Improving Data-flow Analysis with Path Profiles[*]

Glenn Ammons
ammons@cs.wisc.edu

James R. Larus[†]
larus@cs.wisc.edu

Department of Computer Sciences
University of Wisconsin-Madison
1210 West Dayton St.
Madison, WI 53706

Abstract

Data-flow analysis computes its solutions over the paths in a control-flow graph. These paths—whether feasible or infeasible, heavily or rarely executed—contribute equally to a solution. However, programs execute only a small fraction of their potential paths and, moreover, programs' execution time and cost is concentrated in a far smaller subset of *hot paths*.

This paper describes a new approach to analyzing and optimizing programs, which improves the precision of data flow analysis along hot paths. Our technique identifies and duplicates hot paths, creating a *hot path graph* in which these paths are isolated. After flow analysis, the graph is reduced to eliminate unnecessary duplicates of unprofitable paths. In experiments on SPEC95 benchmarks, path qualification identified 2–112 times more non-local constants (weighted dynamically) than the Wegman-Zadek conditional constant algorithm, which translated into 1–7% more dynamic instructions with constant results.

1 Introduction

Data-flow analysis computes its solutions over the paths in a control-flow graph. The well-known, meet-over-all-paths formulation produces safe, precise solutions for general data-flow problems. All paths—whether feasible or infeasible, heavily or rarely executed—contribute equally to a solution. This egalitarian approach, unfortunately, is at odds with the realities of program behavior. Even moderately large programs execute only a few tens of thousands of paths (out of a universe of billions of acyclic paths) and, moreover, programs' execution time and cost is concentrated in a far smaller subset of *hot paths* [BL96, ABL97].

This paper presents a new data-flow analysis technique that attempts to compute more precise solutions along the hot paths in a program. Improved analysis along these paths can aid a compiler in optimizing these heavily executed portions of a program. *Path-qualified data-flow analysis* consists of the following steps:

1. Identify hot paths by profiling a program. We use a *Ball-Larus path profile* [BL96] to determine how often acyclic paths in a program execute.
2. Identify and isolate the hot paths in the program's control-flow graph (CFG). This step produces a new CFG in which each hot path is duplicated. Since a hot path is separated from other paths, data-flow facts along the path do not merge with facts from other, overlapping paths. Moreover, as programs do not execute many hot paths, this *hot-path graph (HPG)* is not much larger than the original graph.
3. Perform data-flow analysis on the HPG. The solutions found by this technique are conservative in the hot path graph—not in the original control-flow graph.
4. Reduce the graph to preserve only valuable solutions. The HPG duplicates code for paths whose solutions did not improve. Extra code both increases the cost of subsequent compiler analyses and adversely affects a processor's instruction cache and branch predictor. Reduction uses results from the data-flow analysis and frequencies from the path profile to decide which paths to preserve in the *reduced hot-path graph (rHPG).*
5. Translate the original path profile into a path profile for the rHPG, so profiling information is available for subsequent analyses and optimizations. Ball-Larus path profiles are determined by a set of recording edges, which start and end paths. The algorithm that produces an HPG also identifies recording edges in the HPG, which allows interpretation of the original path profile as a path profile of the HPG. The reduction step properly maintains these recording edges.

The technique can be applied to any data-flow problem, although this paper focuses on constant propagation. In experiments on SPEC95 benchmarks, path qualification identified 2–112 times more non-local constants (weighted dynamically) than the Wegman-Zadek conditional constant algorithm, which translated into 1–7% more dynamic instructions with constant results. Moreover, the technique is practical. With the exception of the `go` benchmark, the hot-path graphs were 3–32% larger and the reduced hot-path graphs were only 1–7% larger than the original CFG. On `go`, the hot-path graphs were 184% larger and the reduced hot-path graphs 70% larger.

[*]This research supported by: NSF NYI Award CCR-9357779, with support from Sun Microsystems and Intel, and NSF Grant MIP-9625558.

[†]On sabbatical at Microsoft Research.

To appear at the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada, June 17–19, 1998.

1.1 Qualified Flow Analysis

Our implementation is based on Holley and Rosen's qualified flow analysis technique [HR81]. A qualified data-flow problem is a conventional data-flow problem together with a deterministic finite automaton, A, whose transitions are labelled by the edges of the control-flow graph, G. A encodes additional information about the program—in path-qualified analysis, it recognizes hot paths. Data-flow analysis answers questions of the form "What can be said about the data-flow value at vertex v?" Qualified data-flow analysis answers questions of the form "What can be said about the data-flow value at vertex v given that A is in state q?"

Holley and Rosen used qualified data-flow analysis to identify infeasible paths and exclude them from analysis. They created an automaton in which infeasible paths ended in a failure state. The best solution at v, given that A is not in the failure state, is the meet over the non-failure states of A. For path qualification, we use the Aho-Corasick algorithm [Aho94] to construct an automaton that recognizes hot paths in a path profile.

Holley and Rosen describe two techniques for solving qualified problems, *data-flow tracing* and *context tupling.* This paper uses data-flow tracing, which constructs a new graph G_A whose vertices encode the vertex from G and the state from A. The qualified problem is then solved as a conventional data-flow problem over G_A—qualified solutions in G have become true solutions in G_A. G_A is, of course, our hot-path graph.

The qualified solution is never lower in the lattice than the unqualified solution. To see why, consider the solution at vertex v of G. If P is the set of all paths from routine entry to v and l_p is the data-flow value from path $p \in P$, then the meet-over-all-paths solution l_v at v is given by

$$l_v = \bigwedge_{p \in P} l_p.$$

Now partition P by the state of A. If Q is the set of states of A, $P_q \subseteq P$ is the set of paths in P that drive A to state $q \in Q$. It is clear that

$$\bigwedge_{p \in P} l_p = \bigwedge_{q \in Q} \bigwedge_{p \in P_q} l_p.$$

Or, omitting the outer meet on the right hand side and converting the equality to an inequality, for all $q \in Q$

$$\bigwedge_{p \in P} l_p \leq \bigwedge_{p \in P_q} l_p.$$

The inequality is not strict, so the qualified solution is not necessarily sharper than the meet-over-all-paths solution. However, when it is sharper, it is doubly beneficial to find this increased precision in heavily executed code.

1.2 Contributions

This paper makes four contributions:

- It shows how path profiles can improve the precision of data-flow analysis through guided code duplication.
- It describes how to reduce the hot-path graph, by eliminating paths that prove unnecessary or unprofitable.
- It shows how to preserve path-profiling information through the CFG transformations.
- It applies path qualification to constant propagation and demonstrates a significant improvement over the widely-used Wegman-Zadek technique.

1.3 Outline of the Paper

This paper is structured as follows. Section 2 sketches the theoretical groundwork and formalizes path profiles. Section 3 describes the automaton that recognizes the hot path in the profile. Section 4 shows how data-flow tracing constructs a new control-flow graph with duplicated hot paths. Section 5 shows how to reduce the traced graph. Section 6 presents the results of our experiments on SPEC95 benchmarks. Section 7 discusses related work.

2 Preliminaries

This section states definitions and theorems used in the rest of this paper.

2.1 Data Flow Problems

We begin with standard definitions of data-flow problems and their solutions.

Definition 1 *A* monotonic data-flow problem *D is a tuple $(L, \wedge, F, G, r, l_r, M)$ where:*

- *L is a complete semilattice with meet operation $\wedge$.*
- *F is a set of monotonic functions from L to L.*
- *$G = (V, E)$ is a control-flow graph with entry vertex r.*
- *$l_r \in L$ is the data-flow fact associated with r.*
- *$M : E \to F$ maps the edges of G to functions in F.*

M can be extended to map every path $p = [e_0, e_1, \ldots, e_k]$ in G to a function $f : L \to L$:

$$f = M(p) = M(e_k) \circ M(e_{k-1}) \circ \cdots \circ M(e_0)$$

The next three definitions come from Holley and Rosen [HR81].

Definition 2 *A* solution *I of D is a map $I : V \to L$ such that, for any path p from r to a vertex u, $I(u) \leq (M(p))(l_r)$.*

Definition 3 *A* fixpoint *J of D is a map $J : V \to L$ such that $J(r) \leq l_r$ and, if e is an edge from vertex u to vertex v, $J(v) \leq (M(e))(J(u))$.*

Definition 4 *A* good solution *I of D is a solution of D such that, for any fixpoint J, $J(u) \leq I(u)$ for all $u \in V$.*

2.2 Traced Data Flow Problems

As mentioned in Section 1, a qualified data-flow problem is qualified with respect to a finite automaton. This paper only considers qualification automata:

Definition 5 *If $G = (V, E)$ is a control-flow graph, a* qualification automaton *is a complete, deterministic finite automaton with transitions labelled by elements of E.*

Given a data-flow problem D over G and a qualification automaton A, a traced data-flow problem separates solutions by states of A, as well as by vertices of G. More formally:

Definition 6 *Let $D = (L, \wedge, F, G, r, l_r, M)$ be a data-flow problem over G and A be a qualification automaton. Then, the* traced data-flow problem *D_A is a data flow problem $(L, \wedge, F, G_A, r_A, l_{r_A}, M_A)$ where*

- *L, $\wedge$, and F are as in D.*
- *$G_A = (V_A, E_A)$ is a control-flow graph with entry vertex $r_A = (r, q_\epsilon)$. For $v \in V$ and $q \in Q$, $(v, q) \in V_A$ iff there exists a path p from r to v that drives A from its start state q_ϵ to q. In addition, $((v_0, q_0), (v_1, q_1)) \in E_A$ iff there exists an edge $(v_0, v_1) \in E$ and a transition in A from q_0 to q_1 on (v_0, v_1).*
- *$l_{r_A} = l_r$.*
- *$M_A : E_A \to F$ is defined by $M_A(((v_0, q_0), (v_1, q_1))) = M((v_0, v_1))$ where $(v_0, v_1) \in E_A$.*

D_A is a data-flow problem that can be solved by conventional means. Holley and Rosen prove the following theorem [HR81, Theorem 4.2]:

Theorem 1 *If I_A is a good solution of D_A, then the solution I of D given by $I(v) = \wedge\{I_A((u, q)) : (u, q) \in V_A, u = v\}$, for all $v \in V$, is a good solution of D.*

The meet in the theorem may lose information. As mentioned above, since I_A at (v, q) need only meet over paths to v that also drive the finite state machine to q, $I_A((v, q))$ can be strictly more precise than $I(v)$, even if I is the meet-over-all-paths solution.

An algorithm for producing D_A from D is given in Section 4.

2.3 Path Profiles

A path profile counts the number of times that a program traverses acyclic paths in a routine's CFG. Path profiles, along with a low-overhead algorithm to obtain them, are described in papers by Ball and Larus [BL96] and by Ammons, Ball, and Larus [ABL97].

The acyclic paths recorded in a profile start and end at *recording edges.* The set of recording edges, R, is, at a minimum: edges from the entry vertex, edges into the exit vertex, and retreating edges. Thus, removing the recording edges turns G into an acyclic graph. Additional edges may also be designated recording edges.

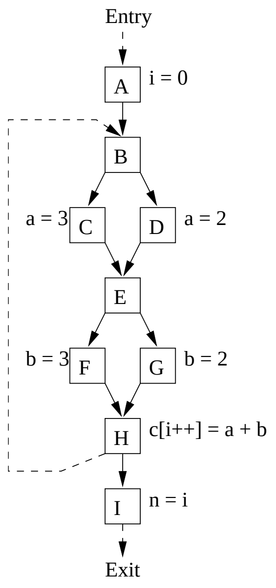


Figure 1: Our running example. Dashed lines indicate edges along which the Ball-Larus path profiling algorithm records the current path. Without path qualification, only the assignments of constants are constant instructions.

Definition 7 *Given a control-flow graph G and a set of recording edges R, a* Ball-Larus path *is a placeholder ($\bullet$) followed by a path in G from the target v_0 of some $r_0 \in R$ to the target v_1 of some $r_1 \in R$, which contains no recording edges besides r_1. The set of all such paths is denoted by P_{BL}. Given a Ball-Larus path $p_{BL} = [\bullet, e_0, e_1, \ldots, e_k]$ and a path p in G, we say that p_{BL} is a subpath of p if $[e_0, e_1, \ldots, e_k]$ occurs as a subpath of p that immediately follows an edge in R.*

The $\bullet$ is a reminder that Ball-Larus paths start with a recording edge. More than one recording edge may target a vertex, as for a doubly-nested loop in which both loops share a header. For Ball-Larus paths that start at such a node, the reminder does *not* specify which recording edge started a path.

As a running example, we will use the problem of finding instructions with constant results in the program in Figure 1. Without path qualification, the only constant instructions are the assignment statements in vertices A, C, D, F, and G.

The dashed edges in Figure 1 designate recording edges. In this example, $[\bullet, \mathrm{B}, \mathrm{C}, \mathrm{E}, \mathrm{F}, \mathrm{H}, \mathrm{B}]$ is a Ball-Larus path, while $[\bullet, \mathrm{C}, \mathrm{E}, \mathrm{F}, \mathrm{H}, \mathrm{I}]$ is not a Ball-Larus path.

Definition 8 *Given a multiset of paths P through a control-flow graph G and a set of recording edges R, a* path profile *is a multiset P_{PP} of Ball-Larus paths such that $p_{BL} \in P_{BL}$ occurs in P_{PP} with multiplicity equal to the number of times p_{BL} occurs as a subpath of paths in P.*

Consider the example in Figure 1. Writing the paths as lists of vertices, if P were:

(70 times) $[\mathrm{Entry}, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{E}, \mathrm{F}, \mathrm{H}, \mathrm{I}, \mathrm{Exit}]$

Path	Frequency
[•, A, B, C, E, F, H, I, Exit]	70
[•, A, B, D, E, F, H, B]	30
[•, B, D, E, G, H, B]	100
[•, B, D, E, F, H, I, Exit]	30

Figure 2: A path profile for the example.

(5 times)	[Entry, A, B, D, E, F, H] · [B, D, E, G, H]^5 ·[B, D, E, F, H, I, Exit]
(25 times)	[Entry, A, B, D, E, F, H] · [B, D, E, G, H]^3 ·[B, D, E, F, H, I, Exit]

the path profile is shown in Figure 2.

3 Creating the Automaton

This section describes an algorithm to construct a deterministic finite automaton that recognizes hot paths. The algorithm is an application of the Aho-Corasick algorithm for matching keywords in a string [Aho94]. In our case, the keywords are hot Ball-Larus paths. The constructed DFA is used as the qualification automaton for data-flow tracing.

The Aho-Corasick algorithm begins by constructing a retrieval tree (also known as a *trie*) from a set of keywords. A retrieval tree is a tree with edges labelled by letters from the alphabet, which satisfies two properties. First, each path from the root of the tree to a node corresponds to a prefix of a keyword from the set. Second, every prefix of every keyword has a unique path from the root that is labelled by letters of the prefix. Given a set of keywords, constructing its retrieval tree takes time proportional to the sum of the lengths of the keywords.

In our case, the alphabet is edges in a CFG and keywords are hot paths. Assuming that all paths in Figure 2 are hot, Figure 3 shows the retrieval tree. Our algorithm for constructing the retrieval tree consists of the following steps:

1. Identify the hot paths. In our experiments, we selected the minimal set of paths that executed a fixed fraction c_A (e.g., 97%) of the dynamic instructions in a training run. Hot paths were selected by considering each path, ordered by the number of instructions executed along the path (length times frequency), and marking paths hot until c_A of the dynamic instructions were covered.

2. Trim the final recording edge from each hot path. The constructed automaton will recognize these trimmed paths. Trimming paths ensures that the automaton returns to the same state after any recording edge.

3. Construct the retrieval tree for the set of trimmed hot paths.

Note that only one edge in the retrieval tree is labelled by •. In general, we make this definition:

Definition 9 *$q_\bullet$ is the target of the retrieval tree edge labelled by •.*

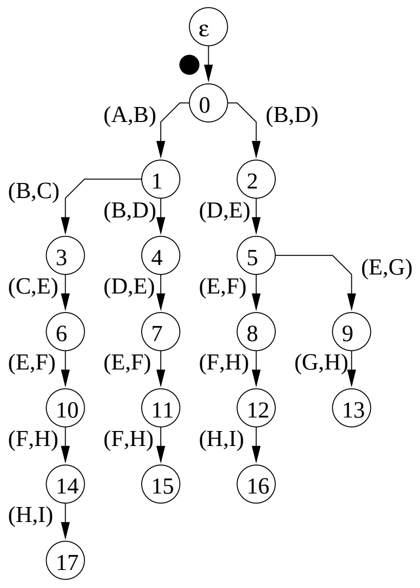


Figure 3: A retrieval tree for the path profile in our example.

In Figure 3, $q_\bullet = q_0$.

In Aho-Corasick, pattern matching steps through an input string while making transitions between vertices of the retrieval tree. At each step, if an edge from the current vertex in the tree is labelled by the next letter in the string, that edge is followed. If a leaf of the tree is reached, a match has been found. If no edge from the current vertex is labelled by the next letter (a), the current vertex (q) is reset by consulting a *failure function*, $h(q, a)$.

The failure function avoids rescanning the input string, by resuming scanning in the retrieval tree state corresponding to the longest keyword prefix that could lead to a match. If this prefix is nonempty, it must consist of a proper suffix of the match that just failed followed by a.

A Ball-Larus path p starts with a • representing a recording edge and ends with another recording edge. No other edges in p are recording edges, by definition. Thus, no paths start with a substring from the middle of another path, so the failure function always resets the automaton. The following theorem shows that the failure function becomes trivial.

Theorem 2 *Say q_u is the retrieval tree vertex representing the keyword prefix u. For any Aho-Corasick recognizer produced from a set of trimmed Ball-Larus paths, $h(q_u, a) = q_\epsilon$ if a is not a recording edge and $h(q_u, a) = q_\bullet$ if a is a recording edge.*

Proof*: Suppose v is the longest proper suffix of u that is also a prefix of some trimmed path in the profile. v must start with a •, which represents a recording edge. But no proper suffix of u contains a recording edge, so $|v| = 0$. If a is not a recording edge, it cannot begin a Ball-Larus path and $h(q_u, a) = q_\epsilon$. If a is a recording edge, then it is equivalent to • and so $h(q_u, a) = q_\bullet$.*

Since the failure function is trivial, our implementation only stores retrieval tree edges, which greatly reduces its size. If the automaton is in state q and sees the input a, the next state is found by checking:

- If there exists a retrieval tree edge from q labelled by a, the next state is the target of the edge.
- If a is a recording edge, the next state is $q_\bullet$.
- Otherwise, the next state is q_ϵ.

4 Building the HPG

This section explains how a hot path graph (HPG) is constructed. The algorithm both produces a graph for data flow analysis and also identifies recording edges in that graph, so that path profiling information can be carried over to later stages of compilation.

Section 4.1 applies Holley and Rosen's data-flow tracing algorithm to the original graph and the path qualification automaton from Section 3. The output of the tracing algorithm is a hot path graph without recording edges—G_A from Definition 6. In this HPG, every path from entry represents both a path in the original CFG and a path in the automaton. Moreover, two paths from entry end at the same vertex iff the corresponding paths in the CFG end at the same vertex and the corresponding paths in the automaton end in the same state. Thus, data-flow solutions over the HPG do not merge values from paths that reach different automaton states.

Section 4.2 explains how our algorithm also identifies the recording edges in the HPG, so that the path profile information can be correctly interpreted in the modified CFG.

Holley and Rosen discuss two qualification methods, one of which is data-flow tracing. The other method is context tupling. Section 4.3 explains why we use data-flow tracing instead of context tupling for path qualification.

4.1 Tracing the HPG

Figure 4 presents Holley and Rosen's algorithm for data-flow tracing, extended to identify recording edges (discussed in the next subsection). The algorithm is a worklist algorithm that finds all pairs of CFG vertices and automaton states reachable from the entry of the CFG (r) and the initial state of the automaton (q_ϵ). The vertices of the HPG are these pairs. Initially, the worklist holds (r, q_ϵ). Each iteration of the **While** loop removes a pair (v, q) from the worklist. The algorithm iterates over each pair (v', q') reachable in one step from (v, q). If (v', q') is not in the HPG, it is added to the HPG and the worklist. In any case, an edge is added from (v, q) to (v', q'). The algorithm terminates when the worklist is exhausted, at which point all possible pairs have been added to the HPG.

The constructed HPG fits the definition of G_A in Definition 6. The following theorem, together with Theorem 1, justifies performing data-flow analysis on the HPG.

Theorem 3 *When the algorithm in Figure 4 completes,*

i) $(v, q) \in V_A$ *iff there exists a path p in G from r to v that drives A from its start state q_ϵ to q.*

ii) $((v_0, q_0), (v_1, q_1)) \in E_A$ *iff there exists an edge $(v_0, v_1) \in E$ and a transition in A from q_0 to q_1 on (v_0, v_1).*

```
$G = (V, E)$ is a control-flow graph.
$A$ is a qualification automaton.
$Q$ is the set of states of $A$.
$q_\epsilon$ is the start state of $A$.
$T$ is the set of transitions in $A$.
$R \subseteq E$ is the set of recording edges.
$W$ is a worklist of pairs $(v, q)$, where $v \in V$ and $q \in Q$.
$G_A = (V_A, E_A)$ is the hot path graph.
$R_A \subseteq E_A$ is the new set of recording edges.

$V_A \leftarrow \{(r, q_\epsilon)\}$
$E_A \leftarrow \emptyset$
$R_A \leftarrow \emptyset$
$W \leftarrow (r, q_\epsilon)$
While $W \neq \emptyset$
    $(v, q) \leftarrow Take(W)$
    ForeachEdge $(v, v') \in E$
        $(q, (v, v'), q') \in T$ (it is unique)
        If $(v', q') \notin V_A$
            $V_A \leftarrow V_A \cup (v', q')$
            Put$(W, (v', q'))$
        $E_A \leftarrow E_A \cup \{((v, q), (v', q'))\}$
        If $(v, v') \in R$
            $R_A \leftarrow R_A \cup \{((v, q), (v', q'))\}$
```

Figure 4: An algorithm for data-flow tracing. The original Holley-Rosen algorithm has been extended to mark recording edges in the traced graph.

Proof*: The "only-if" direction of both requirements is obvious. The "if" direction follows by induction on the length of the paths p. For paths of length 0, both requirements are trivially true. If i) holds for all paths up to some length n and ii) holds for all edges reachable along such paths, then all final nodes of such paths must have been added to the worklist at some point in the algorithm. After these nodes are processed, the requirements hold for all paths up to length $n + 1$.*

Figure 5 shows the example after data-flow tracing. The automaton is in state q_ϵ at shaded vertices and state q_0 at vertices filled with diagonal lines. Only these vertices are targeted by multiple edges, as q_ϵ and q_0 are the only states in the automaton reached by multiple transitions.

The original graph had no constant results other than simple assignments, but the graph in Figure 5 has several new constant results: $\texttt{a} + \texttt{b}$ is always 6 at H14, 5 at H12 and H15, and 4 at H13, $\texttt{i}++$ is 1 at H14 and H15, and n is always 1 at I17.

Unfortunately, although the original flow graph in Figure 1 is reducible, the rHPG in Figure 5 is not. For example, the edge (Hϵ, B0) is a retreating edge but not a backedge in a natural loop since B0 does not dominate Hϵ. Because of this problem, tracing should only be used with data-flow solvers that can handle irreducible graphs.

4.2 Identifying Recording Edges in the HPG

The algorithm in Figure 4 makes an HPG edge $((v_0, q_0), (v_1, q_1))$ a recording edge iff (v_0, v_1) is a recording edge in the original graph. The next two lemmas show

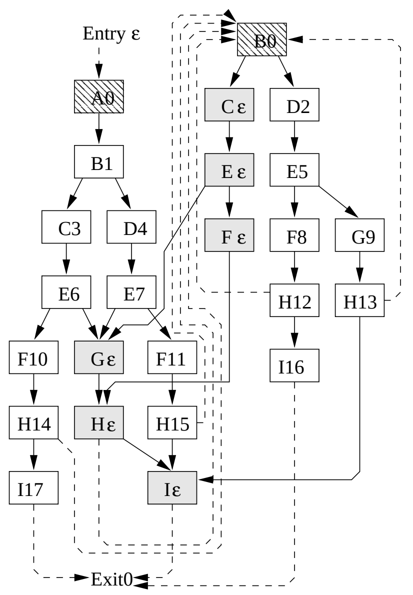


Figure 5: The control-flow graph of our example, traced with respect to the automaton in Figure 3. Dashed lines indicate edges along which the Ball-Larus path profiling algorithm records the current path. The automaton is in state q_ϵ at shaded vertices and state q_0 at vertices filled with diagonal lines.

Original Path	Traced Path
[•, A, B, C, E, F, H, I, Exit]	[•, A0, B1, C3, E6, F10, H14, I7, Exit0]
[•, A, B, D, E, F, H, B]	[•, A0, B1, D4, E7, F11, H15, B0]
[•, B, D, E, G, H, B]	[•, B0, D2, E5, G9, H13, B0]
[•, B, D, E, F, H, I, Exit]	[•, B0, D2, E5, F8, H12, I16, Exit0]

Figure 6: For each Ball-Larus path in the original path profile, the corresponding Ball-Larus path through Figure 5.

that these edges suffice to interpret the path profile from the original graph in the HPG.

Lemma 1 *Let p be a path from entry to exit in G and p_A the corresponding path in the HPG. Then, a Ball-Larus path begins at the kth edge in p iff a Ball-Larus path begins at the kth edge in p_A.*

Proof*: By the construction in Figure 4, if the kth edge in p is a recording edge, the kth edge in p_A is a recording edge, and vice versa.*

Lemma 2 *Given a Ball-Larus path $[\bullet, v_0, v_1, \ldots, v_k]$ in G, the corresponding HPG contains exactly one Ball-Larus path $[\bullet, (v_0, q_0), (v_1, q_1), \ldots, (v_k, q_k)]$.*

Proof*: By Lemma 1, some such Ball-Larus path exists in the HPG. All automaton transitions on recording edges target $q_\bullet$. Thus, $q_0 = q_\bullet$. By requirement ii) of Theorem 3 and the determinism of A, the rest of the path is determined.*

Lemma 1 implies that there is a one-to-one and onto map between any path profile of G and a path profile of the HPG. Lemma 2 implies that this map is unique and gives a way to construct the map: given a path from the original path profile that begins at v, start the path for the HPG profile at $(v, q_\bullet)$. The rest of the path can be laid out easily. Figure 6 shows the Ball-Larus path through Figure 5 for each path in Figure 2.

It is instructive to consider graphs in which the lemmas would fail. Lemma 1 would fail if the edge (H15, B0) in Figure 5, which is not a retreating edge, were not a recording edge. This is simply because (H, B) in Figure 1 is a recording edge, so the original path profile does not track correlations across it.

Lemma 2 would fail if recording edges targeted more than one B vertex. This could happen if, for example, tracing were allowed to unroll loops.

4.3 Context Tupling

In their original paper on qualified data-flow problems, Holley and Rosen presented two techniques for solving qualified problems [HR81]. Their first technique, data-flow tracing, has already been discussed. Their other technique is context tupling.

While data-flow tracing solves a conventional data-flow problem over an expanded graph, context tupling solves a "tupled" data-flow problem over the original graph. Intuitively, data-flow tracing tracks the state of A in the control-flow graph, while context tupling tracks the state of A in the lattice of values.

We chose data-flow tracing over context tupling for three reasons:

- Data flow tracing is easier to understand and explain.
- We envision that path profiles will be used throughout a sequence of compiler analyses and optimizations. With data-flow tracing, each analysis sees the results of previous passes through a modified control-flow graph. It is not obvious how to pass this information with context tupling.
- Holley and Rosen did not find context tupling to be any more efficient than data-flow tracing. Its main advantage is that context tupling does not produce irreducible CFGs, so elimination solvers can be used with context tupling.

5 Reducing the Traced CFG

This section describes our algorithm for identifying and eliminating vertices in the HPG that have either coarse data-flow results or low execution frequencies.

Figure 7 illustrates the need for this step. This chart shows the cumulative distribution of the number of executions of instructions with constant results by basic block. Constants that can be found solely through analysis within a basic block are excluded. For example, just 11 vertices account for virtually all nonlocal constants in `compress`. At the other extreme, 10,000 vertices are necessary to cover the constants in `go`. Tracing adds about 100 vertices to `compress` and over 93,000 vertices to `go`. Even if all vertices that contribute constants were included among the duplicated vertices, most new vertices would contribute only an insignificant improvement in the solution. The reduction algorithm eliminates many of these useless vertices.

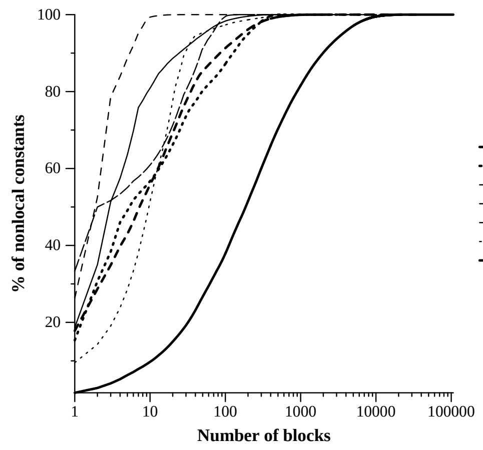


Figure 7: The distribution of dynamic executions of constant instructions by basic block in selected SPEC95 benchmarks.

In this paper, we describe the reduction algorithm in terms of constant propagation. However, the algorithm is not restricted to constant propagation. All that is necessary is a way to assign a benefit to duplicating a vertex in the HPG.

The reduction algorithm is a heuristic algorithm with the following steps:

1. Identify the hot vertices. First, the vertices are ordered by the number of dynamic constants they execute, as computed from the path profile. In our experiments, we chose a fixed fraction c_R of the (nonlocal) dynamic constants as a goal. Vertices are added to the set of hot vertices until c_R is reached.

   In our running example, H12 weighs 30, H13 weighs 100, H14 weighs 140, H15 weighs 60, and I17 weighs 70. All the other vertices have weight 0. For the sake of the example, suppose c_R is chosen such that H13 and H14 are the only hot vertices.

2. For each vertex v in the original graph, partition the vertices (v, q) in the HPG into sets of vertices that are compatible. This is the heuristic step of the algorithm. Call the partition Π. At this stage, two vertices are *compatible* if neither vertex is hot or, if one or both is hot, lowering both solutions to the meet of their lattice values does not destroy any constants in a hot vertex.

   Compatibility is not an equivalence relation (it is not transitive), so Π cannot be found by looking for equivalence classes. Instead, Π is formed greedily by "throwing in" vertices one at a time. As each vertex (v, q) is thrown in, it merges with the first $S \in \Pi$ for which adding (v, q) to S does not destroy constants in a hot vertex. If there is no such S, (v, q) starts a new set. Our implementation tries to keep hot vertices together by considering the vertices in descending order by weight.

   In the example, since H13 and H14 are the only hot vertices, Π is

   $$\{\mathrm{Entry}\epsilon\}, \{\mathrm{A0}\}, \{\mathrm{B0}, \mathrm{B1}\}, \{\mathrm{C}\epsilon, \mathrm{C3}\}, \{\mathrm{D2}, \mathrm{D4}\}, \{\mathrm{E}\epsilon, \mathrm{E5}, \mathrm{E6}, \mathrm{E7}\}, \{\mathrm{F}\epsilon, \mathrm{F8}, \mathrm{F10}, \mathrm{F11}\}, \{\mathrm{G}\epsilon, \mathrm{G9}\}, \{\mathrm{H}\epsilon, \mathrm{H12}, \mathrm{H15}\}, \{\mathrm{H13}\}, \{\mathrm{H14}\}, \{\mathrm{I}\epsilon, \mathrm{I16}, \mathrm{I17}\}, \{\mathrm{Exit0}\}$$

3. Use the standard DFA minimization algorithm [Gri73] to produce a partition Π' which respects the dataflow solutions. The complexity of this algorithm is $O(n \log n)$.

   Why is this algorithm applicable? The HPG can be thought of as a finite automaton with edges labelled by the edges of the original graph. The elements of Π can be thought of as equivalence classes of final states of an automaton that recognizes several different kinds of tokens.

   The only way to lower the solution over a set $S \in \Pi$ is to cause some new path p from entry to reach a vertex in S. Viewing p as a string, that would mean that p was not recognized as a token of type S before minimization but is recognized as such a token after minimization. This cannot happen.

   In our example, the minimized partition Π' is

   $$\{\mathrm{Entry}\epsilon\}, \{\mathrm{A0}\}, \{\mathrm{B0}\}, \{\mathrm{B1}\}, \{\mathrm{C}\epsilon\}, \{\mathrm{C3}\}, \{\mathrm{D2}\}, \{\mathrm{D4}\}, \{\mathrm{E}\epsilon, \mathrm{E7}\}, \{\mathrm{E5}\}, \{\mathrm{E6}\}, \{\mathrm{F}\epsilon, \mathrm{F8}, \mathrm{F11}\}, \{\mathrm{F10}\}, \{\mathrm{G}\epsilon\}, \{\mathrm{G9}\}, \{\mathrm{H}\epsilon, \mathrm{H12}, \mathrm{H15}\}, \{\mathrm{H13}\}, \{\mathrm{H14}\}, \{\mathrm{I}\epsilon, \mathrm{I16}, \mathrm{I17}\}, \{\mathrm{Exit0}\}$$

4. Replace the vertices in each set in Π' with a representative and produce a new set of recording edges. If $S_0, S_1 \in \Pi'$ have representatives s_0 and s_1, respectively, an edge (s_0, s_1) exists iff an edge $((v_0, q_0), (v_1, q_1))$ exists in the HPG, where $(v_0, q_0) \in S_0$ and $(v_1, q_1) \in S_1$. If $((v_0, q_0), (v_1, q_1))$ is a recording edge, (s_0, s_1) is a recording edge. This is well-defined: for $(v_0, q_0), (v_0, q_0') \in S_0$ and $(v_1, q_1), (v_1, q_1') \in S_1$, $((v_0, q_0), (v_1, q_1))$ is a recording edge iff (v_0, v_1) is a recording edge in the original graph and the same for $((v_0, q_0'), (v_1, q_1'))$, so $((v_0, q_0), (v_1, q_1))$ is a recording edge iff $((v_0, q_0'), (v_1, q_1'))$ is a recording edge.

   Figure 8 shows the reduced hot path graph for our running example.

6 Experimental Results

This section presents measurements of the benefits and costs of using path-qualified flow analysis for constant propagation. We implemented the analysis as two new passes in the SUIF compiler [WFW$^+$]. The first pass, PP, instrumented a C program for path profiling. The other pass, PW, used a path profile to perform path-qualified constant propagation.

The first step was to produce a path profile for each routine in the program. In this stage, SUIF compiled a C program into its low-SUIF intermediate form. The PP pass instrumented this intermediate code for path profiling. We did not run SUIF's optimization passes. The SUIF-to-C converter transformed PP's output into C code, which was compiled by GCC into an instrumented program. When run, this program produced a path profile.

The next step was to optimize programs. The program was again compiled by SUIF. This time, the SUIF code was

Program	Nodes	Paths	Hot Paths ($c_A = .97$)	Compile Time	Anal. Time ($c_A = 0$)
go	12952	15236	3533	501	91
m88ksim	6404	449	88	626	35
compress	279	90	28	99	1
li	3321	345	62	220	11
ijpeg	6718	960	104	414	32
perl	14995	588	74	539	602
vortex	21190	1729	152	1042	163

Table 1: General information about the benchmarks. **Nodes** is the total number of CFG nodes in the original program. **Paths** is the number of Ball-Larus paths executed in the training run. **Hot Paths** is the number of paths needed to cover 97% of a training run's dynamic instructions. **Compile Time** is the total compile time (seconds) *without* constant propagation. **Anal. Time** is the total time (seconds) required for constant propagation with $c_A = 0$.

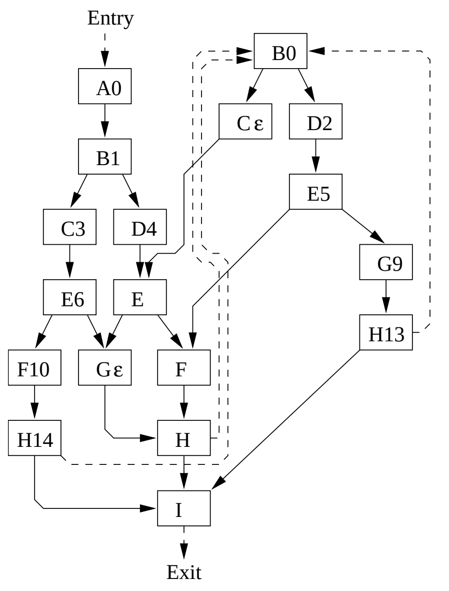


Figure 8: The control-flow graph after reduction. State numbers have been dropped from all merged vertices.

fed to PW together with the previously obtained path profile. PW used the path profile to construct a hot path graph, discover constants, produce a reduced hot path graph, and finally generate optimized code. The output of PW was converted to C code, which was compiled by GCC (-O2) into an optimized executable.

As SUIF did not directly generate assembly or machine code, our evaluations are in terms of the SUIF intermediate code. In this paper, by "instruction" we always mean SUIF instructions, not machine instructions.

The constant propagator in PW uses Wegman and Zadek's *Conditional Constant* algorithm [WZ91]. This algorithm is a worklist algorithm that symbolically executes a routine, starting at its entry node and propagating values only across the legs of branches that can execute, given the current assignment of values to variables. Our implementation is conservative, as it does not track pointers or constants manipulated through pointers or structures, assumes that calls and assignments through pointers write to all aliased variables, and initially sets all variables to $\perp$. Since we ran the constant propagator immediately after SUIF's front end, the constant propagator saw code that was very close to the original C.

We ran PP and PW on seven of the C SPEC95 benchmarks on a Sun UltraSPARC SMP. In all cases, we used an input data set from the SPEC `train` data to produce the path profile that drove the flow analysis. A different and larger input from the `ref` data set produced a path profile used to evaluate the effectiveness of the constant propagator. The path profile of the reference input did not affect the optimization; it was only used to compute the dynamic number of constants discovered by the propagator.

Path-qualified analysis becomes more expensive as the number of hot paths increases. On the other hand, considering more paths can improve a solution, as it increases the portion of the program's execution covered by an analyzed path. To quantify this tradeoff, we ran the path-qualified analysis several times, varying *path coverage*—the c_A parameter in Section 3. That is, the analysis was first run on the minimum set of paths that covered three quarters of the program's execution, then on paths that covered seven eighths of the execution, and so forth.

The other parameter in our analysis is c_R, the benefit cutoff for the graph reduction algorithm. In the experiments, we set c_R to .95, so reduction preserved approximately 95% of the nontrivial constants discovered by constant propagation. This value was arrived at empirically.

Table 1 lists basic information about the benchmarks. Most of the analysis time for `perl` was spent in two huge routines, `yylex` and `eval`, for which the non-linear running time of constant propagation became a problem.

6.1 Benefit of Path Qualification

Figure 9 shows that the number of executed instructions with statically constant results increased as the hot path coverage increased. At full coverage ($c_A = 1$), the improvement ranged from 7% for `m88ksim` and `vortex` to 0.6% for `perl`. In all benchmarks, most of the benefit of path qualification was attained before full coverage was reached—typically somewhere above 90% coverage. `ijpeg` attained most of its benefit at $c_A = 0.75$ (the lowest non-zero value tested) but all benchmarks saw virtually all of their benefit by $c_A = 0.97$. In two cases, the improvement degraded slightly at high coverage, because of heuristics in the reduction algorithm. These results confirm earlier path profiling

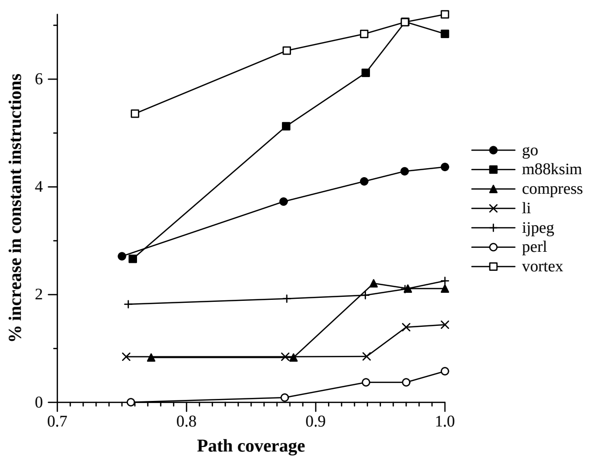


Figure 9: Increase in instructions with constant results (weighted dynamically) versus the level of path coverage. The baseline is the number of constants at $c_A = 0$ (Wegman-Zadek).

Program	**Base** ($c_A = 0$)	**Optimized** ($c_A = 0.97, c_R = 0.95$)	**Speedup**
`go`	253	231	9.2%
`m88ksim`	391	356	9.8%
`compress`	237	244	-2.9%
`li`	345	361	-4.4%
`ijpeg`	144	147	-1.9%
`perl`	173	176	-2.0%
`vortex`	417	410	1.9%

Table 2: Effects of path-qualified constant propagation on running time. **Base** is the running time in seconds of the program after Wegman-Zadek constant propagation. **Optimized** is the running time in seconds of the program after path-qualified constant propagation with c_A set to 0.97 and c_R set to 0.95. All running times are on the `ref` data set. **Speedup** reports the improvement in running time.

measurements, which show that a small kernel of hot paths dominate a program's execution.

6.1.1 Running Time

We measured each benchmark's execution time after constant propagation with $c_A = 0.97$ and $c_R = 0.95$ and compared this time against the program's time with only Wegman-Zadek constant propagation ($c_A = 0$). The runs were on the `ref` data set. The best time from three runs is reported.

The relationship between constants and program speed is not clear. The three benchmarks with the largest number of newly discovered constants sped up, while the other benchmarks slowed down. However, the change in execution time was not proportional to the increase in the number of constants. For example, the speedup for `go` was almost equal to that of `m88ksim`, but `go` only showed a 4% increase in constant instructions while `m88ksim` showed a 7% increase. (Keep in mind that the increase in constant instructions is a dynamic measure.) Also, `m88ksim` and `vortex` had approximately the same increase in constant instructions, but `vortex` sped up by 1.9% while `m88ksim` sped up by 9.8%. Similarly, the largest slowdown was in `li`, yet `perl` had the smallest increase in constant instructions.

Furthermore, running times do not seem to relate easily to the increase in program size. For example, `go` had a good speedup, but its size increased by the largest amount (Figure 11).

To be fair, this experiment did not control for several significant factors. First, the IMPACT group's work on superblock scheduling [mWHMC$^+$93] found that tail duplication, like that done to isolate hot paths, can expose large amounts of instruction-level parallelism. Thus, running time improvements are not necessarily due to improvements in constant propagation. Second, we rely on GCC to perform all optimizations beyond constant propagation, but GCC may produce poor code for the irreducible graphs produced during tracing. Third, because a node can have at most one fall-through predecessor, tracing can introduce extra jumps. For example, E can fall through to F in Figure 1, but it is impossible for both E5 and E to fall through to F in Figure 8. PW could use the path profile to place these jumps more intelligently or to further duplicate code to avoid jumps altogether, but our implementation does neither. Fourth and last, our experiments did not measure the effect on the instruction cache or branch predictor.

6.2 Classifying Constants

This section examines the constants discovered by path qualification. The Venn diagram in Figure 13 classifies dynamic instructions based on the type of analysis required to identify them as either constant or dynamic. The categories are:

Local These instructions can be determined to be constant with local analysis—that is, by scanning their enclosing basic block. In Figure 1, assignments to `a` and `b` are local constants.

Iterative These instructions can be determined to be constant by Wegman-Zadek iterative analysis. We found these constants by running the constant propagator with $c_A = 0$.

MOP These instructions are found to be constant by a meet-over-all-paths solution. Constant propagation is not a distributive problem, so an iterative solution may be less precise than a meet-over-all-paths solution. We cannot measure this category directly.

Qualified These instructions are found to be constant by the path-qualified analysis. This set does not contain **MOP**, nor does **MOP** contain **Qualified**. The intersection of the two sets includes **Iterative** and possibly other constants. Because we cannot measure **MOP**, we cannot measure the intersection precisely. The **Identical** and **Variable** sets below attempt to approximate this intersection.

Identical These instructions include all **Iterative** instructions, plus instructions not found by Wegman-Zadek for which path-qualified analysis finds the same constant value everywhere they are duplicated. These constants would also be found by meet-over-all-paths.

Variable These instructions are found to be constant by the qualified analysis, but have different values at different sites in the reduced graph. For example, in Figure 8, the value of $\mathtt{a} + \mathtt{b}$ is 6 at H14 and is 4 at H13. Only duplication will reveal these constants. Meet-over-all-paths will not find these constants.

Unknowable Instructions in this category either are not constant or cannot be identified as constant because of other limitations of the analyses. Our analyses do not track pointers, values stored in memory, or the results of calls. Therefore, instructions that consume these values will never be found constant. We estimated this set by counting the number of values produced within a basic block, yet found equal to $\perp$.

Figure 10 divides instructions (dynamically weighted) into these categories. Most instructions in each benchmark fall in the **Unknowable** or **Local** categories. Path qualification does not affect these categories.

The other part of Figure 10 focuses on the instructions targeted by constant propagation algorithms. Our technique found many (2–122) times more knowable and nonlocally constant instructions. Interestingly, most instructions found constant by qualified analysis were neither **Identical** nor **Variable**. These instructions had one constant value at one or more sites and were also unknown at one or more sites. The exceptions are `vortex` and `go`, both of which contained a significant, but small, number of **Variable** constants. Other techniques, which do not duplicate paths, will not find these constants.

Although the direct improvement from our technique is large, the instructions it finds constant still make up a small percentage of all dynamic instructions. This further explains why we did not see speedups for most of the benchmarks.

In the above discussion, we assumed that the **MOP** is not attainable for constant propagation. This is true for the non-distributive Wegman-Zadek formulation. Recently, Bodík and Anik published a distributive formulation of constant propagation [BA98]. It would be interesting to compare path-qualified analysis against this formulation.

6.3 Cost of Path Qualification

This section examines the cost of path qualification.

6.3.1 Cost of Duplication

Figure 11 shows that CFG size only increased significantly for `go` and that the reduction algorithm successfully controlled the increase in CFG size.

The cost of data-flow analysis is proportional to the CFG's size before reduction. For `go`, the maximum increase was 722%, and for the other programs the maximum increase was 80%. However, Figure 9 showed that 100% coverage offers little benefit. 97% coverage achieves almost all of the benefit, and limits CFG growth to 184% for `go` and 32% for the other programs.

The CFG's size after reduction is an indirect measure of the spatial locality of the constants found. Our experiments show that this locality is high—with $c_R = 0.95$, only `go` grew by more than 10% at any level of coverage. `go` grew by 77% at full coverage, but, again, full coverage is unnecessary: at $c_A = 0.97$, its increase was 70%. The cost of subsequent

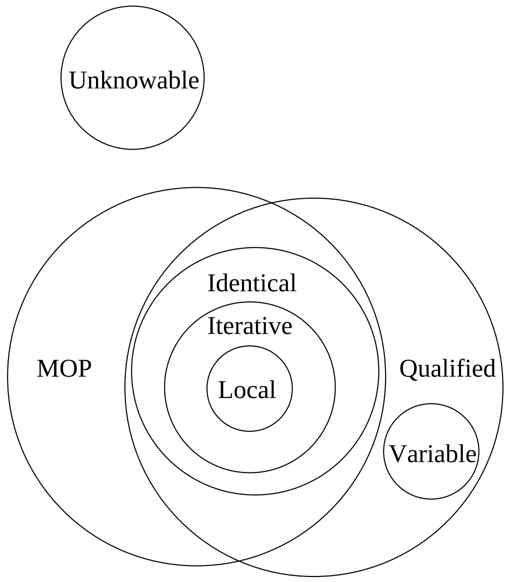


Figure 13: A Venn diagram classifying a program's dynamic instructions.

analysis and the running time of the program may degrade as the CFG grows, but these increases seem manageable.

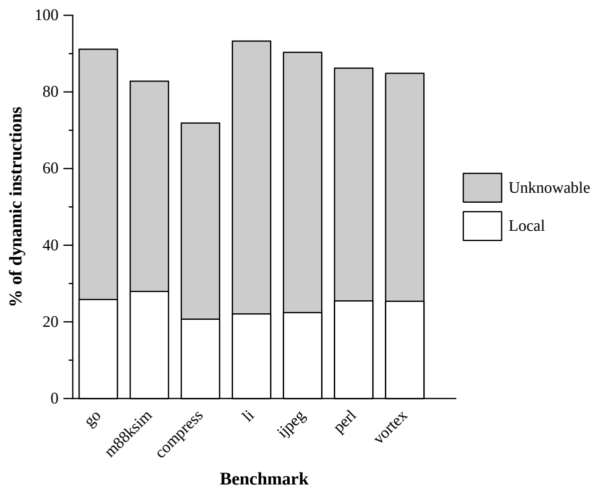
Why was `go` exceptional? Table 1 shows that `go` executed many more paths than other programs and also required more paths to reach high coverage levels. Further experiments are necessary to see whether `go`'s distribution is atypical or not.

6.3.2 Analysis Time

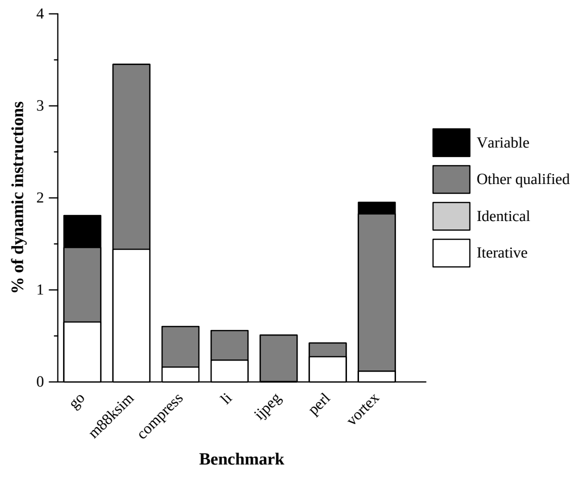
Path-qualified data-flow analysis increases analysis time, both by adding three new steps—building the qualification automaton, tracing, and reduction—and by running the data-flow solver on larger graphs. Figure 12 shows the relative increase in analysis time as c_A is increased. Once again, `go` was exceptional. For the other benchmarks, the increase was less than 61% at almost full coverage. Figure 9 shows that most of the benefit is gained before full coverage, so these increases are reasonable. For `go`, analysis time increased sixfold at $c_A = 0.97$. The observed analysis time seems to grow a bit faster than linearly with the size of the hot path graph.

7 Related Work

Feasible path analysis attempts to identify and eliminate infeasible paths. Holley and Rosen introduced qualified data-flow analysis to separate known infeasible paths from the remaining paths, some of which might be feasible [HR81]. Goldberg et al. applied theorem proving techniques to identify infeasible paths in testing a program's path coverage [GWZ94]. Bodík et al. used a weaker (but less expensive) decision technique to determine if all paths between a definition and use were infeasible, and therefore the def-use pair actually did not exist [BGS97b]. Our work differs from these, as we focus on directly improving the precision of



(a) Local and unknowable



(b) Other categories

Figure 10: Fraction of dynamic instructions that fall into categories in Figure 13. The qualified analysis was done at full coverage ($c_A = 1$).

program analysis along a subset of important paths, rather than improving analysis everywhere by eliminating spurious paths. However, the two techniques are certainly complementary, as our technique would work well in a CFG from which infeasible paths were eliminated.

Paths have long been used in program analysis and optimization. Fisher's trace scheduling technique heavily optimized the hot paths (called traces) in a CFG [Fis81]. Trace scheduling did *not* duplicate paths, instead it introduced fixup code along control flow edges into or out of the middle of a trace. More recently, Hwu et al. eliminated this fixup code by duplicating paths to form superblocks, which is a collection of traces without control flow into the middle of a trace [mWHMC$^+$93]. Our approach differs from both techniques. First, it is a technique for improving program analysis, not a technique for optimization and instruction scheduling. Second, although it duplicates paths, like superblocks, its duplication is guided by path profiles. Finally, both scheduling techniques attempted to maximize the size of traces. This work evaluates the improvement from duplication, and eliminates duplicated blocks that provide little or no improvement.

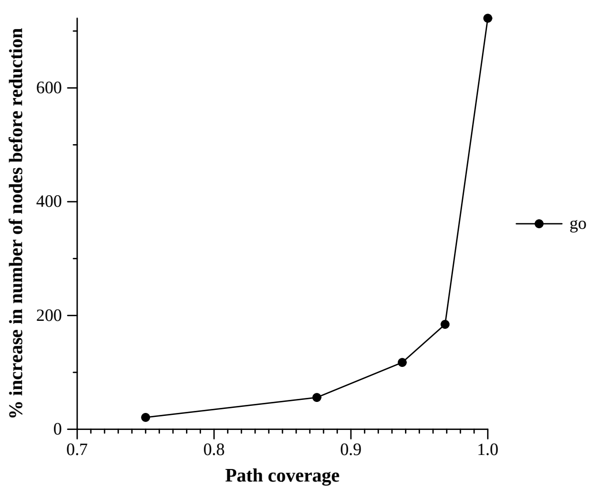
Mueller and Whalley used an ad-hoc framework and code duplication to eliminate certain partially redundant branches [MW95]. Mueller and Whalley's code duplication algorithm can be seen as a qualification algorithm in which states in the qualification automaton encode information about the direction of the partially redundant branches. Bodík et al. used a limited form of interprocedural analysis to detect redundant branches along interprocedural paths [BGS97a]. This work differs by incorporating paths into a more precise and general framework, by using paths to derive more precise data-flow analyses, and by using path frequencies to overcome the costs of exploiting increased precision (code duplication).

Bodík et al. presented an algorithm for complete partial redundancy elimination using both code motion and code duplication [BGS98]. Their technique also used profiles (either edge or path) to drive code duplication. Our paper is not directly comparable with their paper, as their paper used duplication to carry out an optimization while our paper uses duplication to improve analysis. However, there is a difference in philosophy between the two papers. They first analyzed the original control flow graph to identify vertices for which duplication would enable better code motion. Using a profile, their algorithm decides which of these candidates *should* be duplicated. Our work takes the other tack: a profile guides an initial round of duplication. Analysis of the duplicated flow graph, together with the profile, identifies blocks that should *not* have been duplicated. By contrast, their approach starts and performs analysis over a smaller graph. Our approach, however, can find solutions not found by a meet-over-all-paths analysis.

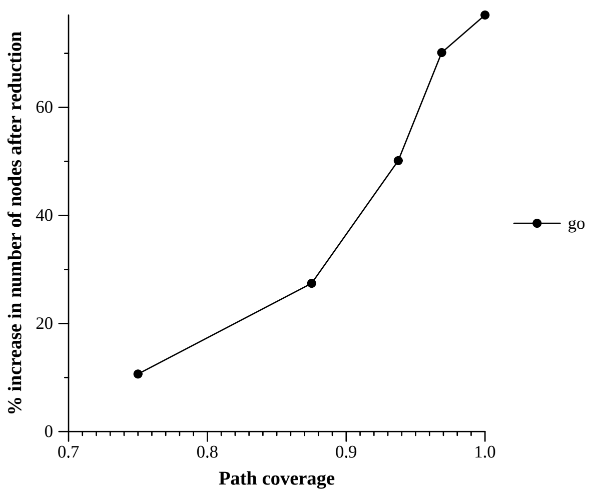
Ramalingam combined data-flow analysis with program frequency information by associating probabilities with data-flow values and developing a data-flow framework for combining these pairs of values [Ram96]. Our goal differs. Instead of incorporating frequencies into the meet-over-all-paths framework, we use frequency information to improve analysis precision in heavily executed code.

8 Conclusion

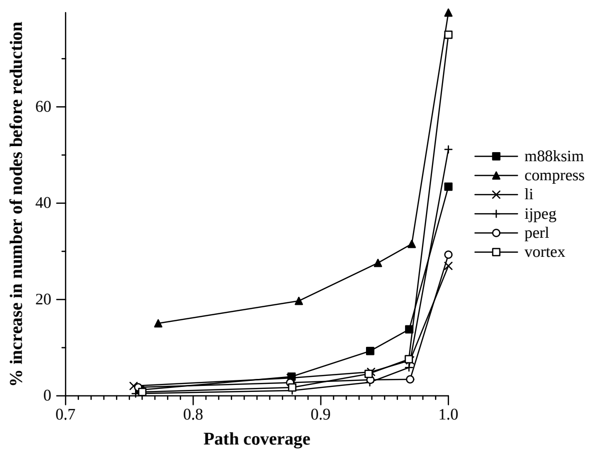
This paper describes a new approach to analyzing and optimizing programs. Our technique starts with a path profile that identifies the hot paths that incur most of the program's cost. This information provides the basis for a hot path graph, in which hot paths are isolated in order to compute data-flow values more precisely. After analysis, the hot path graph is reduced to eliminate unnecessary or unprofitable paths. We applied this technique to constant propagation



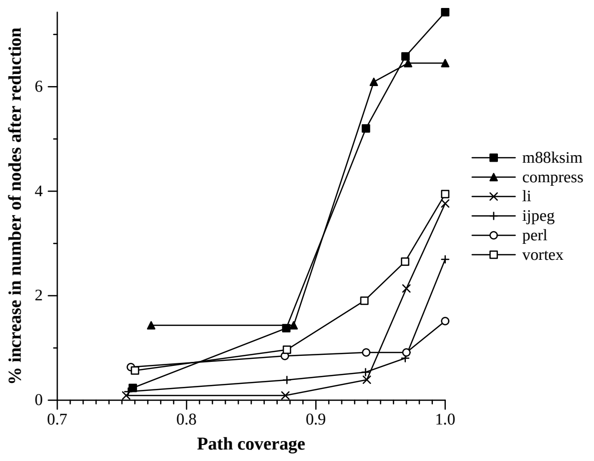
(a) Before reduction (go)



(b) After minimization (go)



(c) Before reduction (other benchmarks)



(d) After minimization (other benchmarks)

Figure 11: Increase in the number of CFG nodes before and after reduction versus the level of path coverage. The baseline is the unoptimized program. The scale of the y-axis differs in each graph. The graph on the left (i.e., before reduction) is approximately an order of magnitude larger than the graph after reduction. Also, the scale for go is about an order of magnitude larger than the other graphs.

and obtained significant improvement over the widely-used Wegman-Zadek technique, without a large increase in program size.

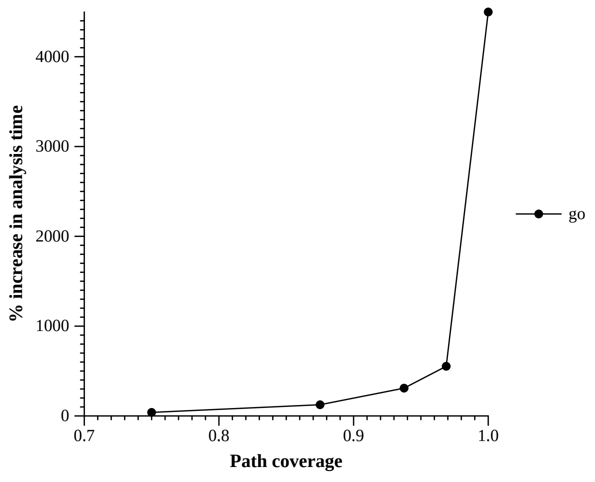
This technique is applicable to other data-flow problems, as well. Aside from its simplicity, its primary advantage is that it improves the precision of an analysis (by excluding the effect of infeasible or infrequently executed paths) in the heavily executed portions of a program, where the benefits are largest.

Acknowledgements

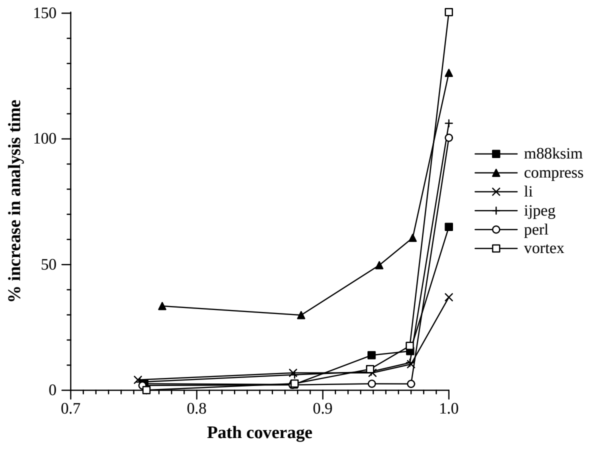
Thomas Reps spotted a problem with an earlier formulation of our interpretation of path profiles of the hot path graph. Tom Ball, Rajiv Gupta, Rastislav Bodík, and the anonymous reviewers provided many helpful suggestions and comments on this work.

References

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.



(a) go



(b) Other benchmarks

Figure 12: Time required for qualified flow analysis versus path coverage (c_A). The baseline is the time required at $c_A = 0$.

[Aho94] Alfred V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, pages 255–300. MIT Press, 1994.

[BA98] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the SIGPLAN '98 Symposium on Principles of Programming Languages (POPL)*, January 1998.

[BGS97a] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 146–158, June 1997.

[BGS97b] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *Fifth ACM SIGSOFT Symposium on Foundations of Software Engineering and Sixth European Software Engineering Conference*, September 1997.

[BGS98] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant computations. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, June 1998. To appear.

[BL96] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of MICRO 96*, pages 46–57, December 1996.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[Gri73] David Gries. Describing an algorithm by hopcroft. *Acta Informatica*, 2:97–109, 1973.

[GWZ94] Allen Goldberg, T. C. Wang, and David Zimmerman. Applications of feasible path analysis to program testing. In *International Symposium on Software Testing and Analysis*. ACM SIGSOFT, August 1994.

[HR81] L. Howard Holley and Barry K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, SE-7(1):60–78, January 1981.

[MW95] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 56–66, June 1995.

[mWHMC$^+$93] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, May 1993.

[Ram96] G. Ramalingam. Data flow frequency analysis. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 267–277, May 1996.

[WFW$^+$] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. An overview of the SUIF compiler system. Published on the World Wide Web at http://suif.stanford.edu/suif/suif1/suif-overview/suif.html.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.