Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams*

Xiangyu Zhang Rajiv Gupta
The University of Arizona
Dept. of Computer Science
Tucson, Arizona 85721

Youtao Zhang
The Univ. of Texas at Dallas
Dept. of Computer Science
Richardson, Texas 75083

Abstract

Dynamic slicing algorithms can greatly reduce the debugging effort by focusing the attention of the user on a relevant subset of program statements. Recently algorithms for `forward computation` *of dynamic slices have been proposed which maintain dynamic slices of all variables as the program executes. An advantage of this approach is that when a request for a slice is made, it is already available. The main disadvantage of using such an algorithm for slicing realistic programs is that the space and time required to maintain a large set of dynamic slices corresponding to all program variables can be very high. In this paper we analyze the characteristics of dynamic slices and identify properties that enable space efficient representation of a set of dynamic slices. We show that by using reduced ordered binary decision diagrams (roBDDs) to represent a set of dynamic slices, the space and time requirements of maintaining dynamic slices are greatly reduced. In fact not only can the* `latest` *dynamic slices of all variables be easily maintained, but rather* `all` *dynamic slices of all variables throughout a program's execution can be maintained. Our experiments show that our roBDD based algorithm for forward computation of dynamic slices can maintain 107-217 million dynamic slices arising during long program runs using only 28-392 megabytes of storage. In addition, the performance of the roBDD based forward computation method compares favorably with the performance of the LP backward computation algorithm.*

1. Introduction

The concept of program slicing was first introduced by Mark Weiser [17]. 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 both algorithms and tools for intraprocedural and interprocedural static slicing [15, 7]. It is widely recognized that for programs that make extensive use of pointers, the highly conservative nature of data dependency analysis leads to highly imprecise and considerably larger slices [7, 18, 13]. For program debugging, where the objective of slicing is to reduce the debugging effort by focusing the attention of the user on the relevant subset of program statements, conservatively computed large slices are clearly undesirable. To address this problem Korel and Laski proposed the idea of *dynamic slicing* [8]. It has been shown that precise dynamic slices can be considerably smaller than static slices [16, 7, 18]. Dynamic slicing has also been used in other applications including software testing [5], software maintenance [3], and measuring module cohesion [6].

Two types of algorithms for computing dynamic slices have been proposed: *backward computation* methods [8, 1, 18]; and *forward computation* methods [3, 9]. In *backward computation* methods the program dependences that are exercised during a program execution are captured and saved in form of a dynamic dependence graph. Dynamic slices are constructed upon user's requests by backward traversal of the dynamic dependence graph. One of the strengths of this approach is that it allows computation of *all* dynamic slices of all variables for the entire execution. A problem with this method is its space and time costs. The dynamic dependence graph grows in size as a program executes causing a typical machine to run out of memory and the time spent on building the dependence graph is also large. To address this problem we presented the LP algorithm in [18] which keeps the execution trace on disk and constructs the relevant part of the dynamic dependence graph in memory *on demand* in response to slicing queries.

In *forward computation* methods [3, 9] dynamic slices of all program variables are maintained as the program executes. Advantages of this approach are that when a request for a slice is made, it is already available and since slices are explicitly maintained, there is no need to construct the

*Supported by grants from IBM, Microsoft, Intel and National Science Foundation grants CCR-0324969, CCR-0220334, CCR-0208756, CCR-0105355, and EIA-0080123 to the University of Arizona.

dynamic dependence graph. One of the limitations of this approach is that only the *latest* dynamic slices for all variables are maintained. This is because, if we maintain *all* dynamic slices of all variables for the entire execution, the space requirements increase dramatically. The time cost of updating dynamic slices as program executes is also quite significant as, for large programs, set operations need to be performed on large sets of statements. As far as we know, no experimental data has been published for forward computation algorithm.

In this paper we develop an effective algorithm for *forward computation* of dynamic slices, experimentally evaluate the cost of the developed algorithm, and compare its performance with the most practical backward computation algorithm (i.e., the LP algorithm [18]). The key contributions of our work are as follows:

Identifying statistical characteristics of dynamic slices. By executing several benchmark programs, we collected large numbers of dynamic slices and studied these slices to identify characteristics of these slices that could be exploited for reducing the space needed to store the slices. We identified three distinct characteristics: same dynamic slices tend to *reappear* from time to time during execution, different slices tend to *share statements*, and *clusters of statements* located near each other in the program often appear in a slice.

Space and time efficient forward computation algorithm. We develop a forward computation algorithm that reduces the space and time costs by exploiting the three characteristics of dynamic slices mentioned above. First the *reappearance* of slices is exploited by eliminating duplicates and simply saving a set of distinct slices. Second the *sharing and clustering* properties are exploited by representing the sets corresponding to dynamic slices using reduced ordered BDDs (roBDDs). The resulting algorithm is not only space efficient, it is also time efficient because set operations can be performed efficiently using their roBDD representations.

Maintaining all dynamic slices for an execution. We show that due to their high space requirements existing *forward computation* algorithms can only maintain *latest* dynamic slices for all variables in memory. However, the dramatic improvement in space efficiency achieved by our algorithm makes it possible to maintain *all* dynamic slices for all variables for the entire program execution with a modest increase in space requirements. However, an additional extension is required. The indices that relate different execution instances of statements with their corresponding dynamic slices need to be compressed to limit space needs. This goal is achieved using the SEQUITUR algorithm [14].

Comparison of backward computation and forward computation algorithms. In addition to providing an experimental evaluation of our forward computation algorithm, we also compare its performance with the most practical *backward computation* algorithm., i.e. the LP dynamic slicing algorithm [18]. The LP algorithm was also carefully designed to be space and time efficient. Our evaluation shows that while LP algorithm is more suitable when only a few slicing computations are performed, our new *forward computation* algorithm is more desirable when a larger number of dynamic slicing operations are to be performed.

The remainder of the paper is organized as follows. In section 2, we briefly describe the existing *forward computation* algorithm and highlight its costs. In section 3 we describe the three characteristics of dynamic slices. In section 4 we develop our algorithm which exploits the three characteristics identified in section 3. Experimental evaluation is presented in section 5. We conclude with a summary of our contributions in section 6.

2 Forward Computation of Dynamic Slices

Given a program execution, the **dynamic slice** of a variable v at the execution point of i_j, which denotes the jth execution instance of statement i, is the set of statements that contributed to the value of v at that point. These statements are identified by taking backward closure over data and control dependences starting from statement that defined the value of v at i_j. A dynamic slicing query is therefore represented by a pair of the form $< v, i_j >$.

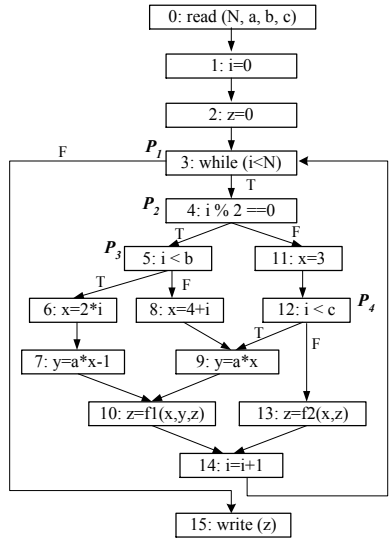

Figure 1. An example program.

Consider the program shown in Fig. 1. Assume the execution follows the path $[0.1.2.3.4.11.12.13.14.3.15]$ for

some program input. The answer to the query $< z, 15_1 >$ (i.e., dynamic slice of variable z at execution point that immediately follows the first execution of statement 15) is $\{0, 1, 2, 3, 4, 11, 12, 13, 15\}$. In this set statements $\{0, 1, 2, 11, 13\}$ and $\{3, 4, 12\}$ were added due to data and control dependences respectively.

Let us now briefly discuss the manner in which dynamic slices are computed by a *forward computation* algorithm. We use the following notation in the discussion. Given a statement i, $Def[i]$ and $Use[i]$ denote the *dynamic* sets of variables that are defined and used by statement i while $CD[i]$ denotes the set of predicate statements on which i is *statically* control dependent. Each statement i is assigned a timestamp $time[i]$ to remember when it was last executed. The latest execution instance number of statement i is denoted by $lei[i]$. Finally $slice[i][j]$ denotes the dynamic slice for the jth execution instance of statement i and $def_slice[v]$ denotes the dynamic slice for the latest definition of v.

As mentioned earlier, a forward computation algorithm continuously computes dynamic slices as statements are executed. Although all slices are computed, only the latest slices are saved. After execution of jth instance of statement i, the dynamic slice $slice[i][j]$ is computed to include the following: statements that belong to latest dynamic slices of variables used by i (i.e., in $Use[i]$); statements that belong to the dynamic slice of predicate on which i_j is control dependent; and the statement i itself. If statement i defines variable v (i.e., $v \in Def[i]$)), the latest dynamic slice for variable v is $def_slice[v] = slice[i][j]$. While a statement may be statically control dependent upon multiple predicates, for the above slice computation we need to identify the predicate instance on which i_j is dynamically control dependent. The *dynamic control dependence* corresponds to the executed predicate in $CD[i]$ with the highest timestamp. Below the updating of dynamic slicing information following execution of statement i is summarized.

Algorithm 1 *Updating Slicing Information*

Procedure Update(i)

1: $slice = \{i\}$;
2: $time[i] = timestamp$++;
3: **for** (each use u in $Use[i]$) **do**
4: $\quad slice = slice \cup def_slice[u]$;
5: **end for**
6: dcd = the statement s in $CD[i]$ which has the maximum $time[s]$ value;
7: $slice = slice \cup slice[dcd][lei[dcd]]$;
8: $slice[i][lei[i]] = slice$;
9: $lei[i]$++;
10: **for** (each definition d in $Def[i]$) **do**
11: $\quad def_slice[d] = slice$;
12: **end for**

Forward computation of dynamic slices for example in Fig. 1 is shown in Table 1. In the execution step of 10_1, which is $z = f1(x, y, z)$, z is defined, $\{x, y, z\}$ are the variables that are used. We can tell from the table that at this point, $def_slice[x] = \{0, 1, 3, 4, 11\}$, $def_slice[y] = \{0, 1, 3, 4, 9, 11, 12\}$, and $def_slice[z] = \{2\}$. Although statement 10 is statically control dependent upon 4 and 12, since $time[12] = 6 > time[4] = 4$, this instance of 10 is control dependent upon 12. Therefore $slice[z] = slice[10][1] = def_slice[x] \cup def_slice[y] \cup def_slice[z] \cup slice[12][1] \cup \{10\} = \{0, 1, 2, 3, 4, 9, 10, 11, 12\}$.

Table 1. Forward computation of slices.

$i_{lei[i]}$	dynamic $Def[i]$	dynamic $Use[i]$	static $CD[i]$	$time[i]$	$slice[i][ei[i]] = def_slice[d \in Def[i]]$
0_1	{N,a,b,c}	∅	∅	0	{0}
1_1	{i}	∅	∅	1	{1}
2_1	{z}	∅	∅	2	{2}
3_1	∅	{i,N}	∅	3	{0,1,3}
4_1	∅	{i}	{3}	4	{0,1,3,4}
11_1	{x}	∅	{4}	5	{0,1,3,4,11}
12_1	∅	{i,c}	{4}	6	{0,1,3,4,12}
9_1	{y}	{a,x}	{5,12}	7	{0,1,3,4,9,11,12}
10_1	{z}	{x,y,z}	{4,12}	8	{0,1,3,4,9,10,11,12}
14_1	{i}	{i}	{3}	9	{0,1,3,14}
3_2	∅	{i,N}	∅	10	{0,1,3,14}
15_1	∅	{z}	∅	11	{0,1,3,4,9,10,11,12}

Even though the forward computation algorithm is straightforward, its implementation must be carefully designed because the space costs of maintaining dynamic slices as sets of statements and the time cost of performing multiple set operations following execution of each statement can be very high. Because a real program may contain hundreds of thousands or even millions of statements, using a *bit vector* representation for sets of statements (i.e., slices) is not space efficient. If we implement sets as an *ordered list* of statement numbers, then the space used by a set will be proportional to the number of elements in the set. Unioning two sets of size n and m will take $O(n + m)$ time. Next we estimate the space and time costs for maintaining dynamic slices.

Let $|V|$, $|SE|$ and $|DSE|$ respectively denote the number of variables used, number of statements executed, and number of statically distinct statements that are executed at least once during a program run. The number of dynamic slices computed over the course of the program run is equal to $|SE|$ and during each computation a small number of set union operations are performed. Since the maximum size of each set (dynamic slice) is equal to $|DSE|$, the worst case time complexity of forward computation algorithm is $O(|SE| \times |DSE|)$. If *all* dynamic slices for all variables for entire execution are saved, the space complexity of the algorithm is a $O(|SE| \times |DSE|)$ as the number of slices that need to be saved can be as high as $|SE|$. However, if only *latest* dynamic slices for all variables are kept, the space complexity reduces to $O(|V| \times |DSE|)$.

To guage the above complexities of some real application programs we considered the following pro-

grams: `008.espresso` from the `Specint92` suite, and `130.li`, `134.perl`, `099.go` and `147.vortex` from the `Specint95` suite. The sizes of these programs in terms of number of statements are given in column *Size* of Table 2. The program runs on which the complexities are studied involved execution of 107 to 217 million statements. The average values for $|SE|$, $|DSE|$, and $|V|$ for these program runs are 151 million, 34 thousand, and 72 thousand. Therefore it is apparent that the cost of forward computation of dynamic slices is very high. As we go from maintaining *latest* slices to *all* slices for all variables, the number of slices increases by *four orders of magnitude* (from tens of thousands to well over 100 million). Therefore, not surprisingly, we found that if we attempted to maintain *all* dynamic slices, our machine ran out of memory even for small program runs.

Table 2. Cost of forward computation.

Program	Lines of C Code	$\|V\|$	$\|DSE\|$	$\|SE\|$ (millions)
008.espresso	14,850	43535	22470	165.1
130.li	7,741	156102	10399	126.6
134.perl	27,044	7040	21767	217.5
099.go	29,629	91574	50371	139.5
147.vortex	67,213	65285	69249	107.5
Average	29,295	72707	34851	151.2

3 Characteristics of Dynamic Slices

In order to gain insight into possible means by which large number of slices could be compactly represented, we analyzed dynamic slices collected for several benchmarks runs and identified the following three characteristics.

Reappearing slices. The number of distinct slices does not increase steadily with the length of an execution run. The rate at which number of distinct slices increases becomes slower as the execution goes longer. This is due to the *reappearing* nature of dynamic slices. As execution goes longer, the likelihood that a newly computed dynamic slice has been seen before increases. Fig. 2 plots the number of distinct slices seen and the number of statements executed. As we can see, initially the number of distinct slices increases rapidly, but later the increase occurs at much slower pace. We also observed that when the set of slices that correspond to different execution instances of a given statement are considered, a small number of distinct slices are found to appear repeatedly. Note that while we have studied reappearing slices in individual program runs, in [4] researchers report observing similar phenomenon across different runs of a program.

Overlapping. Slices corresponding to different variables may not be identical, but they often have many statements in common. Table 3 tells us the average percentage

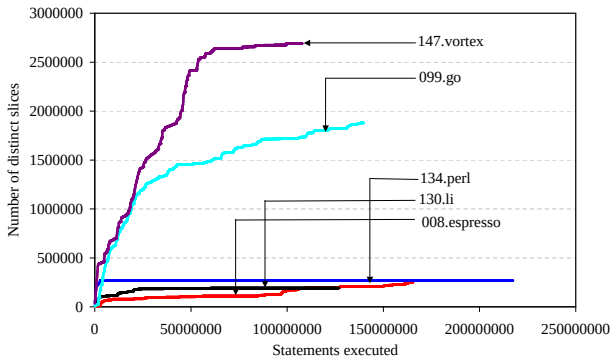

Figure 2. The number of distinct slices.

of slices in which a statement appears when slices at all execution points (All Points) and slices at latest execution point (Latest Point) are considered. As the data shows on an average a statement appears in 33.3% and 39.6% of all slices at all execution points and current execution point respectively. The reason for repeated occurrences of same statements in different slices is due to sharing of data and control dependences by different statements.

Table 3. Average statement appearance rate.

Program	All Points	Latest Point
008.espresso	26.5%	41.8%
130.li	33.2%	52.6%
134.perl	30.4%	35.0%
099.go	43.0%	16.2%
147.vortex	33.6%	52.6%
Average	33.3%	39.6%

Clustering. If a statement is in a slice, other statements in the program that are in proximity of this statement are likely to also be in the slice. In fact, statements belonging to a slice appear in scattered clusters across the program. Fig. 3 demonstrates this phenomenon. We computed a single slice at regular intervals during program execution for benchmark `130.li` and plotted them as follows. The x-axis represents the statements in the program in the order they appear in the program. The y-axis corresponds to execution time. At a sampling point the statements in the computed slice are plotted in the graph. As we can see, each slice appears as a set of scattered clusters along the x axis.

4 Efficient Forward Computation

Using the characteristics identified in the preceding section, we develop an algorithm for forward computation that is both space and time efficient. We assume that *all* dynamic slices are to be saved so that it is possible to respond

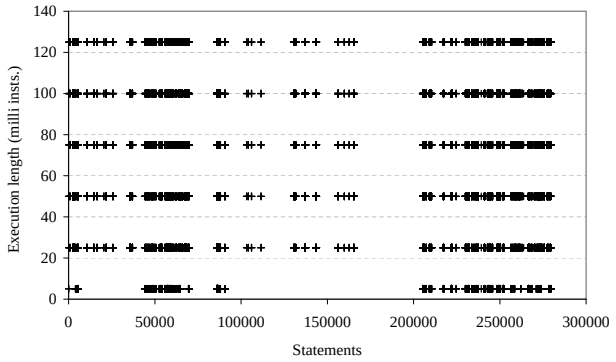

Figure 3. Spatial distribution of statements in a slice.

to slicing requests for all variables at all execution points. Of course our algorithm can be easily adapted to maintain latest dynamic slices of all variables. We develop our algorithm in two steps. First we show how the *reappearing slices* can be exploited to reduce the amount of information being saved. Second we show how the characteristics of *overlapping* and *clustering* are exploited to generate a compact representation of slices by using reduced ordered BDDs.

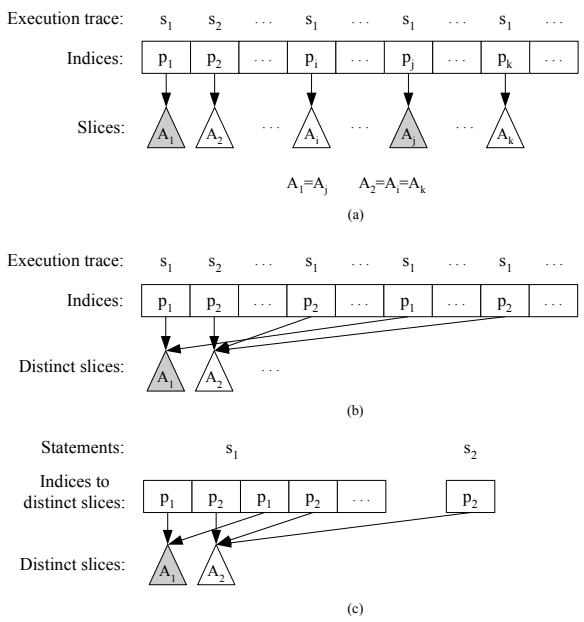

Figure 4. Eliminating duplicates.

4.1 Exploiting Reappearing Slices

Let us begin with a simple representation of the slicing information that we need to maintain. It consists of an execution trace which simply gives the sequence of statements executed during program execution. Corresponding to each entry in the trace, the dynamic slice for execution of the statement the entry represents is also saved. This representation is shown in Fig. 4(a). Since identical sets of statements appear multiple times, we first save space by eliminating the duplicates. An array of distinct slices is created and for each entry of the execution trace the index of the *distinct slices* array where the corresponding slice is stored is remembered. This step results in a representation of the form shown in Fig. 4(b). In this way we have exploited the characteristic of reappearing slices.

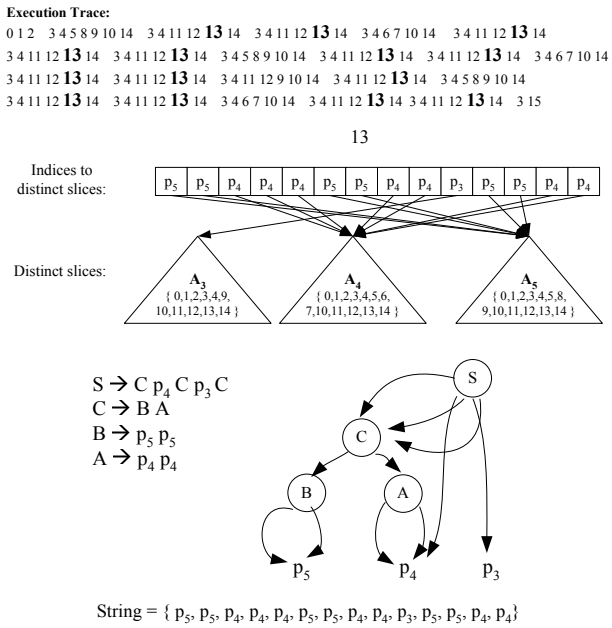

Figure 5. Compressing indices.

While we have reduced the space taken by dynamic slices by eliminating duplicates, the indices still occupy a great deal of space. To reduce this space requirement we reorganize the indices such that the indices for dynamic slices corresponding to multiple execution instances of each distinct statement are grouped together (see Fig. 4(c)). As mentioned in the preceding section, we have observed that when the set of slices that correspond to different execution instances of a given statement are considered, a small number of distinct slices are found to appear repeatedly. This property leads to presence of patterns in the sequence of indices for a given statement. These patterns can be exploited to achieve compression of an index sequence. To carry out such compression we make use of the SEQUITUR algorithm [14]. SEQUITUR is an online algorithm that detects patterns in a string and factors them out by producing a context-free grammar and it does so in linear time.

We illustrate the application of SEQUITUR through an example shown in Fig. 5. For the example program of

Fig. 1, we present an execution trace that is shown first in Fig. 5. As we see statement 13 appears multiple times in the execution trace. By examining the code in Fig. 1 it is easy to determine that statement 13 can have at most five distinct dynamic slices no matter how many times it is executed. Three of these slices are actually encountered during the given run. The composition of these three slices and the order in which they occur during executions of statement 13 are shown next in Fig. 5. When we look at the sequence of indices for statement 13 we notice that the pattern $p_5 : p_5 : p_4 : p_4$ occurs multiple times. The SEQUITUR takes advantage of this pattern and produces the grammar shown in Fig. 5. A DAG representation for the grammar is also shown. The greater the degree of repeating patterns in the string, the greater is the amount of compression achieved by SEQUITUR.

4.2 Exploiting Overlapping and Clustering

In the preceding section we have already shown how to compress the indices and eliminate duplicated slices. Our next step is to show how the distinct dynamic slices can be compressed based upon the overlapping and clustering characteristics. The key issue here is to represent sets of statements (i.e., dynamic slices) in a compact form while allowing efficient implementation of set operations. It is well known that *reduced ordered* BDDs can compactly represent large sets and efficiently implement set operations [11, 12]. In this section we show that the characteristics of dynamic slices can be exploited to effectively make use of roBDDs.

We begin by describing how ordered BDDs are used to represent sets and how they are reduced. Given π, the total order on a set of variables $v_0, ..., v_n$, an *ordered* BDD is a directed acyclic graph that satisfies the following properties:

- There are exactly two nodes without outgoing edges, which are labeled by the constants 1 and 0 respectively. They are called **sinks**.
- Each non-sink node is labeled by a variable v_i, and has two outgoing edges, which are called the **1-edge** and the **0-edge**. The *1-edge*s are drawn as solid arrows while *0-edge*s are drawn as dashed arrows.
- The order in which the variables appear on a path in the graph is consistent with the variable order π.

Let us assume that we are given a universal set which contains integers 0 through 15. We show how any set drawn from this universal set can be represented using an ordered BDD. Since each element of the set can be uniquely represented using four bits, we represent it using an ordered BDD with four variables, corresponding to the four bits, with v_3, v_2, v_1, v_0 as the variable order. The ordered BDD representing the set $\{0, 1, 2, 3, 4, 9, 10, 11, 12\}$ is shown in Fig. 6(a). This is how we determine where an element from the universal set belongs to the set represented by the ordered BDD. Given the value 4 (i.e., 0100) we follow the path corresponding to edge sequence 0100 to see if 4 is present in the set. The sink of this path is the node labeled 1 which is considered to indicate that 4 is part of the set. On the other hand for value 5 (i.e., 0110) when we follow the path 0110 we reach the sink node labeled with 0 indicating that 5 does not belong to the set.

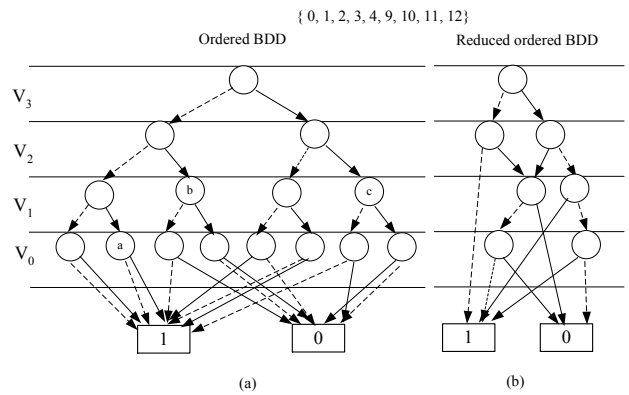

Figure 6. Reduced ordered BDD for a set.

An ordered BDD can be converted to a more compact *reduced ordered* BDD (roBDD) using two simple rules: the *Elimination Rule* and the *Merging Rule*. According to the *elimination rule*, if both edges of a node n point to the same successor s (i.e., the value of the variable corresponding to the current level in the BDD does not effect path selection), then the node n is eliminated by redirecting all incoming edges of n to its successor s. According to the *merging rule* if two nodes n_1 and n_2 are isomorphic, we can remove the redundancy by merging the two nodes and redirecting the incoming edges of these nodes to the merged node. The rules are applied repeatedly till a point is reached when no more rules can be applied. The resulting BDD is called a reduced ordered BDD or roBDD. For the example ordered BDD of Fig. 6(a), node a can be removed using the elimination rule and nodes b and c can be merged using the merging rule. The final roBDD is shown in Fig. 6(b).

Since we are interested in maintaining multiple dynamic slices, i.e. multiple sets of statements, we will need to construct multiple roBDDs. However, these roBDDs can also share nodes and thus we obtain a multiple rooted roBDD where each root corresponds to a distinct slice. The example in Fig. 7 illustrates how roBDDs of two sets $\{0, 1, 2, 3, 4, 9, 10, 11, 12\}$ and $\{0, 1, 2, 3, 4, 8, 9, 10\}$ are represented. As we can see, the two roBDDs share nodes. If two dynamic slices are identical, they share the same root in the graph. Thus, roBDDs can also exploit the *reappearing slices* characteristic of dynamic slices.

Let us now see how the *clustering* characteristic of dy-

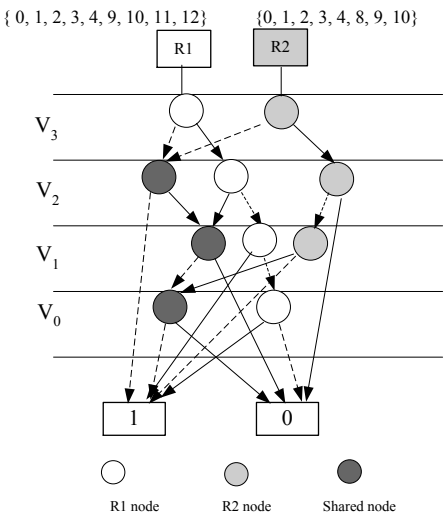

Figure 7. roBDDs for two sets.

namic slices can be used to increase the likelihood of opportunities arising for applying the above reduction rules. If we use statement numbers (i.e., numbers assigned based upon the order in which statements appear in the program) to identify the statements, then presence of a single cluster will enable applications of the elimination rule and the presence of multiple clusters will create isomorphic nodes enabling the application of the merging rule. The *overlapping* characteristic further facilitates sharing of nodes across different slices. Thus, we can see that the presence of clustering and overlapping characteristics make the roBDD a perfect choice for representing dynamic slices.

Table 4 shows average number of statements in dynamic slices computed during runs of programs characterized in Table 2. It also shows the average number of nodes in the corresponding roBDDs for these slices. As we can see, the number of nodes in the roBDDs is significantly fewer than the number of statements in the slices. Thus, this data indicates that roBDDs are more compact than a representation that maintains explicitly the elements in each set.

Table 4. Average slice size vs. roBDD size

Program	Slice Size (statements)	roBDD Size (nodes)
008.espresso	5947	1990
130.li	3450	1431
134.perl	6616	2716
099.go	21650	6219
147.vortex	23287	7527

Set operations can be performed efficiently using roBDDs. Most importantly, equivalence test can be performed in $O(1)$ time [11]. Other binary operations (e.g., union) on two sets whose roBDD representations contain n and m nodes can be performed in time $O(n \times m)$ [11]. Elements of a given set can also be efficiently enumerated. The complexity of this operation is $O(|S|)$ where $|S|$ is the size of the set [11].

5 Experimental Results

5.1 Implementation

For our experimentation we used the *Trimaran* system [21] that takes a `C` program as its input and produces a lower level intermediate representation (IR). This intermediate representation is used as the basis for slicing by our implementations of the algorithms. In other words, when slicing is performed, we compute the slices in terms of a set of statements from this IR. We use the BuDDy [10] BDD package and an implementation of SEQUITUR to carry out our experiments. The *optimized* algorithms we studied in our experiments are listed below. *We do not consider unoptimized implementation of forward computation algorithm because it runs out of memory even for small program runs.* We support two versions of the forward computation algorithms for situations where *latest* and *all* slices are saved.

`Non-BDD` exploits reappearing slices by eliminating duplicates and uses SEQUITUR to compress indices. Sets are implemented using ordered lists. Therefore the union operation can be performed in time proportional to the total number of elements in the two sets combined. To speedup the equivalence test for sets we use hashing.

`roBDD` exploits all three characteristics of dynamic slices and thus uses reduced ordered BDDs as well as SEQUITUR. It is important to understand that when reduction rules are applied, the storage that is freed is not garbage collected immediately. Rather garbage collection is performed when the number of nodes allocated exceeds a preset threshold. This approach greatly limits the overhead of garbage collection at the cost of using some extra memory.

`LP` is the backward computation algorithm that was introduced in [18]. We compare its performance with the above `roBDD` algorithm.

`oFP` is the fastest known backward computation algorithm introduced in [19]. We also compare its performance with the above `roBDD` algorithm.

To achieve a fair comparison among the various dynamic slicing algorithms, we have taken great care in implementing them. The dynamic slicing algorithms that are implemented share code whenever possible and use the same basic libraries.

The programs and the characteristics of program runs used in experimentation were given earlier in Table 2 – in these runs the number of statements executed range from 107 to 217 million. The system used in our experiments is a 2.0 GHz Pentium 4 linux workstation with 1.0 GB RAM and 1.0 GB of swap space.

5.2 Memory Requirements: roBDD and Non-BDD

Table 5 shows the memory consumed by the *forward* dynamic slicing algorithms to store distinct slices when only the *latest slices* are saved. As we can see, the memory requirement of the `roBDD` algorithm is 1.36 to 10.8 times less than that of the `Non-BDD` algorithm. We also performed the same experiment for the case when *all slices* are saved. In this situation the `Non-BDD` algorithm ran out of memory in all program runs being considered. However, we were able to *estimate* the amount of memory needed by `Non-BDD` – this estimation was made by traversing the BDD which is free of duplicates. As we can see, the space requirements of `Non-BDD` range from 1.753 GB to 199.7 GB. In contrast the space from the data in Table 6, space requirements of the `roBDD` algorithm range between 28 MB and 392 MB. Thus we can see that while the use of roBDDs allows us to keep *all slices* in memory, the same cannot be achieved using a `Non-BDD` algorithm which only eliminates duplicates.

Table 5. Non-BDD vs. roBDD: *latest slices***.**

Program	Space Required		Non-BDD / roBDD
	Non-BDD (MB)	roBDD (MB)	
008.espresso	60.0	44.1	1.36
130.li	41.9	28.0	1.50
134.perl	71.7	48.7	1.47
099.go	150.6	100.0	1.51
147.vortex	1084.4	100.0	10.8

Table 6. Non-BDD vs. roBDD: *all slices***.**

Program	Space Required		Non-BDD / roBDD
	Non-BDD *Estimate* (MB)	roBDD (MB)	
008.espresso	5667.8	44.1	128.44
130.li	1753.0	28.0	62.51
134.perl	5192.7	48.7	106.69
099.go	116788.7	255.8	456.54
147.vortex	199713.9	392.8	508.42

Note that the data for `roBDD` in Tables 5 and 6 appears to show that for the first three programs the memory used by the roBDD does not appear to increase as we go from saving *latest slices* to *all slices*. Actually this is not the case. The roBDD for *latest slices* is much smaller but the nodes freed when older slices of variables are discarded were not garbage collected because the total number of nodes created did not reach the threshold at which garbage collection kicks in.

In addition to the memory needed to store distinct slices, memory is needed also to store indices. Recall that these indices are compressed through the use of SEQUITUR. Memory needed to store the indices before and after compression is given in Table 7. The compression achieved by SEQUITUR is quite substantial though it varies widely across the benchmarks (compression factor ranges from 3.62 to 131.59).

Table 7. SEQUITUR compression.

Program	Space Required		Before / After
	Before (MB)	After (MB)	
008.espresso	660.3	29.9	22.08
130.li	506.5	25.4	19.90
134.perl	868.2	6.6	131.59
099.go	557.9	115.2	4.84
147.vortex	430.2	118.7	3.62

Finally we give the total memory needs of the `roBDD` algorithm when *latest slices* and *all slices* are used. The total is computed by summing the memory used to store the roBDD and the compressed indices. The results in Table 8 show that the memory usage of the `roBDD` algorithm increases by factors ranging from 1.14 to 5.11 when we go from saving *latest slices* to saving *all slices*. Recall from the discussion of data in Table 2 in section 2 that the number of *all slices* is roughly four orders of magnitude greater than the number of *latest slices* for these program runs as they involve execution of large number of statements (107 to 217 million). Thus while the number of slices being saved increase by a factor of 10^4, the memory needs of the algorithm increase only by a factor of 1.14 to 5.11. In other words, roBDDs are highly effective in exploiting the clustering and overlapping characteristics of dynamic slices.

5.3 Execution Times: roBDD and Non-BDD

Table 9 presents the execution times of the `roBDD` algorithm when *all slices* are saved. The time is divided into time spent on maintaining the roBDD and time spent on maintaining the indices using SEQUITUR. As we can see neither activity is dominant – sometimes maintaining the roBDD takes more time while in other cases maintaining the indices takes more time.

Table 8. roBDD: *latest* **vs.** *all* **slices.**

Program	Space Required		All Slices / Latest Slices
	Latest (MB)	All (MB)	
008.espresso	44.1	74.0	1.68
130.li	28.0	53.4	1.90
134.perl	48.7	55.3	1.14
099.go	100.0	371.0	3.71
147.vortex	100.0	511.5	5.11

Table 9. roBDD: *all slices*.

Program	roBDD (min.)	SEQUITUR (min.)	Total
008.espresso	99.7	79.1	178.8
130.li	57.3	66.6	123.8
134.perl	180.8	132.8	313.6
099.go	100.5	81.2	181.7
147.vortex	175.5	109.2	284.6

We also compare the execution times of the `Non-BDD` and `roBDD` algorithms in Table 10. Because the `Non-BDD` algorithm runs out of memory when *all slices* are computed even for short program runs, we only ran this experiment when *latest slices* are maintained. Moreover because the *Non-BDD* algorithm is very slow, we ran this experiment for short program runs in which 15 to 30 million statements were executed. The speedup column in Table 10 shows that the speedup factor achieved by the `roBDD` algorithm over the `Non-BDD` algorithm ranges from 33 to 99. Thus it is clear that roBDD is also time efficient.

Table 10. Non-BDD vs. roBDD: *latest slices*.

Program	$\|SE\|$ (millions)	Non-BDD (min.)	roBDD (min.)	Speedup
008.espresso	21.4	487.82	7.74	63
130.li	17.5	544.06	7.26	75
134.perl	30.6	803.81	24.55	33
099.go	20.9	742.30	10.29	72
147.vortex	15.3	990.10	9.98	99

5.4 roBDD vs. LP & oFP Algorithms

Finally we compare the performance of the `roBDD` algorithm with the performance of the LP algorithm which was found to be an effective backward computation algorithm in [18] and oFP which is the fastest known dynamic slicing algorithm proposed in [19].

Dynamic slicing is memory intensive no matter whether we use forward computation or backward computation. The LP algorithm solved the memory problem by keeping execution traces on disk and then keeping the relevant portion of the dynamic dependence graph in memory which is built on demand from the execution trace in response to slicing requests. The oFP algorithm developed optimizations through which a number of dependence instances are allowed to share a single representative edge thereby dramatically reducing the size of the dynamic dependence graph. In contrast, the `roBDD` algorithm solved the memory problem by using a very compact representation of dynamic slices. While the LP and oFP algorithms must compute the slice when a slicing request is made, the `roBDD` algorithm precomputes all the slices and thus has to merely access the required slice. The `roBDD` algorithm responds to individual slicing requests much faster than LP and slightly faster than oFP. However, the preprocessing time for `roBDD` is higher than the preprocessing times of the LP and oFP algorithms. These results are demonstrated by the data plotted in Fig. 8. The cumulative time for computing up to 25 slices is plotted for all the benchmarks. In comparing the algorithms we used them to compute the exact same dynamic slices in the exact same order. In these graphs the times corresponding to 0 slices represent the preprocessing times of the algorithms.

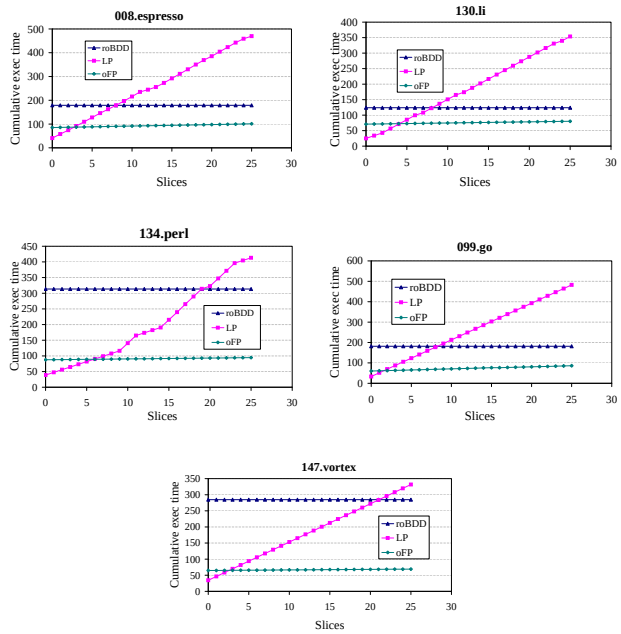

Figure 8. Execution Times: LP, oFP, & roBDD.

Finally, one of the advantage of using backward computation algorithms is that they provide the dynamic dependence graph which may contain valuable information. In contrast, the forward computation algorithm only provides the set of statements in the slice. This drawback can be overcome by saving a small amount of dynamic control flow and data dependence information from which, if needed, the entire dynamic dependence graph can be recovered. Since the recovery is time consuming, the slicing requests are quickly satisfied from the roBDD while the requests for the dynamic

dependence graph can be satisfied using the additional information saved. Table 11 shows the percentage of control flow edges and data dependence edges that must be remembered to recover the full dynamic dependence graph as well as the space taken to store this information. The main idea of the recovery method is that given partial knowledge of the control flow path followed, the complete control flow path can be inferred. From the full control flow path many (not all) of the dynamic data dependences can be recovered. The ones that cannot be recovered must be remembered. The details of this recovery method can be found in [20].

Table 11. Dynamic Dependence Graph.

Program	Control Edges	Data Dep. Edges	Size (MB)
008.espresso	28%	7%	38
130.li	34%	4%	81
134.perl	26%	4%	13
099.go	22%	7%	43
147.vortex	21%	6%	26

6 Conclusions

In this paper we have shown that while forward computation of dynamic slices has some attractive features, a straightforward implementation of these algorithms as proposed in [3, 9] has very high space and time costs. We have developed a forward computation algorithm that uses roBDDs and SEQUITUR to drastically reduce the space and time costs of forward computations. We have shown that the Non-BDD algorithm that uses SEQUITUR but not roBDDs runs out of memory even for short program runs if *all slices* are kept. In contrast, when roBDDs are used, we are able to store 107-217 million dynamic slices arising during long program runs using only 28-392 MB of storage. In addition, the performance of the roBDD based forward computation method compares favorably with the performance of the LP and oFP backward computation algorithms. The one drawback of forward computation methods, the lack of dynamic dependence graph, can be overcome at a reasonable additional cost.

References

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

[4] A. Beszedes, C. Farago, Z.M. Szabo, J. Csirik, and T. Gyimothy, "Union Slices for Program Maintenance," *International Conference on Software Maintenance*, Montreal, Oct. 2002.

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

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

[7] T. Hoffner, "Evaluation and Comparison of Program Slicing Tools," *Technical Report*, Dept. of Computer and Information Science, Linkoping University, Sweden, 1995.

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

[9] B. Korel and S. Yalamanchili. "Forward computation of dynamic program slices," *International Symposium on Software Testing and Analysis*, August 1994

[10] J. Lin-Nielsen. "BuDDy, A Binary Decision Diagram Package," Department of Information Technology, Technical University of Denmark, *http://www.itu.dk/research/buddy/*.

[11] C. Meinel and T. Theobald, "Algorithms and Data Structures in VLSI Design," Springer, 1998.

[12] S. Minato, "Zero-suppressed BDDs and their Applications," *International Journal on Software Tools for Technology Transfer*, Volume 3, pages 156-170, 2001.

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

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

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

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

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

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

[19] X. Zhang and R. Gupta, "Cost Effective Dynamic Program Slicing," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[20] X. Zhang and R. Gupta, "Recovering Dynamic Dependence Graphs," Tech. Report TR04-01, University of Arizona, Department of Computer Science, Tucson, AZ, 2004.

[21] *Trimaran. Compiler Research Infrastructure*. Tutorial Notes, November 1997. http://www.trimaran.org.