Comprehensive Path-sensitive Data-flow Analysis

Aditya Thakur
Supercomputer Education and Research Centre
aditya@hpc.serc.iisc.ernet.in

R. Govindarajan
Supercomputer Education and Research Centre
govind@serc.iisc.ernet.in

Abstract

Data-flow analysis is an integral part of any aggressive optimizing compiler. We propose a framework for improving the precision of data-flow analysis in the presence of complex control-flow. We initially perform data-flow analysis to determine those control-flow merges which cause the loss in data-flow analysis precision. The control-flow graph of the program is then restructured such that performing data-flow analysis on the resulting restructured graph gives more precise results. The proposed framework is both simple, involving the familiar notion of product automata, and also general, since it is applicable to any forward or backward data-flow analysis. Apart from proving that our restructuring process is correct, we also show that restructuring is profitable in that it necessarily leads to more optimization opportunities. Furthermore, the framework handles the trade-off between the increase in data-flow precision and the code size increase inherent in the restructuring. We show that determining an optimal restructuring is NP-hard, and propose and evaluate a greedy strategy. The framework has been implemented in the Scale research compiler, and instantiated for the specific problem of Constant Propagation.

1. Introduction

Compiler optimization is tough, as epitomized by Proebsting's Law [18]. But developers still want that extra 5-15% improvement in the running times of their applications, and the compiler optimizations are a safer alternative to manual optimizations carried out by developers which might introduce errors [19].

Data-flow analysis [1] is an integral part of any aggressive optimizing compiler. Code optimizations such as constant propagation, dead-code elimination, common subexpression elimination, to name a few, are made possible due to data-flow analysis. Imprecision in data-flow analysis leads to a reduction in optimization opportunities. The loss of data-flow precision occurs due to the approximation or merging of differing data-flow facts along incoming edges of a control-flow merge. In this paper, we present a new framework to overcome this imprecision. We initially perform data-flow analysis to determine those control-flow merges that cause the loss in data-flow analysis precision, which we call *Destructive Merges*. The control-flow graph (CFG) of the program is then restructured so that performing data-flow analysis on the resulting restructured graph gives more precise results, and effectively eliminates the destructive merges. This leads to more optimization opportunities in the resulting CFG.

The framework presented in this paper is simple and clean, and uses the familiar notion of product automata [12] to carry out the restructuring transformation. Further, the framework is general since it can be instantiated to any forward or backward data-flow analysis. This clean formulation allows us to show that the restructuring is correct, in the sense that the original and restructured CFGs are equivalent. Also, we prove that the restructuring is profitable in that it necessarily leads to more optimization opportunities in the restructured CFG. The restructuring inherently entails an increase in code size due to code duplication i.e., the increase in precision comes at the cost of an increase in code size. Furthermore, we show that determining an optimal restructuring is NP-Hard and, hence, propose a greedy heuristic for restructuring. Our framework explicitly handles the trade-off between precision and code size increase and also makes use of low-cost basic-block profile information.

We have implemented the proposed framework in the Scale research compiler [21] and instantiate it with the specific problem of Constant Propagation [1]. We compare our technique with the Wegman-Zadeck conditional constant propagation algorithm [25](*Base*) and with a purely path profile-guided restructuring technique [2](*HPG*). On the SPECINT 2000 benchmark suite [22], our technique exposes, on an average, *3.5 times more dynamic constants* as compared to the HPG technique. Further, we observe an average speedup of *4%* in the running times over Base and *2%* over the HPG technique.

The rest of the paper is structured as follows. We first illustrate our technique using a simple example. We then go

To appear in the International Symposium on Code Generation and Optimization (CGO'08) April 6–9, Boston, Massachusetts.

on to present the general framework. In Section 3, we describe the method to determine which nodes of the CFG are to be duplicated for restructuring. Section 4 discusses the actual restructuring algorithm. In this section, we demonstrate the correctness and profitability of our approach. We then discuss the trade-off between the increase in precision and increase in code-size in Section 5. Section 6 discusses the related work and compares our technique to the HPG technique [2]. This is followed by the experimental results in Section 7, after which we conclude.

2. Overview

We illustrate our technique using the problem of Constant Propagation [1]. Consider the program P in Figure 1(a). There are two definitions of variable x at nodes B and C, while node G contains a *use* of variable x. The remaining nodes do not define or use variables x or y. We see that node G cannot be optimized since the *use* of variable x cannot be replaced by a constant. Our technique would restructure the program P by duplicating node D, E, F and G to obtain the Split Graph P', shown in Figure 1(b). After this restructuring, the *uses* of variable x at nodes $G1$ and $G2$ can now be replaced with constants 1 and 2 respectively.

In order to understand how to obtain the program P', we first look at control-flow merge D in program P. At node D, the incoming data-flow facts $\{\mathtt{x} = 1\}$ and $\{\mathtt{x} = 2\}$ are merged to get the data-flow fact $\{\mathtt{x} = \mathtt{nc}\}$, where nc stands for not-constant. We call such a control-flow merge where data-flow approximation leads to loss of precision a *Destructive Merge*. Due to this loss of precision at node D, only the data-flow fact $\{\mathtt{x} = \mathtt{nc}\}$ reaches node G, and hence, node G cannot be optimized. Further, we also notice that if (somehow) the data-flow fact $\{\mathtt{x} = 1\}$ were to hold at node D, then the *use* of variable x at node G can be replaced by the constant 1. Thus, we see that in order to optimize node G, it is *useful* for data-flow fact $\{\mathtt{x} = 1\}$ to hold at node D. This also holds for the data-flow fact $\{\mathtt{x} = 2\}$. On the other hand, it is not useful for the data-flow fact $\{\mathtt{x} = \mathtt{nc}\}$ to hold at node D. Hence, in program P' in Figure 1(b) node D is duplicated and at the copies $D1$ and $D2$ of D, the data-flow facts $\{\mathtt{x} = 1\}$ and $\{\mathtt{x} = 2\}$ hold respectively. It can be seen that nodes E, F and G also need to be duplicated to preserve the data-flow facts $\{\mathtt{x} = 1\}$ or $\{\mathtt{x} = 2\}$ at nodes $G1$ and $G2$ respectively. Further, nodes H, I and J cannot be optimized even if the data-flow facts $\{\mathtt{x} = 1\}$ or $\{\mathtt{x} = 2\}$ hold at node D. Thus, these nodes are not duplicated in Program P' in Figure 1(b). We say that these nodes are not *influenced* by the destructive merge at node D. The nodes D, E, F, and G are called the *Region of Influence* for the destructive merge D and are exactly those nodes which have multiple copies in program P'.

In program P, there are four paths which reach node G. Along the two paths going through edge (B, D), data-flow fact $\{\mathtt{x} = 1\}$ holds. Similarly, along the two paths going through edge (C, D), data-flow fact $\{\mathtt{x} = 2\}$ holds. Hence, program P is restructured in such way these two sets of paths are separated and the differing data-flow facts along them do not merge in program P'. We use the concept of product automaton [12] to carry out this restructuring. The increase in the number of *uses* which can be replaced by constants (the benefits of our approach) comes at the cost of increase in code size.

Though not illustrated in this pedantic example, our restructuring technique is also applicable when there are multiple destructive merges which we wish to eliminate and control-flow structures such as loops.

3. What to Split

The reader is assumed to be familiar with preliminary data-flow analysis [1, 15, 2]. Central to our approach, is the notion of a destructive merge. Intuitively, a destructive merge is a control-flow merge where data-flow analysis loses precision. The notion of precision is governed by the partial-order of the lattice of the data-flow problem.

Definition 1. (DESTRUCTIVE MERGE) *For a given forward data-flow problem and its corresponding solution, a control-flow merge m is said to be a* Destructive Merge *if* $\exists p \in \mathtt{pred}(m)$ *s.t.* $\mathtt{in\_dff}(m) \prec \mathtt{out\_dff}(p)$,
where $\mathtt{pred}(n)$ *denotes the set of control-flow predecessors of a node n in the CFG,* $\mathtt{in\_dff}(n)$ *and* $\mathtt{out\_dff}(n)$ *denotes the data-flow facts holding true at the beginning and end of a node n in the fixed-point solution repectively, and $\prec$ is the partial order of the lattice in the data-flow problem.*

Definition 2. (DESTROYED DATA-FLOW FACTS) *Given a destructive merge m, a data-flow fact d is said to be a* Destroyed Data-flow Fact *, i.e.* $d \in \mathtt{destroyed\_dff}(m)$, *iff* $d \in \mathtt{out\_dff}(p)$, *where node* $p \in \mathtt{pred}(m)$, *and* $\mathtt{in\_dff}(m) \prec d$.

Example. For the problem of Constant Propagation [1], Figure 2 illustrates the different scenarios possible at a control-flow merge. Nodes $D1$ and $D2$ are destructive, while nodes $N1$ and $N2$ are not. More specifically, in Figure 1 node D is a destructive merge since $\mathtt{in\_dff}(D) = \{\bot\}$, while $\mathtt{out\_dff}(B) = \{\mathtt{x} = 1\}$. Further, $\mathtt{destroyed\_dff}(m) = \{\mathtt{x} = 1, \mathtt{x} = 2\}$. □

Our restructuring targets such destructive merges. In the rest of this section, we determine which nodes are influenced by the loss of precision lost at a destructive merge and which nodes need to be duplicated in order to eliminate the effects of a destructive merge.

3.1 Single Destructive Merge

The following definitions assume that we are given a CFG and a data-flow problem and a corresponding solution.

Definition 3. (USEFUL DATA-FLOW FACTS) *For a given node n, a data-flow fact is said to be a* Useful Data-flow

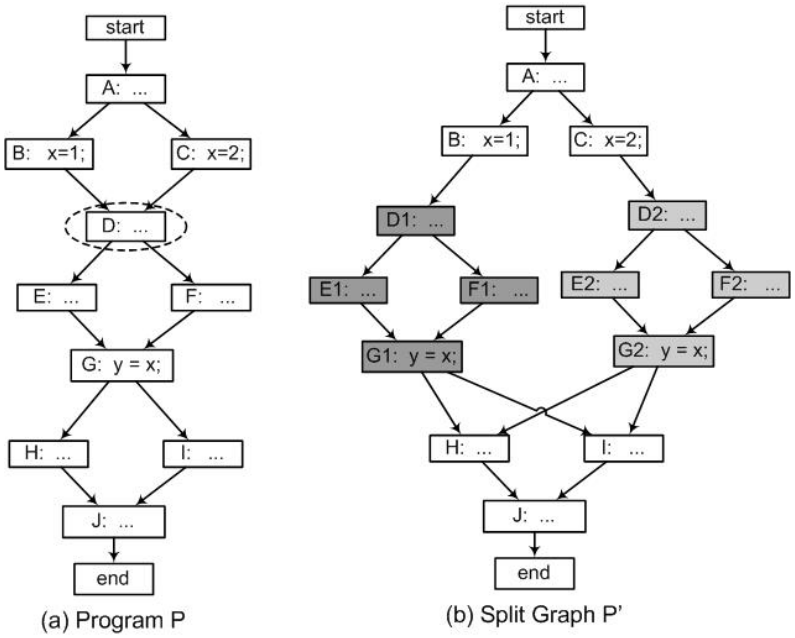

Figure 1: (a) In program P, node D is a destructive merge. *Use* of variable x cannot be replaced by a constant at node G. (b) In Split Graph P', *uses* of variable x at $G1$ and $G2$ can now be replaced by constants 1 and 2 respectively.

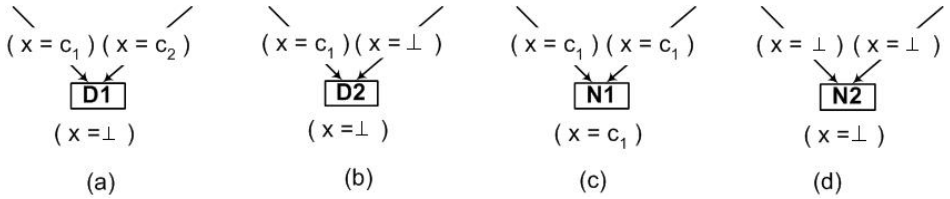

Figure 2: For the problem of Constant Propagation, (a)-(b) show two destructive merges and (c)-(d) illustrate two control-flow merges which are not destructive. c_1 and c_2 are two differing constants, while $\perp$ represents not-constant.

Fact, *i.e.* $d \in \texttt{useful\_dff}(n)$, *iff data-flow fact* d *holding true at node* n *implies that the node* n *can be optimized.*

The above definition implicitly captures the interaction between a data-flow analysis and the client optimization. It allows us to abstract out the details of this relation by providing an oracle which knows which data-flow facts enable the optimization for a particular program statement. We can now generalise this notion of useful data-flow facts.

Definition 4. (USEFUL DATA-FLOW FACTS) *Given nodes* m *and* n*, a data-flow fact is said to be* Useful Data-flow Fact *for node* n *at node* m*, i.e.* $d \in \texttt{useful\_dff}(m, n)$*, iff data-flow fact* d *holding true at node* m *implies that the node* n *can be optimized.*

It is easy to see that Definition 3 is a special case of Definition 4 with node m being the same node n.

Example. Consider the program in Figure 1.

$\texttt{useful\_dff}(G) = \{\ldots, \texttt{x} = -1, \texttt{x} = 0, \texttt{x} = 1, \texttt{x} = 2, \ldots\}$,

$\texttt{useful\_dff}(D, G) = \{\ldots, \texttt{x} = -1, \texttt{x} = 0, \texttt{x} = 1, \texttt{x} = 2, \ldots\}$.

This is due to the fact that if the value of variable x were to be a constant at node D, then it would remain a constant at node G. This would enable us to optimize node G by replacing the *use* of variable x with a constant. □

Definition 5. (INFLUENCED NODES) *Given a destructive merge* m*, we say a node* n *is* influenced *by the destructive merge* m*, i.e.* $n \in \texttt{influenced\_nodes}(m)$*, iff*

$$\texttt{destroyed\_dff}(m) \cap \texttt{useful\_dff}(m, n) \neq \emptyset.$$

Definition 6. (REVIVAL DATA-FLOW FACTS) *Given a destructive merge* m*, a data-flow fact is said to be a* Revival Data-flow Fact*, i.e.* $d \in \texttt{revival\_dff}(m)$*, iff there exists a node* $n \in \texttt{influenced\_nodes}(m)$ *such that*

$$d \in \texttt{destroyed\_dff}(m) \cap \texttt{useful\_dff}(m, n).$$

In other words, a node n is *influenced* by a destructive merge m when, if one of the data-flow facts d which are destroyed by the node m were to hold true at node m, then node

n could be optimized. Intuitively, a node is *influenced* by a destructive merge if eliminating the destructive merge can enable the optimization of the node. Further, these are the only nodes which can be optimized if the destructive merge is eliminated. The Revival data-flow facts denote those data-flow facts which if they would hold true at the destructive merge m would enable the optimization of some influenced node.

Example. Consider Figure 1,

$$\texttt{influenced\_nodes}(m) = \{G\},$$

$$\texttt{destroyed\_dff}(m) = \{\texttt{x} = 1, \texttt{x} = 2\},$$

$$\texttt{useful\_dff}(D, G) = \{\ldots, \texttt{x} = -1, \texttt{x} = 0, \texttt{x} = 1, \texttt{x} = 2, \ldots\},$$

$$\texttt{revival\_dff}(m) = \{\texttt{x} = 1, \texttt{x} = 2\}.$$

□

Since the influenced nodes are the only nodes which can be optimized by eliminating the destructive merge, we would like to determine the largest such set of nodes. Unfortunately, for the specific problem of Constant Propagation determining whether a node n belongs to $\texttt{influenced\_nodes}(m)$ for a destructive merge m is undecidable in general.

Theorem 1. *For the problem of Constant Propagation, given a node n and destructive merge m determining if $n \in \texttt{influenced\_nodes}(m)$ is undecidable.*

Proof. (Sketch) The proof follows from the undecidability of constant propagation for programs with loops even if all branches are considered to be non-deterministic [16, 10, 20]. The reduction is based on the Post correspondence problem [12]. We show that the node $n \in \texttt{influenced\_nodes}(m)$ iff the Post correspondence problem is not solvable. □

As we see in the Section 4, our restructuring is guaranteed to improve data-flow precision and optimize the nodes which are *influenced* by the destructive merge. The above theorem implies that we cannot determine all nodes which are *influenced* by a destructive merge. If we include a node which is not actually influenced by the destructive merge in $\texttt{influenced\_nodes}(m)$, then the restructuring will not enable any optimizations. Thus, the corresponding increase in code size will be unnecessary. Hence, in practice, we under-approximate the set of *influenced* nodes. In particular, for Constant Propagation, if the data-flow fact for variable x is destroyed at merge node m, then the *uses* which are reachable only along paths from node m which do not contain any definitions of variable x are *influenced* by the destructive merge m. This set is an under-approximation and there might be nodes which are influenced by the destructive merge m and not be included in the set $\texttt{influenced\_nodes}(m)$ which is actually determined. Note, this under-approximation is orthogonal to our overall approach.

Having determined which nodes are influenced, we now find those nodes which need to be duplicated in order to eliminate the effects of a destructive merge.

Definition 7. (REGION OF INFLUENCE) *Given a destructive merge m, a node n is said be in the* Region of Influence, *i.e. $n \in \texttt{RoI}(m)$, iff $n \in \texttt{reachable\_nodes}(m)$ and there exists a node $u \in \texttt{influenced\_nodes}(m)$ and $u \in \texttt{reachable\_nodes}(n)$, where $\texttt{reachable\_nodes}(l)$ denotes the set of nodes reachable from node l in the CFG.*

Figure 3 shows a schematic diagram showing $\texttt{RoI}(m)$ for the destructive merge m.

Having determined the set of influenced nodes, the Region of Influence consists of those nodes which are sufficient and necessary to be duplicated in order to improve precision and optimize the influenced nodes.

Example. In Figure 1, node D is a destructive merge.

$$\texttt{influenced\_nodes}(D) = \{G\},$$

$$\texttt{reachable\_nodes}(D) = \{D, E, F, G, H, I, J\},$$

$$G \in \texttt{reachable\_nodes}(D),\ G \in \texttt{reachable\_nodes}(E),$$

$$G \in \texttt{reachable\_nodes}(F),\ G \in \texttt{reachable\_nodes}(G),$$

$$\texttt{RoI}(m) = \{D, E, F, G\}.$$

□

3.2 Multiple Destructive Merges

In the previous section, we dealt with the situation where we were given a single destructive merge to eliminate. In practice, we will have a set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$ which are to be eliminated. In this section, we extend the concepts of the previous section to handle multiple destructive merges.

Definition 8. (INFLUENCED NODES) *Given a set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$, the set of* Influenced Nodes *is defined as*

$$\texttt{influenced\_nodes}(\mathcal{M}) = \bigcup_{m \in \mathcal{M}} \texttt{influenced\_nodes}(m)$$

Definition 9. (REGION OF INFLUENCE) *Given a set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$, the* Region of Influence *corresponding to $\mathcal{M}$ is defined as*

$$\texttt{RoI}(\mathcal{M}) = \bigcup_{m \in \mathcal{M}} \texttt{RoI}(m)$$

Extending the definitions to handle multiple destructive merges is straight forward. A node n belongs to $\texttt{influenced\_nodes}(\mathcal{M})$ iff it is *influenced* by at least one destructive merge $m \in \mathcal{M}$. Similarly, a node n belongs to $\texttt{RoI}(\mathcal{M})$ iff it belongs to the *Region of Influence* of at least one destructive merge $m \in \mathcal{M}$.

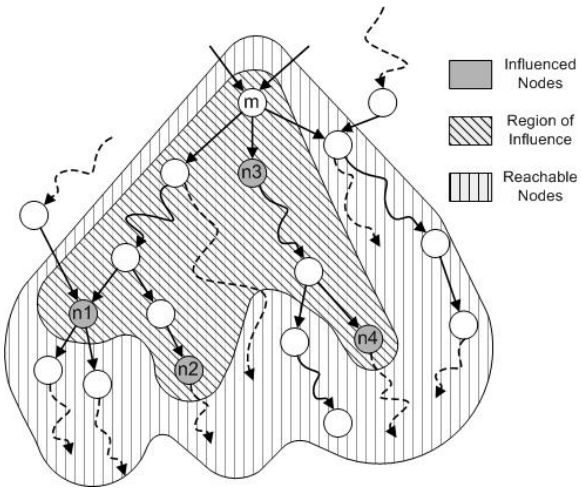

Figure 3: Schematic diagram showing the Region of Influence for a destructive merge m.

4. How to Split

In this section, we discuss the actual restructuring algorithm. There are two main constraints which such a restructuring transformation should satisfy.

- **Equivalence.** The original and transformed programs should be equivalent.
- **Profitability.** The transformation should guarantee that the improvement in data-flow precision leads to optimization opportunities.

We explain a clean and simple technique which makes use of familiar notions of product automaton [12], and show that indeed our restructuring satisfies both the constraints. As before, we first restrict ourselves to a single destructive merge, and then generalise this to multiple destructive merges.

4.1 Single Destructive Merge

Corresponding to each destructive merge, we construct a *Split Automaton* which will then be used to restructure the CFG and eliminate the effects of the destructive merge.

Definition 10. (KILL EDGES) *Given a Region of Influence for a destructive merge m, an edge $e = (u, v)$ is a* Kill Edge, *i.e. $e \in \texttt{kill\_edges}(m)$, iff $u \in \texttt{RoI}(m)$ and $v \notin \texttt{RoI}(m)$.*

In other words, Kill Edges are those edges whose source node is in the Region of Influence and target node is not in the Region of Influence for a destructive merge m.

Definition 11. (REVIVAL EDGES) *Given a Region of Influence for a destructive merge m, an edge $e = (u, m)$ is said to be a* Revival Edge, *i.e. $e \in \texttt{revival\_edges}(m)$, iff*

$$\texttt{out\_dff}(u) \in \texttt{revival\_dff}(m).$$

In other words, Revival Edges are those incoming edges of the destructive merge which correspond to Revival Data-flow facts. Further, let $d_1, d_2, \ldots d_k$ be the k *distinct* Revival Data-flow facts for destructive merge m. We can partition the incoming edges of destructive merge m into $k+1$ equivalence classes $R_0, R_1, \ldots, R_k$. An edge $(u, m) \in R_i, 1 \leq i \leq k$ if $\texttt{out\_dff}(u) = d_i$, and $(u, m) \in R_0$ otherwise.

Figure 4(a) shows a schematic diagram illustrating the above concepts.

Example. In Figure 1,

$$\texttt{out\_dff}(B) = \{\texttt{x} = 1\},\ \texttt{out\_dff}(C) = \{\texttt{x} = 2\},$$

$$\texttt{revival\_dff}(m) = \{\texttt{x} = 1, \texttt{x} = 2\},$$

$$\texttt{revival\_edges}(m) = \{(B, D), (C, D)\},$$

$$\texttt{RoI}(m) = \{D, E, F, G\},$$

$$\texttt{kill\_edges}(m) = \{(G, H), (G, I)\}.$$

□

Armed with the above concepts we are now ready to define the Split Automaton.

Definition 12. (SPLIT AUTOMATON) *The* Split Automaton A_m *corresponding to a destructive merge m is a finite-state automaton defined as:*

- *the input alphabet $\Sigma = E$, the set of all edges in the CFG.*
- *a set of $k+1$ states $Q = \{s_0, s_1, s_2, \ldots, s_k\}$.*
- *$s_0 \in Q$ is the initial and accepting state.*
- *the transition function $\delta : Q \times \Sigma \rightarrow Q$ defined as*
  $(s_i, e) \rightarrow s_j, e \in R_j$ (Revival Transitions)
  $(s_i, e) \rightarrow s_0, e \in \texttt{kill\_edges}(m)$ (Kill Transitions)
  $(s_i, e) \rightarrow s_i$, *otherwise* (No Transition)

Intuitively, state $s_i, 1 \leq i \leq k$, corresponds to the Revival Data-flow fact d_i and whenever an edge $e \in R_i$ is seen, the Split Automaton makes a transition to state s_i. We call

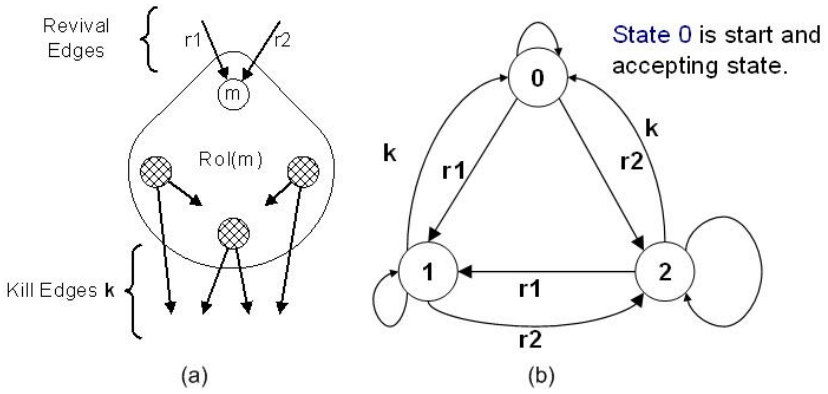

Figure 4: (a) Schematic diagram of the Region of Influence for a destructive merge m, showing the Revival and Kill edges. (b) The corresponding Split Automaton A_m.

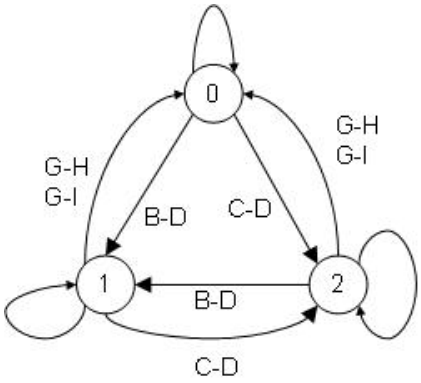

Figure 5: The Split Automaton A_D corresponding to destructive merge D in Figure 1.

this the Revival Transitions. Further, whenever a Kill edge is seen, the Split Automaton transitions to state s_0. We call these the Kill Transitions. In all other cases, the automaton remains in the same state, and makes no transitions.

Example. Figure 4(b) shows a schematic the Split Automaton corresponding to destructive merge m in Figure 4(a). Figure 5 shows the Split Automaton corresponding to destructive merge D in Figure 1. □

A CFG can be viewed as a finite-state automaton with nodes of the CFG corresponding to the state of the automaton, and the edges defining the alphabet as well as the transition function. The entry and exit node of the CFG correspond to the start and accepting states in the automaton respectively.

Definition 13. (SPLIT GRAPH) *Given a CFG P and and a Split Automaton A_m, we define the* Split Graph *S to be $S = P \times A_m$, where $\times$ is the product automaton operator [12].*

Each node n in the Region of Influence of node m in P will have multiple copies n_i in the Split Graph S corresponding to the states in the Split Automaton.

Example. The Split Graph P' in Figure 1(b) is obtained by performing the product of the CFG P in Figure 1(a) and the Split Automaton A_D in Figure 5. □

Theorem 2. *The Split Graph S is equivalent to the original CFG P, where $S = P \times A_m$.*

Proof. (Sketch) Viewing CFG as finite-state automaton, we say that the two CFGs G_1 and G_2 are *equivalent* if the languages accepted by G_1 is equal to language accepted G_2, i.e., $L(G_1) = L(G_2)$. Such a notion of equivalence suffices since the restructuring only duplicates nodes and does not change the semantics of the individual nodes. Since S is the product of P and A_m, we can say that $L(S) = L(P) \cap L(A_m)$ [12]. Futher, we can show that $L(P) \subset L(A_m)$. Intuitively, A_m is a control-flow abstraction of P and accepts more words. We ommit the detailed proof for lack of space. □

Thus, using simpe concepts from automaton theory [12], we are able to prove the correctness of our restructuring transformation. Next, we prove the profitability of the restructuring.

Theorem 3. *If node $n \in \texttt{influenced\_nodes}(m)$ in CFG P, then in the Split Graph S,node $n_i, i \neq 0$ can be optimized, where $S = P \times A_m$.*

Proof. (Sketch) Let d_i be a Revival Data-flow fact and R_i be the corresponding equivalence class of edges. Since $n \in \texttt{influenced\_nodes}(m)$, by definition, if d_i (somehow) holds true at node m, then node n can be optimized in CFG P. The theorem proceeds in two steps. We first show that data-flow fact d_i holds true at the start of node m_i in the Split Graph S. Then, we show that this data-flow fact does not merge with other differing data-flow facts. This is due to the nature of Revival Transitions. □

4.2 Multiple Destructive Merges

We now consider eliminating multiple destructive merges. Consider the set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$ which are to be eliminated. Let $A_1, A_2, \ldots, A_k$ be the corresponding Split Automata.

Definition 14. (SPLIT GRAPH) *Given a CFG P and and a set of Split Automata $A_1, A_2, \ldots, A_k$, we define the* Split Graph *S to be $S = P \times A_1 \times A_2 \times \ldots \times A_k$, where $\times$ is the product automaton operator [12].*

The correctness and profitability results also hold in this general setting. It is interesting to note that we placed no restriction of the nature of the CFG. Thus, our restructuring can handle programs with complex loop structures.

5. Which to Split

The discussion till now has not dealt with the trade-off between the data-flow precision achieved and the increase in the code size due to the restructuring inherent to this approach. We have seen that the benefit of eliminating a destructive merge m is related to the number of nodes *influenced* by node m. Also, the size of the *Region of Influence* of the destructive merge determines the resulting cost in terms of increase code size. Thus, this trade-off depends directly on the set of destructive merges we choose to eliminate to form the Split Graph. In this section, we will prove that the problem of picking the best set of destructive merges which maximize the benefit in terms of data-flow precision, for a given cost in terms of code size, is NP-Hard. We then proceed to describe certain heuristics for the same.

Definition 15. *The problem* SPLIT *is defined by the triple $(P, \mathcal{A}, C)$ where:*

- *P is a program,*
- *$\mathcal{A}$ is a set of split automata corresponding to the various destructive merges, and*
- *C is maximum increase in code size that is permitted.*

A solution to SPLIT *is a subset B of $\mathcal{A}$ such that applying B to the program P does not increase the code-size by more than C and which maximizes the number of influenced nodes which can be optimized in the resulting program P'.*

Theorem 4. SPLIT *is NP-Hard.*

Proof. We shall reduce *KNAPSACK* [17] to it. We are given a set $\mathcal{I}$ of n items, each item i having a specific weight w_i and a profit p_i. The goal of *KNAPSACK* is to pick a subset $J \subseteq \mathcal{I}$ of the items so as to maximize the total profit subject to the condition that the total weight is less than a specified weight W.

We will only give a brief outline of the proof due to lack of space. Intuitively, in our reduction, picking an item i in *KNAPSACK* will correspond to selecting an split automaton in the solution of *SPLIT*. Thus, we construct a program P, in which for each item i in *KNAPSACK* there exists a destructive merge D_i and a split automaton a_i so that $|\texttt{influenced\_nodes}(D_i)| = p_i$ and $|\texttt{RoI}(D_i)| = w_i$. Further, the profits and costs of items in *KNAPSACK* are independent of each other i.e. the cost of picking an item i does not depend on whether item j has been placed in the knapsack. To ensure this the constructed regions are such that $\texttt{RoI}(D_i) \cap \texttt{RoI}(D_j) = \emptyset, i \neq j$. The constraint of the total weight W of the knapsack is mapped to the increase in code size C which we are allowed in *SPLIT*. It can be shown that split automaton a_i is in the optimal solution of *SPLIT* if and only if item i is selected in the optimal solution of *KNAPSACK*. □

It is interesting to note that this hardness result does not rely on the complexity of the underlying data-flow analysis used, since we are already given the set of influenced nodes and the Region of Influence for each destructive merge. Further, the program P does not even contain any loops, and is acyclic. Thus, restricting the problem *SPLIT* any further does not result in a computationally tractable problem.

This lead us to device an aggressive greedy heuristic to solve this problem. Our approach is based on estimating the benefit obtained and cost incurred by eliminating a destructive merge. In the absence of profile information, we define the *fitness* of a destructive merge m to be

$$fitness(m) = |\texttt{influenced\_nodes}(m)|/|\texttt{RoI}(m)|.$$

Otherwise, we can make use of a low-cost basic-block profile to estimate the potential run-time benefit of eliminating a destructive merge. Let $count(m)$ be the number of times the destructive merge was executed in the profile run. We now define the fitness to be

$$fitness(m) = count(m) * |\texttt{influenced\_nodes}(m)|/|\texttt{RoI}(m).$$

In this way, frequently executed destructive merges are more likely to be eliminated, and our approach can concentrate on the hot regions of code. Finally, we choose the k fittest destructive merges to be eliminated. It should be noted that while this heuristic method does not guarantee that the code size increase is within some bound (C), it ensures that the code growth is not unbounded and works well in practice.

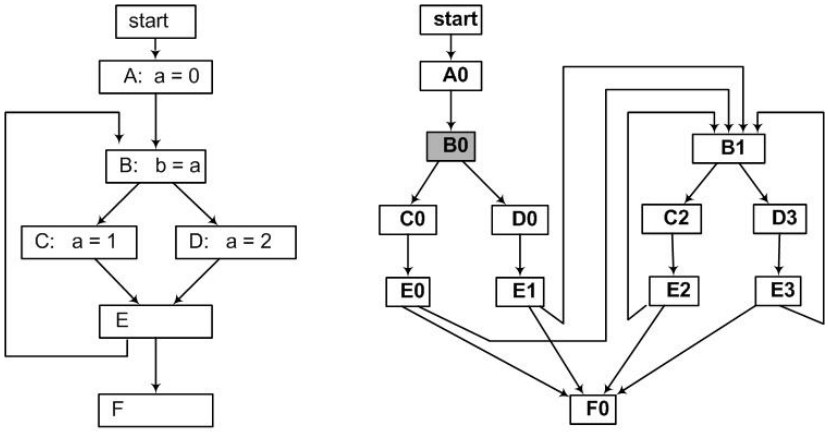

Figure 6: (a) Node E is a destructive merge.*Use* of variable a at node B cannot be replaced with a constant. (b)The *Hot Path Graph* corresponding to program $P1$. Node $B1$ is a destructive merge, and *use* of variable a still cannot be replaced by a constant.

Ball-Larus Acyclic Path	*Frequency*
$A \rightarrow B \rightarrow C \rightarrow E$	10
$B \rightarrow C \rightarrow E$	60
$B \rightarrow D \rightarrow E$	20
$B \rightarrow D \rightarrow E \rightarrow F$	10

Table 1: A path profile for the example in Figure 6. The frequency of the acyclic path denotes the number of times it was taken at runtime.

6. Related Work

6.1 Hot Path Graph Approach

An earlier proposal by Ammons and Larus [2] uses an acyclic path profile to try and improve the precision of the data-flow solution along hot paths. The approach consists of first using a Ball-Larus path profile [3] to determine the hot acyclic paths in the program. The next step in [2] consists of constructing a new CFG, called the *Hot Path Graph (HPG)*, in which each hot path is duplicated. The duplication eliminates control-flow merges along hot paths. The assumption being that this will alone will improve precision of data-flow analysis on the hot paths.

Consider the example code in Figure 6(a). Assume a path profile as shown in Table 1. Figure 6(b) shows the resulting HPG constructed assuming 100% coverage i.e. all taken paths are considered. Notice that in the HPG there are no control-flow merges along any of the acyclic paths listed in Table 1. For example, the two overlapping acyclic paths $B \rightarrow C \rightarrow E$ and $B \rightarrow D \rightarrow E$ in Figure 6(a) are separated into two separate paths $B1 \rightarrow C2 \rightarrow E2$ and $B1 \rightarrow D3 \rightarrow E3$ in the HPG. After performing conventional data-flow analysis on the HPG, the *use* of the variable a at node $B0$ can be replaced by the constant 0. However, the restructuring failed to optimize the two hot paths $B \rightarrow C \rightarrow E$ and $B \rightarrow D \rightarrow E$, and could not replace the *use* at node $B1$ with a constant value. The destructive merge E in the original CFG is removed in the HPG by duplicating code and creating two copies, $E2$ and $E3$. But the effect of the destructive merge has shifted to node $B1$, which is now a destructive merge since the data-flow facts $a = 1$ and $a = 2$ flowing along the incoming edges are merged at node $B1$. Thus, we see that simply duplicating acyclic paths does not always guarantee an increase in data-flow precision. Also, concentrating only on acyclic paths implies that all loop-back edges ($E2, B1$ and $E3, B1$ in the HPG) merge at a common loop-header (node $B1$ in the HPG). Thus, loop-headers which are destructive merges cannot be eliminated by the Ammons-Larus approach and data-flow precision is lost in these cases.

In comparison, the Split Graph constructed by our approach is shown in Figure 7. The destructive merge at node E is completely eliminated. In the Split Graph, *uses* of variable a at nodes $B0$, $B1$ and $B2$ can be replaced by constants 0, 1 and 2 respectively. Thus, we see that our approach effectively handles loop structures, guarantees additional optimization opportunities, and does not rely on expensive path profile information[1]. Note that several conventional control-flow graph restructuring approaches such as tail-duplication, superblock formation [15] can be seen to be similar to the HPG approach since they use only profile information. We compare the HPG method with our approach quantitatively in Section 7.

[1] For constructing the HPG the Ammons-Larus approach relies on the path profile information which is relatively more expensive than the simple basic block profile used in our Split Graph construction.

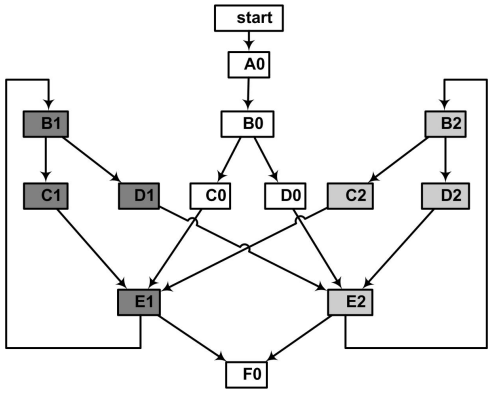

Figure 7: The *Split Graph* constructed from program $P1$ in which the *uses* of variable a at nodes $B0$, $B1$ and $B2$ can be replaced by constants 0, 1 and 2 respectively.

6.2 Other related approaches

In [6], an approach for complete removal of partial redundancy is described. Data-flow analysis is used to identify those regions of code which obstruct code motion. Code duplication and code motion are then used to eliminate the partial redundancy. Another approach targeted for PRE is discussed in [23]. Unlike our approach, [23] will result in unbounded code growth for finite height infinite lattices such as those seen in Constant Propagation. Further, [23] does not handle the precision versus code size trade-off and it is not clear from [23] how this gcan be accomplished. Martel [13] present an algorithm which uses graph subsitution to unroll loops so as to improve the precision of invariants detected via static analysis. Our control-graph restructuring is more general and can handle more cases. For example, the programs in Figure 1(a) and Figure 6(a) cannot be optimized using such loop unrolling.

Code restructuring need not necessarily be limited to within a procedure. An extension of Ammons-Larus approach to the interprocedural case is described in [14]. A more recent framework [24] for whole-program optimization also considers code duplication to perform *area specialization*, which is purely profile-driven.

There have also been several other approaches which do not restructure the CFG in order to improve data-flow analysis precision. Holley and Rosen presented a general approach to improve data-flow precision by adding a finite set of predicates [11]. In [5], the precision of def-use analysis is improved by determining infeasible paths by using a low overhead technique based on detection of static branch correlations. Interestingly, path-sensitivity can also be obtained by synthesizing the name space of the data-flow analysis [4]. *Property simulation* is introduced in ESP [7] and is used to verify temporal safety properties. This approach keeps track of the correlation between "property state" and certain execution states. In [8], data-flow analysis is performed over a *predicated lattice*. The predicates used are determined automatically using a counterexample refinement technique. In [9], the context-sensitivity of the pointer analysis is adjusted based on the requirements of the client application. These approaches are complementary to the approach described in this paper.

7. Experimental Results

We have implemented our approach in the Scale research compiler framework [21]. The framework is parameterised with the definition of a destructive merge, which depends on the data-flow analysis used, and on the definition of influenced nodes, which captures the interaction between the specific optimization and analysis. We present experimental results for the specific problem of Constant Propagation [1]. We compare our approach (*Split*) with the Wegman-Zadeck conditional constant propagation algorithm [25](*Base*) and the Hot Path Graph approach(*HPG*) [2] using the SPECINT 2000 benchmark suite [22].

7.1 Benefits of Split Approach

We instantiate the constant propagation phase of the O1 pass of the Scale compiler with the default approach (*Base*), the *HPG* approach, and *Split*. The HPG approach uses a path profile generated using the *train* inputs for the respective programs, while the our Split approach uses a basic block profile from the same *train* inputs. The benchmarks are compiled for DEC ALPHA and were run on the 500MHz 21264 Alpha workstation. Running times were measures as average over multiple runs using the larger *ref* inputs.

Table 2 shows the speedup obtained by our Split approach over the Base approach and over the HPG approach. Split gives an average speedup of 4% over the Base case, and it gives an average speedup of 2% over the HPG approach.

To understand where the speedup comes from, we calculate the number of dynamic instructions which have constant *uses* indentfied by the restructuring transformation. This is computed by first performing constant propagation and replacing all constant *uses* in the original program. Restructuring (HPG or Split) is then carried out. The constant *uses*

Benchmark	*% speedup of Split over Base*	*% speedup of Split over HPG*
175.vpr	5	1
186.crafty	-2	2
197.parser	2	3
256.bzip2	0	3
300.twolf	3	-2
181.mcf	12	4
164.gzip	3	2
average	4	2

Table 2: Percentage speedup in the running times using Split in comparison to Base and HPG.

Benchmark	*Split / HPG*
175.vpr	1.15
186.crafty	1.10
197.parser	1.27
256.bzip2	1.11
300.twolf	0.93
181.mcf	13.75
164.gzip	2.32
average	3.5

Table 3: Ratio of the number of dynamic instructions with constant *uses* in Split over HPG.

discovered can be attributed only to the restructuring. Thus, each instruction is weighted by the product of its execution count (using the *ref* inputs)and the number of new constant *uses*. The sum over all instructions gives us the number of dynamic instructions which have constant *uses* only because of restructuring. This metric has also been used in [2]. Table 3 shows the ratio of these instructions for Split over than of HPG. We observe an average of *3.5 times more* dynamic instructions with constants *uses* in Split as compared to HPG. In the 181.mcf Split results in as many as 13.75 times dynamic constant use instructions. This is because Split can handle cyclic structures effectively.

7.2 Cost of Split Approach

As mentioned earlier, the increase in precision comes at the cost of code duplication. We measured the code size in terms of the number of Scale intermediate instructions. Table 4 shows the ratio of the code size of Split over that of Base. We observe an average of *1.8× (80%) increase* due to Split. The Table also shows the ratio of code size of Split over that of HPG. We notice that Split incurrs less code size increase in comparison to HPG. Split shows an average of *0.65× (35%) decrease in code size* as compared to HPG.

8. Conclusion

We proposed a general framework to improve data-flow analysis precision based on restructuring the CFG of the program. The framework can be instantiated to any data-flow analysis. The actual transformation uses a known concepts of product automaton. We have proved that the transformation guarantees increase in optimization opportunities. Further, we showed that getting the optimal restructuring is NP-hard and proposed and evaluated a greedy heuristic. Our results indicate that our approach performs better than existing path profile driven approach [2].

References

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

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *PLDI*, pages 72–84, 1998.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[4] R. Bodík and S. Anik. Path-sensitive value-flow analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 237–251, 1998.

[5] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 361–377. Springer–Verlag, 1997.

[6] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, 1998.

[7] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

Benchmark	*Split / Base*	*Split / HPG*
175.vpr	1.5	0.7
186.crafty	2.0	0.7
197.parser	1.8	1.1
256.bzip2	1.9	0.5
300.twolf	2.0	0.7
181.mcf	1.9	1.0
164.gzip	1.5	0.8
average	1.8	0.65

Table 4: Ratio of code size increase of Split over Base, and of Split over HPG.

[8] J. Fischer, R. Jhala, and R. Mujumdar. Joining data flow with predicates. In *Foundations of Software Engineering*, pages 227–236, 2005.

[9] S. Guyer and C. Lin. Client-driven pointer analysis. In *International Static Analysis Symposium*, 2003.

[10] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[11] L. H. Holley and B. K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering (TSE)*, 7(1):60–78, 1981.

[12] D. C. Kozen. *Automata and Computability*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[13] M. Martel. Improving the static analysis of loops by dynamic partitioning techniques. In *Source Code Analysis and Manipulation (SCAM)*, pages 13–21, 26-27 Sept. 2003.

[14] D. Melski and T. Reps. The interprocedural express-lane transformation. In *Compiler Construction*, 2003.

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann, San Fransisco, CA, 1997.

[16] M. Müller-Olm and O. Rüthing. On the complexity of constant propagation. In *ESOP '01*, pages 190–205, London, UK, 2001. Springer-Verlag.

[17] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[18] T. A. Proebsting. Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years. https://research.microsoft.com/ toddpro/papers/law.htm. 1998.

[19] W. W. Pugh. Is Code Optimization (Research) Relevant?. http://www.cs.umd.edu/ pugh/IsCodeOptimizationRelevant.pdf.

[20] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *POPL '77*, pages 104–118, New York, NY, USA, 1977. ACM Press.

[21] Scale. A scalable compiler for analytical experiments. www-ali.cs.umass.edu/Scale/, 2006.

[22] SPEC. Standard Performance Evaluation Corporation. http://www.spec.org.

[23] B. Steffen. Property-oriented expansion. In *Third Static Analysis Symposium*, pages 22–41, 1996.

[24] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *PLDI*, June 2006.

[25] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *ACM Transactions on Programming Languages and Systems*, pages 181–210, 1981.