Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture, pp. 112–123, December 2000.
Copyright ©2000 by The Institute of Electrical and Electronics Engineers, Inc.

Accurate and Efficient Predicate Analysis with Binary Decision Diagrams

John W. Sias Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois
Urbana-Champaign, IL 61801
{sias, hwu}@crhc.uiuc.edu

David I. August
Department of Computer Science
Princeton University
Princeton, NJ 08544
august@cs.princeton.edu

Abstract

Functionality and performance of EPIC architectural features depend on extensive compiler support. Predication, one of these features, promises to reduce control flow overhead and to enhance optimization, provided that compilers can utilize it effectively. Previous work has established the need for accurate, direct predicate analysis and has demonstrated a few useful techniques, but has not provided an efficient, general framework. This paper presents the Predicate Analysis System (PAS), which maps knowledge of predicate and condition relations in general control flow onto a convenient logical substrate, the reduced ordered binary decision diagram. PAS is the first such framework to demonstrate direct, accurate, and efficient analysis of arbitrary condition and predicate define networks in arbitrary control flow.

1. Introduction

The success of EPIC architectures such as IA-64 [1] hinges on the ability of compilers to expose and express instruction-level parallelism. Predication, a key feature of such architectures, improves the efficiency of program control, allows co-execution of instructions from multiple paths, aids in efficient modulo scheduling, and enables other new optimizations. In the *predicated representation*, each operation possesses a Boolean source operand, its *guard predicate*, the value of which determines whether the instruction is executed or nullified. The values of predicates are manipulated by a set of predicate defining instructions. Although *if-conversion*, the process by which branching control is replaced with predicate defining instructions and predicates [2], is typically responsible for most instances of predication, it may also be introduced in hand-written assembly segments or by specialized optimizations (for example, the replacement of Boolean values, originally allocated to general-purpose registers, with predicates). Other work demonstrates the value of a predication-enabling compiler [3, 4, 5, 6, 7, 8].

As the first predicated compilation systems generated predication only by direct if-conversion, analysis systems that could only generate predicate relations from control flow or that could accurately analyze only the forms generated by if-conversion were adequate [5, 9]. Other systems were capable of analyzing arbitrary local use of predication, but relied on potentially expensive symbolic techniques [4]. Two important concerns, however, demand advances in predicate analysis. First, the introduction of predicate optimization techniques [10] and non-if-conversion use of predication pose accuracy, efficiency, and phase ordering problems for existing analysis systems. Second, post-link compilation and re-compilation technologies [11] require accurate analysis and transformation of arbitrary predication. The predicate analysis engine here described supports work in these new areas.

This work presents three chief contributions: First, this technique uses a general Boolean representation framework, the binary decision diagram (BDD), to accommodate arbitrary predicate formulations accurately and efficiently. Second, this mechanism supports analysis of general predication, including loop-carried predication; that is, it is not limited to predication produced by if-conversion. The system is also easily adaptable to new predicate define types, as have become necessary in predicate optimization work. Finally, this mechanism incorporates the analysis of condition relations (i.e. $(\mathtt{r1} = \mathtt{1}) \rightarrow (\mathtt{r1} > \mathtt{0})$) into the same efficient framework, enabling both more extensive optimization of predication and the accurate analysis of optimized predicate networks.

2. Support for predicated compilation

The presented system was developed in the context of the IMPACT research compiler, a retargetable instruction-level parallel compiler used extensively in computer architecture research. The IMPACT low-level machine-independent internal representation, Lcode, and the compiler back-end fully incorporate predication. Generation of predicated code in the IMPACT compiler follows the progression shown in Figure 1. Appropriate regions of code are if-

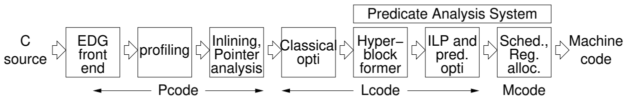


Figure 1: IMPACT predicated code generation path.

Table 1: IMPACT EPIC predicate deposit types.

p_g	C	ut	uf	ot	of	at	af	ct	cf	∨t	∨f	∧t	∧f
0	0	0	0	–	–	–	–	–	–	–	1	0	0
0	1	0	0	–	–	–	–	–	–	1	–	0	0
1	0	0	1	–	1	0	–	0	1	1	1	0	–
1	1	1	0	1	–	–	0	1	0	1	1	–	0

converted in the hyperblock formation phase [3]. Following if-conversion, classical optimizations are reapplied in concert with a full suite of ILP optimizations (unrolling, critical path reduction, etc.) and predicate optimizations [10]. Finally, machine-specific code is generated, scheduled, and register-allocated. In this model, the compiler back-end must be able to understand and to use predication effectively.

The IMPACT EPIC model of predication [7] generalizes Hewlett-Packard PD predication support [12] and subsumes IA-64 predication [1]. This model specifies a set of independent single-bit predicate registers, denoted pi, of which p0 is defined as holding the value 1. Each instruction is augmented with a guard predicate. The predicate defining instruction ($\mathtt{p}_{guard}$) $\mathtt{p}_{dest}$_*type* = src_0 *cmp* src_1 computes the condition $C = src_0\ cmp\ src_1$ and optionally assigns a value to $\mathtt{p}_{dest}$[1] according to $type$, C, and the guarding predicate $\mathtt{p}_{guard}$, as shown in Table 1 (a "–" indicates that the destination register is unchanged). The IMPACT EPIC Architecture specifies six deposit types which dictate how the instruction updates its destination. The (U)nconditional, (O)r, (A)nd, and (C)onditional types are as defined in the HP Labs PlayDoh Specification [12] and, under different names, in the IA-64 architecture [1]. The disjunctive and conjunctive types (∨ and ∧) were developed for use in predicate optimization. Although simple if-conversion generates only unconditional and or-type defines with disjoint subexpressions [2], an effective predicate analysis system should support the other types which are useful in optimization of predicate define networks [10, 13] and modulo scheduling.

2.1. Role and scope of predicate analysis

In predicated codes, while the position of an instruction in the control flow graph specifies the *fetch condition* of an instruction, the actual *execution condition* is a function both of the fetch condition and of the guard predicate. A compiler must thus be modified to analyze code containing predication. For example, the traditional notion of instruction dominance "I_1 *dom* I_2 iff every fetch path from the unique entry node $START$ to I_2 includes I_1 [14]" (expressing fetch dominance, or *fdom*) must be replaced by, "I_1 *edom* I_2 iff I_1 *fdom* I_2 and $\mathtt{p}_{I_1} \supseteq \mathtt{p}_{I_2}$ ($\mathtt{p}_{I_1}$ is true whenever $\mathtt{p}_{I_2}$ is true)." Predicate analysis provides the compiler with answers to Boolean queries such as "Is $\mathtt{p}_{I_1} \supseteq \mathtt{p}_{I_2}$?"

As a simple example, Figure 2 shows two code segments which are considered for application of a constant propagation optimization. In Figure 2(a), constant propagation cannot be applied because instruction 2, the assignment to r3, does not dominate instruction 4, the assignment to r4. In Figure 2(c), however, the first definition dominates the second, so the optimization is valid. After if-conversion of (a) into (b) and (c) into (d), which does not alter program semantics, instruction 2 fetch-dominates instruction 4 in both cases. Predicate analysis allows the compiler to distinguish between a legal (d) and an illegal (b) optimization.

While the execution dominance relationship requires a subset query, other relationships such as equivalence, non-intersection, inverse, etc. are desirable for other purposes. The Predicate Analysis System answers these queries for the rest of the compiler. Dataflow analysis is perhaps the primary consumer of predicate relations. Other works [9] address using the predicate analysis results to formulate dataflow algorithms for use in predicated code.

Before considering the database mechanism, it is important to clarify the idea of relational scope. The simplest example that makes this clear is a single if-then-else construct placed in a loop, where the instructions in each side of the construct are controlled by either predicate p1 or predicate p2. These predicates are set as complements in each loop iteration, based on a varying condition. Are p1 and p2 mutually exclusive? When considering a single iteration of the loop, the answer is clearly "yes;" across iterations, however, the answer is "no." Depending on the application of the analysis, one answer or the other may be desired. Since this analysis is based on an single static assignment (SSA) representation (more specifically, gated SSA), we choose to represent the strictest relationships which hold as invariants among static instances of variables. In this example, the static definitions of p1 and p2 are always made together and are always opposite, so we treat them as mutually exclusive. This assertion renders the represented relations useful for instruction-level optimization and scheduling.

3. Previous mechanisms

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 [3]. The PHG relates predicates by keeping

[1] In the IMPACT EPIC Predication model, two destination predicates, $\mathtt{p}_{dest1}$ and $\mathtt{p}_{dest2}$, may be written in a single instruction using two independently selected semantics. To simplify discussion, predicate defines with two destinations are here conceptually split into two instructions.

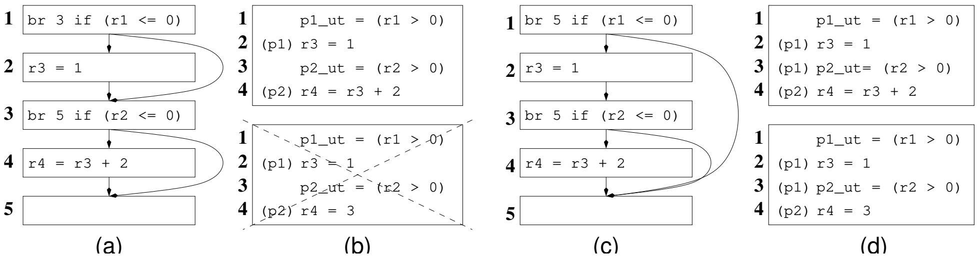


Figure 2: Constant propagation optimization in a predicated region.

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 [10] and [13].

A more sophisticated approach, the *Predicate Query System (PQS)* [9], 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∪p3 and p2∩p3=∅.) 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. Conservative approximations of relations among other define types have been used in practice; unfortunately, the mechanisms for coping with partition graph inaccuracy introduce the possibility of building for a given set of predicate defining instructions several different partition graphs with varying degrees of accuracy [5]. Thus, while PQS can accurately analyze the example of Figure 3(b) (disregarding condition information), it cannot accurately analyze Figure 3(c). 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 [4]. 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. Additionally, this work addresses issues involved in using predication in general control flow, while the techniques of [4] were limited to a single predicated block.

4. The Predicate Analysis System

PAS represents the relationships among predicates defined using the full complement of predicate define types, as well as relations derived from comparison conditions. The following section describes the mapping of conditions and the predicate define network to the underlying binary decision diagram (BDD). BDD naturally describe logical relations in, for example, combinational VLSI circuits; the contribution of PAS is an effective mapping of the predicate analysis problem to this efficient representation.

4.1. Reduced ordered binary decision diagrams

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) forms. Unfortunately, two operations important for predicate analysis, tautology on DNF and satisfiability on CNF, are NP-hard. A different form which is efficient both in tautology and satisfiability is the *if-then-else normal form* (INF) [15]. INF uses only the if-then-else (ITE) operator, where $ITE(x, y, z) \equiv (x \wedge y) \vee (\overline{x} \wedge z)$, to represent Boolean functions. Functions are expressed by recursive decomposition, in the form of a Shannon expansion, using the ITE operator:

$$
\begin{gathered}
f(x_0, \ldots, x_n) = x_n f(x_0, \ldots, x_{n-1}, 1) \vee \overline{x_n} f(x_0, \ldots, x_{n-1}, 0) \\
\Downarrow ITE(x, y, z) = xy \vee \overline{x} z \\
f(x_0, \ldots, x_n) = ITE(x_n, f_1(x_0, \ldots, x_{n-1}), f_0(x_0, \ldots, x_{n-1}))
\end{gathered}
$$

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 subtree. 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. Much work has been done in the development of efficient ROBDD implementations, mostly intended for use in the domain of Boolean logic circuit optimization [15]. BDD have also been applied in software problems, usually in the verification domain.

PAS uses the Colorado University Decision Diagram (CUDD) implementation of ROBDD [16]. CUDD implements "invert" arcs, which can be used instead of "else" arcs to implement the construct:

$$f(x_0,\ldots,x_n) = x_n f_1(x_0,\ldots,x_{n-1}) \vee \overline{\overline{x_n} f_0(x_0,\ldots,x_{n-1})}$$

Here we observe the same recursive formulation as before, but the formula inverts an existing subgraph (f_0) for use as a 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.

Initially, the BDD consists only of the node 1. An interface exists to add new variables to the BDD, each of which is created as a single ITE node with a then-arc and an invert-arc to 1. 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 $ITE(f,g,h)$. 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 are computed (using recursive calls to `ite` which assume $x_t = 1$ and $x_t = 0$, respectively). 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 [17] and operates in time proportional to the size of the resulting function graph.

4.2. Mapping predicate defines to the BDD

Figure 3 shows an example of BDD construction consisting of a single hyperblock. 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. Here, predicate analysis informs the scheduler, for example, that the predicates on statements `B (p4)` and `C (p2)` are mutually exclusive; thus these instructions may be reordered freely. Although this is obvious in the original control flow graph, the CFG is either lost during if-conversion or may not be available if code is input in a predicated form—hence the need for predicate analysis. Since the logical relations among predicates change infrequently during the compilation process, an efficient approach is to engage in a possibly expensive analysis phase during which a database of relations is built, and from which results may be obtained rapidly.

The code is next subjected to predicate optimization and rescheduled using a predicate analysis of the type described here (Figure 3(c)). One 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. Like more sophisticated optimizations, this has an effect on the ability of future predicate analyses to determine the relation of predicates `p2` and `p4`. In (b), derivation of predicate expressions alone demonstrates that $\mathtt{p4} \cap \mathtt{p2} = \emptyset$. 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.

4.3. 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* [18]. 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 4(a) represents the condition family of `r1` in Figure 3. The set of values causing any condition to evaluate to "true" is represented as the union of disjoint intervals. The rela-

```
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)

Figure 3: An example hyperblock: (a) source, (b) if-converted, (c) optimized.

tions among all possible outcomes on this family are represented in the BDD by creating a Boolean space, known as a *finite domain* [19], and assigning all intervals to mutually exclusive and collectively exhaustive expressions. The expressions must be mutually exclusive, as a value can belong to only one interval at a time, and also collectively exhaustive, so that an expression such as "`(x>5) || (x<=5)`" is recognized as always true.

Figures 4(a) through (d) show the construction of the condition layer for the conditions of Figure 3(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 which implement the finite domain. In general, representing i intervals 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), a 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 $\overline{ITE(v_1, 1, ITE(v_0, 1, \overline{1}))}$ 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 4(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, $C_0 = v_1' v_0 + v_1 v_0' + v_2' v_1 v_0 + v_2 v_1 v_0 = v_0 + v_1$, represented in the BDD as $ITE(v_1, 1, ITE(v_0, 1, \overline{1}))$. As shown in Figure 4(d), this simplified expression is computed automatically in the BDD as the disjunction representing C_0 is formed, one ITE at a time. At this stage, 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)`, $v_1 v_0'$, show that C_2 implies C_0. This process is described in detail in [18].

4.4. Construction of the predicate layer

The mapping of predicate defines to the BDD is somewhat more straightforward than the mapping of conditions. Consider the acyclic-rendered control flow graph in which all predicates have been given SSA subscripts (as in PQS), and in which all predicate uses are constrained to have a single forward-control-flow-reaching definition (i.e. there exist no predicate ϕ functions). In this form, predicates may not be live around backedges and no predicate use may see definitions from different blocks. The predicate graph is thus constructed in a single topological traversal of the control flow graph by adding at each predicate define a new expression according to Table 2. In the table, $x \,?\, y \,:\, z \equiv ITE(x, y, z)$ and $n_{i.j}$ represents the BDD node associated with the predicate p$i.j$ (j represents the SSA subscript). As indicated in the table, a new subgraph representing the defined predicate is generated from previously generated subgraphs representing the predicate source, the previous value of the predicate destination, and the condition. The stated forward-flow constraint guarantees that these expressions are available when they are required.

Returning to the example of Figure 3, we construct the corresponding local relation BDD. A BDD expression has already been defined for each condition used in the generation of predicates, as shown in Figure 4(d). In the following topological traversal, predicate define instruction semantics are applied to generate the form shown in Figure 4(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

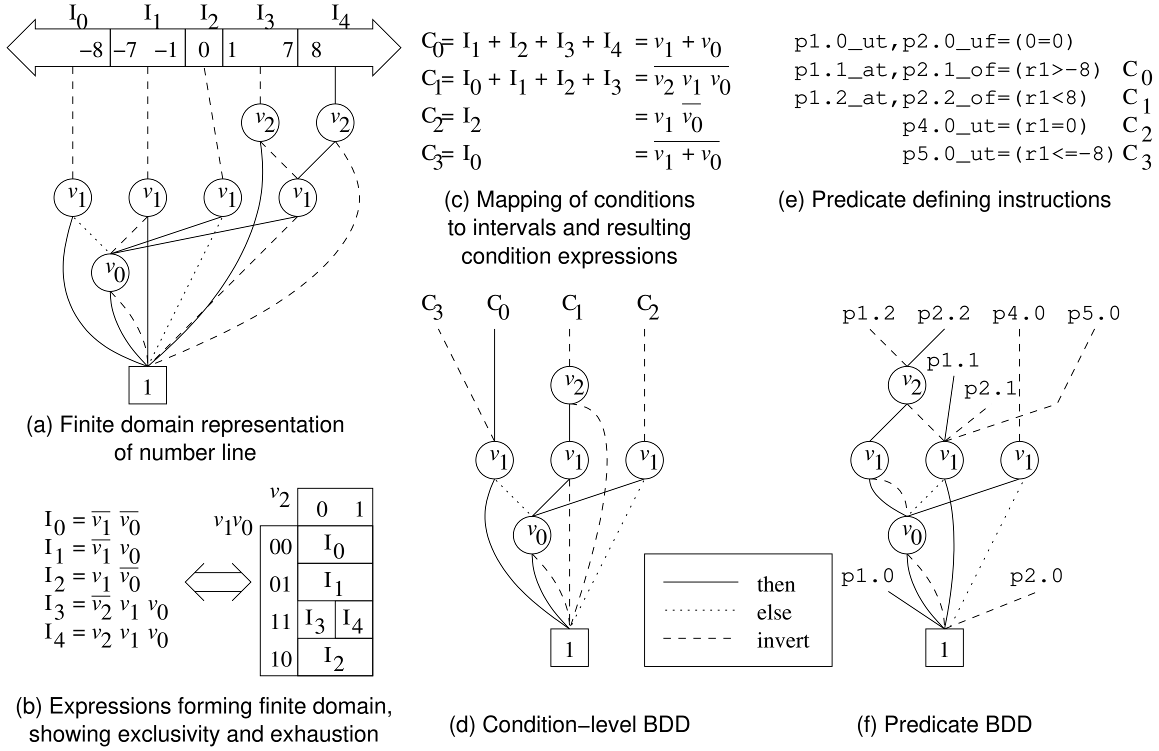


Figure 4: Assembly of Predicate BDD for the example of Figure 3(c).

Table 2: Predicate deposit logic.

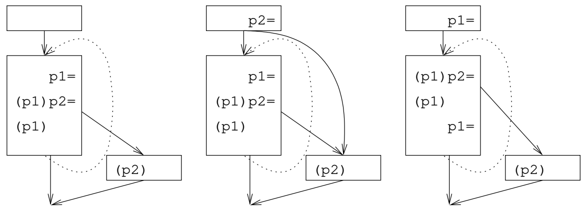
SSA pred. def.	*ITE* Expression
(p_g) p$i.j$_ut = C	$n_{i.j} = C?n_g : 0$
(p_g) p$i.j$_uf = C	$n_{i.j} = C?0 : n_g$
(p_g) p$i.j$_ot = C	$n_{i.j} = C?(n_g?1 : n_{i.j-1}) : n_{i.j-1}$
(p_g) p$i.j$_of = C	$n_{i.j} = C?n_{i.j-1} : (n_g?1 : n_{i.j-1})$
(p_g) p$i.j$_at = C	$n_{i.j} = n_g?(C?n_{i.j-1} : 0) : n_{i.j-1}$
(p_g) p$i.j$_af = C	$n_{i.j} = n_g?(C?0 : n_{i.j-1}) : n_{i.j-1}$
(p_g) p$i.j$_ct = C	$n_{i.j} = C?n_g : n_{i.j-1}$
(p_g) p$i.j$_cf = C	$n_{i.j} = C?n_{i.j-1} : n_g$
(p_g) p$i.j$_∨t = C	$n_{i.j} = C?1 : (n_g?1 : n_{i.j-1})$
(p_g) p$i.j$_∨f = C	$n_{i.j} = C?(n_g?1 : n_{i.j-1}) : 1$
(p_g) p$i.j$_∧t = C	$n_{i.j} = n_g?(C?n_{i.j-1} : 0) : 0$
(p_g) p$i.j$_∧f = C	$n_{i.j} = n_g?(C?0 : n_{i.j-1}) : 0$

shown. `p2.1` is an or-false-type definition with a constant-true guard predicate and condition C_0. Consulting Table 2, `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 in composing this expression compute the node indicated for `p2.2`. Figure 4(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. In a more complex example based on multiple comparison families, several initially independent condition BDD (based on different variables) would be 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.

4.5. Handling of general predicate liveness

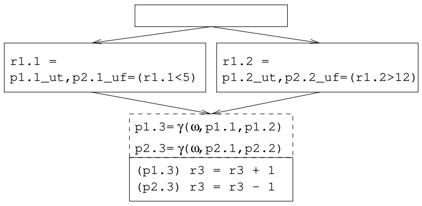
To simplify discussion, only the analysis of forward-flow, single-definition predication was considered. While this is sufficient for code generated by direct if-conversion, various transformations subsequent to if-conversion, such as hyperblock loop rotation, or other applications of predication can result in more complex forms. This work generalizes predicate analysis to include the remaining forward-flow, multiple-definition and cyclic flow forms. This taxonomy is illustrated in Figure 5. "Forward-multiple" extends upon "forward-single" by allowing control flow merges in the forward acyclic control flow subgraph, and "cyclic" extends further to include flow around loop backedges.

Multiple-definition form. When multiple distinct definitions from different locations in the flow graph reach a given predicate use, the original ϕ-less representation is insufficient. Simply adding ϕ functions and treating these as writes of an unknown value provides a conservative result. Accuracy demands that reaching values be correlated with their paths of origin in a canonical logical value merge.



(a) Forward, single (b) Forward, multiple (c) Cyclic

Figure 5: Predicate flow forms.



(a) Control flow merge showing GSA augmentation

Condition	BDD Var
(r1.1<5)	a
(r1.2>12)	b
ω	c

```
p1.1 = a
p2.1 = a'
p1.2 = b
p2.2 = b'
p1.3 = ca+c'b
p2.3 = ca'+c'b'
```

(b) BDD mapping and expressions

Figure 6: Handling of a control flow merge.

Such a merge can be expressed using gated SSA (GSA), a popular representation in parallelizing compilers [20]. Here, gated ϕ functions (γ functions) are inserted for all merges of predicate value. GSA γ functions are of the form $d_3 = \gamma(\omega, d_1, d_2)$, where ω is a logical variable that selects from among the reaching definitions. This is easily incorporated into PAS by treating γ as a new predicate definition type, with the `ite` expression $n_{d_3} = n_\omega ? n_{d_2} : n_{d_1}$. The ω BDD functions can be derived either from the control dependences of surrounding control flow or in a variety of simpler and less expensive ways. Since in this form flow around backedges is still prohibited, a single topological traversal of the acyclic-rendered graph is sufficient to compute the proper graph. Furthermore, control relations need only be encoded to relate those paths which traverse merges of predicate value. Figure 6 shows an example, in which two different sets of definitions for `p1` and `p2` reach the uses in the bottom block. In this case, however, the predicates are identically related in both cases (always opposites), although the two definitions of each predicate are unrelated. Augmentation of the code for GSA form is shown in the dashed box and with the arrows indicating addition of SSA subscripts. BDD nodes are built for the defines and the γ functions as described; Figure 6(b) shows the resulting BDD expressions. Since the same variable guards both the γ functions, the BDD correctly "correlates" the two pairs of definitions, and the fact that `p1` and `p2` are always opposites in the bottom block is captured.

Cyclic flow. Only one extension remains: the treatment of predicate value flow around backedges in the control flow graph. Previous approaches, assuming an intra-hyperblock scope, did not require such support (hyperblocks are inherently acyclic regions [3]). Subsequent transformations, however, may generate flow around backedges. For example, a hyperblock loop may be rotated (as in Figure 5(a) and (c)) to achieve a better instruction schedule, or a Boolean flag initially coded as a variable in a loop by a programmer may be allocated a predicate instead. This transformation, like modulo scheduling, creates predicates whose live ranges cross a backedge.[2] If transformations such as these are to be allowed, a predicate analysis system should be able to accommodate them as well.

In the acyclic case, a topological traversal of the control flow graph ensures that functions needed in predicate definitions are defined at the point of reference, but this no longer holds in the presence of loop-carried dependence. A simple approximation, however, captures relations generated across an arbitrary number of previous iterations of a loop body. In the case of a modulo scheduled loop, this might be equal to the number of modulo stages. In arbitrarily transformed code, this might be some more arbitrary limit, perhaps one previous iteration. We present an extension which allows the BDD form to represent relations that happen to cross loop backedges.

The BDD can be thought of as an efficient symbolic manipulation engine, in which variables allocated to represent unknown inputs can later be rendered irrelevant by further manipulation of expressions or can be used to correlate other values, as in the previous example. We use these properties, combined with μ semantics from GSA, to manage cyclic dependence. The μ function is a special γ function for use at the loop header, at which the analysis must differentiate values from the preheader and the backedge. In $v_{def} = \mu(\omega, v_{\text{preheader}}, v_{\text{backedge}})$ the symbolic variable ω selects either of the two values based on whether execution has come from the preheader or from the loop backedge. While analysis of a use inside the loop body cannot (usually) bind ω to a known logical expression, it can use ω to express useful relations among variables (predicates and registers used in comparisons) that may appear to differ between the first and subsequent loop iterations. The values appearing to the μ function flow along one of two arcs, either from the preheader or from a previous iteration of the loop. Constraints imposed by the effects of a previous iteration on the relations of values flowing around the backedge are expressed in the BDD through a technique called *virtual unrolling*.

As an example, Figure 7(a) shows a loop with a predicate dependence around the backedge. In this case, the loop has been rotated, pushing a define around the backedge and a corresponding initialization into the loop preheader. Clearly, the uses of `p1` and `p2` are always mutually ex-

[2] Kernel-only modulo scheduling of a hyperblock loop is actually a simpler special case, in which no cyclic techniques are *required*. They are, however, *applicable*.

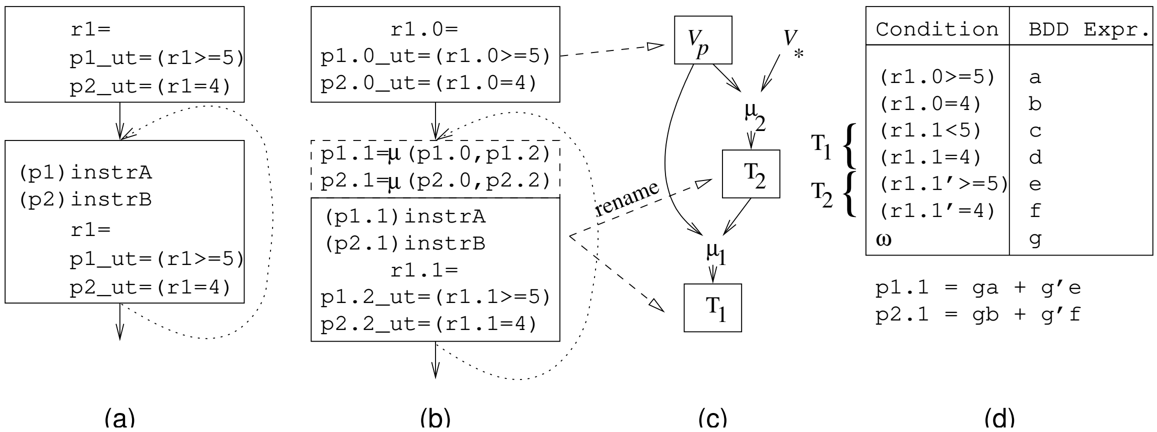


Figure 7: Cyclic predicate flow example.

clusive. Figure 7(b) shows the loop in SSA form. The μ functions share a common free variable ω, as did the γ functions. Suppose we know *a priori* that we can capture all desired relations with one iteration of "look-back" (this will shortly be generalized). When the loop header is encountered, we generate the following (Figure 7(c)): For each predicate pi live in from the backedge, we generate a free variable which we will denote pi^*. These are the predicates annotated in the μ functions; the pi^* versions represent the loop-carried values at the start of the *previous* iteration, about which we know nothing. In Figure 7(c), $V_p = \{\mathrm{p1.0}, \mathrm{p2.0}\}$ and $V_* = \{\mathrm{p1.2}^*, \mathrm{p2.2}^*\}$ denote the sets of values flowing from the loop preheader and from previous iterations, respectively. Now, reasoning that any flow though the backedge must have come through a previous iteration of the loop body, the loop body is "virtually unrolled" and variables are renamed (including ω) as necessary to preserve SSA, creating the copy T_2 (T implies the notion of "transfer function"). The connection among the original loop body (T_1), T_2, and the μ functions is shown in (c). This copy is then processed as normal, generating BDD nodes representing the values flowing from the backedge into the original loop body T_1. These values capture the past iteration's relational information. At this point, both definitions reaching each original μ are available, and the relation graph for the real loop body can easily be computed. Once T_1 is processed, all node handles internal to T_2 are freed, and the garbage collector removes all unnecessary information, as before. In this simple example, since both predicates are unconditionally redefined in each loop iteration, the backedge expressions at μ_1 no longer depend on the free variables pi^*, and the true relations between `p1` and `p2` are expressed. Figure 7(d) shows the expressions for the predicates of interest; given the exclusivity of `a` and `b` and the exclusivity of `e` and `f`, encoded in the condition information, `p1` and `p2` can clearly be identified by the BDD as disjoint. Note that this is the case even though the expressions depend on conditions from a previous iteration (as indicated by the presence of variables related to `r1.1'`).

Virtual unrolling can be performed an arbitrary number of times, depending on the number of iterations of correlation that are required. This could continue until, for example, all pi^* are eliminated from predicate expressions or until some bound is reached. Note that the bound is necessary, since there exist true cyclic dependences for which no amount of look-back will eliminate the pi^*s. This would occur, for example, in a loop containing a predicated or-type predicate define with an initialization outside the loop. (The accurate handling of such recurrences poses a perhaps interesting analysis problem, but lies beyond the scope of this work.) The presence of pi^* variables in an expression, however, does not necessarily preclude determination of logical relations among expressions. Just as the ω variables whose values are not correlated to other variables in the BDD, these free variables serve to "partition the unknown" and to correlate other subexpressions in useful ways.

Figure 8 shows a case in which both predicates and conditions must be treated as live around the backedge. The original code is shown in (a); the effects of virtual unrolling are indicated in (b). After virtual unrolling, the assignment to `p2.0` relies on a variable, `r1.1`, which is given by μ(`ω,r1.0,r1.2'`); that is, the value used in definition may come from a previous iteration. This is handled in the BDD by taking the two possible condition nodes for `r1.1>32`, a = `r1.0>32` and b = `r1.2'>32`, and connecting them using the normal semantics for a μ function. The node for `p2.0` therefore contains the expression $\omega a + \overline{\omega} b$. On the other hand, `p1.1` is the μ merge of two definitions, `p1.0` and `p1.2'`. Supposing c = `r1.0<10` and d = `r1.2'<10`, the expression for `p1.1` is $\omega c + \overline{\omega} d$. Here, a and c share a condition family and are disjoint; likewise for b and d. Since the ω variable properly correlates these definitions, `p1.1` and `p2.0` are correctly recognized by the BDD as disjoint.

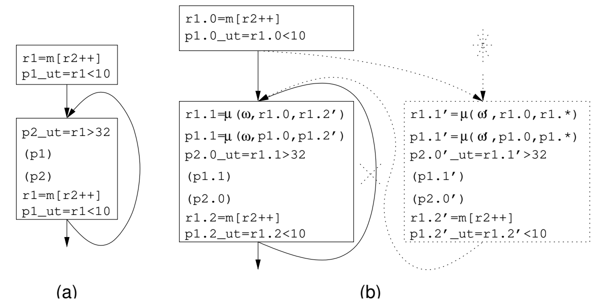


Figure 8: Virtual unrolling. Dotted lines and prime (') marks denote conceptual modifications.

The techniques presented are capable of faithfully representing relations among predicates in the assumed SSA model. The remainder of the paper discusses, first, how the resulting BDD is queried to produce useful results and, second, the efficiency of the approach.

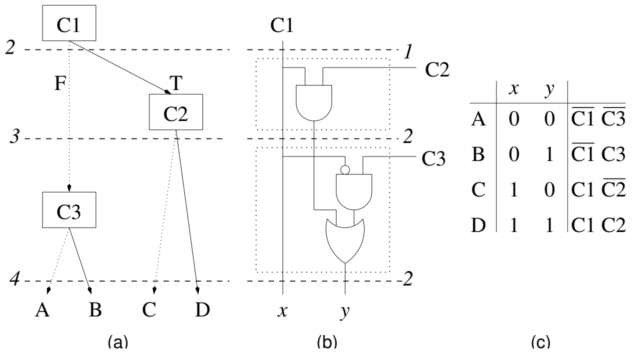
5. Predicate relationship query interface

Following construction of the predicate BDD, PAS provides a query interface for compiler routines to abstract away the details of the BDD implementation. In the IMPACT infrastructure, predicate register operands are tagged with an SSA equivalent that contains a pointer to the representative BDD node. Queries include unary (identity to 1 or 0), binary (subset, intersection, etc.), and multi-predicate (does the disjunction of these predicates subsume another?) functions which reference appropriate nodes in the BDD. Such queries are used to determine, for example, if a predicate is constant-true or constant-false, if dependences should be drawn between two predicated instructions in scheduling, or if code at the end of a block is dead because the expressions for side exit predicates sum to 1.

BDD canonicity guarantees that predicates which are same or opposite are trivially recognized as such (i.e. `p1.2` and `p2.2` from Figure 4(f)). Other queries are composed using the `ite` function with which the predicate BDD was constructed. For example, the query, "Is `p3` a subset of `p1.2`?" is solved in the BDD by computing `q = ite(p3,!p1.2,0)` and determining if `q` = 0, since `p3` $\subseteq$ `p1.2` if and only the intersection of `p3` with the complement of `p1.2` is empty. Similar `ite` constructions provide the other query functions. CUDD prevents the accumulation of nodes from retired queries by performing reference counting garbage collection, and employs dynamic programming techniques to eliminate redundant query computations [16].

6. Performance of the PAS

Like all canonical representations of Boolean functions, ROBDD are provably exponential in the worst case. As described earlier, the canonicity of the ROBDD requires a



	x	y	
A	0	0	$\overline{C1}\,\overline{C3}$
B	0	1	$\overline{C1}\,C3$
C	1	0	$C1\,\overline{C2}$
D	1	1	$C1\,C2$

Figure 9: A control flow graph (a), its equivalent circuit (b), and a truth table representing the trajectory.

single fixed ordering of its constituent variables. This ordering has a significant impact on the size of the resulting ROBDD, and many methods of finding the optimal variable ordering for ROBDD have been proposed for various circuits [21, 22]. Unfortunately, variable reordering cannot prevent the representation of some circuits, such as multipliers, from growing exponentially with circuit size [19]. This was a serious point of concern for compiler writers wishing to use a BDD. The following discussion, targeted at this particular application, shows that in the theoretical limit only one very manageable factor leads to exponential growth of the BDD. Furthermore, empirical results are given to demonstrate that this term is not in practice a significant concern.

6.1. Space and Time Complexity of PAS

Since most predication begins as control flow, it follows that the size of the BDD should be related to the complexity of the control flow it replaces. Properties of real programs thus limit the size of the analysis BDD. First, consider using the BDD to represent a control flow graph (CFG). Figure 9 shows a control flow graph and its derived circuit to be represented by the BDD for analysis. Any slice in a topographically sorted control flow graph (with backedges removed) cuts one or more control flow arcs. The relationships among these arcs in such a slice can be represented by the variables of a corresponding slice in an equivalent circuit. For example, in Figure 9, the relationship of the arcs A, B, C, and D can be represented by the two variables x and y, in a manner illustrated by the truth table. Each arc in a slice represents a mutually exclusive execution condition; only one edge in the slice may be selected for each traversal of the graph. Also, at least one arc must be traversed which makes each slice collectively exhaustive with respect to traversals. These properties allow representation by a truth table in which each arc is given one or more rows to cover all rows (collective exhaustion) and in which each row belongs to one and only one arc (mutual exclusion). In our example, A is represented by $(x, y) = (0, 0)$, B by $(0, 1)$, C by $(1, 0)$, and D by $(1, 1)$. The variables x and y have no particular individual meaning, but together encode all possible relationships. As in the finite domain technique, we need no more than $O(\log_2 i)$ variables to represent the arc outcomes for a slice (or the rows in the truth table), where i is the number of arcs in the CFG slice.

The worst-case size of a ROBDD for some ordering is $O(nm2^W)$, where n is the number of input variables, m is the number of output variables, and W is the maximum width of all slices in the circuit being represented [23]. (A fully expanded decision tree for a Boolean function has $O(2^n)$ nodes. At each slice, the BDD can be split into two parts with the first part having W output variables and the second part having W input variables. BDD are at worst a fully expanded decision tree, so the size of the second BDD is $O(2^W)$.) The following discussion will consider the 2^W term as the rest is polynomial.

The exponential term in the size of the ROBDD representing any CFG is 2^W, but W is the maximum of all $O(\log_2 i)$. Therefore, the exponential term becomes $O(2^{\log_2 I})$ or $O(I)$, where I is the maximum control flow graph slice, in the worst case using the ordering provided by the control flow graph. This result is intuitive since the ROBDD and the program's control flow graph both represent the equivalent relations using the ITE structures.

Now, we relate this polynomial upper bound for representing control flow graphs in ROBDD to representing predicated codes. The function computed by the predicate network of a program is identical to the function computed by the control flow graph. The property of canonicity requires that the BDD representing this function is unaffected by the manner in which it is constructed (assuming fixed condition ordering). All predicate networks computing this function are polynomial in the worst case using the ordering provided by the topological sort of the control flow graph.

Loss of the control flow graph does not preclude derivation of an ordering yielding a polynomially-sized BDD. A general method is to apply a register allocation algorithm to the BDD [23]. Other methods such as sifting provide fast and relatively accurate approximations [24]. Alternatively, the compiler could record the order of the conditions before if-conversion. (Note that this is not the same as recording the full CFG.) In practice, the order of the conditions as appearing in even aggressively predicated codes tend to yield good results since in the aggregate a good correspondence exists to the original condition orderings.

The size of the analysis BDD is also influenced by the inclusion of condition relations, which, as we have seen, create fairly full subtrees at a low level of the BDD. Define, as before, *condition family* as the set of conditions whose results are mutually related (non-independent). Define the *live-range* of a condition family in the variable ordering to start at the first condition and end at the last condition in the family. To represent the information encoding all relationships of conditions in the family an additional k variables may be necessary over the family's live range. The size of k for a family is $O(\log_2 r)$, where r is the number of intervals on the previously discussed segmented number line. The size of r can be shown to be to be $O(c)$, where c is the number of conditions in the family, since each condition can create only 0, 1, 2, or 3 new segmented regions on the number line. The size of the BDD with one condition family is at worst polynomial since $O(2^{\log_2 r}) = O(c)$. This can be bounded further by the observation that control flow arcs often encode information redundant with the condition values.

The size of the BDD is guaranteed to remain polynomial for codes with any number of non-overlapping live ranges in the topological sort of the circuit or CFG. In the PAS, condition family live ranges are typically short lived and do not overlap to a large degree since a condition family's live range maps directly to the live range of its defining register (i.e. `r1` in `r1 < 10` and `r1 > 20`). The worst case size of the BDD having condition family live ranges that both overlap and participate in the same predicate computations has an exponential term, 2^L, where L is the maximum number of condition families simultaneously live. Again, this is worst case and can be bounded further if control flow encodes redundant information. If the overhead of condition analysis is a concern, the number of overlapping condition families live can be directly controlled by exclusion or by live range splitting. Loss of some precision will occur, though the level of precision will still exceed PHG and PQS. Experimental evaluation shows that the PAS is well behaved in the codes studied without such techniques.

Another contribution of this work is the handling of general predicate liveness. The methods presented to deal with cyclic and multiple-definition forms do not change the upper bound presented here. Virtual unrolling increases the length of the circuit, not the width. The GSA ω simply makes part of the control flow graph relevant to the predicate network.

Once the BDD is built all query functions are performed in polynomial time. Equivalence and inverse queries are constant time. Subset, intersection, and exhaustion are all polynomial in time and space with respect to the size of the functions subject to the query.

6.2. PAS in Practice

To evaluate the performance of the described techniques, the analysis was applied to SPEC CINT95 benchmarks. The benchmarks were compiled aggressively for instruction-level parallelism, with 60% profile-guided selective inlining, formation of very aggressive hyperblock regions, and extensive code transformation and optimization. Instruction scheduling and register allocation were performed. It should be noted that the IMPACT compiler currently does not generate predication making use of the forward-multiple or cyclic schemata, so these techniques are not reflected in these numerical results. Were they to be applied, the results would scale by a linear factor as previously described.

PAS was instrumented to determine the analysis time and maximal BDD size for these benchmarks. Experiments were performed on an HP 9000/785/400 workstation operating at a clock frequency of 400MHz with 1GB RAM. Binary decision diagrams for the final code of all SPEC CINT95 benchmarks were built in 2.4 seconds (excluding the assumed SSA construction time). Especially since the BDD typically needs to be rebuilt only when predicate definition optimizations are performed, the time required for BDD construction is acceptable even in a production environment. To measure query efficiency, all pairs of predicates within each hyperblock were tested for subset, superset, and disjoint relationships. A total of 1,177,491 queries were performed in 3.8 seconds. This rapid query response is due in part to the canonicity of the BDD and in part to memoization techniques applied in CUDD [16]. This result is expected, as the control flow of structured programs results in predicate relationship equations which are relatively small and well-behaved in comparison to the large circuits the BDD is capable of managing.

Figure 10 shows the number of BDD nodes necessary to represent the predicate and condition networks plotted against the number of predicate definitions in each function of each benchmark in SPEC CINT95. In each graph, two types of BDD were built. The first type did not include condition analysis information, while the second type did. Figure 10(a) shows the sizes of the BDD created using the condition variable ordering found in predicated codes aggressively optimized by the IMPACT compiler. Figure 10(b) shows the same graph after an application of sifting was applied to find a better variable ordering [24].

One function was excluded from Figure 10(a) to equalize the scales. This function was *strength_reduce_loop* from *126.gcc*, the sixth largest function in terms of predicate define count (333 predicate defines). With basic condition analysis it required 4995 nodes, fairly typical for functions of this size. However, with family analysis it created 18,529 nodes, the greatest of any function. Despite having 26 two-member, 9 three-member, 2 four-member, and 6 nine-member families this was a growth of only a factor of 3.7 times. With less than 19K nodes, a relatively small number when compared to those seen in many VLSI circuits, the BDD package was able to handle this function very well as indicated by a build time of only 0.11 seconds. Reduction in size through variable sifting reduced the size of this BDD from 18,529 to 4462 nodes.

Experimental results indicated that worst case results for overlapping condition family life-times did not materialize. Exploration of the code revealed that this additional information was often partially redundant or that the cost of overlapping condition live ranges was hidden by the larger predicate network functions. The largest growth factor, computed as (BDD size using families / BDD size using simple conditions), was 4.2 for before sifting and 6.0 afterwards; the average was 1.1 and 1.2 respectively. This may indicate that further analysis may reveal upper bounds more restrictive than those presented in the previous section.

Polynomial, linear, exponential, and power curves were fit to the graphs in Figure 10. In each case, the best fit was a power curve with exponents of between 0.87 to 1.07 and with $0.88 < R^2 < 0.91$ (R^2 approaching 1 indicates higher predictive accuracy). The linear curve shown has $0.41 < R^2 < 0.67$ and the exponential model, $0.55 < R^2 < 0.60$.

7. Conclusions

This paper demonstrated a means of accurately mapping the predicate analysis problem to a powerful and efficient logical substrate, the reduced ordered binary decision diagram (ROBDD). Besides demonstrating a high level of performance, this work extended on previous attempts by incorporating the analysis of predication in general control flow and the analysis of conditions into the same logical framework. Finally, concerns about the growth of the representational medium were addressed. The presented system has the power and flexibility to provide predicate analysis for advanced predicate optimization and recompilation environments.

While control of BDD size is not a practical concern at this point, preliminary results shown here indicate that it may eventually become profitable to study efficient means of achieving near-optimal variable orderings in general predication problems.

Other interesting future work includes the possible extension of the framework to include other types of logical information, perhaps in the form of extensions to the condition analysis framework capable of understanding general arithmetic flow.

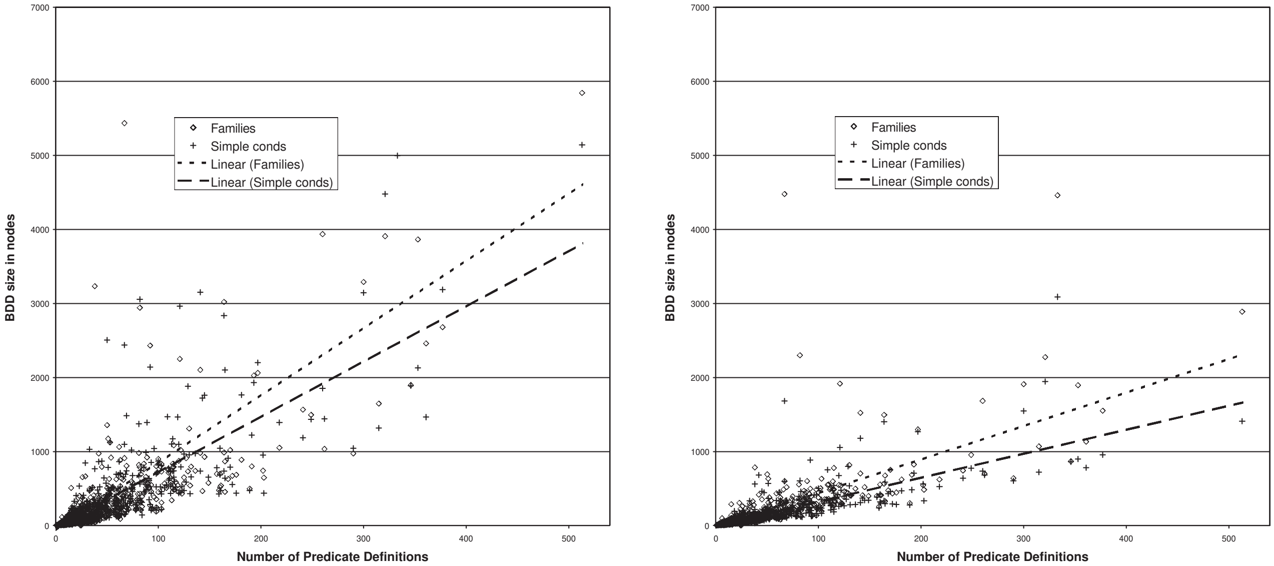
Acknowledgments

We thank Professor Farid Najm for suggesting the use of BDD during work on [10] and Professor Sharad Malik for insight which enabled the complexity analysis of PAS. This study was supported in part by a grant from Intel Corporation. John Sias was supported by a National Defense Science and Engineering Fellowship.

References

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

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



(a) (b)

Figure 10: SPEC CINT95 function-level BDD size before (a) and after (b) variable sifting.

[3] 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*, pp. 45–54, December 1992.

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

[5] 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*, pp. 114–125, December 1996.

[6] 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*, pp. 92–103, December 1997.

[7] 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*, pp. 227–237, June 1998.

[8] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante, "Predicated single static assignment," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

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

[10] 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*, pp. 208–219, May 1999.

[11] A. Srivastava, "Vulcan," Tech. Rep. TR-99-76, Microsoft Research, September 1999.

[12] 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.

[13] 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*, pp. 155–168, May 1999.

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

[15] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transaction on Computers*, vol. C-35, pp. 677–691, August 1986.

[16] F. Somenzi, "CUDD: Colorado University Decision Diagram package, release 2.30," University of Colorado at Boulder, http://vlsi.colorado.edu/˜fabio/CUDD/, 1998.

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

[18] J. W. Sias, "Condition awareness support for predicate analysis and optimization," Master's thesis, University of Illinois, Urbana, IL, 1999.

[19] 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.

[20] P. Tu and D. Padua, "Gated SSA-based demand-driven symbolic analysis for parallelizing compilers," in *Conference proceedings of the 1995 International Conference on Supercomputing*, pp. 414–423, 1995.

[21] S. B. Akers, "Binary decision diagrams," *IEEE Transaction on Computers*, vol. C-27, pp. 509–516, June 1978.

[22] S. J. Friedman and K. J. Supowit, "Finding the optimal variable ordering for binary decision diagrams," in *Proc. 24th Annual ACM/IEEE DAC*, pp. 348–355, June 1987.

[23] C. L. Berman, "Circuit width, register allocation, and ordered binary decision diagrams," *IEEE Transactions on Computer-Aided Design*, vol. 10, pp. 1059–1066, August 1991.

[24] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 42–47, November 1993.