Global Predicate Analysis and its Application to Register Allocation

David M. Gillies, Dz-ching Roy Ju
Hewlett-Packard California Language Lab
11000 Wolfe Road, Cupertino, CA 95014
{ dgillies, royju }@cup.hp.com

Richard Johnson, Michael Schlansker
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304
{ rjohnson, schlansk }@hpl.hp.com

Abstract

To fully utilize the wide machine resources in modern high-performance microprocessors it is necessary to exploit parallelism beyond individual basic blocks. Architectural support for predicated execution increases the degree of instruction level parallelism by allowing instructions from different basic blocks to be converted to straight-line code guarded by boolean predicates. However, predicated execution also presents significant challenges to an optimizing compiler. For example, in live range analysis, a predicated definition does not necessarily end the live range of a virtual register.

This paper describes techniques to analyze the relations among predicates in order to improve the precision and effectiveness of various compiler analysis and transformation phases in the presence of predicated code. Our predicate analysis operates globally to obtain relations among predicates. Moreover, we analyze control flow and predication in a single unified framework. The result can be queried by subsequent optimization and analysis phases. Based on this framework, we extend a traditional method to a predicate-aware register allocator which takes global predicate relations into account. We have implemented the proposed algorithms to effectively reduce register pressure. Our experimental results show 24.6% of a large test suite obtain, on average, 20.71% better register allocation due to the algorithms presented in this paper.

1 Introduction

VLIW (Very Long Instruction Word) and superscalar architectures can exploit significant amounts of instruction level parallelism (ILP) to achieve improved performance in application programs. Parallelism within individual basic blocks are generally insufficient to fully utilize wide machine resources. Predicated execution [9, 11] is an architectural model to exploit parallelism across basic blocks. In this model instructions from different basic blocks are converted to straight-line code guarded by boolean predicates. Architectural support for predicated execution varies from general support for predication in the Cydra 5 [16] and HPL PlayDoh architectures [11] to conditional skip instructions in the HP PA-RISC architecture [8], conditional nullify instructions in the Sun SPARC architecture [19], and conditional move instructions in the DEC Alpha architecture [3]. In a general predicated execution model, the execution of an instruction is guarded by a boolean qualifying predicate. Each qualifying predicate can be regarded as a 1-bit predicate register. An example of a predicated instruction is

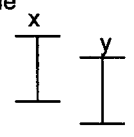
```
x = y + z          if p
```

where p is the qualifying predicate which controls whether the instruction executes and updates the architectural state. To explore predication, a compiler generally incorporates a technique called *if-conversion*, which eliminates branch instructions and converts affected instructions to appropriate predicated forms. If-conversion effectively converts control flow into data flow [1].

However, in order to achieve these benefits, predication presents a challenge to many conventional analysis and transformation phases performed in an optimizing compiler. For example, consider the code in Example 1.

Example 1:

```
S1:  p,q = cmpp.un.uc  (a < b)     if true
S2:  x = ..                        if p
S3:  y = ..                        if q
S4:  .. = x                        if p
S5:  .. = y                        if q
```



The cmpp instruction sets p true and q false if the condition (a < b) is true and reverses these values if the condition is false. (The details of the architectural model are described in Section 2.) Variable x is defined and used under p and variable y is defined and used under q. A conventional live range analysis would find the live range of x from S2 to S4 and the live range of y from S3 to S5. Based on this analysis, a traditional register allocator would conclude that x and y interfere with each other and subsequently assign different physical registers to them. However, since p and q are complementary, x and y will never hold valid values at the same time and thus they can share the same physical register. The focus of this work is to develop a practical and efficient mechanism to track global predicate relations in an entire procedure. This mechanism can be used to improve data flow analysis and optimizations in the presence of predicated code. As an important application, we demonstrate how a register allocator can benefit from understanding and utilizing the predicate relations.

1072-4451/96 $5.00 © 1996 IEEE

The IMPACT compiler uses a predicate hierarchy graph (PHG) to track the boolean equations for all of the predicates in a hyperblock [14]. The scope of their predicate analysis is a hyperblock and does not extend globally. When a companion data structure, like PHG, is maintained with the code, then it may require updating whenever the program is transformed. The work in [18] proposes a reverse if-conversion scheme to map predicates from the data flow domain back to the more familiar control flow domain. A major drawback of this approach is that, during the remapping process, some originally non-existent control paths may be created. This typically occurs when code is if-converted and then permuted by instruction scheduling. When this code is reverse if-converted, non-existent paths may cause conservative treatment in subsequent analyses and transformations. The work in [7] solves this register allocation problem by representing predicates as P-facts -- logically invariant expressions. The mechanism concludes that two live ranges do not interfere if the intersection of the two sets of P-facts can be simplified to false using a symbolic package. This approach is also restricted to the scope of a hyperblock, and may have limited practicality given the potential exponential compilation time behavior with respect to the number of predicates. The authors in [10] propose a representation called *predicate partition graph* to track the relations in predicates, and they describe how to construct a partition graph for the predicates in a hyperblock. Their work uses the partition graph to provide information to support data flow analysis.

The false interference in Example 1 can be avoided by a predicate analysis which analyzes the local predicate relations in the straight-line code between two branches. The code fragment in Example 2 shows the importance of recognizing predicate relations in a global scope, where code may be partially if-converted based on the profitability model in an if-converter. Both of x and y are defined and used in different basic blocks. To check for interferences, one would propagate the use of x under p to the else-clause towards the definition of x under r. While crossing the definition of y under s during this process, an interference will be assumed between x and y unless one can assert that p and s are disjoint. Since p and s are defined in the then- and else- clauses, respectively, p and s can never both be true at the same time. However, it requires global analysis to systematically take control flow into account to assert the disjointness between p and s. This relation cannot be captured by a hyperblock or basic block based analysis. Further, with global predicate analysis, one can also assert that p and r are disjoint. Therefore, the definition of x under r can never reach the use of x under p, and this definition is dead.

Our work is based on the predicate partition graph in [10]. The main contributions of our work are as follows: (1) We uniformly track relationships describing both control flow and explicit uses of predicates at a global scope by mapping them to a single partition graph. To the best of our knowledge, this is the first global predicate analysis framework. (2) We have developed and implemented practical algorithms for global predicate analysis and global predicate-aware register allocation using a generalized data flow framework. (3) We have demonstrated the effectiveness and practicality of this approach through experiments which show significant benefits of predicate-aware register allocation.

Example 2:

```
p, q, r, s = false
if ( .. ) then {
   p,q = cmpp.un.uc   (...)           if true
   x = ..                             if p
} else {
   r,s = cmpp.un.uc   (...)           if true
   x = ..                             if r
   y = ..                             if s
}
.. = x                                if p
.. = y                                if s
```

The rest of the paper has the following structure: Section 2 illustrates the architectural model for predication. Section 3 provides a brief overview of the predicate partition graph which is used to track the relations of predicates. Section 4 discusses the details on how to construct a single connected partition graph including both control flow and explicit predicates in an entire procedure. Section 5 presents the details of a predicate-aware register allocator, which exploits the relations of predicates during live range analysis and interference graph construction. Section 6 shows the experimental results in comparing the predicate-aware register allocator against a conventional register allocator. Section 7 provides conclusions.

2 Architectural Model

In this paper, we will assume the architectural support of general predicated execution model provided in the HPL PlayDoh architecture [11], in which the execution of an instruction can be guarded by a qualifying predicate. The following form of compare instructions is provided to set predicates.

```
p1, p2 = cmpp.<d1><d2>  (a rel b)   if qp
```

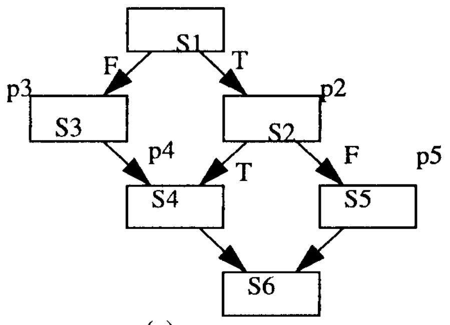
Predicates p1 and p2 are two destination predicates. Each of <d1> and <d2> is a two-letter descriptor that specifies a type and mode for the compare instruction. There are four comparison types: *unconditional* (u), *conditional* (c), *parallel-or* (o), and *parallel-and* (a), specified by the first letter of the descriptor. Each type has both a normal mode (n) and a complement mode (c), specified by the second letter of the descriptor. Descriptors <d1> and <d2> control destination predicates p1 and p2, respectively. The condition, (a rel b), is in the form of a logical comparison of two variables a and b, where *rel* can be eq, ne, lt, etc. Predicate qp is the qualifying predicate.

To simplify the discussion, for the unconditional and conditional types, we assume they are always generated in form of cmpp.un.uc and cmpp.cn.cc, respectively. Therefore, for a comparison with the unconditional type, the values of p1 and p2 are complementary. If the comparison type is *and* (*or*) and qp is true, the *an* (*on*) target predicate (say p1) is equal to the result of (a rel b) "*and*"ed ("*or*"ed) with the old value of p1.

Table 1. Behaviors of predicates in compare instructions

qp	(a rel b)	un	uc	cn	cc	on	oc	an	ac
F	X	F	F	nc	nc	nc	nc	nc	nc
T	T	T	F	T	F	T	nc	nc	F
T	F	F	T	F	T	nc	T	F	nc

Table 1 is the summary on how the destination predicates are set in these compare instructions. In the table, T and F stand for true and false values, respectively. X means "don't care" and nc means that the result is unchanged. We also assume that there is a special predicate register p0, where any read is always true and any write is discarded.



(a)

```
I1: p2, p3 = cmpp.un.uc  (s1 cond)  if true
I2: p4 = cmpp.uc  (s1 cond)         if true
I3: S2                              if p2
I4: p5 = cmpp.uc (s2 cond)          if p2
I5: p4 = cmpp.on  (s2 cond)         if p2
I6: S3                              if p3
I7: S5                              if p5
I8: S4                              if p4
I9: S6                              if true
```
(b)

Figure 1. (a) Control flow graph and (b) if-converted code.

Figure 1 shows an example, which contains the original control flow graph and the if-converted code. The branch condition of basic block containing S*n* is represented as (S*n* cond). If-conversion converts all six basic blocks into a single basic block with predicated code. Each of the switch points at the respective basic blocks containing S1 and S2 is converted to a compare instruction with an unconditional type. The merge (or confluence) point at the basic block containing S4 is converted to the combination of one compare instruction with an unconditional type and one compare instruction with an *or* type.

3 Overview of Predicate Partition Graph

In this work, we use the notion of predicate partition graph proposed in [10] to track the relations among predicates. To simplify the discussion, we assume that predicates are in static single assignment (SSA) form, where each predicate is statically defined no more than once. However, this is not required by our implementation. For the example in Figure 1, p4 is defined twice. To describe references to p4 in SSA form, the related definitions and uses are translated into the following intermediate form.

```
...
p4_1 = cmpp.uc (...)          if true
...
px = cmpp.un(s2 cond)         if p2
p4_2 = p4_1 | px              if true
...
S4                            if p4_2
...
```

We use the concept of an execution set to describe the relations among predicates. We first define the following notations for straight-line code and extend them to incorporate control flow in the next section. An execution *trace* includes all of the instructions being executed from the beginning to the end in straight-line code. A trace belongs to the *domain* of predicate p, if all of the instructions on this trace are executed when p is true. The domain of p includes all such traces. If unambiguous, we will simply use p to mean the domain of p. A *partition* of a predicate is a division of the domain of the predicate into multiple disjoint subsets, where the union of these subsets is equal to the domain. For example, the pictorial view of the relations among the predicates in instruction `p1,p2=cmpp.un.uc (...) if p3` is in Figure 2. When p3 is true, the values of p1 and p2 are always complementary. In another words, no trace exists where p1 and p2 are both true. One can also derive that if p1 or p2 is true, p3 must be true.

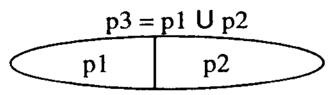


Figure 2. Set relations between predicate domains.

In a *predicate partition graph*, G=(V, E), each node p in V represents predicate p and each edge (p, q) represents that there exists a partition in p such that q is a subset in this partition. An edge in G is directed and the edges created from the same partition are given the same label. For example, partition p0 = p1 U p2 would be represented by two edges: p0->p1 and p0->p2, where both edges are assigned the same label. G is a directed acyclic graph, and there may be multiple edges between two given nodes due to different partitions. A partition graph is *complete* if the universal set, p0, is the unique root. This makes every node reachable from the root as is required by the algorithms used in predicate analysis. Because the partition graph approximates predicate relations, one may construct different partition graphs with varying accuracy for a single code sequence.

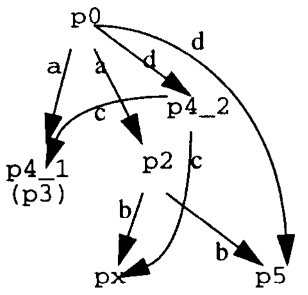


Figure 3. A partition graph for the example in Figure 1.

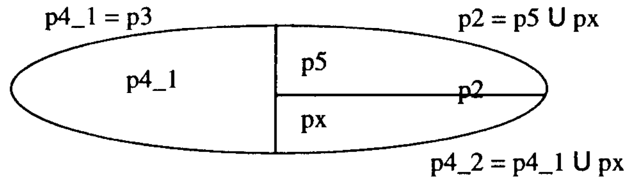


Figure 4. Domains of predicates in Figure 1.

A partition graph for the if-converted code in Figure 1(b) is shown in Figure 3. The pictorial view of the partitions shown in Figure 4 helps understand how domains are partitioned. The root of the partition graph, p0, is partitioned into p2 and p3 in instruction I1. Since instructions I1 and I2 both have an unconditional compare type and check for the same condition, these two instructions are essentially value congruent. Therefore, p3 and p4_1 always have the same value and can be mapped to the same predicate node. In I4, p2 is partitioned into p5 and an implicit predicate px, which is created to complete this partition. In I5, the union of px and p4_1 makes up the domain for p4_2. We can also find that the complement of p4_2 is p5 and generate another partition with p0 as the parent predicate and p4_2 and p5 as the child predicates. This step is necessary to make p4_2 reachable from the root.

Once a partition graph is constructed, one can use simple graph traversal algorithms to support a number of different queries on predicate relations. Readers are referred to [10] for further details on the properties of the partition graph and the implementation of various queries. We list below only the queries used in our predicate-aware register allocator.

`IsDisjoint(p,q)`: asks whether the domain of predicate p overlaps with that of predicate q. Two predicates are disjoint if they can reach a common ancestor through different edges of the same partition. For example, in Figure 3, p3 and p5 are disjoint, but p2 and p4_2 are not. Note this is obvious from the domain relations in Figure 4, where px is a subset of both p2 and p4_2. It is straightforward to extend this query to `IsDisjoint`(p, Q), where Q is a set of predicates, and the answer is true if p is disjoint from every predicate in Q.

`LeastUpperBoundSum(p,Q)` : adds predicate p to a set of predicates, Q. The domain of the resultant set is the smallest superset of the union of the domain of p and the domain of Q. The resultant set is expected to be simplified in the way that if all of the child predicates in a partition appear in the union of p and Q, these child predicates are replaced with their parent predicate.

`LeastUpperBoundDiff(p,Q)` : subtracts predicate p from a set of predicates, Q. The domain of the resultant set is the smallest superset of the domain of Q subtracted by the domain of p. For the example in Figure 3, px is equal to `LeastUpperBoundDiff`(p5, p2).

4 Global Predicate Partition Graph and its Construction

Previous work in predicate analysis are generally based on the scope of a *hyperblock*, which is an if-converted single-entry multiple-exit control flow region. In our work, each node in a control flow graph (CFG) is still a basic block, which is a single-entry single-exit straight line sequence of possibly predicated code. One advantage of this representation is that this is the most familiar representation to compiler analyses and optimizations. We assume that a CFG has a single start node and a single stop node. Our global predicate analysis operates on an entire procedure. Global predicate analysis provides more precise predicate relations when predicates are defined and used in different basic blocks.

We now extend previously defined notations to an entire procedure. A *trace* includes all of the executed instructions on an acyclic path from the start node to the end node in a CFG. A trace belongs to the *domain* of predicate p, if all of the instructions on this trace are executed when p is true. Analogous to predicates, we define the domain of a basic block to be all of the traces such that the basic block is executed. Therefore, when a basic block can be reached from another basic block (or vice versa), the domains of these two basic blocks are not disjoint. This control flow reachability information is expressed in a global partition graph to track relations among basic blocks regardless of the existence of predicates.

In order to uniformly treat control flow and predication, we assign a predicate to each basic block. We call a predicate assigned to a basic block a *control predicate* and a predicate which explicitly appears in the instruction stream a *materialized predicate*. A control predicate of a basic block is viewed as a predicate combining all of the conditions which control whether the basic block will be executed or not. We perform global predicate analysis based on building a single *complete* predicate partition graph which includes both control predicates and materialized predicates. This single partition graph allows us to uniformly track the semantics of conditional execution through program branches as well as predicated execution. Note that our if-converter assures that if a control predicate is also materialized to a materialized predicate, they share the same domain and are mapped to the same node in the partition graph. This further improves the precision of the predicate analysis.

The graph construction mechanism described below can be invoked anywhere throughout an optimizer. This can be done either before or after if-conversion. This flexibility is again due to that fact that we treat control flow and materialized predicates in a uniform way. Although if-conversion is the major source of creating materialized predicates, phases like expanding high level pseudo code may also emit predicated code. Data flow analyses both before and after if-conversion can be more precise for predicated code by knowing the global relations among materialized predicates.

The above discussion associates the control flow and data flow aspects of predicates. The following two subsections illustrate the construction of a partition graph for control predicates and materialized predicates, respectively.

4.1 Handling Control Predicates

In order to analyze control flow, control predicates are assigned to basic blocks and partitions are formed at control flow switch and merge points. Although special attention is given to the following control flow structures, our predicate analysis handles any arbitrary control flow graph including irreducible graphs.

Critical Edges - A critical edge is defined as an edge whose source has more than one successor and whose destination has more than one predecessor. For example, the edge from S2 to S4 in Figure 1(a) is a critical edge. At a switch (merge) point, a critical edge prevents the use of the destination (source) node's control predicate as a child predicate in the partition. This occurs because, on this critical edge, the source (destination) node does not dominate (post-dominate) the destination (source) node. To resolve this problem, we conceptually create a node on a critical edge and assign an implicit predicate to the node. Note that this edge splitting is only done at a conceptual level without actually changing the CFG. With this virtual edge splitting, all predecessors dominate their successors at switch points and all successors post-dominate their predecessors at merge points. This simplifies the creation of partitions. Note that if node p dominates or post-dominates node q, p can always be an ancestor of q in the partition graph.

Back Edges - Back edges are those edges which complete cycles in a CFG. With current techniques, if we were to take back edges into account in predicate analysis, there is little useful information that we can derive for predicates. For example, for an if-then-else construct enclosed in a loop, the then- and else- clauses will never both be executed in the same iteration, but they may be in different iterations. Therefore, the predicates assigned to the two clauses are disjoint within a particular iteration, but not necessarily across iterations. The results of this predicate analysis are interpreted with back edges ignored. This has little impact on the accuracy of the analysis. This is partially due to the fact that the if-converter is already restricted to acyclic regions. Section 5 will discuss the approximation of data flow information through back edges.

During partition graph construction, back edges are also split by assigning virtual nodes to them. Our analysis is also applicable to irreducible graphs. However, because the back edges are selected rather arbitrarily for an irreducible graph, the results may be less precise.

Figure 5 presents an algorithm to construct a partition graph based on control predicates. The input is a CFG. Edge splitting is first performed on critical edges and back edges. Finding control equivalent nodes is not essential for correctness, but instead improves the accuracy of predicate analysis as predicates with equivalent domains are mapped to the same node. We then create partitions at all program switch and merge points to track the predicate relations. Finally, if any non-start node does not have a parent node, a partition is created to link it to its immediate dominator to make the partition graph *complete*.

We now demonstrate how to construct the partition graph in Figure 3 based on the CFG in Figure 1(a) in analogy to an earlier discussion in Section 3 on constructing the same graph based on the if-converted code. Node S1 is the start node in the CFG and is given p0 as its control predicate. Node S6 is control equivalent to S1 and is assigned with p0 as well. The control predicates assigned to the rest of nodes are as shown in Figure 1(a). The a partition (as marked in Figure 3) is created due to the switch point at S1. The edge from S2 to S4 is a critical edge, and px is assigned to this edge to achieve edge splitting. The b partition is created due to the switch point at S2. The c and d partitions are created due to the merge points at S4 and S6, respectively.

```
ConstructPartitionGraphForControlPredicates(CFG) {
  Create a virtual node on each critical edge or back edge
  Find control equivalent nodes
  Assign a predicate to each set of control equivalent nodes

  for every node in the CFG {
    v = current node
    p = control predicate assigned to v
    if( number of successors > 1)
      Create a partition with p as the parent predicate and
      predicates assigned to the successors as the child
      predicates
    if( number of predecessors > 1)
      Create a partition with p as the parent predicate and
      predicates assigned to the predecessors as the child
      predicates, if this partition has not been generated yet
  }
  if a non-start node, u, has no parent, create a partition with
  the immediate dominator of u as the parent predicate, and
  u and an implicit predicate as the child predicates
}
```

Figure 5. Algorithm of constructing partition graph for control predicates.

4.2 Handling Materialized Predicates

This subsection discusses the details of constructing a partition graph which captures local relations among materialized predicates. The relations among materialized predicates are created at each point where predicates are defined, i.e. compare instructions. Figure 6 presents an algorithm to construct a partition graph for materialized predicates. The most common case of defining materialized predicates is a compare instruction with an unconditional type. In Figure 6, a partition is created with pp as the parent predicate and p1 and p2 as the disjoint child predicates. If qp is p0, we can view that the current instruction is guarded by bp which is the control predicate of the basic block, and the union of the domains of p1 and p2 is the domain of bp. Therefore, we build a partition with control predicate bp as the parent predicate. This is the key to establish the relations between control predicates and materialized predicates and to allow predicate analysis at a global scope. If qp is not p0, all of the control conditions affecting whether this instruction will be executed have been synthesized into the definition of qp. This is ensured in our implementation. Therefore, if qp is not p0, the union of p1 and p2 is qp and there is no need to build a partition to link p1 and p2 to bp. One can conclude that the domain of the actual predicate (pp) guarding the compare instruction is always a subset of the domain of bp.

The other three types of compare instructions involve updating a predicate, where the previous value of a predicate affects the new predicate result. We treat these compare types conservatively because they appear infrequently and are not easily modeled by the partition graph. For a compare instruction with a conditional type, if qp is p0, this instruction is effectively an unconditional type and is treated in the same way as the unconditional type. If qp is not p0, we simply map p1 and p2 to a dummy predicate node and during queries to the predicate analysis they are treated in a conservative manner. Note that if both of *on* and *oc* (or *an* and *ac*) target predicates appear in a compare instruction, we treat them being split into two single-target cmpp's, which are then processed separately. For an *and* type compare instruction, the domain of p1 is always a subset of that of p1_old. For an *or* type compare instruction, the domain of p1 is always a superset of that of p1_old and a partition is created to link p1 to control predicate cp to make the partition graph *complete*.

```
ConstructPartitionGraphForMaterializedPredicates
(instruction stream + CFG) {
  for every compare instruction in CFG {
    cinst = current compare instruction
    qp = qualifying predicate of cinst
    bp = control predicate assigned to the basic block
      containing cinst
    p1 = the first destination predicate
    p1_old = the old definition of p1 if p1 is an update
    p2 = the second destination predicate if exists
    pp = parent predicate
    if( qp == p0 )  pp = bp  else  pp = qp

    switch (compare type) {
      case  .un.uc:
        Create a partition with pp as the parent predicate
        and p1 and p2 as child predicates
      case  .cn.cc:
        if( qp == p0 )  process in the same way as a
          .un.uc case
        else   map p1 and p2 to a dummy predicate node
      case  .an (.ac):
        Create a partition with p1_old as the parent predi-
        cate and p1 and an implicit predicate as child pred-
        icates
      case  .on (.oc):
        Create a partition with p1 as the parent predicate
        and p1_old and an implicit predicate as child predi-
        cates
        Create a partition with pp as the parent  predicate
        and p1 and an implicit predicate as child predicates
    }
  }
}
```

Figure 6. Algorithm of constructing partition graph for materialized predicates.

Based on the algorithms in Figures 5 and 6, the partition graph in Figure 7 is constructed to capture the global relations for predicates in Example 2. The basic block containing the if-clause is assigned with p0 as its control predicate to become the root of the partition graph. The then- and else- clauses are assigned with p_then and p_else as their control predicates, respectively. The initialization of predicates to false does not create any predicate node in the partition graph. The a partition is created at the switch point in the control flow due to the if-then-else construct. The b partition is created due to the compare instruction assigning the materialized predicates, p and q, in the then-clause. The c partition is created due to the compare instruction assigning the materialized predicates, r and s, in the else-clause. This partition graph tracks the global predicate relation that p, q, r, and s are disjoint, which is necessary to assert that x and y do not interfere with each other.

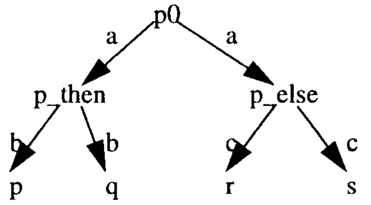


Figure 7. The partition graph for Example 2.

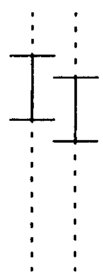
We have implemented the algorithms in Figures 5 and 6 and used them in our predicate-aware register allocator to obtain the experimental results in Section 6. There are two common patterns of compare instructions generated by our if-converter. For a program switch point, a cmpp.un.uc instruction is generated, and the algorithm in Figure 6 models this precisely. For an unstructured program merge point (often due to a critical edge), a combination of cmpp.un.uc and cmpp.on (or cmpp.oc) instructions is generated. The algorithm in Figure 6 is always correct but may be conservative for a comparison instruction with an *or* type. This is because in this initial implementation we did not track the compare conditions controlling predicates nor apply value numbering on predicates as proposed in [10]. For example, for the if-converted code in Figure 1(b), without value numbering we do not recognize that p3 and p4_1 can be mapped to the same predicate node and are unable to obtain that p4_2 is the union of p4_1 and px. However, this type of unstructured merge only accounts for a small portion (< 10% in our experience) of the if-converted compare instructions. In the future, we will model comparison instructions with an *or* type more precisely.

5 Predicate-aware Register Allocation

The problem of global register allocation is well known [4, 5, 6] and good heuristic approximations have been developed to solve the problem. However, predication creates new challenges for the established techniques. In the quest for greater instruction level parallelism through predication many register live ranges are created which, with current analysis techniques, will appear to overlap. Without applying knowledge of the relationships among predicates a large number of false interferences will arise. Our experiments show the resulting register allocation will conservatively allocate far more registers than are necessary.

Example 3:

```
loop:
S1:    p, q = cmpp.un.uc (a<b)   if true
S2:    x = ...                   if p
S3:    y = ...                   if q
S4:    ... = x                   if p
S5:    ... = y                   if q
S6:    z = ...                   if true
S7:    ... = z                   if true
S8:    w = ...                   if true
S9:    ... = w                   if true
       Branch loop
```



Example 3 illustrates the need for predicate-aware register allocation. This code sequence arises from if-conversion and simple list scheduling. As was pointed out in Section 1, without any knowledge of the relationship between p and q we will conservatively infer that x and y interfere. Additionally, it is difficult to conclude that a predicated definition ends a live range. To do so we must know that the definition's qualifying predicate is a superset of all predicates under which it is currently live. In this example, the live range of y begins at S5 but, without tracking all the predicates under which y is live, we cannot deduce that it is ended at S3. Conservative data flow analysis will then cause the live range for y to extend around the back-edge of the loop to interfere with w at S8 and z at S6. This originally short live range now interferes with every other live range defined in the loop. If the number of live ranges within a loop body of this type is greater than n, where n is the number of registers available on the target machine, spilling must occur. This example is presented in terms of liveness around the back-edge of a loop but the same situation also occurs in straight-line code.

When predicates cannot be accurately analyzed, the spilling of predicated code must be handled carefully to ensure that the graph coloring algorithm terminates. That is, if we are not able to infer the end of a predicated live range x correctly then we cannot do so for x's predicated spill code either.

Figure 8(b) shows the desired spill code for the two instruction sequence shown in Figure 8(a). To allow coloring to progress after spilling, without predicate analysis and from the code stream alone, we must arrange for each component of the spilled live range to be definitively ended by an unpredicated definition. To this end we insert unpredicated *kill* pseudo-ops to mark the end of live ranges.

```
x = ...     if m            y = ...          if m
                            st[spill] = y    if m
. . .                       . . .
                            ld z = [spill]   if n
... = x     if n            ... = z          if n
   (a)                         (b)

S1:    <Kill y>           if true
S2:    y = ...            if m
S3:    st [spill] = y     if m
       . . .
S4:    <Kill z>           if true
S5:    ld z = [spill]     if n
S6:    ... = z            if n
          (c)
```

Figure 8. Spill code difficulties: (a) Sequence to be spilled. (b) Desired spill code. (c) Worst case spill code.

There are a great number of practical problems in performing register allocation for predicated code. We have enhanced traditional global data flow analysis and interference graph construction to address these problems. Section 5.1 describes extensions necessary to enhance a traditional register allocation scheme to build a predicate-aware interference graph for a single basic block using conservative data flow information. Section 5.2 describes enhancements to the traditional global liveness data flow calculation, procedure-wide interference graph construction and the benefit of using global scheduling regions as boundaries for predicated live range analysis. Section 5.3 discusses approximations which may occur in live range analysis.

5.1 Local Analysis

Register allocators typically construct an interference graph by first calculating a bit-vector of all currently live registers at each point in the CFG. The bit-vector is initialized at the bottom of each basic block to all those registers which were computed to be *live out* by a previous pass of data flow analysis. The bit-vector is then propagated through the instruction stream to the top of the basic block. At each instruction an interference is recorded between each *def* and all elements of the bit-vector, then the *defs* are removed from the bit-vector and all *uses* added.

Our technique builds upon the standard bit-vector style interference graph construction algorithm. We first scan the basic block and collect the set of predicates used in the code stream and then augment the set with p0, the always true predicate. This set is termed the *basis of analysis* for the basic block or, more simply, the *basis*. We then allocate a bit-vector for each (predicate) element in the basis. This bit-vector represents liveness with respect to each of its basis elements. We call this data structure a *LiveSet*. We augment each LiveSet with an identifying basis *tag* to ease data flow propagation. In the absence of predicated code this scheme reduces to the traditional unpredicated case, since the basis contains only p0, and a LiveSet is a single bit-vector.

		w		x		y		z	
p0		0		0		0		0	
p		0		1		0		0	
q		0		0		1		0	

Figure 9. LiveSet for example in Example 3 after processing S4.

Figure 9 depicts the LiveSet for Example 3 after processing the instruction marked S4 in the backward traversal. Liveness is indicated by the bit value 1. This LiveSet represents the fact that x is live only under p and y is live only under q at this point. At S3, y is defined under q. Since x is live under p, and p and q are disjoint, we discover there is no interference between x and y.

In the next few subsections we will develop the machinery necessary to build the interference graph in the presence of predication. This machinery naturally extends from single basic blocks to arbitrary regions or entire procedures.

```
AssertLiveUnderPredicate(x, qp) {
   let A = { p in the basis | x is live under p }
   // Already live
   if qp is a member of A return
   // Minimize the liveness information
    B = leastUpperBoundSum(qp, A)
    mark x as dead under all predicates in
        the basis
    for each element r in B {
        if r is in the basis
          // Add x to the LiveSet
          liveSet[r] += x
        else
          for each element s in the basis
             if (!isDisjoint(r, s))
                // Approximation
                liveSet[s] += x
    }
}
```

Figure 10. Algorithm for asserting register liveness.

AssertLiveUnderPredicate - For each register x used in an instruction we will update liveness under the qualifying predicate qp. Instructions guarded by predicate true are treated as if they were predicated with the control predicate of the containing basic block. AssertLiveUnderPredicate is a function which asserts the liveness of a virtual register under a predicate. Pseudo code for this operation is shown in Figure 10 (All the pseudo code is written in an object-oriented manner: each method is assumed to be operating on a LiveSet). If x is already live under qp then no work is necessary. If x is not currently live under qp then we compute a minimal set of predicates which encapsulate the liveness of x by a call to leastUpperBoundSum. This allows us to keep the underlying representation compact. In the event that the minimal set contains a predicate which is not part of the basis we will conservatively update the liveness information to conform to the basis by checking disjointness with each basis element. This situation may arise due to the limiting the size of a basis.

AssertDeadUnderPredicate - For each register x defined in an instruction we will update deadness under the qualifying predicate qp. AssetDeadUnderPredicate is a function which asserts that a virtual register has *gone dead* under a predicate. The pseudo code for this operation is shown in Figure 11. We always minimize the number of basis elements required to represent the liveness of a variable. Therefore, if x is already live under qp we simply remove liveness under qp and return. Otherwise, we form a set of all predicates under which x is currently live and subtract qp from the set by using leastUpperBoundDiff to keep the number of predicates involved to a minimum. As is the case with AssertLiveUnderPredicate, some approximation may be introduced.

```
AssertDeadUnderPredicate(x, qp) {
        let A = { p in the basis | x is live under p}
        if qp is an element of A {
          // Remove x from the liveSet
          liveSet[qp] -= x
          return
        }
        // Call to predicate analysis
        B = leastUpperBoundDiff(qp, A)
        mark x as dead under all predicates in the basis
        for each element r in B
          if r is in the basis
            // Add x to the liveSet
            liveSet[r] += x
          else
            for each element s in the basis
              if (!isDisjoint(r, s))
                 liveSet[s] += x    // Approximation
        }
```

Figure 11. Algorithm for recording register deadness.

AllCurrentlyLive - In order to compute interferences or conservatively conduct an analysis across basic block boundaries we need to be able to produce a set of all registers which are currently live under a given predicate qp. For example, each def of an instruction interferes with the set of registers given by AllCurrentlyLive(qp), where qp is the qualifying predicate of the instruction. Data flow information can be propagated conservatively across basic blocks by use of AllCurrentlyLive(p0). The pseudo code for this operation is shown in Figure 12. AllCurrentlyLive computes a single bit-vector which is the union of all bit-vectors within a LiveSet whose basis element is not disjoint from qp.

```
AllCurrentlyLive(qp) {
   S = {}
   for each p in the basis
     if (!isDisjoint(p, qp))
        S += liveSet[p]    // Set union
   return S
}
```

Figure 12. Algorithm for producing a list of all registers live with respect to a given predicate.

Interference Graph Construction - The algorithm in Figure 13 describes, in general terms, how the interference graph is constructed in the presence of predicates. The method is quite similar to the traditional method of interference graph construction -- it is simply augmented by the use of LiveSet's and the support routines described above. By controlling the widths of the LiveSet's, the runtime of this method is kept in harmony with compile speed requirements[1].

```
BuildInterferenceGraph() {
  for each basic block bb in the program {
    // work is a LiveSet
    work = dataflowLiveOut[bb]
    for each instruction inst in bb in
    backward order {
      qp = inst.qualifyingPredicate()
      if (qp == P0) qp=bb.homePredicate()
      S = work.AllCurrentlyLive(qp)
      for each def d in inst
         interfere(d, S)   // Add edge to d from set S
      for each def d in inst
         work.AssertDeadUnderPredicate(d, qp)
      for each use u in inst
         work.AssertLiveUnderPredicate(u, qp)
    }
  }
}
```

Figure 13. Algorithm for building the predicate-aware interference graph.

1.In some cases the coloring algorithm as a whole can even be faster due to the simpler interference graph.

5.2 Global Analysis

The local framework allows for detailed analysis of single basic blocks and is sufficient to analyze Example 3. However, to obtain higher degrees of ILP, aggressive optimizations, such as global scheduling [2] or trace scheduling [13], must be employed to move instructions into ILP-poor basic blocks. In this section we extend the local framework to a global framework. The scheduling techniques cited above generally restrict their region of analysis to sub-regions of the CFG. Our approach recursively merges the bases of analysis of basic blocks within *global scheduling regions* (GSR's) to form a single common basis for data flow analysis.

```
RegionSelection() {
   for each interval I in the procedure
      in depth-first order {
      A = {}
       for each GSR R in I {
       B =  {basis for R}
        if I A+B I <= LIMIT {
          mark R as merged
           A += B
        }
     }
     for each direct descendent interval J of I {
        C = {basis of J}
         if I A+C I <= LIMIT {
           mark J as merged
           A += C
         }
     }
     set the basis of I and all merged GSR's
           and intervals to A
   }
}
```

Figure 14. Algorithm for selecting analysis regions.

Region Selection - Our approach is aimed at analyzing entire procedures. However, global analysis of predicated live ranges can be very expensive if limits are not placed on the algorithm. In practice, it may not be desirable to always analyze an entire procedure in an aggressive manner. We accommodate such cases by hierarchically building the regions of analysis from the most deeply nested portions of the CFG. That is, bases for several basic blocks are merged only if the size of the resulting basis does not exceed some limit[1].

Good predication can be accomplished only with intimate familiarity with the underlying machine architecture. In our approach we assume that most interesting predicated live ranges will be created by an if-converter and transported across basic blocks by a global instruction scheduler. Therefore, the scheduling regions used by a global scheduler will limit the scope of predicated code motion and hence the extent of predicated live-ranges[2]. Therefore, GSR's are a natural building block for basis construction for register allocation.

While GSR's limit scheduling induced code motion and thus the length of live ranges, naturally occurring (e.g. programmer created) live ranges will span many GSR's. Limiting the scope of analysis strictly to GSR's is too conservative. We recursively build-up the basis, first by attempting to coalesce the basis of all GSR's within an interval [12], and then coalescing the bases of inner intervals into a single basis of analysis. Intervals are coalesced in a depth-first traversal of the interval tree. In the best case this recursive selection will result in the entire procedure being analyzed with a single basis. In the event that multiple regions are selected, data flow calculations can then be performed accurately within each region and summarized at region boundaries. Figure 14 shows pseudo-code for the region selection process.

Finally, the basis used by each basic block in the program is recorded by *tagging* its LiveSet with a unique basis identifier. A bit-vector style data flow analysis can now be performed by extending traditional techniques.

```
LiveSetUnion(Target, Source) {
   if (Target.tag == Source.tag)
      for i=0 to Target.setCount - 1
         // Set union
            Target.liveSet[i] += Source.liveSet[i]
   else
      // Conservative
      Target.liveSet[p0] +=
                Source.AllCurrentlyLive(p0)
}
```

Figure 15. Algorithm for computing the union of LiveSet's.

Data Flow Transfer Functions - Standard bit-vector liveness calculation techniques can be used by extending the algorithm to use LiveSet's rather than simple bit-vectors. Some additional complication arises when propagating the information across basic block boundaries. To pass predicated liveness information across basic blocks boundaries we need two pieces of information about the source and target basic blocks:

1.The current liveness information.

2.The analysis basis used.

Information to identify identical bases is kept with each LiveSet as its basis tag. The liveness information is contained in the bit-vectors of each LiveSet. If the two

1.We used 32 as the limit in this study.

2.Recursive motion between nested GSR's may also occur. The recursive nature of our region selection attempts to address this issue.

LiveSet's use the same basis then data flow information can be transferred without approximation. Conservative liveness information is used otherwise. Figure 15 shows pseudo-code for a data flow LiveSet *union* operation. Other data flow operations are quite similar.

5.3 Approximation

In some cases the predicate analysis may return an answer containing predicates outside of the present basis. This can arise from levels of nesting outside the current region or from asymmetries in the CFG requiring internal predicate nodes to be generated (e.g. a critical edge) to represent the missing symmetrical relationship. These cases are handled conservatively.

Secondly, the analysis of predicate relations ignores back edges. Consequently, we cannot accurately analyze predicated live ranges which are defined on one iteration of a loop and consumed on a subsequent iteration. We detect such live ranges and use conservative data flow information for them at loop back edges.

6 Experimental Results

This section contains experimental measurements which show the effects of predicate-aware register allocation on a large number of procedures. Several benchmarks from the SPECint-92 suite were compiled and the number of *colors* required for procedure-wide graph coloring register allocation with and without our technique were recorded. In all, we compiled 1009 procedures and observed that 248 cases (24.6%) showed improved register allocation when using our technique. Of those cases which improved, the average improvement was a 20.71% decrease in the number of colors required. The standard deviation was 15.10%. Figure 16 graphically depicts the distribution of improvement. While most procedures saw improvements in the 1-35% range some procedures improved by as much as 75%. Some results of the experiments are shown in Table 2. The first column of Table 2 shows the name of the function compiled. The second column lists the number of colors required when using a traditional predicate-unaware graph coloring register allocator on the predicated code. The third column shows the number of colors required when using our predicate-aware register allocator. The fourth column shows the percentage of improvement.

While many procedures improved with our technique the majority did not. The dominant cause was absence of any predicated code in the procedure due to the simple nature of the if-converter [15]. With a more aggressive if-conversion technique [17] more programs should benefit from our work. Also, even when predicated code was present, it was sometimes of such a simple form as to allow the predicate-unaware method to achieve good results.

Since graph coloring based register allocation is a heuristic approximation to an optimal solution we expected to see some cases which degraded simply due to the different structures of the interference graphs. That is, the heuristic approximation used to color the predicate-unaware interference graph could use fewer colors than the same heuristic approximation used to color the predicate-aware interference graph. However, we have not observed any such case to date.

As the number of predicated live ranges increase in proportion to the number of unpredicated live ranges the impact of our approach should be even more beneficial. Furthermore, predicate-aware register allocation will be even more important when register pressure is increased by aggressive function inlining.

7 Conclusion

To maximize the effectiveness of predicated execution, it is very important to take into account the semantics of predicates during compilation analysis, such as data flow analysis. In this paper, we have proposed global techniques to analyze the relations among predicates, and these relations can then be queried by a subsequent compilation analysis or transformation phase. In contrast to previous work, our predicate analysis integrates the relations among control flow and predication. We have also developed a predicate-aware register allocator by naturally extending a traditional method. The precision of data flow analyses supporting this register allocator is greatly enhanced by taking into account global predicate relations. The importance of a predicate-aware register allocator is demonstrated by the significant reduction in register pressure as shown in our extensive experimental results. For cases which show improvement, on average, our predicate-aware register allocator reduces the number of colors by 20.71%.

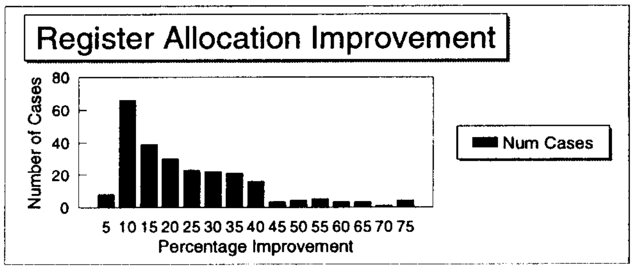


Figure 16. Distribution of register allocation improvement.

Table 2: Experimental results for register allocation

Function	Predicate-Unaware	Predicate-Aware	Improvement
set_dist	30	9	70.00%
xlremprop	13	4	69.23%
mv_reduce	102	42	58.82%
massive_count	101	44	56.44%
cdist	32	14	56.25%
explode	25	15	40.00%
map	63	39	38.10%
tagblock	19	12	36.84%
doprod	22	14	36.36%
yyparse	94	63	32.98%
sharp	29	20	31.03%
opo_leaf	31	22	29.03%
xlgo	14	10	28.57%
sweep	25	18	28.00%
closerow	22	16	27.27%
rmcvd	12	9	25.00%
opo	45	34	24.44%
openrow	29	22	24.14%
OpenConnection	22	17	22.73%
printfile	94	73	22.34%
xload	18	14	22.22%
dobindings	34	27	20.59%
duple	25	20	20.00%

Acknowledgements - We would like to thank the compiler optimization teams at California Language Lab and HP Labs for their support and comments on this work.

References

[1] J.R. Allen, K. Kennedy, C. Portfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," In Conf. Record of the 10th Annual ACM Symp. on Principles of Programming Languages, pp. 177-189, January 1983.

[2] David Bernstein and Michael Rodeh, "Global Instruction Scheduling for Superscalar Machines," SIGPLAN Notices 26(6):241-255. Proc. of the ACM SIGPLAN '91 Conference on Programming Languages and Implementation.

[3] D.P. Bhandarkar, *Alpha Implementations and Architecture*, Digital Press, Butterworth Heinemann, Newton, MA, 1996.

[4] Preston Briggs, *Register Allocation via Graph Coloring*, Ph.D Thesis, TR92-183, Rice University, 1992.

[5] Gregory J. Chaitin, "Register Allocation and Spilling via Graph Coloring," SIGPLAN Notices 17(6):98-105, June 1982. Proc. of the ACM SIGPLAN '82 Symp. on Compiler Construction.

[6] Fred C. Chow and John L. Hennessy, "The Priority-Based Coloring Approach to Register Allocation," ACM Trans. on Programming Languages and Systems, 12(4):501-536, October 1990.

[7] Alexandre E. Eichenberger and Edward S. Davidson, "Register Allocation for Predicated Code," In Proc. of the 28th Annual Int'l Symp. on Microarchitecture, November 1995.

[8] Hewlett-Packard Company. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, Second edition, 1992.

[9] P.Y. Hsu and E.S. Davidson, "Highly Concurrent Scalar Processing," In Proc. of the 13th Annual Int'l Symp. on Computer Architecture, pp. 386-395, June 1986.

[10] Richard Johnson and Michael Schlansker, "Analysis Techniques for Predicated Code," In Proc. of the 29th Annual Int'l Symp. on Microarchitecture, December 1996.

[11] V. Kathail, M. Schlansker, B. Rau, *HPL PlayDoh Architecture Specification: Version 1.0*, Hewlett-Packard Laboratories Technical Report, HPL-93-80, Feb. 1993.

[12] T. Lengauer and R. E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flow Graph", ACM Trans. on Prog. Languages and Systems, 1:1, pp 121-141, July 1979.

[13] P. G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell and J.C. Ruttenberg, "The Multiflow Trace Scheduling Compiler," The Journal of Supercomputing, 7, 51-142, 1993.

[14] S.A. Mahlke, D.C. Lin, W. Y. Chen, R.E. Hank, and R.A. Bringmann, "Effective Compiler Support for Predicated Execution Using Hyperblock," In Proc. of the 25th Annual Int'l Symp. on Microarchitecture, pp. 45-54, Dec. 1992.

[15] J.C.H. Park and M. Schlansker, *On Predicated Execution*, Technical Report HPL-91-58, Hewlett-Packard Laboratories, May 1991.

[16] B.R. 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, Jan. 1989.

[17] Michael Schlansker and Vinod Kathail, "Critical Path Reduction for Scalar Programs", In Proc. of the 28th Annual Int'l Symp. on Microarchitecture, November 1995.

[18] N.J. Waters, S.A. Mahlke, W.-M. W. Hwu, and B.R. Rau, "Reverse If-conversion," In Proc. of the SIGPLAN'93 Conf. on Programming Language Design and Implementation, pp. 290-299, June 1993.

[19] D. Weaver and T. Germond, *The SPARC Architecture Manual - Version 9*. Prentice-Hall, Englewood Cliffs, NJ 1993.