predicates and four predicate defines. The last predicate define conjoins p1 and C_3 with the previous contents of p2 (C_0) to finish the computation of the p2 expression.

The primary use of the *conjunctive* type predicate defines is to reduce the number of instructions required to compute factored expressions. Figures 9(b) and 9(c) show two generation options that do not use the *conjunctive* type. In Figure 9(b), the sum-of-products is generated directly, without factoring. In this case, six predicate defines are required and dependence height is increased. In Figure 9(c), factorization is performed, but the *conjunctive* type is not used. Here, five predicate defines are used, partially to construct the complement of the factored expression using DeMorgan's theorem. The predicate p2 is nullified by the final predicate define if this complement is true. In general, it should be noted that not all architectures allow arbitrary pairs of predicate destination types on single predicate defines, and not all expressions are so easily complemented as the simple disjunction treated here. Thus, the cost of factorization in the absence of special predicate destination types can be significant.

VI. Experimental results

To determine the effectiveness of the presented Boolean minimization techniques, particularly due to the sensitivity of performance to the success of factorization heuristics, it was necessary to implement the proposed techniques in a state-of-the-art ILP compiler and to test the results on a processor with aggressive issue capabilities. The described system was thus implemented within the IMPACT experimental compiler / simulator framework.

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 may be a branch. Instruction latencies match those of the HP PA-7100 microprocessor. The instruction set contains non-trapping versions of all potentially excepting instructions, with the exception of branch and store instructions, to support aggressive compiler-directed speculative execution without generation of recovery code. The instruction set also contains support for predicated execution as described in Section II.

The IMPACT emulation-driven simulator assessed the execution time for each benchmark according to the static latencies described in the processor model. In the interest of minimizing experimental noise, some dynamic effects such as branch mispredictions, cache misses, and TLB misses were not measured. Since 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 generally 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*. The best code generated by the IMPACT compiler for the selected architecture using traditional hyperblock compilation techniques was selected as the baseline for performance comparisons, again in an attempt to isolate the particular effects of the predicate optimizations. To this was compared the code transformed using the described Boolean reduction, factorization, and regeneration techniques. Following the regeneration step, an additional scheduling pass was performed to allow the code to benefit from the shortening of critical paths in predicate optimization. The presentation of results begins with a discussion of performance at the benchmark and function levels. An examination of some of the details such as factorization strategies and the new predicate define types follows the presentation of overall performance results.

Figure 10 shows whole-benchmark speedups relative to the hyperblock baseline, as measured by simulation. For each benchmark, two results are reported. The first, the light bar in the graph, 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 ex-

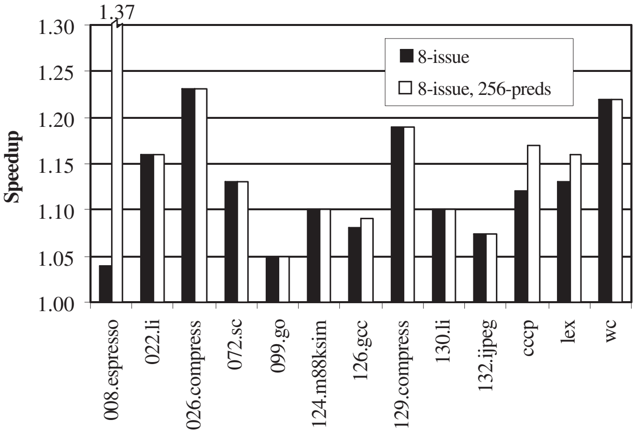


Fig. 10. Speedup from minimization of program decision logic.

ecuted portions of the code; when this height is reduced by our techniques, speedups of around 1.2 were achieved.

The second result presented for each benchmark, represented by the dark bar and labeled "8-issue, 256-preds," is the speedup on a hypothetical machine capable of issuing eight non-predicate-define instructions and, in addition, 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. The benchmark *008.espresso* is affected severely due to its being very decision-height-limited. Unfortunately, the current minimization heuristic chose to reduce height too aggressively in *008.espresso*, resulting in code which excelled on an infinite-issue machine but which over-saturated the target architecture. 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 whole-benchmark results are encouraging. In most cases, the described techniques succeeded in matching control dependence height to surrounding constraints, effectively hiding the cost of predicate computation and significantly improving performance. During our experimental exploration, we observed that as optimizations targeting 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 importance of control optimization in the general scheme of ILP optimization can be expected to increase.

To better understand the effects of program decision logic minimization apart from the averaging tendencies of whole-program results, we measured the performance and code size characteristics of a number of selected functions. Table II 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 regeneration, two-level predicate synthesis and factorization-based synthesis, by indicating for each strategy the number of static predicate define instructions generated, the performance gain on an 8-issue processor with unconstrained predicate define resources (∞), and the performance gain on the 8-issue processor. In addition, the table indicates the static number of predicate define instructions in the code prior to minimization.

As indicated in the table, two-level synthesis shows mixed results. For the unconstrained machine, height reduction produces large speedups. However, the unconstrained performance does not translate directly into the same performance gain on the 8-issue processor. This effect 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 increase in the number of predicate define instructions. The predicate defines created over-saturate processor resources and result in loss of performance. Correspondingly, when the number of predicate defines is not unduly increased, the unconstrained performance does indeed translate directly into performance on the 8-issue processor. Clearly, factored synthesis is necessary for practical optimization of program decision logic.

These results show the factored approach yielding both larger and more consistent speedups. Both methods reduce the predicate computation height, but the factored approach dramatically reduces the number of predicate defines relative to the number generated by the two-level approach. The function *126.gcc, canon_hash* provides a good example of this behavior. Both methods achieve respectable speedups on the unconstrained processor. However, the two-level synthesis approach requires 149 predicate defines to accomplish the improvement. On the 8-issue processor, resource constraints caused by this instruction bloat consume most of the gains that were made available by height reduction. The factored approach reduces the number of predicate defines to 116, increasing the 8-issue speedup to 1.74. Although this constitutes a significant increase relative to the original 89 predicate defines, the net effect is improved performance. The Boolean minimization/factorization approach systematically balances instruction count and resource height 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. 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-step 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.

Another set of data examines the effectiveness of the new predicate types (*conjunctive* and *disjunctive*, described in Section II) in the context of Boolean minimization and justifies these proposed architectural extensions. Table III presents the

TABLE II
SPEEDUP AND PREDICATE DEFINE COUNT FOR SELECTED FUNCTIONS.

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

effects of the new predicate define types on the speedup for an 8-issue processor, the dynamic predicate define count, and the static predicate define count. The new 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. When the new predicate define types are removed from the target architecture and the compiler must generate an alternative, code size suffers. 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.

TABLE III
EFFECTS OF *conjunctive* TYPE PREDICATE DEFINES ON SPEEDUP AND INSTRUCTION COUNT.

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%

VII. RELATED 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 control and data flow simultaneously to identify correlations among branches [20][21]. 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 [21], 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 in [20]. In the second category of control flow optimization, branch reordering, the order in which branches are evaluated is changed to reduce the average depth traversed through a network of branches [22].

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 [23]. 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. While the mechanisms introduced herein also reduce control dependence height, they comprise a minimization approach far more general than those previously proposed.

Three general approaches to predicate analysis have been described previously in the literature, two of which apply to hyperblock code with restricted predicate define types. The first and simplest, the *Predicate Hierarchy Graph (PHG)*, was introduced with the IMPACT hyperblock compilation framework [8]. The PHG relates predicates by keeping track of which predicates guarded the definition of each predicate, or of each component term for *or* type expressions. The PHG thus understands only "genealogical" relationships, and is inaccurate when predicates do not fit neatly into a hierarchical graph. The PHG is unable to represent networks which contain *and* type, *conjunctive* type, and *disjunctive* type predicate defines, precluding direct analysis of code generated by control height reduction optimizations such as those presented in [7] and [24].

A second, more sophisticated approach, the *Predicate Query System (PQS)* [25], exists within the Hewlett-Packard Elcor framework. The representational mechanism of PQS, the *partition graph*, can describe accurately only those predicate expressions which can be expressed as logical partitions. (`p2` and `p3` partition `p1` iff $p1{=}p2 \cup p3$ and $p2 \cap p3{=}\emptyset$.) This relation is generally satisfied only for unconditional predicate defines and for *or* type predicate defines with disjoint terms. Thus, although PQS performs a direct analysis of assembly code containing predication, it can accurately represent only predication conforming to the style of if-conversion. In practice, other define types have been represented in the partition graph using conservative approximations or special constructs. Unfortunately, such accommodations pollute the logical relations of the partition graph with artifacts of the order in which defines were processed and of the approximations themselves; while these techniques allow the partition graph to function safely in a predicate-aware dataflow system [26], they are insufficient as a basis for a logic minimization system. Thus, although PQS can accurately analyze the example of Figure 5(b) (disregarding condition information), it cannot accurately analyze the logically equivalent Figure 5(c), which uses types not easily partitioned. The primary advantages of PAS over PQS, therefore, are its ability to perform fully accurate direct analysis of code utilizing any desired predicate defining semantics, and its ability to incorporate knowledge of condition relations into its logical database.

In the third approach, Eichenberger developed a predicate analysis mechanism for use in register-allocating predicated codes. His mechanism collected logical expressions, termed *P-facts*, which related predicates and, in some cases, related conditions. These *P-facts* were evaluated with respect to each other in a symbolic manipulation environment [27]. Eichenberger's results do not indicate the expense involved in applying this technique. For single-hyperblock analysis, this technique is functionally equivalent to the technique proposed in this paper, but in this work the BDD replaces the symbolic framework, demonstrating the same local accuracy at a low cost, and constructs on top of the analysis a powerful optimization engine.

VIII. Conclusion

In this article, we have presented a comprehensive method for optimizing a program's "decision logic." 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-PD 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. Single-function results indicated speedups up to 1.90 in significant functions. 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.

More important than the optimization technique itself is the approach demonstrated in achieving it. A sequence of techniques exposed, abstracted, analyzed, optimized, and replaced a significant program component. Hyperblock formation was used not just to eliminate branches but to define and separate a program control component. The predicate analysis system abstracted both the relationships among condition tests and those among predicates into a fully Boolean form, the ROBDD, in which expressions representative of the program decision logic could be compared and manipulated efficiently. Traditional Boolean optimization techniques were then applied on this abstracted form, performing transformations that would have been difficult to achieve within the context of the instruction set architecture and traditional compiler analyses. Knowledge of resource and dependence limitations was applied to render an efficient implementation of the improved decision logic back into the target ISA. Finally, it was demonstrated that all this can take place without sacrificing analysis accuracy in subsequent compilation steps. The results achieved, combined with the increasing complexity and variety of architectures, encourage the further development of abstraction-based compilation models.

Acknowledgments

The authors would like to thank Jean-Michel Puiatti for his help with our initial work in Boolean optimization and Professor Farid Najm for the insight he has provided into the nature of the BDD. This research has been supported by the National Science Foundation (under grants CCR-9629948 and CCR-0086096), Advanced Micro Devices, Hewlett-Packard, and Intel.

REFERENCES

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

[2] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, no. 1, pp. 12–35, January 1989.

[3] 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*, January 1983, pp. 177–189.

[4] 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*, June 1998, pp. 227–237.

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

[6] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, May 1999.

[7] D. I. August, J. W. Sias, J. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu, "The program decision logic approach to predicated execution," in *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999, pp. 208–219.

[8] 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*, December 1992, pp. 45–54.

[9] S. A. Mahlke, R. E. Hank, J.E. 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*, June 1995, pp. 138–150.

[10] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transaction on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

[11] 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*, June 1993, pp. 290–299.

[12] 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*, December 1997, pp. 92–103.

[13] J. W. Sias, D. I. August, and W. W. Hwu, "Accurate and efficient predicate analysis with binary decision diagrams," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000, pp. 112–123.

[14] F. Somenzi, "CUDD: Colorado University Decision Diagram package, release 2.30," http://vlsi.colorado.edu/∼fabio/CUDD/, 1998.

[15] K. S. Brace, R. R. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proc. of the 27th ACM/IEEE Design Automation Conference*, January 1990, pp. 40–45.

[16] J. W. Sias, "Condition awareness support for predicate analysis and optimization," M.S. thesis, University of Illinois, Urbana, IL, 1999.

[17] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams," Tech. Rep. CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1992.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, October 1991.

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

[20] 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*, June 1997, pp. 146–158.

[21] 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*, June 1995, pp. 55–66.

[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*, June 1998, pp. 130–141.

[23] M. Schlansker and V. Kathail, "Critical path reduction for scalar programs," in *Proceedings of the 28th International Symposium on Microarchitecture*, December 1995, pp. 57–69.

[24] M. S. Schlansker, S. A. Mahlke, and R. Johnson, "Control CPR: A branch height reduction optimization for EPIC architectures," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, May 1999, pp. 155–168.

[25] R. Johnson and M. Schlansker, "Analysis techniques for predicated code," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 100–113.

[26] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker, "Global predicate analysis and its application to register allocation," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 114–125.

[27] A. E. Eichenberger and E. S. Davidson, "Register allocation for predicated code," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 180–191.

Wen-mei Hwu(S'81-M'87-F'98) is Franklin Woeltge Professor at the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. From 1997 to 1999, he served as the chairman of the Computer Engineering Program at the University of Illinois. His research interest is in the area of architecture, implementation, and compilation for high performance computer systems. He is the director of the IMPACT project, which has delivered new compiler and computer architecture technologies to the computer industry since 1987. For his contributions to the areas of compiler optimization and computer architecture, he received the 1993 Eta Kappa Nu Outstanding Young Electrical Engineer Award, the 1994 Xerox Award for Faculty Research, the 1994 University Scholar Award of the University of Illinois, the 1997 Eta Kappa Nu Holmes MacDonald Outstanding Teaching Award, the 1998 ACM SigArch Maurice Wilkes Award and the 1999 ACM Grace Murray Hopper Award. He is an IEEE Fellow. Dr. Hwu received his Ph.D. degree in Computer Science from the University of California, Berkeley.

David August is an Assistant Professor in the Department of Computer Science at Princeton University. His research interests lie in computer architecture and back-end compilation. He directs the Liberty computer architecture research group. The Liberty group is currently developing an open-source cycle-accurate simulator builder and retargetable compiler. Please see http://liberty.cs.princeton.edu for more information. David received his Ph.D. in Electrical and Computer Engineering from the University of Illinois at Urbana-Champaign in 2000. At Illinois, as a member of the IMPACT research compiler group, he invented a complete framework for aggressive predicate analysis and optimization.

John Sias (S '95) received a B.S. in Computer Engineering and an M.S. in Electrical Engineering from the University of Illinois at Urbana-Champaign in 1997 and 1999, respectively. He is currently pursuing his Ph.D. in Electrical Engineering, also at Illinois. His work has been supported by the National Defense Science and Engineering Graduate Program and the IBM Centre for Advanced Studies. He has spent summers at the IBM Toronto Laboratory, aiding in the development of static and dynamic compilation techniques for EPIC architectures. His primary research interests surround the interface between compiler technology and computer architecture.