CONDITION AWARENESS SUPPORT FOR PREDICATE ANALYSIS AND OPTIMIZATION

BY

JOHN WOLLENBURG SIAS

B.S., University of Illinois at Urbana-Champaign, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

ACKNOWLEDGMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support. This work would not have been possible without the many years of development effort invested in the IMPACT Research Compiler by many previous generations of students. I would also like to acknowledge the advice and assistance of those IMPACT members with whom I collaborated on related projects. Special thanks are due to David August, without whose many hours of advice, guidance and assistance this work would not have been possible. To my family, thanks for support and understanding during this endeavor. I was financially supported by fellowship grants from the National Defense Science and Engineering Graduate Fellowship Program and International Business Machines' Centre for Advanced Studies during the development of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

A key trend in the achievement of higher performance in computer microarchitectures has been the ability to execute an increasing number of instructions per clock cycle. In just the past decade, processors have developed from, in 1990, supporting a single instruction per cycle [2], [3] to, today, executing four or more instructions per cycle [4]. Processors of the near future are expected to be capable of executing eight or more instructions per cycle. Wide-issue processors such as these demand that a large amount of instruction-level parallelism (ILP) be exposed to capitalize on their offered performance. Furthermore, due largely to the frequency of control operations in code, fetching enough instructions to feed such high-performance cores has become a significant challenge [5].

One approach to meeting these demands for ILP and efficient instruction fetch has been dubbed explicitly parallel instruction computing (EPIC), and is the basis for the IA-64 architecture released recently by Intel and Hewlett-Packard [6]. EPIC architectures permit the compiler to express instruction-level parallelism directly to the processor using features such as compiler-controlled branch prediction, compiler-controlled speculation, and predication. These new compiler responsibilities require significant changes in compiler architecture.

Perhaps the most significant of these, and also the most challenging to the compiler writer, is predication. Effective compilation for predication requires the compiler to analyze and manipulate control, even across code from logically independent paths. In order to produce high-quality predicated code, it is not sufficient to simply introduce predication as a final compilation step; the compiler must perform significant manipulation in the predicated domain [7]. Effectively applying predication and optimizing in a predicated environment demands accurate analysis of the logic underlying program control structures [8], [9]. Furthermore, given an advanced understanding of this "program decision logic," a compiler can dramatically transform program control to expose higher levels of ILP and to increase performance [1]. The framework presented in this thesis is an analysis system designed to provide the predicate analysis infrastructure with information about the logical relationships between the condition tests on which program control is based.

1.1 Organization of this Work

The following chapter describes predication in general and the IMPACT Predication Compilation Framework, of which the work described is a component. Chapter 3 describes the two key IMPACT components currently served by the condition analysis framework, the Predicate Analysis System (PAS) and the Program Decision Logic Optimizer (PDLO), as well as the binary decision diagram (BDD) representation shared by these modules and used as the basis of the condition relationship database. Chapters 4 and 5 present in detail the techniques implemented as part of this thesis and provide some brief examples of their application in the IMPACT environment. Chapter 6 reports experimental findings related to the techniques described, including an apprisement of the prevalence of conditions to which the method can be applied, and a brief discussion of the techniques' contribution to PDLO. This work is followed by an appendix which should provide necessary background for readers unfamiliar with IMPACT [10] or Hewlett-Packard HP-PD [11] predication.

2. PREDICATED COMPILATION AND THE IMPACT ENVIRONMENT

The IMPACT research compiler is a portable, multitarget compiler framework which exists as a test bed for instruction-level parallelism (ILP) compilation. The IMPACT compiler has three significant representational stages, which share two Intermediate Representations (IR). The first stage and highest-level IR is Pcode, which is generated from C source files using a preprocessor. Flattening, program profiling, profile-driven inlining [12], loop-level transformations [13], [14], and interprocedural pointer analysis [15], among other transformations, are performed in Pcode.

Low-level portable optimization is performed in Lcode, the second stage and low-level IR. Lcode is a general register-transfer language similar in many respects to a typical load/store processor's instruction set. Lcode is designed to be sufficiently powerful to be a convenient architecture within which to optimize code, but also sufficiently simple and general enough to describe and convert well to a wide range of target architectures. All machine independent classical optimizations are performed on Lcode. These optimizations include constant propagation, forward copy propagation, backward copy propagation, common subexpression elimination, redundant load elimination, redundant store elimination, strength reduction, constant folding, constant combining, operation folding, operation cancellation, code reordering, dead code removal, jump optimization, unreachable code elimination, loop invariant code removal, loop global variable migration, loop induction variable strength reduction, loop induction variable elimination, and loop induction variable reassociation [16].

Also during the Lcode phase, code is restructured into superblocks [17], which are not predicated, or into hyperblocks [18], which use predication. Superblock-specific [16] and hyperblock-specific [19] transformations are performed to exploit the code optimization and scheduling benefits of these two ILP compilation models. Lcode-based classical optimizations are also performed iteratively after the formation of superblocks and hyperblocks, since more opportunities for their application are exposed by these two techniques. This entails, in the case of predicated code, that the classical optimizations must work in predicated regions of code. This is accomplished with the support of the predicate analysis system, of which the condition analysis mechanism here is a part. Additionally, at this point, the program decision logic may be extracted, optimized and re-expressed using the Program Decision Logic Optimizer (PDLO) [1]. This mechanism is critically dependent on the condition analysis package described in this work.

Finally, architecture-specific translation, machine dependent optimization, register allocation, and scheduling are performed in the Mcode phase. Mcode is Lcode which has been specially modified to match the instruction and operand characteristics of the target architecture rather than those of the IMPACT Lcode specification. Additionally, a detailed machine description database, *Mdes*, is also available to all Lcode modules [20]. The Mdes describes characteristics of the target machine, such as its functional unit population, instruction latencies, and other constraints. The Mdes is queried by the optimization phases to make intelligent decisions regarding the applicability of transformations. As compilation advances, particularly in the scheduler and register allocator, the compiler relies more heavily on the Mdes to generate code appropriate for the target.

Code is scheduled using either an acyclic global scheduling technique [21], [22] or modulo scheduling [23]. Both models support control and data speculation for aggressive enrichment of ILP [10], [24], [25]. Register allocation, sandwiched between a prepass and a postpass schedule in the acyclic model, is performed using a graph coloring approach [26].

Since predication is introduced during the Lcode low-level optimization phase and is perpetuated throughout the rest of the compiler, the predicate analysis framework is required to perform accurate analysis on Lcode and on Mcode for machines supporting predication.

2.1 Predication

The *predicated representation* is a low level N-address program representation in which each operation has a Boolean source operand, its *guard predicate*, whose value determines whether the operation is executed or nullified. Predicate registers, which contain the values of the guard predicates, are manipulated by a set of architecturally specified predicate definition instructions. The use of predicates to guard instruction execution can reduce or even eliminate the need for branch control dependences within selected acyclic regions of code. When all operations control dependent on a branch are predicated to guard execution in the same manner as the branch, that branch can be removed. The process of replacing branches with appropriate predicate computations and guards is called *if-conversion*.

The predicated representation provides an efficient and useful model for code optimization and transformation in the presence of branches. Through the removal of branches, code can be transformed to contain few if any control dependences. What would have amounted to complex control flow transformation can then be performed in the predication domain as traditional straight-line code optimizations. In the same way, the predicated representation allows scheduling among branches to be performed in a domain without control dependencies. The removal of these control dependences increases scheduling scope and affords new freedom to the scheduler. Compilation of code in the predicated representation, however, requires a predicate analysis system such as the one here described.

Predicated execution is the direct execution of code containing predication on a target architecture designed to support it. With respect to a conventional instruction set architecture, the new features are an additional Boolean source operand guarding each operation, a set of predicate registers in which to store Boolean guards, and a set of compare operations used to compute predicate values. This work focuses on the predicated representation as opposed to predicated execution models, but the latter are important as they provide the central motivation for the former. One predicated execution model, also the basis for the representation used in the IMPACT compiler, is described in an appendix to this thesis.

2.2 Predicated Regions

In the IMPACT predication framework, predicated code is generated within regions called *hyperblocks*. A hyperblock is defined to be a single-entry, multiple-exit, acyclic region of code in which internal control is represented in the form of predication. The hyperblock framework was developed as a means of selectively applying predication to integer codes to achieve high performance in architectures supporting predicated execution [18]. Hyperblocks are formed by applying tail duplication and if-conversion over a set of carefully selected paths. Inclusion of a path into a hyperblock is performed on the basis of a set of profitability heuristics. These heuristics operate on four types of information: resource utilization, dependence height, hazard presence, and execution frequency. For purposes of effective program optimization, hyperblocks are formed aggressively to allow the maximal regions of code to be subject to predicate-enabled classical optimization and to optimizations that are conveniently implemented in a predicate-specific way, such as instruction merging, node splitting, and code motion [27]. In the IMPACT framework, hyperblocks can even be formed in a manner that is over-aggressive for the target architecture, for maximal optimization benefit, and then can be partially reverse if-converted at schedule time to fit the resources available [7]. This approach makes maximal use of the predicated representation in the compiler, and thus increases the applicability of the predicate analysis system.

3. PREDICATE ANALYSIS AND OPTIMIZATION FRAMEWORK

The condition analysis and representation techniques described in this thesis were conceived as part of a complete predicate analysis and optimization framework in the IMPACT research compiler. The Predicate Analysis System here described pre-existed the condition analysis techniques presented. Developed in conjunction with the condition framework was the Program Decision Logic Optimizer (PDLO), which extracts the essential logic from predication in a program, optimizes it, and generates a new predicate definition network, which is then reintroduced into the program. This chapter describes these two technologies to motivate condition analysis and to elucidate the role it plays in these two important systems.

The reader not familiar with the IMPACT [10] or the Hewlett-Packard HP-PD [11] implementation of predication is here referred to the appendix for a description of predicate definition instruction semantics, which may contribute to understanding of the following sections.

3.1 Predicate Analysis System

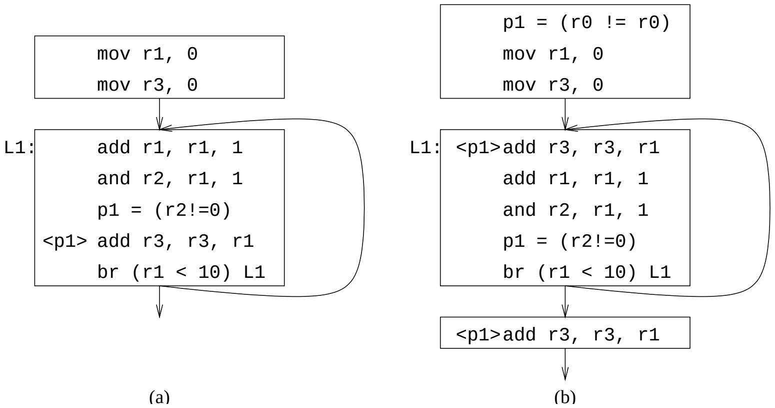
The IMPACT Predicate Analysis System (PAS) was developed by David August to provide accurate information about the logical relationships between predicates, for reference by transformation modules during all phases of predicated compilation. In the IMPACT framework, classical and ILP optimization continues after if-conversion. These optimizations require knowledge of the relation of the predicates guarding the instructions to be optimized. For example, given the instructions `<p1> r1 = 1` and `<p2> r2 = r1 + 1`, a constant propagation optimization can be applied between the two instructions to replace the second with `<p2> r2 = 2` if and only if the predicate `p1` is a superset of `p2`. PAS makes this knowledge available to the compiler. Furthermore, previous work has shown the value of accurate predicate relation information in other areas of compilation, particularly in the effectiveness of register allocation [9], [28].

Salient details of PAS are discussed here because they contribute significantly to the design considerations taken in development of the condition analysis system, particularly to the scope of condition analysis in the control flow graph. In addition, the use of binary decision diagrams as the representational form for logical relationships in PAS lead to the BDDs application in the condition analysis framework.

3.1.1 Scope

The goal of the Predicate Analysis System is to describe in a query-able database the logical interrelation of predicates within a particular scope of analysis. Specification of a scope of analysis involves the desired application. If the system is to be queried by tree-based dataflow and optimizations, that is, with regard to the relation of instructions within the same hyperblock, the relation system should describe invariant relationships between the instances of predicates within the same invocation of that hyperblock, not relationships extending across multiple iterations of the hyperblock. The core queries applied to PAS in IMPACT are of this type. Therefore, the Predicate Analysis System focuses on providing accurate, program invariant information about the static relation of predicates in a function. That is, it will return a reasonable answer to a query about any two static instances of predicates in a function as long as they can be related logically in an invariant way, but it is not query-able with respect to dynamic relationships such as those between instances of predicates across multiple iterations of a loop. The meaning of the static relation of predicates and how this is dictated by the juxtaposition of definitions and uses in the code requires some further description, which will be given in the context of the following example.

Figure 3.1 shows two pieces of code, both of which compute the sum of the odd numbers between zero and ten. The only difference between the two pieces of code is that in (b) the `add` instruction predicated on p_1 is rotated around the loop backedge. This rotation requires the introduction of a predicate-clearing operation to prevent the `add` instruction from executing unnecessarily on the first loop iteration. It also introduces a quandary about which definition of p_1 should be treated as the logical definition of the predicate, either the new definition in the loop pre-header or the definition that is live



```
      mov r1, 0
      mov r3, 0

L1:   add r1, r1, 1
      and r2, r1, 1
      p1 = (r2!=0)
<p1>  add r3, r3, r1
      br (r1 < 10) L1
```

```
      p1 = (r0 != r0)
      mov r1, 0
      mov r3, 0

L1: <p1>add r3, r3, r1
      add r1, r1, 1
      and r2, r1, 1
      p1 = (r2!=0)
      br (r1 < 10) L1

<p1>add r3, r3, r1
```

Figure 3.1: Example of predicate analysis scope limitation.

around the backedge. Neither answer is (statically) correct, since on the first iteration of the loop, the pre-header definition reaches the use of p_1, while in subsequent iterations, the backedge definition reaches the use. The problem evoked here can also be described as a *selection merge* of predicate values, since two different definitions can reach the use of p_1 whose interaction cannot be described by predicate definition instruction semantics alone.[1] The predicate analysis engine thus imposes some constraints on the liveness and merging of predicate register values in a program to simplify analysis.

In the IMPACT compiler, predicated code is generated within the context of hyperblock regions, which are single-entry, multiple-exit acyclic code regions [18]. Within such regions, if-conversion is performed according to the algorithms presented in [29]. This results in predicates that are defined at the top of a hyperblock and that are referenced throughout it, but which die on all hyperblock exits. Thus, for example, if-conversion would not create a case such as the one in 3.1(b), where a predicate is live around a

[1] Strictly speaking, any use of a predicate value defined in a parallel-compare construct has multiple partially reaching definitions, but this is legal since their interaction is logically well-defined as a conjunctive or disjunctive, but not a selection, relationship.

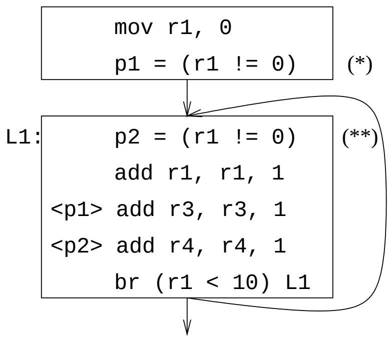
backedge. Further optimizations, such as branch combining, create predicates which may be live out of hyperblocks, but whose live ranges again do not cross backedges. Thus, in order for the system to be useful, relationships between predicates defined and used in the same hyperblock must be accurately represented, and these definitions may be propagated out of hyperblocks into flow successors, but not across backedges. This is convenient, since entry of the hyperblock region guarantees that the execution conditions of each instruction will be determined solely by guarding predicates (with the slight qualification that in partially-resolved hyperblocks, side-exit branches also guard instruction execution) [19]. Thus, discerning the logical relationships between predicates in the hyperblock does not involve control-flow analysis outside the hyperblock region or analysis around backedges, which would dilute accuracy and require some level of iteration.

There exists, however, the possibility of extending the analysis system to include control flow as well as predicates. For example, within a maximal single-entry, acyclic region of code, an effective predicate could be assigned to each basic block by performing control dependence analysis on the region, as occurs in if-conversion [27]. These predicates could then be used in the analysis engine to provide results for a region containing a mix of predication and (acyclic) control flow. Optimization of code throughout such a mixed region, however, is more complex than optimization within a fully if-converted region, since control flow must be navigated.

3.1.2 Set representation

The problem of representing logical relationships between predicates and conditions is essentially a problem of set representation. Consider the set $\mathcal{E}$ of all possible execution records of a program, where an execution record is defined to be a record of all control logic settings in the sequential single-assignment (SSA) representation of a program execution. For notational convenience, let us assume that the registers in the program are cast into single static assignment form, where the jth definition of predicate p_i is denoted by $p_{i,j}$. Consider that any given SSA predicate $p_{i,j}$ will have a corresponding set of execution records $\mathcal{T}(p_{i,j}) \subseteq \mathcal{E}$ in which the predicate evaluated to $\mathbf{1}$, and any SSA-based condition evaluation $C_{i,k}$ will have its $\mathcal{T}(C_{i,k}) \subseteq \mathcal{E}$. The relationships between conditions and predicates are completely expressed by operations on these "true sets." For example, if one predicate implies another, as p_2 implies p_1 in $p_2 \xleftarrow{UT} (C)\ \langle p_1 \rangle$, then $\mathcal{T}(p_2) \subseteq \mathcal{T}(p_1)$. The problem of representing relationships among predicates and conditions, then, is equivalent to characterizing relationships among the true sets of a program.

In general, full enumeration of the true sets of a program is impracticable. First, programs with loops of arbitrary iteration count cannot be represented using true sets in a closed form. Second, the number of execution records for even a trivial program can be impossibly large due to the exponential explosion of paths with respect to branch control dependences. Fortunately, the resolution of the scope issue above also solves the set enumeration problem. If the predicate analysis system is limited to code in which



```
     mov r1, 0
     p1 = (r1 != 0)

L1:  p2 = (r1 != 0)
     add r1, r1, 1
<p1> add r3, r3, 1
<p2> add r4, r4, 1
     br (r1 < 10) L1
```

Figure 3.2: Condition differentiation in the control flow graph.

predicates are not live around backedges and is further limited to making judgments based on a static instance paradigm, the representation problem is much simplified.

Within this model, predicate analysis is performed on a version of the control flow graph which has been rendered acyclic, and selection merges of predicate values are disallowed. These limitations provide a well-defined meaning for the static instance paradigm. Thus, in order to process all predicate definitions in an order consistent with their logical dependence, all that is required is to examine the predicate definitions in a topological ordering based on the acyclic-rendered control flow graph.

Predicate definitions, in turn, depend on the evaluation of conditions for some of their logical inputs. Conditions must therefore be processed in a likewise manner. Two condition instances are identified as being logically equivalent for purposes of predicate analysis if they share the same condition type and sign, the same definition(s) of the same source register(s) in the context of the potentially cyclic control flow graph (not the acyclic-rendered flow graph), and the same constants, if applicable. Figure 3.2 shows why conditions must be considered in terms of the complete control flow graph. In

$$\boxed{\begin{array}{c} \mathit{equivalent}\left(p_a, p_b\right) \equiv \mathcal{T}\left(p_a\right) = \mathcal{T}\left(p_b\right) \\ \mathit{inverse}\left(p_a, p_b\right) \equiv \mathcal{T}\left(p_a\right) = \mathcal{E} - \mathcal{T}\left(p_b\right) \\ \mathit{subset}\left(p_a, p_b\right) \equiv \mathcal{T}\left(p_a\right) \subset \mathcal{T}\left(p_b\right) \\ \mathit{intersect}\left(p_a, p_b\right) \equiv \mathcal{T}\left(p_a\right) \cap \mathcal{T}\left(p_b\right) \neq \emptyset \\ \mathit{exhaust}\left(p_a, p_{b0}, p_{b1}, \ldots, p_{bn}\right) \equiv \mathcal{T}\left(p_a\right) = \bigcup_i \mathcal{T}\left(p_{bi}\right) \end{array}}$$

Figure 3.3: Predicate true set operations.

the figure, predicate definitions (*) and (**) have similar conditions, but on all but the first loop iteration are based on different contents of register `r1`, and are hence not logically equivalent. If only the acyclic-rendered flow graph is considered, the two predicate definition instructions appear to be evaluating the same condition. If this were allowed to occur, analysis would become inaccurate.

3.1.3 Interface and representational mechanism

PAS answers the predicate queries indicated in Figure 3.3, within the scope here defined. Various previous approaches have answered these queries on the basis of predicate definition expressions drawn directly from properties of the predicate definition types themselves [8]. The PAS approach, on the other hand, defines a level of abstraction at the set level, and implements the query system on top of a general logical representation package. This frees PAS from restrictions based on particular predicate definition structures, such as those resulting from direct if-conversion, and allows the use of existing technology, in our case, the BDD, for efficient representation of logical relationships. PAS builds a BDD to represent the logical relationship between predicates by examining

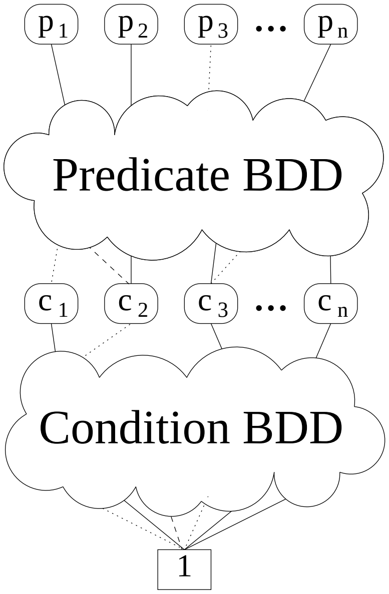


Figure 3.4: Combined analysis BDD.

predicate definition instructions in the topological order previously described, and by following the semantics of the architecturally defined predicate definition instructions to enter into the BDD an accurate logical representation of the predicate definition network. This can be thought of as building a logic circuit, the input variables to which (and the variables of the BDD) correspond to the conditions evaluated by the program, and the gates and interconnect of which correspond to the predicate definition network itself. At this point, conditions are considered to be independent elements; as will be shown in Chapter 4, the condition relation mechanisms that constitute this thesis are incorporated by instead connecting the predicate BDDs inputs to the outputs of smaller BDDs representing condition relationships. Figure 3.4 gives a conceptual picture.

3.2 Binary Decision Diagrams

A binary decision diagram (BDD) for a Boolean function $f\left(x_0, x_1, \ldots, x_n\right)$ is a directed acyclic graph having interior nodes I and the leaf nodes **0** and **1**. Each interior node n is labeled with a variable and has two children, $n \rightarrow then$ corresponding to the evaluation of the variable to **1** and $n \rightarrow else$ corresponding to the evaluation of the variable to **0**. Thus, the value of a function is defined recursively as the Shannon expansion of the function down a path from the root to terminal nodes, as $f\left(x_0, x_1, \ldots, x_n\right) = x_0 f_{then}\left(x_1, \ldots, x_n\right) \vee \overline{x}_0 f_{else}\left(x_1, \ldots, x_n\right)$, where f_{then} and f_{else} are the functions represented by the subtrees pointed to by $n \rightarrow then$ and $n \rightarrow else$, respectively. Due to this behavior, interior nodes are referred to as *if-then-else* (*ITE*) nodes. One restriction is that a particular variable can appear only once in any path of the BDD from root to a leaf [30].

Much work has been done in the development of efficient BDD implementations, mostly intended for use in the domain of Boolean logic circuit optimization [31]. The IMPACT compiler uses the Colorado University Decision Diagram (CUDD) framework to represent BDDs [32]. CUDD is an implementation of a reduced ordered binary decision diagram (ROBDD) based on the description of [33]. A ROBDD is a BDD in which variables appear in the same order in each path from root to leaf, in which all identical *ITE* nodes are shared, and in which no redundant tests are allowed. Within this framework, a "forest" of BDDs is created to represent the interaction of the various predicate functions.

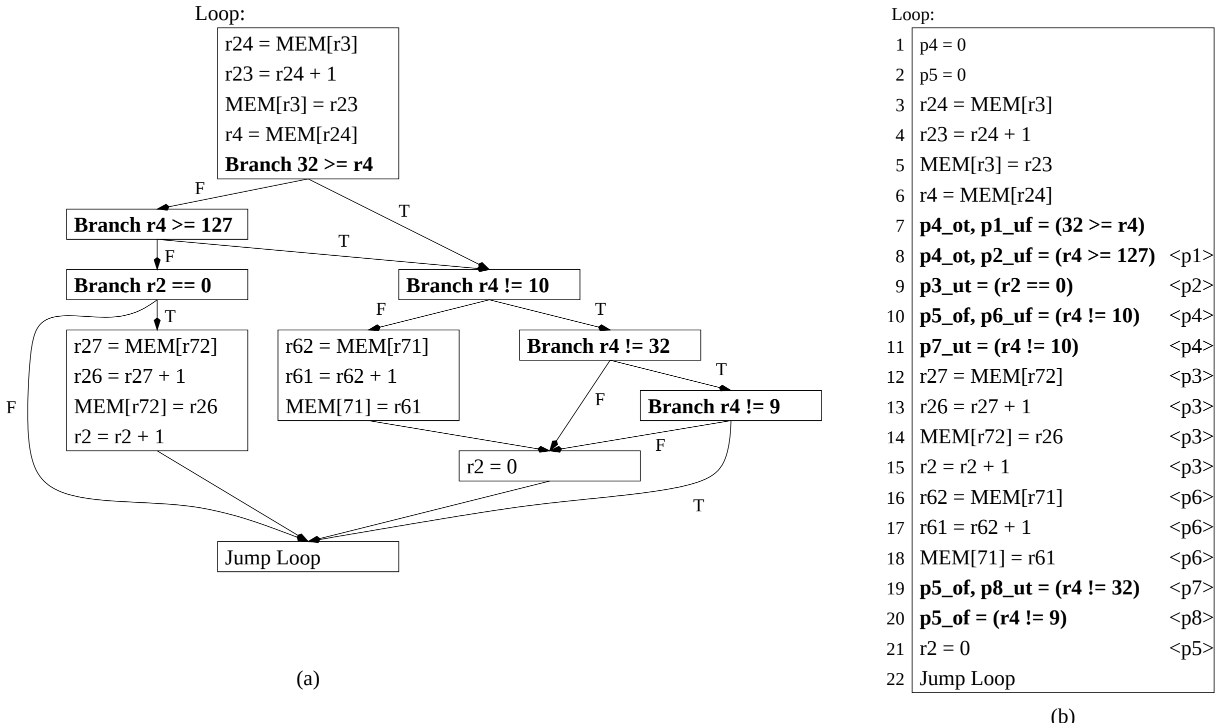
The ROBDD starts out with a single terminal node having the value **1**. Variables are then created and added to the ROBDD as nodes. New nodes, representing expressions on the defined variables, each of which is root to a ROBDD, are created using the function $ITE(x, y, z)$, where x, y, and z are existing nodes in the ROBDD. Recall that ITE stands for "if-then-else"; the node returned has the logical value given by "if(x) then y else z." The CUDD ROBDD manipulation package also has "invert" arcs, which can be used instead of "else" arcs to implement the function $f\left(x_0, x_1, \ldots, x_n\right) = x_0 f_{then}\left(x_1, \ldots, x_n\right) \vee \overline{x}_0 \overline{f_{invert}\left(x_1, \ldots, x_n\right)}$, where f_{invert} is the subtree pointed to by $n \rightarrow invert$. The use of invert arcs allows for constant-time inversion and avoids the addition of extra internal nodes when the complement of an existing subgraph is required [33]. A strong canonical form is maintained by requiring that invert arcs can only occur as "else" arcs.

Just as the BDD is constructed to represent logical relationships using $ITE()$, the same function is used to query the BDD as to the relation of two or more functions described by it. For example, consider testing if f_a implies f_b. This implication holds as long as whenever f_a is true, f_b is never false. Thus, the query node $q = ITE(f_a, NOT(f_b), 0)$ is formed and tested for identity to **0**. If $q = \mathbf{0}$, the implication holds. Similar queries can be constructed for, among others, the useful relations indicated in Figure 3.3. A garbage collection mechanism built into CUDD prevents build-up of retired query nodes in the forest. Additionally, CUDD provides query functions which do not result in the aggregation of BDD nodes to accelerate some Boolean queries. The implementation details of such a system are well described in [33] and will not be recounted here.

3.3 Program Decision Logic Optimization

This section summarizes the Program Decision Logic Optimizer, presented at ISCA-26 [1], which currently constitutes the most significant application of condition analysis information. PDLO acts on code after hyperblocks have been formed and after the code they contain has been optimized traditionally. When decision logic optimization is desired, function inlining takes place and if-conversion is aggressively performed to expose large amounts of control to the optimization techniques, which operate exclusively on the predicate definition network. In addition, techniques such as loop unrolling, peeling, and accompanying ILP transformations are typically applied to the code. Incidentally, many of the loop techniques tend to generate large families of logically related conditions, which are particularly suited to condition analysis.

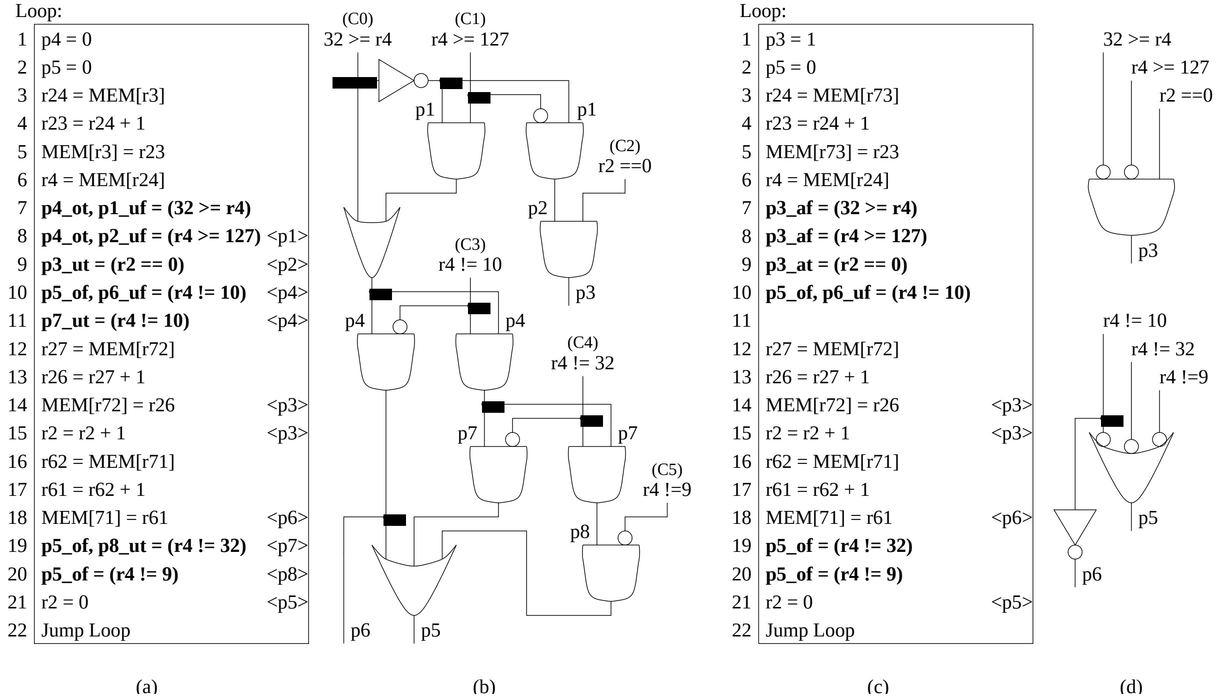
An example extracted from the UNIX utility *wc* illustrates the application and benefit of the described techniques. Figure 3.5 shows the code segment before and after complete if-conversion. As shown in Figure 3.5(a), the code before if-conversion consists of basic blocks and conditional branches (shown in bold) which direct the flow of control through the basic blocks. As shown in Figure 3.5(b), the code after if-conversion consists of only a single block of sequential instructions, a hyperblock [34]. The conditional branches have been replaced with predicate definition instructions (shown in bold) and the predicate registers defined have been placed as source operands on all guarded instructions in accordance with their execution conditions.



```
Loop:
p4 = 0
p5 = 0
r24 = MEM[r3]
r23 = r24 + 1
MEM[r3] = r23
r4 = MEM[r24]
p4_ot, p1_uf = (32 >= r4)
p4_ot, p2_uf = (r4 >= 127)  <p1>
p3_ut = (r2 == 0)           <p2>
p5_of, p6_uf = (r4 != 10)   <p4>
p7_ut = (r4 != 10)          <p4>
r27 = MEM[r72]              <p3>
r26 = r27 + 1               <p3>
MEM[r72] = r26              <p3>
r2 = r2 + 1                 <p3>
r62 = MEM[r71]              <p6>
r61 = r62 + 1               <p6>
MEM[71] = r61               <p6>
p5_of, p8_ut = (r4 != 32)   <p7>
p5_of = (r4 != 9)           <p8>
r2 = 0                      <p5>
Jump Loop
```

Figure 3.5: A portion of the inner loop of the UNIX utility *wc*. The control flow graph (a), and the corresponding hyperblock formed after complete if-conversion (b).

After if-conversion, control speculation is performed to increase opportunities for optimization. Control speculation is a means of breaking a control dependence by allowing an instruction to execute more frequently than is necessary. In a predicated representation, this is performed in *predicate promotion*, the process by which predicate flow dependences are broken and instructions are made to execute speculatively by changing an instruction's guard predicate to another predicate, whose expression subsumes that of the original [18]. When instructions are aggressively promoted, some predicates may no longer be utilized as guards on computation. When a predicate is no longer necessary, the program decision logic is simplified. Figure 3.6(a) shows the *wc* hyperblock segment



```
Loop:
1  p4 = 0
2  p5 = 0
3  r24 = MEM[r3]
4  r23 = r24 + 1
5  MEM[r3] = r23
6  r4 = MEM[r24]
7  p4_ot, p1_uf = (32 >= r4)
8  p4_ot, p2_uf = (r4 >= 127) <p1>
9  p3_ut = (r2 == 0)          <p2>
10 p5_of, p6_uf = (r4 != 10)  <p4>
11 p7_ut = (r4 != 10)         <p4>
12 r27 = MEM[r72]
13 r26 = r27 + 1
14 MEM[r72] = r26             <p3>
15 r2 = r2 + 1                <p3>
16 r62 = MEM[r71]
17 r61 = r62 + 1
18 MEM[71] = r61              <p6>
19 p5_of, p8_ut = (r4 != 32)  <p7>
20 p5_of = (r4 != 9)          <p8>
21 r2 = 0                     <p5>
22 Jump Loop
```

```
Loop:
1  p3 = 1
2  p5 = 0
3  r24 = MEM[r73]
4  r23 = r24 + 1
5  MEM[r73] = r23
6  r4 = MEM[r24]
7  p3_af = (32 >= r4)
8  p3_af = (r4 >= 127)
9  p3_at = (r2 == 0)
10 p5_of, p6_uf = (r4 != 10)
11
12 r27 = MEM[r72]
13 r26 = r27 + 1
14 MEM[r72] = r26             <p3>
15 r2 = r2 + 1                <p3>
16 r62 = MEM[r71]
17 r61 = r62 + 1
18 MEM[71] = r61              <p6>
19 p5_of = (r4 != 32)
20 p5_of = (r4 != 9)
21 r2 = 0                     <p5>
22 Jump Loop
```

Figure 3.6: The *wc* hyperblock after speculation but before logic minimization (a) and its corresponding logic diagram (b). The hyperblock after logic minimization (c) and its corresponding logic diagram (d).

after predicate promotion. Comparison with Figure 3.5(b) shows that four instructions (12, 13, 16, and 17) have had their predicates promoted to the TRUE predicate, denoted in the figure as the absence of a source predicate. However, no predicates were rendered completely unused by this process.

Next, the program decision logic network is constructed. Since predicates can only assume Boolean values, predicates and predicate definitions can be viewed as a combinational logic circuit. To derive the Boolean function from a hyperblock, the compiler needs only to examine the predicate definition instructions. Consider instructions 7 and 8 in Figure 3.6(a), in which the expression for $p1$ can be written as $p1 = \overline{C_0}$ and $p2$ can be written as $p2 = p1\overline{C_1}$, where C_0 is the condition (32 $\geq$ r4) and C_1 is the condition (r4

$\geq$ 127). The expression for $p2$, in terms of conditions, is $p2 = \overline{C_0}\ \overline{C_1}$. In the course of this complete back substitution, expressions based on condition variables are formulated for all predicate definition instructions. The composition of all these expressions is the program decision logic network. This network can be modeled as a logic circuit that represents all the decisions made in the program. The logic circuit has conditions as its input and the predicates which control computation as its output. The multiple-output Boolean logic circuit for the *wc* code segment is shown in Figure 3.6(b).

Once the logic circuit has been derived, many CAD techniques can be employed to simplify the program decision logic network. The predicate analysis BDD, described previously, contains the relationship among predicates as defined by the network of predicate definition operations. For the purposes of decision logic minimization, the BDD provides a simple method by which expressions describing the hyperblock logic can be derived. The only expressions requested from the BDD are those expressions describing the *essential predicates*. Essential predicates are those predicates that guard real computation instructions (any instruction that is not a predicate definition). In Figure 3.6(a), the essential predicates are $p3$, $p5$, and $p6$. Predicates $p1$, $p2$, $p4$, $p7$, and $p8$ are *nonessential predicates* as they are used only as intermediates in evaluation of the essential predicates.

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

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

Figures 3.6(c) and 3.6(d) show the reformulated hyperblock and corresponding logic circuit after the minimization process is complete. The number of logic gates in the circuit implementation is reduced from 10 to 3. In addition, the six-level gate network in Figure 3.6(b) is reduced to a single-level gate network in Figure 3.6(d). All non-essential predicates were also eliminated as part of this process. An example optimization performed on the logic circuit takes the form $C_0 + C_1\,\overline{C_0} \rightarrow C_0 + C_1$. An application of this optimization occurs between instructions 7 and 8 when computing $p4$.

The values of variables in the decision logic network are supplied by evaluating conditions on predicate definition instructions. The importance of the condition analysis system to the program decision logic optimizer is that these variables are not necessarily independent, and that knowledge of the relationships between these variables can allow for significant further optimization of the predicate definition structure. Consider the computation of $p6$ in Figure 3.6(a). Instruction 10 computes $p6\_uf = C_3 \langle p4 \rangle$. Logically, this leads to the expression $p6 = \overline{C_3}\ (C_0 + C_1)$, where $C_0 = (32 \geq r4)$, $C_1 = (r4 \geq 127)$, and $C_3 = (r4 \neq 10)$. Here, since $\overline{C_3}$ implies C_0 and excludes C_1, the expression for $p6$ can be simplified to $p6 = \overline{C_3}$. The role of the condition analysis system is to provide the requisite relational information to the optimization algorithms in an accessible way.

The overall effectiveness of the program decision logic minimization process on the *wc* example is best shown by comparing the schedules of the code before and after optimization. For illustration purposes, a six-issue processor with no restrictions on the combination of instructions that may simultaneously be issued is assumed. Furthermore, all instructions are assumed to have a latency of one cycle. The reformulated hyperblock requires only a single level of predicate definitions to compute the essential predicates as opposed to the five-level network used in the original code, yielding a significant increase in performance. After logic optimization, the number of cycles required per iteration is cut in half, from eight to four.

4. INTEGER CONDITION ANALYSIS

The objective of integer condition analysis is to describe as completely as possible the logical relationships between conditions, providing a logical database as a substrate for analysis or optimization of predication or control flow. This chapter lays the groundwork for such a mechanism. The types of conditions to be examined within the IMPACT architecture are described, and a means of classifying the conditions present in a function into related families is presented. These families will be the subject of the description mechanism presented in the following two chapters.

4.1 Interface

As will eventually be shown, the condition analysis system interfaces seamlessly into the predicate analysis system since it relies on the same internal representation for the description of Boolean expressions. Thus, no explicit interface is required between these two modules. For other applications, such as program decision logic optimization, it is useful to provide an interface which the compiler can query directly to determine the relationship between conditions. Although these queries could be performed directly as BDD operations, it is cleaner to encapsulate the BDD references. Such queries include, for example, `cond_implies(`C_1`,`C_2`)`, which returns **T** iff $C_1 \rightarrow C_2$; `conds_exhaust((`C_1`,`C_2`,`$\ldots C_{n-1}$`),` C_0`)`, which returns **T** iff the disjunction of conditions $1 \ldots n-1$ subsumes C_0; and `cond_opposite(`C_1`,`C_2`)`, which returns **T** iff $C_1 = \overline{C_2}$. These queries, and a few other similar ones, compose the query mechanism underlying condition optimizations performed in the program decision logic optimization framework.

4.2 Architectural Background

In the IMPACT EPIC architecture [10], a condition is defined to be of the form

$$(register)(< \mid \leq \mid = \mid \geq \mid >)_{(s|u)}(register|constant)$$

A condition is composed of, from left to right, the first operand to be compared, the comparison type, the sign of the comparison type, and the second operand to be compared. The sign is indicated in this text by a subscripted s (signed comparison) or u (unsigned comparison) on the comparison operator. The first operand is restricted to be a register, and the second may be either a register or a constant. This definition divides conditions into two classes: those between two registers, and those between a register and a constant. These two types of comparisons require two methods of representation, which will be described separately in the chapter to follow.

In the IMPACT EPIC architecture, all comparisons are to operands of the standard register size. For architectures that support comparison of operands of varying sizes, such as IA-64, multiple operand sizes must also be supported in the analysis.

4.3 Comparison Families

The goal of condition analysis is to provide a logical database for reference by predicate analysis. The region over which the database should be valid must thus correspond to the region over which the predicate analysis and optimization are applied. IMPACT PAS covers the acyclic-rendered control flow graph. A condition family is thus defined basically to be the set of conditions having the same register names and contents as they appear in the acyclic subset of the control flow graph. However, for two comparisons that appear to be related in the acyclic-rendered flow graph to be considered related, they must be identical in terms of their relation to definitions in the complete flow graph. Thus, two conditions which are in fact related by a flow around a backedge are *not* considered related for the purposes of family formation, since such an inclusion would violate the basic assumptions of the predicate analysis framework. The example of Section 3.1.1 provides further clarification. Given this definition, families can consist either of all register-constant comparisons or of all register-register comparisons. Register-register condition families have the additional restriction that all comparisons between the registers must be of the same sign convention, either all signed or all unsigned,[1] for a reason that will be

[1] The equality operator can be taken either as a signed or as an unsigned comparison for this purpose.

addressed in Section 5.1. This definition of families allows all members of a given family to be related logically in the database mechanism to be described.

This definition decomposes the conditions of a function into families. Register-constant comparisons, since all register-constant comparisons can be intermingled in a family, are each mapped uniquely to a family. Mapping of register-register comparisons, on the other hand, may be ambiguous, since signed and unsigned comparisons may not be mixed within a family, but equality and inequality may be placed in either. If, for example, a segment of code were presented which contained the comparisons `(r1` $<_s$ `r2)`, `(r1` $<_u$ `r2)`, and `(r1 == r2)`, where all definitions of `r1` and `r2` were the same, two families would be formed, one for signed comparison and the other for unsigned comparison. The test of equality could be mapped arbitrarily to either, but subsequent queries would treat it as being independent of the family to which it was not mapped. It is, however, highly unlikely that mixed signed and unsigned comparisons of the same two values would be encountered in real code. This is, thus, not much of a concern.

Since condition families are identified by register assignments, techniques such as single static assignment form, value numbering, or a simple iterative matching algorithm can be used to select them from the acyclic-rendered control flow graph, depending on what approach is supported in the host compiler.

Extensions to this condition analysis system would likely take the form of increasing the size of families. A more sophisticated analysis, for example, might be able to identify values as flowing through computation, correctly identify the "extended family," and

```
     p1 = (r1 < 1)  C1
<p1> r1 = r1 - 1
<p1> p2 = (r1 > 1)  C2
<p2> stmt
```

Figure 4.1: Missed opportunity due to interspersed computation.

adjust the analysis to account for value modification between the different comparisons. A simple example is given in Figure 4.1. Here, the comparisons on registers C_1 and C_2 may be logically related, although the presented analysis system is incapable of recognizing or representing their relationship. In this example, if `r1` $<$ 1, p_1 is set, and then `r1` is decremented and tested to see if it is greater than 1. The condition analysis system presented here cannot detect any relationship between the two comparisons because of the interspersed assignment to `r1`. Here, provided that register underflow (subtraction of one from either zero or the maximally negative representable number, causing it to roll over to a positive value) can be ruled out by analysis, p_2 can be shown always to evaluate to **F**. If subtractive underflow cannot be ruled out, no such determination can safely be made. This simple example thus points out both a potential benefit and a challenging pitfall of expanding condition analysis to support analysis of general computation chains.

Furthermore, in many regions of code, such as unrolled loops, for example, other compiler optimizations such as induction variable elimination tend to convert chains of simple computation into families suitable for the presented condition analysis system. Since the code generated by these approaches is in general more suitable for parallel execution than would be code with serial dependences through register computation, an

ILP compiler will in general perform these optimizations, allowing this analysis to be effective in such regions.

5. REPRESENTATION MECHANISMS

5.1 Register-Register Comparison Families

The first representation mechanism of the condition analysis system is one to describe the relation of register-register families. A register-register family consists of a set of similarly-signed conditions, each of which compares the same pair of register contents. For purposes of discussion, let the values be identified here as $\mathtt{r1}$ and $\mathtt{r2}$. As indicated previously, comparisons can be of five basic types: *less than*, *less than or equal to*, *equal to*, *greater than or equal to*, and *greater than.* Within comparisons of the same sign, implication relationships are well defined. For example, ($\mathtt{r1} <_s \mathtt{r2}$) implies ($\mathtt{r1} \leq_s \mathtt{r2}$). The goal of this mechanism is to represent the logical relation of member conditions in BDD form.

The BDD representation is evoked by enumerating three basic categories of comparison outcomes. Here, the same pair of values can be related in exactly one of three ways: (I) ($\mathtt{r1} < \mathtt{r2}$), (II) ($\mathtt{r1} = \mathtt{r2}$), or (III) ($\mathtt{r1} > \mathtt{r2}$). Denote these the *disjoint outcomes* of the comparison family. The outcomes of all member comparisons, the *expressed outcomes*, can be derived given the knowledge of to which category the pair of presented values apply. It is convenient to identify with each expressed outcome the set of disjoint outcomes which imply its truth. By representing the comparisons in terms of these basic outcomes, the relationships between the comparisons can be derived. For example, consider the two conditions (A) ($\texttt{r1} \leq \texttt{r2}$) and (B) ($\texttt{r1} > \texttt{r2}$). Here, the evaluation of (A) to **T** indicates that either a (I) or a (II) outcome exists, but that a type (III) outcome is impossible. Thus, it is demonstrated that (B), which requires a type (III) outcome, must evaluate to **F**.

These relationships are conveniently expressed in BDD form using the technique of finite domains [35]. Mapping the disjoint outcome space to two Boolean variables $\{v_1, v_0\}$ we get the relations ($\texttt{r1} < \texttt{r2}$) $\Leftrightarrow \{0,0\}$, ($\texttt{r1} > \texttt{r2}$) $\Leftrightarrow \{0,1\}$, and ($\texttt{r1} = \texttt{r2}$) $\Leftrightarrow \{1,-\}$. The membership of the disjoint outcomes in the expressed outcome sets is then represented by constructing each expressed outcome node as the logical disjunction of the member disjoint outcome nodes, using the $ITE()$ function of the BDD. The complete BDD representing the logical relationship between all possible comparison operators in a register-register family is shown in Figure 5.1. This BDD, or a subset of it, as needed to represent the operators that are actually used in a family, would be included as part of the analysis BDD condition layer. Predicate definition instructions using the various conditions of the family would be connected to the appropriate condition nodes at the top of this layer.

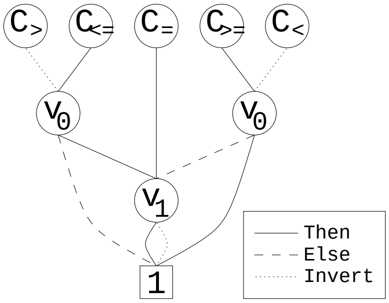


Figure 5.1: BDD representing a register-register comparison family.

Register-register families related using the three disjoint outcomes represented by this method must consist of comparisons which all use the same sign convention, either all signed or all unsigned. The equality operator can belong to either class, since signed equality is identical to unsigned equality. This restriction is allowed because there is no well-defined implication between signed and unsigned comparisons; thus, no additional accuracy could be garnered by combining the two representations. Furthermore, mixing of signed and unsigned comparisons of the same value is unlikely in real code.

A complete register-register family can be represented using two BDD variables, where a naïve implementation would require three (one for $=$ and $\neq$, one for $<$ and $\geq$, and one for $>$ and $\leq$). In addition, whereas the prior implementation recognized only a condition and its complement as being related, the presented implementation is capable of identifying all logical relationships among arbitrary sets of member conditions. Since the BDD required to represent the family is a standard form, the cost of the construction of these BDDs is linear in the number of register-register families.

The register-register family representation technique could be extended to be useful for floating point comparisons between two registers. Floating point comparison has four disjoint outcomes, including the three useful for integer comparison, and a fourth, *unordered*, which is unique to floating point due to its representing infinite and not-a-number (NaN) results. Since this results in up to twelve distinct expressed outcomes, a more complex but still very manageable BDD representation would be required.

5.2 Integer Register-Constant Comparison Families

The representation of integer register-constant comparison families is of the same general nature as the representation of register-register families. In this case, as well, disjoint outcomes are identified, of which the program's expressed outcomes can be built by set composition. Unlike register-register families, however, the number of disjoint outcomes in a given family depends on the particular expressed outcomes used in the program, and the construction of the BDD to represent the register-constant family is unique to each instance.

The first step in analyzing a family of register-constant comparisons is the identification of disjoint outcomes. A register-constant family consists of the comparison of a single variable value with various constant ones. The mechanism for describing disjoint outcomes of the comparisons is based on visualizing the conditions as splitting a number line into segments. The variable register value can be represented as the continuum, or number line, of integers which covers the entire range of numbers representable by the register. As a convenient convention, the number line is considered to extend from 0, with a binary representation of all '0' bits, to the maximum unsigned number representable in the register, denoted MAX_INT, having a binary representation of all '1' bits. Although this notation will be discussed in terms of unsigned integers, it applies equally well (and simultaneously) to signed integers. Additional useful constants include MAX_POS, which has a binary representation of all '1' bits, except for the most significant (sign) bit, which is a '0'; and MAX_NEG, which has a '1' in its most significant (sign) bit, and all other bits '0'. These constants are useful in mapping signed comparisons onto the unsigned number line, as will be demonstrated.

5.2.1 Disjoint outcome identification

Consider the number line $\mathcal{L}$ to be a set of disjoint intervals of the form $I \equiv [i_{low}, i_{high}]$. Thus, a number line of n intervals is the set $\mathcal{L} = \{I_0, I_1, ...I_{n-1}\}$. Furthermore, stipulate that taking the intervals themselves to be sets of the numbers they include, the union of the sets exhausts all representable numbers. In other words, any given integer representable in the register must belong to exactly one interval.

Now consider the set of numbers which, when present in the family's integer register, cause a particular condition C_j to evaluate to $\mathbf{T}$. Let us represent this set of numbers as the true set $\mathcal{T}(C_j) = \{I_0, I_1, ...I_{m-1}\}$, a set of disjoint intervals that covers the true set exactly. Table 5.1 shows the true sets for all signed and unsigned comparisons. Since unsigned and signed comparisons are treated uniformly in this system, they can

Table 5.1: True set intervals for comparisons to constants.

Operator	Constant sign	True set intervals
$=$	$+/-$	$[c, c]$
$\neq$	$+/-$	$[0, c-1], [c+1, \texttt{MAX\_INT}]$
$>_s$	$+$	$[c+1, \texttt{MAX\_POS}]$
	$-$	$[0, \texttt{MAX\_POS}], [c+1, \texttt{MAX\_INT}]$
$\geq_s$	$+$	$[c, \texttt{MAX\_POS}]$
	$-$	$[0, \texttt{MAX\_POS}], [c, \texttt{MAX\_INT}]$
$<_s$	$+$	$[0, c-1], [\texttt{MAX\_NEG}, \texttt{MAX\_INT}]$
	$-$	$[\texttt{MAX\_NEG}, c-1]$
$\leq_s$	$+$	$[0, c], [\texttt{MAX\_NEG}, \texttt{MAX\_INT}]$
	$-$	$[\texttt{MAX\_NEG}, c]$
$>_u$	$+$	$[c+1, \texttt{MAX\_INT}]$
$\geq_u$	$+$	$[c, \texttt{MAX\_INT}]$
$<_u$	$+$	$[0, c-1]$
$\leq_u$	$+$	$[0, c]$

(Ill-defined intervals $[i_l, i_h]$ where $i_h < i_l$ are defined to be $\emptyset$.)

be intermingled in a comparison family without modification or loss of accuracy. At most two intervals are included in the true set of each comparison. For convenience in description, intervals $[i_l, i_h]$ where $i_h < i_l$ are defined to be $\emptyset$. These may occur for some values of c, and must be avoided explicitly in a code implementation.

The true set interval description for comparisons is used to divide the continuum $\mathcal{L}$, initialized to $\{[0, \texttt{MAX\_INT}]\}$, into intervals representing the disjoint outcomes of the comparison family. Figure 5.2 gives pseudo-code for the algorithm which iterates over the true sets of the family's conditions, splitting the intervals of $\mathcal{L}$ as necessary. At completion of the algorithm, the number line $\mathcal{L} = \{D_0, D_1, ...D_{i-1}\}$, where there are i disjoint outcome intervals D. The significant property of this set of intervals is that for any $C \in F$, where F is the set of conditions related by the family, $\mathcal{T}(C)$ can be

```
foreach $C \in F$ do
   foreach $I \in \mathcal{T}(C)$ do
      $S$ := find_interval_containing($\mathcal{L}$, $I \rightarrow low$)
      if ($I \rightarrow low$ != $S \rightarrow low$)
         $\mathcal{L}$ := split_interval($\mathcal{L}$, $S$, $I \rightarrow low$)
      $S$ := find_interval_containing($\mathcal{L}$, $I \rightarrow high$)
      if ($I \rightarrow high$ != $S \rightarrow high$)
         $\mathcal{L}$ := split_interval($\mathcal{L}$, $S$, $I \rightarrow high$ + 1)
   done
done
```

Figure 5.2: Algorithm generating disjoint outcomes in a register-constant family.

constructed as the union of disjoint outcome intervals D_j. This is the same relationship between the expressed outcomes and the disjoint outcomes in the register-register case, and leads to a finite domain BDD representation, as before.

5.2.2 Finite domain representation

For query support and for inclusion in the predicate analysis BDD, the information about the membership of the disjoint outcomes in the expressed outcome sets must be encoded in BDD form. Again, this is accomplished using the technique of finite domains [35]. Here, unlike in the register-register case, the domain can be of varying size, depending on the conditions belonging to the family. Consider a family having i disjoint outcomes, such that $\mathcal{L} = \{D_0, D_1, \ldots D_{i-1}\}$.

The finite domain technique creates a Boolean n-space and subdivides that space completely between the elements of the set to be represented. The size of the space must be sufficiently large to accommodate all the set's members. In this case, $n = \log_2(i)$

```
for $j$ = 0..$i - 1$ do
   $d_j$ := BDD_One();
done
$extra\_vertices$ = $2^n$ - $i$
for $j$ = 0..$extra\_vertices$ - 1 do
   for $k$ = 0..$n - 2$ do
      if ($j$ & (1 << $k$))
         $d_j$ := BDD_Ite($v_k$,$d_j$,0)
      else
         $d_j$ := BDD_Ite($v_k$,0,$d_j$)
   done
done
for $j$ = $extra\_vertices$..$i - 1$ do
   for $k$ = 0..$n - 1$ do
      if ($j$ & (1 << $k$))
         $d_j$ := BDD_Ite($v_k$,$d_j$,0)
      else
         $d_j$ := BDD_Ite($v_k$,0,$d_j$)
   done
done
```

Figure 5.3: Algorithm representing disjoint outcomes in a variable-size finite domain.

BDD variables, $v_0, v_1 \ldots v_{n-1}$, are allocated to form the space $\{0,1\}^n$. This space is allocated exhaustively to i BDD nodes $d_0, d_1, \ldots d_{i-1}$ which represent the disjoint outcomes $D_0, D_1, \ldots D_{i-1}$ using the algorithm of Figure 5.3. The importance of exhaustively mapping the space, which is accomplished by assigning up to half the disjoint outcome nodes an extra vertex in the Boolean n-space, is that in analysis, the disjunction of all the disjoint outcome nodes must return **T**. This should be the case, for example, if the disjunction `(r1 < 1)||(r1 >= 1)` is computed. The finite domain allocation algorithm runs in $O(i \log_2 i)$ time and generates $\log_2 i$ BDD variables.

```
foreach $C_i$ in $F$ do
    $c_i$ := BDD_Zero();
    foreach $D_j$ in $\mathcal{L}$ do
        if ($D_j \subseteq \mathcal{T}(C_i)$)
            $c_i$ = BDD_Leq($d_j$, 1, $c_i$)
    done
done
```

Figure 5.4: Algorithm mapping expressed to disjoint outcomes in a finite domain.

The disjoint outcome nodes having been created, the nodes representing the expressed outcomes (the conditions represented by the family) are simply generated by computing for each a node that is the disjunction of the disjoint outcomes comprising the expressed outcome set. A simple $O(i^2)$ algorithm for accomplishing this is shown in Figure 5.4. It would be possible to incorporate this step into the procedure which divides the number line into disjoint outcomes, by dynamically adjusting the membership of the expressed outcome sets as the continuum is further divided, but since families consist typically of no more than sixteen conditions, this additonal implementation complexity is probably not warranted.

This completes the process of register-constant family formation. Figure 5.5 summarizes this procedure in four concise steps: creation and initialization of data structures, determination of disjoint outcomes and true set memberships, finite domain creation, and assignment of conditions to sets of outcomes in the finite domain BDD.

Consider the number line $\mathcal{L}$ to be a set of intervals $I_j \equiv [i_{j,lower}, i_{j,upper}]$ such that each integer $i \in [0, \texttt{MAX\_INT}]$ is a member of some I_j.

1. Initialize the number line $\mathcal{L}$ to be a single segment, represented in unsigned form as $[0..\texttt{MAX\_INT}]$
2. $\forall C_i$: Determine the true set $\mathcal{T}(C_i)$, defined to be the interval(s) of unsigned integers which yield an evaluation to $\mathbf{T}$. Split the intervals $I_j \in \mathcal{L}$ such that $\mathcal{T}(C_i)$ can be composed exactly as a union of discrete intervals of $\mathcal{L}$. Call the resulting disjoint outcome intervals $D_j \in \mathcal{L}$.
3. Create a BDD finite domain of size $||\mathcal{L}||$ and associate with each D_j a finite domain node f_j.
4. $\forall C_i$: Form the condition node n_i as the logical disjunction of the nodes f_j corresponding to the intervals D_j composing $\mathcal{T}(C_i)$.

Figure 5.5: Summary, formation of a family of register-constant conditions C_i.

5.2.3 Incorporation into PAS and PDLO

As described in Chapter 3, the condition analysis mechanisms were designed to work with two major IMPACT modules, the Predicate Analysis System and the Program Decision Logic Optimizer. The condition analysis techniques were designed to be easily interfaced with both PAS and PDLO.

Incorporation into PAS is simple. As described in Section 3.1, PAS builds the predicate BDD from variables enumerated for each conditon in the program. The condition analysis information is incorporated simply by replacing these nodes with the condition nodes formed as the roots of the finite domain BDDs formed in condition analysis, as shown in Figure 3.4. No additional query interface is necessary for incorporation of condition information into PAS, since the BDD automatically generates results based on the relations expressed in both layers of the combined BDD.

More than for its effect on improving the accuracy of predicate analysis, the condition analysis framework was developed to aid in PDLO. As previously described, the optimizer extracts from the predicate BDD sum-of-products expressions which give the logical definition of all useful predicates of the function to be optimized, within the topological context of analysis. These expressions are then optimized using traditional Boolean circuit minimization techniques and some specialized factorization algorithms, and are re-expressed as a new, more efficient predicate definition network. The condition analysis system is used in PDLO in two ways, first in conjunction with traditional Boolean optimization techniques to eliminate redundant comparisons from predicate definition expressions, and second, to more effectively acommodate factorization of the resulting expressions for efficient regeneration. More details about the implementation and performance of PDLO can be found in [1].

A predicate expression is presented to the optimizer as a sum-of-products expression, where the literals correspond to the conditions upon which the predicates are defined. This is referred to as the *complete back-substituted form* since it is the result of fully expanding predicate definition semantics using previously generated predicate results.

Optimization of these expressions is performed in PDLO by an iterative consensus technique [36]. This optimization, however, treats the condition variables as being independent of each other. Thus, a separate mechanism must be provided to optimize the resulting expressions with respect to condition relationships. This is accomplished in two phases. First, each conjunctive term of the sum-of-products expansion is examined to determine if it has any redundant literals. This is accomplished by examining each family having conditions represented in the literal. By querying the condition nodes of the family's BDD, the minimal set of conditions required to generate a logically equivalent term can be ascertained. Second, each sum-of-products expansion is examined for redundent terms which the condition information can prove are always covered by another term in the expression. Again, the BDDs provide the logical underpinnings for a mechanism which detects this case. Since these algorthims depend on matching elements of families or terms in an expression against possibly multiple other elements or terms, the algorithms used to perform these optimizations are computationally expensive, ranging from $O(n^2)$ to $O(2^n)$ in the worst case, but are fortunately applied only to relatively small sets of input. Furthermore, the savings incurred in traditional Boolean optimization time by reducing expressions by these means more than compensates for the time spent in condition optimization.

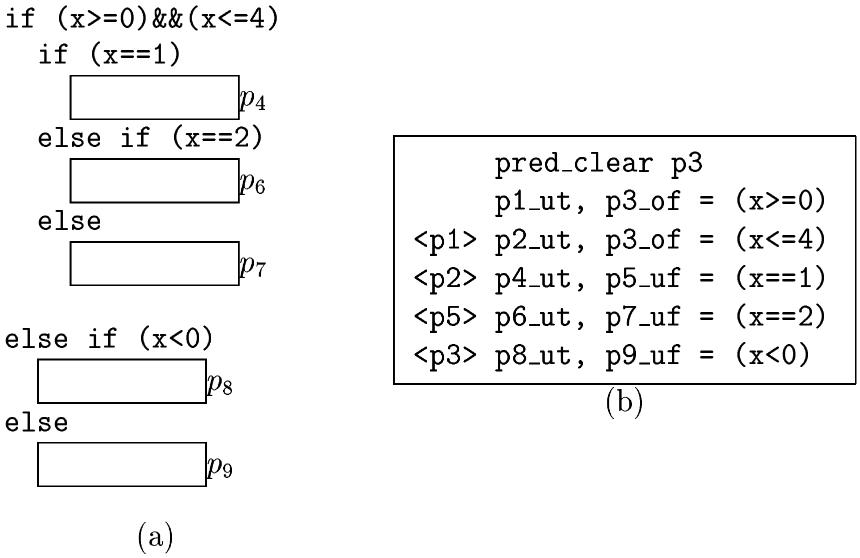
The second application of the condition analysis framework is to aid in factorization. In some cases, factorization of the predicate definition network regenerates dependence height to take advantage of computed subexpressions, thus reducing the number of comparisons required in evaluating the predicate definition network. In such cases, condition queries are sometimes required in order to recognize subexpressions which may be reused. For example, consider that condition information reveals that C_4 implies C_0. Suppose the subexpression $p_1 = C_0(C_2 + C_3)$ is available, and the generation of predicate $p_2 = C_4(C_2+C_3)$ is desired. The fact that p_2 can be formed as the factorization $p_2 = p_1C_4$ is observable only given the information that C_4 implies C_0. To detect these situations, the condition analysis interface is queried in the algorithm that matches factorable subexpressions. This can have a powerful enhancing effect on the reuse of available predicates when extensive factoring is applied.

One additional feature implemented in the condition analysis system which could be applied to PDLO is the ability to generate a new simpler condition based on expressions containing component conditions. For example, an expression such as $(x = 1)||(x > 1)$ could be restated as $(x >= 1)$ by using BDD operations first to identify the disjoint outcomes expressed in the complex condition, and then by generating a new simple condition to represent that same set of disjoint outcomes. This will be demonstrated further in a simple code example in the next section. This approach has not been applied in PDLO since it would require the generation of predicate definition instructions with new conditions which, due to the mulitple-output capability of such instructions in IMPACT, might incur some unanticipated expansion cost if the original component conditions were still used to compute other predicates. Furthermore, given the properties of condition families in the benchmarks studied, it was not anticipated that this feature would make a significant performance difference in any instances.

Another possibility is to perform decision logic optimization on a combined BDD, which includes the predicate and condition subtrees. In this case, the variables internal to condition finite domains would be exposed to optimization. Thus, the Boolean techniques applied to reduce predicate expressions would also perform the optimization now done as separate condition optimization steps. After optimization, however, another stage of processing would be required to regenerate representable simple conditions from the optimal, but complex, conditions rendered for each family. Furthermore, this extension could cause more variables and more freedom to be exposed to the optimizer, resulting in slower optimization times. For these reasons, this otherwise attractive option was not pursued.

5.2.4 Code example

The application of the integer register-constant comparison description mechanism can best be understood by means of a simple example. Figure 5.6(a) shows a simple region of code which, for our purposes, we will assume to be predicated using the typical IMPACT hyperblock predicated compilation process, resulting in a single hyperblock containing the indicated control blocks. As as result of this process, the instructions in each block are guarded with the predicate indicated to its right. Figure 5.6(b) shows the predicate definition structure resulting from if-conversion. It is further assumed that this



```
if (x>=0)&&(x<=4)
  if (x==1)
```

```
  else if (x==2)
```

```
  else
```

```
else if (x<0)
```

```
else
```

(a)

```
     pred_clear p3
     p1_ut, p3_of = (x>=0)
<p1> p2_ut, p3_of = (x<=4)
<p2> p4_ut, p5_uf = (x==1)
<p5> p6_ut, p7_uf = (x==2)
<p3> p8_ut, p9_uf = (x<0)
```

(b)

Figure 5.6: Integer register-constant comparison family code example.

region contains no assignments to the condition variable x. Since this region contains no such assignments and is acyclic, it follows that the five instances of comparisons based on the value stored in x are members of a single register-constant comparison family. The remainder of this discussion applies the procedure described in the previous part and summarized in Figure 5.5.

The algorithms presented in Section 5.2 are applied. First, disjoint outcomes are derived from the member conditions using the true sets given in Table 5.1 and the algorithm of Figure 5.2. The resulting disjoint outcome set is shown on the corresponding number-line in Figure 5.7. Individual disjoint outcomes are labelled "D$_n$." The mapping of nodes to the finite domain's Boolean 3-space and the *ITE* expressions used to construct nodes representing the family's conditions, as they would be generated by the algorithm of Figure 5.4, are shown in Figure 5.8. Here, in the interest of compactness, *ITE* expressions

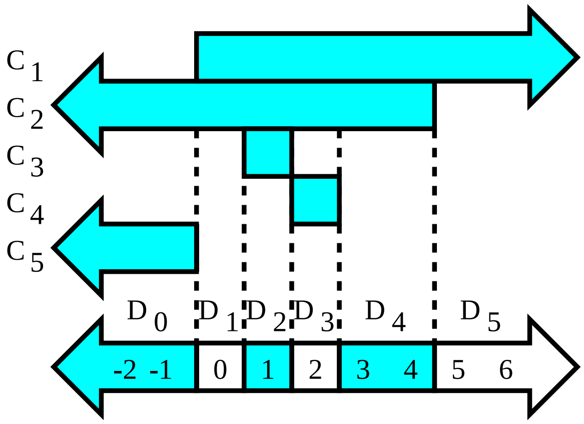


Figure 5.7: Integer register-constant comparison family number line.

	(v_2, v_1, v_0)	*ITE* Representation
d_0	$(0, 0, -)$	$v_2?0 : (v_1?0 : 1)$
d_1	$(0, 1, -)$	$v_2?0 : (v_1?1 : 0)$
d_2	$(1, 0, 0)$	$v_2?(v_1?0 : (v_0?0 : 1)) : 0$
d_3	$(1, 0, 1)$	$v_2?(v_1?0 : (v_0?1 : 0)) : 0$
d_4	$(1, 1, 0)$	$v_2?(v_1?(v_0?0 : 1) : 0) : 0$
d_5	$(1, 1, 1)$	$v_2?(v_1?(v_0?1 : 0) : 0) : 0$

Condition	*ITE* Representation
C_1 `(x>=0)`	$d_1?1 : (d_2?1 : (d_3?1 : \ldots$ $\ldots (d_4?1 : d_5)))$
C_2 `(x<=4)`	$d_1?0 : (d_1?1 : (d_2?1 : \ldots$ $\ldots (d_3?1 : d_4)))$
C_3 `(x==1)`	d_2
C_4 `(x==2)`	d_3
C_5 `(x<0)`	d_0

Figure 5.8: Integer register-constant comparison family BDD *ITE* expressions.

are shown in the ternary operator form familiar to C programmers:*(condition)?(true-evaluation-result):(false-evaluation-result).* This complete, a BDD node is created for each condition, which can be referenced in the predicate BDD or used in conjunction with other condition nodes to minimize the number of condition evaluations required in generation of a predicate value.

Examining Figure 5.6(b), it is shown that if-conversion generates a predicate definition network with a height of four to compute the predicates requried to guard the labelled blocks of code. Optimization of this code using condition information is now

$$\boxed{\begin{array}{lcl} p_4 & = & C_1\,C_2\,C_3 \\ p_6 & = & C_1\,C_2\,\overline{C_3}\,C_4 \\ p_7 & = & C_1\,C_2\,\overline{C_3}\,\overline{C_4} \\ p_8 & = & (\overline{C_1} + C_1\,\overline{C_2})\,C_5 \\ p_9 & = & (\overline{C_1} + C_1\,\overline{C_2})\,C_6 \end{array}}$$

(a) Before optimization

$$\boxed{\begin{array}{lcl} p_4 & = & C_1\,C_2\,C_3 \\ p_6 & = & C_1\,C_2\,\overline{C_3}\,C_4 \\ p_7 & = & C_1\,C_2\,\overline{C_3}\,\overline{C_4} \\ p_8 & = & (\overline{C_1} + \overline{C_2})\,C_5 \\ p_9 & = & (\overline{C_1} + \overline{C_2})\,\overline{C_5} \end{array}}$$

(b) Optimized without condition information.

$$\boxed{\begin{array}{lcl} p_4 & = & C_3 \\ p_6 & = & C_4 \\ p_7 & = & C_1\,C_2\,\overline{C_3}\,\overline{C_4} \\ p_8 & = & C_5 \\ p_9 & = & \overline{C_2} \end{array}}$$

(c) Optimized using condition information.

Figure 5.9: Predicate optimization with condition information.

considered, but for reasons of human readability, further examination of this example will be performed in the context of Boolean expressions rather than BDDs. Figure 5.9(a) shows the logical expressions for essential predicates in terms of the condition variables, as rendered by a full back-substitution into the predicate definition network created by if-conversion. Boolean optimization of these expressions generates only two optimizations, those being applications of consensus in the expressions for predicates p_8 and p_9, as indicated in Figure 5.9(b).

Awareness of condition relations allows much further optimization, as shown in Figure 5.9(c). In this example there is a great deal of redundancy in the comparisons performed, which is typically a remnant of control structures inserted by the programmer and perpetuated by branching control. In this case, for example, the programmer has tested to see if `x` lies within the range `[0..4]`, and within that range handles individual cases (Figure 5.6(a)). When the code segment is if-converted, the original range check

```
pred_set p7
p7_at, p8_uf = (x>=0)
p7_at, p9_uf = (x<=4)
p4_ut, p7_af = (x==1)
p6_ut, p7_af = (x==2)
```

Figure 5.10: Re-expressed optimized predicate definition network.

remains in the predicate definition structures for p_4 and p_6, although it is no longer useful in guarding those two blocks. In both those cases, a condition expressed in the predicate's expression (in the case of p_4, C_3 or `(x==1)`) implies the conditions used in the enclosing guard (`(x>=0)&&(x<=4)`). In this and similar cases, these condition evaluations, and in many cases intermediate predicates which held their results, can be removed.

Figure 5.10 shows the re-expressed optimized predicate definition network. Here the number of comparisons required is reduced by one, and the dependence height of the predicate definition network is reduced from four cycles to a single cycle. This dramatic reduction in decision complexity is largely due to the ability of condition analysis in assisting predicate optimization to remove redundant nested control.

6. EXPERIMENTAL RESULTS

This chapter describes the results of experimentation using the framework presented in this thesis within the context of the IMPACT research compiler. The results and characteristics reported are for a cross-section of benchmarks, including Unix utilities, SPEC CINT 92 and SPEC CINT 95 benchmarks. Since the analysis mechanism and the program decision logic optimizer operate on code which already contains predication, the starting point for the exercises shown is the highest quality of hyperblock code generated by the IMPACT research compiler, with function inlining, pointer disambiguation, and classical and ILP optimizations fully applied. Included in the ILP optimizations applied are loop unrolling and induction variable optimizations, which tend to increase the applicability of condition analysis techniques.

6.1 Characteristics of Benchmark Condition Families

An examination of code prepared in the manner described reveals a significant penetration of the types of conditions on which condition analysis can be effective. Analyzed here are the benchmarks *008.espresso*, *022.li*, *023.eqntott*, *026.compress*, *072.sc*, *124.m88ksim*, *129.compress*, *130.li*, *cccp*, *cmp*, *eqn*, *grep*, *lex*, *qsort*, *wc*, and *yacc*. Each of these was compiled for predicated execution using the aforementioned techniques. The benchmarks were then executed, and profile information was gathered using standard reference inputs.

One useful statistic for gauging the applicability of condition analysis is the percentage of predicate definitions whose conditions fall into multiple-member families, and are thus subject to logical description by these mechanisms with respect to other family members. Another statistic is the typical size of these families. Larger families typically provide more opportunity for optimization. Using profile information can yield dynamic results, weighting influential, high-trip-count parts of the program more than infrequently executed parts. Normalizing the number of total comparisons computed across the benchmarks, we find that 30% of dynamically executed conditions fall into families of two or more members. The distribution for families up to a size of 10 is shown in Figure 6.1. Families of size greater than 10 conditions are typically rare. The largest family encountered was in *023.eqntott*, with a size of 33 members. Some benchmarks had significantly higher-than-average proportions of relatable conditions. Notably, 83% of *wc*'s dynamic conditions are members of families of size six, and in *130.li*, over half of

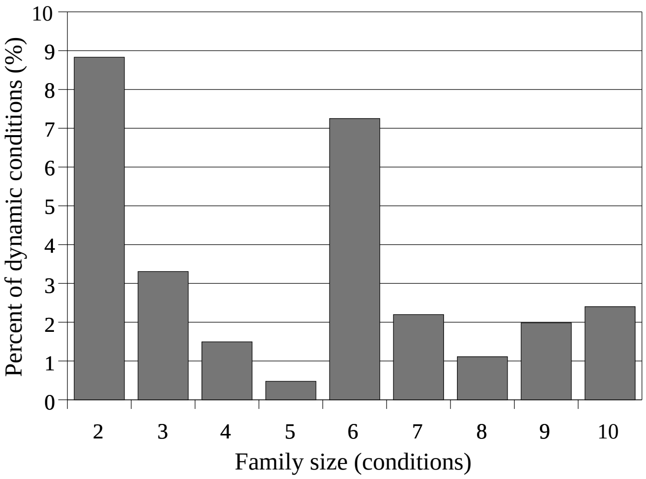


Figure 6.1: Dynamic prevalence of family-member conditions, by family size.

conditions are included in 9-10 member families. Other benchmarks were not particularly amenable to condition analysis, including *023.eqntott* and *yacc*, in which less than 1% of condition evaluations were relatable.

It should be noted that the results presented here are dependent on a particular compilation path having been taken within a particular research compiler environment, but that the general character of the results could be expected to hold in another ILP compilation framework which applies predication effectively.

6.2 Effectiveness of Program Decision Logic Optimization

It is difficult to separate the function of the condition analysis subsystem from the operation of the program decision logic optimizer due to the fact that the latter's efficient and useful operation depends heavily on the former [1]. Previous sections have made the argument that program decision logic optimization requires condition analysis to avoid the propagation and often duplication of unnecessary comparison instructions. Previous code examples also demonstrated that significant optimizations beyond the capability of Boolean techniques alone are enabled by the addition of the condition relation description framework. Finally, the results of the previous section indicate the prevalence of multiple-member families in important code segments of the benchmarks studied, further suggesting that condition analysis is a critical component of program decision logic optimization. Figure 6.2 shows the benchmark speedups achieved with the Program Decision Logic Framework earlier described. The two sets of results presented are for, first, an eight-issue, in-order processor with register interlocking, and, second, for a hypothetical machine of the same description, except in which 256 predicate definition instructions can be executed in each cycle, exclusive of the eight normal instruction issues. The results show a significant speedup for several of the benchmarks, and reveal the potential for further gains if the number of predicate definitions used could be reduced further. For more details, the interested reader is referred to [1].

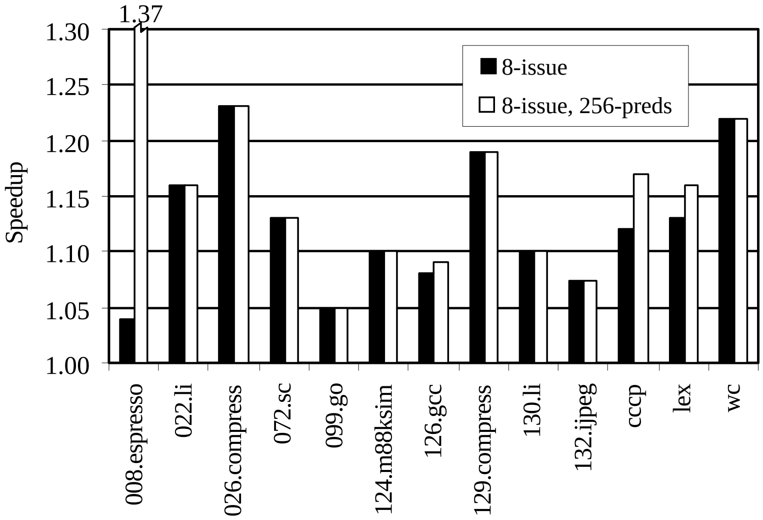


Figure 6.2: Speedup due to program decision logic optimization [1].

7. CONCLUSIONS

This thesis has presented techniques for extracting the logical relationships between the conditions in a low-level code representation and for representing these relationships for use in a predicate analysis and optimization environment. The mechanisms described have been implemented in the IMPACT research compiler and have been used successfully in improving the accuracy of predicate analysis. This work was key in developing the Program Decision Logic Optimization Framework [1], as it allows free transformation of predicate definition networks in regions where conditions are populous and strongly related. Prime examples of this significance were presented in Sections 3.3 and 5.2.4.

Also discussed were potential extensions to the described framework, including the possibility of extending logical analysis to cover relationships between different definitions throughout chains of computation. While these extensions might have some limited value in a few special cases, it is not clear that they would be generally useful and so have not been pursued aggressively. While this work has largely reached its limit insofar as its application to simple condition analysis, the techniques and principles described could be applied to other important problems in compilation. The range analysis framework could be recast in a more general sense to describe the values possibly residing in registers at various points in the control flow graph. This has applications to redundant bounds-check removal, advanced dead code removal, and code specialization. Such an adaptation raises interesting representational issues, since it might be desired not only to describe contiguous ranges of numbers, but also odds and evens, or even more complex sets. Furthermore, understanding of the effect of computation chains on values and representation of issues like register overflow would need to be addressed. This thesis has laid the groundwork for some of these issues, and provides a basic environment in which early efforts in these directions can be accommodated.

The second direction in which this work points is to a greater scope of program analysis. The combination of effective analysis of control, combined with the new forms of data value flow analysis suggested previously, may provide an extremely powerful way of analyzing code. Such a tool could have useful applications not just in compiler transformation of code for ILP execution, but also in compiler-based specialization of hardware to perform selected parts of algorithms and in preparation of software for more effective run-time optimization or problem specialization. Interesting representational and usability issues abound in pursuing this goal, some of which were just hinted at in Section 3.1.1. Furthermore, a mixed-mode analysis for developing data- and control-flow information in conjunction with each other features an interesting interplay between control and data flow. An analysis system that could understand this interplay and effectively represent it would blur traditional compiler boundaries, perhaps opening the way to a new, highly semantic-oriented, understanding of compilation. It is my intent to continue to pursue this conceptual progeny of this work in my Ph.D. research.

APPENDIX A. PREDICATION IN THE IMPACT ARCHITECTURE

The IMPACT EPIC predication model contains a set of predicate registers, predicate definition instructions, and semantics implementing predicated execution. The architecture is defined to have 64 independently addressable single-bit predicate registers and a predicate definition instruction which takes a source predicate, computes a condition, and deposits a result in each of two predicate registers according to the deposit type specified for each destination. Predicate register 0 is defined always to hold the value **1**.

The predicate definition instruction $\langle guard \rangle$ `pred_`*comparison* $dest_0$`,` $type_0$`,` $dest_1$`,` $type_1$`,` src_0`,` src_1 computes the condition $C = src_0 \; comparison \; src_1$ and may assign values to each $dest_i$ according to $type_i$, C, and the guarding predicate *guard*, as shown in Table A.1.

The IMPACT EPIC architecture defines six ways in which the result of a predicate/condition computation can be deposited into a destination predicate register. These six deposit types can be used to implement a variety of logical computations on the set of conditions and input predicates. The first four types are as defined in the HP Labs

Table A.1: IMPACT EPIC predicate deposit types.

$guard$	C	UT	UF	OT	OF	AT	AF	$\vee T$	$\vee F$	$\wedge T$	$\wedge F$	CT	CF
0	0	0	0	-	-	-	-	-	1	0	0	-	-
0	1	0	0	-	-	-	-	1	-	0	0	-	-
1	0	0	1	-	1	0	-	1	1	0	-	0	1
1	1	1	0	1	-	-	0	1	1	-	0	1	0

PlayDoh Specification [11]. The and-, or-, conjunctive-, and disjunctive-type predicate deposits are referred to as being parallel compares since multiple and- and or-type definitions can commit to a single (previously initalized) predicate register simultaneously. While if-conversion generates only unconditional and or-type definitions [29], the other types are useful in optimization of predicate definition networks, so an effective predicate analysis system should fully support their general use. Since the IMPACT EPIC predication model here described is a superset of the Intel IA-64 predication model, techniques presented here for analysis of IMPACT-style predication are directly applicable to IA-64 [6].

REFERENCES

[1] 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.

[2] J. H. Crawford, "The i486 CPU: Executing instructions in one clock cycle," *IEEE Micro*, vol. 10, pp. 27–36, February 1990.

[3] M. Forsyth, S. Mangelsdorf, E. Delano, C. Gleason, and J. Yetter, "CMOS PA-RISC processor for a new family of workstations," in *Proceedings of COMPCON*, February 1991, pp. 202–207.

[4] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan, "Superscalar instruction execution in the 21164 Alpha microprocessor," *IEEE Micro*, vol. 15, pp. 33–43, April 1995.

[5] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 290–302.

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

[7] 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.

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

[9] 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.

[10] 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.

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

[12] B. Cheng, "A profile-driven automatic inliner for the impact compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1997.

[13] S. Anik, "Architectural and software support for executing numerical applications on high performance computers," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[14] Y. Yamada, "Data relocation and prefetching for programs with large data sets," Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[15] B. C. Cheng and W. W. Hwu, "A practical interprocedural pointer analysis framework," IMPACT, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-99-01, 1999.

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

[17] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, no. 1, pp. 229–248, January 1993.

[18] 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.

[19] D. I. August, "Hyperblock performance optimizations for ILP processors," M.S. thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.

[20] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[21] R. A. Bringmann, “Compiler-controlled speculation,” Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[22] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, “The importance of prepass code scheduling for superscalar and superpipelined processors,” *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 353–370, March 1995.

[23] N. J. Warter, “Modulo scheduling with isomorphic control transformations,” Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[24] D. M. Gallagher, “Memory disambiguation to facilitate instruction-level parallelism compilation,” Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[25] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, “Dynamic memory disambiguation using the memory conflict buffer,” in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1994, pp. 183–193.

[26] R. E. Hank, “Machine independent register allocation for the IMPACT-I C compiler,” M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

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

[28] 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.

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

[30] S. B. Akers, “Binary decision diagrams,” *IEEE Transaction on Computers*, vol. C-27, no. 8, pp. 509–516, June 1978.

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

[32] F. Somenzi, “CUDD: CU Decision Diagram package, release 2.30,” University of Colorado at Boulder, http://vlsi.colorado.edu/˜fabio/CUDD/, Tech. Rep., 1998.

[33] 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*, January 1990, pp. 40–45.

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

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

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