The Interprocedural Express-lane Transformation

David Melski[1] and Thomas Reps[2]

[1] GrammaTech, Inc., melski@grammatech.com
[2] Comp. Sci. Dept., Univ. of Wisconsin, reps@cs.wisc.edu

Abstract. The express-lane transformation isolates and duplicates frequently executed program paths, aiming for better data-flow facts along the duplicated paths. An express-lane p is a copy of a frequently executed program path such that p has only one entry point at its beginning; p may have branches back to the original code, but the original code never branches into p. Classical data-flow analysis is likely to find sharper data-flow facts along an express-lane, because there are no join points.

This paper describes several variants of interprocedural express-lane transformations; these duplicate hot interprocedural paths, i.e., paths that may cross procedure boundaries. The paper also reports results from an experimental study of the effects of the express-lane transformation on interprocedural range analysis.

1 Introduction

In path profiling, a program is instrumented with code that counts the number of times particular finite-length path fragments of the program's control-flow graph—or *observable paths*—are executed. One application of path profiling is to transform the profiled program by isolating and optimizing frequently executed, or *hot*, paths. We call this transformation the *express-lane transformation*. An *express-lane* p is a copy of a hot path such that p has only one entry point at its beginning; p may have branches back to the original code, but the original code never branches into p. Classical data-flow analysis is likely to find sharper data-flow facts along the express lanes, since there are no join points. This may create opportunities for program optimization.

We use the interprocedural express-lane transformation together with range analysis to perform program optimization. Our approach differs from the literature on profile-driven optimization in one or more of the following aspects:

1. We duplicate interprocedural paths. This may expose correlations between branches in different procedures, which can lead to more optimization opportunities [5].
2. We perform code transformation before performing data-flow analysis. This allows us to use classic data-flow analyses.
3. We guide path duplication using interprocedural path profiles. This point may sound redundant, but [7], for example, uses edge profiles to duplicate intraprocedural paths. The advantage of using interprocedural path profiles is that we get more accuracy in terms of which paths are important.
4. We perform interprocedural range analysis on the transformed graph.
5. We attempt to eliminate duplicated code when there was no benefit to range analysis. This can help eliminate code growth.

This paper describes algorithms and presents experimental results for the approach to profile-driven optimization described above. Specifically, our work makes the following contributions:

1. [3] provides an elegant solution for duplicating intraprocedural paths based on an intraprocedural path profile; this paper generalizes that work by providing algo-

rithms that take a program supergraph (an interprocedural control-flow graph) and an interprocedural path profile and produce an *express-lane supergraph*.
2. We show that interprocedural express-lane transformations yield benefits for range analysis: programs optimized using an interprocedural express-lane transformation and range analysis resolve (a) 0–7% more dynamic branches than programs optimized using the intraprocedural express-lane transformation and range analysis, and (b) 1.5–19% more dynamic branches than programs optimized using range analysis alone.
3. We show that by using range analysis instead of constant propagation, the intraprocedural express-lane transformation can lead to greater benefit than previously reported. We also show that code growth due to the intraprocedural express-lane transformation is not always detrimental to program performance.
4. Our experiments show that optimization based on an interprocedural express-lane transformation does benefit performance, though usually not enough to overcome the costs of the transformation. These results suggest that software and/or hardware support for *entry* and *exit* splitting may be a profitable research direction; entry and exit splitting are described in Section 3.1.

The remainder of the paper is organized as follows: Section 2 describes the relevant details of the interprocedural path-profiling techniques. Section 3 describes the interprocedural express-lane transformations. Section 4 presents experimental results. Section 5 describes related work.

2 Path Profiling Overview

To understand the interprocedural express-lane transformation, it is helpful to understand the interprocedural paths that are duplicated. This section summarizes the relevant parts of [10] and [11]. In these works, the Ball-Larus technique [4] is extended in several directions:

1. **Interprocedural vs. Intraprocedural:** [10] presents interprocedural path-profiling techniques in which the observable paths can cross procedure boundaries. Interprocedural paths tend to be longer and to capture correlations between the execution behavior of different procedures.
2. **Context vs. Piecewise:** In piecewise path profiling, each observable path corresponds to a path that may occur as a subpath (or piece) of an execution sequence. In context path profiling, each observable path corresponds to a pair $\langle C, p\rangle$, with an *active-suffix* p that corresponds to a subpath of an execution sequence, and a *context-prefix* C that corresponds to a context (*e.g.*, a sequence of pending calls) in which p may occur. A context path-profiling technique generally has longer observable paths and maintains finer distinctions than a piecewise technique.

In this paper, we use three kinds of path profiles: Ball-Larus path profiles (*i.e.*, intraprocedural piecewise path profiles) and the interprocedural piecewise and context path profiles of [10, 11]. (Our techniques could be applied to other types of path profiles.)

Interprocedural path profiling works with an interprocedural control-flow graph called a *supergraph*. A program's supergraph G^* consists of a unique entry vertex $Entry_{global}$, a unique exit vertex $Exit_{global}$, and a collection of control-flow graphs.

The flowgraph for procedure P has a unique entry vertex, Entry_P, and a unique exit vertex, Exit_P. The other vertices of the flowgraph represent statements and predicates in the usual way, except that each procedure call in the program is represented a *call* vertex and a *return-site* vertex. For each procedure call to procedure P (represented, say, by call vertex c and return-site vertex r), G^* contains a *call-edge*, $c \rightarrow \mathit{Entry}_P$, and a *return-edge*, $\mathit{Exit}_P \rightarrow r$. The supergraph also contains the edges $\mathit{Entry}_{global} \rightarrow \mathit{Entry}_{main}$ and $\mathit{Exit}_{main} \rightarrow \mathit{Exit}_{global}$.

As in the Ball-Larus technique, the observable paths in the interprocedural path-profiling techniques are not allowed to contain backedges. Furthermore, an observable path cannot contain a call-edge or return-edge from a recursive call-site. (Recursive call-sites are those that are the source of a backedge in the call graph.)

An observable path in an interprocedural context path profile may contain *surrogate edges*; surrogate edges are required because observable paths are not allowed to contain backedges. Unlike other edges in an observable path, a surrogate edge is not an edge in the supergraph. A surrogate edge $\mathit{Entry}_P - \rightarrow v$ in an observable path p represents an unknown path fragment q that starts at the entry vertex Entry_P of a procedure P and ends with a backedge to vertex v in procedure P. An observable path from an interprocedural path profiling technique may also contain *summary edges*. A summary edge connects a call vertex with its return-site vertex.

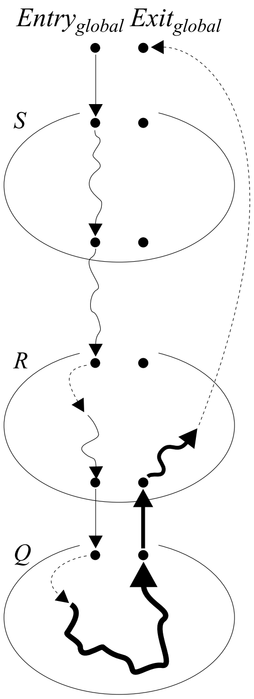

Fig. 1. Example of an interprocedural context path. The active-suffix is shown in bold and surrogate edges are shown using dashed-lines.

In the context path-profiling technique, a context-prefix is a sequence of path fragments in the supergraph, each fragment connected to the next by a surrogate edge. The context-prefix summarizes both the sequence of pending call-sites and some information about the path taken to each pending call-site. Fig. 1 shows a schematic of an observable path from an interprocedural context path profile.

Fig. 2 shows the average number of SUIF1 instructions in an observable path for several SPEC95 benchmarks. (For technical reasons discussed in [11], there are some situations where an interprocedural piecewise path is considered to have a context-prefix, cf. `m88ksim`, `li`, `perl`, and `vortex`.)

3 The Interprocedural Express-lane Transformation

The intraprocedural express-lane transformation takes a control-flow graph and an intraprocedural, piecewise path profile and creates an express-lane graph [3]. In this section, we describe how to extend this algorithm to take as input the program supergraph and an interprocedural path profile, and produce as output an express-lane supergraph.

There are several issues that must be addressed. The definition of an express-lane must be extended. In a context path profile, a path may consist of a non-empty context-

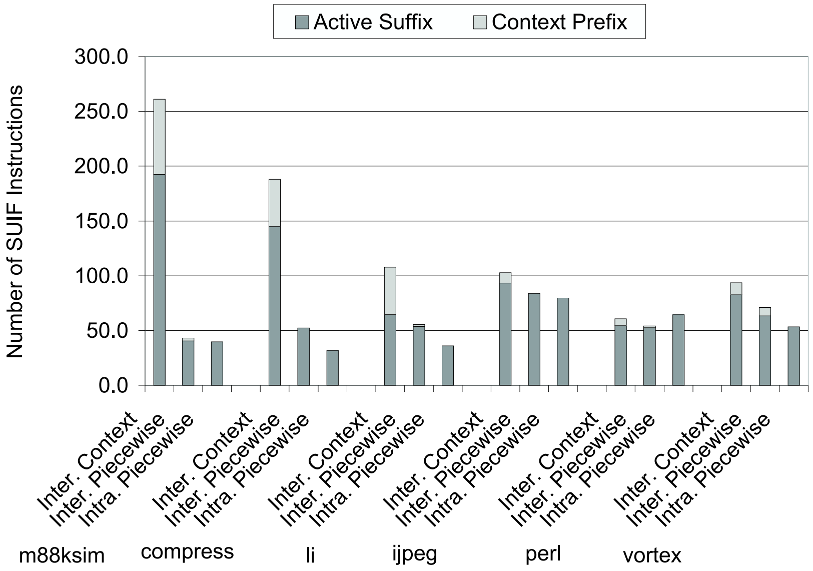

Fig. 2. Graph of the average number of SUIF instructions in an observable path for interprocedural context, interprocedural piecewise, and intraprocedural piecewise path profiles of SPEC95 benchmarks when run on their reference inputs. Each observable path was weighted by its execution frequency.

prefix as well as an active-suffix. Also, an observable path may contain "gaps" represented by surrogate edges. An express-lane version of an observable path may have a context-prefix and an active-suffix, and may have gaps just as the observable path does.

There are also technical issues that must be resolved. The interprocedural express-lane transformation requires a mechanism for duplicating call-edges and return-edges. We will use a straightforward approach that duplicates a call edge $c \to \mathit{Entry}_P$ by creating copies of c and Entry_P and duplicates a return edge $\mathit{Exit}_P \to r$ by creating copies of Exit_P and r.

Many modifications of the intraprocedural algorithm are required to obtain an algorithm for performing the interprocedural express-lane transformation. The Ammons-Larus express-lane transformation uses a *hot-path automaton* — a deterministic finite automaton (DFA) for recognizing hot-paths — and takes the cross product of this automaton with the control-flow graph (CFG), which can be seen as another DFA.

To create an automaton that recognizes a set of interprocedural hot-paths, we require a *pushdown* automaton (PDA). The supergraph can be seen as a second PDA. Thus, if we mimic the approach in [3], we would need to combine two pushdown automata, a problem that is uncomputable, in general. Instead, we create a collection of deterministic finite automata, one for each procedure; the automaton for procedure P recognizes hot-paths that start in P.

3.1 Entry and Exit Splitting

The algorithm for performing the interprocedural express-lane transformation uses entry splitting to duplicate call-edges and exit splitting to duplicate return-edges [5, 6].

Entry splitting allows a procedure P to have more than one entry. Exit splitting allows a procedure P to have multiple exits, each of which is assigned a number. Normally, when a procedure call is made, the caller provides a return address. In the case where a procedure has multiple exits, the caller provides a vector of return addresses. When the callee reaches the i^{th} exit vertex, it branches to the i^{th} return address. Our implementation uses a semantically equivalent but inferior method of entry (and exit) splitting: each call vertex sets an entry number before making a normal procedure call; the called procedure (calling procedure) then executes a switch on the entry (exit) number to jump to the proper entry (return) point.

3.2 Defining the Interprocedural Express-Lane

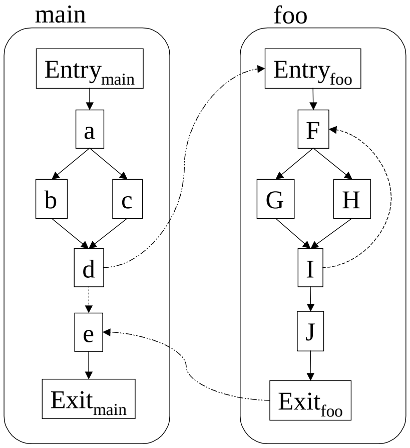

Fig. 3. Example supergraph.

In this section, we give a definition of an interprocedural express-lane. First we consider a simple example to develop intuition about what should happen when we duplicate an observable path from an interprocedural context path profile.

Example 1. Consider the supergraph shown in Fig. 3. Suppose we wish to create an express-lane version of the observable path $p = [Entry_{main} \to a \to b \to d \to Entry_{foo} - \to F \to H \to I]$ The context-prefix $[Entry_{main} \to a \to b \to d \to Entry_{foo}]$ indicates a path taken in main to the call-site on *foo*. The active-suffix of p is $[F \to H \to I]$. The principal difficulty in duplicating p has to do with the edge $Entry_{foo} - \to F$: this surrogate-edge appears in the middle of the observable path, but does not appear in the supergraph. What does it mean to duplicate this edge?

Peeking ahead, Fig. 9 shows an express-lane graph with an express-lane version of p. When we create an express-lane version of p, we create copies of the path's context-prefix and its active-suffix. The copy of the context-prefix ends at a copy $[Entry_{foo}, 4]$ of vertex $Entry_{foo}$. The copy of the active-suffix begins at a copy $[F, 8]$ of vertex F. We desire that any time execution reaches $[F, 8]$, it came along a path from $[Entry_{foo}, 4]$: we want to make sure that the duplicated active-suffix executes in the context of the duplicated context-prefix.□

We can now give a technical definition of an interprocedural express-lane: let G^* be a supergraph and let p be an observable path. Let H^* be a supergraph where every vertex of H^* is a copy of a vertex in G^*. Then an *express-lane* version of p is a sequence of vertices $[a_1, a_2, \ldots, a_n]$ in H^* such that the following properties are satisfied:

Duplication property: a_i is a copy of the i^{th} vertex in p.

Minimal predecessor property: A vertex a_i may have multiple predecessors if $a_i \equiv a_1$, or the $(i-1)^{th}$ edge of p is a surrogate edge, or a_i is a copy of a return-site vertex; otherwise a_i has exactly one predecessor, which is a_{i-1}. If a_i is a copy of return-site vertex r then let c be the call vertex associated with r:

  - If there is a copy of c in $[a_1 \ldots a_{i-1}]$, then a_i is associated with one call vertex, the last copy of c in $[a_1 \ldots a_{i-1}]$; otherwise, a_i may be associated with many call vertices.
  - If a_{i-1} is a copy of an exit vertex, then a_i is targeted by exactly one return-edge, $a_{i-1} \to a_i$. If a_i is a_1 or a_{i-1} is a copy of a call vertex, then a_i may be targeted by multiple return-edges.

Context property: For a vertex a_i in procedure P, if there is a copy of Entry_P in $[a_1 \ldots a_i]$, then a_i can reached by an intraprocedural path from the last copy of Entry_P in $[a_1 \ldots a_i]$ and not from any other copy of Entry_P.

These properties sometimes allow a vertex on an express-lane to have multiple predecessors (*i.e.*, there may be branches into the middle of an express-lane). This is necessary because: (1) a surrogate edge $u \to v$ does not specify a direct predecessor vertex of v in the supergraph; (2) a return-site vertex always has both an intraprocedural predecessor (the call site vertex) and an interprocedural predecessor.

3.3 Performing the Interprocedural Express-Lane Transformation

We now present two algorithms for performing the interprocedural express-lane transformation, one for interprocedural piecewise path profiles, and one for interprocedural context path profiles.

Our approach to constructing the express-lane supergraph consists of three phases:

1. Construct a family $\mathcal{A}$ of automata with one automaton A_p for each procedure P. The automaton A_P is specified as a DFA that recognizes (prefixes of) hot-paths that begin in P.
2. Use the Interprocedural Hot-path Tracing Algorithm (see below) to combine $\mathcal{A}$ with the supergraph G^* to generate an initial express-lane supergraph.
3. Make a pass over the generated express-lane supergraph to add return-edges and summary-edges where appropriate. This stage finishes connecting the intraprocedural paths created in the previous step.

The two algorithms for performing the interprocedural express-lane transformation differ slightly in the first step.

The Hot-path Tracing Algorithm treats the automata in $\mathcal{A}$ as DFAs, though technically they are not: an interprocedural hot path p may contain "gaps" that are represented by surrogate- or summary-edges. These gaps may be filled by *same-level valid paths*, or SLVPs; an SLVP is a path in which every return-edge can be matched with a previous call-edge, and vice versa. An automaton that recognizes the hot-path p requires the ability to skip over SLVPs in the input string, which requires a PDA. However, we can treat the hot-path automata as DFAs for the following reasons:

1. The automata in $\mathcal{A}$ have transitions that are labeled with summary-edges. A transition $(q_i, c \to r, q_j)$ that is labeled with a summary-edge $c \to r$ is considered to be an "oracle" transition that is capable of skipping over an SLVP in the input string. The oracle required to skip an SLVP is the supergraph-as-PDA.
2. When we combine a hot-path automaton with the supergraph, an oracle transition $(q_i, c \to r, q_j)$ will be combined with the summary-edge $c \to r$ of the supergraph to create the vertices $[c, q_i]$ and $[r, q_j]$ and the summary-edge $[c, q_i] \to [r, q_j]$ in the express-lane supergraph. The justification for this is that the set of SLVPs that an

oracle transition $(q_i, c \to r, q_j)$ should skip over is precisely the set of SLVPs that drive the supergraph-as-PDA from c to r.

Throughout the following sections, our examples use the program shown in Fig. 3.

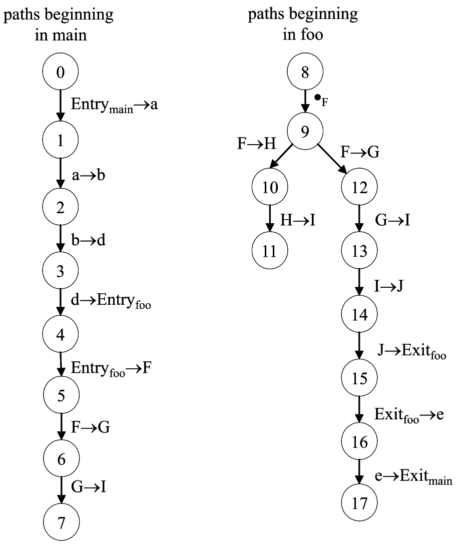

Fig. 4. Path trie for an interprocedural piecewise path profile of the supergraph in Fig. 3. For $i \in [4..15]$ and a backedge e in *foo*, $h(q_i, e) = q_9$; For $i \in [4..15]$ and a non-backedge e in *foo*, $h(q_i, e) = q_8$. For $i \in ([0..3] \cup [16..17])$ and an edge e in *main*, $h(q_i, e) = q_0$.

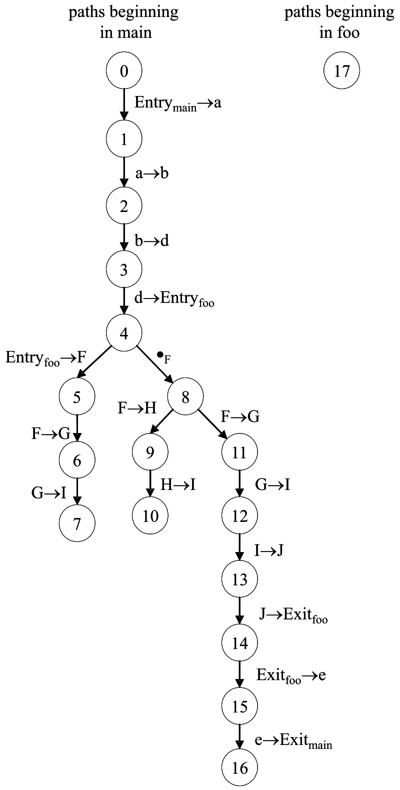

Fig. 5. Path trie for an interprocedural context path profile of the supergraph in Fig. 3. For $i \in ([0..3] \cup [15..16])$ and an edge e in *main*, $h(q_i, e) = q_0$. For $i \in [4..14]$ and a backedge e in *foo*, $h(q_i, e) = q_4$; for $i \in [4..14]$ and a non-backedge e in *foo*, $h(q_i, e) = q_8$. For q_{17} and any edge e in *foo*, $h(q_{17}, e) = q_{17}$.

The Hot-Path Automata for Interprocedural Piecewise Paths In this section, we show how to construct the set $\mathcal{A}$ of hot-path automata for recognizing hot interprocedural piecewise paths. We expand our definition of $\mathcal{A}$ to allow each automaton $A_P \in \mathcal{A}$ to transition to other automata in $\mathcal{A}$; thus, it is more accurate to describe $\mathcal{A}$ as one large automaton with several sub-automata.

As in [3], we build a hot-path automaton for recognizing a set of hot paths by building a trie A of the paths and defining a failure function that maps a vertex of the trie and a supergraph edge to another vertex of the trie [2]. We then consider A to be a DFA whose transition function is given by the edges of the trie and the failure function.

For each procedure P, we create a trie of the hot paths that start in P. Hot paths that can only be reached by following a backedge $u \to v$ are prefixed with the special symbol $\bullet_v$ before they are put in the trie. A transition that is labeled by $\bullet_v$ can match any backedge that targets v. Fig. 4 shows the path tries for the supergraph in Fig. 3 and the following paths:

$\mathit{Entry}_{main} \to a \to b \to d \to \mathit{Entry}_{foo} \to F \to G \to I$
$\bullet_F\ F \to H \to I$
$\bullet_F\ F \to G \to I \to J \to \mathit{Exit}_{foo} \to e \to \mathit{Exit}_{main}$

Every hot-path prefix corresponds to a unique state in a path trie. If a hot-path prefix ends at a vertex v and drives an automaton to state q, we say that q represents v; the root of the path trie for procedure P is said to represent Entry_P. The fact that q represents vertex v is important, since for a vertex $[v, q]$ in the express-lane supergraph, either $[v, q]$ is not on an express-lane and q represents an entry vertex, or q represents v.

As in [3], we define a failure function $h(q, u \to v)$ for a state q of any trie and an intraprocedural or summary-edge $u \to v$; the failure function is not defined for interprocedural edges. If q represents a vertex w of procedure P and $u \to v$ is not a backedge, then $h(q, u \to v) = \mathit{root\_trie}_P$, where $\mathit{root\_trie}_P$ is the root of the trie for hot paths beginning in P. If $u \to v$ is a backedge, then $h(q, u \to v) = q_{\bullet_v}$, where $q_{\bullet_v}$ is the target state in the transition $(\mathit{root\_trie}_P, \bullet_v, q_{\bullet_v})$; if there is no transition $(\mathit{root\_trie}_P, \bullet_v, q_{\bullet_v})$, then $q_{\bullet_v} = \mathit{root\_trie}_P$.

The later phases of the express-lane transformation make use of two functions, *LastActiveCaller* and *LastEntry*, which map trie states to trie states. For a state q that represents a vertex in procedure P, $\mathit{LastActiveCaller}(q)$ maps to the most recent ancestor of q that represents a call vertex that makes a non-recursive call to P. $\mathit{LastEntry}(q)$ maps to the most recent ancestor of q that represents Entry_P. $\mathit{LastActiveCaller}(q)$ and $\mathit{LastEntry}(q)$ are undefined for q if there is no appropriate ancestor of q in the trie.

The Hot-Path Automata for Interprocedural Context Paths The principal difference with the previous section is in how the failure function is defined. As above, a path trie is created for each procedure. Before a path is put into a trie, each surrogate edge $u \to v$ is replaced by an edge labeled with $\bullet_v$. As before, $\bullet_v$ matches any backedge that targets v. Fig. 5 shows the path tries for the supergraph in Fig. 3 and the paths:

$\mathit{Entry}_{main} \to a \to b \to d \to \mathit{Entry}_{foo} \to F \to G \to I$
$\mathit{Entry}_{main} \to a \to b \to d \to \mathit{Entry}_{foo}\ \bullet_F\ F \to H \to I$
$\mathit{Entry}_{main} \to a \to b \to d \to \mathit{Entry}_{foo}\ \bullet_F\ F \to G \to I \to J \to \mathit{Exit}_{foo} \to e \to \mathit{Exit}_{main}$

A state q that represents an entry vertex Entry_P corresponds to a hot-path prefix p that describes a calling context for procedure P. For this reason, states in the trie that represent entry vertices take on special importance in this section. Also, the map *LastEntry* will be important.

The maps *LastActiveCaller* and *LastEntry* are defined as in the last section. The failure function is defined as follows: if $u \to v$ is not a backedge, then $h(q, u \to v) = \mathit{LastEntry}(q)$. If $u \to v$ is a backedge, then $h(q, u \to v) = q'$, where q' is the state reached by following the transition labeled $\bullet_v$ from $\mathit{LastEntry}(q)$; if there is no such state, then $q' = \mathit{LastEntry}(q)$.

We now give some intuition for how the Hot-path Tracing Algorithm interacts with an automaton for interprocedural context paths. For any context-prefix p that leads to a procedure P, the Interprocedural Hot-path Tracing Algorithm may have to clone parts of P. This is required to make sure that the Context Property is guaranteed for express-lanes that begin with p (see Example 1). To accomplish this, the Hot-path Tracing Algorithm may generate many vertices $[x, q]$, where q is the automaton state in hot-path automaton A that corresponds to the context-prefix p: when the hot-path automaton A is in the state q and is scanning a path $[u \to v \to w \ldots]$ in procedure P that is cold in the context described by p, the automaton will stay in state q. Thus, the Interprocedural Hot-path Tracing Algorithm generates the path $[[u, q] \to [v, q] \to [w, q] \ldots]$. Only when the tracing algorithm begins tracing a path that is hot in the context of p does the hot-path automaton move out of state q.

Phase 2: Hot-Path Tracing of Intraprocedural Path Pieces This section describes the hot-path tracing algorithm that combines the family $\mathcal{A}$ of hot-path automata with the supergraph. A state q is a *reset state* if $h(q, u \to v) = q$ for some non-backedge $u \to v$. Reset states are important for several reasons: (1) a context-prefix p always drives a hot-path automaton to a reset-state; (2) for every vertex $[v, q]$ in the express-lane supergraph that is *not* part of an express-lane (*i.e.*, $[v, q]$ is part of residual, cold code), q is a reset state; and (3) for a reset state q and an express-lane supergraph vertex $[v, q]$, either v is an entry vertex represented by q, or $[v, q]$ is a cold vertex. We use these facts to determine whether an express-lane supergraph vertex $[v, q]$ is part of an express-lane.

Fig. 6 and 7 show the Interprocedural Hot-path Tracing Algorithm. The bulk of the work of the Interprocedural Hot-Path Tracing Algorithm is done by lines 19–21 of Fig. 6: these process each express-lane supergraph vertex $[v, q]$ that is not a call or exit vertex. This part of the algorithm is very similar to [3]: given an express-lane supergraph vertex $[v, q]$, a supergraph edge $v \to v'$ (which represents the transition $(v, v \to v', v')$ in the supergraph-as-PDA), and a transition $(q, v \to v', q')$, lines 19–21 “trace out” a new edge $[v, q] \to [v', q']$ in the express-lane supergraph. If necessary, a new vertex $[v', q']$ is added to the express-lane supergraph and the worklist W.

The Interprocedural Hot-Path Tracing Algorithm differs from its intraprocedural counterpart in the processing of call and exit vertices. Fig. 7 shows the function $ProcessCallVertex$ that is used to process a call-vertex $[c, q]$. $ProcessCallVertex$ has two responsibilities: (1) it creates call-edges from $[c, q]$; and (2) it must creates return-site vertices $[r, q']$ that could be connected to $[c, q]$ by a summary-edge in Phase 3 of the construction. If Phase 3 does not create the summary-edge $[c, q]$, then $[r, q']$ is unnecessary and will be removed from the graph in Phase 3.

Phase 3: Connecting Intraprocedural Path Pieces The third phase of the interprocedural express-lane transformation is responsible for completing the express-lane supergraph H^*. It must add the appropriate summary-edges and return-edges. Formally, this phase of the interprocedural express-lane transformation ensures the following:

```
For each call vertex $[c, q]$
    For each call-edge $[c, q] \to [Entry_P, q']$
        For each exit vertex $[Exit_P, q'']$ reachable from $[Entry_P, q']$ by an SLVP
            There must be a return-site vertex $[r, q''']$ such that
                1. There is a summary-edge $[c, q] \to [r, q''']$
                2. There is a return-edge $[Exit_P, q''] \to [r, q''']$
```

The algorithm for Phase 3 is given in [11], Section 7.3.4.

```
$G^*$ is the input supergraph.
$\mathcal{A}$ is a family of hot-path automata, with one automaton for each procedure in $G^*$
$A_P \in \mathcal{A}$ denotes the automaton for procedure $P$
$T_P$ denotes the transition relation of $A_P$
$\mathcal{T}$ is the disjoint union of all $T_P$
$root\_trie_{main}$ is the start state of $A_{main}$
$W$ is a worklist of express-lane supergraph vertices
$H^* \equiv (V, E)$ is the express-lane supergraph

**Main**()
    /* First, create all the vertices that might begin a hot-path */
    $V = \{Entry'_{global}, Exit'_{global}\}$
    **Foreach** procedure $P$
        $CreateVertex([Entry_P, root\_trie_P])$ /* See Figure 8 */
        **If** there is a transition $(root\_trie_P, \bullet_r, q')$ where $r$ is a return-site vertex
            /* For hot-paths that begin at return-sites, start the express-lane. */
            $CreateVertex([r, q'])$
    $E = \{Entry'_{global} \rightarrow [Entry_{main}, root\_trie_{main}]\}$

    **While** $W \neq \emptyset$
        $[v, q] = Take(W)$ /* select and remove an element from W */
        **If** $v$ is a call vertex
            $ProcessCallVertex([v, q])$
        **Else If** $v$ is an exit vertex
            **ForeachEdge** $v \rightarrow r$ in $G^*$
                /* $v \rightarrow r$ is a return-edge */
                **If** there is a transition $(q, v \rightarrow r, q') \in \mathcal{T}$.
                    $CreateVertex([r, q'])$ /* See Figure 8 */
        **Else**
            **ForeachEdge** $v \rightarrow v'$ in $G^*$
                Let $q'$ be the unique state such that $(q, v \rightarrow v', q') \in \mathcal{T}$.
                $CreateVertex([v', q'])$
                $E = E \cup \{[v, q] \rightarrow [v', q']\}$
    **Foreach** vertex $[Exit_{main}, q] \in V$
        $E = E \cup \{[Exit_{main}, q] \rightarrow Exit'_{global}\}$
**End Main**
```

Fig. 6. Interprocedural Hot-Path Tracing Algorithm.

4 Experimental Results

This section is broken into two parts. Section 4.1 discusses the effects of the various express-lane transformations on interprocedural range analysis. Section 4.2 presents experimental results on using the express-lane transformation and range analysis to perform program optimization.

4.1 Effects of the Express-Lane Transformation on Range Analysis.

We have written a tool in SUIF 1.3.0.5 called the Interprocedural Path Weasel (IPW) that performs the interprocedural express-lane transformation.[3] The program takes as input a set of C source files for a program P and a path profile *pp* for P. IPW first identifies the smallest subset pp' of *pp* that covers 99% of the SUIF instructions exe-

[3] The tool is named after, and based on, Glenn Ammons's tool Path Weasel, which performs the intraprocedural express-lane transformation [3].

```
**CreateVertex**($[v,q]$)
    **If** $[v,q] \notin V$
        $V = V \cup \{[v,q]\}$
        $Put(W, [v,q])$
**End CreateVertex**

**ProcessCallVertex**($[c,q]$) /* $c$ is a call vertex */
    Let $r$ be the return-site vertex associated with c
    /* Create call edges to all appropriate entry vertices */
    **ForeachEdge** $c \rightarrow Entry_P$
        /* v may have many callees if it is an indirect call-site */
        **If** $(q, c \rightarrow Entry_P, q') \in T$
            /* There is a hot path continuing from c along the edge $c \rightarrow Entry_P$ */
            $CreateVertex([Entry_P, q'])$
            $E = E \cup \{[c,q] \rightarrow [Entry_P, q']\}$
            Label $[c,q] \rightarrow [Entry_P, q']$ with "$(_{[c,q]}$"
        **Else**
            /* Hook up $[c,q]$ to a cold copy of $Entry_P$ */
            $CreateVertex([Entry_P, root\_trie_P])$
            $E = E \cup \{[c,q] \rightarrow [Entry_P, root\_trie_P]\}$
            Label the call-edge $[c,q] \rightarrow [Entry_P, root\_trie_P]$ with "$(_{[c,q]}$"
    /* Create every return-site vertex [r,q'] that could be needed in phase 3 */
    Let $q'$ be the unique state such that $(q, v \rightarrow r, q') \in T$
    $CreateVertex([r, q'])$
**End ProcessCallVertex**
```

Fig. 7. The procedures *CreateVertex* and *ProcessCallVertex* used in Fig. 6.

cuted.[4] Next, IPW performs the appropriate express-lane transformation on P, creating an express-lane version of each path in pp'. Finally, IPW performs interprocedural range analysis on the express-lane (super)graph.

The experiments with IPW were run on a 550 MhZ Pentium III with 256M RAM running Solaris 2.7. IPW was compiled with GCC 2.95.3 -O3. Each test was run 3 times, and the run times averaged. Cols. 3–5 of Table 1 compare the code growth and the increase in range-analysis time for the different express-lane transformations.

To evaluate the results of range analysis on a program P, we weighted each data-flow fact in vertex v by the execution frequency of v. Columns 6–8 of Table 1 compare the results of range analysis after the express-lane transformations have been performed. Three comparisons are made: the percentage of instruction operands that have a constant value; the percentage of instructions that have a constant result; and the percentage of *decided branches*, or conditional branch instructions that are determined to have only one possible outcome. In all cases, the interprocedural express-lane transformations do better than the intraprocedural express-lane transformation.

The range analysis we use allows the upper bound of a range to be increased once before it widens the upper bound to $(\text{MaxVal} - 1)$. Lower bounds are treated similarly. Our range analysis is similar to Wegman and Zadeck's conditional constant propagation [14] in that (1) it simultaneously performs dead code analysis and (2) it uses conditional branches to refine the data-flow facts.

[4] The value 99% was arrived at experimentally; duplicating more paths does not cause a greater benefit for range analysis, but it does cause a significant increase in code growth [11].

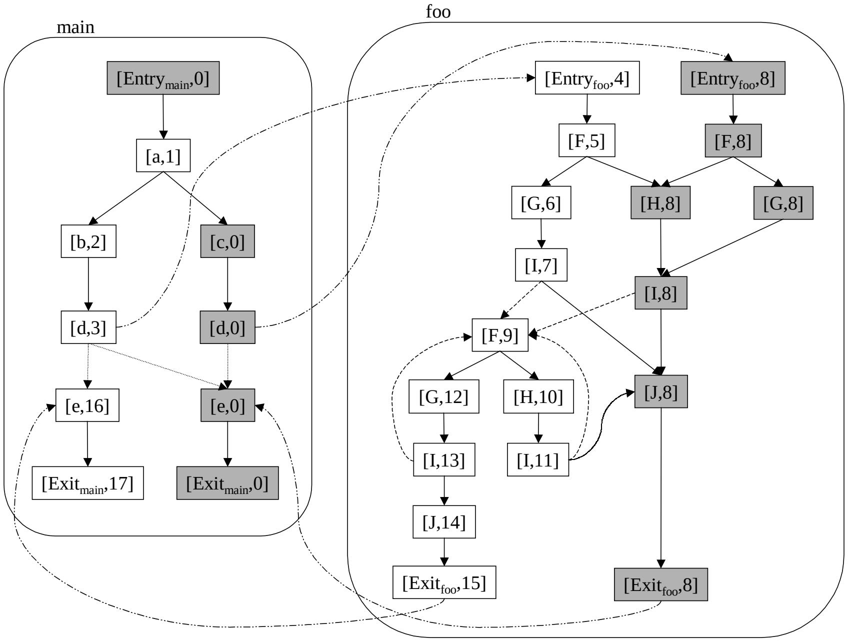

Fig. 8. Express-lane supergraph for the supergraph in Fig. 3 and the hot-path automaton in Fig. 4. Most of the graph is constructed during Phase 2 of the construction. The edges $[d,3] \to [e,16]$, $[d,3] \to [e,16]$, $[d,3] \to [e,0]$, $[d,0] \to [e,0]$, and $[\mathit{Exit}_{foo},8] \to [e,0]$ are added during Phase 3. Each shaded vertex $[v,q]$ has a state q that is a reset state; except for $[\mathit{Entry}_{main},0]$ and $[\mathit{Entry}_{foo},4]$, these are cold vertices.

4.2 Using the Express-Lane Transformation for Program Optimization

As mentioned in the introduction, it is possible to reduce the express-lane graph while preserving "valuable" data-flow facts. We used three different reduction strategies:

1. Strategy 1 preserves data-flow facts that determine the outcome of a conditional branch. Strategy 1 is based on the Coarsest Partitioning Algorithm [1, 11].
2. Strategy 2 preserves all data-flow facts. Strategy 2 is based on the Coarsest Partitioning Algorithm and the Edge Redirection Algorithm given in [11].
3. Strategy 3 is similar to Strategy 2, but only preserves data-flow facts that decide conditional branches (as in Strategy 1).

[11] contains more details, and more discussion of the trade-offs between these strategies. Fig. 10 compares the amount of reduction achieved by these strategies.

Tables 2 through 4 show the results of using various forms of the express-lane transformation together with Range Analysis to optimize SPEC95Int benchmarks. Specifically, we followed these steps:

1. Perform an express-lane transformation.
2. Perform interprocedural range analysis on the express-lane (super)graph.
3. Reduce the express-lane (super)graph.
4. Eliminate decided branches and replace constant expressions.
5. Emit C source code for the transformed program.
6. Compile the C source code using GCC 2.95.3 -O3.

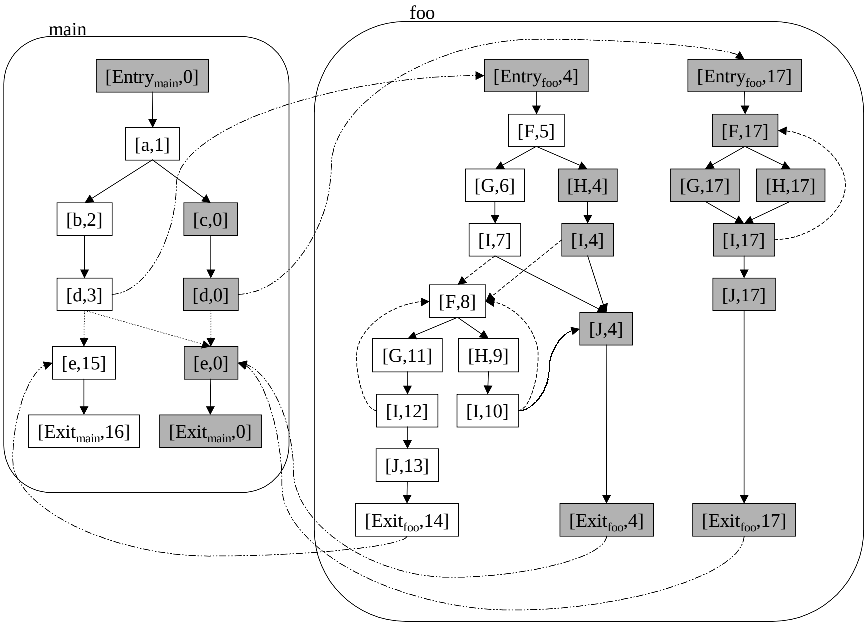

Fig. 9. Express-lane supergraph for the hot-path automaton in Fig. 5 and the supergraph in Fig. 3. Most of the graph is constructed during Phase 2. The edges $[d,3] \to [e,15]$, $[d,3] \to [e,0]$, $[d,0] \to [e,0]$, and $[Exit_{foo},17] \to [e,0]$ are added during Phase 3. Each shaded vertex $[v,q]$ has a state q that is a reset state; except for $[Entry_{main},0]$ and $[Entry_{foo},4]$, these are cold vertices.

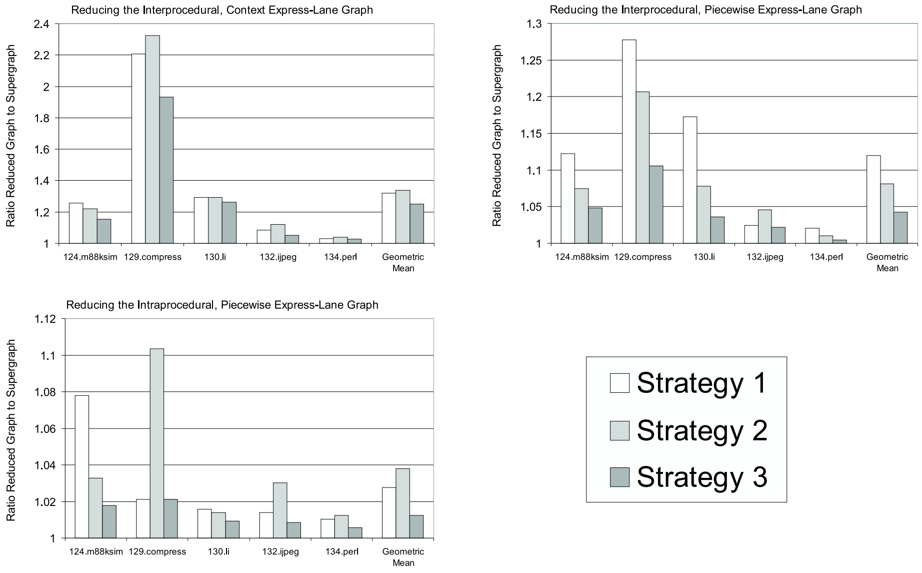

Fig. 10. Comparison of the strategies for reducing the express-lane supergraph.

Benchmark	E-Lane Transform.	Transform. Time (sec)	# Vertices in E-Lane Graph	Range Prop. Time (sec)	% const. operands	% const. results	% decided branches
124.m88ksim	Inter., Context	9.8	24032	569.5	28.5	33.1	19.7
	Inter., Piecewise	4.9	15113	508.4	28.6	33.2	20.0
	Intra., Piecewise	3.0	14218	734.2	27.7	32.3	17.5
	None	-	11455	300.8	25.9	31.1	0.8
129.compress	Inter., Context	1.4	2610	14.7	21.3	26.9	9.8
	Inter., Piecewise	0.3	1014	9.4	21.3	26.9	9.8
	Intra., Piecewise	0.2	696	10.2	21.3	26.2	2.2
	None	-	522	5.2	20.8	25.8	0.0
130.li	Inter., Context	12.9	23125	99.1	24.1	27.3	4.0
	Inter., Piecewise	5.3	11319	73.2	24.1	27.3	3.9
	Intra., Piecewise	1.9	7940	35.7	23.6	26.8	2.2
	None	-	7240	29.0	23.3	26.5	0.0
132.ijpeg	Inter., Context	13.0	18087	628.8	16.8	23.6	4.0
	Inter., Piecewise	8.5	13768	526.1	16.8	23.6	4.0
	Intra., Piecewise	7.1	12955	504.3	16.6	23.3	1.4
	None	-	12192	488.2	15.9	22.7	0.0
134.perl	Inter., Context	10.3	33863	713.8	24.3	28.8	3.3
	Inter., Piecewise	9.0	30189	655.2	24.2	28.8	3.0
	Intra., Piecewise	6.7	29309	718.6	24.1	28.7	2.8
	None	-	27988	573.9	23.0	28.5	1.3

Table 1. Columns 3–5 show a comparison of the (compile-time) cost of performing various express-lane transformations and the (compile-time) cost of performing interprocedural range analysis after an express-lane transformation has been performed; times are measured in seconds. Columns 6–8 show a comparison of the results of range analysis after various express-lane transformations have been performed.

		Reduction Strategy				
Benchmark	Base run time (sec)	None	Strategy 1	Strategy 1, No Step 4	Strategy 2	Strategy 3
124.m88ksim	146.70	-34.7%	-9.3%	-29.5%	-13.1%	-11.4%
129.compress	135.46	-14.0%	**1.0%**	-4.3%	**2.4%**	**2.0%**
130.li	125.81	-57.2%	-20.4%	-27.8%	-30.4%	-25.4%
132.ijpeg	153.83	-7.5%	-1.6%	-1.2%	-4.5%	-4.8%
134.perl	109.04	-21.3%	**4.9%**	**6.0%**	-3.1%	-3.0%

Table 2. Program speedups due to the **interprocedural, context** express-lane transformation and range propagation. For the base run times shown in Col. 2, the benchmarks were optimized by removing decided branches and constant expressions (but without any express-lane transformation) and then compiled using GCC 2.95.3 -O3.

		Reduction Strategy				
Benchmark	Base run time (sec)	None	Strategy 1	Strategy 1, No Step 4	Strategy 2	Strategy 3
124.m88ksim	146.70	-13.6%	-0.7%	-11.4%	**5.7%**	**5.4%**
129.compress	135.46	-14.0%	**0.5%**	-4.5%	-0.2%	**2.0%**
130.li	125.81	-68.1%	-26.7%	-40.1%	-11.4%	**2.5%**
132.ijpeg	153.83	-2.3%	-2.2%	-0.8%	-2.2%	-4.2%
134.perl	109.04	-19.4%	2.8%	2.7%	**6.1%**	**3.6%**

Table 3. Program speedups due to the **interproc., piecewise** express-lane trans. and range prop.

		Reduction Strategy				
Benchmark	Base run time (sec)	None	Strategy 1	Strategy 1, No Step 4	Strategy 2	Strategy 3
124.m88ksim	146.70	**10.6%**	**13.0%**	**1.2%**	**11.6%**	**7.4%**
129.compress	135.46	**6.4%**	**5.5%**	-2.1%	**2.1%**	**0.1%**
130.li	125.81	**8.1%**	**10.3%**	**7.2%**	-1.7%	-0.6%
132.ijpeg	153.83	**1.0%**	**0.7%**	-0.1%	-1.6%	-2.0%
134.perl	109.04	**9.7%**	**10.0%**	**6.3%**	**9.9%**	**5.4%**

Table 4. Program speedups due to the **intraproc., piecewise** express-lane trans. and range prop.

7. Compare the runtime of the new program with the runtime of the original program.

For a base case, we performed range analysis without any express-lane transformation (repeated as Col. 2 in Tables 2 through 4). We ran experiments with three different express-lane transformations. For each of the transformations, we tried the three reduction strategies listed above. We also ran experiments where we performed an express-lane transformation, then used Strategy 1 to reduce the express-lane (super)graph and then skipped Step 4 above. The reported run time is always the average of three runs.

The best results were for the intraprocedural express-lane transformation (Tables 4). The intraprocedural express-lane transformation together with the range analysis optimizations has a benefit to performance even when no reduction strategy is used to limit code growth. In fact, aggressive reduction strategies can destroy the performance gains. There are several possible reasons for this:

1. GCC may be able to take advantage of the express-lane transformation to perform its own optimizations (*e.g.*, code layout [7]).
2. Reduction of the hot path graphs may result in poorer code layout that requires more unconditional jumps along critical paths [12].
3. The more aggressive reduction strategies seek only to preserve decided branches, and may destroy data-fbw facts that show an expression to have a constant value.
4. The code layout for the reduced graph may interact poorly with the I-cache.

The results shown in Tables 4 and 5 are often (but not always) negative. There are two likely reasons for this:

1. It would have been difficult to modify an x86 code generator or a hardware simulator to support entry and exit splitting; instead, we used a straightforward implementation in software. This incurred overhead on each procedure entry and exit.
2. There is a significant increase in code growth.

Col. 4 of Tables 4 and 5 (and 6) show the performance overhead incurred by the transformations. Fig. 10 shows reasonable code growth for the interprocedural express-lane transformation, and we assume that most of the performance degredation is due to entry and exit splitting. Using the reduction strategies with the interprocedural express-lane transformations usually helps performance. (Graph reduction may eliminate the need for entry and exit splitting.) With aggressive reduction, the interprocedural piecewise express-lane transformation usually leads to performance gains (see Col. 6 of Table 3).

It should also be noted that the interprocedural express-lane transformations combined with the range-analysis optimizations do have a strong positive impact on program performance, although it is usually not as great as the costs incurred by the transformations. This can be seen in the experiments where we did not eliminate branches and replace constants: cf. Columns 3 and 4 of the Tables 4 and 5. (In those few cases where performance showed a slight improvement, we assume there was a change in code layout that had instruction cache effects.) This suggests that software and/or hardware support for entry and exit splitting would be a profitable research direction.

5 Related Work and Conclusions

The work in this paper is an interprocedural extension of the work in [3]. This paper and [3] are related to other work that focuses on improving the performance of particular program paths. A partial list of such works includes [8, 7, 9, 13, 6, 15]. A more detailed discussion of related work can be found in [11]. As stated in the introduction, the interprocedural express-lane transformation differs from other techniques in the literature on one or more of the following points:

1. We duplicate interprocedural paths before performing analysis.
2. We guide path duplication using interprocedural path profiles.
3. We perform interprocedural range analysis on the transformed graph.
4. We eliminate duplicated code when there was no benefit to range analysis.

We have shown that the interprocedural express-lane transformations have a beneficial effect on interprocedural range analysis. The performance gains from the interprocedural express-lane transformation are slight or negative — but we have shown that it has potential. Specifically, we have shown that a greater percentage of dynamic branches can be decided statically, and that performance improvements are likely with a better hardware and/or software implementation of entry and exit splitting.

References

1. A. V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.
2. Alfred V. Aho. *Algorithms for finding patterns in strings*, chapter 5, pages 255–300. MIT Press, 1994.
3. G. Ammons and J. Larus. Improving data-flow analysis with path profiles. In *PLDI98*.
4. T. Ball and J. Larus. Efficient path profiling. In *MICRO 1996*, 1996.
5. R. Bodik, R. Gupta, and M.L. Soffa. Interprocedural conditional branch elimination. In *PLDI'97*.
6. Rastislav Bodik. *Path-sensitive, value-flow optimizations of programs*. PhD thesis, University of Pittsburg, 2000.
7. P.P. Chang, S.A. Mahlke, and W.W. Hwu. Using profile information to assist classic code optimizations. *Software practice and experience*, 1(12), Dec. 1991.
8. J. A. Fisher. Trace scheduling: A technique for global microcode compaction. In *IEEE Trans. on Computers*, volume C-30, pages 478–490, 1981.
9. R.E. Hank. *Region-Based Compilation*. PhD thesis, UIUC, 1996.
10. D. Melski and T. Reps. Interprocedural path profiling. In *CC99*.
11. D.G. Melski. *Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation*. PhD thesis, University of Wisconsion, 2002.
12. F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. In *PLDI92*.
13. M. Poletto. Path splitting: a technique for improving data flow analysis, 1995.
14. M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. In *POPL85*.
15. Reginald Clifford Young. *Path-based Compilation*. PhD thesis, Harvard University, 1998.