Interprocedural Conditional Branch Elimination†

Rastislav Bodík Rajiv Gupta Mary Lou Soffa

Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{bodik,gupta,soffa}@cs.pitt.edu

Abstract

The existence of statically detectable correlation among conditional branches enables their elimination, an optimization that has a number of benefits. This paper presents techniques to determine whether an interprocedural execution path leading to a conditional branch exists along which the branch outcome is known at compile time, and then to eliminate the branch along this path through code restructuring. The technique consists of a demand driven interprocedural analysis that determines whether a specific branch outcome is correlated with prior statements or branch outcomes. The optimization is performed using a code restructuring algorithm that replicates code to separate out the paths with correlation. When the correlated path is affected by a procedure call, the restructuring is based on procedure *entry splitting* and *exit splitting*. The *entry splitting* transformation creates multiple entries to a procedure, and the *exit splitting* transformation allows a procedure to return control to one of several return points in the caller. Our technique is efficient in that the correlation detection is demand driven, thus avoiding exhaustive analysis of the entire program, and the restructuring never increases the number of operations along a path through an interprocedural control flow graph. We describe the benefits of our interprocedural branch elimination optimization (ICBE). Our experimental results show that, for the same amount of code growth, the estimated reduction in executed conditional branches is about 2.5 times higher with the ICBE optimization than when only intraprocedural conditional branch elimination is applied.

Keywords: interprocedural data flow analysis, conditional branch correlation, path-sensitive optimization, optimization of object-oriented languages.

1 Introduction

Recent research in branch prediction [16, 24, 25], profiling [3], and the elimination of conditional branches [19] has reported the existence of significant amounts of correlation among conditional branches, presenting opportunities for optimizations. A conditional branch has *static correlation* along a path if its outcome can be determined along the path at compile time from prior statements or branch outcomes. Such a conditional branch is redundant along the *correlated path* and can be eliminated by code restructuring. Elimination of conditional branches has a number of benefits, which are discussed in Section 6, including

- enhancing instruction scheduling and software pipelining,
- improving speculative execution and hardware branch prediction, and
- optimizing C++/Java virtual functions.

Previous work on conditional branch elimination through static correlation [19] demonstrated substantial performance improvements despite its restricted focus on eliminating conditionals within loops. Experimentally, we show that substantially more static correlation is detected at compile time when programs are analyzed interprocedurally. Using programs from the SPEC95 suite, we discovered that interprocedural detection of correlation enables elimination of 3% to 18% of executed conditionals, which is a factor of about 2.5 improvement over strictly intraprocedural analysis. As illustrated below, this high correlation among branches when procedures are considered is due to the modular fashion in which we write procedures:

- In a procedure, the value returned is often selected by an if-statement. This value may again be checked by the caller. For example, consider a call to a procedure that removes an element from a linked list. The procedure tests whether the list is empty and, if so, returns *nil*. The caller performs an identical test on the return value to determine if *nil* was returned. The later test is fully correlated with the earlier one.
- In order to keep the procedure interface simple by passing few arguments, procedures frequently include checks on the parameters that are also performed by the caller or even by previous calls to the same procedure. For example, procedures from the same library module may be called one after another, propagating values. These procedures often perform correlated tests on the propagated values. With our optimization, the repeated testing can be eliminated.

Conditional branch elimination is a form of partial redundancy elimination (PRE). However, the code motion techniques useful for PRE of assignments [14, 5] do not suffice for removing conditional branches. To eliminate a conditional, the control flow graph must be restructured in order to separate the correlated path from the rest of the paths [19]. After code replication isolates the correlated path, the

†Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371, a National Science Foundation grant CCR-9402226, and a grant from Hewlett-Packard to the University of Pittsburgh.

conditional on this path becomes fully redundant and can be removed. Procedures are traditionally viewed as inherently single-entry/single-exit regions of code which means that all paths through the procedure must pass through the unique entry and exit points. To exploit interprocedural opportunities for conditional branch elimination, the correlated paths crossing procedure entry/exit must be isolated by splitting procedure entry/exit nodes.

In this paper, we present a new optimization, Interprocedural Conditional Branch Elimination (ICBE). The optimization consists of a demand-driven interprocedural static correlation analysis and a code restructuring algorithm that uses the detected correlation to eliminate conditional branches. We implemented the optimization and experimentally investigated the amount of interprocedural correlation detected and the benefits and costs of conditional branch elimination.

Demand-driven interprocedural correlation analysis. Using the demand-driven data flow framework for distributed data flow problems [9], we developed a demand-driven correlation detection analysis algorithm. The analysis is interprocedural and thus considers correlated paths spanning procedural boundaries, as well as correlations that occur within the same procedure. In the analysis phase, given a conditional branch, a query propagation search is performed to find assertions on program variables that indicate the correlation along paths leading to the conditional. The exhaustive approach to analysis would naively determine at each node all existing assertions on program variables (including irrelevant ones), resulting in exponential worst-case analysis time. Because our analysis is restricted to discovering useful assertions on relevant variables, we are able to achieve polynomial analysis time. If correlation is found along some path, code restructuring by path duplication is required to eliminate the conditional. Our analysis determines the upper bound on the amount of code duplication required to eliminate the conditional. If profile information is available, our analysis also provides an estimate of the reduction in the execution frequency of the conditional. The above measures can be used to guide the optimizer in making a decision on whether and how to transform the program to eliminate the conditional.

Interprocedural restructuring. The correlated conditional is eliminated by path duplication, which separates paths with correlation from those where correlation was not detected. Two approaches to interprocedural elimination of branches can be used. The first is to inline procedures involved in the correlation and duplicate paths strictly within a procedure. We present an alternative approach that incurs less code growth than inlining because only paths with correlation are duplicated. Another advantage of our algorithm is that it enables optimizations of call sites where inlining is not possible [2]. The algorithm is based upon procedure entry splitting and exit splitting. The *entry splitting* transformation creates multiple entries to a procedure through which the procedure can be entered from different call sites. The *exit splitting* transformation allows a procedure to return control to one of several return points in the caller instead of always returning control to the call site. Intuitively, entry splitting enables the elimination of interprocedurally redundant code in the callee, and exit splitting enables elimination in the caller. Our restructuring algorithm restructures a control flow graph such that the number of operations in the control flow graph does not increase along any path. After exit splitting, additional return addresses may need to be passed during a procedure call. Within the scope of a procedure, our restructuring algorithm is similar to that of Mueller and Whalley [19], except that our restructuring techniques takes advantage of correlation that spans nested loops. Our algorithm is able to create two versions of a loop, one for each known outcome of the conditional, enabling the elimination of the conditional in each loop version.

Experimental evaluation. Our measurements performed on a set of application programs provide insight into the interprocedural correlation that can be detected statically and its usability for compiler optimizations. We found that not only the number of conditionals with some correlated paths greatly increases with interprocedural analysis, but also the effect of branch elimination is more significant because many short, frequently taken interprocedural correlated paths exist. Since some correlated branches may need long analysis times to detect the correlation and also require extensive code replication, we developed and evaluated simple heuristics that control the extent of the analysis and the amount of code growth due to ICBE. We show that for the same code growth, ICBE removes significantly more executed conditional branches than what is possible with the intraprocedural conditional branch elimination optimization.

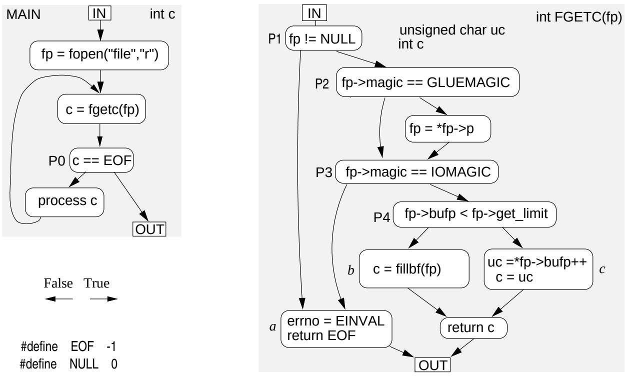
Intraprocedural elimination of conditional branches in loops was developed by Mueller and Whalley [19]. We extend their technique in several respects. First, we can detect and eliminate partial redundancy of branches in loop nests and across procedure boundaries. Second, even in the scope of a single procedure, our approach is more powerful because we can detect redundancy that is apparent by examining multiple basic blocks along a path, as opposed to a redundancy due to a single basic block detected in their analysis. In addition, in our technique, the analysis cost and the code growth incurred due to program restructuring can be controlled. Mueller and Whalley [18] also investigated avoiding unconditional jumps by code replication. Krall [16] developed code replication techniques to improve the accuracy of semi-static branch prediction to the accuracy of dynamic prediction.

This paper is organized as follows. The next section presents an example to motivate the technique and gives the overview of ICBE. Section 3 presents the algorithms and Section 4 presents our experimental results. Section 5 highlights the benefits of ICBE.

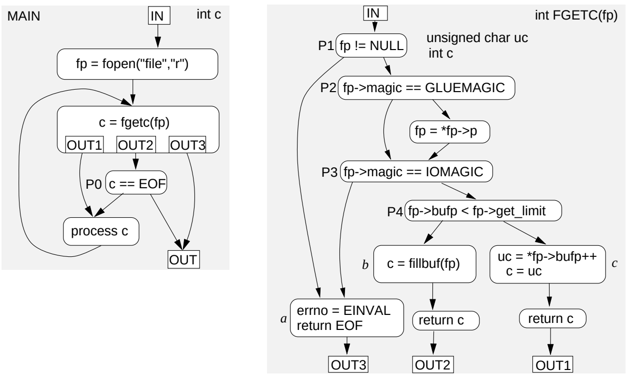
2 An Example and Overview

We illustrate ICBE on a small application program that uses the stdio GNU C library (glibc1.09). The program is shown in Figure 1(a). Function MAIN first opens a text file by a call to fopen and then iterates through each character in the file until EOF is reached. The characters are obtained by a call to fgetc, which returns a character from a buffer that is filled by calling fillbuf.

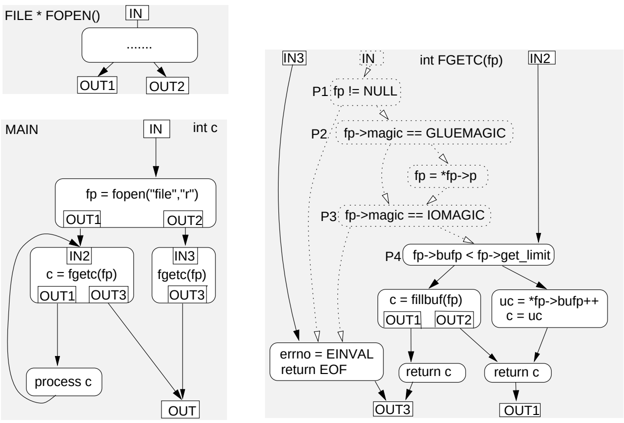
First consider applying ICBE to optimize the conditional P0 in MAIN. Since EOF equals -1, our demand driven analysis algorithm raises the query $(c = -1)$ at P0 and propagates it backwards into function fgetc where the analysis quickly terminates at nodes a, b, and c with answers TRUE, UNDEF, and FALSE, respectively. The TRUE result at a means that along the path from node a to P0, the outcome of P0 is true.



(a) The source program.



(b) After optimization of P0.



(c) Elimination of P1, P2, P3; exit splitting on fillbuf.

Figure 1: The example program using the GNU C library.

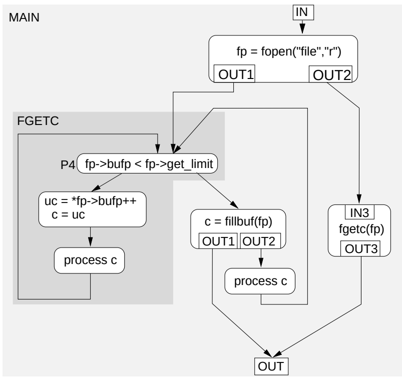


Figure 2: Inlining of optimized fgetc.

Thus, P0 is statically correlated along the path from a to P0. Node c fetches from the buffer an unsigned character with a non-negative value which resolves the query to FALSE indicating that the outcome of P0 is false. Thus, P0 is statically correlated along the path from node c to P0. Assuming for now that the code of function fillbuf is unavailable and nothing is known about its return value, the query must resolve to UNDEF at the call site node b, meaning that the behavior of P0 cannot be determined along the path from node b to P0 and thus it is not statically correlated along this path. From the analysis, the conditional P0 is redundant along two out of three paths reaching it. Our optimization algorithm restructures the program by exit splitting to separate the paths from node a to P0 and from node c to P0. After the transformation, the conditional P0 is bypassed each time, except when the buffer is refilled at node b (see Figure 1(b)). Exit splitting is implemented by passing additional return addresses to the callee.

Next, we optimize conditionals P1, P2, and P3 in function fgetc. The individual queries raised at these conditionals are all resolved in function fopen where the analysis reveals that either fp = NULL or fp $\neq$ NULL $\wedge$ fp->magic = IOMAGIC holds. In either case, all three conditionals are *fully* redundant (that is, they can be eliminated along all paths). Our optimizer splits the exit of fopen and the entry of fgetc to bypass these conditionals. The result is shown in Figure 1(c). The statements in fgetc that are reachable only from its original entry can be deleted if no other call site of this entry exists.

When the code of fillbuf is available, propagation of the query raised at P0 does not terminate at its call site but detects that fillbuf either returns EOF or an unsigned value, resolving the query to TRUE and FALSE, respectively. The resulting exit splitting of fillbuf enables the complete elimination of conditional P0, shown in Figure 1(c).

In the original loop, during each loop iteration five conditional branches are executed. After the optimization, only one conditional remains. This optimization cannot be carried out by intraprocedural branch elimination [19]. The overhead of the optimization is passing two return addresses to fgetc. However, the additional instructions related to argument passing can be freely scheduled ahead of the procedure call because they have no incoming data dependencies. Furthermore, the code size of procedure fgetc is reduced

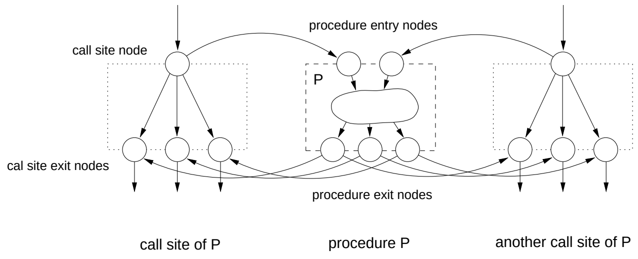


Figure 3: Interprocedural CFG in call site normal form.

which enables its inlining into MAIN, where the resulting loop can be efficiently pipelined (see Figure 2). We discuss the use of inlining further in Section 5.

3 The ICBE Optimization

For each conditional branch considered, the ICBE optimization performs analysis followed by restructuring. First, the conditional is analyzed to detect correlated paths and to determine the amount of code duplication required to eliminate the conditional. If correlation is found and the demands on code growth are acceptable, the program is restructured to create paths along which the conditional and instructions that compute its predicate condition are eliminated. The analysis and restructuring algorithms are explained in the following subsections.

The *interprocedural control flow graph (ICFG)* used in the algorithm is a graph that combines CFGs of all program procedures by connecting procedure entries and exits with their call sites as depicted in Figure 3. All edges in the figure define the predecessor-successor relation for nodes. Each procedure can have multiple *procedure entry* nodes and multiple *procedure exit* nodes. The successors of a *call site* node are the procedure entry node and the associated *call site exit* nodes. The analysis algorithm requires the ICFG to be in the *normal call site form*, where a) each call site node has a single procedure entry successor and b) each call site exit node has exactly one call site predecessor and one procedure exit predecessor. We assume that the above nodes are dummy nodes with no program statements.

3.1 Interprocedural Correlation Detection

Our analysis is demand-driven from a given conditional in the sense that only the nodes that may lie on a correlated path are visited and only the relevant data flow information is computed. The analysis is initialized by raising a query at the conditional that corresponds to asking a question "is the outcome of the conditional with the predicate (v relop c) known along some incoming paths?" The form of the raised query is (v relop c), where v is a variable and c a constant. The query is then propagated from the conditional backwards along all paths in the ICFG until it can be resolved on these paths. Resolving a query at a node produces one of three answers: TRUE, FALSE, UNDEF. The first two answers indicate that the path along which the query reached the node is correlated. TRUE means that the outcome of the conditional along the path is true and FALSE means the opposite. The UNDEF means that the outcome is unknown because the variable is assigned an unknown value.

For resolving a query, we have identified four sources of static correlation. First, a query is always resolved TRUE or FALSE at a node that assigns a constant to the variable v from the query. The second source is a conditional branch that involves the variable v. The assertions on variables that exist on the true and false out-edges of the conditional may define the outcome of the predicate in the query. Note that a conditional correlates with itself if there is a path around a loop along which the query variable is not defined. The third source is a type conversion from an unsigned to signed value, as in the example in Figure 1(a). The result is always non-negative, which may determine the branch predicate outcome. Last, after a pointer variable is dereferenced, its value is guaranteed to be non-zero; otherwise a segmentation fault would have occurred.

During the propagation, a copy assignment to the query variable may be encountered, e.g., v := w. When this happens, the query is modified to reflect this assignment before it continues to propagate. This simple form of symbolic back-substitution is essential to capture assignments to and from temporaries, common subexpressions, procedure return values, and parameter passing. As a consequence of this substitution, multiple distinct queries can be raised at a single node. Our analysis can support more general symbolic back-substitution and is restricted only by the capabilities of symbolic manipulation available in the compiler. Since query propagation may not terminate under a general symbolic analysis, we stop query propagation with the UNDEF answer when a sufficient number of nodes has been processed.

After the analysis terminates, the resolved queries are rolled back along the paths they traversed. The goal is to collect all resolved answers to each query raised at a node. Starting at the successors of nodes where a query was resolved, answers are propagated forward and merged by a set-union operation at control flow merge nodes. At any node, including the conditional itself, each query may have from one to all three possible answers from {TRUE, FALSE, UNDEF}. For example, if the query raised at the conditional has answers TRUE and FALSE, then there are some correlated paths leading to the conditional where the outcome is true, some correlated paths where it is false, and no paths along which it is unknown. Such a conditional has full correlation.

The demand-driven framework of [9] computes procedure summary nodes on demand to improve the efficiency of interprocedural analysis. Since in our analysis the queries are propagated through procedures backwards, summary node entries are stored at procedure exit nodes and for each query raised at the exit node we maintain: a) the answers resolved in the procedure, and b) the corresponding queries at each entry of the procedure, if the query propagated all the way to the entry node. (Remember that the analysis is invoked on a restructured program in which procedures can have multiple entries.) All queries raised at procedure exit nodes are used to compute summary nodes and are, therefore, treated specially. When a *summary node query* reaches a procedure entry, it is not propagated to the callers, but resolved with the fourth kind of query answer, TRANS. This

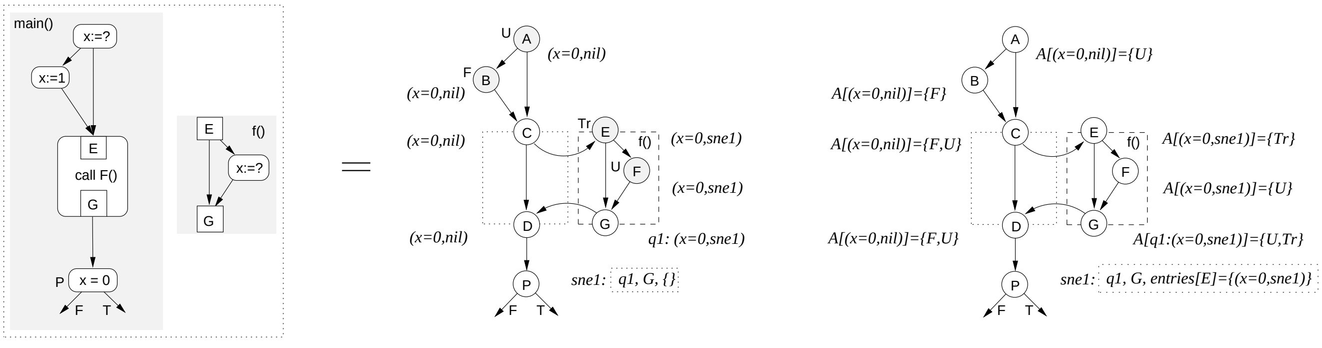
```
Analyze predicate (v relop c) in conditional branch node b
1   initialize Q[n] to {} at each node n
2   form the initial query q_b = (v, relop, c, nil)
3   raise_query(pred(b), q_b)
4   while worklist is not empty do
5       remove pair (node n, query q) from worklist
6       case n is entry node of a procedure p:
7           if q is a summary node query then A[n,q] := TRANS; add q to q.sne.entries[n]
8           else if n has no predecessors then A[n,q] := UNDEF
9           for each call site node predecessor m of entry node n do
10              if q is a summary node query for jth exit of p then
11                  if q.sne.q_in is raised at jth exit of m then raise_query(m,q)
12              else raise_query(m,q)
13          end for
14      case n is call site exit node:
15          let ex be the procedure exit predecessor of n
16          let m be the call site predecessor of n and en the entry node invoked by m
17          if summary node entry sne[ex,q] does not exist then
18              let q_sn be a copy of q
19              sne[ex,q] := (q_sn, ex, {}); q_sn.sne := sne[ex,q]
20              raise_query(ex, q_sn)
21          else if sne[ex,q].entries[en] does not exist then
22              sne[ex,q].entries[en] := {}
23              raise_query(ex, sne[ex,q].q_in)
24          end if
25          add A[ex, sne[ex,q].q_in] \ {TRANS} to A[n,q]
26          for each query q_a in sne[ex,q].entries[en] do raise_query(m, q_a)
27      otherwise :
28          answer := resolve(n,q)
29          if answer ∈ {TRUE, FALSE, UNDEF} then A[n,q] := {answer}
30          else for each m ∈ Pred(n) do raise_query(m, substitute(n,q))
31      end case
32  end while

    Procedure raise_query(node n, query q)
33      if q ∉ Q[n] then add q to Q[n]; add pair (n,q) to worklist
    end
```

Figure 4: The interprocedural static correlation analysis.

answer marks paths through the procedure along which the query was not resolved. The procedure is *transparent* along such paths and the summary node lookup must propagate queries (backward) and collect answers (forward) across call sites of transparent procedures. Our analysis handles both call-by-value and call-by-reference parameters.

The analysis algorithm is given in Figure 4. The algorithm computes summary nodes without interrupting the analysis. Each query is a tuple $(v, \mathrm{relop}, c, sne)$, where sne is used by summary node queries to keep a pointer to their summary node entries; for non-summary queries, this field is nil. The summary node entry for query q raised at exit node ex is a tuple $sne[ex, q] = (q_{sn}, ex, entries)$, where q_{sn} is the summary node query raised on the procedure exit node ex and $entries[en]$ is the set of queries propagated to a particular entry node en. The analysis is started at line 3 by raising the initial query at the predecessor of the conditional to be analyzed. Line 4 terminates the analysis when no node with an unresolved query remains. Lines 6–13 handle procedure entry nodes. Summary node queries are resolved here to TRANS and are added to the summary node entry as having reached the particular entry node, as described above. The non-summary query is propagated to all call sites of this entry (lines 9 and 12). The summary node query is propagated only when the computation of the summary node was initiated at the exit of the call site (lines 9–11). Lines 14–26 process a call site exit node n. Predecessors of n are determined according to Figure 3. If summary node lookup in line 17 fails, a new summary node entry is created and a summary node query q_{sn} is raised. Lines 21–23 update the summary node after a previous split of a procedure entry/exit node. Line 25 resolves the query based on the answers saved in the summary node and line 26 propagates the query across the procedure when a transparent path through the procedure exists. Finally, any other kind of node may be a source of correlation (lines 27–30). Function **resolve** attempts to resolve a query. If it fails, the query is propagated after it is back-substituted. The algorithm for collecting the analysis answers by query rollback can be easily derived from the analysis algorithm, and we omit it from this paper.

During the rollback, the upper bound on the amount of code duplication required to optimize the conditional is



(a) source program (b) analysis (c) rollback

Figure 5: An example of interprocedural correlation analysis.

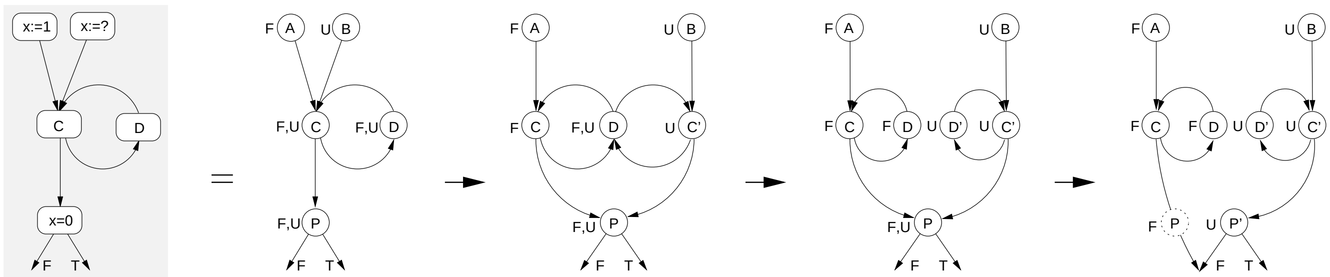


Figure 6: Intraprocedural restructuring.

calculated by determining how many copies of each node must be created. Given a node with k answers to a query raised at that node, the node must be split k-ways (k is at most 4). When the node hosts more than one query, then the number of copies needed is bound by the cross product of answers to all queries raised at that node. The actual code growth is usually lower due to the fact that a node split on one query may separate answers to other queries raised at that node.

We illustrate the analysis with an example in Figure 5. The four possible query answers are abbreviated in the figure as T, F, U, and Tr, for query answers TRUE, FALSE, UNDEF, and TRANS. The analysis of conditional node P is initiated by raising a query $q : (x = 0, nil)$ at the predecessor of P (Figure5(b)). The entry nil signifies that the query does not compute a summary node entry. Since x is a global variable, it cannot be propagated across the intraprocedural edge (C, D). Instead, it is raised at the exit of procedure f, where it initiates computation of a summary node entry sne_1. The summary node entry is computed by raising a summary node query $q_1 : (x = 0, sne_1)$ at the procedure exit node G. The query is resolved at node F to UNDEF because an unknown value is assigned to x. The nodes where a query is resolved are highlighted in the figure. The scope of the summary node is limited to the procedure and hence q_1 is resolved at the procedure entry node to TRANS. Also, the query is recorded in the $entries$[G] field of the summary node entry. Whenever a summary node query reaches the procedure entry, a corresponding query is raised at the call site node. In our case, query $q : (x = 0, nil)$ is raised at node C. This query is subsequently resolved at nodes A and B to UNDEF and FALSE, respectively. The analysis is followed by the rollback phase (Figure 5(c)). The answer for a query q is stored in $A[q]$ and consists of all answers for q reaching the node. Note that the UNDEF answer for q at node D was propagated from node C through the TRANS answer of the summary node query. The following section and the example in Figure 7 show how the collected answers are used to restructure the ICFG.

3.2 CFG Restructuring

The restructuring of the ICFG is performed to isolate correlated paths and then remove the correlated conditionals on these paths. The underlying technique is to split each node on which a query has multiple answers so that each duplicate of the node can host a single answer.

Intraprocedural Restructuring. Restructuring, when the correlation is not affected by a procedure call or return, is similar to that proposed by Mueller and Whalley [19] except that we handle correlations that cut across loop iterations. Restructuring proceeds in the forward direction starting from each node that hosts multiple answers to a query and at least on one of its predecessors hosts only a single answer. Figure 6 illustrates intraprocedural restructuring

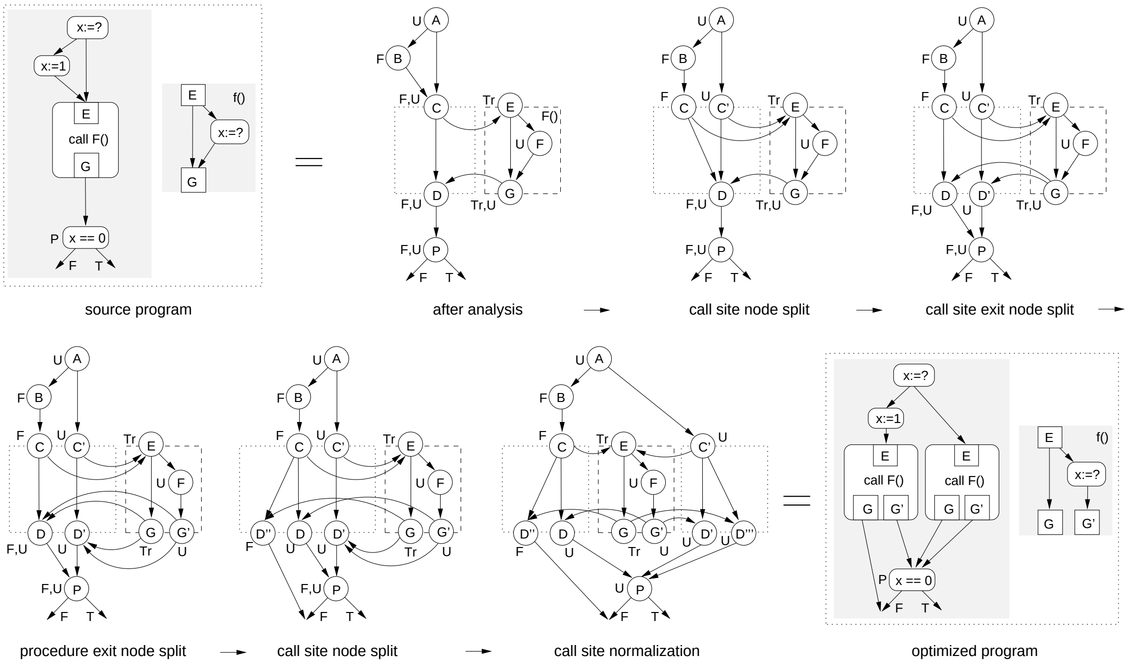


Figure 7: Interprocedural restructuring.

that duplicates a loop; on the left is the source program, followed by the equivalent CFG labeled with the answers to the query raised at node P: x=0. Restructuring starts at the loop header C and continues until every node hosts only a single query answer. At this point, the correlated path has been separated. In the last step, after the conditional itself is split, the copy of the conditional that hosts the TRUE or FALSE answer is redundant and is removed.

Interprocedural Restructuring. Because of the representation of the ICFG in our algorithm (see Figure 3), the splitting of the procedure entry and exit nodes does not require special handling. *Entry splitting* occurs when the correlated path is entering the procedure through a procedure entry node. Entry splitting always involves call site splitting. *Exit splitting* occurs when a correlated path crosses a procedure exit node. Exit splitting always involves splitting call site exit nodes and requires passing additional return addresses to the procedure. Figure 7 illustrates interprocedural restructuring. The nodes of the ICFG are labeled with answers to the queries raised in the analysis shown in Figure 5. The splitting process starts by duplicating the call site node C, followed by splitting of the call site exit node D. Note that after these two steps the node D', which corresponds to the call site node C', hosts a single answer, while the node D hosts two answers. After splitting of the procedure exit node G, node D can be split again, resulting in each copy of D hosting a single answer, thus enabling the optimization of the conditional P. After the optimization of the conditional is complete, the graph must be converted to *call site normal form*, as defined previously. The last figure illustrates the converted graph.

The restructuring algorithm is shown in Figure 8. The initial worklist is determined during the analysis phase. Before a node n is split under a query q at line 7, it is verified at line 5 that all answers to q that are hosted at n are still present at some predecessor of n. The answers may have disappeared if n was previously split on some other query and some predecessors have been disconnected from n. If an answer was removed at line 7, then the call to **fix-edges** removes edges so that only nodes hosting the same answer for a query are connected. The successors of n are added to the worklist when the node is split in order to continue splitting in the forward direction, or when an answer was removed in order to adjust the answers and edges at successors. Line 2 terminates the restructuring if there is no node that requires splitting, adjusting incident edges, or removing an answer to a query. Procedure **split** duplicates a node to create a copy of the node for each answer hosted on the original node. Each new copy is added to the worklist because it may still need to be split under the remaining queries raised at the original node. Lines 15 and 16 perform the actual elimination of the conditional; when the conditional is split under the query that was initially raised on it, the copies hosting TRUE and FALSE answers are fully redundant and are removed.

3.3 Complexity and Safety

The cost of the ICBE optimization is determined by the asymptotic complexity of the correlation analysis. Although the worst-case time complexity for the restructuring phase is exponential, it is only polynomial for the correlation analysis. The cost of the analysis dominates because ICBE performs restructuring only when the required code growth is moderate, resulting in the size of the ICFG always remaining

Restructure interprocedural CFG to eliminate branch node b

begin
  let *worklist* be set of nodes n s.t. a query has multiple answers at n and a single answer at $m \in Pred(n)$
  while *worklist* is not empty **do**
    remove a node n from *worklist*
    for each query q from $Q[n]$ **do**
      remove from $A[n, q]$ answers to q no longer hosted at predecessors of n
      fix-edges(n, q)
      if $A[n, q]$ contains multiple answers **then split**(n, q)
      if n was split or answer was removed from $A[n, q]$ **then** add Succ(n) to *worklist*
    end for
  end while
  for each visited call site m **do** normalize m
end

Procedure split(node n, query q)
  for each answer a from $A[n, q]$ **do**
    let n_a be a duplicate of n including incident edges and $Q[n]$, $A[n, \star]$, $sne[n, \star]$ information
    set $A[n_a, q]$ to $\{a\}$; add n_a to *worklist*
    if $n = b$ and $q = q_b$ and $a \in \{\mathsf{TRUE}, \mathsf{FALSE}\}$ **then**
      change n_a into empty node and remove edges to false/true successors
    end if
    fix-edges(n_a, q)
  end for
  remove n and edges incident on n
end

Procedure fix-edges(node n, query q)
  remove each edge (m, n) s.t. n and m no longer host a common answer for q
end

Figure 8: The interprocedural restructuring algorithm.

within a constant factor of its original size. Let V denote the number of program variables, P the number of conditional nodes, and N the total number of nodes in the ICFG. When the format of the analyzed predicate expressions is restricted to (*var* relop *const*), and only copy-assignment statements v := w are interpreted, at most V different queries can be created during the analysis of a single conditional. Consequently, at most V queries can be raised at each node. In addition, we also have to consider the summary node queries raised at each node. For each query, multiple summary node queries may be raised at each node, one for each exit of the relevant procedure. Since conservative application of restructuring keeps the number of procedure exits within constant bounds, $O(NV)$ node-query pairs are inserted into the worklist at line 33 during the analysis of each conditional. This results in $O(PNV)$ cost to analyze all conditionals. We should point out that the rollback algorithm is bound by the same cost. Even though the data flow problem of static branch correlation does not conservatively merge query answers at confluence nodes (but instead propagates forward their set-union), each query can have at most four possible answers. As a result, each node-query pair needs to be updated at most four times during propagation of query answers.

The analysis cost can be reduced by *caching* at all nodes the results of all queries resolved in previous analyses. The overall number of queries is bound by $6VC$, where 6 is the number of different relational operators and C the number of unique literals that appear in predicates of conditionals (C is typically a small number). The cost with query caching is then $O(CNV)$. However, maintaining the cache proved counterproductive in our implementation due to increased memory requirements.

The ICBE optimization is *safe* because it never increases the number of operations along any path in the ICFG. However, argument lists of procedures with split exits must be extended in order to communicate alternative return addresses. It is highly likely that eliminating a conditional at the expense of passing additional arguments is advantageous because return addresses are all compile-time constants and, having no incoming data dependences, argument passing instructions can be scheduled more freely than conditionals. Furthermore, some of the return addresses may be unnecessary at a procedure entry because not all procedure exits may be reachable from the entry (see Figure 1(c)). Alternatively, if some call site cannot take advantage of the split exits in the callee, an unmodified copy of the callee may be called from this call site, eliminating the need to pass additional arguments.

4 Experiments

Implementation. We implemented the analysis and restructuring algorithms in our interprocedural compiler that is based on the retargetable compiler `lcc` described in [10]. Our implementation considered the correlation of those conditionals that compared a scalar variable (not a structure

Benchmark program	source lines	procedures		nodes		cond/prog [%]	
		defined	library	all	cond	static	dynamic
099.go	29 246	372	11	38 806	5 304	21.4	29.0
124.m88ksim	19 915	252	35	21 657	2 416	16.5	30.9
129.compress	1 934	24	6	957	89	13.5	20.9
130.li	7 597	357	26	10 718	875	12.9	26.7
132.ijpeg	31 211	467	30	25 420	2 355	12.2	11.7
134.perl	26 871	276	69	50 596	5 623	16.6	29.1
147.vortex	67 202	923	63	104 154	9 646	12.9	28.0
lcc.3.5	26 467	470	21	49 775	5 863	18.1	32.1

Table 1: Benchmark programs.

Benchmark program	time [*sec*]		memory [MB]		node-query pairs	
	overall	analysis	progrep	analysis	total	per cond
099.go	98.4	83.8	50.4	1.7	198 180	120.1
124.m88ksim	56.1	40.0	67.3	1.9	236 252	168.8
129.compress	2.1	0.7	10.4	0.3	6 620	120.4
130.li	9.8	4.6	35.9	0.8	27 201	102.6
132.ijpeg	33.3	19.8	52.7	0.6	32 961	33.5
134.perl	135.2	117.0	49.6	2.6	317 719	197.6
147.vortex	1070.2	1016.9	119.3	3.4	1 378 890	241.5
lcc.3.5	166.4	138.1	60.5	2.4	352 089	217.5

Table 2: The cost of correlation analysis.

member) with a constant. Overall, 45% of conditionals in the benchmark programs could be analyzed using this pattern. We implemented both an intraprocedural optimization, which used MOD and USE [7] procedure summary information at call sites, and the ICBE optimization that considered both intra- and interprocedural correlations. The analysis recognized two sources of correlation: constant assignments and conditional branches.

Benchmarks. The experiments were performed on the integer SPEC95 suite. Since `lcc` does not generate correct code for the `126.gcc` benchmark, we used `lcc` itself as a compiler benchmark program. The programs are characterized in Table 1. The number of procedures, both defined in the program as well as the library procedures called are given in the table. The correlation analysis did not analyze library procedures and thus assumed the worst case behavior at their call sites. Each node in our representation corresponds to a dag of multiple operations and may be viewed as a high-level node. Therefore, the ratio of the number of conditional nodes to the number of all nodes that are executable is higher than usually reported (last 2 columns). Note that the number of all nodes in column 5 includes unexecutable label nodes. The dynamic profile information was collected from the `ref` input set.

Behavior of statically detectable correlation. We first performed experiments to determine the amount of statically detectable correlation for paths restricted to a procedure and for paths that cross procedure boundaries. The top-left graph in Figure 9 depicts the number of conditionals that exhibit *some* correlation; that is, those whose outcome is known along some, but not necessarily all, incoming paths. Using the total number of conditionals in a program as a base, the graph shows for each program the percentage of conditionals that were analyzable using our implementation, the percentage of conditionals that were found correlated using intraprocedural analysis and the percentage that were found correlated using interprocedural analysis. The results show that at least twice as many correlated branches are detected using interprocedural analysis than by using intraprocedural analysis. The top-right graph presents the same information weighted by the execution count of each conditional, showing that correlation is detected on conditionals that execute frequently.

The bottom two graphs in Figure 9 show the number of conditionals that had *full* correlation. The outcome of such conditionals is known along all paths and hence they can be completely eliminated; however code duplication might be necessary if both TRUE and FALSE correlations are discovered. Here, the benefit of interprocedural analysis is even more evident. If only fully correlated conditionals were to be optimized, the programs would execute between 3% and 19% less conditionals, while intraprocedural analysis enables reduction of only up to 8%. The fact that more useful correlation exists when procedures are considered supports our hypothesis that we write procedures in an isolated fashion with repeated computation in the caller and callee.

The branch elimination optimizer replicates code to eliminate conditionals by creating separate paths. Since the amount of code duplication increases with the distance between the correlated branch and the source of the correlation, the extent of code duplication must be estimated before the interprocedural optimization is applied. Figure 10 plots the cost-benefit relationship for each correlated conditional. Each point in the graphs represents one conditional with a correlation. The x-coordinate of the point is the number of nodes that are created due to code duplication when the con-

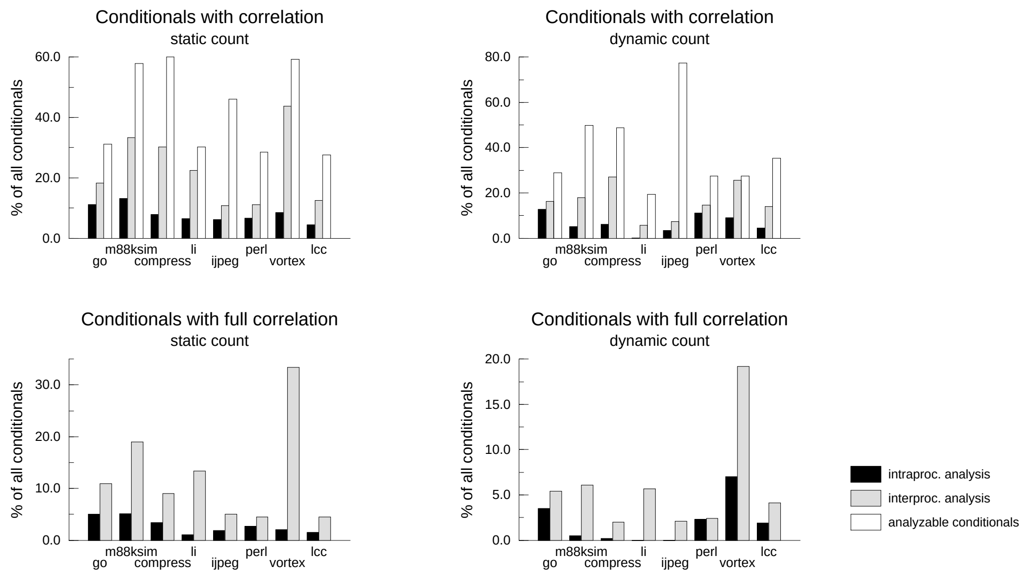


Figure 9: Characteristics of statically detectable branch correlation.

ditional is eliminated. The y-coordinate shows the amount of dynamic instances of conditionals that were avoided by the elimination of this conditional. A comparison with the intraprocedural results reveals that substantially more correlation is detected when procedures are considered, as the full-correlation graphs in Figure 9 suggest. But interprocedural correlation also requires more code duplication in many cases because the correlation may span a large part of the call graph. However, the amount of frequently executed correlated conditionals with low duplication needs, positioned in the upper-left quadrant, has increased with interprocedural analysis. These conditionals make ICBE more beneficial than intraprocedural elimination because with less code growth a higher reduction in eliminated banches can be achieved. We estimated the number of eliminated dynamic instances of each optimized conditional from the execution counts of the nodes where the analysis query was resolved.

Eliminated Branches. The goal of eliminating only conditionals causing reasonable code growth is easily achieved in our approach, for ICBE optimizes conditionals one by one, performing first the analysis and then the restructuring optimization for each conditional. The amount of code growth necessary to optimize the conditional is determined during the analysis phase, as described in Section 3.1. The restructuring phase is executed only if the number of new nodes that must be created is less than a predetermined limit. We optimized the benchmarks with various values of the per-conditional duplication limit. Each conditional was optimized only if the number of node duplicates required for its optimization did not increase N, where N ranged from 5 to 200. Figure 11 shows the amount of conditionals eliminated and the incurred code growth. Each point in a graph corresponds to one duplication limit value. Note the different y-ranges in the bottom row.

In this experiment, the analysis was terminated after 1000 node-query pairs were examined (see line 5 in Figure 4) even though not all queries were resolved. Since in each program there are numerous conditionals that test global variables, early termination of demand-driven analysis avoids far-reaching propagation of their queries and dramatically reduces the analysis time. The early termination is made possible by demand-driven analysis. All queries remaining after the analysis termination limit is reached are conservatively resolved to `UNDEF`. Terminating the analysis after 1000 nodes is sufficient to find correlated branches that require up to approximately 300 duplicated nodes. Even though not all correlation is detected with early termination, the missed opportunities are likely to be prohibitive due to high code duplication demands. Terminating the analysis early thus seems to be a practical improvement. However, note that for some values of the duplication limit, the interprocedural analysis may produce worse optimization for the `134.perl` benchmark than its intraprocedural counterpart. The reason is that the analysis termination limit was reached by examining the callees, missing the intraprocedural opportunity. This problem can be alleviated by experimentally increasing the analysis termination limit. Note that the results in Figure 9 and Figure 10 were computed with an infinite termination limit.

We can conclude that: 1) for a given code increase, ICBE can eliminate significantly more dynamic conditionals than its intraprocedural counterpart; 2) when more code growth can be tolerated, ICBE offers opportunities for additional branch elimination; and 3) the per-conditional limit on code duplication is an effective way to control overall code growth. A better heuristic for deciding whether to apply the optimization would also consider the amount of conditionals eliminated, as opposed to the incurred code growth alone, as was done in our experiments.

Analysis Cost. The analysis is the dominating component of ICBE's cost. The running time of the analysis that determined the results in Figure 11 is given in Table 2. In the column labeled overall time, we give the time spent by the compiler in parsing, building the internal program rep-

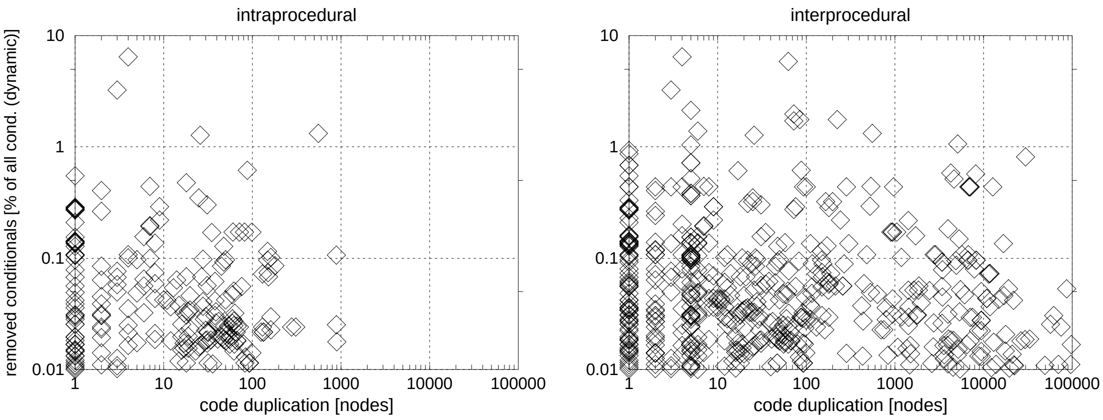


Figure 10: Contribution to branch removal vs. code duplication requirements for each correlated conditional.

resentation and performing the correlation analysis. The third second column gives the time to perform the analysis. The memory use listed in the analysis column indicates the amount needed to store the queries and summary nodes. We compare it to the memory required for the internal representation of the program, listed in the progrep column. The last two columns report the number of node-query pairs processed by the analysis, overall and per optimized conditional.

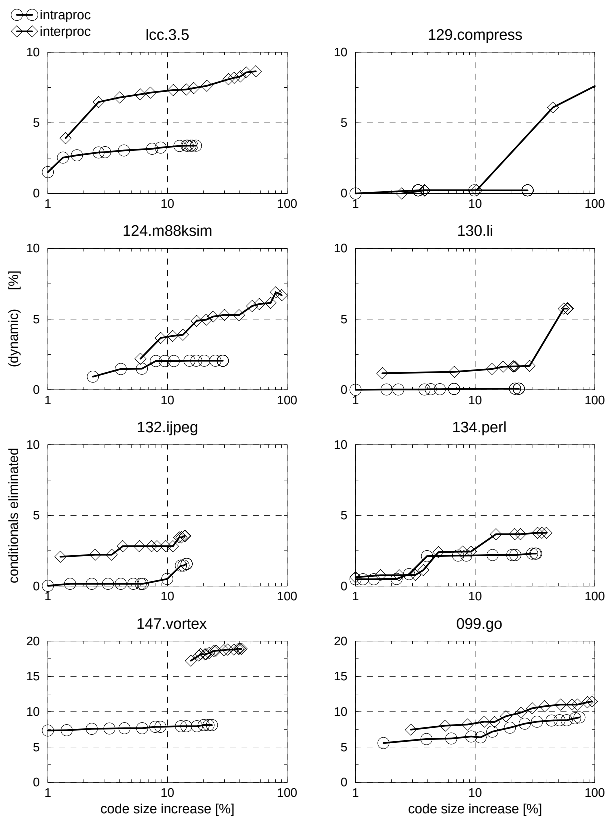


Figure 11: Reduction in executed conditional nodes vs. program code growth, for various values of the per-conditional code duplication limit.

5 Benefits of ICBE

The primary benefit of ICBE is the reduction in the instruction count (and the schedule length) through the elimination of correlated conditionals and the operations that compute their predicate. In this section we discuss how both the correlation analysis and the interprocedural restructuring can be applied in other areas of compiler optimization.

Procedure inlining. Most interprocedurally-visible opportunities for branch elimination can be exploited by inlining and subsequent application of intraprocedural elimination of conditionals [19]. However, without the knowledge of correlated paths in the call graph, the pre-pass inlining process must resort to exhaustive inlining, at least in the critical program regions. Short of folding all procedures into a single, flat procedure, there is no guarantee that all statements involved in a correlation will end up within the same procedure, which is necessary to remove the branch. Clearly, pre-pass inlining incurs large code growth.

Inlining becomes more practical when it is directed by our interprocedural correlation analysis. After correlation of a branch is detected, the procedures involved in the correlation can be merged by post-analysis inlining. Such a solution to ICBE may be desirable in an existing compiler where inlining and intraprocedural branch elimination are already supported. The code growth of post-analysis inlining may be further lowered by performing full ICBE (with interprocedural restructuring), followed by *partial inlining* [12], in which only frequently executed paths through the optimized procedure are inlined. However, inlining of recursive, virtual, or library procedures may not be feasible. In this case, our interprocedural restructuring can be applied to carry out ICBE.

Regardless of the exact ICBE scenario, the correlation analysis produces an upper bound on the code growth required to eliminate the conditional and, if profile information is available, provides also a profile-based estimate of the cost-effectiveness of the optimization before it is applied. The inlining algorithm in [2] inlines procedures one by one based on their execution rate until a code growth budget is exhausted. Our correlation analysis can be used in the inliner to give procedures that generate correlation a higher priority so that correlated branches can be removed after inlining [6, 8]. Our restructuring algorithm can be used to eliminate correlated branches after the code growth budget for inlining has been exhausted because its code growth demands are smaller than those of inlining. Richardson and Ganapathi [23] observed that the benefit of inlining comes mainly from eliminated procedure call overhead. Our analysis is able to identify procedures whose inlining will create intraprocedural optimization opportunities for branch removal.

Dynamic dispatching of virtual procedures. Object-oriented languages complicate interprocedural compilation because call sites invoking member procedures of polymorphic types may transfer control to one of many procedures, depending upon the concrete type of the receiver object. Since such call sites require expensive dynamic dispatching, methods for their elimination through concrete type inference have been developed [1, 20, 21]. In these methods, demand-driven interprocedural analysis determines for each call site the set of "reaching concrete types." Subsequent program restructuring separates out paths and clones procedures with the goal of creating call sites reached by a single type of the receiver. There is an analogy between concrete type inference and our work in that both methods compute at optimizable nodes the set-union of all optimization opportunities and enable optimization by making the opportunities unique through path separation. While ICBE collects values of variables that determine branch outcomes, type inference is interested in the types of receiver objects. With respect to the restructuring algorithms, however, our transformation is more powerful than cloning because exit splitting is able to separate out paths that cross the exit node, which cloning cannot achieve. The following paragraph describes how entry/exit splitting is performed at dynamic dispatch call sites. Our restructuring can prove valuable for object-oriented languages because the cost of passing additional return addresses is small compared to that of a dynamic dispatch.

While concrete type analysis is very successful in enabling inlining at virtual call sites, some call sites will still require dynamic dispatch. These call sites are, however, amenable to ICBE. Each procedure that may be invoked from a virtual call site can be independently analyzed and optimized by entry/exit splitting. Optimizing in turn each possible callee will provide the cumulative effect of entry/exit splitting in each callee with respect to the original call site. What results is a set of multiple call sites, each invoking some entry of each callee and returning to multiple points in the caller. If any of the callees was left unoptimized, then all call sites invoke the original entry and the procedure always returns to the original return point in the caller. ICBE thus allows both optimized and unoptimized procedures to be called from a single call site.

Fine-grain computer architectures. The elimination of conditional branches is especially important for wide-issue superscalar and VLIW architectures, in which instructions are pre-fetched and executed speculatively across conditional branches based on predictions of their outcomes. With increasing processor parallelism, branch density in the stream of instructions is becoming critical because expensive mechanisms are required to predict and issue multiple conditional branches in a single cycle [13]. Our experiments have shown that between 3% and 18% of executed conditionals can be eliminated by ICBE, reducing branch density.

A mispredicted branch stalls the processor for many cycles and pollutes the instruction cache. Research in correlation-based hardware branch prediction [25] shows that unpredictable branches exhibit correlation with earlier branches. Some unpredictable branches can arguably be eliminated by ICBE. Consider, for example, a procedure that removes an element from a linked list. When the average list length is low, the conditional that tests for an empty list is unpredictable. Nevertheless, the test is correlated with the conditional that tests the return value in the caller. Optimization of unpredictable branches has an especially high payoff.

ICBE can also be used to improve the effectiveness of software pipelining [17, 22] by reducing the number of conditionals and other statements in the loop body, as illustrated by the example in Figure 2. Elimination of branches can significantly speed-up the loop schedule when conditionals that form recurrent cycles of control dependencies are eliminated. Branches testing a flag whose value is assigned within the loop are examples of such conditionals.

Assisting hardware branch prediction. Run-time prediction schemes have been proposed that predict the outcome of a branch using its correlation with the last k branches [24]. Since the exact source of the correlation is not known, all k outcomes are maintained and used for prediction, slowing down the learning process of the predictor. If the correlation is statically detectable, our analysis can provide the prediction hardware with directions about which recent branch(es) should be used for prediction.

Library procedures. Even when it is not possible to compile the library procedures together with the application program, we can take advantage of correlation that crosses the application-library boundary. The library procedures can be pre-split by optimization with respect to characteristic application programs and the summary nodes describing the resulting entry/exit splitting can be conveniently stored with the library interface for later lookup during the optimization of the user program. For example, a separate exit from malloc would exist that would be taken when the return value is NULL. Since a large portion of correlation exists across calls to the same or related library procedures, the characteristic program may be as small as the one in Figure 1. The original unoptimized procedure entry must be maintained for library procedures. When this entry is invoked, all procedure exits return control to the standard return address so that compilers without ICBE can also use the library.

Interprocedural optimizations. Because path separation and entry/exit splitting eliminate control flow merge points, conservative merging of data flow information at procedure boundaries is reduced. As a result, other optimizations, such as procedure cloning, partial redundancy and dead code elimination, may be more effective following interprocedural restructuring. The ICBE optimization can be used to optimize array bounds checks [15, 11] which typically exhibit correlation. Finally, branch elimination can be used as a component of aggressive program transformations, such as slicing-based partial dead code elimination [4].

6 Acknowledgement

We want to thank Mooly Sagiv for his help with preparing the final version of this paper. The comments of the anonymous referees aided the presentation of this paper.

References

[1] Ole Agesen and Urs Hölzle. Type feedback vs. type inference: A comparison of optimization techniques for object-oriented languages. In *OOPSLA'95 Conference Proceedings*, pages 91–107, Austin, TX, 1995.

[2] Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. *SIGPLAN Notices*, 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *29th Annual IEEE/ACM International Symposium on Microarchitecture*, Paris, France, 1996.

[4] Rastislav Bodík and Rajiv Gupta. Partial dead code elimination using slicing transformations. *SIGPLAN Notices*, 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[5] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[6] Paul R. Carini. Automatic inlining. Technical Report IBM research Report RC-20286, IBM T.J. Watson Research Center, November 1995.

[7] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. *SIGPLAN Notices*, 23(7):57–66, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[8] Jack. W. Davidson and Anne. M. Holler. A study of a C function inliner. *Software, Practice and Experience*, 18(8):775–790, August 1988.

[9] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, California, January 1995.

[10] Christopher W. Fraser and David R. Hanson. *A retargetable C compiler: design and implementation.* Benjamin/Cummings, 1995. ISBN 0-8053-1670-1.

[11] Rajiv Gupta. A fresh look at optimizing array bound checking. *SIGPLAN Notices*, 25(6):272–282, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[12] Richard E. Hank, Wen-mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: An introduction and motivation. In *28th Annual IEEE/ACM International Symposium on Microarchitecture*, Ann Arbor, Michigan, November 1995.

[13] William Johnson. *Superscalar Microprocessor Design.* Prentice Hall, 1991. ISBN 0-13-875634-1.

[14] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[15] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. *SIGPLAN Notices*, 30(6):270–278, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[16] Andreas Krall. Improving semi-static branch prediction by code replication. *SIGPLAN Notices*, 29(6):97–106, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[17] Daniel M. Lavery and Wen-mei W. Hwu. Modulo scheduling of loops in control-intensive non-numeric programs. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 126–137, Paris, France, December 2–4, 1996.

[18] Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. *SIGPLAN Notices*, 27(7):322–330, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[19] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. *SIGPLAN Notices*, 30(6):56–66, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[20] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. *OOPSLA'94, ACM SIGPLAN Notices*, 29(10):324–335, 1994.

[21] John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Eighth Annual Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, volume 1033, pages 566–580, Columbus, Ohio, August 1995.

[22] B. Ramakrishna Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[23] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, 1989.

[24] Stuart Sechrest, Chih-Chieh Lee, and Trevor Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 22–32, Philadelphia, Pennsylvania, May 22–24, 1996.

[25] Cliff Young, Nicolas Gloy, and Michael D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Intl. Symposium on Computer Architecture*, Italy, 1995.