1 Introduction

1.1 Predicated execution

Predicated execution [HD86, RYYT89, DT93, KSR93] is an architectural model in which each operation is guarded by a boolean operand whose value determines whether the operation is executed or nullified. From the viewpoint of the instruction set architecture, the main new features are an additional boolean operand guarding each operation and a set of compare operations used to compute predicates. Frequent and unpredictable branches present a major barrier to exploiting greater amounts of instruction-level parallelism. Predicated execution addresses this issue by generalizing the rules for code motion between basic blocks and by increasing the size of straight line code blocks by eliminating many branches completely. Often, an entire acyclic control flow region can be converted into a single, branch-free block of predicated code. This process of replacing branches with appropriate predicate computations and guards is called *if-conversion* [AKPW83, DHB89, DT93, PS91, MLC+92].

Figure 1 illustrates the benefits of predicated execution and if-conversion. Conventional code for the program represented in Figure 1(a) is shown in Figure 1(b). Even assuming unit latencies, no mispredict penalty, and a sufficiently wide-issue machine, execution will take from four to six cycles depending on the path taken. If the branch latency increases or if there is a mispredict penalty, then execution time will increase. In Figure 1(c), the code has been if-converted. Explicit branches that control execution are replaced with guarding predicates on operations, together with compare operations which compute the appropriate guard values. The result is a single, branch-free block of predicated code. This code can be executed in three cycles, independent of branch latency or mispredict penalty.

The primary benefit of if-conversion is that it enlarges scheduling scope without dramatically increasing code size; thus, predication offers an attractive alternative to increasing block size through code replication [MLC+92]. A secondary benefit of predication is the elimination of unpredictable branches that would otherwise cause branch prediction stalls at run-time [Tys94, MHB+94].

1.2 Architectural model

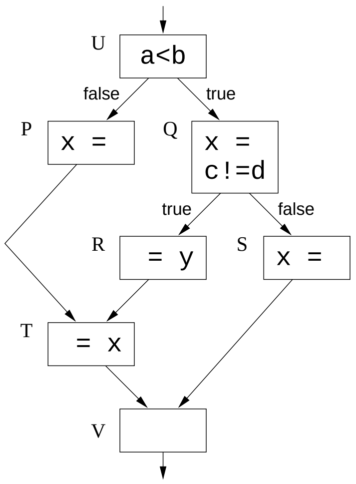
Examples described in this paper are illustrated using the HPL PlayDoh research architecture [KSR93]. This architecture provides support for predicated execution through an enhanced version of the predication capabilities of the Cydra 5 processor [DHB89, RYYT89]. In addition to guarded operations, PlayDoh introduces a family of compare operations which support efficient if-conversion of structured and unstructured control flow graphs, as well as parallel computation of high fan-in logical expressions. A compare operation has the following form.

```
p1,p2 = cmpp.<d1>.<d2> (r1 <cond> r2) if q
```

Each such operation is a two-target compare with the following interpretation.

- `p1`, `p2` are destination predicate register operands.
- `cmpp` is the generic compare opcode.
- `<d1>`, `<d2>` are two-letter descriptors that specify an action and mode for the compare operation. PlayDoh includes unconditional (`u`), conditional (`c`), parallel-or (`o`), and parallel-and (`a`) actions, specified by the first letter of the descriptor. Each action has both a normal mode (`n`) and a complement mode (`c`), specified by the second letter of the descriptor.
- `r1 <cond> r2` specifies the actual comparison; the set of comparisons is the same as in the HP PA-RISC architecture [Hew92].
- `q` is a source predicate register operand. Although `q` appears as a guarding predicate, it can be viewed as an input to the compare operation.

These two-target compare operations support the generation of high-quality predicated code as produced by if-conversion of structured or unstructured regions, parallel computation of logical expressions, and



(a) control flow graph

```
U: l = cmp (a<b)
   branch l, Q
P: x =
   branch T
Q: x =
   m = cmp (c!=d)
   branch m, R
S: x =
   branch V
R:   = y
T:   = x
V: ...
```

(b) sequential code

```
U: p,q = cmpp.uc.un (a<b)
   t = cmpp.uc (a<b)
   x =          if p
   x =          if q
   r,s = cmpp.un.uc (c!=d) if q
   t = cmpp.on (c!=d) if q
   x =          if s
     = y        if r
     = x        if t
V: ...
```

(c) if-converted code

unit 1	unit 2	unit 3	unit 4
p,q = cmpp.uc.un (a<b)	t = cmpp.uc (a<b)		
r,s = cmpp.un.uc (c!=d) if q	t = cmpp.on (c!=d) if q	x = if p	x = if q
	= y if r	x = if s	= x if t

(d) scheduled code (predicate-sensitive analysis)

unit 1	unit 2	unit 3	unit 4
p,q = cmpp.uc.un (a<b)	t = cmpp.uc (a<b)		
r,s = cmpp.un.uc (c!=d) if q	t = cmpp.on (c!=d) if q	x = if p	
	= y if r		x = if q
		x = if s	
			= x if t

(e) scheduled code (conventional analysis)

Figure 1: Example of predicated code

input predicate	result of compare	`un`	`uc`	`on`	`oc`
0	0	0	0	-	-
0	1	0	0	-	-
1	0	0	1	-	1
1	1	1	0	1	-

Table 1: Behavior of compare operations

height-reduction of control and boolean data expressions. For this paper, we are mainly concerned with unconditional and parallel-or actions, which are the actions used in if-conversion. Table 1 shows the execution behavior of the unconditional and parallel-or actions in both normal and complement modes. Each entry describes the result on the destination predicate; note that the destination may be assigned a value or may be left untouched (denoted as "-"). From the table, we see that an unconditional compare operation always writes a value into its destination register. The parallel-or compare operation only writes a value if both its guarding predicate and compare condition are satisfied. This form can be used to efficiently compute a disjunction by accumulating terms into a predicate register that was initially cleared. Since the operations computing the terms write the same value, they can execute in any order or even in parallel.

1.3 Predicate-sensitive analysis

To best exploit available instruction-level parallelism, the predicated code must be scheduled and its variables must be bound to physical registers. Both scheduling and allocation require predicate-sensitive analysis to generate the highest quality code. To illustrate this point, let's reconsider our example in Figure 1. Prior to scheduling, the compiler constructs precedence constraints between operations that are satisfied by any legal schedule. This is done separately for each variable in the program. Consider the variable `x` in the if-converted code of Figure 1(c). There are three assignments to `x` (guarded by predicates `p`, `q` and `s`) and a single use of `x` guarded by predicate `t`. To achieve the schedule in Figure 1(d), the compiler must determine that the first two writes to variable `x` in Figure 1(c) are independent. Similarly, it must determine that the third write to `x` is independent of the read of `x`. Because predicates `p` and `q` cannot be true simultaneously, there is no output dependence edge between the first two assignments to `x`. Similarly, because predicates `s` and `t` cannot both be true, there is no flow dependence edge between the final assignment and the final use of `x`. We can see that data flow analysis must exploit relations between predicates in order to generate precise precedence edges, which are critical to ILP scheduling.

After precedence edges have been constructed, our compiler schedules code on actual machine resources to optimize the use of multiple pipelined function units. In this simple example, we show four functional units that execute any operation with unit latency. The scheduling process yields wide-word code similar to the code shown in Figure 1(d). This schedule is significantly better than the schedule formed without predicate-sensitive analysis, shown in Figure 1(e).

Once code is scheduled, variables must be bound to physical registers. A variable is live at a point if its value at that point can reach a subsequent use of the variable along some control flow path. In graph-coloring register allocation [Cha82], two variables are said to *interfere* if one variable is live at a point where the other is written. Variables that interfere must be bound to separate physical registers, so that the writes to one variable do not overwrite live values of the other. In predicated code, overlapping lifetimes do not necessarily interfere. We augment the data flow information at each point to indicate the predicate conditions under which each variable is live. Overlapping variable lifetimes may share a common register provided the lifetimes conflict under predicate conditions which are mutually exclusive and therefore guaranteed not to occur.

1.4 Overview

In this paper, we discuss techniques for direct analysis of predicated code. This work has been motivated by a variety of tasks within Elcor, our research compiler for instruction-level parallel machines. In particular, we use predicate-sensitive analysis for optimization, scheduling, and register allocation of predicated code.

This work addresses two basic problems. First, we address the problem of analyzing predicated code to discover relations that exist between predicates. A typical relation is that two predicates are disjoint (i.e. they are never both true at the same time), or that a predicate `p` is true whenever a predicate `q` is true (this is analogous to control flow dominance). We will present a predicate query system that answers basic questions about predicate relations. The queries are used during predicate-sensitive data flow analysis as well as by optimizations on predicated code. Predicated code is analyzed directly, so that no correspondence to the original control flow is required.

The second problem we address is predicate-sensitive data flow analysis. Here, the objective is to extend conventional bitvector-style methods to predicated code. Our approach builds upon the analysis of predicate relations, using the predicate query system to manipulate predicate expressions, which represent executions for which each data flow property holds. We use the methods presented here to perform scheduling and register allocation on predicated code, achieving very high quality results using efficient analysis algorithms.

The rest of the paper is organized as follows. In Section 2 we sketch the outline of our approach and provide necessary terminology. In Section 3, we describe a simple translation of PlayDoh code to a sequential, single assignment form. This form facilitates analysis of predicate relations. We give an algorithm for extracting local predicate relations directly from the code. In Section 4, we define the partition graph, a data structure for representing predicate relations and for manipulating predicate expressions. Efficient partition graph algorithms appear in Section 5; these algorithms form the basis for predicate-sensitive data flow analysis. The extension of conventional bitvector data flow analysis to predicate-sensitive analysis is discussed in Section 6.

2 Terminology

In the introduction, we motivated the need for predicate-sensitive analysis, where relations between run-time values of predicates are incorporated into data flow analysis. In this section we introduce the basic ideas underlying our approach.

2.1 From paths to executions

Data flow analysis computes facts about a program at each point in a control flow graph representation. These facts are often of the form, "property W holds on all paths through point A," or "property W holds on some path through point A." For example, a definition of variable `x` in operation i reaches operation j provided there is some path from i to j containing no redefinition of `x`. In Figure 1(a), we see that the definitions in blocks P and Q reach the use in block T, but the definition in block S does not reach the use in block T. This analysis is used to determine precedence constraints between operations prior to instruction scheduling.

In Figure 1(c), there is a single control flow path through the block. To determine which definitions of `x` actually reach the use, we must consider the effect of predicated execution. For the definition of `x` under predicate `p` to reach the use of `x` under predicate `t`, we need to know whether there exists an execution such that (1) both `p` and `t` are assigned true; and (2) predicates `q` and `s`, which guard the intervening definitions of `x`, are assigned false.

Intuitively then, questions about whether a property holds on some or all paths through a control flow graph become questions about whether the property holds on some or all executions through a sequence of predicated operations. In the remainder of this section, we develop terminology for reasoning about executions of predicated code sequences.

2.2 Execution sets

Predicate analysis, as presented here, focuses on relations between predicate values within straight line sequences of operations.

Definition 1 *A **predicate block** is a straight line sequence of predicated operations.*

Predicate blocks are typically formed by if-conversion of single-entry single-exit control flow regions. To capture relations between run-time values of predicates, we look at relations between different possible executions of a predicate block.

Definition 2 *An **execution trace** is a record of boolean values assigned to predicate variables during an execution of a predicate block. We use* $\mathbf{1}$ *to denote the set of all possible execution traces. An **execution set** for a predicate* `p`*, denoted* P*, is the set of traces in which predicate* `p` *is assigned true. Note that* $P \subseteq \mathbf{1}$ *for any predicate* `p`*. Other non-empty subsets of* $\mathbf{1}$ *are also execution sets, although they are not explicitly named.*

> **Example:** Consider the if-converted code in Figure 1(c). There are five predicate variables, and $2^5 = 32$ potential predicate assignments. Only three of these assignments are execution traces, corresponding to the three paths in the source control flow graph shown in Figure 1(a). These traces are listed below.
>
> | trace | p | q | r | s | t |
> |---|---|---|---|---|---|
> | e_1 | 1 | 0 | 0 | 0 | 1 |
> | e_2 | 0 | 1 | 1 | 0 | 1 |
> | e_3 | 0 | 1 | 0 | 1 | 0 |
>
> All other combinations of predicate assignments are invalid, since they are not possible in any execution. For example, it is not possible for both `p` and `q` to be assigned true in a single execution; this fact can either be determined from correspondence to the original control flow graph or from the semantics of the compare operation which assigns values to `p` and `q`. The execution sets are as follows: $P = \{e_1\}$, $R = \{e_2\}$, $S = \{e_3\}$, $Q = \{e_2, e_3\}$, $T = \{e_1, e_2\}$, and $\mathbf{1} = \{e_1, e_2, e_3\}$.

In the above example, we enumerated execution traces manually and then formed execution sets for each predicate. From the execution sets, we see simple run-time relations between predicates. For example, the fact that predicates `s` and `t` are never both true in any execution is evident from the fact that execution sets S and T are disjoint.

Clearly it is impractical to form execution sets and determine relations among them by enumerating a potentially exponential number of traces. Instead, we will derive relations between execution sets directly from the predicate-assigning operations. The algorithms for deriving relations between execution sets is given in Sections 3 and 4. The most basic relation is the partition relation.

2.3 The partition relation

Just as computing control dependence or dominance requires knowledge of control flow predecessors and successors (i.e. the graph connectivity), computing predicate relations requires basic information about local predicate properties. For example, consider the following code fragment.

```
m,n = cmpp.un.uc (i<k) if u
```

Locally, we can determine that several facts hold following execution of these statements.

- if `m` or `n` is true, then `u` must be true,
- if `m` is true, then `n` must be false and vice versa, and
- if `u` is true, then one of `m` or `n` is true.

These local facts are akin to knowing local connectivity in the control flow graph; for example, this code sequence would result from if-converting a basic block U having two successors M and N, where the condition `i<k` controls transfer to either M or N. From facts of this form, we will compute global relations akin to dominance and control dependence. The local relation between source and destination predicates induced by a compare operation can be expressed in terms of the predicates' execution sets:

- $M \cup N \subseteq U$,
- $M \cap N = \emptyset$, and
- $U \subseteq M \cup N$.

In other words, the execution set U is partitioned into disjoint sets M and N, denoted $U = M \mid N$.

2.4 Predicate expressions

During data flow analysis, we will need to reason about more general execution sets than the base executions sets corresponding to predicates. For example, during analysis we may need to represent the fact that a variable is live on all executions where either `p` or `s` is true. This set is represented by the union of sets P and S, denoted $P + S$. Such symbolic expressions representing general execution sets are called *predicate expressions*, defined below.

Definition 3 *A* **predicate expression** *is a symbolic expression that represents an execution set. The base symbols of predicate expressions are the names of execution sets for individual predicates. Base expressions are combined using the operators sum $(+)$, difference $(-)$, and product $(\cdot)$. A predicate expression is interpreted as the execution set formed by applying corresponding set operators to the base execution sets.*

In our example, the expression $P + S$ represents the execution set $P \cup S = \{e_1\} \cup \{e_3\} = \{e_1, e_3\}$. Whether P denotes a symbol or an execution set depends on the context of its usage.

3 Extracting local relations

Each predicate-assigning operation induces a local relation between the predicate variables being assigned. In general, each local relation can be represented symbolically as a partition relation between the predicates' execution sets. In this section, we describe an algorithm for directly analyzing predicated code and extracting local relations between predicates. The process has two steps. First, the code is translated to a sequential static single assignment intermediate form, which simplifies subsequent processing. Second, this intermediate form is processed to discover relations between predicates.

3.1 Translation to sequential SSA form

PlayDoh operation	sequential form
p = cmpp.un (r1 <cond> r2) if q	p = (r1 <cond> r2) · q
p = cmpp.uc (r1 <cond> r2) if q	p = !(r1 <cond> r2) · q
p = cmpp.on (r1 <cond> r2) if q	p = p + (r1 <cond> r2) · q
p = cmpp.oc (r1 <cond> r2) if q	p = p + !(r1 <cond> r2) · q

Table 2: Sequential forms of compare operations

To simplify subsequent processing, we translate the PlayDoh compare operations to a single-target sequential form; this translation is performed before converting to static single assignment form. Foremost, this translation replaces conditional-overwrite parallel-or operations with a read-modify-write sequential version. Table 2 shows the translation to sequential form, which will be used in the remainder of the paper. In the sequential form, the destination predicate value is a boolean function of the predicate input value and the compare condition.

Since the sequential compare operations execute unconditionally, converting them into static single assignment form within an if-converted region is as simple as performing SSA renaming of scalars within a basic block: for each predicate variable, maintain a counter which is incremented at every definition of that predicate, and use the current counter value as the SSA subscript at every reference of the predicate.

```
U: p,q = cmpp.uc.un (a<b)
   t = cmpp.uc (a<b)
   x =         if p
   x =         if q
   r,s = cmpp.un.uc (c!=d) if q
   t = cmpp.on (c!=d) if q
   x =         if s
     = ..y..   if r
     = ..x..   if t
V: ...
```

(a) original predicated code

```
U:  p = !(a<b) · true
    q =  (a<b) · true
   t1 = !(a<b) · true
    x =         if p
    x =         if q
    r =  (c!=d) · q
    s = !(c!=d) · q
   t2 = t1 + (c!=d) · q
    x =         if s
      = ..y..   if r
      = ..x..   if t2
```

(b) sequential SSA form

Figure 2: Example of translation to sequential single assignment form

Figure 2 shows the translation to single-target sequential SSA form applied to our running example. Note that `t1` and `p` are equivalent predicates. During the initial analysis of predicated code, we will detect such equivalences and map both predicates to a single symbolic predicate.

3.2 Gathering partition relations

Given a sequence of predicated operations in sequential SSA form, we scan the operations in order and perform a predicate-insensitive value numbering to match compare conditions and map predicate registers to predicate symbols. As compare operations are processed, we emit partition relations associated with the predicate symbols of the compare operands.

To perform value numbering, we maintain a counter for each variable. This counter is incremented at every definition and is appended as a suffix to subsequent uses of the variable. Compare conditions are normalized, with input operands being lexicographically ordered. The value number for an input operand consists of a variable name together with its counter. Note that PlayDoh compare operations have an additional degree of freedom during normalization, since negations can be pushed into the two-letter destination descriptor. For example, the comparisons $a < b$ and $a \geq b$ both normalize to the same form, although one will be complemented on output. Normalization allows us to detect equivalent and complementary compare conditions with hashing.

A hash table records the mapping between predicate names in the source and predicate symbols allocated during this phase. We use integers numbered from 1 as node names. Initially, the table contains the true predicate mapped to 1. Compare operations are processed in order, and the hash table is updated to keep track of the mappings from compare expressions to symbolic name and from symbolic names to predicate registers. At each compare operation, the right-hand side is reduced by normalizing the compare condition and replacing predicate operands with their symbolic name.

Figure 3 shows the algorithms for scanning a predicated block to extract local partition relations. The main routine is **scan_ops**, which iterates over compare operations in the block. As each operation is visited, the right-hand side compare operation and guarding predicate is normalized into a string that serves as a hash key. The functions **lookup_AND_string** and **lookup_OR_string** process keys from unconditional and parallel-or compare operations, respectively. Both functions return a symbolic name corresponding to the predicate variable computed by the compare operation. In both cases, if a name for the computation does not already exist in the hash table, an entry is created. As a result of adding entries to the hash table, new partition relations may be generated. Note that in the case of an parallel-or operation, we require that the previous predicate value and the new condition being “OR-ed” be disjoint. This is always the case for parallel-or compares generated by if-conversion.

```
lookup_AND_string (String S)
{
    // S has form [!](r1<cond>r2)·i
    if S not in table then
        create entry for S, say m;
        create entry for complement of S, say n;
        emit partition relation i = m | n;
    endif
    return symbol for S;
}
```

```
lookup_OR_string (String S)
{
    if S not in table then
        if S has form j+[!](r1<cond>r2)·i then
            let S' be the right-hand subexpr of S;
            k = lookup_AND_string (S');
            further reduce S to j + k;
        endif
        // S has form j + k
        create entry for S, say m;
        emit partition relation m = j | k;
    endif
    return symbol for S;
}
```

```
scan_ops (List ops)
{
    for each compare operation in ops list do
        reduce the right-hand side to a normalized string S
        if compare is unconditional form then
            m = lookup_AND_string (S);
        else // compare is or-style
            m = lookup_OR_string (S);
        endif
        map destination predicate to m;
    endfor
}
```

Figure 3: Algorithm for gathering initial partition relations

3.3 Example

Below is our example from Figure 2(b), with the predicate computations reordered to make things more interesting. Compares are normalized; other operations are omitted to save space.

```
p  = !(a<b) · true
q  =  (a<b) · true
t1 = !(a<b) · true
t2 = t1 + !(c=d) · q
s  =  (c=d) · q
r  = !(c=d) · q
```

Operations are processed in order, producing the following effects.

- The initial table maps `true` to 1 and creates a symbolic name for the execution set.
- `p = !(a<b)` · `true`. Reduces to `p` = `!(a<b)` · 1. During lookup, the right-hand side and its complement are added to the table and a partition relation is emitted:

symbol	string	source names	
1	`true`	`true`	
2	`!(a<b)` · 1	`p`	
3	`(a<b)` · 1		$\Rightarrow 1 = 2 \mid 3$

- `q = (a<b)` · `true`. Reduces to `q` = `(a<b)` · 1, which further reduces to `q` = 3.

  Action: add `q` to list of source names for symbol 3.
- `t1 = !(a<b)` · `true`. Reduces to `t1` = 2.

  Action: add `t1` to list of source names for symbol 2.
- `t2 = t1 + !(c=d)` · `q`. Reduces to `t2` = 2 + `!(c=d)` · 3. During lookup, `!(c=d)` · 3 and its complement are added to the table and a partition relation is emitted. The outer expression further reduces to `t2` = 2 + 4, which is added to the table and a partition relation is emitted:

symbol	string	source names	
1	`true`	`true`	
2	`!(a<b)` · 1	`p`, `t1`	
3	`(a<b)` · 1	`q`	
4	`!(c=d)` · 3		
5	`(c=d)` · 3		$\Rightarrow 3 = 4 \mid 5$.
6	$2 + 4$	`t2`	$\Rightarrow 6 = 2 \mid 4$.

- `s = (c=d)` · `q`. Reduces to `s` = 5.

  Action: add `s` to list of source names for symbol 5.
- `r = !(c=d)` · `q`. Reduces to `r` = 4.

  Action: add `r` to list of source names for symbol 4.
- The final table and the partitions emitted:

symbol	string	source names
1	`true`	`true`
2	`(a<b)` · 1	`p`, `t1`
3	`!(a<b)` · 1	`q`
4	`!(c=d)` · 3	`r`
5	`(c=d)` · 3	`s`
6	$2 + 4$	`t2`

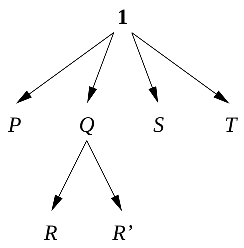
  Partitions: $1 = 2 \mid 3, \quad 3 = 4 \mid 5, \quad 6 = 2 \mid 4$.

```
if (a<b)
   S0;
else
   S1;
   if (m>n)
      S2;
if (i!=5)
   S3;
else
   S4;
```

(a) source code fragment

```
p =  (a<b) · true
q = !(a<b) · true
         S0         if p
         S1         if q
r =  (m>n) · q
         S2         if r
s =  (i!=5) · true
t = !(i!=5) · true
         S3         if s
         S4         if t
```

(b) sequential SSA form



(c) partition graph

Figure 4: Example of partition graph for structured code

4 The partition graph

Local partition relations must be combined to answer general questions about relations between predicates. For example, if we know the relations $\mathbf{1} = P \mid Q$ and $Q = R \mid S$, then we can deduce that P is disjoint from both R and S. The *partition graph* is a data structure that combines local partition relations in a form amenable to answering general queries about predicate relations. In this section, we define the partition graph and give its interpretation in terms of execution sets, and we show how the graph is built from local partition relations. Graph algorithms for answering general predicate queries are presented in Section 5.

4.1 Definition and interpretation

Definition 4 *A* **partition graph** *is a directed acyclic graph whose nodes represent execution sets and whose labeled edges represent partition relations between nodes. Specifically, a partition $U = M \mid N$ is represented by labeled edges $U \overset{r}{\to} M$ and $U \overset{r}{\to} N$, where the common label r indicates that these edges all belong to the same partition. A partition graph is* **complete** *if it contains a unique node having no predecessors from which all nodes are reachable.*

Figure 4 shows the partition graph for predicated code resulting from if-conversion of a structured program region. Nodes in the graph represent execution sets. To reduce clutter, we draw edges having the same label (representing a single partition) so that their tails touch, and then we omit edge labels. The root of the partition graph is the symbol $\mathbf{1}$, which represents the universal set of valid executions through this code. This set is partitioned two different ways, with one partition for each of the two sequentially composed if-then-else statements: $\mathbf{1} = P \mid Q$ and $\mathbf{1} = S \mid T$. Set Q is partitioned as $Q = R \mid R'$, where R' is created to complete the partition, even though there is no corresponding predicate computed in the source code. This graph is complete.

We extend several familiar tree properties to partition graphs as follows.

Definition 5 *The* ***root*** *of a complete partition graph is the unique node having no predecessors. A* ***leaf*** *is any node having no successors. An* ***ancestor*** *of a node P is any node on a path from $\mathbf{1}$ to $\boldsymbol{P}$. A* ***descendant*** *of node P is any node reachable from P. The* ***level*** *of a node is its shortest-path distance from the root. The root has level zero. A lowest common ancestor of a set of nodes $\{Q_i\}$, denoted lca($\{Q_i\}$), is a node with greatest level number from the set of common ancestors of $\{Q_i\}$.*

Each node of the partition graph corresponds to an execution set for some predicate. A set of edges having the same label represents a partition relation. Each individual edge represents a subset relation, as does any path. Thus, we can test non-local relations by traversing paths in the partition graph. This technique underlies our algorithms for efficiently analyzing predicated code. For example, to determine that predicates `p` and `r` are disjoint in Figure 4(c), we simply show that execution sets P and R are disjoint in the partition graph. This is shown by finding a partition $X = Y \mid Z$ such that P is a descendant (i.e. subset) of Y and R is a descendant (i.e. subset) of Z. In this example, the partition $\mathbf{1} = P \mid Q$ suffices. In contrast, there is no partition that separates P and S, which are not disjoint.

4.2 Graph construction

In this section, we describe our approach to building a partition graph directly from predicated code. This approach frees us from incrementally maintaining the partition graph as the predicated code is transformed, and it also removes many phase-ordering constraints from the compiler. Fundamentally, we believe the facility to work with predicate code as effectively as we can with ordinary code is of paramount importance to ILP compilers for architectures supporting predication. The process has two parts: (1) building an initial graph from local partition relations, and (2) completing the partition graph. These steps are detailed below.

Building the initial graph

The initial partition graph is constructed from a list of partition relations, which can be generated either from the source control flow graph prior to if-conversion or from analysis of the predicated code itself. By inserting a set of labeled edges to represent each partition relation, we get a partition graph.

One source of local partition relations is the control flow graph itself. Suppose predicates are assigned to the basic blocks of an acyclic control flow region just prior to if-conversion. We assume empty basic blocks are inserted as necessary to prevent edges from directly connecting branch and merge points, such as happens on the false side of an if-then statement. We generate partition relations in a single traversal of the control flow region as follows. If a block with predicate $\mathtt{r}$ has successor (predecessor) blocks with predicates $\mathtt{p}_i$ through $\mathtt{p}_k$, we generate the relation $R = P_i \mid \ldots \mid P_k$. During if-conversion, predicate computations are introduced to replace branches, and the predicates assigned to blocks are used in the predicate computations and as guards. The main advantage of this approach is the ease of generating partition relations; the main disadvantage is that the partition graph must be correctly maintained as the predicated code is transformed, since the correspondence to the original control flow region may be lost.

However, not all predicated code is generated via if-conversion. Predicated code may be introduced through inlining of hand-coded intrinsic functions, through schematic transformations such as control critical path reduction [SK95], and through optimizations that make use of predicated execution. In each of these cases, it is essential that we be able to analyze the predicated code directly, since no corresponding, predicate-free program representation exists. The approach presented here is to gather partition relations directly from predicated code. Other advantages of this direct approach are that it frees us from incrementally maintaining the partition graph as the code is further transformed, and it removes many phase-ordering constraints from the compiler.

Completing the partition graph

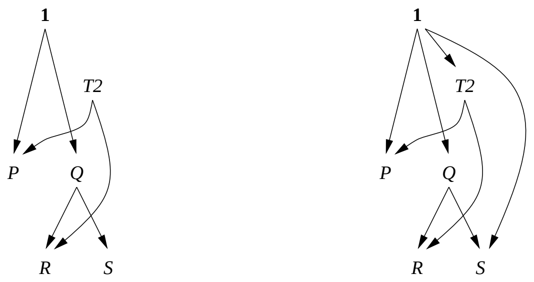
Each initial partition relation $r : P = Q_1|\ldots|Q_k$ translates into a set of labeled edges $U \xrightarrow{r} Q_i$ in the partition graph. The resulting graph may be incomplete, with not all nodes reachable from a unique root. In particular, nodes corresponding to unstructured control regions, whose partitions come from parallel-or compare operations, initially will not be reachable. To complete the partition graph, we synthesize additional partitions consistent with existing partitions so that all nodes are reachable from root.

Suppose $U = M \mid N$ is a partition and U is not reachable from $\mathbf{1}$. If M and N are reachable from $\mathbf{1}$, there exists a lowest common ancestor of M and N, denoted $lca(M, N)$, which is reachable from $\mathbf{1}$. Since $U = M \cup N$, $U \subseteq lca(M, N)$. Therefore, we can create the partition $lca(M, N) = U \mid Q_i$, where the union of Q_i's is the relative complement of U with respect to $lca(M, N)$. After inserting this new partition, U is reachable from $\mathbf{1}$. The algorithms for computing lowest common ancestors and relative complements is given in Section 5. Note that the least common ancestor and relative complement are computed in the partition subgraph consisting of nodes and edges reachable from the root.

```
U:  p = !(a<b) · true
    q =  (a<b) · true
   t1 = !(a<b) · true

    r =  (c!=d) · q
    s = !(c!=d) · q

   t2 = t1 + (c!=d) · q
```



(a) sequential SSA form (b) initial partition graph (c) completed graph

Figure 5: Example of partition graph for unstructured code

Example: Figure 5 is our running example. The incomplete partition graph in Figure 5(b) is formed by inserting appropriately labeled edges to represent each partition relation extracted from the predicated code[1]. Node $T2$ is not reachable from $\mathbf{1}$, but its successors P and R are reachable from $\mathbf{1}$. Searching back through the rooted subgraph, we find that $\mathbf{1}$ itself is the *lca* of P and R. The relative complement of $T2$ with respect to $\mathbf{1}$ (i.e. $\mathbf{1} - T2$) can be computed as $(\mathbf{1} - R) - P$. The relative complement of a node N with respect to an ancestor M is found by collecting the union of siblings of nodes along any path from M to N. For example, P and S are the siblings of nodes on the path from $\mathbf{1}$ to R, so $\mathbf{1} - R = P + S$. Repeating this process, we compute $(P + S) - P = (P - P) + (S - P) = S$. The partition $\mathbf{1} = T2 \mid S$ is added, completing the partition graph.

In general there may be multiple nodes requiring completion. These nodes can be processed in increasing order of their distance from the connected subgraph, so that when a node is processed it is guaranteed that its successors are reachable from the root.

5 The predicate query system

The predicate query system (PQS) allows for manipulating predicate expressions and for testing general predicate relations. The query system builds a partition graph for each predicate block using the method from Section 4, and subsequently uses the partition graph to answer queries and perform operations on predicate expressions. The PQS interface is motivated by the needs of predicate-sensitive data flow analysis, described in Section 6. The interface includes the following functions.

- **true_expr** (): a predicate expression representing the universal set of executions.
- **false_expr** (): the null predicate expression representing the empty set of executions.

  These functions are used to initialize predicate expressions in data flow vectors to the appropriate default value. For example, in liveness analysis every point is initialized to the false expressions, representing no variables live.

- **lub_sum** $(P, \mathcal{E})$: an expression representing a smallest superset of the execution set $P \cup \mathcal{E}$.
- **glb_sum** $(P, \mathcal{E})$: an expression representing a greatest subset of the execution set $P \cup \mathcal{E}$.

  These functions are used to generate a property at a point under a guarding predicate. The functions need not be precise and may approximate the exact result from above and below respectively. This interface permits trade-offs between precision and efficiency of the solution. The choice between least upper bound and greatest lower bound is determined by the data flow problem being solved. For

[1]For sake of readability, we will continue to substitute execution set symbols in place of the arbitrary numeric names output by the local partition extraction process. As before, edge labels are omitted from the figure.

example, in liveness analysis, if variable `x` is live in executions represented by expression $\mathcal{E}$ just after an operation in which `x` is used under guarding predicate `p`, then `x` will be live in executions represented by $\mathcal{E} + P$ just before the operation. Since liveness is an "any path" problem, over-approximation of the execution set is conservative and so we compute $\mathcal{E} + P$ using **lub_sum** $(P, \mathcal{E})$.

- **lub_diff** $(\mathcal{E}, P)$: an expression representing a smallest superset of the execution set $\mathcal{E} - P$.
- **glb_diff** $(\mathcal{E}, P)$: an expression representing a greatest subset of the execution set $\mathcal{E} - P$.

  These functions are used to kill a property at a point under a guarding predicate. For example, in liveness analysis, if variable `x` is live in executions represented by expression $\mathcal{E}$ just after an operation in which `x` is defined under guarding predicate `p`, then `x` will be live in executions represented by $\mathcal{E} - P$ just before the operation. Since liveness is an "any path" problem, over-approximation of the execution set is conservative and we compute $\mathcal{E} - P$ using **lub_diff** $(\mathcal{E}, P)$.

- **is_disjoint** $(P, \mathcal{E})$: true if the execution set $P \cap \mathcal{E}$ is the empty set.

  Used to test whether a property holds in *some* execution at a point under a guarding predicate. For example, two variables interfere (i.e. they must be assigned to different physical registers) if one is live at a definition point of the other. If variable `y` is live in executions represented by $\mathcal{E}$ at a point where variable `x` is defined under predicate `p`, then the variables interfere if $\mathcal{E}$ is not disjoint from P. This is tested using **is_disjoint** $(P, \mathcal{E})$.

- **is_subset** $(p, \mathcal{E})$: true if P is a subset of the execution set represented by expression $\mathcal{E}$.

  Used to test whether a property holds in *all* executions at a point under a guarding predicate. For example, a computation of expression `x/y` is redundant if prior computations are available (i.e. computed after the most recent definition of either operand) on all executions to that point. If expression `x/y` is available in executions represented by $\mathcal{E}$ at a point where expression `x/y` is computed under predicate `p`, then the computation is redundant if P is a subset of $\mathcal{E}$. This is tested using **is_subset** $(P, \mathcal{E})$.

5.1 Expressions in 1-disjunctive normal form

In this section, we describe efficient partition graph algorithms for manipulating predicate expressions and for answering queries about predicate relations.. The approach we've taken is to restrict predicate expressions to a simple normal form that is easy to manipulate, making conservative approximations as necessary to ensure correctness. In our experience, this solution provides a good trade-off between accuracy and efficiency.

The variables of predicate expressions are symbols denoting execution sets for individual predicates, and they are represented explicitly by nodes in the partition graph. Since a predicate expression will be used to represent a set of executions for which some property holds, we can conservatively approximate this set by either taking a subset or a superset, depending on the kind of analysis being performed. For example, in an "any-path" problem such as liveness, we test liveness at a point under a predicate `p` by asking whether any execution in P is represented in the predicate expression for liveness at that point; i.e. we query whether P intersects $\mathcal{E}$. To make a conservative approximation, we must overestimate liveness, and thus any predicate expression $\mathcal{E}'$ representing a larger set than $\mathcal{E}$ is correct. Ideally, we would chose an expression that is both easy to represent and that is a good approximation, so that our analysis remains relatively precise. For other analysis problems, the correct approximation may require finding a representation of a set smaller than $\mathcal{E}$. When an algorithm may require performing an approximation, we provide both least upper bound (lub) and greatest lower bound (glb) versions. To save space, we restrict our attention to least upper bound algorithms; the greatest lower bound algorithms are similar, and differ only in the direction of their approximations.

We have explored a range of solutions, from simple (more approximate) to complex (more accurate) algorithms for predicate queries. Here, we present an efficient and accurate solution which restricts predicate expressions to disjunctions of individual symbols, known as 1-disjunctive normal form (1-dnf). Such expressions can be stored either as a list of symbols or as a bitvector having one bit per symbol (i.e. one bit per node in the partition graph). Since the number of predicate registers used within a predicate block is small, the bitvector usually fits in a single machine word and many of the algorithms can make use of bitvector operations for greater efficiency. The accuracy given up is minor; using a 1-dnf representation, only **lub_diff** requires any approximation.

is_disjoint (*Symbol* P, *Symbol* Q)
{
  if $\exists$ partition $W = X \mid Y \mid Z_i$
    s.t. $P \subseteq X$ and $Q \subseteq Y$ **then**
    return true;
  else
    return false;
  endif
}

is_disjoint (*Symbol* P, *Expression* $\mathcal{E}$)
{
  `// assumes` $\mathcal{E}$ `is reduced`
  for each symbol Q in $\mathcal{E}$ **do**
    if is_disjoint $(P, Q) =$ false **then**
      return false;
  endfor
  return true;
}

(a) algorithms for **is_disjoint**

is_subset (*Symbol* P, *Symbol* Q)
{
  if Q is an ancestor of P **then**
    return true;
  else
    return false;
  endif
}

is_subset (*Symbol* P, *Expression* $\mathcal{E}$)
{
  `// assumes` $\mathcal{E}$ `is reduced`
  for each symbol Q in $\mathcal{E}$ **do**
    if is_subset (P, Q) **then**
      return true;
  endfor
  return false;
}

(b) algorithms for **is_subset**

Figure 6: Algorithms for **is_disjoint** and **is_subset**

5.2 Predicate query algorithms

We now describe algorithms for manipulating predicate expressions in 1-disjunctive normal form. We assume each predicate expression is a list of predicate symbols (possibly represented as a bitvector, where each bit position corresponds to a symbol). There are N symbols, where N is the number of nodes in the partition graph. The false expression, denoted $\mathcal{E}_{false}$, is simply an empty list. The true expression, denoted $\mathcal{E}_{true}$, is the list containing **1**, the root of the partition graph. Note that many predicate expressions are equivalent to $\mathcal{E}_{true}$; the algorithms in this section maintain predicate expressions in their reduced form.

Figure 6(a) shows the algorithms for testing disjointness between a pair of symbols, and between a symbol and a predicate expression (in reduced 1-dnf form). The algorithm attempts to show disjointness between two symbols P and Q by finding a partition such that P and Q are contained in disjoint parts. If no such partition exists, P and Q are assumed to intersect.

Figure 6(b) shows the algorithms for testing the subset relations. Set P is a subset of Q if there is a reverse path from node P to node Q in the partition graph. As we have mentioned, the subset relation is analogous to dominance and post-dominance, but is insensitive to temporal ordering of the predicate computations.

The algorithms for summing a single symbol into a predicate expression are shown in Figure 7(a). First, easy special cases are considered, such as the new symbol already being contained as a subset of one of the existing symbols in the expression. If the symbol needs to be added, it is added to the list and the resulting expression is recursively reduced. This step is performed by **sum_reduce**. Note that although we call this function **lub_sum**, no approximation is performed.

The algorithms for subtracting a symbol P from a predicate expression are shown in Figure 7(b). This is the only place where approximation is required. As in **lub_sum**, easy cases are handled first. If P is a subset of a symbol Q in the expression, we compute $Q - P$ by replacing Q with the relative complement of P with respect to Q, which can always be represented by a 1-dnf expression. If P intersects a symbol Q in the expression but is neither a subset or a superset, then approximation is required to represent the set difference. Any superset of the set difference is a valid approximation, and we use the relative complement of P with respect to the lowest-common ancestor of P and Q. We have found this approximation works well for liveness in real examples.

lub_sum (*Symbol* P, *Expression* $\mathcal{E}$)
{
$\mathcal{E}' = \mathcal{E}_{false}$;
for each symbol Q in $\mathcal{E}$ **do**
  if $Q \subseteq P$ **then**
    continue;
  else if $P \subseteq Q$ **then**
    return $\mathcal{E}$;
  else
    $\mathcal{E}' = \mathcal{E}' + Q$;
  endif
endfor
return sum_reduce $(P, \mathcal{E}')$;
}

sum_reduce (*Symbol* P, *Expression* $\mathcal{E}$)
{
add P to $\mathcal{E}$;
for each partition $R = P \mid Q_i$ **do**
  if all Q_i are members of $\mathcal{E}$ **then**
    $\mathcal{E}$ = **sum_reduce** $(R, \mathcal{E})$;
    break;
  endif
endfor
if R is a member of $\mathcal{E}$ **then**
  remove all proper descendents of R from $\mathcal{E}$;
endif
return $\mathcal{E}$;
}

(a) algorithms for **lub_sum**

lub_diff (*Expression* $\mathcal{E}$, *Symbol* P)
{
$\mathcal{E}' = \mathcal{E}_{false}$;
for each symbol Q in $\mathcal{E}$ **do**
  if $Q \subseteq P$ **then**
    continue;
  else if $P \subseteq Q$ **then**
    $\mathcal{E}' = \mathcal{E}'$ + **rel_cmpl** (P, Q);
  else if P disjoint from Q **then**
    $\mathcal{E}' = \mathcal{E}' + Q$;
  else
    $\mathcal{E}' = \mathcal{E}'$ + **approx_diff** (P, Q);
  endif
endfor
return $\mathcal{E}$;
}

approx_diff (*Symbol* P, *Symbol* Q)
{
`// over-approximate` $Q - P$
return rel_cmpl $(P,$ **find_lca** $(P, Q))$;
}

rel_cmpl (*Symbol* P, *Symbol* Q)
{
`// return expression for` $Q - P$
if is_subset (P, Q) = false **then**
  return false_expr ();
endif
find a path from Q to P;
let $\mathcal{E} = \mathcal{E}_{false}$;
for each edge $R \rightarrow S$ on path **do**
  let $R = S \mid T_i$ be the partition containing edge $R \rightarrow S$;
  add each T_i to $\mathcal{E}$;
endfor
return $\mathcal{E}$;
}

find_lca (*Symbol* P, *Symbol* Q)
{
let S_P be the set of ancestors of P;
let S_Q be the set of ancestors of Q;
return member of $S_P \cap S_Q$ having the largest level number
}

(b) algorithms for **lub_diff**

Figure 7: Algorithms for **lub_sum** and **lub_diff**

5.3 Treatment of the false predicate

Although rare, we must also be able to correctly analyze predicate operations whose guarding predicate is the constant false. Such code is dead; it is essentially the same as no-ops. The execution set for the false predicate is the empty set, because there are no execution traces in which an operation guarded by false is executed. Unlike other execution sets, the false set is not represented in the partition graph. This decision is motivated by the fact that the empty set has a special relation to all sets. Namely, the empty set is both a subset of every set and disjoint from every set.

We could represent the special relation of the empty set to other sets in the partition graph by creating a node for the false predicate, denoted as $\mathbf{0}$, and by adding partitions of the form $P = \mathbf{0} \mid P$ to the graph. This approach has two drawbacks. First, it makes the partition graph cyclic. Second, it roughly doubles the number of partitions; at minimum, we would add a new partition from the existing leaf nodes to the false node, making the false node the unique leaf node in the resulting graph. Because of these drawbacks, we choose to omit the false predicate from the partition graph and instead add special-case code to handle the false predicate in our partition graph algorithms. The extensions are straightforward and are omitted.

6 Predicate-sensitive data flow analysis

We now have the tools needed to perform predicate-sensitive data flow analysis in if-converted regions. The techniques described in this paper have been implemented in our ILP research compiler and are used for computing data dependences prior to scheduling, for computing up and down exposed definitions and uses, and for computing live range and interference information during register allocation.

Conventional bitvector analysis computes whether some property holds for each variable (or expression) at every point in a control flow graph. Each variable maps to a particular position within all bitvectors. In predicate-sensitive analysis, this framework is extended so that a property holds at a point *for a set of executions*, i.e. at a point within an execution set. Each vector position is extended to hold a predicate expression instead of a single bit. When an operation generates a property under a predicate `p`, then the property holds for all executions in P at that point, and so set P is unioned with the execution set at that point. If an operation kills a property under predicate `q`, then Q is subtracted from the execution set at that point. The notion of testing a property at a point is extended to testing whether the property holds at a point in some subset of executions. These expression manipulations are performed efficiently using PQS functions. In the following section, we illustrate our approach by extending traditional liveness analysis to be predicate-sensitive.

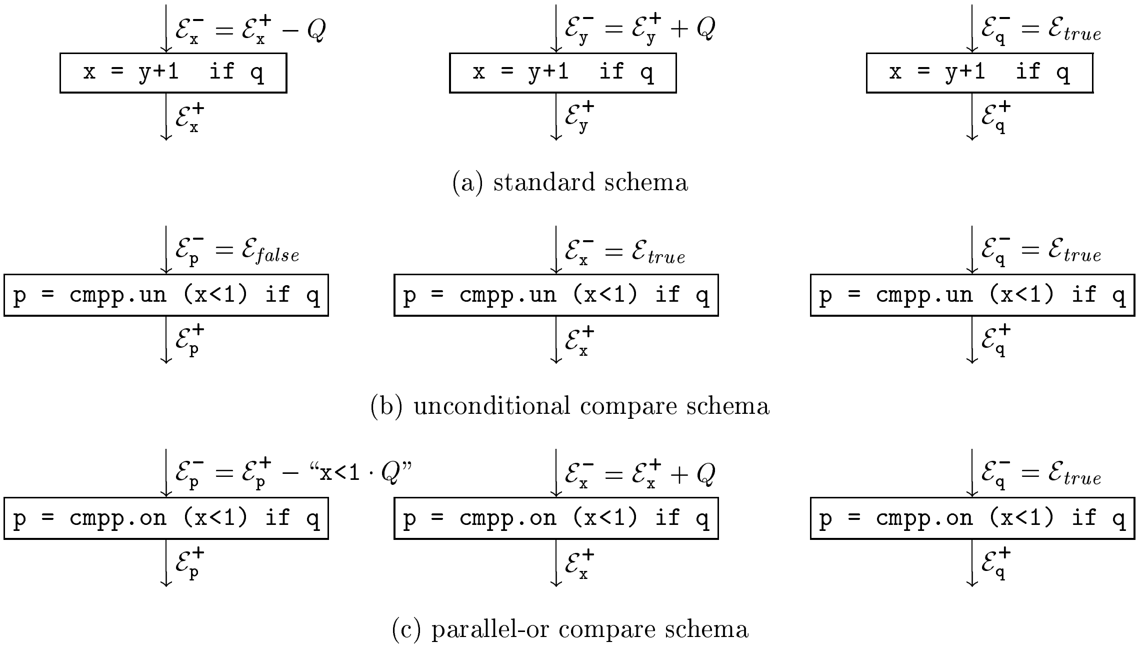
6.1 Liveness: an example of local predicate-sensitive analysis

In traditional liveness analysis[2], a variable `x` is live at a point if there is a path containing no definition of `x` from that point to a use of `x`. Liveness is solved using a backward propagation of data flow information throughout the control flow graph. During this backward traversal, liveness is *generated* at each variable use (i.e. the bit position representing liveness for the variable is set to true), and liveness is *killed* at each variable definition (i.e. the bit position is set to false).

In a predicated block, we extend this notion to track the set of executions for which each variable is live. We use a vector of predicate expressions having one position for each variable in the code. Each predicate expression represents the set of executions in which the corresponding variable is live at that point. The predicate expressions are updated as each operation is visited in reverse order.

At an operation, we use $\mathcal{E}^-$ to denote an expression just prior to the operation, and $\mathcal{E}^+$ to denote an expression just after the operation. Consider a use of variable `x` guarded by predicate `p`. The use generates liveness in all executions in P just prior to the operation, i.e. at the operation, $\mathcal{E}^-_{\mathtt{x}} = \mathcal{E}^+_{\mathtt{x}} + P$. A definition of `x` guarded by predicate `q` kills liveness of `x` in all executions in Q, i.e. at the operation, $\mathcal{E}^-_{\mathtt{x}} = \mathcal{E}^+_{\mathtt{x}} - Q$. At a definition of `y` guarded by predicate `r`, `y` interferes with `x` if `x` is live under predicate `r`, i.e. if $\mathcal{E}^+_{\mathtt{x}} \cap R \neq \emptyset$. With respect to interference, it is conservative to overestimate the set of executions in which a variable is live. Therefore, we use **lub_sum** and **lub_diff** to manipulate predicate expressions. Interference is tested using **is_disjoint**.

[2] An introduction to standard data flow analysis can be found in the "Dragon Book" [ASU86].



(a) standard schema

(b) unconditional compare schema

(c) parallel-or compare schema

Figure 8: Gen/Kill schemata for liveness analysis of predicated code

Figure 8 shows the gen and kill schemata in detail. Operations are processed in reverse order. Within each operation, definitions are processed before uses. Figure 8(a) shows the gen and kill processing for ordinary predicated definitions and uses. Note that the use of the guarding predicate is different from other variable uses in an operation: it is used unconditionally and therefore generates liveness in all executions.

Compare operations require special attention. The unconditional compare operation always reads its source operands and always writes a value to its destination operand(s). Therefore, liveness is killed for destination operands in all executions, yielding the live set $\mathcal{E}_{false}$; liveness is generated for source operands in all executions, yielding the live set $\mathcal{E}_{true}$. This is the same result as for standard predicated operations when the guarding predicate is the constant true.

The parallel-or compare operation writes the value true whenever both the compare condition and guarding predicate are true. This conjunctive condition does not necessarily correspond to any predicate variable in the source code, however it is represented by a predicate symbol in the partition graph. This is due to the way partition relations are generated during extraction. Recall that the extraction algorithm performs a lookup of the string "`(r1<cond>r2)` $\cdot$ `q`" representing the right-hand subexpression from the sequential SSA form of the parallel-or compare operation. The resulting predicate symbol is the appropriate execution set for which liveness of the destination variable is killed. The gen conditions for source operands are the same as with standard predicated operations.

Now consider predicate-sensitive liveness analysis for variables `m` and `n` in Figure 9. Initially, both variables are dead on all executions at the bottom of the predicate block, so at `end`, $\mathcal{E}^-_{\mathtt{m}} = \mathcal{E}^-_{\mathtt{n}} = \mathcal{E}_{false}$. At the use of `n` under predicate `t` in operation k, `n` becomes live in all executions in set T:

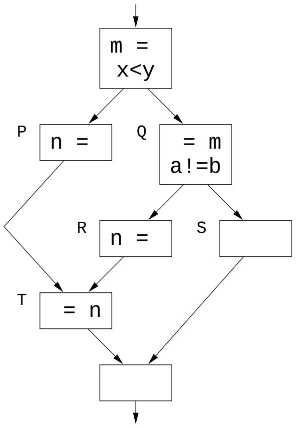
$$\mathcal{E}^-_{\mathtt{n}} = \textbf{lub\_sum}\ (T, \mathcal{E}^+_{\mathtt{n}}) = \textbf{lub\_sum}\ (T, \mathcal{E}_{false}) = T.$$

At the definition of `n` under predicate `r` in operation j, `n` is killed in all of R:

$$\mathcal{E}^-_{\mathtt{n}} = \textbf{lub\_diff}\ (\mathcal{E}^+_{\mathtt{n}}, R) = \textbf{lub\_diff}\ (T, R) = P.$$

Finally, `n` is defined under `p` in operation i:

$$\mathcal{E}^-_{\mathtt{n}} = \textbf{lub\_diff}\ (\mathcal{E}^+_{\mathtt{n}}, P) = \textbf{lub\_diff}\ (P, P) = \mathcal{E}_{false}.$$



(a) original code

```
       t = false              if true
       m =                    if true
     p,q = cmpp.un.uc (x<y)   if true
       t = cmpp.on (x<y)      if true
i:     n =                    if p
         = m                  if q
     r,s = cmpp.un.uc (a!=b)  if q
j:     n =                    if r
       t = cmpp.on (a!=b)     if q
k:       = n                  if t
```

(b) liveness in if-converted code

Figure 9: Liveness analysis of predicated code

We compute liveness for `m` in a similar way.

Figure 9(b) shows the live ranges for `n` and `m` annotated with predicate expressions. Although the live ranges overlap temporally, the two variables do not interfere. This is seen by examining predicate expressions at definition points. The only definition within both live ranges is the definition of `n` under predicate `p` in operation i. Since $P \cap \mathcal{E}^{+}_{\mathtt{m}} = P \cap Q = \emptyset$, there is no execution in which both variables are live simultaneously. Therefore, `m` and `n` can be assigned the same physical register.

6.2 Global analysis

Predicate-sensitive analysis is performed within predicate blocks and can be used in conjunction with a conventional bitvector based solver. First, local predicate-sensitive analysis is performed on each block to compute transfer functions. These transfer functions summarize the effect of the block without exposing predicate expressions outside the block.

For liveness, a transfer function for a block consists of two bitvectors, GEN and KILL, which represent the overall effect of the block on liveness for each variable. We compute these bitvectors as follows. First, we assume all variables are dead at the block exit, i.e.

$$\mathcal{E}_{\mathtt{x}} = \mathcal{E}_{false}, \text{ for all } \mathtt{x}.$$

Starting with this initial vector of false expressions, we process the operations in reverse order, applying the operator-level liveness rules from Figure 8. At the entry, any variable whose predicate expression is not false is said to be generated by the block, since there is some definition free execution from the block entry to a use of the variable. The bit positions in GEN corresponding to such variables are set to true. Similarly, we compute KILL by assuming that all variables are live in all executions at the block exit, i.e.

$$\mathcal{E}_{\mathtt{x}} = \mathcal{E}_{true}, \text{ for all } \mathtt{x}.$$

Again, we perform predicate-sensitive analysis operation-by-operation in reverse order through the block. Any variable whose predicate expression is false is said to be killed, since on every execution a definition is encountered before reaching the first use or the block exit.

A conventional bitvector solver can now be used to compute the solution at block boundaries. Block transfer functions have the following form:

$$live_{in} = (live_{out} - KILL) + GEN.$$

Since predicate-sensitive analysis is localized within block, we retain the efficiency of the global solver. The solution at block boundaries is projected to points within the block by performing a final predicate-sensitive pass over the block, using the boundary values as the initial predicate expression values. More specifically, at each block exit, we assign $\mathcal{E}_{\mathtt{x}} = \mathcal{E}_{true}$ if x is live out, else $\mathcal{E}_{false}$.

An alternative approach is to extend predicate-sensitive analysis beyond individual blocks. A more global predicate-sensitive analysis is beneficial when predicate lifetimes extend beyond predicate blocks. Additionally, global predicate-sensitive analysis can exploit correlations between branch conditions in control flow and therefore enhance the accuracy of analysis beyond traditional methods. Gillies et al [GJJS96] present an approach to global predicate-sensitive analysis and demonstrate its utility in register allocation.

7 Conclusion

New architectural features for high-performance computing continue to present new challenges and opportunities to compiler writers. Predicated execution offers substantial benefits to exploiting instruction-level parallelism, from enlarging scheduling scope and eliminating mispredicted branches to providing a means for parallelizing while loops and supporting software pipelining of loops containing complex control flow. To take full advantage of these opportunities, we must be able to work with predicated code as easily as existing compilers work with non-predicated code.

Some early work on understanding predicate relations has been done by Wen-mei Hwu's Impact compiler group at Illinois. They describe a data structure called the predicate hierarchy graph (PHG) [Lin90], which represents the boolean equation under which each predicate is defined. The main use of PHG mentioned in Lin's thesis is to determine if a pair of predicate registers is disjoint. Two predicates are disjoint only if the conjunction of their associated boolean expressions can be simplified to false. Rather than perform predicate-sensitive live range analysis, the approach adopted in the Impact compiler is to apply reverse if-conversion [WMHR93] to predicated code, and then perform standard data flow analysis on the resulting control flow graph. This analysis can yield conservative results because the flow graph constructed by reverse if-conversion will in general contain paths that cannot be traversed during any execution.

Eichenberger and Davidson [ED95] perform local, predicate-sensitive live range analysis by reducing each pair-wise interference query to a satisfiability test of a conjunctive expression representing all the conditions under which the particular interference may occur. Each such query is solved using a powerful symbolic package. Their paper demonstrates the potential benefit of considering predicate relations during register allocation of predicated code, but we believe their approach is impractical for use in a production compiler.

In this paper, we have presented several techniques for analysis of predicated code. First, we addressed the problem of understanding relations between predicates. We model relations between predicates in terms of relations between execution sets, and we developed an algebra of predicate expressions to approximate execution sets. We introduced the predicate partition graph for representing and manipulating predicate expressions, and we showed how this graph can be built directly from predicated code.

We described the predicate query system, a system for manipulating predicate expressions to answer queries about predicate relations. The predicate query system makes use of partition graphs to perform these tasks efficiently. We have explored several classes of algorithms that trade-off accuracy for efficiency, and we presented one class of efficient algorithms that sacrifice little accuracy.

Finally, we showed how predicate expressions and the predicate query system are used to perform predicate-sensitive data flow analysis within predicate blocks. We use this analysis in our research compiler to perform data dependence and live range analysis to support hyperblock scheduling, software pipelining, and register allocation. By localizing predicate-sensitive analysis within predicate blocks, we retain the efficiency of bitvector solvers globally while performing high-quality analysis locally.

This work provides a starting point for further study of predicated code. In future work, we hope to show how performance of generated code varies as trade-offs are made between accuracy and efficiency of analysis. Predicated execution also opens up new opportunities in optimization; we intend to explore this area in detail.

Acknowledgments

The authors would like to thank Roy Ju and other members of the HP California Language Laboratory for their many interesting and useful discussions of predicates.

References

[AKPW83] J. R. Allen, K. Kennedy, C. Portfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, Austin, Texas, January 24–26, 1983.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105, Boston, Massachusetts, June 23–25, 1982. Published as ACM SIGPLAN Notices 17(6).

[DHB89] James C. Dehnert, Peter Y.-T Hsu, and Joseph P. Bratt. Overlapped loop support in the Cydra 5. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, Boston, Massachusetts, April 3–6, 1989.

[DT93] James C. Dehnert and Ross A. Towle. Compiling for the Cydra 5. *IEEE Computer*, 7(1/2):181–227, 1993.

[ED95] Alexandre E. Eichenberger and Edward S. Davidson. Register allocation for predicated code. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 180–191, Ann Arbor, Michigan, November 29–December 1, 1995.

[GJJS96] David M. Gillies, Dz-ching Roy Ju, Richard Johnson, and Michael Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 114–125, Paris, France, December 2–4, 1996.

[HD86] Peter Y. T. Hsu and Edward S. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 386–395, Tokyo, Japan, June 2–5, 1986.

[Hew92] Hewlett-Packard Company. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Prentice-Hall, second edition, 1992.

[KSR93] Vinod Kathail, Mike Schlansker, and Bob Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, February 1993.

[Lin90] David Chu Lin. Compiler support for predicated execution in superscalar processors. Master's thesis, University of Illinois at Urbana-Champaign, 1990.

[MHB+94] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen-mei Hwu. Characterizing the impact of predicated execution on branch predication. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 217–227, San Jose, California, November 30–December 2, 1994.

[MLC+92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, Oregon, December 1–4, 1992.

[PS91] Joseph C. H. Park and Mike Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett-Packard Software and Systems Laboratory, May 1991.

[RYYT89] B. Rau, D. Yen, W. Yen, and R. Towle. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computer*, 22(1):12–35, January 1989.

[SK95] Michael Schlansker and Vinod Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 57–69, Ann Arbor, Michigan, November 29–December 1, 1995.

[Tys94] Gary Scott Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 196–206, San Jose, California, November 30–December 2, 1994.

[WMHR93] Nancy J. Warter, Scott A. Mahlke, Wen-mei W. Hwu, and B. Ramakrishna Rau. Reverse if-conversion. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 290–299, Albuquerque, New Mexico, June 23–25, 1993.