

Cost and Precision Tradeoffs of Dynamic Data Slicing Algorithms

XIANGYU ZHANG and RAJIV GUPTA

The University of Arizona

and

YOUTAO ZHANG

University of Texas at Dallas

Dynamic slicing algorithms are used to narrow the attention of the user or an algorithm to a relevant subset of executed program statements. Although dynamic slicing was first introduced to aid in user level debugging, increasingly applications aimed at improving software quality, reliability, security, and performance are finding opportunities to make automated use of dynamic slicing. In this paper we present the design and evaluation of three *precise* dynamic data 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 data slices. Our experiments show that the LP algorithm is a fast and practical precise data slicing algorithm. In fact we show that while precise data slices can be orders of magnitude smaller than imprecise dynamic data slices, for small number of data slicing requests, the LP algorithm is faster than an imprecise dynamic data slicing algorithm proposed by Agrawal and Horgan.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Debuggers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, testing tools, tracing*

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Program slicing, data dependences, pointer references, debugging

1. INTRODUCTION

The concept of static program slicing was first introduced by Weiser [1979, 1982]. The program slice corresponding to a variable at a specific program

A preliminary version of this article appeared in the Proceedings of the ICSE 2003.

This work was supported by grants from Intel, IBM, Microsoft, and National Science Foundation (NSF) grants CCR-0324969, CCR-0220334, CCR-0208756, CCR-0105355, and EIA-0080123 to the University of Arizona.

Authors' addresses: X. Zhang and R. Gupta, The University of Arizona, Department of Computer Science, Gould-Simpson Bldg., 1040 East Fourth Street, Tucson, AZ 85721; email: {xyzhang, gupta}@cs.arizona.edu; Y. Zhang, Computer Science Department, EC 31, The University of Texas at Dallas, Richardson, TX 75083; email: zhangyt@utdallas.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2005 ACM 0164-0925/05/0700-0631 \$5.00

point is defined to contain the subset of program statements which can potentially contribute to the computation of the value of the variable across all program executions. Weiser introduced program slicing as a debugging aid and gave the first *static slicing* algorithm. According to this algorithm the static slice is computed by taking a transitive closure over data and control dependences that directly or indirectly influence the value of the variable at a program point. 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 Tip [1995] and Hoffner [1995]. Other works on slicing have explored the applications of static 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 [Agrawal et al. 1993; Kamkar 1993].

For programs that make extensive use of pointers, the highly conservative nature of data dependence analysis leads to highly imprecise (i.e., considerably larger) program slices [Mock et al. 2002]. In other words, statements are included in the slice that do not truly influence the value of the variable. Since the objective of slicing is to focus the attention of the user or an algorithm to the relevant subset of program statements, conservatively computed large slices are clearly undesirable. Recognizing the need for precise slicing, Korel and Laski [1988] proposed the idea of *dynamic slicing*. The dependences that are exercised during a program execution are captured precisely and saved in form of a *dynamic dependence graph*. Dynamic program slices are constructed in response to requests by traversing the captured dynamic dependence information. It has been observed that precise dynamic slices can be considerably smaller than static slices [Venkatesh 1995; Hoffner 1995].

The importance of dynamic slicing extends well beyond debugging of programs [Agrawal et al. 1993; Duesterwald et al. 1992a; Korel and Rilling 1997]. Increasingly applications aimed at improving software quality, reliability, security, and performance are making automated use of dynamic slicing algorithms. For example, dynamic slicing algorithms are being used for: detecting spyware that has been installed on systems without the user's knowledge (S. Jha, private communication, 2003) carrying out dependence-based software testing [Duesterwald et al. 1992b; Kamkar 1993], measuring module cohesion for purpose of code restructuring [Gupta and Rao 2001], and guiding the development of performance enhancing transformations requiring estimating criticality of instructions [Zilles and Sohi 2000] and identifying instruction isomorphism [Sazeides 2003].

While precise dynamic slices can be very useful, it is also 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. Agrawal and Horgan [1990] were the first to propose two imprecise dynamic slicing algorithms, *Algorithm I* and *Algorithm II*. While such algorithms preserve the safety of slicing and have the potential of reducing the cost of slicing, they have also been found to greatly increase reported slice sizes thus diminishing their effectiveness. Another approach considered

Table I. Effectiveness of Dynamic Slicing

Program	Static	Executed	PDDS		
			AVG	MIN	MAX
008.espresso	74,039	27,333	350	2	1,304
099.go	95,459	61,350	5,382	4	8,449
130.li	31,829	10,958	206	2	834
126.gcc	585,491	170,135	6,614	2	11,860
134.perl	116,182	21,451	765	2	2,208
181.mcf	4,655	2,278	135	3	394
197.parser	49,461	3,217	72	2	217
255.vortex	253,296	74,359	826	2	4,017
256.bzip2	12,104	8,048	395	2	1,393
300.twolf	94,050	16,810	1,159	2	2,597

PDDS stands for *precise dynamic data slices*.

is to use limited dynamic information to refine static slices. In Mock et al. [2002], the authors 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”. Gupta and Soffa [1995] 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. From the above observations, we conclude that it is worthwhile to spend effort in designing efficient precise dynamic data slicing algorithms.

Next, we present some data to support the two key points made above: dynamic slices are useful in narrowing the focus of attention; and imprecise dynamic slicing algorithms have greatly diminished effectiveness. In producing this data, and throughout the article, we compute *data slices* that are obtained by taking the transitive closure over data dependences. The main motivation for using *data slices*, as opposed to *full slices* that are obtained by taking transitive closure over both data and control dependences, is as follows. Imprecision in computation of data dependences is the main source of imprecision in slices. Control dependences do not introduce significant imprecision as a vast majority of statements in a program are typically statically control dependent upon a single branch predicate. Therefore, control dependences can be accurately determined by all slicing algorithms, static as well as dynamic. It should be noted that program slices that consider both data and control dependences yield slices that represent executable programs. In other words, if S represents the slice of variable v at program point p on input I , then S can be reexecuted on I to produce the value of v at point p . In contrast, data slices are not executable.

The data in Table I shows the effectiveness of dynamic data slicing. For each of the benchmark programs, we computed 25 distinct dynamic slices at the end of program’s execution. The average (AVG), minimum (MIN) and maximum (MAX) precise dynamic data slice sizes that were observed are given in the precise dynamic data slices (PDDS) column. The slice sizes are measured in

Table II. Comparison between Precise and Imprecise Dynamic Data Slicing

Program	IDDS-II/PDDS				IDDS-I/PDDS			
	AVG	STD	MIN	MAX	AVG	STD	MIN	MAX
008.espresso	4	8.3	1	49	46	224.6	1	1,274
099.go	39	185.2	1	946	1,413	926.0	1	3,328
126.gcc	419	1,417.4	1	5,188	2,698	3,841.6	2	13,759
130.li	13	15.5	1	42	143	353.2	1	1,373
134.perl	8	19.9	1	96	41	172.0	1	974
181.mcf	1	0.31	1	2	2	2.8	1	11
197.parser	4	9.3	1	32	7	14.6	1	49
255.vortex	69	132.1	1	475	1,993	3,176.1	1	8,161
256.bzip2	35	65.7	1	229	57	91.4	1	319
300.twolf	9	20.2	1	75	27	65.8	1	242
Average	60	187.4	1	713	643	887	1.1	2,949

IDDS stands for *imprecise dynamic data slices*.

terms of intermediate representation (IR) statements. In the rest of the paper when we refer to statements we mean IR statements. 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 statements and, during the collection of the execution trace, 170135 of these statements were executed at least once. When 25 precise dynamic data slices were computed, they had average, minimum and maximum sizes of 6614, 2 and 11860 statements respectively. As we can see, the *PDDS* values are much smaller than the *Static* and *Executed* values. Thus, dynamic data slices could be very helpful in focusing the attention of the user or algorithm on a small subset of statements during debugging.

We have implemented extended versions of the two imprecise algorithms proposed by Agrawal and Horgan [1990] and compared the sizes of the imprecise dynamic data slices (*IDDS-I* and *IDDS-II*) with corresponding precise dynamic data slices (*PDDS*). The ratios *IDDS-I/PDDS* and *IDDS-II/PDDS* are given in Table II (average, standard deviation, minimum, and maximum values of the ratios are given). As we can see, imprecise slices can be many times larger than precise dynamic data slices. In the worst case, for program gcc, the imprecise dynamic data slice *IDDS-II* was over 5188 times larger than the precise dynamic data slice. The *IDDS-I* sizes are even larger. Therefore, these imprecise algorithms can be quite inaccurate.

From the above observations, we conclude that it is worthwhile to spend effort in designing efficient precise dynamic data slicing algorithms. We observe that once a program is executed and its execution trace collected, precise dynamic data slicing typically involves two tasks: [*preprocessing*] which builds a dependence graph by recovering dynamic data dependences from the program's execution trace, which is called *dynamic data dependence graph* in this article; and [*slicing*] which computes slices for given slicing requests by traversing the *dynamic data dependence graph*. We present three precise dynamic data slicing algorithms that differ in the degree of preprocessing they carry out prior to computing any dynamic data slices. The *full preprocessing (FP)* algorithm builds the entire data dependence graph before slicing. The *no preprocessing*

(*NP*) algorithm does not perform any preprocessing but rather during slicing it uses *demand-driven analysis* for recovering dynamic data 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 uses demand-driven analysis to recover the dynamic data dependencies 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 data 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 several times lower than the latency for obtaining the first slice by *Algorithm II*.

Thus, this article shows that while imprecise dynamic data slicing algorithms could be very imprecise, a carefully designed precise dynamic data slicing algorithm such as the *LP* algorithm is practical as it provides precise dynamic data slices at reasonable space and time costs.

The remainder of the article is organized as follows: Related work is discussed in Section 2. In Sections 3 and 4, we present the precise and imprecise data slicing algorithms respectively. In Section 5, we present our experimental studies. Conclusions are given in Section 6.

2. RELATED WORK

Agrawal and Horgan [1990] proposed two imprecise and two precise dynamic slicing algorithms. We have briefly shown the weaknesses of the imprecise algorithms in the introduction to motivate our work. More detailed descriptions of extended versions of these imprecise algorithms will be presented later in the article. More detailed comparisons with our new algorithms will also be given. 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 data dependence edges with instances, Agrawal and Horgan [1990] construct multiple instances of nodes and edges.

To reduce the size of the dependence graph, Agrawal and Horgan [1990] 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 Tip [1995], a program of size n can have $O(2^n)$ different dynamic slices in the worst case.

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 which the direct dependences are merely added as edge labels but no slices are computed. Moreover, since *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 develop 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*.

While, in this article, we propose the *LP* algorithm that reduces space requirements by constructing the relevant part of the dynamic dependence graph in a demand-driven fashion, we have recently developed a complementary strategy for reducing space requirements. We have developed a compressed representation of the dynamic dependence graph [Zhang and Gupta 2004]. Therefore, the dynamic dependence graphs of reasonably long program runs can be held in memory. To achieve further scalability, the use of compressed dynamic dependence graphs can be combined with demand-driven loading of these graphs into memory.

In Beszedes [2001], 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 [1994] 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 Tip [1995] that executable dynamic slices produced may be inaccurate in the presence of loops.

No experimental data is presented to evaluate the forward computation of dynamic slices in any of the above works [Agrawal and Horgan 1990; Beszedes et al. 2001; Korel and Yalamanchili 1994]. Recently, we proposed a method that allows storage of forward computed dynamic slices in a space efficient fashion [Zhang et al. 2004]. In particular, we showed that by using *reduced ordered binary decision diagrams (roBDDs)* we can store a set of dynamic slices in a space efficient manner. We also compared the performance of the *roBDD* based

forward computation algorithm with the *LP* algorithm described in this article. While the preprocessing time of the *roBDD* algorithm is higher than the *LP* algorithm, after preprocessing is complete, *roBDD* based algorithm responds quicker to slicing requests than the *LP* algorithm. However, the *LP* algorithm is scalable to longer executions than the *roBDD*-based algorithm.

If static program slicing is considered as one extreme which is very imprecise and has very low cost, dynamic slicing is the other extreme which is very precise for one execution and very expensive. There are many other works trying to find a balance between the static slicing and dynamic slicing. In Mock et al. [2002], the authors use *dynamic points-to* data to improve the accuracy of slicing, which is very similar to our extended version of *Algorithm II*. They show that the improved precision of points-to data did not improve the precision of slices in general. Gupta and Soffa [1995] proposed the *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. In Nishimatsu et al. [1999], the authors use the dynamic calling information to augment static slicing. Beszedes et al. [2002], use the union of dynamic slices from multiple executions to approximate the *realizable* slice which is smaller than the static slice but larger than the dynamic slice from any single execution. There are also some works which are variants of static slicing with dynamic flavor. *Quasi-static* [Venkatesh 1991] slicing is a technique in which the prefix of the program input is specified. In this way, the computed slice is for a group of executions, different from one execution in dynamic slicing and from all possible executions in static slicing. In *conditioned slicing* [Harmon et al. 2001], the pre- or post-conditions of the execution are specified. In *parametric slicing* [Field et al. 1995] the notion of static and dynamic slices is generalized to that of a *constrained slice*, where any subset of the inputs of a program may be supplied.

3. PRECISE DYNAMIC DATA SLICING

The basic approach to dynamic data slicing is to execute the program once and produce an execution trace that is processed to construct *dynamic data dependence graph* that in turn is traversed to compute dynamic data 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 data 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 at each point where data is referenced through pointers, we know the address at 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.

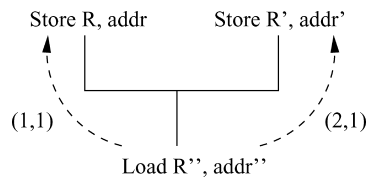
For a dynamic data slice to be computed, dynamic data dependences that are exercised during the program execution must be identified by processing the trace. The precise algorithms that we present differ in the degree to which

dependences are extracted from the trace prior to dynamic data slicing. The *full preprocessing (FP)* algorithm follows an approach that is typical of precise algorithms proposed in the literature [Korel and Laski 1988; Agrawal and Horgan 1990]. The execution trace is fully preprocessed to extract all dependences and the full dynamic data dependence graph is constructed. Given this graph, any dynamic data 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 data slicing requests. Therefore, each time the execution trace is examined, only data 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.

3.1 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 [1990]. 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 statement may be executed several times during program execution. When this happens different instances of a load statement may be dependent upon different store statements or different instances of the same store statement. For precisely recording data dependences, we associate the instances of load and store statements to the dependence edges. 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 above in which $addr$, $addr'$ and $addr''$ are addresses that may or may not be same at run time. We assume an execution in which the load statement 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 the 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 data slicing algorithm first preprocesses the execution trace and introduces labeled dependence edges in the data 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 example slices does not require these edges; only the edges for all nonpredicate uses are shown. Edges are labeled with the execution instances of statements involved in the data dependences. The precise dynamic data 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 data 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.

3.2 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 we run out of memory very often 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 data 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 data 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

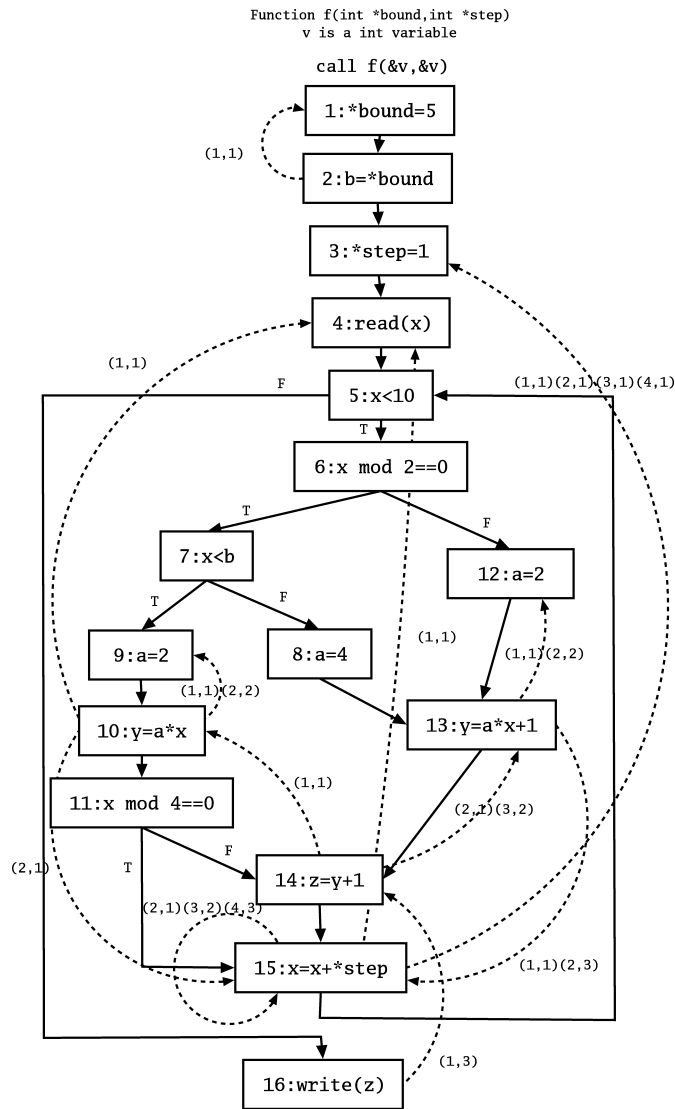


Fig. 1. The path taken by the program on input $x = 6$ is $\{1_1 2_1 3_1 4_1 5_1 6_1 7_1 9_1 10_1 11_1 14_1 15_1 5_2 6_2 12_1 13_1 14_2 15_2 5_3 6_3 7_2 9_2 10_2 11_2 15_3 5_4 6_4 12_2 13_2 14_3 15_4 5_5 16_1\}$. The precise dynamic slice for the use of z by the only execution of statement 16 is $\{16,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

the dynamic data 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 in 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 data 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 2nd instance of statement 10 is computed, it first traverses along the dependence edges $(10_2, 9_2)$, $(10_2, 15_2)$, and then all the dependences from 15_2 are already present in the graph and thus the trace is not reexamined.

3.3 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 time recovering different relevant data dependences for a different slicing request.

In light of the above discussion, we can say that the *NP* algorithm does too little preprocessing leading to high slicing costs while the *FP* algorithm does too much preprocessing leading to space problems. For example, during our experiments we found that a run of the *FP* over a trace of around one hundred million statements for 126 . gcc is expected to generate a graph of 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* by terminating blocks at function call/return boundaries or the points where the *sizes* of the blocks reach a predefined threshold. At the end

of each trace block, we store a *summary of all downward exposed definitions* of variable names and memory addresses in the order of occurrences and their *indices* to the trace block. 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, its *index* is used to seek to the corresponding point in the trace block and then the rest of the trace block is traversed. Otherwise, the whole trace block is skipped.

The time of traversing the trace mainly depends on the number of comparisons being performed. 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 as a whole, it can be partly skipped using the *indices*. So the number of comparisons does not increase in either case, which means the effect of *LP* is always positive.

The savings of *LP* can be affected by two factors:

- Definition Redundancy*, which is the ratio between the number of definitions that are not in the summary and the total number of definitions in the trace blocks. The higher the ratio is, the fewer comparisons are required in the summaries, the more savings *LP* can get when the trace blocks are partially or fully skipped. If it is 0, which means all the definitions in the trace block are downward exposed and then stored in the summary, *LP* has no benefit.
- Skipping Rate*, which is the rate of trace blocks being skipped. Higher skipping rate implies more savings.

Both of these factors are influenced by the *size* of trace block. Larger trace blocks have a high *definition redundancy* but very likely low *skipping rate* during the traversal. Smaller trace blocks have low *definition redundancy* and a good chance of being skipped. According to our experiments, function boundaries and the threshold of 200 trace entries (each entry corresponds to a single basic block) makes a good division criterion for most of the traces.

4. IMPRECISE DYNAMIC DATA SLICING

One goal of our work is to compare the precision and cost of the above precise slicing algorithms with that of imprecise dynamic data slicing algorithms. For this purpose, we also implemented the two imprecise algorithms proposed by Agrawal and Horgan [1990] (*Algorithms I* and *Algorithms II*). In this section, we explain our extended designs of these two imprecise algorithms. The extensions were necessary because Agrawal and Horgan [1990] does not discuss handling of pointers.

4.1 Algorithm I

The first algorithm proposed by Agrawal and Horgan [1990] uses a static dependence graph in which all executed nodes are marked so that during slicing

when the graph is traversed, nodes that are not marked as executed are avoided as they cannot be part of the slice.

We used an enhanced version of this algorithm for our work. As pointed out in Mock et al. [2002], the conservative nature of pointer analysis in C programs can cause severe imprecision in static program slicing when call-by-reference semantics, functions pointers and dynamically allocated heap objects are extensively used. Therefore, two enhancements are used to address this problem.

Data Dependences. Instead of using conservative pointer analysis to build the dependence graph we extract sets of addresses that are dynamically read/written by each load/store statement during program's execution. If a *store writes* to an *address* that is also *read by a load*, and the load is *reachable* from the store, then a backward data dependence edge is introduced from the load to the store. This enhancement is useful due to extensive use of pointers to heap data in C programs.

Dynamic Call Sites. We determine the *dynamic set of call sites* that are exercised for each function. When a store and a load, that respectively write and read from the same address, belong to different procedures, then the above information is used to compute *interprocedural reachability*. In the example code fragment shown below function $f()$ contains two calls to function $g()$; however, let us assume that the first call is executed and the second is not executed. Under this condition the store before the first call reaches the load in $g()$ while the store before the second call does not reach the load in $g()$. Therefore, a single data dependence $(L, S1)$ is introduced. However, if we do not consider dynamic call site information during identification of interprocedural dependences, we would in addition introduce a dependence edge $(L, S2)$. This enhancement is useful when C programs make use of function pointers.

```

void f() {
    ...
    S1: v = ...      void g(int *p) {
        g(&v);      ...
        ...         L:  ... = *p
    S2: v = ...      ...
        if (...) g(&v); }
    ...
}

```

The preprocessing step of this algorithm makes backward pass over the execution trace to construct the program's data dependence graph in which executed nodes are marked, edges are introduced between each function and subset of relevant call sites, and data dependence edges are introduced using the dynamically extracted information for loads and stores. The computation of a dynamic slice is straightforward. Starting at the statement at which a variable (address) is read, a backward pass over the dependence graph gives the dynamic slice.

4.2 Algorithm II

This algorithm is different from the first algorithm in one important respect, the manner in which data dependence edges are computed. Instead of using the dynamic sets of addresses read/written by load/store statements, the data dependences among statements that are actually exercised are determined. An edge is introduced from a load to a store if during execution, at least once, the value stored by the store is indeed read by the load. This method is clearly more precise than *Algorithm I*. This is because when dependence edges are added by *Algorithm II* from a load to store simply because they reference a common address, it is possible that such an edge represents a false dependence edge when at no time during the execution the load reads a value written by the store. This can happen due to presence of intervening stores to the same address. In *Algorithm II*, such false edges are never introduced. Another consequence of directly capturing all data dependences is that, unlike *Algorithm I* which uses dynamic call site set for each function to perform interprocedural slicing, we no longer require any additional dynamic information for interprocedural slicing. This is because dependences that are directly captured and added to the dependence graph by *Algorithm II* include both intra and interprocedural dependences.

The preprocessing step of this algorithm makes one backward pass through the execution trace to recover the exercised data dependences and adds them to the dynamic dependence graph. The computation of a dynamic slice requires a simple backward traversal over the data dependence graph.

4.3 Examples

The differences between the imprecise and precise algorithms discussed are illustrated through an example in Figures 2 and 3. Examples are given to illustrate progressively improving precision of dynamic data slices as we go from *Algorithm I* to *Algorithm II* and from *Algorithm II* to the precise algorithm (e.g., FP).

5. EXPERIMENTAL EVALUATION

5.1 Implementation

For our experimentation we used the *Trimaran* system [Trimaran 1997] 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 data slices for C programs. The key cause of imprecision in approximate data slices is the presence of pointers in C programs.

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 statements. An interpreter for the IR is available which is used to execute instrumented versions

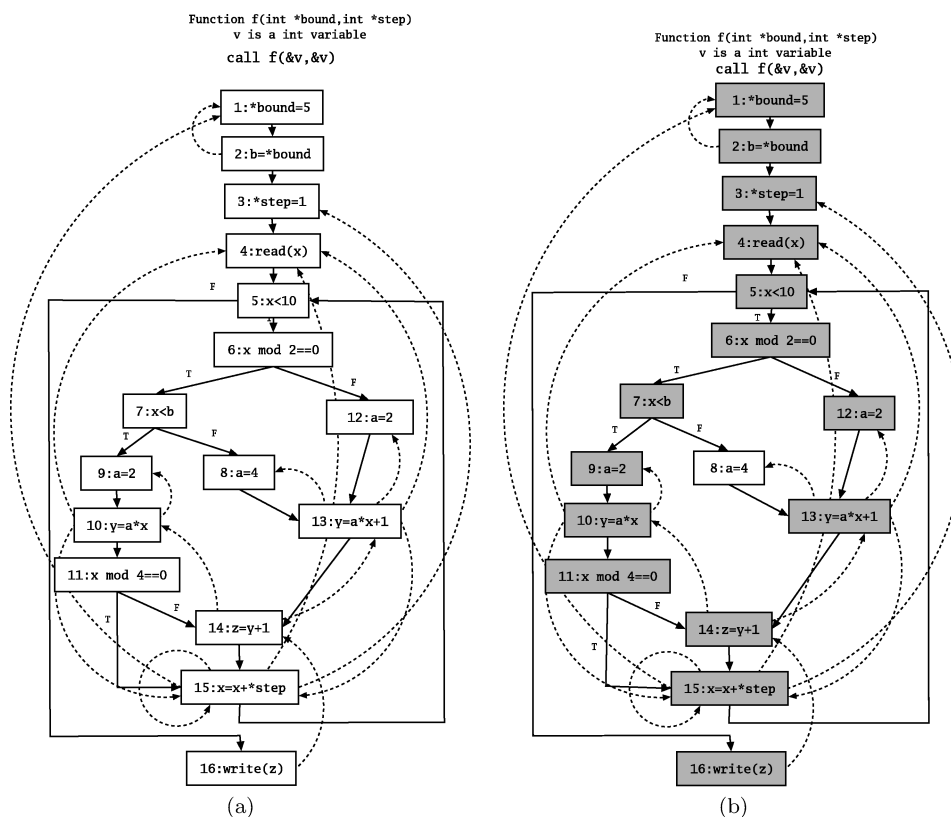


Fig. 2. For input $x = 7$, the program follows the path: $\{1_1 2_1 3_1 4_1 5_1 6_1 12_1 13_1 14_1 15_1 5_2 6_2 7_1 9_1 10_1 11_1 15_2 5_3 6_3 12_2 13_2 14_2 15_3 5_4 16_1\}$. (a) shows the memory data dependence edges introduced using dynamic set of addresses referenced by loads and stores using dotted edges; (b) shows that Algorithm I marks the nodes that are executed and during slicing only marked nodes are visited. The dynamic data slice for z at statement 16 contains $\{1, 3, 4, 9, 10, 12, 13, 14, 15, 16\}$. During backward traversal of data dependence edges the edge $(13, 8)$ is not explored by this algorithm and statement 8 is not added to the dynamic data slice because statement 8 is not marked.

of the IR for obtaining execution traces consisting of both the control flow trace (sequence of basic blocks executed) and memory trace (sequence of memory addresses referenced). 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 *longjumps* 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 data slicing algorithms, we have taken great care in implementing them. Different dynamic slicing algorithms that are implemented share code whenever possible and use the same basic libraries.

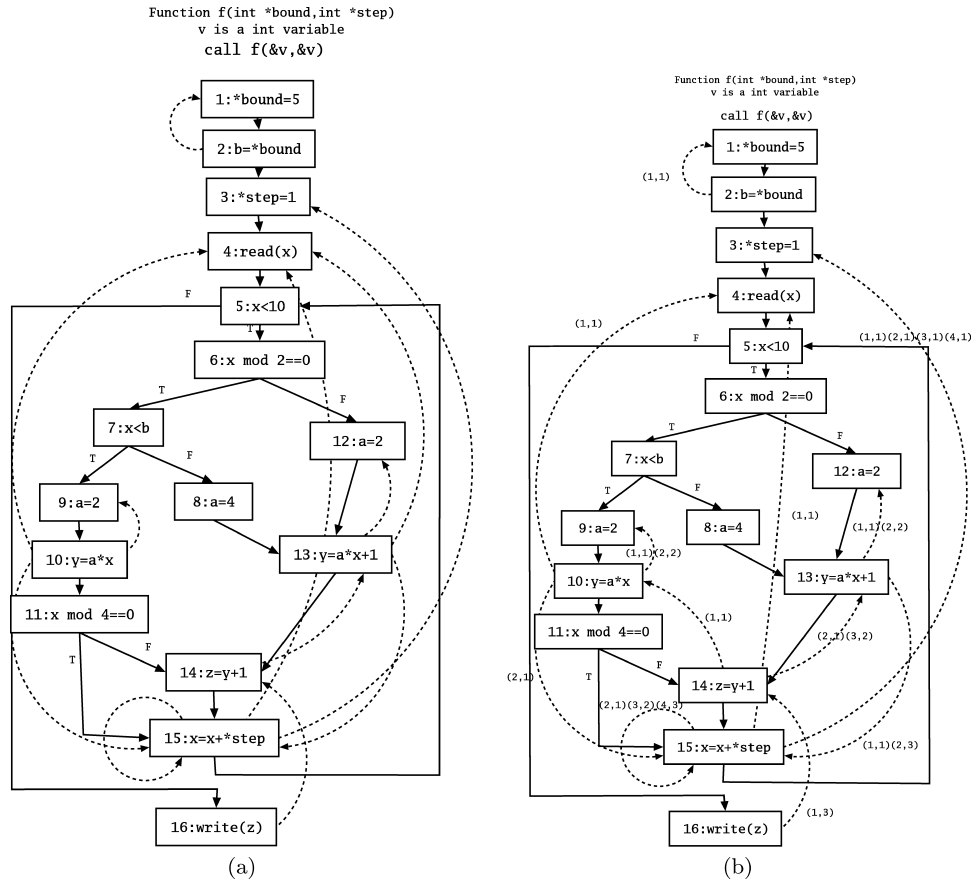


Fig. 3. (a) Again consider the execution of the program for input $x = 7$. Algorithm II captures the actual data dependence edges exercised at runtime. Therefore in comparison to Algorithm I: even though $*bound$ and $*step$ are aliases, the data dependence edge (15,1) is not added because the value of $*step$ used in 15 comes from 3; and the data dependence edge (14,10) is not added as the path from 10 to 14 is not followed during this execution. The dynamic data slice of z at statement 16 is $\{3,4,12,13,14,15,16\}$ which contains 3 less statements, 1, 9 and 10, than the slice computed by Algorithm I; (b) To show the difference of Algorithm II and precise dynamic slicing we use the input, $x = 6$. The path taken by the program 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 data dependence edge (14,10) is introduced because the path from 10 to 14 is exercised. The dynamic slice for Algorithm II is $\{3,4,9,10,12,13,14,15,16\}$ while precise algorithm produces the slice $\{3,4,12,13,14,15,16\}$ with 2 fewer statements. This is because the precise algorithm considers statement instances among which dependences exist and thus does not traverse the data dependence edge (14,10) as the value of z at 14 only depends upon the value of y at 13.

5.2 Benchmark Characteristics

The programs used in our experiments include 008.espresso from the Specint92 suite, 130.li, 134.per1, 099.go and 126.gcc from the Specint95 suite, and 181.mcf, 197.parser, 255.vortex, 256.bzip2 and 300.twolf from the Specint2000. Other Specint2000 benchmarks could not be compiled by the version of Trimaran we were using. The attributes of the programs, including

Table III. Benchmark Characteristics

Program	Lines of C Code	Num. of Funcs
008.espresso	14,850	361
099.go	29,629	372
130.li	7,741	357
126.gcc	207,483	2001
134.perl	27,044	277
181.mcf	2,412	26
197.parser	11,391	324
255.vortex	67,213	923
256.bzip2	4,650	74
300.twolf	20,474	191

Program	Executed Statement Instances			Trace Size (Bytes)		
	(1)	(2)	(3)	(1)	(2)	(3)
008.espresso	1,142,979	37,512,537	20,203,629	4,665,571	124,988,145	79,512,147
099.go	117,503,888	120,467,852	62,271,885	500,000,788	500,000,747	250,000,022
126.gcc	103,156,356	112,037,569	104,650,291	500,000,391	500,000,757	500,000,215
130.li	122,017,736	21,590,453	109,323,294	581,574,086	99,999,705	500,000,365
134.perl	32,729,474	55,416,461	111,023,934	132,807,387	250,000,192	500,000,438
181.mcf	92,544,144	130,563,056	96,323,797	499,900,970	503,974,383	385,120,404
197.parser	95,262,111	93,255,888	75,900,041	499,999,827	399,999,920	327,168,069
255.vortex	84,244,903	84,245,034	118,109,212	499,999,684	417,064,840	584,079,792
256.bzip2	95,303,451	90,512,339	57,778,596	364,989,398	348,221,994	202,512,817
300.twolf	101,547,297	159,734,018	154,502,097	499,992,542	634,520,369	621,538,455

the number of lines of C code and the number of functions are given in Table III. Each of the programs was executed on three different inputs and execution traces for the three inputs were collected. The number of statements executed and the sizes of execution traces for these program runs are also given in Table III.

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.

5.3 Precise Slicing Algorithms

In order to study the behaviors of the proposed precise dynamic data slicing algorithms, we computed the following slices. We collected execution traces on three different input sets for each benchmark. For each execution trace, we computed 25 different data slices. These data slices were performed for the latest executions of 25 distinct values loaded using load statements by the program; that is, these slices were computed with respect to the end of program's execution (@ End).

5.3.1 Slice Sizes. Let us first examine the sizes of slices. In Table I, in the introduction, the precise dynamic slice sizes of the programs on the first input were given and it was observed that the number of statements in the dynamic slice is a small fraction of the statically distinct statements that are actually executed. Thus, they are more likely to help in focusing the user or

Table IV. Precise Dynamic Slice Sizes for Additional Inputs

Program @ End	Statements Executed(2)	PDDS(2)			Statements Executed(3)	PDDS(3)		
		AVG	MIN	MAX		AVG	MIN	MAX
008.espresso	22,897	448	4	1,443	19,356	227	2	1,229
099.go	56,051	4,982	2	6,934	46,497	1,268	2	5,178
126.gcc	136,269	1,268	2	10,702	194,162	7,359	2	15,388
130.li	8,462	21	2	232	8,854	19	2	323
134.perl	15,327	98	2	611	22,897	151	2	599
181.mcf	2,171	157	2	363	2,095	136	2	292
197.parser	3,210	94	2	222	3,321	87	2	233
255.vortex	74,395	1,632	4	5,681	74,405	1,317	4	8,575
256.bzip2	8,064	479	2	1,345	7,018	408	2	1,056
300.twolf	15,184	1,098	4	2,438	19,629	1,457	2	2,681

PDDS stands for *precise dynamic data slices*.

algorithm on a small range of executed statements. In Table IV, the precise dynamic data 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 benchmarks.

5.3.2 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. Figures 4, 5, 6, 7 and 8 show the cumulative execution time in seconds as additional slices are computed one by one. The three graphs for each benchmark correspond to the three different inputs. These graphs include both the preprocessing times and slice computation times. From these figures, we note that:

- For those algorithms which perform preprocessing, the time at which the first slice is available is relatively high for them since preprocessing must be performed before the slice is computed.
- 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 V (the estimates are based upon the number of dynamic data dependences).

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. We will show the memory consumptions for these algorithms are much smaller than the *FP*'s.
- 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.
- For *NPwC* that uses caching, the cumulative execution time increases less rapidly than *NPwoC*, which does not use caching, for some programs but not

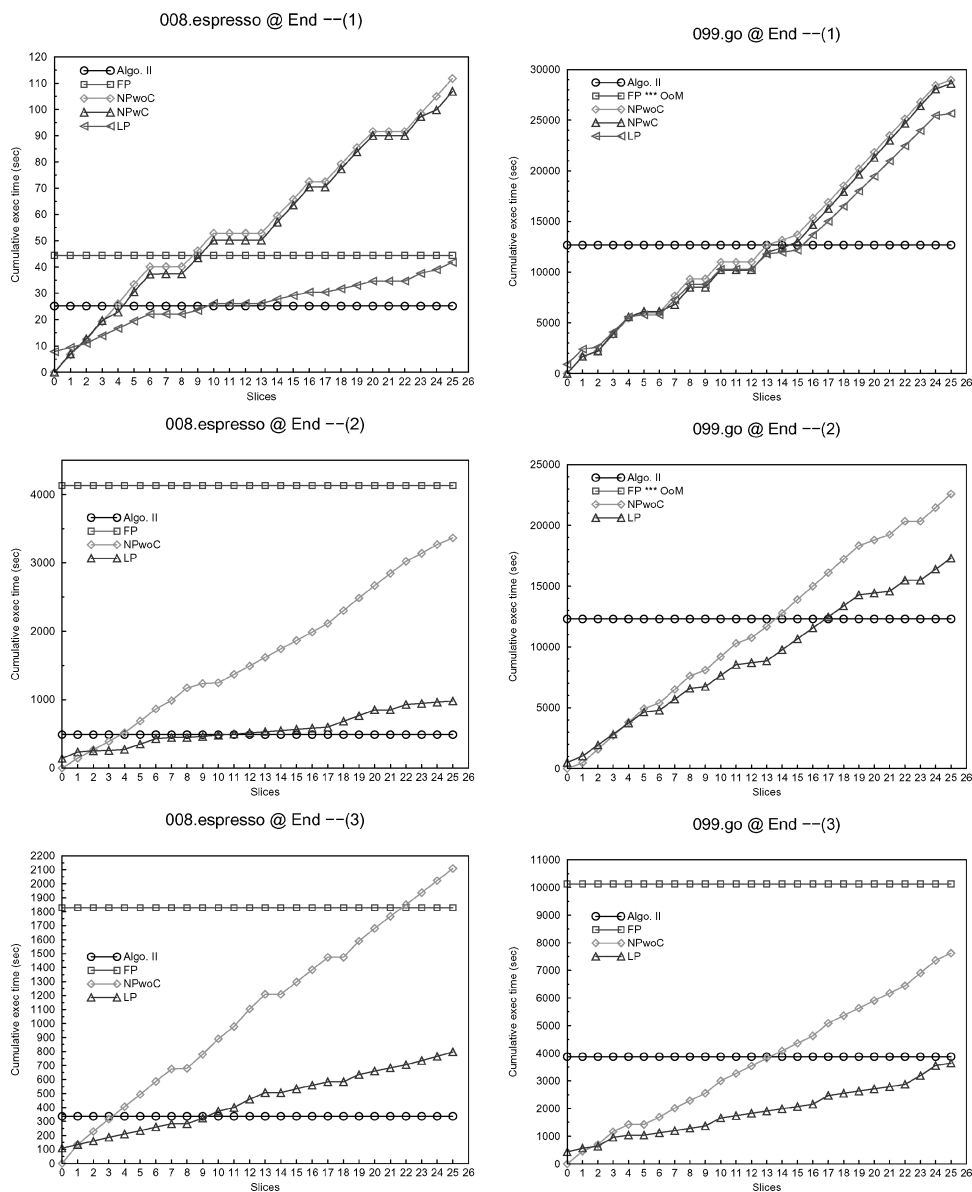


Fig. 4. Data slicing times for 008.espresso and 099.go. (OoM—out of memory, this algorithm could not be run).

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 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.

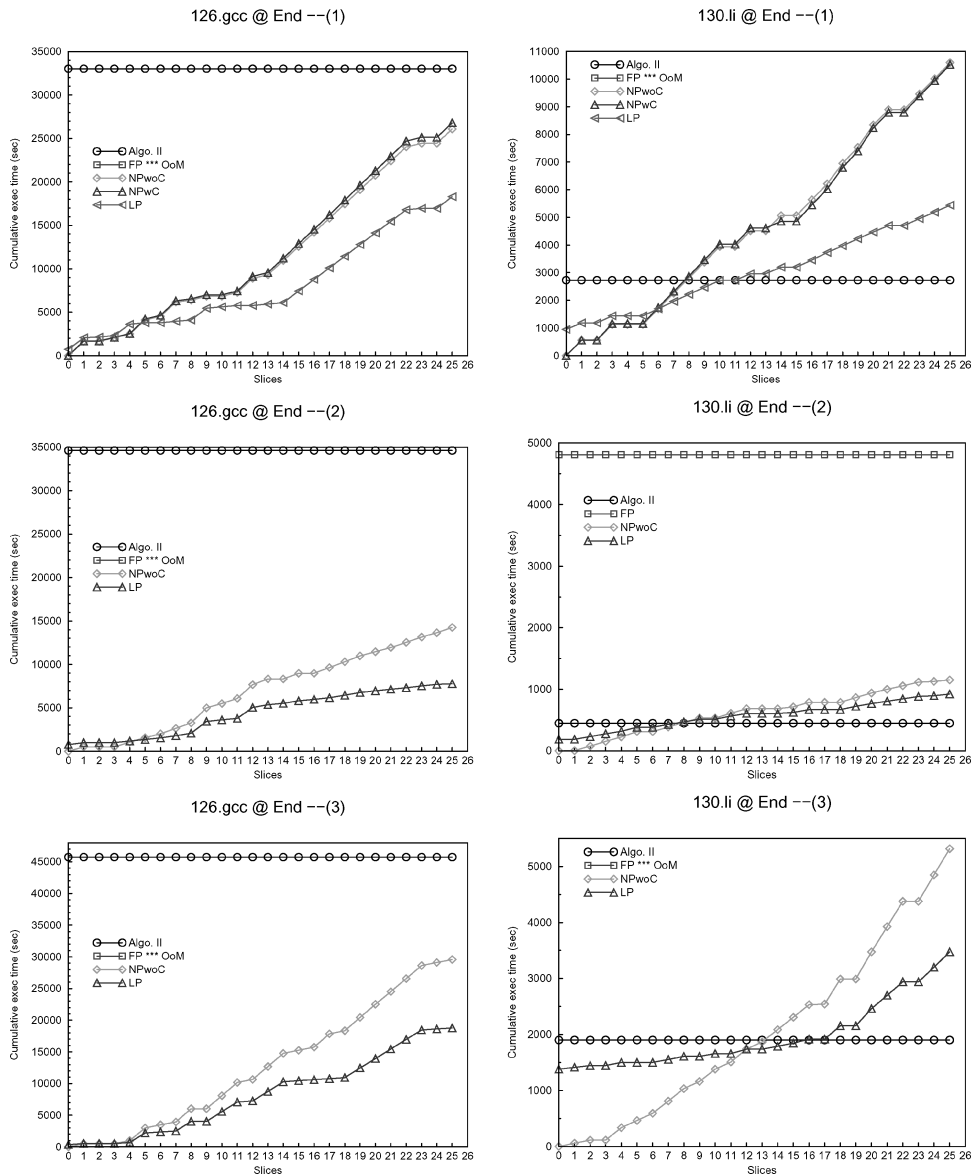


Fig. 5. Data slicing times for 126.gcc and 130.li.

—The limited preprocessing in the *LP* algorithm 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 1.09 to 3.19 times less than the *NP* algorithm for the first input set (see Table VI). This is not surprising when one considers the percentage of trace blocks that are skipped by the *LP* algorithm (see Table VII).

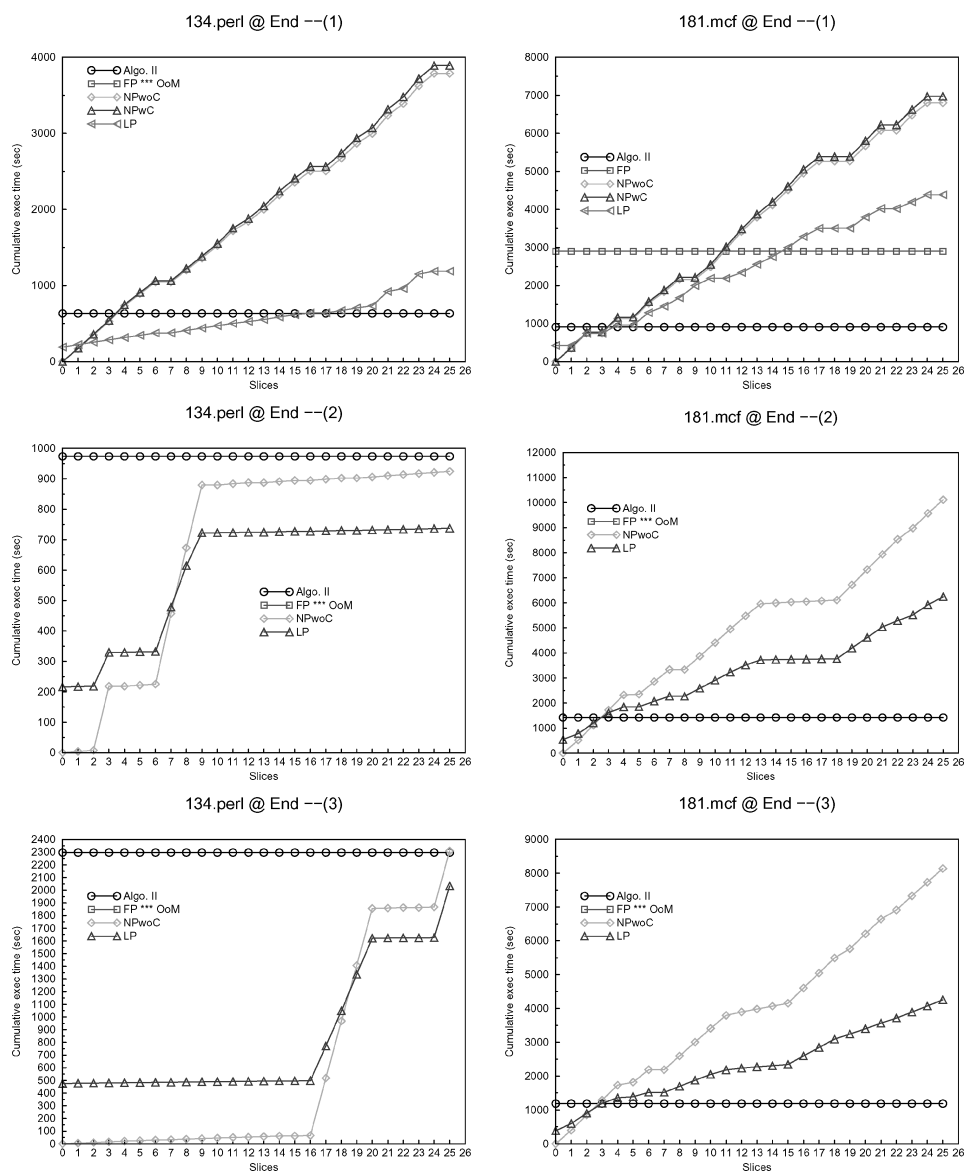


Fig. 6. Data slicing times for 134.perl and 181.mcf.

5.4 The Effect of Varying Trace Block Size in *LP*

During the discussion of the *LP* algorithm, we mentioned that changing the size of the trace block can affect the *definition redundancy* and the *skipping rate* and thus the savings obtained using the *LP* algorithm. In our implementation, we divided the trace at points corresponding to function boundaries and further we limited the maximum trace block size to 200 trace entries where each entry corresponds to a single basic block. In this section, we justify this choice by showing that for most programs this choice was a good one.

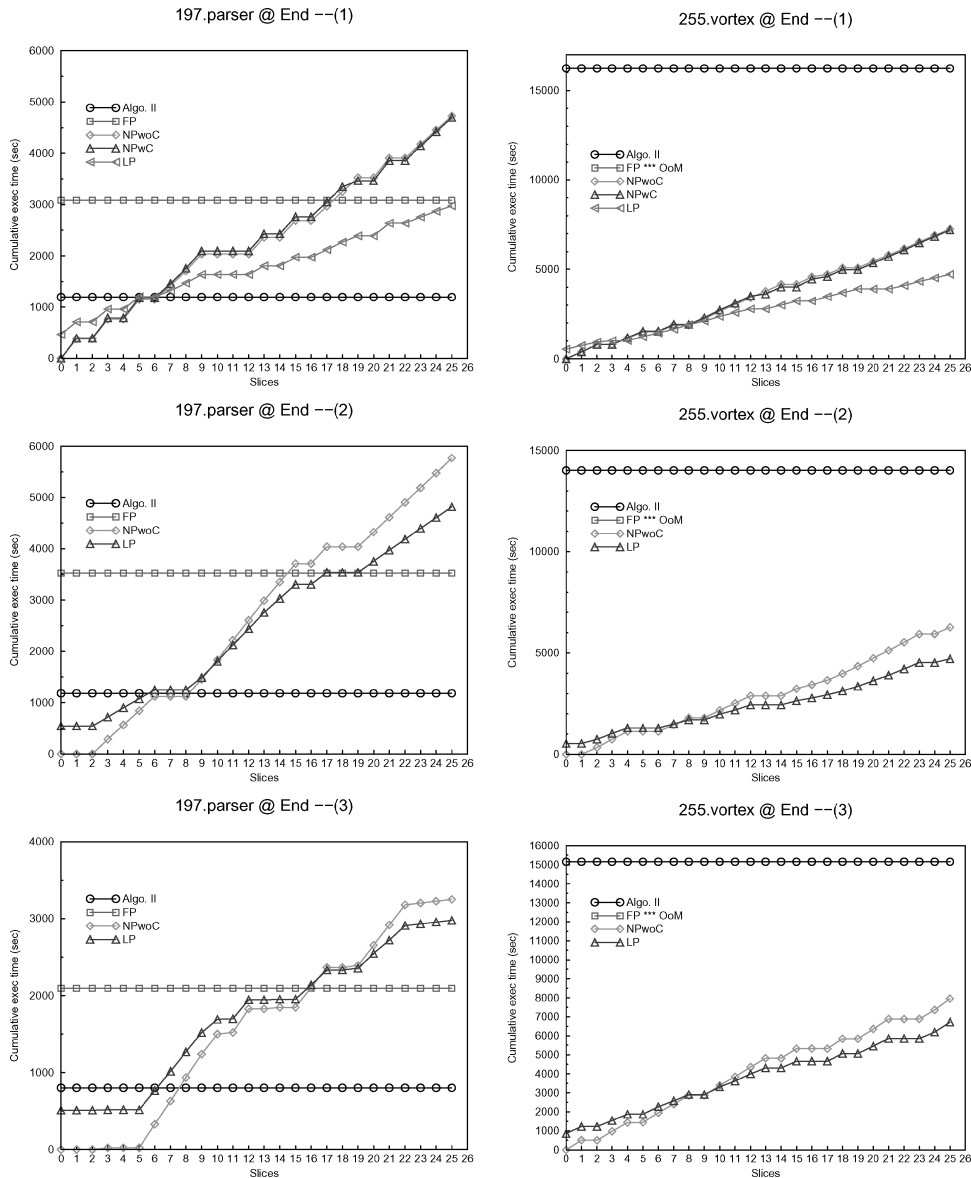


Fig. 7. Data slicing times for 197.parser and 255.vortex.

Let C be the cost of traversing the trace without the summary augmentation. When LP is used, the cost of traversing the trace is reduced by an amount determined by the two characteristics: the *skipping rate* Sr ; and *definition redundancy* Dr . This reduced cost of traversing the augmented trace can be approximated as follows. The cost of traversing the part of the trace that is not skipped can be considered to be roughly unchanged. The cost of the part of the trace that is skipped is the function of definition redundancy. Thus, the reduced

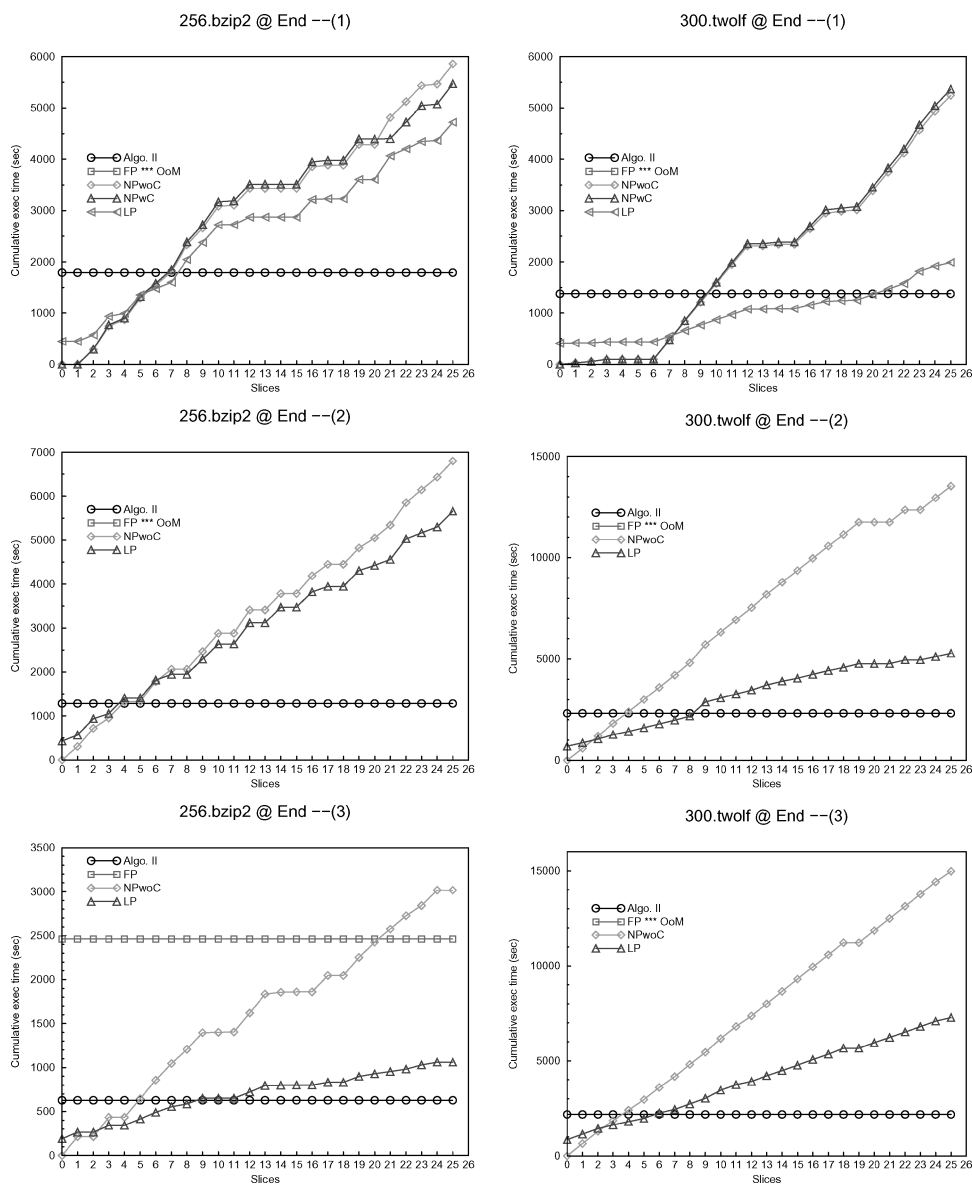


Fig. 8. Data slicing times for 256.bzip2 and 300.twolf.

cost can be approximated by:

$$\begin{aligned}
 & Cost_{not-skipped} + Cost_{skipped} \\
 &= (1 - Sr) \cdot C + Sr \cdot C \cdot (1 - Dr) \\
 &= C \cdot (1 - Sr \cdot Dr)
 \end{aligned}$$

Therefore, to minimize the cost, we need to maximize $Sr \cdot Dr$. To show our current choice of trace block size is good for most of the traces, we varied the

Table V. Estimated Full Graph Sizes

Program	Size (MB)		
	(1)	(2)	(3)
008.espresso	26.4	755.5	409.3
099.go	2,366.6	2,276.5	1,076.2
126.gcc	4,931.4	5,064.7	5,055.9
130.li	1,808.6	316.2	1,614.3
134.perl	1,975.8	5,629.2	8,977.5
181.mcf	1,571.6	2,892.1	3,470.7
197.parser	1,312.4	1,285.9	1,040.8
255.vortex	2,010.8	2,034.3	2,850.8
256.bzip2	2,169.4	2,119.8	1,001.2
300.twolf	3,746.4	6,475.5	5,188.5

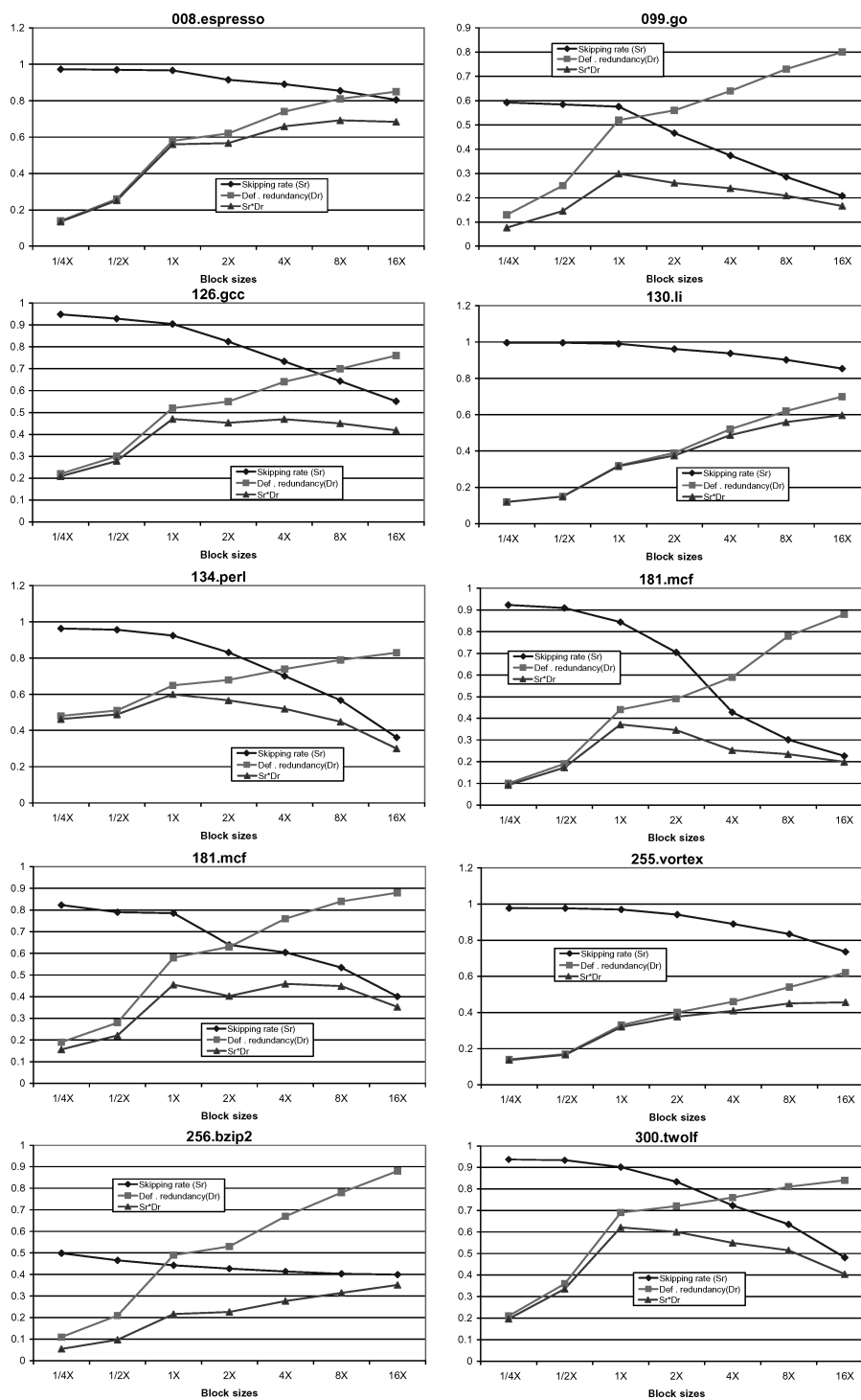
Table VI. Cumulative Times: *NP* vs *LP*

Program	<i>NP/LP</i>		
	(1)	(2)	(3)
008.espresso	2.67	3.43	2.65
099.go	1.13	1.31	2.09
126.gcc	1.43	1.833	1.58
130.li	1.95	1.25	1.53
134.perl	3.19	1.25	1.14
181.mcf	1.55	1.62	1.91
197.parser	1.59	1.20	1.09
255.vortex	1.54	1.33	1.18
256.bzip2	1.24	1.20	2.85
300.twolf	2.64	2.56	2.06
Average	1.89	1.70	1.81

Table VII. Trace Blocks Skipped by *LP*

Program	% Blocks Skipped		
	(1)	(2)	(3)
008.espresso	96.6	97.15	98.68
099.go	57.52	65.38	91.7
126.gcc	90.43	97.63	89.39
130.li	99.02	99.51	99.70
134.perl	92.42	98.99	98.26
181.mcf	84.39	86.26	90.13
197.parser	78.56	80.48	76.33
255.vortex	96.95	92.95	94.81
256.bzip2	44.24	35.7	76.95
300.twolf	90.11	88.21	78.80
Average	83.02	84.23	89.48

trace block size to see how it affects S_r , D_r , and $S_r \cdot D_r$. Figure 9 shows these results. Along the x -axis $1X$ refers to our current setting of maximum block size (200 entries) while $1/2X$ is half of the current setting, $2X$ is the double of this setting and so on. From the graphs in Figure 9 we observe the following. For many benchmarks (e.g., 300.twolf, 197.parser, 126.gcc etc.), the current setting corresponds to maximum value of $S_r \cdot D_r$. This implies that the current setting works perfectly for these traces. We can also see that the chosen trace

Fig. 9. The effect of varying block size in *LP* for input (1).

block size did not give the best savings for some benchmarks. This is to be expected as it is not possible to find a unique trace block size which is best for all programs because the trace characteristics of programs vary. However, the chosen setting performs quite well across all the benchmarks.

5.5 LP vs. Imprecise Algorithm II

In this section, we compare the performance and the memory consumption 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 I, *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*. The *LP* algorithm solves the memory 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 data 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 subsets 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 data 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 itself would require a complete traversal of the trace.

5.5.1 Memory Consumption. The dependence graph constructed is the only thing which needs to be kept in memory for slice computations. Therefore, we study the constructed graph sizes as the indicator of memory consumption for the two algorithms; that is, the partial *dynamic data dependence graph* for the *LP* algorithm and the approximate data dependence graph for *Algorithm II*. Table VIII compares the graph sizes for input(1). In the column *LP*, we take the maximum graph size constructed across the 25 slicing computations for each benchmark. In the column *Algo.-II*, we give the approximate dependence graph

Table VIII. Memory Consumption *LP* vs. *Algo.-II*

Program	<i>LP</i> (Byte)	<i>Algo.-II</i> (Byte)
008.espresso	665,092	122,540
099.go	36,525,944	258,948
130.li	11,861,832	44,732
126.gcc	10,923,816	731,008
134.perl	2,384,492	94,384
181.mcf	13,924,752	9,992
197.parser	12,687,840	12,008
255.vortex	1,367,988	314,908
256.bzip2	45,178,868	10,208
300.twolf	16,805,772	70,256

Table IX. IDDS-II vs. PDDS: Inputs (2) and (3)

Program	IDDS-II/PDDS(2)				IDDS-II/PDDS(3)			
	AVG	STD	MIN	MAX	AVG	STD	MIN	MAX
008.espresso	3.67	11.70	1	60.95	119.42	337.49	1	1,561.5
099.go	222.55	829.78	1	4,124	16.07	36.08	1	162.5
126.gcc	285.79	899.84	1	4,167.25	519.54	1,757.12	1	6,533.5
130.li	20.09	47.94	1	178.5	2.66	4.42	1	21.50
134.perl	56.56	158.13	1	737	37.13	136.43	1	703
181.mcf	1.27	0.53	1	3.47	1.18	0.27	1	2.10
197.parser	3.12	6.60	1	31.25	1.15	0.17	1	1.55
255.vortex	43.22	89.3	1	384.24	157.60	610.64	1	3,123.60
256.bzip2	29.91	68.40	1	273.57	43.27	87.78	1.40	268
300.twolf	2.90	6.97	1	34.46	4.28	8.84	1	33.11
Average	66.91	211.92	2	999.47	90.23	297.92	1.04	1,241.04

PDDS stands for *precise dynamic data slices*;
 IDDS stands for *imprecise dynamic data slices*;

size, which is fixed for a given input for each benchmark. From this table, we can observe:

- Both of these algorithms consume very small amount of memory compared to the *FP* algorithm. Thus, memory is not a problem for these algorithms.
- The partial dependence graph constructed in the *LP* is larger than the approximate dependence graph in the *Algorithm II*, which is a static graph.

5.5.2 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 IX, 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 *slice-independent* approach for comparing the algorithms by simply comparing the *dynamic data dependence graphs* constructed by them and measuring the imprecision in these data

Table X. Slice Independent Comparison of Algorithms

Program	Number of Dynamic Memory Dependences					
	Input (1)		Input (2)		Input (3)	
	Algo.-II	Precise Algo.-II	Algo.-II	Precise Algo.-II	Algo.-II	Precise Algo.-II
008.espresso	374,329	0.21	14,414,220	0.09	6,686,934	0.14
099.go	148,995,060	0.07	114,054,794	0.08	42,549,505	0.09
126.gcc	35,378,836	0.24	74,601,341	0.11	38,311,969	0.23
130.li	310,899,820	0.03	6,175,836	0.30	37,056,110	0.25
134.perl	22,114,062	0.13	6,853,213	0.68	13,566,847	0.69
181.mcf	33,673,831	0.15	41,460,615	0.12	24,614,417	0.15
197.parser	10,878,165	0.64	10,645,290	0.63	8,753,145	0.64
255.vortex	68,276,853	0.14	69,647,039	0.13	91,038,138	0.14
256.bzip2	122,559,274	0.06	106,810,886	0.07	108,751,941	0.02
300.twolf	39,193,823	0.19	58,070,415	0.23	56,123,952	0.22

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 data slices computed by this algorithm.

The number of dynamic data dependences recovered by the precise algorithm is exact. However, when the imprecise algorithm is used, the imprecision is introduced in the *dynamic data dependence graph* in form of false dependences. Therefore, if we compute the equivalent number of dynamic data dependences introduced by the imprecise algorithm, they will be higher than those for the precise algorithm. The greater the number 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 . The *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., data dependences between a store and a load operation) present in the *dynamic data dependence graphs* constructed for *Algorithm II* and *LP* algorithm. The results of this computation are given in Table X. 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.

5.5.3 Execution Times. The cumulative execution times for *Algorithm II* for slices computed at the end of execution were shown in Figures 4, 5, 6, 7 and 8. Table XI shows the preprocessing and slice computation times of these two algorithms. When we compare the execution times of the two algorithms,

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

Program	Algorithm II	LP	LP/Algo. II
008.espresso	25.15 + 0.11	7.83 + 33.98	1.66
099.go	12,671.8 + 5.37	893.92 + 24,766.14	2.02
126.gcc	33,014.1 + 11.46	727.82 + 17,556.48	0.55
130.li	2,725.32 + 3.43	858.08 + 4,493.45	1.96
134.perl	631.32 + 0.76	190.18 + 996.82	1.88
181.mcf	906.61 + 0.04	420.4 + 3,962.58	4.83
197.parser	1,195.3 + 0.04	466.98 + 2,506.86	2.48
255.vortex	16,234 + 2.48	534.56 + 4,187.60	0.29
256.bzip2	1,786.02 + 0.25	447.18 + 4,273.15	2.64
300.twolf	1,375.53 + 0.34	407.8 + 1,579.89	1.44

we observe the following:

- The total time (i.e., sum of preprocessing and slicing times) taken by *LP* is 0.29 to 4.83 times the total time taken by *Algorithm II*.
- The latency of producing the results of the first slice using *LP* is 2.15 to 21.97 times smaller than that of producing the first slice using *Algorithm II*.

On examining the graphs in Figures 4, 5, 6, 7 and 6, 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 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.

6. CONCLUSIONS

In this article, we have shown that a careful design of a dynamic slicing algorithm can greatly improve its practicality. We designed and studied three different precise dynamic data 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 data 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 2.15 to 21.97 times less than the latency for obtaining the first slice by *Algorithm II*.

In conclusion, this article shows that while imprecise dynamic slicing algorithms could be very imprecise, a carefully designed precise dynamic data

slicing algorithm such as the *LP* algorithm is practical as it provides precise dynamic data slices at reasonable space and time costs. We would like to point out that one limitation of our work is that we generated slicing requests in a synthetic manner. It is possible that slicing requests generated in context of various applications of dynamic slicing differ. Thus, a future direction of work would be to take our *LP* algorithm and carry out more extensive comparisons of precise and imprecise slices in context of specific applications.

REFERENCES

- AGRAWAL, H. AND HORGAN, J. 1990. Dynamic program slicing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 246–256.
- AGRAWAL, H., DEMILLO, R., AND SPAFFORD, E. 1993. Debugging with dynamic slicing and backtracking. *Softw. Prac. Exper.* 23, 6, 589–616.
- BESZEDES, A., FARAGO, C., SZABO, Z. M., CSIRIK, J., AND GYIMOTHY, T. 2002. Union slices for program maintenance. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, Calif., 12–21.
- BESZEDES, A., GERGELY, T., SZABO, Z. M., CSIRIK, J., AND GYIMOTHY, T. 2001. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*. 105–113.
- DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. 1992a. Distributed slicing and partial re-execution for distributed programs. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, vol. 757. Springer-Verlag, New York, 497–511.
- DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. 1992b. Rigorous data flow testing through output influences. In *Proceedings of the 2nd Irvine Software Symposium*. 131–145.
- FIELD, J., RAMALINGAM, G., AND TIP, F. 1995. Parametric program slicing. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, 379–392.
- GUPTA, N. AND RAO, P. 2001. Program execution based module cohesion measurement. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society Press, Los Alamitos, Calif., 144–153.
- GUPTA, R. AND SOFFA, M. L. 1995. Hybrid slicing: an approach for refining static slices using dynamic information. In *Proceedings of the SIGSOFT 3rd Symposium on the Foundations of Software Engineering*. ACM, New York, 29–40.
- HARMAN, M., HIERONS, R. M., DANICIC, S., HOWROYD, J., AND FOX, C. 2001. Pre/post conditioned slicing. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, Calif., 138–147.
- HOFFNER, T. 1995. Evaluation and comparison of program slicing tools. Tech. Rep., Dept. of Computer and Information Science, Linköping University, Sweden.
- KAMKAR, M. 1993. Interprocedural dynamic slicing with applications to debugging and testing. Ph.D. dissertation, Linköping University, Sweden.
- KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Inf. Proc. Lett.* 29, 3, 155–163.
- KOREL, B. AND RILLING, J. 1997. Application of dynamic slicing in program debugging. In *Proceedings of the Automated and Algorithmic Debugging*. 43–59.
- KOREL, B. AND YALAMANCHILI, S. 1994. Forward computation of dynamic program slices. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, New York, 66–79.
- MOCK, M., ATKINSON, D. C., CHAMBERS, C., AND EGGERS, S. J. 2002. Improving program slicing with dynamic points-to data. In *Proceedings of the SIGSOFT 10th Symposium on the Foundations of Software Engineering*. ACM, New York, 71–80.
- NISHIMATSU, A., JIHIRA, M., KUSUMOTO, S., INOUE, K. 1999. Call-mark slicing: an efficient and economical way of reducing slice. In *Proceedings of the 21st International Conference on Software Engineering*. ACM/IEEE, New York, 422–431.

- SAZEIDES, Y. 2003. Instruction-isomorphism in program execution. In *Proceedings of the 1st Value Prediction Workshop*.
- TIP, F. 1995. A survey of program slicing techniques. *J. Prog. Lang.* 3, 3 (Sept.), 121–189.
- VENKATESH, G. A. 1991. The semantic approach to program slicing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 26–28.
- VENKATESH, G. A. 1995. Experimental results from dynamic slicing of C programs. *ACM Trans. Prog. Lang. Syst.* 17, 2, 197–216.
- WEISER, M. 1979. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. dissertation. University of Michigan, Ann Arbor, Michigan.
- WEISER, M. 1982. Program slicing. *IEEE Trans. Softw. Eng. SE-10*, 4, 352–357.
- ZHANG, X. AND GUPTA, R. 2004. Cost effective dynamic program slicing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 94–106.
- ZHANG, X., GUPTA, R., AND ZHANG, Y. 2004. Effective forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, 502–511.
- ZILLES, C. B. AND SOHI, G. 2000. Understanding the backward slices of performance degrading instructions. In *Proceedings of the 27th International Symposium on Computer Architecture*. ACM/IEEE, New York, 172–181.
- The Tramaran Compiler Research Infrastructure*. 1997. Tutorial Notes. <http://www.trimaran.org>.

Received May 2003; revised January 2004; accepted September 2004