Cost Effective Dynamic Program Slicing*

Xiangyu Zhang Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
{xyzhang,gupta}@cs.arizona.edu

ABSTRACT

Although dynamic program slicing was first introduced to aid in user level debugging, applications aimed at improving software quality, reliability, security, and performance have since been identified as candidates for using dynamic slicing. However, the dynamic dependence graph constructed to compute dynamic slices can easily cause slicing algorithms to run out of memory for realistic program runs. In this paper we present the design and evaluation of a cost effective dynamic program slicing algorithm. This algorithm is based upon a dynamic dependence graph representation that is highly compact and rapidly traversable. Thus, the graph can be held in memory and dynamic slices can be quickly computed. A compact representation is derived by recognizing that all dynamic dependences (data and control) need not be individually represented. We identify sets of dynamic dependence edges between a pair of statements that can *share* a single representative edge. We further show that the dependence graph can be transformed in a manner that increases sharing and sharing can be performed even in the presence of aliasing. Experiments show that transformed dynamic dependence graphs explicitly represent only 6% of the dependence edges present in the full dynamic dependence graph. When the full graph sizes range from 0.84 to 1.95 Gigabytes in size, our compacted graphs range from 20 to 210 Megabytes in size. Average slicing times for our algorithm range from 1.74 to 36.25 seconds across several benchmarks from SPECInt2000/95.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Debuggers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Testing tools, Tracing*

General Terms

Algorithms, Measurement, Performance

Keywords

dynamic dependence graph, debugging, testing

*Supported by grants from Intel, IBM, Microsoft, and NSF grants CCR-0324969, CCR-0220262, CCR-0208756, CCR-0105535, and EIA-0080123 to the Univ. of Arizona.

PLDI'04, June 9–11, 2004, Washington, DC, USA.

1. INTRODUCTION

The notion of program slicing was first proposed by Mark Weiser [25]. He introduced program slicing as a debugging aid and gave the first *static slicing* algorithm. Since then a great deal of research has been conducted on static slicing and a survey of many of the proposed techniques and tools can be found in [23] and [12]. For C programs that make extensive use of pointers, the highly conservative nature of static data dependency analysis leads to highly imprecise and considerably larger program slices. Since the main purpose of slicing is to identify the subset of program statements that are of interest for a given application, conservatively computed large slices are clearly undesirable. Recognizing the need for accurate slicing, Korel and Laski proposed the idea of *dynamic slicing* [14]. The dependences that are exercised during a program execution are identified and a precise dynamic dependence graph is constructed. Dynamic program slices are constructed upon user's requests by traversing the dynamic dependence graph. It has been shown that the dynamic slices can be considerably smaller than static slices [24, 12].

The importance of dynamic slicing extends well beyond debugging of programs [2, 15]. Increasingly applications aimed at improving software quality, reliability, security, and performance are being identified as candidates for making automated use of dynamic slicing. Examples of these applications include: detecting spyware that has been installed on systems without the user's knowledge [13], carrying out dependence based software testing [8, 16], measuring module cohesion for purpose of code restructuring [11], and guiding the development of performance enhancing transformations based upon estimation of criticality of instructions [28] and identification of instruction isomorphism [22]. The benefit of using dynamic slicing is shown in Table 1. Each dynamic slice contains a subset of statements that are executed at least once. The number of unique statements that are executed at least once (USE) and average dynamic slice size (SS) are given. SS contains 2.46 to 56.08 times fewer statements than USE.

While the notion of dynamic slicing is very useful for the above mentioned applications [2, 13, 8, 16, 11], an impediment to their widespread use in practice has been the high cost of computing them. Consider the cost data in Table 1. For program runs involving execution of 67 to 220 million statements, the sizes of dynamic dependence graphs required to carry out dynamic slicing take 0.84 to 1.95 Gigabytes of storage. Given the large sizes of these graphs, it is not possible to keep them in memory. While for small program runs it may be possible to maintain dynamic dependences in memory and use them in dynamic slicing, for realistic program runs this is not possible. To address this problem recently we proposed the LP algorithm in [26] where the dynamic dependence graph is constructed *on-demand* in response to dynamic slicing requests from

Table 1: Cost of dynamic slicing.

Benchmark	Suite	Statements Executed (Millions)	Benefit			Costs	
			Unique Stat. Exec. (USE)	Av. Slice Size (SS)	USE/ SS	Full Graph Size (MBs)	LP's Average Slicing Times (Min.)
300.twolf	SPECInt2000	140.63	15,955	5,789	2.75	1,568.44	13.98
256.bzip2	SPECInt2000	67.19	1,420	25	56.08	1,296.14	9.19
255.vortex	SPECInt2000	108.37	70,920	18,587	3.82	1,442.66	10.17
197.parser	SPECInt2000	123.00	2,942	583	5.04	1,816.95	9.91
181.mcf	SPECInt2000	118.57	2,179	858	2.54	1,535.84	12.32
164.gzip	SPECInt2000	71.05	4,474	336	13.32	834.74	4.69
134.perl	SPECInt95	220.08	21,984	3,588	6.13	1,954.40	25.21
130.li	SPECInt95	124.91	10,215	2,849	3.59	1,745.72	11.28
126.gcc	SPECInt95	131.24	151,420	26,436	5.73	1,534.37	12.04
099.go	SPECInt95	138.15	49,577	20,158	2.46	1,707.36	10.67

the execution trace that is saved on disk. While this approach greatly reduces the size of dynamic dependence graph held in memory, the on-demand construction of the dynamic dependence graph is quite slow since it requires repeated traversals of the trace stored on disk. To enable faster traversal of the trace, we augmented the trace with summary information which allowed us to skip irrelevant parts of the trace during traversal. As shown by data in Table 1, we found that even after enabling faster traversal, across the different benchmarks, on an average it took 4.69 to 25.21 minutes to compute a single dynamic slice.

A dynamic slicing algorithm would be cost effective if the dynamic dependence graphs could be compacted so that they are small enough to hold in memory and the design of the compacted graphs is such that they can be rapidly traversed to compute dynamic slices. One approach proposed by researchers sacrifices the precision of dynamic data dependences to construct a dynamic dependence graph that is greatly reduced in size [1]. However, recent work has shown that algorithms that sacrifice precision in order to limit the graph size are ineffective as they can produce slices that are many times larger than accurate slices [26, 19].

In this paper we present a practical dynamic slicing algorithm which is based upon a novel representation of the dynamic dependence graph that is highly compact and rapidly traversable. The contributions of this paper include the following key ideas on which the design of our algorithm is based and the experimental evaluation of the algorithm.

Sharing a dependence edge across multiple dynamic instances of a data dependence. In general, it is not merely sufficient to remember whether a pair of statements was involved in a dynamic (data or control) dependence. For computing dynamic slices it is also necessary to remember the specific execution instances of the statements that are involved in a dynamic dependence. We identify conditions under which we do not need to remember the execution instances of statements involved in a dependence. Thus, a single representative edge can be shared across all dynamic instances of an exercised dependence. In particular, in these situations there is an one-to-one correspondence between *all* execution instances of a pair of statements involved in a dependence because the statements involved are *local* to the same basic block. In presence of *aliasing*, multiple definitions of a variable may reach a use even if the definitions and use are local to a basic block. We show that in such situations *partial* sharing is possible.

Transformations for increasing sharing. It is possible to construct a transformed dynamic dependence graph in a manner that converts *non-local* dependence edges into *local* dependence edges and therefore increases the amount of sharing. First we show that in some situations *non-local def-use* dependence edges can be replaced by *local use-use* edges. Second we show that by performing *path specialization* we can convert *non-local def-use* dependence edges into *local def-use* dependence edges. To limit the increase in static code size due to path specialization, we apply this transformation selectively in a *profile guided* fashion. Third we show that in the presence of aliasing, through *node specialization*, full local sharing can be achieved.

Experimental evaluation. Our experimental evaluation shows that once sharing of edges is achieved, the number of dependence edges is reduced to roughly 6% of total edges. When the full graph sizes range from 0.84 to 1.95 Gigabytes in size, our corresponding compacted graphs range from 20 to 210 Megabytes in size. Average slicing times for our algorithm range from 1.74 to 36.25 seconds across the benchmarks studied while average slicing times of the LP algorithm range from 4.69 to 25.21 minutes.

The remainder of the paper is organized as follows. Section 2 presents the unoptimized dynamic slicing algorithm. In section 3 we identify the conditions that give rise to *sharable* dependence edges, develop transformations for increasing sharing, and present algorithm details. Experimental results are presented in section 4. Related work and conclusions are given in sections 5 and 6.

2. DYNAMIC SLICING USING FULL DYNAMIC DEPENDENCE GRAPH

We first begin by describing the dynamic dependence graph representation used to capture the dynamic data and control dependences exercised during program execution. Once we have described this representation we will present a series of optimizations that lead to a compact representation whose edges can be rapidly traversed during slicing.

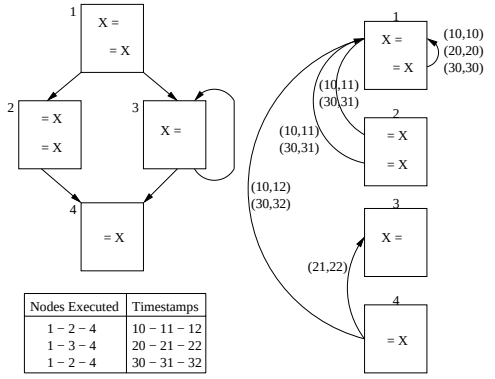
When the program begins execution, the dynamic dependence graph (dyDG) merely consists of a fixed number of nodes corresponding to the basic blocks in the program. As the program executes, the graph is transformed by introducing edges for the dynamically exercised control and data dependences. Since the same dependence may be exercised many times, the edge is labeled with additional information to uniquely identify the execution instances of the statements which are involved in the dependence. Execution instances are identified by generating timestamps. A global timestamp value is maintained and each time a basic block is executed, it is assigned a new timestamp value generated by incrementing the current timestamp value. Each execution of a statement (also every definition and use) in the basic block is uniquely identified by this

timestamp. Thus, dynamic dependences exercised can be captured by introducing the following dependence edges:

Data dependence edge $d_m u_n(t_m, t_n)$ represents the *use* of a value during execution of statement u_n with timestamp value t_n that was *defined* during the execution of statement d_m with timestamp value t_m. We will refer to the subgraph of dyDG showing dynamic data dependence edges as the dyDDG.

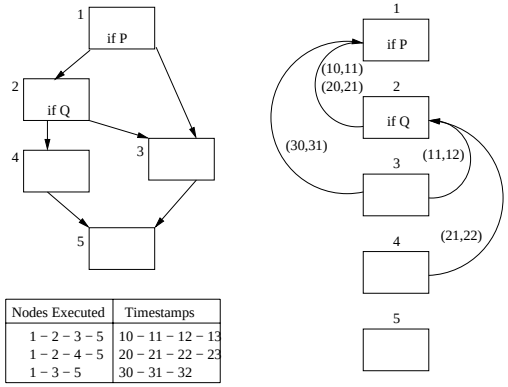
Control dependence edge $c_m d_n(t_m, t_n)$ represents the execution of statement d_n with timestamp value t_n that was *control dependent* upon execution of conditional statement c_m with timestamp value of t_m. We refer to the subgraph of dyDG showing dynamic control dependences as the dyCDG.

Let us consider examples of dyDG's. We present two examples, one to illustrate dyDDG and the other to illustrate dyCDG. The control flow graph in Fig. 1(a) represents a function that contains static dependences due to definitions and uses of variable x. Let us assume that this function is invoked three times during execution and the paths taken in each of the executions and the timestamps assigned to the basic block executions are as shown. The corresponding dyDDG contains nodes for the basic blocks and edges for data dependences where the labels on the edges uniquely identify the dynamically exercised data dependences. Similarly the control flow graph in Fig. 1(b) represents a function that is invoked three times with execution histories as shown. The corresponding dyCDG containing labeled control dependence edges is shown.



Nodes Executed	Timestamps
1 – 2 – 4	10 – 11 – 12
1 – 3 – 4	20 – 21 – 22
1 – 2 – 4	30 – 31 – 32

(a) CFG; Sample execution history; and dyDDG.



Nodes Executed	Timestamps
1 – 2 – 3 – 5	10 – 11 – 12 – 13
1 – 2 – 4 – 5	20 – 21 – 22 – 23
1 – 3 – 5	30 – 31 – 32

(b) CFG; sample execution history; and dyCDG.

Figure 1: Examples of dyDDG and dyCDG.

For enabling program slicing, for each variable (data address) v, we remember the statement s and its corresponding timestamp t_s that most recently assigned a value to v. The program slice for v can be computed by starting at execution instance $s(t_s)$ and traversing the dyDG backwards following the relevant dynamic data and control dependence edges. Recall that if execution instance $s(t_s)$ is dependent upon execution instance $s'(t_{s'})$, an edge from $s(t_s)$ to $s'(t_{s'})$ denoted by $s's(t_{s'}, t_s)$ is present in the dyDG. All statements (including s) that are visited during this traversal are added to the dynamic slice. This computation, denoted as $Slice(s(t_s))$, is expressed below:

$$Slice(s(t_s)) = \{s\} \cup \bigcup_{\forall s's(t_{s'}, t_s)} Slice(s'(t_{s'})).$$

Let us consider the space and time costs of dynamic slicing briefly. The *space* needed to hold dynamic dependence graph of a given program in general cannot be statically bounded across all executions since as the program continues to execute, it continues to exercise dependences which must be captured by the graph. Thus, it is clear that space is spent on labeling the edges and the lists of labels continue to grow as the program executes. The *time* spent on dynamic slicing is simply the time spent on traversing the graph. In general it is clear that this time is not bounded across all executions because it depends on the number of dynamic dependence edges in the graph which cannot be bounded across all program executions. However, it is also clear that the time for traversing a single edge can be significant as we must examine labels of all outgoing edges to locate the relevant edge for traversal. Therefore in order to limit the space and time costs of dynamic slicing we must develop techniques for the following:

- To save *space* we must develop techniques that will lead to the saving of fewer dynamic labels (i.e., timestamp pairs).
- To save *time* we must develop techniques that avoid searching through large number of dynamic labels to locate the relevant dynamic dependence edge to follow.

In the next section we present a series of optimizations that reduce the number of labels that need to be stored in the dyDG. Once these optimizations are presented, it will also become clear that after optimization, the traversal of edges can be carried out in a much more time efficient manner.

3. OPTIMIZED DYNAMIC DEPENDENCE GRAPH REPRESENTATION

While in general we need to remember all dynamic instances of all dependences, next we show that all dynamic instances need not be remembered explicitly. We develop optimizations that have the effect of eliminating timestamp pairs. These optimizations can be divided into the the following three categories:

Infer - Static edge is introduced for a dependence and the timestamp pairs corresponding to the dynamic instances of the dependence are *inferred* and thus need not be explicitly remembered.

Transform - While some timestamp pairs cannot be inferred from the original static dependence graph, transformations can be applied to the static graph so that the timestamp pairs can be inferred from the transformed graph. Thus, these transformations enable inferring of timestamp pairs.

Redundant - There are situations in which different dependence edges are guaranteed to have identical timestamp pairs. Redundant copies of timestamp pairs can thus be discarded.

3.1 dyDDG Optimizations

Given an execution instance of a use $u(t_u)$, during dynamic slicing, we need to find the corresponding execution instance of the relevant definition $d(t_d)$. There are two steps to this process: (finding d) in general many different definitions may reach the use but we need to find the relevant definition for $u(t_u)$; and (finding t_d) even if the relevant definition d is known we need to find the execution instance of d, i.e. $d(t_d)$, that computes the value used by $u(t_u)$. The following optimizations show how the above determinations can be made even in the absence of some timestamp pairs.

3.1.1 (OPT-1) Infer

(OPT-1a) Infer Local Def-Use for Full Elimination.
Consider a definition d and a use u that are *local* to the same basic block, d appears before u, and there is no definition between d and u that can ever prevent d from reaching u. In this case there is one-to-one correspondence between execution instances of d and u. Since d and u belong to the same basic block, the timestamps of corresponding instances are *always the same*, i.e. given a dynamic data dependence $du(t_d, t_u)$ it is always the case that $t_d = t_u$. Therefore, given the use instance $u(t_u)$, the corresponding d is known statically and the corresponding execution instance is simply $d(t_u)$. Thus we do not need to remember dynamic instances individually – it is enough to introduce a static edge from u to d.

In the dynamic slicing algorithm based upon the full dependence graph, we began with a set of nodes (basic blocks) and introduced all dependence edges dynamically. To take advantage of the above optimization we simply introduce the edge from u to d statically prior to program execution. No new information will be collected or added at runtime for the use u as the edge from u to d does not need any timestamp labels. In other words all dynamic instances of def-use edge from u to d are statically replaced by a single shared representative edge.

The impact of this optimization is illustrated using the dyDDG of Fig. 1(a). As shown in Fig. 2, basic block 1 contains a labeled local def-use edge which is replaced by a static edge that need not be labeled by this optimization. We draw static edges as dashed edges to distinguish them from dynamic edges.

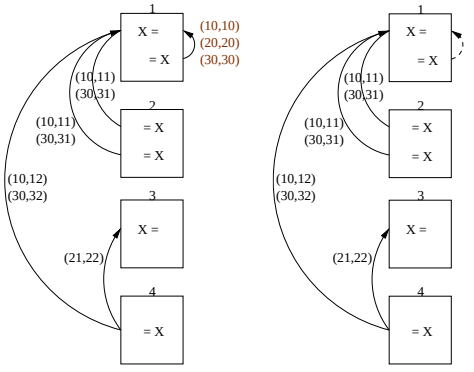


Figure 2: Effect of applying OPT-1a.

(OPT-1b) Infer Local Def-Use for Partial Elimination.
In the above optimization it was important that certain subpath was free of definitions of the variable involved (say v) so that a dependence edge involving v that is free of labels could be used. In programs with pointers, the presence of a definition of a *may alias* of v may prevent us from applying the optimization even though at runtime this definition may rarely redefine v. To enable the application of preceding optimization in presence of definitions of may aliases of v we proceed as follows. We introduce a static unlabeled edge from one definition to its potential use. If at runtime another may alias turns out to truly refer to v, additional dynamic edges labeled with timestamp pairs will be added. The effect of this optimization is that the timestamp labels corresponding to the statically introduced data dependence are eliminated while the labels for the dynamically introduced data dependence edge are not, i.e. labels have been *partially eliminated*.

During traversal, first the labels on dynamic edges are examined to locate the relevant dependence. If the relevant dependence is not found, then it must be the case that the dependence involved corresponds to the static edge which can then be traversed. It should also be clear that greater benefits will result from this optimization if the edge being converted to an unlabeled edge is the more *frequently exercised* dependence edge. Thus, if *profile data* is available we can make use of it in applying this optimization.

In the example shown in Fig. 3 let us assume that $*P$ is a may alias of X and $*Q$ is a may alias of Y. Further assume that the code fragment is executed twice resulting in the introduction of the following labeled dynamic edges: between the uses of X and definitions of X and $*P$; and between the uses of Y and the definitions of Y and $*Q$. We introduce the following static unlabeled edges: from the use of X to the definition of X (as in OPT-1a); and later the use of Y to the earlier use of Y (as in OPT-2b described later). The dynamic edges introduced are: from the use of X to the definition of $*P$; and from the later use of Y to the definition of $*Q$. Thus some, but not all, labels have been removed.

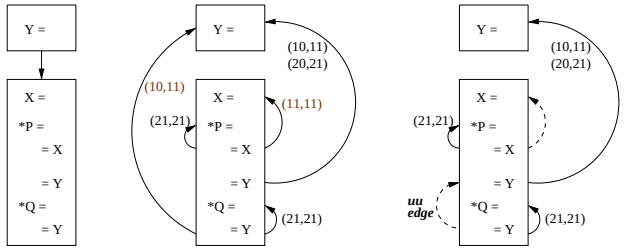


Figure 3: Effect of applying OPT-1b.

3.1.2 (OPT-2) Transform

(OPT-2a) Transform Local Def-Use for Full Elimination.
While the above optimization was able to achieve partial elimination of labels, next we present an optimization that can eliminate all of the labels present in situations with aliasing. Full elimination of labels is achieved through *specialization*. Given a use of variable v in a node (basic block) that is reachable by two distinct definitions (say d_1 and d_2) that may define v, we create two copies of the node. One copy is used to exclusively represent dynamic dependences between d_1 and the use of v while the other copy is used to represent only the dynamic dependences between d_2 and use of v. Since in each copy of the node the use of v is always data dependent upon the same definition point of v, we do not need to maintain the timestamp labels on these edges.

Consider the example shown in Fig. 4. One use of X is reached by the definition of X in statement $X = f(Y)$ while the second use of X is reached by the definition of X in statement $*P = g(Z)$. By making two copies of the basic block that contains the two definitions and the use, we are able to introduce static edges to represent both of the above dependences and thus the labels corresponding to these edges are eliminated. Note that the dependence edges cor-

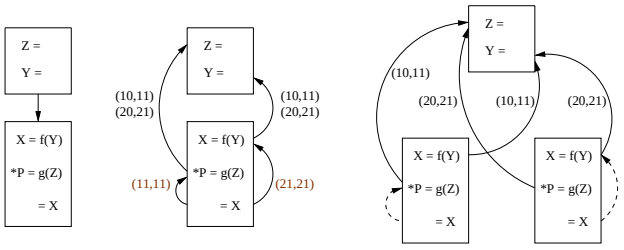


Figure 4: Effect of applying OPT-2a.

responding to the uses of Y and Z in the basic block must also be replicated and appropriately labeled.

In the above example, two copies of the node were sufficient to eliminate the local labels. In general, if uses of multiple variables have multiple definitions due to aliasing, we will require greater number of copies to be created to eliminate all of the local labels. If the list of labels is very long, node replication may be justified. However, if there are only few labels, partial elimination may be preferable to full elimination.

Since the above optimizations show that timestamp labels on local dependences edges can be eliminated, we further develop optimizations that convert non-local dependence edges into local dependence edges. Once non-local dependence edges have been converted to local dependence edges, their labels can be eliminated using the above optimizations.

(OPT-2b) Transform Non-Local Def-Use to Local Use-Use. Consider two uses u_1 and u_2 such that u_1 and u_2 are *local* to the same basic block, u_1 and u_2 always refer to the same location during any execution of the basic block, and there is no definition between u_1 and u_2 that can cause the uses to see different values. Now let us assume that a non-local definition d reaches the uses u_1 and u_2. In this case each time u_1 and u_2 are executed, two non-local def-use edges $du_1(t_d, t_{u_1})$ and $du_2(t_d, t_{u_2})$ are introduced. Let u_1 appear before u_2. We can replace the non-local def-use edge $du_2(t_d, t_{u_2})$ by a local use-use edge u_1u_2. The latter does not require a timestamp label because t_{u_1} is always equal to t_{u_2}. By replacing a non-local def-use edge by a local use-use edge, labels on the edge are eliminated. During slicing an extra edge (the use-use edge) will be traversed. Moreover, use-use edges are treated differently. In particular, a statement visited by traversing a use-use edge is not included in the dynamic slice.

Using static analysis we can identify uses local to basic blocks which always share the same reaching definition. Once having identified these uses we statically introduce use-use edges from later uses to the earliest use in the basic blocks. After having introduced these edges, there will not be any need to collect or introduce any dynamic information corresponding to the later uses.

The impact of this optimization is illustrated by further optimizing the dyDDG obtained by applying OPT-1a. As shown in Fig. 5, basic block 2 contains a two uses of X each having the same reaching definition from block 1. The labeled non-local def-use edge from the second use to the definition is replaced by an unlabeled static use-use edge by this optimization. We draw use-use edge using a dashed edge to indicate it is static and further indicate that it is a use-use edge.

(OPT-2c) Transform Non-Local Def-Use to Local Def-Use. Given non-local def-use edge $du(t_d, t_u)$ between basic blocks b_d and b_u, by creating a specialized node for the *path* (say p) that when executed *always establishes* the def-use edge $du(t_d, t_u)$ (i.e.,

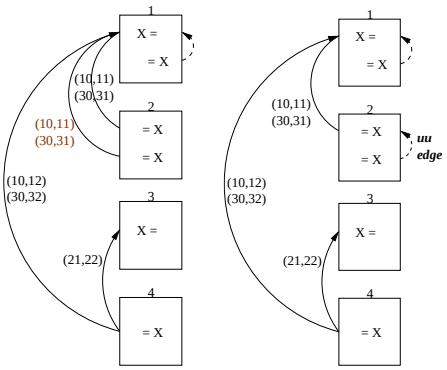


Figure 5: Effect of applying OPT-2b.

d cannot be killed along p prior to reaching u), we can convert this non-local dynamic edge into a local dynamic edge $du(t'_d, t'_u)$ for path p. While for the original edge $du(t_d, t_u)$ the values of t_d and t_u are not equal, for the modified edge $du(t'_d, t'_u)$ the values of t'_d and t'_u are equal. At runtime if the dependence between d and u is established along path p, then that dependence would be represented by an unlabeled edge local to node for path p. However, if the dependence is established along some path other than p, it is represented using a labeled non-local edge between b_d and b_u.

The consequence of earlier optimizations was that the initial graph that we start out with contains some statically introduced data dependence edges. The consequence of this optimization is that instead of starting out with a graph that contains only basic block nodes, we start out with a graph that contains additional nodes corresponding to paths that have been specialized. During execution we must detect when specialized paths are executed (we will present an algorithm to do so in section 3.3). This is necessary for construction of the dyDG due to the following reasons. The value of global timestamps must be incremented after the execution of code corresponding to a node in the graph. Thus, we no longer will increment the timestamp each time a basic block is executed because nodes representing specialized paths contain multiple basic blocks. At runtime we must distinguish between executions of a block that correspond to its appearance in a specialized path from the rest of its executions so that when we introduce a dynamic data dependence edge in the graph we know which copy of the block to consider.

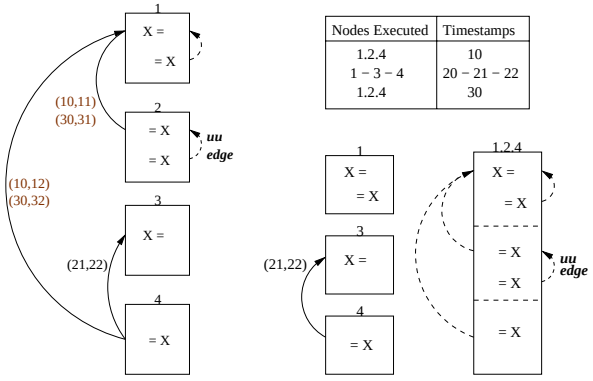


Figure 6: Effect of applying OPT-2c.

The impact of this optimization is illustrated by further optimizing the optimized dyDDG from Fig. 5. As shown in Fig. 6, if we create a specialized node for path along basic blocks 1, 2 and 4,

many of the previously dynamic non-local def-use edges are converted to dynamic local def-use edges within this path. The def-use edges established along this path can now be statically introduced within the statically created node representing this path. Thus, the timestamp labels for these def-use edges are no longer required. Since block 2 can only be executed when path 1-2-4 is executed, we do not need to maintain a separate node for 2 once node for path 1-2-4 has been created. However, the same is not true for blocks 1 and 4. Therefore we continue to maintain nodes representing them to capture dynamic dependences that are exercised when path 1-2-4 is not followed.

After applying multiple optimizations to the dyDDG of Fig. 1(a), we have eliminated all but one of the labels in the dyDDG. In fact this label can also be eliminated by creating another specialized node for path containing blocks 3 and 4.

Finally it should be noted that the above optimization only eliminates labels corresponding to dependence instances exercised along the path for which a specialized node is created. Thus, greater benefits will be derived if the path specialized is a *frequently executed path*. As a result, selection of paths for specialization can be based upon *profile data*.

3.1.3 (OPT-3) Redundancy

(OPT-3) Redundancy Across Non-Local Def-Use Edges. In all the optimizations considered so far we have identified and created situations in which the labels were guaranteed to have a pair of identical timestamps. Now we present an optimization which identifies pairs of dynamic edges between different statements that are guaranteed to have identical labels in all executions. Thus, the statements can be clustered so that they can share the same edge and thus a single copy of the list of labels. Given basic blocks b_d and b_u such that definitions d_1 and d_2 in b_d have corresponding uses u_1 and u_2 in b_u. If it is guaranteed that along every path from b_d to b_u either both d_1 and d_2 will reach u_1 and u_2 or neither d_1 nor d_2 will reach u_1 and u_2, then the labels on the def-use edges d_1u_1 and d_2u_2 will always be identical. The example in Fig. 7 shows that the uses of Y and X always get their definitions from the same block and thus dependence edges for Y and X can share the labels. A shared edge between clusters of statements (shown by dashed boxes) is introduced by this optimization.

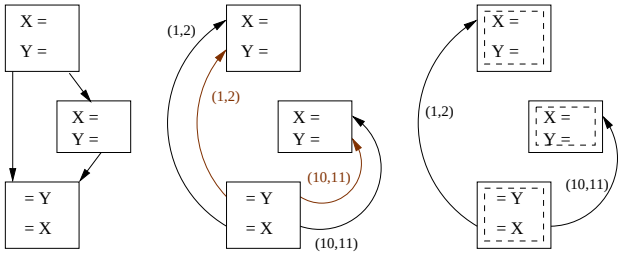


Figure 7: Effect of applying OPT-3.

3.2 dyCDG Optimizations

Control dependences are introduced at the granularity of basic blocks. Next we present the optimizations that enable introduction of static unlabeled control dependence edges.

3.2.1 (OPT-4) Infer

(OPT-4) Infer Fixed Distance Unique Control Ancestor. Often basic blocks (nodes) in a control flow graph have a unique control ancestor. Whenever a node is control dependent upon a unique conditional predicate, the control dependence edge can be introduced statically. In addition, sometimes the difference in the timestamps corresponding to a dynamic control dependence is a compile time constant. Thus, we can remember the difference value and avoid labeling the edge with a timestamp pair each time the dependence is exercised. In particular, for a dynamic control dependence edge $cd(t_c, t_d)$ which satisfies the above conditions, $t_c + \delta = t_d$ because timestamp is incremented by δ whenever after the execution of the predicate when control transfers to the dependent basic block. When this optimization is applied to the example from Fig. 1(b), the δ values of edges from node 2 to node 1 and node 4 to node 2 are determined to be the value 1.

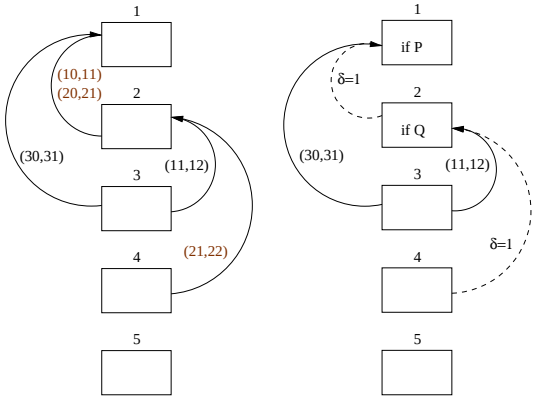


Figure 8: Effect of applying OPT-4.

3.2.2 (OPT-5) Transform

(OPT-5a) Transform Multiple Control Ancestors. If a node has multiple control ancestors, we can replicate the node creating specialized copies for each of the control ancestors. Static control dependence edges can now be introduced and their δ values can be remembered. The dynamic timestamp labels are no longer required. Continuing with the example from Fig. 8, the labeled edges corresponding to the two control ancestors of node 3 can be replaced by static edges after replicating 3 as shown in Fig. 9.

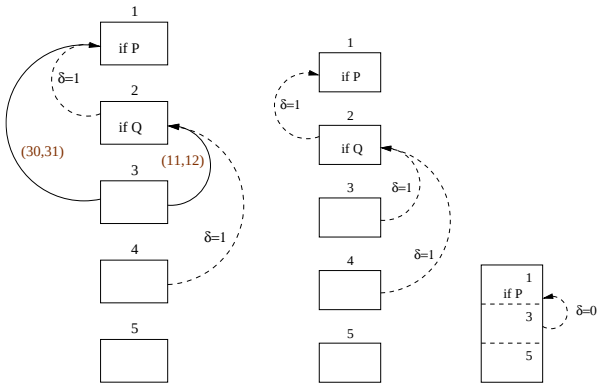


Figure 9: Effect of applying OPT-5a.

Specialization also enables another optimization for control dependences which is analogous to OPT-2b. Following specialization, a node representing a path may contain multiple basic blocks that are control equivalent [9]. Instead of using separate non-local edges for two control equivalent blocks, we can replace the non-local edge for the second block by a local edge which points to the first block.

(OPT-5b) Transform Varying Distance Unique Control Ancestor. In optimization OPT-4 we had shown how to handle the case when a node had a unique control ancestor which was at a constant distance from the node. It is possible that there are multiple paths from the control ancestor to the control dependent node causing the former to be at varying distances from the latter depending upon the path taken. In this case we can apply specialization to create copies of the dependent node such that each copy created is at a constant distance from the control ancestor.

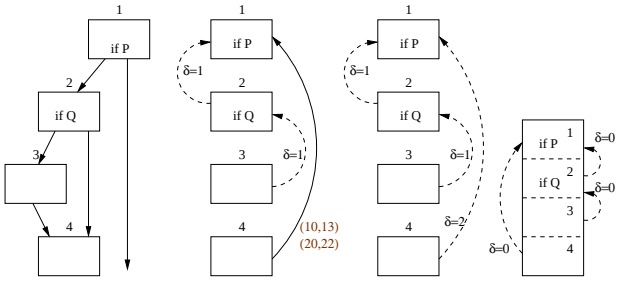


Figure 10: Effect of applying OPT-5b.

In Fig. 10, node 4 is at distance 3 from node 1 along path 1.2.3.4 and at distance 2 from node 1 along path 1.2.4. By specializing path 1.2.3.4 as shown in the figure we are able to convert the control dependence edge from 4 to 1 into a pair of control dependence edges that are each at constant distances of 2 and 0.

3.2.3 (OPT-6) Redundant

(OPT-6) Redundancy Across Non-Local Def-Use and Control Dependence Edges. In OPT-3 we showed how two non-local data dependence edges can share common labels. The same approach can be extended to allow a non-local control dependence edge to share labels with a non-local data dependence edge as long as these edges connect the same pair of blocks. An example illustrating this optimization is shown in Fig. 11.

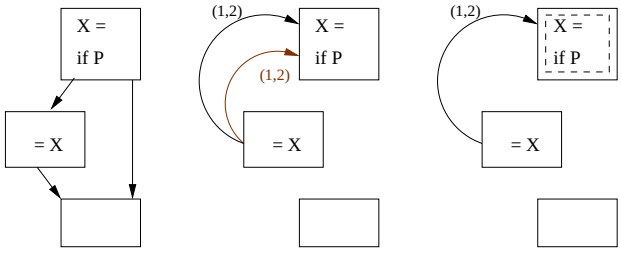


Figure 11: Effect of applying OPT-6.

3.3 Completeness of the Optimization Set

In an unoptimized dyDG any dependence edge may have a long list of labels attached to it. To compact the graph we may wish to apply transformations that can eliminate this list of labels. Given this requirement, it is important that we have available to us an optimization (or a series of optimizations) that can eliminate any list of labels. We consider a set of optimizations to be *complete* if for any given list of labels, we can find a sequence of optimizations in the optimization set that can be used to eliminate the list of labels. The *completeness* property of the optimization set is important because it tells us that we have sufficient optimizations and do not need to continue developing additional ones. In fact we can say that given an optimization set that is complete, it is possible to convert any labeled dyDG into one which has no timestamp pair labels.

[Theorem] (Completeness). The set of optimizations OPT-1 through OPT-6 is *complete*.

[Proof] There are two types of edges in the dyDG, data dependence and control dependence. Lets us consider each of the edge types and show that a list of labels associated with an edge can be eliminated using the optimizations described.

Data dependence labels. (Local Edge) If the labels are associated with an edge that is local to a basic block the labels can be always removed because either they can be *inferred* and hence OPT-1a is applicable or they can be *entirely converted* to labels that can be inferred by carrying out specialization using OPT-2a. (Non-local Edge) If the labels are associated with an edge that is non-local, i.e. it connects two different basic blocks, then it can always be *converted into a local edge* by applying path specialization using OPT-2c. Once it has been converted to a local edge, its labels can always be eliminated as described above. Thus, we conclude that labels associated with all data dependence edges can be eliminated by using the optimizations we provide .....(1)

Control dependence labels. (Fixed Distance from Unique Ancestor) If a node is at a fixed distance from its control ancestor, then the labels can be *inferred* and hence optimization OPT-4 is applicable. (Others) If the node has multiple control ancestors and/or it is at a varying distance from its control ancestors, then path specialization using optimizations OPT-5a and OPT-5b can always be applied to *convert* the labels into ones that can be inferred. Thus, we conclude that labels associated with all control dependence edges can be eliminated using the optimizations we provide.....(2)

From (1) and (2) we conclude that the optimization set we have developed is *complete*. □

It is worth noting that in the above proof no reference was made to optimizations OPT-1b, OPT-2b, OPT-3, and OPT-6. These optimizations are not needed for completeness. They are provided as cheaper alternatives to specialization in situation where they may be found to be applicable.

3.4 dyDG Construction and Dynamic Slicing

Static Component of dyDG. To construct the static component of dyDG we need to perform the following analyses: (i) *reaching definitions* analysis is carried out to compute *def-use* information. *May alias* information is needed to carry out this analysis; (ii) *reaching uses* analysis is carried out to compute *use-use* information; (iii) *simultaneous reachability* analysis is carried out to identify situations in which a pair of non-local data dependence edges can *share labels*; and (iv) *postdominator analysis* is carried out to compute *control dependences* [9]; and (v) *must reachability* analysis is carried out to identify situations in which a pair of non-local data and control dependence edges can *share labels*.

Except for *simultaneous reachability* analysis all other analyses are standard. Therefore next we describe the details of the simultaneous reachability analysis. Given a pair of definitions d_1 and d_2 in block s, with corresponding uses u_1 and u_2 in block d, the edges d_1u_1 and d_2u_2 will share identical labels if and only if whenever s and then d are executed either both data dependences are exercised or neither of them are exercised. We consider the subgraph consisting of s, d, and all nodes along paths from s to d – this is similar to the way chops are computed [21]. We refer to set of nodes in this subgraph excluding s as $reach(sd)$. $KILL_n$ is a two bit value where bits correspond to the two definitions; bit value of 1 indicates that n does not kill the definition while 0 indicates that n kills the definition. The following equations compute for each node in $reach(sd)$ a data flow value which is $\subseteq \{11, 10, 01, 00\}$.

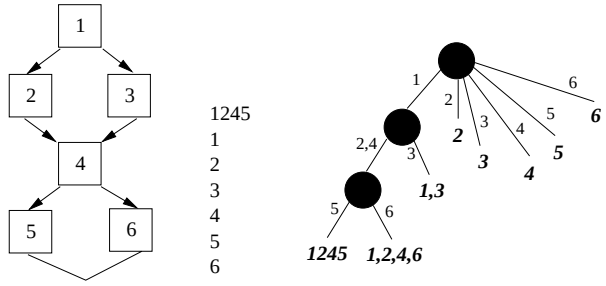
$$\forall n \in succ(s) \bigcap reach(sd), \;\; x_n = \{11\}$$
$$\forall n \in reach(sd) - succ(s),$$
$$x_n = \bigcup_{p \in pred(n) \cap reach(sd)} \{KILL_p \wedge x : x \in x_p\}$$

If the solution for node d is $\{11\}$ (i.e., both definitions always reach d) or $\{11, 00\}$ (either both definitions reach d or neither reaches d), then the two dependence edges will always have identical labels. On the other hand, if the solution contains 10 (01), then there is a path from s to d along which d_1 (d_2) reaches d but d_2 (d_1) does not reach d. This analysis does not need to be carried out for every pair of definitions but rather for those that appear in the same basic block and have corresponding uses in the same basic block. Moreover, transitivity can be used to further reduce the pairs considered (i.e, if (d_1u_1, d_2u_2) can share labels and (d_2u_2, d_3u_3) can share labels, then so can (d_1u_1, d_3u_3)).

Given the results of the above analyses, we have enough information available to construct the static component of dyDG. However, we observe that the static component of dyDG must be constructed once and then used repeatedly to capture dynamic dependence histories of different program runs. In other words the optimizations we have developed must be applied to construct the static component. While many of the optimizations can be applied for every opportunity that exists, there is a subset of optimizations that must be applied selectively. In particular all of the specialization based optimizations should be applied only if we expect that their application will result in more compaction than the graph expansion that is caused by specialization. Therefore we should apply these optimizations in a profile guided fashion. We specialized all Ball Larus paths [3] that were found to have a non-zero frequency during a profiling run. This approach works well because nearly all of the optimizations requiring specialization, are actually based upon path specialization. There are two optimizations that require data dependence profiles – OPT-1b and OPT-2a. Our implementation does not make use of data dependence profiles yet. Instead we applied OPT-1b such that data dependence edges created due to must aliases were given priority for partial elimination over edges due to may aliases. We do not apply OPT-2a because we do not have an effective static heuristic for applying OPT-2a.

Dynamic Component of dyDG. As the program executes, it sends a trace of one basic block at a time to an online algorithm which builds the dyDG. This online algorithm must carry out two tasks. First it must buffer the basic block traces until it is determined which node in the static dyDG must be augmented with additional dynamic edges. This is necessary because there may be multiple copies of a basic block due to specialization. Second it maintains the timestamp value and uses it to create the labels corresponding to the dynamic edges.

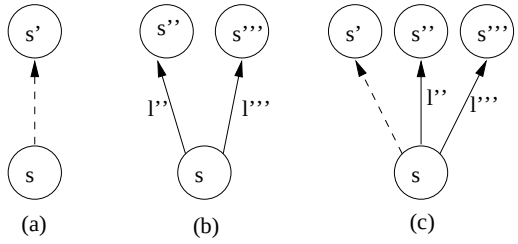
Consider the example shown in Fig. 12. Let us assume that for the CFG shown the static graph constructed has nodes for each of the basic blocks and another node for path 1245 is created due to specialization. When the program executes and generates a trace for block 1, we cannot at this time introduce dependence edges for statements in 1 because we do not know where to introduce these edges – in copy of statements of 1 in node 1 or node 1245. The trace must be buffered till it is clear that either the program has followed path 1245 or that it has taken some other path. To detect when it is the right time to introduce edges we can construct the tree shown in Fig. 12(c). The online algorithm is initially at the root of the tree. Depending upon the basic block executed, the appropriate edge labeled with that block is traversed and the trace is buffered. When we reach a leaf, it is time to process the buffered trace. The leaf is labeled with the list of nodes in the dyDG from which the edges introduced will originate. For example if basic blocks 1, 2, 4, and 5 are executed the edges originate from node 1245 while if blocks 1, 2, 4, and 6 are executed the edges originate from nodes 1, 2, 4, and 6.



(a) CFG (b) Nodes (c) Find and update tree.

Figure 12: Introducing dynamic edges.

Dynamic Slicing. During the computation of a dynamic slice, the dyDG is traversed backwards to identify the statements that belong to the slice. The set of dependence edges E_s going backwards from a statement s can be partitioned into subsets of edges E_{u_s} corresponding to each use u_s and subset of edges E_{c_s} corresponding to all control ancestors of s. In other words, $E_s = \bigcup_{\forall u_s} E_{u_s} \bigcup E_{c_s}$. Given an execution instance of s, say $s(t_s)$, for each subset of edges corresponding to a dependence in E_s (i.e., E_{u_s} or E_{c_s}), we need to locate the specific edge $s's$ in E_s that must be followed. Moreover, since the edge $s's$ may have been exercised many times, we must identify the specific dynamic instance of this edge $s's(t_{s'}, t_s)$ that is involved in the dependence.



(a) (b) (c)

Figure 13: Traversing dependence edges.

There are three situations that arise as shown in Fig. 13. Let us say we are considering a subset of edges $E_{d(s)}$ from E_s due to a dependence d involving s (i.e., $E_{d(s)}$ corresponds to some E_{u_s} or E_{c_s}). In the first case, $E_{d(s)}$ contains a single static edge $s's$ which is thus not labeled dynamically with timestamp pairs. The traversal is straightforward as there is only one choice and the timestamp $t_{s'}$ in $s's(t_{s'}, t_s)$ can be easily determined ($t_{s'} = t_s$ for data dependences and $t_{s'} = t_s - \delta$ for control dependences). In the second case there are multiple dynamic and thus labeled edges (say $s''s$ and $s'''s$). The labels on these edges (l'' and l''') must be searched to locate the relevant edge and its instance – $s''s(t_{s''}, t_s)$ or $s'''s(t_{s'''}, t_s)$. In the third case, there is a single unlabeled static edge $s's$ as well as multiple labeled dynamic edges (say $s''s$ and $s'''s$). The labels on $s''s$ and $s'''s$ (i.e., l'' and l''') are first searched. If we find the relevant dependence $s''s(t_{s''}, t_s)$ or $s'''s(t_{s'''}, t_s)$, we are done; otherwise we select the static edge $s's$ and compute the value of timestamp $t_{s'}$ in $s's(t_{s'}, t_s)$ as discussed in the first case.

It is worth noting that removal of explicit timestamps, as is carried out by the series of optimizations we have developed, not only makes the dependence graph more compact, it also speeds up the traversal process as fewer timestamps are searched to locate the relevant timestamp. The first and third cases contain a static unlabeled edge and hence the search is reduced while the second case represents the situation in which no reduction in search is achieved as all dynamic labels are saved and hence potentially searched.

We have described the key points of the traversal process. Now we summarize our dynamic slicing algorithm. In order to enable computation of slices, for each variable v we maintain the triple $< s, n, t_s >$ such that v was last defined by statement s in node n at time t_s. The dynamic slice for v is computed as shown below. Notice the manipulation of timestamps for unlabeled edges and also note that if $s's$ is a uu-edge then s' is not added to the slice. The sharing of labels between different edges is not explicitly reflected in the algorithm below since it is an implementation detail which affects how the timestamp labels on edges are accessed. In the algorithm below, $sSlice(s(t_s))$ represents the set of statements that belong to the dynamic slice of execution instance $s(t_s)$ and $eSlice(E, t_s)$ represents the subset of statements in the dynamic slice of execution instance $s(t_s)$ that are contributed by the traversal of the subset of dynamic edges E from $s(t_s)$.

$$Slice(v) = \{s\} \bigcup sSlice(s(t_s))$$

$$sSlice(s(t_s)) = \begin{cases} if\ E_s = \bigcup_{\forall u_s} E_{u_s} \bigcup E_{c_s} \neq \phi\ then \\ \quad \bigcup_{\forall u_s} eSlice(E_{u_s}, t_s) \bigcup eSlice(E_{c_s}, t_s) \\ else \\ \quad \phi \\ endif \end{cases}$$

$$eSlice(E, t_s) = \begin{cases} if\ \exists\ labeled\ edge\ s's(t_{s'}, t_s) \in E\ then \\ \quad sSlice(s'(t'_s)) \bigcup \{s'\} \\ elseif\ \exists\ unlabeled\ edge\ s's \in E\ then \\ \quad case\ s's\ is: \\ \quad du\ edge: \ sSlice(s'(t_s)) \bigcup \{s'\} \\ \quad uu\ edge: \ sSlice(s'(t_s)) \\ \quad cd\ edge: \ sSlice(s'(t_s - \delta_{s's})) \bigcup \{s'\} \\ endif \end{cases}$$

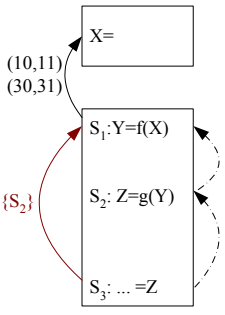


Figure 14: Using shortcuts.

Using Shortcuts to Speed Up Traversal. Finally we present an optimization that is used to speed up the traversal of the dyDG by the above slicing algorithm. As we have shown, our optimized algorithm introduces some dependence edges statically while others are introduced dynamically. It is possible that at some points in the dyDG multiple edges are traversed in sequence that are all static edges. If this is the case, the contributions to the dynamic slice when these edges are traversed is also known statically and always the same. Therefore, to speed up traversal of these edges, we introduce a *shortcut* edge that replaces the traversal of multiple static dependence edges by the traversal of a single shortcut edge. The edge is labeled with the set of statements that are skipped by the shortcut edge so that the dynamic slice can be appropriately updated when the shortcut edge is traversed. In the example shown in Fig. 14, corresponding to the sequence of two static edges $S_3 \rightarrow S_2 \rightarrow S_1$, we introduce a shortcut edge $S_3 \rightarrow S_1$ labeled with $\{S_2\}$.

4. EXPERIMENTAL RESULTS

We have implemented the algorithm described using the *Trimaran* compiler infrastructure which handles programs written in `C`. We make use of this infrastructure because we have implemented several other dynamic slicing algorithms using this system [26, 27]. The programs we use in our experiments include 6 programs from the `SPECInt2000` suite (the first 6 programs in Table 1) and 4 programs from the `SPECInt95` suite (the last 4 programs in Table 1). The reason we could not run remaining `SPECInt2000` programs is because they cannot be successfully compiled using the version of *Trimaran* being used. All experiments were performed on a workstation with 2.4 GHz Pentium IV and 2 GB RAM.

The goal of our experiments was to determine the space and time costs of our dynamic slicing algorithm which we will refer to as the optimized algorithm OPT. We also compare the performance of OPT with the traditional slicing algorithm (FP) for short program runs and the best overall algorithm (LP) for long program runs according to [26].

4.1 Performance Evaluation of OPT

Graph sizes. We measured the size of the full dynamic dependence graph and compared it against the size of the optimized graph obtained after application of all the optimizations described in this paper. These graph sizes are shown in Table 2. As we can see the graphs sizes are reduced by factors ranging from 7.46 to 93.40. As a result, while the full graph sizes range in size from 0.84 to 1.95 Gigabytes, the optimized graphs range from 20 to 210 Megabytes in size.

Table 2: dyDG size reduction.

Program	Graph Size (Megabytes)		Ratio
	Before	After	Before/After
`300.twolf`	1568.44	210.21	7.46
`256.bzip2`	1296.14	50.48	25.68
`255.vortex`	1442.66	64.81	22.26
`197.parser`	1816.95	69.81	26.03
`181.mcf`	1535.84	170.29	9.02
`164.gzip`	834.74	51.57	16.19
`134.perl`	1954.40	20.92	93.40
`130.li`	1745.72	96.50	18.09
`126.gcc`	1534.37	74.71	20.54
`099.go`	1707.36	131.24	13.01

The substantial reduction in the graph size is due to the fact that roughly only 6% of the dynamic dependences are explicitly maintained after the proposed optimizations are applied. The contributions of the various optimizations in reducing the graph size are shown in Fig. 15. Here 100% corresponds to the full graph size and `dyn` corresponds to the size of the graph after application of all the optimizations. The other points in the bar graph show how the size reduces as different categories of optimizations are applied one by one. As we can see, OPT-1 is very effective as it reduces graph

sizes to roughly 35% of the full graph size. However, the other optimizations also contribute significantly as they together reduce the graph size from 35% to 6% of the full graph size. It is important to point out that the distribution obtained is dependent upon the order in which the optimizations are applied since some cases can be handled by multiple optimizations.

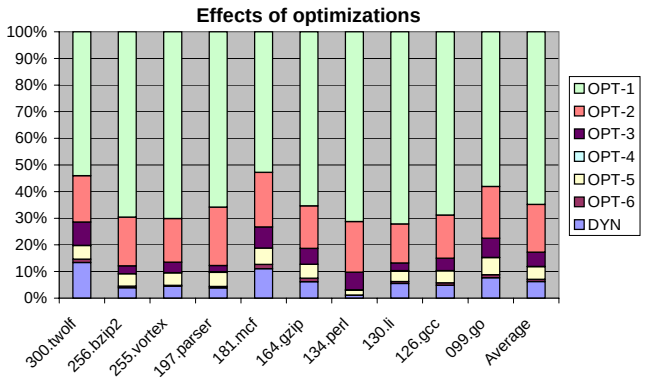


Figure 15: Effect of various optimizations on dyDG size.

We also observe that the majority of the savings comes from applying optimizations for dynamic data dependence edges. This is because the dynamic control dependences represent a small fraction of information contained in dyDG (see the first graph in Fig. 16). This is not surprising because in our implementation control dependence edges are introduced at basic block granularity while data dependence edges have to be introduced at statement granularity. The second and third graphs in Fig. 16 separately show the reductions in the sizes of dyDDG and dyCDG due to the application of optimizations. We further breakdown the contributions of the individual optimizations. Note that the second graph in Fig. 16 does not include OPT-2a because it was never applied.

In recent work, the SEQUITUR algorithm [20] has been effectively used to compress control flow traces [18] and address traces [5]. We also used the same algorithm to compress the labeling information in dyDGs. On an average, SEQUITUR compressed the dyDGs by a factor of 9.18 across the ten benchmarks we considered and our approach compressed the dyDGs by a factor of 23.4. Therefore the approach we propose is much more effective than SEQUITUR.

Execution times. The next step was to analyze the slicing times of our algorithm. To carry out this study we computed multiple program slices at various points during the execution of each program. The reason why we computed multiple slices is because depending upon the memory address or variable chosen, the slicing times can vary. The reason we carried out slicing at different points during execution is because we wanted to study the change in slicing times as the size of the dyDG grows. Moreover this scenario also represents a realistic use of slicing – applications often compute slices at different execution points.

The results of this study are presented in Fig. 17. In this graph each point corresponds to average slicing time for 25 slices. For each benchmark 25 new slices are computed after execution interval of 15 million statements – these slices correspond to 25 distinct memory references. Following each execution interval slices are computed for memory addresses that had been defined since the last execution interval – this was done to avoid repeated computation of same slices during the experiment. As we can see, the increase in slicing times is linear with respect to number of statements

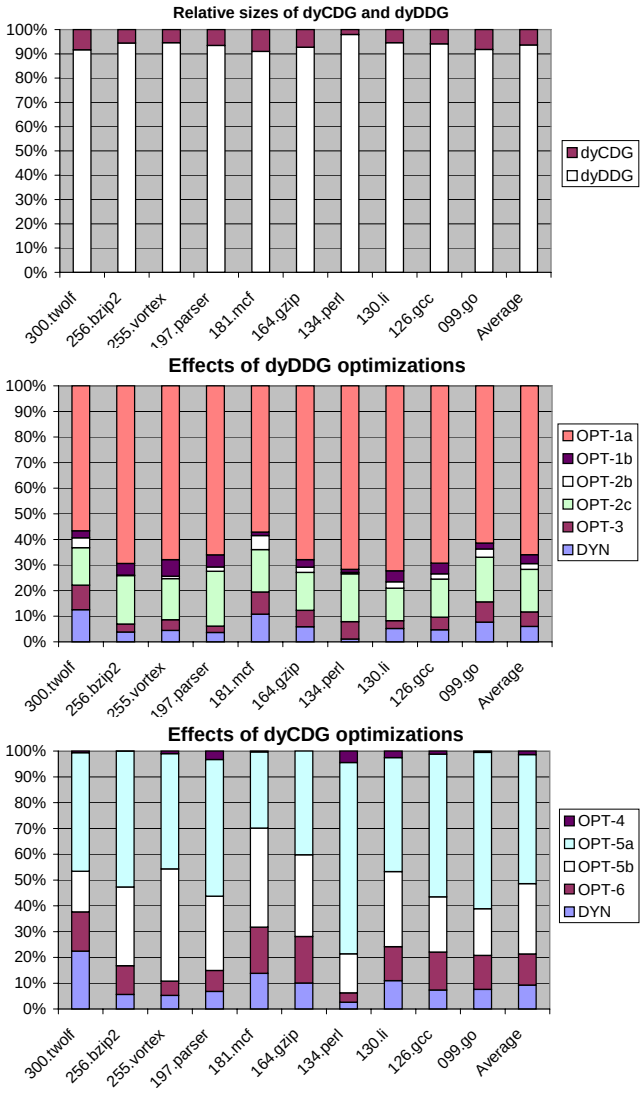


Figure 16: dyDDG vs. dyCDG size reduction.

executed. More importantly the slicing times are very promising. For 9 out of 10 benchmarks the average slicing time for 25 slices computed at the end of the run is below 18 seconds. The only exception is `300.twolf` for which average slicing time at the end of the program run is roughly 36 seconds. It is worth noting that our optimizations did not reduce the graph size for this program as much as many of the other benchmarks. Finally, at earlier points during program runs the slicing times are even lower.

We also performed the above experiment without making use of the shortcut edges in the dyDG. The average slicing times at the end of the program run with and without making use of shortcuts are given in Table 3. In 8 out of 10 benchmarks, by making use of shortcuts, the average slicing time is cut by more than half. Thus, this is an important optimization.

Finally let us consider the cost of generating the dyDGs so that dynamic slicing can be performed, Our implementation performs dyDG construction in two steps. First instrumented programs are run to collect *execution traces* (control flow and data address traces). In the Trimaran environment, the execution of the program slows down roughly by a factor of two when traces are generated. Second execution traces are *preprocessed* to generate dyDGs. The preprocessing times are shown in Table 4.

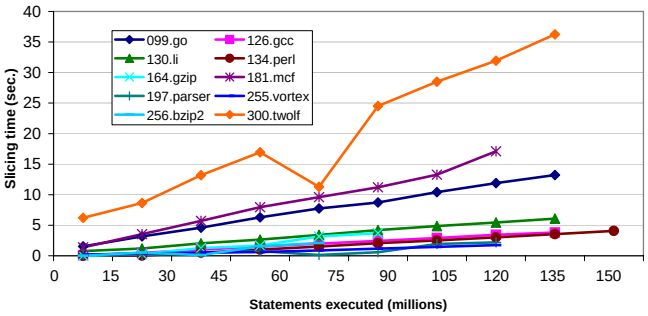


Figure 17: Dynamic slicing times of OPT.

Table 3: Benefit of providing shortcuts.

Program	OPT slicing Times (seconds)		Ratio
	w/o shortcuts	with shortcuts	w/o / with
`300.twolf`	68.01	36.25	1.88
`256.bzip2`	6.14	2.10	2.92
`255.vortex`	5.57	1.92	2.90
`197.parser`	4.86	2.21	2.20
`181.mcf`	22.05	17.10	1.29
`164.gzip`	4.54	1.74	2.61
`134.perl`	12.59	4.05	3.11
`130.li`	15.65	6.09	2.57
`126.gcc`	9.76	3.80	2.57
`099.go`	26.85	11.36	2.36

Table 4: Preprocessing time for OPT.

Program	Preprocessing Time (Minutes)	Program	Preprocessing Time (Minutes)
`300.twolf`	65.29	`256.bzip2`	38.36
`255.vortex`	44.46	`197.parser`	44.06
`181.mcf`	53.64	`164.gzip`	23.52
`134.perl`	51.12	`130.li`	49.88
`126.gcc`	48.83	`099.go`	35.24

4.2 Comparison with Other Algorithms

We also compare the performance of OPT with the LP algorithm and the traditional FP algorithm. The LP algorithm was found to be the best overall in [26] as it does not run out of memory for reasonably long program runs. The traditional FP algorithm runs out of memory for long program runs. However, in order to be able to successfully run FP, we used a machine with 2 Gigabyte RAM which was sufficient to accommodate the original dyDGs for all but one program run (`134.perl`).

LP versus OPT. The cumulative slicing times for computing up to 25 slices at the end of the program run for the two algorithms are plotted in Fig. 18. As we can see, the LP algorithm is much slower than the proposed algorithm. Computing each new slice using LP on an average takes 4.69 to 25.21 minutes depending upon the benchmark while computing the same slice using our optimized algorithm OPT takes 1.74 to 36.35 seconds. The LP algorithm spends a great deal of time traversing the execution trace stored on disk during each slice computation. The point at which the curves intersect the y-axis represents the preprocessing time – for the proposed algorithm this is the time for building the dyDG while for the LP algorithm this is the time for preprocessing the execution trace to enable faster traversal of the trace as described in [26]. The exact preprocessing times are given in Table 5. As we can see, while the preprocessing time of the proposed OPT algorithm is higher, the difference is comparable to the time spent on computing a few slices using the LP algorithm.

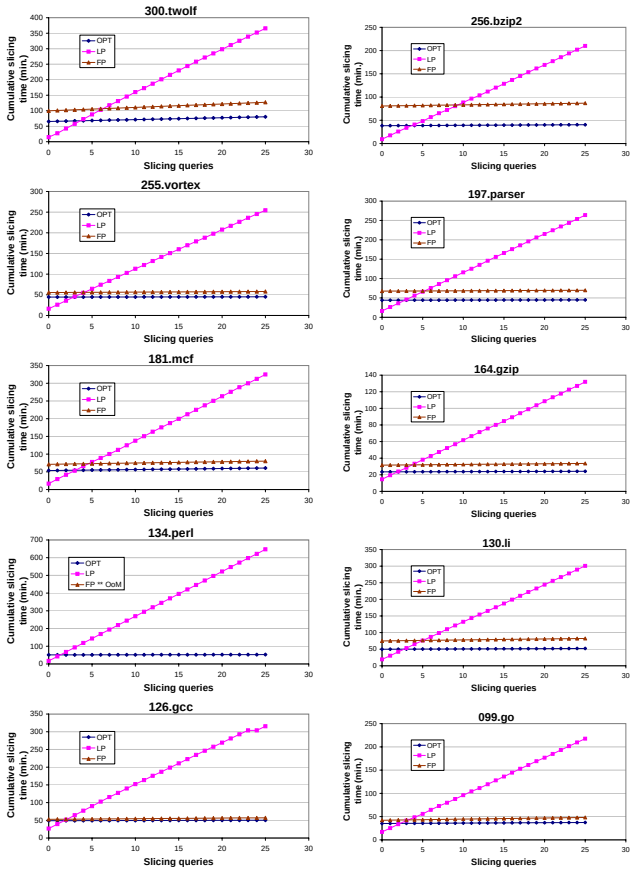


Figure 18: Comparison of OPT with LP and FP.

The memory needed by the OPT and LP algorithms is given in Table 6. While the memory used by the OPT algorithm is the size of reduced dyDG, the memory used by LP is the size of the dyDG subgraph corresponding to a slicing request. Since the latter varies with slicing requests, we report the largest dyDG subgraph size constructed in response to 25 distinct slicing requests. We note that in 5 out of 10 benchmarks the size of the largest dyDG subgraph built by LP is greater than the full reduced dyDG built by OPT. It is clear from this data that on average, the memory needs of LP and OPT are comparable to each other.

Therefore, based upon the above results we can say that OPT is superior to LP because it is much faster than LP and at the same time it uses roughly the same amount of memory as LP.

FP versus OPT. As we know, FP runs out of memory for reasonably long program runs [26]. However, we wanted to see how the slicing times of the OPT algorithm compared to that of the FP algorithm in situations where the program run was short enough to enable the entire (unoptimized) dynamic dependence graph to be kept in memory. We were able to successfully run FP on a machine with 2 Gigabyte RAM for all programs except `134.perl`.

The slicing times of FP and OPT are compared in Table 7. We observe that OPT is faster than FP. This is because of the use of shortcut edges that speed up the traversal of the dyDG. The same optimization cannot be applied to FP because the unoptimized dyDG

only contains dynamic edges. We know that the difference between OPT slicing times and FP slicing times are caused by shortcuts because when we compare the slicing times of OPT without shortcuts (given earlier in Table 3) with slicing time of FP given below, they are quite close.

Table 5: Preprocessing time: LP vs. OPT.

Program	Preprocessing Time (Minutes)		Ratio
	OPT	LP	LP/OPT
300.twolf	65.29	14.54	0.22
256.bzip2	38.36	9.38	0.25
255.vortex	44.46	16.35	0.37
197.parser	44.06	16.23	0.37
181.mcf	53.64	16.64	0.31
164.gzip	23.52	14.56	0.62
134.perl	51.12	17.18	0.34
130.li	49.88	19.23	0.39
126.gcc	48.83	26.65	0.55
099.go	35.24	17.06	0.48

Table 6: dyDG graph sizes: LP vs. OPT.

Program	Graph Size (Megabytes)	
	OPT	LP (Max. of 25 slices)
300.twolf	210.21	296.06
256.bzip2	50.48	80.66
255.vortex	64.81	33.60
197.parser	69.81	40.04
181.mcf	170.29	113.74
164.gzip	51.57	34.75
134.perl	20.92	53.62
130.li	96.50	105.45
126.gcc	74.71	57.70
099.go	131.24	162.28

Table 7: Slicing times: FP vs. OPT.

Program	Slicing Times (seconds)	
	FP	OPT
300.twolf	65.99	36.25
256.bzip2	5.92	2.10
255.vortex	6.17	1.92
197.parser	5.28	2.21
181.mcf	21.71	17.10
164.gzip	4.83	1.74
134.perl	-	4.05
130.li	17.86	6.09
126.gcc	11.03	3.80
099.go	29.79	11.36

Finally we compare the preprocessing times of OPT and FP. We had expected that the preprocessing times for OPT would be higher than FP because the OPT algorithm must spend some extra time for checking whether the timestamp pairs of all exercised dependences should be added to the dyDG or not. However, our experiments show otherwise. As the data in Table 8 shows, the preprocessing times for the FP algorithm are consistently higher than those for OPT. The reason for this behavior is that the list of timestamp pairs that are associated with a dependence edge often grow very large and thus resizing of the array in which these are stored must often be performed. These memory reallocation operations take up significant amount of time in FP while in OPT this is not the case. Thus, the overall effect of this behavior is that the preprocessing times of OPT are lower than that of FP.

Table 8: Preprocessing time: FP vs. OPT.

Program	Preprocessing Time (Minutes)		Ratio
	OPT	FP	FP/OPT
300.twolf	65.29	99.62	1.53
256.bzip2	38.36	80.78	2.11
255.vortex	44.46	55.47	1.25
197.parser	44.06	67.57	1.53
181.mcf	53.64	71.17	1.33
164.gzip	23.52	31.66	1.35
134.perl	51.12	-	-
130.li	49.88	74.86	1.50
126.gcc	48.83	52.70	1.08
099.go	35.24	42.17	1.20

Therefore, based upon the above results we can say that OPT is superior to FP not only because it scales to longer program runs, but also because it has lower preprocessing and slicing times.

Combining idea behind LP with OPT. While the above results indicate that OPT is superior to LP making the latter obsolete, the idea of demand-driven processing behind LP can be integrated into OPT to further increase the scalability of OPT as follows. We continue to use the optimizations of OPT and build the optimized dyDG in memory until the graph in memory reaches a predetermined size. At that point the *current block* of dynamic dependence information labeling the graph is saved on disk freeing up the memory used by these dynamic dependences. This process is applied repeatedly as execution continues. In other words, the new algorithm can be designed such that the graph in memory only contains a *single block* of dynamic dependences while all *other blocks* are held on disk. During slicing, a new block is loaded into memory *on demand* while the old block is discarded to free up memory. Since this algorithm will use disk space to store compacted graphs, it is expected to scale to much longer runs than the OPT algorithm.

5. RELATED WORK

Precise dynamic slicing algorithms can be divided into two broad categories: *backward computation* algorithms [14, 1, 26] that compute slices on-demand through backward traversal of dynamic dependence graph and *forward computation* algorithms [4, 17, 27] that perform forward precomputation of dynamic slices without constructing the dynamic dependence graph. In this section we discuss the drawbacks of prior algorithms and show that the algorithm proposed in this paper is a significant improvement that can be used for a broad range of applications.

When a *backward computation* algorithm constructs the entire dynamic dependence graph prior to slicing (e.g., Algorithm III in [1]), as our prior experience in [26] shows, for realistic program runs the graph is too big and thus we run out of memory while building the graph. When graph is compacted by making it imprecise (e.g., Algorithms I and II in [1]), as our prior experience in [26] shows, the slices computed can be many times larger. An alternative that we proposed in [26] is the LP algorithm which constructs the dynamic dependence graph on demand and thus only a part of the graph is constructed reducing memory needs. However, *on demand* construction of the dynamic dependence graph requires repeated traversals of the execution trace which yields a slow slicing algorithm. We believe the key difference in our approach and the one used in [1] is the use of timestamps instead of node replication to represent dynamic instances of dependences. While we were able to develop a series of optimizations to replace explicit timestamps by implicit ones, it is unclear how similar fine-grained

optimizations can be developed for nodes. In [6] a technique for compressing a dynamic execution trace is presented. Execution instances of statements are classified as critical and non-critical. For non-critical nodes, only the latest execution instances are saved in the trace. The trace compression optimization is analogous to our OPT-1a optimization at the dependence graph level.

In general, the problem with *forward computation* algorithms is that exhaustive precomputation of all dynamic slices at all program points produces large amounts of information which must be stored on disk [4]. Exhaustively computing all slices, and then finding desired slices from the extremely large number of precomputed slices, also takes significant amount of time thus yielding slow algorithms [27]. Algorithm IV in [1] is also a forward computation algorithm in which slices are computed and annotated on the dyDG. The role of the dyDG is to provide access to the slices and to avoid saving of multiple identical slices sometimes generated during repeated executions of the same statements. The forward computation algorithm in [17] sacrifices precision to speedup dynamic slicing. We have recently proposed an algorithm that goes well beyond Algorithm IV [1] in reducing memory reads. It uses reduced ordered binary decision diagrams to store the dynamic slices in compact form [27].

Finally we would like to point out that *backward computation* algorithms are more generally useful than *forward computation* algorithms. This is because, when dynamic slices are computed from dynamically constructed dependence graphs, not only can we compute dynamic slices, we can also identify the exercised dynamic dependences that contribute to the dynamic slices. The identification of dynamically exercised dependences is *essential* for some applications such as carrying out dependence based software testing [8, 16], measuring module cohesion for purpose of code restructuring [11], and performance enhancement by identifying criticality of instructions [28] and presence of instruction isomorphism [22]. For other applications such as debugging of programs [2, 15] and detecting spyware that has been installed on systems without the users' knowledge [13], while it may useful to identify the dependences, it is not necessary to do so. Thus the algorithm we present in this paper is both cost effective and generally applicable.

6. CONCLUSION

In conclusion we can see that the OPT algorithm we have proposed in this paper provides fast slicing times (1.74 to 36.25 seconds) and compact dynamic dependence graph representation (20 to 210 Megabytes) leading to a space and time efficient algorithm for dynamic slicing. In contrast, the prior algorithms are either space inefficient (corresponding graph sizes for FP are 0.84 to 1.95 Gigabytes) or time inefficient (corresponding slicing times for LP are 4.69 to 25.21 minutes) making them unattractive for use in practice. The development of a cost effective dynamic slicing algorithm is an important contribution as a wide range of applications that require analysis of dynamic information are making use of dynamic slicing [2, 15, 13, 8, 16, 11, 28, 22].

7. REFERENCES

[1] H. Agrawal and J. Horgan, "Dynamic Program Slicing," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246-256, 1990.

[2] H. Agrawal, R. DeMillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software Practice and Experience*, 23(6):589-616, 1993.

[3] T. Ball and J.R. Larus, "Efficient Path Profiling," *IEEE/ACM International Symposium on Microarchitecture*, 1996.

[4] A. Beszedes, T. Gergely, Z.M. Szabo, J. Csirik, and T. Gyimothy, "Dynamic Slicing Method for Maintenance of Large C Programs," *5th European Conference on Software Maintenance and Reengineering*, pages 105-113, March 2001.

[5] T.M. Chilimbi, "Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191-202, 2001.

[6] D.M.D. Dhamdhere, K. Gururaja, and P.G. Ganu, "A Compact Execution History for Dynamic Slicing," *Information Processing Letters*, 85:(145-152), 2003.

[7] E. Duesterwald, R. Gupta, and M.L. Soffa, "Distributed Slicing and Partial Re-execution for Distributed Programs," *5th LCPC Workshop*, LNCS 757 Springer, pages 497-511, August 1992.

[8] E. Duesterwald, R. Gupta, and M.L. Soffa, "Rigorous Data Flow Testing through Output Influences," *2nd Irvine Software Symposium*, pages 131-145, UC Irvine, CA, March 1992.

[9] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, 9(3):319-349, 1987.

[10] R. Gupta and M.L. Soffa, "Hybrid Slicing: An Approach for Refining Static Slices using Dynamic Information," *ACM SIGSOFT Symp. on the Foundations of Software Engineering*, pages 29-40, 1995.

[11] N. Gupta and P. Rao, "Program Execution Based Module Cohesion Measurement," *16th IEEE International Conf. on Automated Software Engineering*, pages 144-153, San Diego, CA November 2001.

[12] T. Hoffner, "Evaluation and Comparison of Program Slicing Tools." *Tech. Report*, Dept. of Computer and Info. Science, Linkoping University, Sweden, 1995.

[13] S. Jha, *Private communication*, University of Wisconsin at Madison, Department of Computer Science, 2003.

[14] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, 29(3):155-163, 1988.

[15] B. Korel and J. Rilling, "Application of Dynamic Slicing in Program Debugging," *Automated and Algorithmic Debugging*, 1997.

[16] M. Kamkar, "Interprocedural Dynamic Slicing with Applications to Debugging and Testing," *PhD Thesis*, Linkoping University, 1993.

[17] B. Korel and S. Yalamanchili. "Forward Computation of Dynamic Program Slices," *International Symposium on Software Testing and Analysis*, August 1994.

[18] J.R. Larus, "Whole Program Paths," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259-269, Atlanta, GA, May 1999.

[19] M. Mock, D.C. Atkinson, C. Chambers, and S.J. Eggers, "Improving Program Slicing with Dynamic Points-to Data," *ACM SIGSOFT 10th Symp. on the Foundations of Software Engineering*, 2002.

[20] C.G. Nevil-Manning and I.H. Witten, "Linear-time, Incremental Hierarchy Inference for Compression," *Data Compression Conference*, Snowbird, Utah, IEEE Computer Society, pages 3-11, 1997.

[21] T. Reps and G. Rosay, "Precise Interprocedural Chopping," *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, DC, pages 41-52, October 1995.

[22] Y. Sazeides, "Instruction-Isomorphism in Program Execution," *Value Prediction Workshop* (held with ISCA), June 2003.

[23] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, 3(3):121-189, Sept. 1995.

[24] G. Venkatesh, "Experimental Results from Dynamic Slicing of C Programs," *ACM Transactions on Programming Languages and Systems*, 17(2):197-216, 1995.

[25] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, SE-10(4):352-357, 1982.

[26] X. Zhang, R. Gupta, and Y. Zhang, "Precise Dynamic Slicing Algorithms," *IEEE/ACM International Conference on Software Engineering*, pages 319-329, Portland, Oregon, May 2003.

[27] X. Zhang, R. Gupta, and Y. Zhang, "Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams," *IEEE/ACM International Conference on Software Engineering*, Edinburgh, UK, May 2004.

[28] C.B. Zilles and G. Sohi, "Understanding the Backward Slices of Performance Degrading Instructions," *ACM/IEEE 27th International Symposium on Computer Architecture*, 2000.