Precise Dynamic Slicing Algorithms*

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, TX 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. In this paper we present the design and evaluation of three precise *dynamic slicing algorithms called the full preprocessing (FP), no preprocessing (NP) and limited preprocessing (LP) algorithms. The algorithms differ in the relative timing of constructing the dynamic data dependence graph and its traversal for computing requested dynamic slices. Our experiments show that the LP algorithm is a fast and practical precise slicing algorithm. In fact we show that while precise slices can be orders of magnitude smaller than imprecise dynamic slices, for small number of slicing requests, the LP algorithm is faster than an imprecise dynamic slicing algorithm proposed by Agrawal and Horgan.*

1. Introduction

The concept of program slicing was first introduced by Mark Weiser [10]. 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 an excellent survey of many of the proposed techniques and tools can be found in [8] and [5]. Other works on slicing have explored the applications of slicing in greater depth. Some examples of such works include the use of slicing in debugging sequential and distributed programs as well as testing sequential programs [2, 6].

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. 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. Recognizing the need for accurate slicing, Korel and Laski proposed the idea of *dynamic slicing* [7]. The data dependences that are exercised during a program execution are captured precisely and saved. Dynamic slices are constructed upon users requests by traversing the captured dynamic dependence information. An exception to this approach is various works on forward computation of dynamic slices [11, 12] which precompute all dynamic slices prior to users requests. While no experimental data is reported in [11, 12], the execution times of this approach can be expected to be high. Therefore in this paper we take the standard approach of computing only the dynamic slices requested by the user.

It has been shown that precise dynamic slices can be considerably smaller than static slices [9, 5]. The data in Table 1 shows the effectiveness of dynamic slicing. For each of the benchmark programs, we computed 25 distinct dynamic slices half way through the program's execution (`@Midpoint`) and another 25 at the end of program's execution (`@End`). The average (AVG), minimum (MIN) and maximum (MAX) precise dynamic slice sizes that were observed are given in the precise dynamic slices *PDS* column. In addition, the number of distinct statements in the program (*Static*) and the number of distinct statements that are executed (*Executed*) are also given. For example, program `126.gcc` contains 585491 static instructions and, during the collection of the execution trace, 170135 of these instructions were executed at least once. When 25 precise dynamic slices were computed, they had average, minimum and maximum sizes of 6614, 2, and 11860 instructions respectively. As we can see, the *PDS* values are much smaller that the *Static* and *Executed* values. Thus, dynamic slices greatly help in focusing the attention of the user to a small subset of statements during debugging.

While precise dynamic slices can be very useful, it is also well known that computing them is expensive. Therefore researchers have proposed *imprecise* dynamic slicing algorithms that trade-off precision of dynamic slicing with the costs of computing dynamic slicing. However, while such algorithms have the potential of reducing the cost of slicing,

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

Table 1. Effectiveness of Dynamic Slicing.

Program	Static	Instructions Executed	PDS			IDS-II/PDS			IDS-I/PDS		
			AVG	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX
`126.gcc` @ End	585491	170135	6614	2	11860	2.01	1	5188.25	3.82	1.75	13759.5
`126.gcc` @ Midpoint	585491	144504	2325.8	2	7405	2.69	1	1953.25	6.60	1	4804.75
`099.go` @ End	95459	61350	5382	4	8449	1.20	1	946.30	2.27	1	3328.04
`099.go` @ Midpoint	95459	55538	3560	2	6900	1.43	1	1321.83	2.55	1	5353.50
`134.perl` @ End	116182	21451	765	2	2208	3.43	1	96.15	4.26	1	973.75
`134.perl` @ Midpoint	116182	20934	601	2	2208	3.04	1	813.5	4.80	1	953.25
`130.li` @ End	31829	10958	834	2	834	5.96	1	41.9	12.87	1	667.5
`130.li` @ Midpoint	31829	10429	48	2	584	17.33	1	1036.00	53.40	4.35	1339.5
`008.espresso` @ End	74039	27333	350	2	1304	3.90	1	48.57	5.54	1	1273.8
`008.espresso` @ Midpoint	74039	21027	295	2	1114	3.65	1	300.25	9.78	1	1133.75
Average						4.46	1	1174.6	10.59	1.41	3358.7

they have also been found to greatly increase reported slice sizes thus diminishing their effectiveness. In [3], Atkinson *et al.* use *dynamic points-to* data to improve the accuracy of slicing. Unfortunately, their studies indicate that "improved precision of points-to data generally did not translate into significantly reduced slices". In [4], Gupta and Soffa proposed *hybrid slicing* technique which uses limited amounts of control flow information. However, it does not address the imprecision in data dependency computation in presence of pointers. We implemented two imprecise algorithms, *Algorithm I* and *Algorithm II*, proposed by Agrawal and Horgan [1] and compared the sizes of the imprecise dynamic slices (*IDS-I* and *IDS-II*) with corresponding precise dynamic slices (*PDS*). The ratios *IDS-I/PDS* and *IDS-II/PDS* are given in Table 1. As we can see, imprecise slices can be many times larger than precise dynamic slices. In the worst case, for program `gcc`, the imprecise dynamic slice *IDS-II* was over 5188 times larger than the precise dynamic slice. The *IDS-I* sizes are even larger. Therefore, so far, all attempts to address the cost issue of dynamic slicing have yielded unsatisfactory results. Given the inaccuracy of imprecise slicing algorithms, we conclude that it is worthwhile to spend effort into designing efficient precise dynamic slicing algorithms.

We observe that once a program is executed and its execution trace collected, precise dynamic slicing typically involves two tasks: *[preprocessing]* which builds a dependence graph by recovering dynamic dependences from the program's execution trace; and *[slicing]* which computes slices for given slicing requests by traversing the dynamic dependence graph. We present three precise dynamic slicing algorithms that differ in the degree of preprocessing they carry out prior to computing any dynamic slices. The *full preprocessing* (*FP*) algorithm builds the entire dependence graph before slicing. The *no preprocessing* (*NP*) does not perform any preprocessing but rather during slicing it uses *demand driven analysis* for recovering dynamic dependencies and caches the recovered dependencies for potential future reuse. Finally the *limited preprocessing* (*LP*) algorithm performs some preprocessing to first augment the execution trace with summary information that allows faster traversal of the trace and then during slicing it uses demand driven analysis to recover the dynamic dependences from the compacted execution trace. Our experience with these algorithms shows:

- The FP algorithm is impractical for real programs because it runs out of memory during the preprocessing phase as the dynamic dependence graphs are extremely large. The NP algorithm does not run out of memory but is slow. The LP algorithm is practical because it never runs out of memory and is also fast.
- The execution time of the practical LP algorithm compares well with that of the imprecise *Algorithm II* proposed by Agrawal and Horgan. The LP algorithm is even faster than *Algorithm II* if a small number of slices are computed. Also, the latency of computing the first slice using LP is 2.31 to 16.16 times less than the latency for obtaining the first slice by *Algorithm II*.

Thus, this paper shows that while imprecise dynamic slicing algorithms are too imprecise and therefore not attractive, a carefully designed precise dynamic slicing algorithm such as the LP algorithm is practical as it provides precise dynamic slices at reasonable space and time costs.

The remainder of the paper is organized as follows. In section 2 we present the precise slicing algorithms. In section 3 we present our experimental studies. Related work is discussed in section 4. Conclusions are given in section 5.

2 Precise Dynamic Slicing

The basic approach to dynamic slicing is to execute the program once and produce an execution trace which is processed to construct dynamic data dependence graph that in turn is traversed to compute dynamic slices.

The *execution trace* captures the complete runtime information of the program's execution that can be used by a dynamic slicing algorithm – in other words, there is sufficient information in the trace to compute precise dynamic slices. The information that the trace holds is the full *control* *flow trace* and *memory reference trace*. Therefore we know the complete path followed during execution and each point where data is referenced through pointers we know the address from which data is accessed.

A *slicing request* can be specified both in terms of a program variable and in terms of a memory address. The latter is useful if the slice is to be computed with respect to a field of a specific instance of a dynamically allocated object. Data slices are computed by taking closure over data dependences while full slices are computed by taking closure over both data and control dependences.

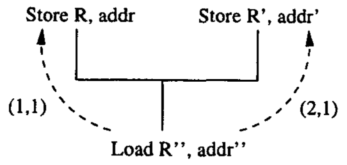
For a dynamic slice to be computed, dynamic dependences that are exercised during the program execution must be identified by processing the trace. The precise algorithms that we present differ in the degree to which dependences are extracted from the trace prior to dynamic slicing. The *full preprocessing* (*FP*) algorithm follows an approach that is typical of precise algorithms proposed in the literature [7, 1]. The execution trace is fully preprocessed to extract all dependences and the full dynamic dependence graph is constructed. Given this graph, any dynamic slicing request can be handled by appropriate traversal of the graph. The *no preprocessing* (*NP*) algorithm does not precompute the full dynamic data dependence graph. Instead, dynamic data dependences are extracted in a *demand-driven* fashion from the trace during the handling of dynamic slicing requests. Therefore each time the execution trace is examined, only dependences relevant to the slice being computed are extracted from the trace. Finally the *limited preprocessing* (*LP*) algorithm differs from the *NP* algorithm in that it augments the execution trace with trace-block summaries to enable faster traversal of the execution trace during the demand-driven extraction of dynamic dependences. Next we describe the above algorithms in detail. Our discussion primarily focuses on data dependences because the presence of pointers primarily effects the computation of data dependences.

2.1 Full Preprocessing

In developing our slicing algorithms one goal that we set out to achieve was to develop a dynamic data dependence graph representation that would not only allow for computation of precise dynamic slices, but, in addition, support computation of a dynamic slice for any variable or memory address at *any execution point*. This property is not supported by the precise dynamic slicing algorithm of Agrawal and Horgan [1]. We take the statement level control flow graph representation of the program and add to it edges corresponding to the data dependences extracted from the execution trace. The *execution instances* of the statements involved in a dynamic data dependence are explicitly indicated on the dynamic data dependence edges thus allowing the above goal to be met.

Consider a situation of memory dependences between stores and loads. A statically distinct load/store instruction may be executed several times during program execution. When this happens different instances of a load instruction may be dependent upon different store instructions or different instances of the same store instruction. For precise recording of data dependences we associate the instances of load and store instructions among which dependences exist. A slicing request not only identifies the use of a variable by a statement for which the slice is needed, but also the specific instance of the statement for which the slice is needed.

Consider the example shown below in which we assume that the load instruction is executed twice. The first instance of the load reads the value stored by the store on the left and the second instance of the load reads the value stored by the store on the right. In order to remember this information we label the edge from the load to the store on the left/right with (1,1)/(2,1) indicating that the first/second instance of the load's execution gets its value from first instance of execution of the store on the left/right respectively. Therefore when we include the load in the dynamic slice, we do not necessarily include both the stores in the dynamic slice. If the dynamic slice for the first instance of the load is being computed, then the store on the left is added to the slice while if the dynamic slice of the second instance of the load is being computed then the store on the right is added to the slice.



Thus, in summary this precise dynamic slicing algorithm first preprocesses the execution trace and introduces labeled dependence edges in the dependence graph. During slicing the instance labels are used to traverse only relevant edges. We refer to this algorithm as the *full preprocessing* (FP) algorithm as it fully preprocesses the execution trace prior to carrying out slice computations.

The example in Figure 1 illustrates this algorithm. The dynamic data dependence edges, from a use to its corresponding definition, for a given run are shown. For readability we have omitted the dynamic data dependence edges for uses in branch predicates as computation of data slices do not require these edges; only the edges for all non-predicate uses are shown. Edges are labeled with the execution instances of statements involved in the data dependences. The precise dynamic slice for the value of z used in the only execution of statement 16 is given. The data dependence edges traversed during this slice computation include: $(16_1, 14_3)$, $(14_3, 13_2)$, $(13_2, 12_2)$, $(13_2, 15_3)$, $(15_3, 3_1)$, $(15_3, 15_2)$, $(15_2, 3_1)$, $(15_2, 15_1)$, $(15_1, 3_1)$, and $(15_1, 4_1)$.

Note that it is equally easy to compute a dynamic slice at any earlier execution point. For example, let us compute the slice corresponding to the value of x that is used during the first execution of statement 15. In this case we will follow the data dependence edge from statement 15 to statement 4 that is labeled $(1,1)$; thus giving us the slice that contains statements 4 and 15.

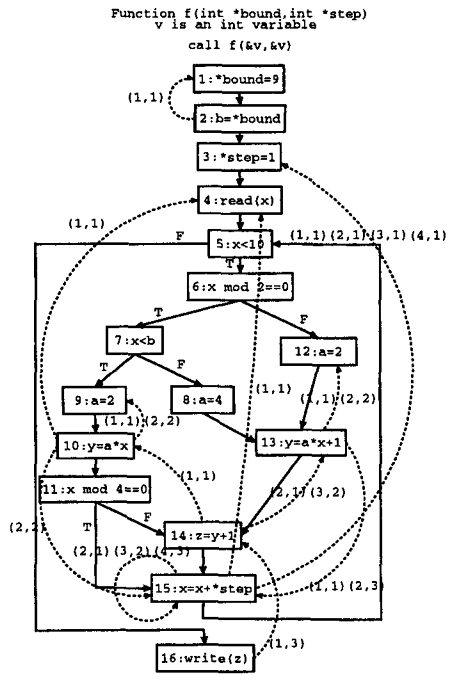


Figure 1. The path taken by the program on input $x = 6$ is $\{1_1 2_1 3_1 4_1 5_1 6_1 7_1 9_1 10_1 11_1 14_1 15_1 5_2 6_2 12_1 13_1 14_2 15_2 5_3 6_3 7_2 9_2$ $10_2 11_2$ $15_3 5_4 6_4 12_2 13_2 14_3 15_4 5_5 16_1\}$. The precise dynamic slice for use of z by the only execution of statement 16 is {16,14,13,12,4,15,3}.

2.2 No Preprocessing

The *FP* algorithm first carries out all the preprocessing and then begins slicing. For large programs with long execution runs it is possible that the dynamic dependence graph requires too much space to store and too much time to build. In fact our experiments show that too often we run out of memory since the graphs are too large. For this reason we propose another precise algorithm that does not perform any preprocessing. We refer to this algorithm as the *no preprocessing* (NP) algorithm.

In order to avoid a priori preprocessing we employ *demand driven analysis* of the trace to recover dynamic dependences. When a slice computation begins we traverse the trace backwards to recover the dynamic dependences required for the slice computation. For example, if we need the dynamic slice for the value of some variable v at the end of the program, we traverse the trace backwards till the definition of v is found and include the defining statement in the dynamic slice. If v is defined in terms of value of another variable w, we resume the traversal of the trace starting from the point where traversal had stopped upon finding the definition of v and so on. Note that since definitions we are interested will always appear earlier than the uses, we never need to traverse the same part of the trace twice during a single slice computation.

In essence this algorithm performs partial preprocessing for extracting dynamic dependences relevant to a slicing request as part of the slice computation. It is possible that two different slicing requests involve common dynamic dependences. In such a situation, the demand driven algorithm will recover the common dependences from the trace during both slice computations. To avoid this repetitive work we can *cache* the recovered dependences. Therefore at any given point in time, all dependences that have been computed so far can be found in the cache. Therefore when a dependence is required during a slice computation, the cache is first checked to see if the dependence is already known. If the dependence is cached we can directly access it; otherwise we must recover it from the trace. Thus, at the cost of maintaining a cache, we can avoid repeated recovery of same dependences from the execution trace.

We will refer to the two versions of this demand driven algorithm, that is, without caching and with caching, as *no preprocessing without caching* (NPwoC) and *no preprocessing with caching* (NPwC) algorithms.

As an illustration of this algorithm let us reconsider the example of Figure 1. When the *NP* algorithm is used, initially the flow graph does not contain any dynamic data dependence edges. Now let us say the slice for z at the only execution of statement 16 is computed. This will cause a single backward traversal of the trace through which the data dependence edges $(16_1, 14_3)$, $(14_3, 13_2)$, $(13_2, 12_2)$, $(13_2, 15_3)$, $(15_3, 3_1)$, $(15_3, 15_2)$, $(15_2, 3_1)$, $(15_2, 15_1)$, $(15_1, 3_1)$, and $(15_1, 4_1)$ are extracted. When caching is used, in addition to obtaining the slice, these edges are added to the program flow graph. Now if the slice for the use of x in the 3rd instance of statement 14 is computed, all dependences required are already present in the graph and thus the trace in not reexamined. On the other hand, if the slice for the use of x by the 2nd instance of statement 10 is requested, the trace is traversed again to extract additional dynamic data dependences.

2.3 Limited Preprocessing

While the *NP* algorithm described above addresses the space problem of the *FP* algorithm, this comes at the cost of increased time for slice computations. The time required to traverse a long execution trace is a significant part of the cost of slicing. While the *FP* algorithm traverses the trace only once for all slicing requests, the *NP* algorithm often traverses the same part of the trace multiple times, each

Table 2. Benchmark Characteristics

Program	Lines of C Code	Num. of Funcs	Code Size (Bytes)	Instructions Executed			Trace Size (Bytes)		
				(1)	(2)	(3)	(1)	(2)	(3)
`126.gcc`	207483	2001	283070464	103156356	112037569	104650291	500000391	500000757	500000215
`099.go`	29629	372	27512832	117503888	120467852	62271885	500000788	500000747	250000022
`134.perl`	27044	277	87438967	32729474	55416461	111023934	132807387	250000192	500000438
`130.li`	7741	357	7964444	122017736	21590453	109323294	581574086	99999705	500000365
`008.espresso`	14850	361	22432108	1142979	37512537	20203629	4665571	124988145	79512147

time recovering different relevant dependences for a different slicing request.

In light of the above discussion we can say that *NP* algorithm does too little preprocessing leading to high slicing costs while *FP* algorithm does too much preprocessing leading to space problems. For example, during our experiments we found that a run of *FP* over a trace of around one hundred million instructions for `126.gcc` is expected to generate a graph of size five gigabytes. Therefore next we propose an algorithm that strikes a balance between *preprocessing* and *slicing* costs. In this precise algorithm we first carry out *limited preprocessing* of the execution trace aimed at augmenting the trace with summary information that allows faster traversal of the augmented trace. Then we use demand driven analysis to compute the slice using this augmented trace. We refer to this algorithm as the *limited preprocessing* (LP) algorithm.

This algorithm speeds up trace traversal as follows: the trace is divided into *trace blocks* such that each trace block is of a fixed *size*. At the end of each trace block we store a *summary of all downward exposed definitions* of variable names and memory addresses. During the backward traversal for slicing, when looking for a definition of a variable or a memory address, we first look for its presence in the summary of downward exposed definitions. If a definition is found, we traverse the trace block to locate the definition; otherwise using the *size* information we skip right away to the start of the trace block.

Since the summary information contains only downward exposed definitions, the number of checks performed to locate the definition being sought is smaller when the summary information is used in contrast with using the trace itself. Thus, if the block is skipped, the net effect is fewer comparisons between the address of the variable whose definition is being sought and addresses defined within the trace block. On the other hand, if the block is not skipped, more comparisons are needed because both the summary information and the trace block are examined till the definition is located. Note that the cost of comparisons, and the size of the summary information, can also be reduced by representing the summary information using bit vectors. Moreover, since the dynamic slices are quite small in comparison to the trace size, it is expected that many of the traversed trace blocks will not contribute to the dynamic slice and would therefore be skipped after examination of the summary information. Thus, fewer address comparisons will be performed in practice, and even more importantly, the I/O cost during trace traversal will be greatly reduced.

3 Experimental Evaluation

3.1 Implementation

For our experimentation we used the *Trimaran* system that takes a C program as its input and produces a lower level intermediate representation (IR) which is actually the machine code for an EPIC style architecture. 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. Our implementation supports computation of both data slices as well as full slices that are based both upon data and control dependences. However, when computing slices for C programs, the key source of imprecision is the presence of pointers. Therefore in our experiments we focus primarily upon computation of dynamic data slices.

In the low level IR the usage of registers and presence of memory references has been made explicit by the introduction of load and store instructions. An interpreter for the IR is available which is used to execute instrumented versions of the IR for obtaining execution traces consisting of both the control flow trace and memory trace. In our implementation we read the execution trace in blocks and buffer it to reduce the I/O cost. Some of the programs we use make use of *longjmps* which makes it difficult to keep track of the calling environment when simulating the call stack. We handle this problem by instrumenting the program to explicitly indicate changes in calling environment as part of the trace. This additional information in the trace is used during traversal of the trace.

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.

Table 3. Precise Dynamic Slice Sizes for Additional Inputs.

Program	Static	Instructions Executed(2)	PDS(2)			Instructions Executed(3)	PDS(3)		
			AVG	MIN	MAX		AVG	MIN	MAX
126.gcc @ End	585491	136269	1268	2	10702	194162	7359	2	15388
099.go @ End	95459	56051	4982	2	6934	46497	1268	2	5178
134.perl @ End	116182	15327	98	2	611	22897	151	2	599
130.li @ End	31829	8462	21	2	232	8854	19	2	323
008.espresso @ End	74039	22897	448	4	1443	19356	227	2	1229

3.2 Benchmark Characteristics

The programs used in our experiments include `008.espresso` from the `Specint92` suite, and `130.li`, `134.perl`, `099.go` and `126.gcc` from the `Specint95` suite. The attributes of the programs, including the number of lines of `C` code, number of functions, and the generated code size are given in Table 2. Each of the programs were executed on three different inputs and execution traces for the three inputs were collected. The number of instructions executed and the sizes of execution traces for these program runs are also given in Table 2. We can see that both the programs and the execution traces collected are quite large.

The system used in our experiments is a 2.2 GHz Pentium 4 linux workstation with 1.0 GB RAM and 1.0 GB of swap space.

3.3 Precise Slicing Algorithms

In order to study the behaviors of the proposed precise dynamic slicing algorithms we computed the following slices. We collected execution traces on 3 different input sets for each benchmark. For each execution trace, we computed 25 different slices. These slices were performed for latest executions of 25 distinct values loaded using load instructions by the program. That is, these slices were computed with respect to the end of program's execution (`@ End`). For the first program input, in addition, we computed 25 slices at an additional point in program's execution: `@ midpoint` - after half of the execution.

Slice sizes. Let us first examine the sizes of slices to establish the relevance of dynamic slicing for program debugging. In Table 1, in the introduction, the precise dynamic slice sizes of the programs on the first input for both `@ End` and `@ midpoint` were given and it was observed that the number of statements in the dynamic slice is a small fraction of the distinct statements that are actually executed. Thus, they are quite useful during debugging. In Table 3, the precise dynamic slice sizes for the other two program inputs for `@ End` are given. As we can see, similar observations hold for different inputs for each of the benchmarks. Thus, dynamic slicing is effective across different inputs for these pointer intensive benchmarks.

Slice computation times. Next we consider the execution times of *FP*, *NPwoC*, *NPwC*, and *LP* algorithms. Our implementation of the *LP* algorithm does not use caching. Figure 2 shows the cumulative execution time in seconds as slices are computed one by one. The three columns in Figure 2 correspond to the three different inputs. These graphs include both the preprocessing times and slice computation times. Therefore for algorithms which perform preprocessing the time at which the first slice is available is relatively high as before the slice is computed preprocessing must be performed.

First we note that in very few cases the *FP* runs to completion, more often it runs *out-of-memory* (OoM) even with 1 GB of swap space available to the program and therefore no slices are computed. Clearly this latter situation is unacceptable. This is not surprising when one considers the estimated graph sizes for these program runs given in Table 4 (the estimates are based upon the number of dynamic dependences).

Table 4. Estimated Full Graph Sizes.

Program	Size (MB)		
	(1)	(2)	(3)
126.gcc @ End	4931.4	5064.7	5055.9
099.go @ End	2366.6	2276.5	1076.2
134.perl @ End	1975.8	5629.2	8977.5
130.li @ End	1808.6	316.2	1614.3
008.espresso @ End	26.4	755.5	409.3

When we consider the other precise algorithms, that is, *NPwoC*, *NPwC* and *LP* algorithms, we find that they all successfully compute all of the slices. Therefore clearly these new algorithms make computation of precise dynamic slices feasible.

Now let us consider the *no preprocessing* algorithms in greater detail. We note that for the *NPwoC* algorithm there is a linear increase in the cumulative execution time with the number of slices. This is to be expected as each slicing operation requires some traversal of the execution trace. Moreover we notice that for *NPwC* that uses caching, the cumulative execution time increases less rapidly than *NPwoC*, which does not use caching, for some programs but not for others. This is because in some cases dependences are found in the cache while in other cases they are not present in the cache. In fact when there are no cache hits, due to the time spent on maintaining the cache, *NPwC* runs slower

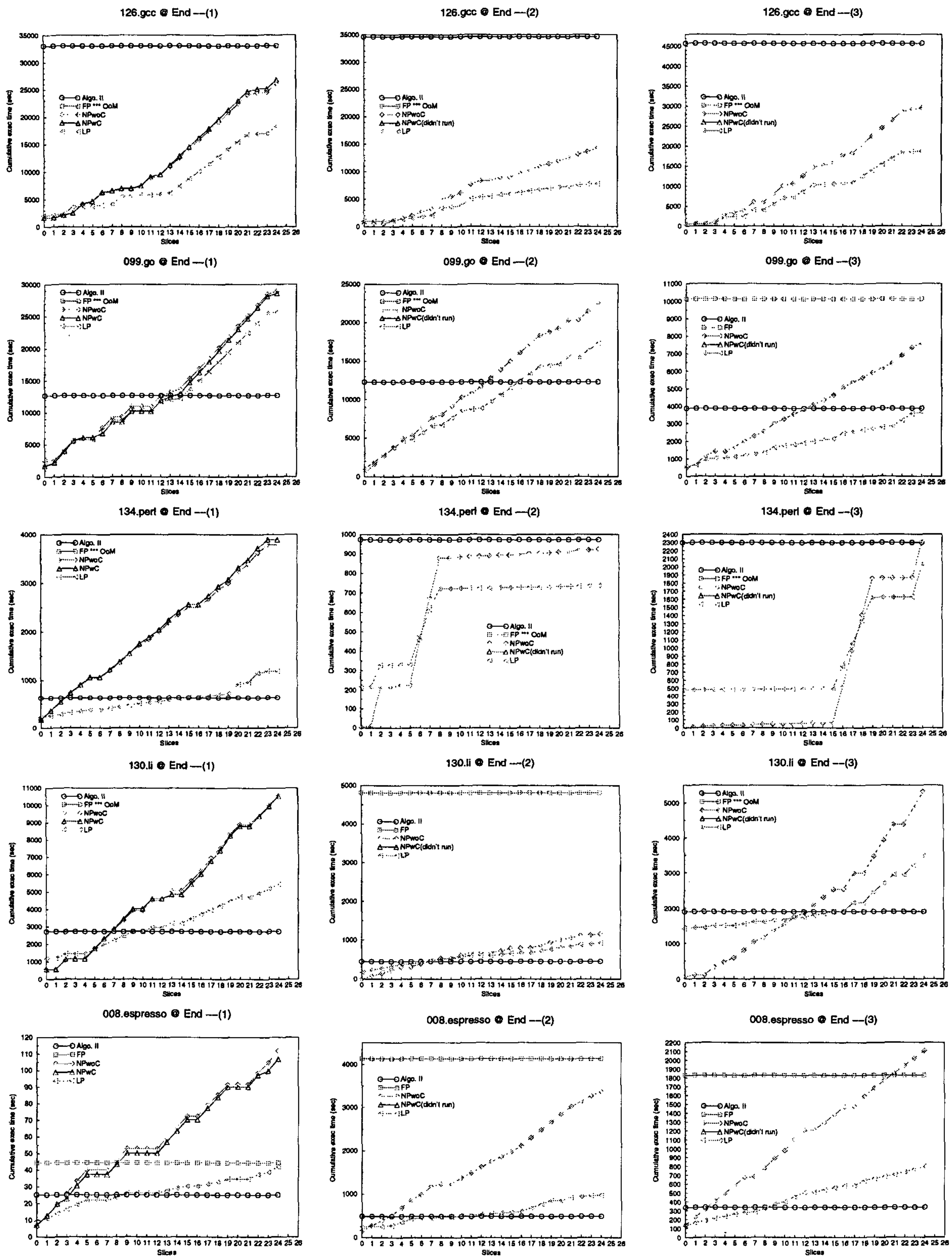


Figure 2. Execution Times.

than *NPwoC*. Since the impact of caching was minimal for the first input, we did not run the *NPwC* version on the other two inputs.

When we look at the results of using *LP* algorithm we find that the limited preprocessing indeed pays off. The *LP* cumulative execution time rises much more slowly than the *NPwoC* and *NPwC* curves. Since limited preprocessing requires only a single forward traversal of the trace, its preprocessing cost is small in comparison to the savings it provides during slice computations. The execution times of the *LP* algorithm are 1.13 to 3.43 times less than the *NP* algorithm for the first input set (see Table 5). This is not surprising when one considers the percentage of trace blocks that are skipped by the *LP* algorithm (see Table 6). Each trace block consisted of the trace of 200 basic blocks. Varying the trace block size makes little difference as the percentage of trace blocks skipped is quite large.

Table 5. Cumulative times: *NP* vs *LP*.

Program	*NP/LP*		
	(1)	(2)	(3)
`126.gcc` @ End	1.43	1.833	1.58
`099.go` @ End	1.13	1.31	2.09
`134.perl` @ End	3.19	1.25	1.14
`130.li` @ End	1.95	1.25	1.53
`008.espresso` @ End	2.67	3.43	2.65
Average	2.07	1.81	1.80

Table 6. Trace Blocks Skipped by *LP*.

Program	% Blocks Skipped		
	(1)	(2)	(3)
`126.gcc` @ End	90.43	97.63	89.39
`099.go` @ End	57.52	65.38	91.7
`134.perl` @ End	92.42	98.99	98.26
`130.li` @ End	99.02	99.51	99.70
`008.espresso` @ End	96.6	97.15	98.68
Average	87.20	91.73	95.65

3.4 LP vs. Imprecise Algorithm II

In this section we compare the performance of our best precise dynamic slicing algorithm, the *LP* algorithm, with Agrawal and Horgan's *Algorithm II*. We do not include data for *Algorithm I* because as shown by the data presented in Table 1, *Algorithm I* is extremely imprecise even in comparison to *Algorithm II*.

Before describing the results it is important to understand the differences between the *LP* algorithm and *Algorithm II*. Both algorithms do not run out of memory and hence solve the problem of large graph sizes. The *LP* algorithm solves this problem by demand-driven construction of relevant part of the precise dynamic dependence graph. *Algorithm II* solves the same problem by constructing an imprecise dynamic dependence graph where the instances of statements among which dependences exist are not remembered. This approximation greatly reduces the size of the graph which is constructed in a single pass over the trace. Thus all the preprocessing is carried out once in the beginning and then slices can be computed very quickly by traversing this graph.

A question that may arise is whether the performance of *Algorithm II* can be further improved by applying the demand-driven approach and limited preprocessing used by the *LP* algorithm. Since the approximate dynamic dependence graph constructed by *Algorithm II* is already small, there is no point in building it in a demand-driven fashion. Moreover, given the approach taken by *Algorithm II*, the demand-driven construction of the approximate dynamic dependence graph will only further slow down *Algorithm II*. This is because *Algorithm II* constructs the complete approximate dynamic dependence graph in a single pass over the trace. If demand-driven approach is used, to extract subset of dependences from the trace, the entire trace may have to be traversed for extracting these dependences. Thus, repeated passes over the trace would have to be carried out to extract different subsets of dependences which will further slow down *Algorithm II*. On the other hand, the *LP* algorithm has to build the precise dependence graph in a demand-driven fashion because the complete graph is too large. Furthermore, since *Algorithm II* traverses the trace only once, there is no point in augmenting the trace with summary information because such augmentation would also require a complete traversal of the trace.

Slice sizes. The data presented in the introduction already showed that precise dynamic slices are much smaller than the imprecise dynamic slices computed using *Algorithm II*. In Table 7 similar data for the two additional inputs is given. As we can see, the same observation holds across these additional inputs.

We have already compared the slice sizes of *LP* and *Algorithm II*. However, since the results of such comparisons are dependent upon the variables for which slicing is performed, we also developed a novel *slice independent* approach for comparing the algorithms by simply comparing the dynamic dependence graphs constructed by them and measuring the imprecision in these dependence graphs that is introduced by *Algorithm II*. This method is motivated by the fact that the imprecision in dynamic dependence graph constructed by *Algorithm II* is the root cause of resulting imprecision in the dynamic slices computed by this algorithm.

The number of dynamic dependences recovered by the precise algorithm is exact. However, when the imprecise algorithm is used, the imprecision is introduced in the dynamic dependence graph in form of false dependences. Therefore if we compute the equivalent number of dynamic dependences for the imprecise algorithm they will be higher than those for the precise algorithm. The greater the number

Table 7. IDS-II vs. LP: Inputs (2) and (3).

Program	IDS-II/PDS(2)			IDS-II/PDS(3)		
	AVG	MIN	MAX	AVG	MIN	MAX
`126.gcc` @ End	6.80	1	4167	2.03	1	6533.5
`099.go` @ End	1.39	1	4124	1.95	1	162.5
`134.perl` @ End	8.65	1	737	8.92	1	703
`130.li` @ End	9.09	1	178.5	3.08	1	21.50
`008.espresso` @ End	2.09	1	60.95	8.26	1	1561.5
Average	5.6	1	1853.49	4.85	1	1796.46

Table 8. Slice Independent Comparison of Algorithms.

Program	Number of Dynamic Memory Dependences					
	Input (1)		Input (2)		Input (3)	
	Algorithm-II	Precise	Algorithm-II	Precise	Algorithm-II	Precise
`126.gcc`	35378836	8632906	74601341	8546040	38311969	8686760
`099.go`	148995060	10079908	114054794	8768685	42549505	3852822
`134.perl`	22114062	2923947	6853213	4668814	13566847	9307883
`130.li`	310899820	10150987	6175836	1828449	37056110	9284206
`008.espresso`	374329	77217	14414220	1232534	6686934	918732

of false dependences, the greater is the degree of imprecision.

Let us say statement S is executed many times and some of its instances are dependent upon values computed by statement T and others on values computed by statement U and in addition S refers to some addresses that are also written by statement V although there is no true dependence between S and V. *LP* algorithm makes each instance of S dependent upon a single instance of either T or U. *Algorithm II* introduces twice the number of dependences as the *LP* algorithm because it makes each instance of S dependent upon both T and U.

We computed the equivalent number of dynamic memory dependences (i.e., dependences between a store and a load operation) present in the dynamic dependence graphs constructed for *Algorithm II* and *LP* algorithm. The results of this computation are given in Table 8. As we can see, the number of dynamic memory dependences for *Algorithm II* are several times that of the number of dynamic memory dependences for the *LP* algorithm. For example, for the first input set, `126.gcc`'s execution produces a dynamic data dependence graph containing 35378836 memory dependences for *Algorithm II* and 8632906 memory dependences for the precise algorithm. Thus, imprecision of *Algorithm II* leads to a 4.1 fold increase in the number of dynamic memory dependences.

We also compared the performance of the two algorithms for two types of slices, *data slices* (based upon transitive closure over data dependences) and *full slices* (based upon transitive closure over both control and data dependences). We observed that while the overall sizes of full slices were several times (3 to 9 times) greater than data slices, the relative performance of the two algorithms was similar. For the first input, while precise data slices computed by the *LP* algorithm were 1.2 to 17.33 times smaller than imprecise data slices computed by *Algorithm II*, precise full slices were 1.09 to 1.81 times smaller than imprecise full slices. More detailed data is omitted due to space limitations.

Execution times. The execution times for *Algorithm II* for slices computed at the end of execution are shown in Figure 2. Table 9 shows the preprocessing and slice computation times of these two algorithms. When we compare the execution times of the two algorithms we observe the following:

- `@ End` of execution the total time (i.e., sum of preprocessing and slicing times) taken by *LP* is 0.55 to 2.02 times the total time taken by *Algorithm II*.
- `@ Midpoint` of execution the total time taken by *LP* is 0.51 to 1.86 times that of *Algorithm II*.
- The latency of producing the results of the first slice using *LP* is 2.31 to 16.16 times smaller than that of producing the first slice using *Algorithm II*.

On examining the graphs in Figure 2 we notice that if we compute only a small number of slices, then the precise *LP* algorithm in fact outperforms *Algorithm II* even in terms of the runtime performance. This is because *Algorithm II* requires that all preprocessing be performed before slicing can begin while *LP* performs much less preprocessing. For each program there is a number L such that if at most L slices are computed, *LP* algorithm outperforms *Algorithm II*. The value of L is higher for execution runs with longer traces as the length of the trace determines the preprocessing time for *Algorithm II*. For the slicing of `gcc @ End` we can compute all 25 slices precisely using *LP* algorithm in time which is less than the time it takes for *Algorithm II* to

Table 9. Preprocessing + Slicing Times: Algo. II vs. LP for Input (1).

Program	Algorithm II	LP
`126.gcc` @ End	33014.1 + 11.46	727.82+17556.48
`126.gcc` @ Midpoint	9271.64+ 5.24	347.61+4371.21
`099.go` @ End	12671.8 + 5.37	893.92+24766.14
`099.go` @ Midpoint	6039.56 +3.21	455.17+8582.18
`134.perl` @ End	631.32 + 0.76	190.18+996.82
`134.perl` @ Midpoint	407.83+0.32	94.16+543.20
`130.li` @ End	2725.32 + 3.43	858.08+4493.45
`130.li` @ Midpoint	1747.38+1.89	464.71+2736.54
`008.espresso` @ End	25.15 + 0.11	7.83+33.98
`008.espresso` @ Midpoint	15.39+0.03	3.74+25.01

carry out preprocessing. On the other hand, for `espresso @ End` we can compute around 10 slices using *LP* algorithm in the same amount of time as it takes *Algorithm II* to carry out preprocessing. We also compared the performance of *LP* and *Algorithm II* for the first input trace at execution midpoint (i.e., `@ Midpoint`). The results are presented in Figure 3. As we can see, at this earlier point in program executions similar observations hold.

4 Related Work

Agrawal and Horgan proposed two imprecise and two precise dynamic slicing algorithms in [1]. We have already compared the performance of our algorithms with the imprecise algorithms in detail. The first precise algorithm they propose, *Algorithm III*, is quite similar to our *FP* algorithm. The difference is in the dynamic dependence graph representation. While *FP* labels dependence edges with instances, Agrawal and Horgan construct multiple instances of nodes and edges. In practice, we found our representation to be more compact.

To reduce the size of the dependence graph, Agrawal and Horgan also proposed another precise algorithm which is *Algorithm IV* in their paper. *Algorithm IV* is based upon the idea of *forward computation* of dynamic slices where slices for all variables can be maintained at all times, and when a statement is executed, the new slice of the variable just defined can be computed from the slices of the variables whose values are used in the definition. *Algorithm IV* maintains the current dynamic slices in terms of the dynamic dependence graph. A new node is added to the dynamic dependence graph only if following the execution of a statement the dynamic slice of the defined variable changes. Thus, the size of the graph is bounded by the number of different dynamic slices. As shown in [8], a program of size n can have $O(2^n)$ different dynamic slices.

Essentially *Algorithm IV* precomputes all of the dynamic slices. While this idea results in space savings, the precomputation time of *Algorithm IV* can be reasonably assumed to be much higher than the preprocessing time in *FP*, in

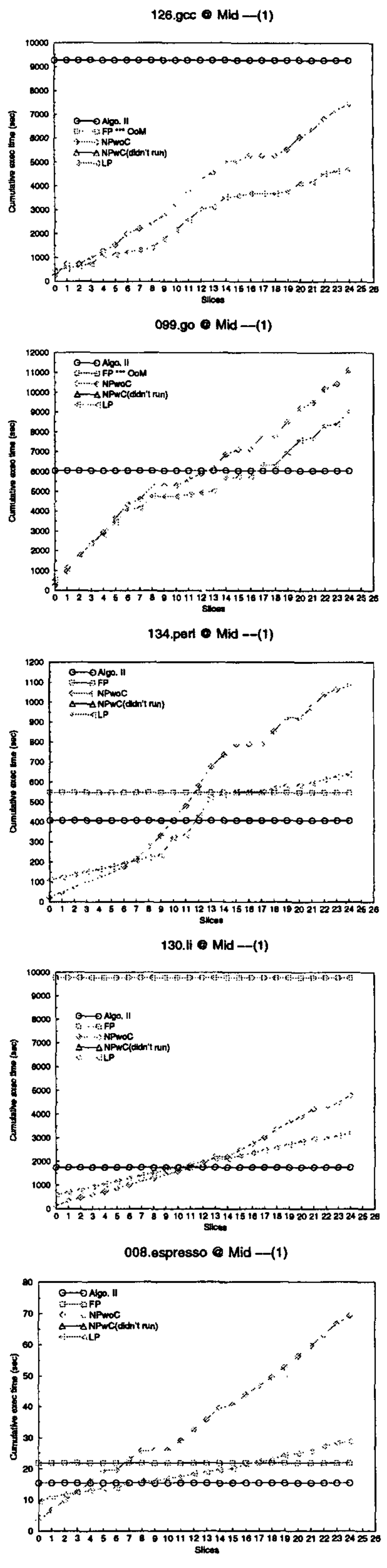


Figure 3. Execution Times `@Midpoint`.

which the direct dependences are merely added as edge labels but no slices are computed. Given the fact that *LP* is faster than *FP*, it is going to perform even better in comparison to *Algorithm IV*. Furthermore, the dynamic dependence graph produced by *Algorithm IV* can be used only to compute dynamic slices for the last definitions of variables. All the algorithms we have developed can be used to compute dynamic slices corresponding to any executed definition of any variable at any program point. In other words, in order to produce a compacted graph, *Algorithm IV* sacrifices some of the functionality of *Algorithm III*.

In [11] another algorithm for forward computation of dynamic slices was introduced which precomputes and stores all dynamic slices on disk and later accesses to them in response to users requests. This algorithm saves sufficient information so that dynamic slices at any execution point can be obtained. Like *Algorithm IV*, it will also take a long time to respond to user's first request due to the long preprocessing time. Some applications of dynamic slicing, such as debugging, may involve only a small number of slicing requests. Thus, the large amount of preprocessing performed is not desirable. Our *demand driven* approach represents a much better choice for such situations.

Korel and Yalamanchili [12] introduced another forward method which computes executable dynamic slices. Their method is based on the notion of *removable blocks*. A dynamic slice is constructed from the original program by deleting *removable blocks*. During program execution on each exit from a block, the algorithm determines whether the executed block should be included in a dynamic slice or not. It is reported in [8] that executable dynamic slices produced may be inaccurate in the presence of loops.

Finally no experimental data is presented to evaluate the forward computation of dynamic slices in any of the above works [1, 11, 12]. In this paper we have shown that it is important to consider practical design tradeoffs when developing a precise backward slicing algorithm. Any similar issues that may exist in the design of algorithms that perform forward computation of dynamic slices have yet to be studied by anyone.

5 Conclusions

In this paper we have shown that a careful design of a dynamic slicing can greatly improve its practicality. We designed and studied three different precise dynamic slicing algorithms: *FP*, *NP*, and *LP*. We made the use of *demand driven analysis* (with and without caching) and *trace augmentation* (with trace block summaries) to achieve practical implementations of precise dynamic slicing. We demonstrated that the precise *LP* algorithm which first performs limited preprocessing to augment the trace and then uses demand driven analysis performs the best. In comparison to the imprecise *Algorithm II* it runs faster when a small number of slices are computed. Also, the latency of computing the first slice using *LP* is 2.31 to 16.16 times less than the latency for obtaining the first slice by *Algorithm II*.

In conclusion this paper shows that while imprecise dynamic slicing algorithms are too imprecise and therefore not an attractive option, a carefully designed precise dynamic slicing algorithm such as the *LP* algorithm is practical as it provides precise dynamic slices at reasonable space and time costs.

References

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

[2] H. Agrawal, R. DeMillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software Practice and Experience* (SP&E), Vol. 23, No. 6, pages 589-616, 1993.

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

[4] R. Gupta and M.L. Soffa, "Hybrid Slicing: An Approach for Refining Static Slices using Dynamic Information," *ACM SIGSOFT 3rd Symposium on the Foundations of Software Engineering* (FSE), pages 29-40, October 1995.

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

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

[7] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters* (IPL), Vol. 29, No. 3, pages 155-163, 1988.

[8] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages* (JPL), Vol. 3, No. 3, pages 121-189, September 1995.

[9] G. Venkatesh, "Experimental Results from Dynamic Slicing of C Programs," *ACM Transactions on Programming Languages and Systems* (TOPLAS), Vol. 17, No. 2, pages 197-216, 1995.

[10] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering* (TSE), Vol. SE-10, No. 4, pages 352-357, 1982.

[11] 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* (CSMR), March 2001.

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