Dynamic Program Slicing Using Hierarchical Path Vectors

Jayakumar HG and Priti Shankar

Indian Institute of Science, Bangalore, India

Abstract. We propose a technique to improve the time and space efficiency of dynamic program slicing. The technique involves the following two novel ideas. Firstly, the precomputation of some static information stored in the form of hierarchical path vectors, which allows large portions of the trace to be skipped during the computation of a dynamic slice. Secondly a technique for storing the trace itself which is better suited for analysis and compaction. We have implemented the proposed ideas in a prototype dynamic slicing tool for programs written in C. The tool can handle programs using pointers, composite types and procedure calls. Further by computing slices at machine code level, the tool can handle (statically-linked) library calls.

1 Introduction

Program slicing [18] is a well established technique used for a wide range of applications such as program comprehension, analysis, debugging and testing. Informally speaking, given a *slicing criterion* in the form of a program statement and a set of variables of interest at that point, a static slice is a subset of statements of the program that might affect directly or indirectly the values of the variables at that point.

The notion of static slicing does not assume anything about the input and is only concerned with finding a subset of the program that *could* affect the slicing criterion for *any* input. It seems logical that a slicing technique that assumes a fixed input would be more useful for program debugging, since the slice would contain only those statements that *actually* affected the slicing criterion for the given run. This notion of program slicing, which is sensitive to the input, is called dynamic slicing. *Dynamic Slice* is defined as a subset of program statements, that *did* affect the values of variables computed at a statement instance, for a given program input. The set of variables, the statement instance and the program input constitute the *dynamic slicing criterion*, with respect to which a slice is computed.

Though the main motivation for computing dynamic slices was program debugging, dynamic slices and slices in general have found varied applications. Jhala *et al.* [7] have utilized dynamic slices in software model checkers to analyze feasibility of counter-example traces. Dynamic Slices have also been used to identify instruction isomorphism [13] with respect to their dynamic behaviour.

Reverse execution, another program debugging concept, when coupled with dynamic slicing has been show to be effective by Akgul *et al.* [2]. Further dynamic slices have also found utility in software testing [5, 8], code re-structuring [6] and program optimization [22]. Thus developing efficient and scalable dynamic slicing algorithms will improve a whole gamut of applications utilizing dynamic slices.

The main problem in the computation of dynamic slices is *scalability.* Dynamic slices are computed on actual program traces which are typically hundreds of millions of bytes in size. Both storage of traces as well as actually computing slices are challenging problems. Developing efficient and scalable dynamic slicing algorithms will improve a whole gamut of applications utilizing dynamic slices. The ability to handle library function calls necessitates slicing on object code as one cannot assume that source code for library functions is available.

The most promising approach to compute dynamic slice is the trace-based demand-driven approach. The approach involves recording the execution trace by running the instrumented program for the given input. The execution trace contains memory address referenced by each statement instance in execution order. The trace is then analyzed in a backward fashion from the point of interest, to extract only the required dynamic dependencies. However the trace size is proportional to the number of statement instances executed, which is an unbounded quantity. Hence for this approach to scale to "real-world" programs, we need to address the time requirements of analyzing the trace and the space requirements of trace itself.

In this paper, we propose the use of precomputed static information to skip over large unrelated portions of the trace. We use a modification of a path vector data structure originally proposed by Bharadwaj *et al.* [3] for instruction scheduling. Further we present a trace representation scheme, taking advantage of locality of reference, to reduce space requirements.

The approach is similar to that of LP [20] in the sense that limited precomputation is done to speed up the slicing computation. However our precomputation is performed on the program, and not on the trace and can therefore be re-used across different runs of the program. Further we present a trace representation scheme which is compact and suitable for compression, and rapidly traversable.

Briefly, the scheme works as follows. The code is first analyzed to detect potential dependencies across a hierarchy of regions in the control flow graph (CFG). A path-vector based representation [14] is used to store the static dependencies within acyclic regions. The program is then instrumented so as to record the trace in the required representation. The instrumented program is run on the given input and the trace is collected. To compute a slice, the trace is traversed backwards from the point of interest. Instead of traversing the entire trace to find reaching definitions, we only visit "regions" of the trace that are not ruled out by the static pre-computed information.

The rest of the paper is organized as follows. Section 2 describes our static pre-computation scheme. The hierarchical trace representation is described in

Section 3. Section 4 describes the interprocedural slicing algorithm using the precomputed information and the new trace representation. Section 5 details our implementation of the proposed ideas. Section 6 describes some experiments carried out by us on standard SPEC benchmark programs. Section 7 briefly reviews related work and finally Section 8 presents our conclusions and suggestions for future work.

2 Static Pre-Computation

A structure-based data flow analysis is performed to collect the desired static def-use information. The control flow graph (CFG) is initially partitioned into a hierarchy of disjoint acyclic regions. The innermost un-processed regions are analyzed first. Path-Vectors are used to represent the def-use information along various paths in a acyclic regions. Once a region has been analyzed, it is replaced by a summary block in the CFG. Thus the data flow analysis proceeds in a hierarchical fashion.

Hierarchical Acyclic Regions A *region* is an induced sub-graph of a graph with a special node marked as header, which dominates every other nodes in the region. A simple definition of acyclic regions in a CFG would be

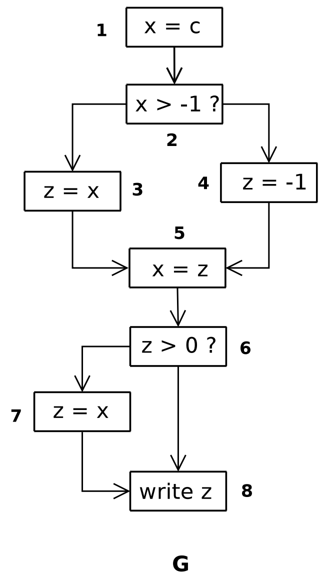
- A basic block is a region (basic region).
- A *natural loop* with-out the corresponding back-edge is a region (summary region).

However it is not required that only *natural loops* correspond to summary regions. We view acyclic procedures (which are transformed to single-entry) as procedure summary regions. Further we divide summary regions into smaller summary regions so that the number of paths is bounded by some constant.

To partition a procedure's CFG into a hierarchy of acyclic-regions, we first find all the back edges and corresponding natural loops in a CFG. After finding the loops within a CFG, we compute the nesting relation among the loops. The nesting relation among the loops provides the hierarchy among acyclic summary regions. The inner-most loops of the CFG are first processed. For each basic block in these loops, we create a basic region corresponding to it. The summary regions, corresponding to these loops without back-edges, are analyzed to determine static dependencies with-in the region. Once the inner-most loops are processed, they are replaced by corresponding summary blocks in the CFG. These blocks summarizes information present in corresponding regions into a single block. With this reduction of the CFG, there is a new set of inner-most loops. The above process is continued, until the CFG is reduced into a single block.

Only reducible flow graphs can be partitioned into regions according to the above scheme. CFG with non-reducible loops (loop header does not dominate all nodes in the loop) requires an initial transformation. For every non-reducible loop, the set of nodes not dominated by the loop header is split so that that a reducible loop results.

Path Vectors A *path vector* is defined as a bit vector in which each bit position maps to a unique path in an acyclic graph. Since number of distinct paths through any acyclic graph is finite, one can use path vectors to represent set of paths which satisfy a property of interest. For example set of control flow paths that flow through a block n is encoded in a block path vector $BPV(n)$. Similarly set of control flow paths along which a definition of variable v at w reaches use of variable v at r is encoded in def-use path vector $DPUV(w, r, v)$. These vectors were first defined in [3] for use in instruction scheduling. Figure 1 shows BPVs and DUPVs for a sample CFG.



Paths in G

P0 = {1,2,3,5,6,7,8}
P1 = {1,2,3,5,6,8}
P2 = {1,2,4,5,6,7,8}
P3 = {1,2,4,5,6,8}

Block Path Vectors

	P0	P1	P2	P3
BPV(1) =	1	1	1	1
BPV(2) =	1	1	1	1
BPV(3) =	1	1	0	0
BPV(4) =	0	0	1	1
BPV(5) =	1	1	1	1
BPV(6) =	1	1	1	1
BPV(7) =	1	0	1	0
BPV(8) =	1	1	1	1

DU Path Vectors for X

	P0	P1	P2	P3
DUPV(1,2) =	1	1	1	1
DUPV(1,3) =	1	1	0	0
DUPV(1,7) =	0	0	0	0
DUPV(5,2) =	0	0	0	0
DUPV(5,3) =	0	0	0	0
DUPV(5,7) =	1	0	1	0

Fig. 1. Example of Path Vectors

The use of path vectors for dependence representation was introduced by Bharadwaj *et al.* [3] for scheduling acyclic regions of CFG. Raghavan *et al.* [12] and Tahir *et al.* [14] have explored the use of path vectors for regression testing. The key reason we chose to utilize path vectors to represent def-use information is that dynamic slicing requires def-use information along a path rather than along all paths.

BPVs and DUPVs are used to represent the static control and data dependencies in each acyclic regions of the CFG. Algorithm 1 enumerates all the paths through an acyclic CFG and computes the block path vectors for each block in the CFG. First number of paths starting at each node is counted. Then the total number of paths is obtained from the number of paths starting from the entry node. Each path is then assigned an index in the path vector and BPVs for each node is computed. The function $PVProject$ essentially allocates blocks of path indexes of paths, through a node, on to the node's successors. It projects into PV, $NP(n)$ contiguous 1s in $BPV(n)$ into k 0's followed by $NP(s)$ 1's and rest 0's.

Algorithm 1 Compute Block Path Vectors

```
procedure ComputeBPV($G(V,E)$,$entrynode$)
    for each node $n$ in reverse topological order do
        $NP(n) \leftarrow 0$
        for each successor s of n do
            $NP(n) \leftarrow NP(n) + NP(s)$
        end for
        if $n$ is an exit node then
            $NP(n) \leftarrow NP(n) + 1$
        end if
    end for
    $npaths \leftarrow NP(entrynode)$
    for each node $n$ in $V$ do
        $BPV(n) \leftarrow$ zero-vector of size $npaths$
    end for
    $BPV(entrynode) \leftarrow$ bit-vector with all bits set
    for each node n in topological order do
        $k \leftarrow 0$
        for each successor s of n, in reverse topologicall order do
            $PV \leftarrow PVProject(BPV(n), k, NP(n), NP(s), count)$
            $BPV(s) \leftarrow BPV(s) \cup PV$
            $k \leftarrow k + NP(s)$
        end for
    end for
end procedure
```

Once we compute the BPV for each node in the graph, we need to compute def-use path vectors $DUPV(w,r,v)$ for all $r \in Use(v)$, $w \in Def(v)$, $v \in Vars$. Algorithm for computing $DUPV$ is pretty straight forward and the reader is refered to paper by Bharadwaj [3]. The main idea of the algorithm is finding the set of paths along which the definition of v in w reaches use in r, by intersecting the BPV's of the w and r. Since we are performing slicing at machine code level, the static def-use information is also in machine code level details. Registers, direct memory references are treated as named variables in the procedure. We also treat memory references through frame pointer as named variables, since frame pointer is altered typically only in procedure's prologue and epilogue. We assume that frame pointer is constant elsewhere and hence these memory references corresponds to an constant offset from the frame pointer. Using this scheme, we improve the accuracy of static dependence information computed. Other memory references like indirect memory references are treated as pointer de-references. The side effects of pointer de-references and procedure calls are handled in a conservative manner. Any pointer de-reference is considered to potentially defining or using any variable. Also worst-case assumptions are made about the side-effects of procedure calls. Any variable that can be written-to in the called procedure is considered to be potentially defined during the call.

Region Summarising For each summary region in the CFG, we determine the def-use information and represent it in path-vectors. We do not compute dependencies with in basic blocks, as we believe that gains of using these during slicing will be insignificant. Once the summary region is processed it is replaced by a summary block which represents the entire region as a single block. The summary block contains the downward-exposed definitions and upward-exposed uses of the region as the def-use information of the block. For a summary region R that actually corresponds to a loop, to take into account the effect of the back-edge, we infer DUPVs of type $DUPV(R, R, v)$. For each variable that are in downward-exposed definitions and upward-exposed uses we define $DUPV(R, R, v)$ to be set of paths along which v is used. Once the summary block replaces the region in the CFG, the block is effectively like any other basic block with downward-exposed definitions and upward-exposed uses.

3 Trace Representation

The trace, both control and data together, is a program execution representation. Using the trace we can reconstruct the dynamic dependencies exercised during the program run. The trace size is usually too large to be held in main memory, especially for long program runs and hence is typically stored on disk. We have designed a trace representation scheme which ensures that unrelated portions of the trace are never brought into main memory from disk. Further the representation allows for employing compression schemes so that space required for storing the trace itself is reduced.

In our representation, the data trace is a sequence of memory references in execution order. Only memory references that cannot be determined statically are recorded, since during analysis one can always look-up the static address references from the code. As stated earlier, to handle library calls uniformly we trace at machine-code level. i.e., we record the dynamic memory references of each instruction. Typical procedure oriented language compilers convert local variable references into memory offsets from stack frame register. The stack frame register itself changes only at beginning and end of functions. We exploit this pattern to reduce the number of instructions traced. Only the stack frame register contents is recorded during the start of the function and since all the offsets are constants, we can skip recording local stack references.

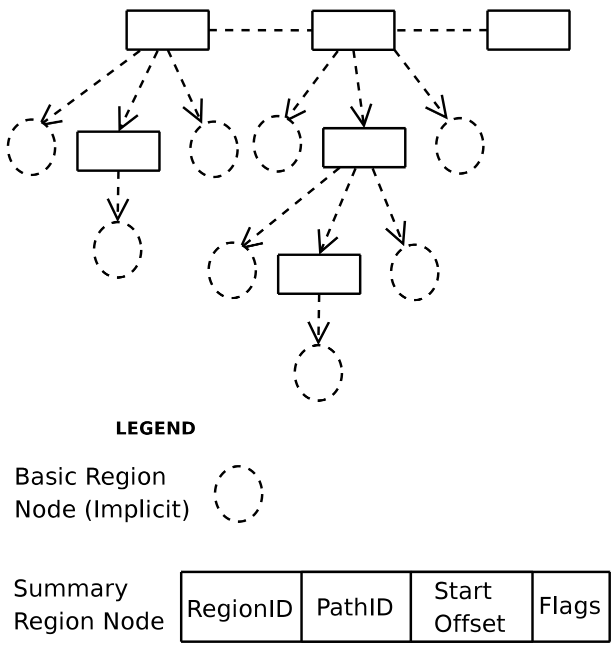


Fig. 2. Trace Representation Scheme

To facilitate skipping of unrelated portions of the trace, we maintain an auxiliary hierarchical trace index. Figure 2 illustrates the proposed trace index representation scheme. This index comprises of nodes which demarcates the data trace corresponding to regions. For each region instance executed, we have a corresponding node instance. The nodes are arranged in a hierarchical fashion. A node N corresponding to a region instance R has a parent node F (if any) which correspond to the region instance Q (which encloses R) whose execution involved execution of R. A child node of N, say C (if any), corresponds to the region instance S (which is enclosed by R) that was executed when R was executed. The *start* field in each node references the offset in data trace where trace of the corresponding the region starts.

start facilitates skipping over an entire unrelated summary region trace. The trace index also serves to maintain the control flow trace, since it records the *pathID* for each summary region. The $flags$ field helps to record the type of summary region (loop or function) and also if the data trace is in compressed form or not. In addition to above depicted fields, node corresponding to functions have an extra field $frame$ which contains the frame register contents for the function.

$pathID$ is an unique number given to a path during path enumeration for computing BPVs. It serves to identify the particular path taken in a summary region during execution. In figure 2 nodes corresponding to basic region instances are represented implicitly in the trace index. This is possible, since we have the $pathID$ for the summary region instance containing a give basic region instance as a child node. The $pathID$ identifies all the basic regions that were part of the path taken during the summary region instance execution. Thus we store only nodes that correspond to summary regions with in the index, resulting in major space savings. To compute the $pathID$ during execution of a summary region, we maintain a *pathvector* P, initially a *zerovector*. As and when we execute a sub-region with in the summary region, we intersect the BPV of the sub-region with the P. Once we come out of the summary region, the position $pathID$ will be the first bit set in P.

Both data trace and its index typically show high amounts of redundancy and hence are very suitable for compression. Data traces show locality of reference which can be exploited by compression algorithms. Since selective trace decompression is necessary, we considered compressing the data trace in parts, especially traces corresponding to summary regions. For summary regions which encloses other summary regions, to analyze the enclosed summary region trace one would need to de-compress the entire trace portion of the enclosing summary region. The scenario for index compression is better. The index size is typically orders lesser than that of data trace. The hierarchical index can be flattened using the last-child, older-sibling list representation. Further the fields of each node in the index are viewed as separate columns that can be individually compressed. The $SEQUITUR$ compression can be applied to these columns so that the index can be traversed (backwards) without entirely decompressing. This works out in case of index since we can afford to scan the entire index, which is orders smaller that the data trace.

4 Dynamic Slicing Algorithm

In this section we will present the outline of the inter-procedural) dynamic slicing algorithm used in our propose approach. The algorithm takes slicing criterion (T, I_k), where T is the trace, I_k is the last k^{th} instance of statement I, as input. and produces a set S, consisting of set of statements that actually affected (through data dependencies) the variables in the slicing criterion.

A brief summary of the algorithm is as follows: First the trace is rewound to point of interest - last k^{th} instance of statement I. Then the data trace is traversed backwards hierarchically utilizing the index. The traversal and slicing procedure is described in Algorithm 2. Through-out the algorithm we maintain a set of variables, X, for which we have to find reaching definitions. For each variable v in X, we also maintain a list of potential regions that could contain definition of v. That is, a list of regions from which a definition of v can reach the

Algorithm 2 Inter-Procedural Slicing Algorithm

```
1: procedure ComputeDS($T$,$I_k$)
2:     $S \leftarrow \phi$                                        ▷ $S$ - set of statement affecting $I_k$
3:     $TracePositionInitToEnd(T)$
4:     $currR \leftarrow prevRegion(T, currR)$
5:     $RewindToBasicRegion(BasicBlock(I), T, k)$
6:     $RewindInBasicRegion(currR, T, I)$                         ▷ rewind $T$ until we get to $I_k$
7:     $traversed \leftarrow false$
8:     $(DEF, USE) \leftarrow CurrentTraceData()$
9:     $X \leftarrow \{(v, currR) \mid v \in USE\}$              ▷ X - set of pairs $(var, list < region >)$
10:    while $X$ not empty do
11:        $Y \leftarrow \{v|(v, L) \in X, currR \in L, DPV[v][currR][pathID] \neq 0\}$
12:        if $Y \neq \phi$ then
13:            if $traversed$ is $false$ then
14:                if $LastChildRegionExists(currR))$ then
15:                    $currR \leftarrow LastChildRegion(currR)$
16:                else
17:                    $traversed = true$
18:                end if
19:            end if
20:            if $currR$ is a basic block then
21:                $SliceInBasicBlock(currR, T, X, S)$
22:                $traversed \leftarrow true$
23:            else if $traversed$ is $false$ then
24:                $currR \leftarrow LastChildRegion(currR)$
25:            end if
26:        else
27:            $SkipRegion(currR)$                                ▷ rewind trace to skip $currR$
28:            $traversed \leftarrow true$
29:        end if
30:        if $traversed$ is $true$ then
31:            $X \leftarrow X \cup \{(v, pR) \mid v \in U_R, DUPV[pR][currR][pathID] \neq 0\}$
32:                                                               ▷ $U_R$ - variables defined in currR
33:            if $PrevSiblingRegionExists(currR)$ then
34:                $currR \leftarrow PrevSiblingRegion(currR)$
35:                $traversed \leftarrow false$
36:            else
37:                $currR \leftarrow ParentRegion(currR)$
38:                $traversed \leftarrow true$
39:            end if
40:        end if
41:    end while
42: end procedure
```

use along the path indicated by $pathID$. This information is obtained from the $DUPV$ triples computed during static pre-computation. We skip trace portions corresponding to regions if they are not present in the list of potential regions to scan. Other regions are scanned, in attempt to find definition(s) of variables in X. On finding the desired definition, we add the statement to the slice and add the variables used in that instance to the set X. The process continue until either trace is completely traversed or the set X is empty. The set S computed is the required dynamic slice for criterion (T, I_k).

In Algorithms 2, $currR$ represents the current region instance whose trace is being scanned. $LastChildRegion$, $PrevSiblingRegion$ and $ParentRegion$ are functions used to traverse the trace index represented as last-child, older-sibling representation. To handle procedure calls we maintain a $returnstack$. This is conceptually opposite of the call stack used during program execution. Since we are traversing the trace in reverse, we encounter data trace of the return before the call. So to take this into account, whenever a $currR$ is updated to refer to a basic block with a call at end, we push the current traversal state into the $returnstack$. Then we update the $currR$ to refer to the return block of the function being called. Once the trace corresponding to the function is scanned, we pop out the traversal state from $returnstack$ and continue with traversal of calling function. This functionality is implemented in $LastChildRegion$, $PrevSiblingRegion$ and $ParentRegion$.

To reduce the number of regions that need to be scanned we propose to use another path vector called DPV. The motivation for using such a path vector is that if DUPV information indicates that a summary region could potentially contain a desired definition, we need to scan the entire trace of that region. But the trace of that region may be huge, probably containing trace of large loops, function calls. It may finally turn out that only a single block in this contained the desired definition. So to filter out a lot more regions from being scanned, we use DPV. $DPV(v, w)$ is defined as set of paths along which a definition of variable v in w reaches the region exit. This essentially a slightly more refined than downward-exposed definition information computed for each region during pre-computation phase. So before scanning a region, we check if the particular path it has executed has any definition we may be interested in. If not, we simply skip scanning the region.

The algorithm 2 demonstrates computation of only data slices. We can also compute full slices by considering dynamic control dependencies along with data dependencies. BPV information computed during pre-computation is used to determine control dependencies.

5 Implementation Details

We have implemented the proposed ideas in a prototype dynamic slicing tool for C language on the x86 platform. The tool requires the binary program to be compiled from a patched *gcc* tool-chain and linked statically. The tool is be able to handle pointers, composite types, procedures and library calls. To handle library calls uniformly, we deal with machine code instead of source code. Figure 3 provides a schematic of the components used in the slicing tool. Shaded blocks corresponds to components that are implemented by us, whereas unshaded blocks correspond to existing components that we utilized.

The static pre-computation component (*Analyzer*) analyzes machine code of the program and computes the static dependence information. We are using *Diablo* [4] to extract the Control Flow Graph (CFG) of each procedure from the machine code. .

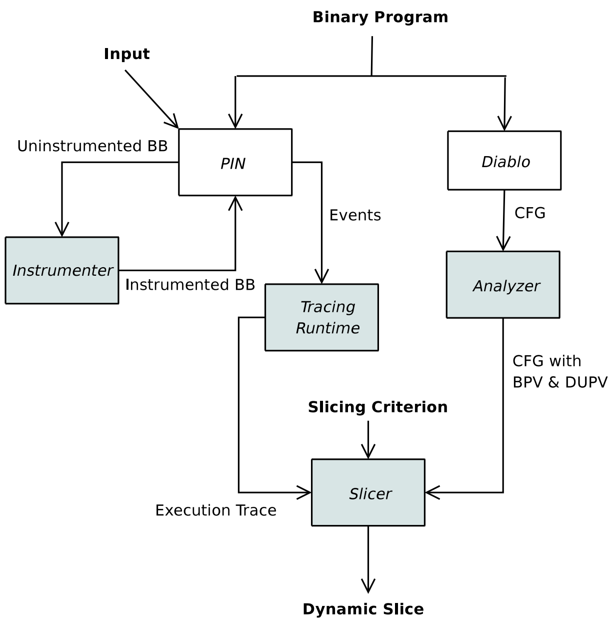


Fig. 3. Dynamic Slicer Framework Schematic

To record the trace, we are using *PIN* [11] as the program instrumentation tool. We have developed a *PIN* tool, called *Tracer*, which consists of the *Instrumenter* and *Tracing Run-time* components shown in Figure 3. *Tracer* adds instrumentation code after each instruction to record (any) dynamic memory reference in the data trace. It also adds instrumentation code before each summary region entry and exit to collect the hierarchical trace index.

Once the instrumented program completes execution for given input, we have the execution trace. The *Slicer* accepts slicing requests on the collected trace and computes the required slices. The slice is obtained as a set of machine-code instructions which can be mapped back to source level statements using the debugging information produced by the compiler.

6 Experimental Results

Benchmark Program	Number of Instructions	Number of Basic Blocks	Number of Regions
164.gzip	88658	22729	2352
175.vpr	117177	27331	2829
176.gcc	442552	110364	11605
186.crafty	128192	30223	2873
197.parser	105575	27182	2922
glibc	80430	20765	2088

Table 1. Test Program Characteristics

We have evaluated the performance of our implementation on a few SPEC CPU2000 benchmark programs. We also implemented the naive trace-based approach NP, with which we make comparisons. The benchmark program characteristics are provided in Table 1. These programs are statically linked and hence also accounts for all instruction with in libraries like libc. The contribution of glibc is presented in the last row of Table 1. All experiments were performed on a machine equipped with Opteron 265 and 2.7 GB of RAM, running GNU/Linux 2.6.24.

In the first experiment, we have evaluated the performance of our analyzer tool, which computes static def-use information in path vectors. Figure 4 shows the breakup of time spent by the analyzer in extracting the CFG and analyzing them. One needs to note that this is the time taken to analyze the entire program with all its libraries for the first time. Suppose the program is re-compiled, we analyze only updated object files without re-computing everything from scratch. Thus static information can be pre-computed once and updated incrementally.

In the second set of experiments, we evaluate the uncompressed traces sizes and how they compare with NP traces. The benchmark programs where run with test inputs and trace collected for 100 million instruction executions in each case. In Table 2, we have tabulated the data trace sizes for NP and our approach. There is around 50-75% space savings by obtained by considering frame pointer constant with in function body. So recording the value of frame pointer during function prologue rather than recording all memory references through a frame pointer turns out be effective.

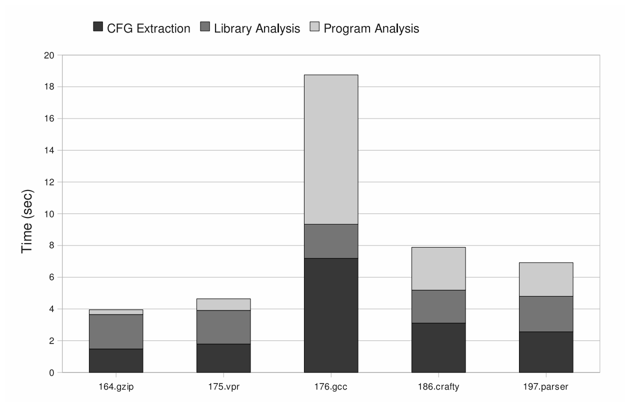


Fig. 4. Static Pre-Computation Time

Benchmark Program	NP Trace Size (Bytes)	Our Trace Size (Bytes)	% Space Savings
164.gzip	618815488	146572200	76.31
175.vpr	513835008	248644136	51.61
176.gcc	600842240	216155264	64.02
186.crafty	587964416	158721640	73.00
197.parser	572047360	271941024	52.46

Table 2. Data Trace Sizes

Benchmark Program	NP Trace Size (Bytes)	Our Trace Size (Bytes)	% Space Savings
164.gzip	399998976	30228480	92.44
175.vpr	399998976	53837824	86.54
176.gcc	399998976	49201152	87.70
186.crafty	399998976	31932416	92.02
197.parser	399998976	43106304	89.22

Table 3. Control Trace Sizes

Benchmark Program	NP Slicing Time (s)	Our Slicing Time (s) + Tracing Overhead (s)	% Time Savings
164.gzip	195.66	130.68 + 10.06	28.07
175.vpr	201.90	121.37 + 5.68	47.70
176.gcc	521.76	393.22 + 9.78	22.76
186.crafty	216.20	107.39 + 14.62	32.20
197.parser	188.07	125.22 + 8.49	28.87

Table 4. Slicing Times

In Table 3, control trace sizes for NP and our approach are compared. As stated earlier, we use a combination of $pathID$ and summary $regionID$ to record the control trace. We have the control trace size for NP same throughout since we are execution 100 million instructions in each case. Thus using the hierarchical trace we around 85-95% space savings.

In the final set of experiments, we have evaluated our slicing approach. Table 4 gives a comparison of the slicing times for NP and our approach. The programs where run for 10 million instructions and slicing was done with the last write instruction executed as the slicing criterion. The tracing overhead indicates the amount of extra-time our instrumented programs take to collect hierarchical trace. We can see that our approach is around 25-45% faster than NP approach. This speed-up is a combination of reduced data and control trace size and skipping of unrelated regions.

7 Related Work

This section provides a brief survey of various dynamic slicing techniques that can be found in the present literature. Agarwal *et al.* [1] where among the first to present a technique for dynamic slicing. They introduced the DDG-based approach, where dynamic slice computation is reduced to a graph reachability problem. The space complexity of DDG, is proportional to (at worst) square of size of execution history. Zhang *et al* [20] have proposed slicing algorithm called OPT which utilizes a compact representation of DDG. The compacted DDG is a multi-graph, where a statement is represented as a single node and the dependencies between statement instances are represented by labelled edges. They demonstrate a significant portion of the edges can share a single representative edge, with suitable transformations on the DDG.

Mund *et al.* [10] have presented a dynamic slicing approach which forward computes dynamic slices for every variable during the execution of the program. This approach does not require to store dynamic dependence information as it eagerly computes slices of all variables during execution. But space required for storing slices for all variables, especially in presence of composite data types, is no better than storing DDG or trace. Zhang *et al.* [20] have suggested the use of reduced-order Binary Decision Diagrams (roBDDs) to efficiently represent the run-time sets.

Zhang *et al.* [19] have described No-Preprocessing (NP), a dynamic slicing approach which performs minimal processing during execution. The price of doing no pre-computation is paid during the slicing phase, since the entire trace needs to be analyzed for each slicing request. To address this issue, Zhang *et al.* [19] have suggested a limited pre-computation approach called LP. It augments the trace with summary information to speed up the traversal of long traces. At end of fixed size *trace block*s, a summary of downward exposed definitions of variables in the block are stored. During the backward traversal of the trace (for computing the slice), if the trace block does not contain any definition of the variable one is interested in the block is skipped, thus avoiding traversal of unrelated blocks.

Wang *et al.* [16] have proposed a neat on-the-fly trace compaction scheme for trace-based approaches. They use an extension of SEQUITUR algorithm, called the RLESe (Run Length Encoded Sequitur) to compress the trace sequences on-the-fly. The main highlight of this scheme is that the compacted trace can be traversed without complete de-compression. Experiments have shown that this scheme can achieve a compression ratios ranging from 5 to 100.

Practical dynamic slicing tools need to handle calls to library functions. But very few papers [16, 21] have addressed this important issue. To record the dynamic dependencies exercised one has to instrument the program, either at source-level or at object-level. Since source code of library functions are typically not available, one cannot use source-level instrumentation to track the dynamic dependencies. Venkatesh [15] suggests working at machine code level, to handle library functions uniformly. By maintaining program execution representation and dynamic slices at machine code details, one can compute precise dynamic slices even in presence of library calls.

8 Conclusion and Future Work

We have proposed a novel approach to dynamic slicing, where pre-computed static def-use information of the program is utilized to speed up the analysis of execution traces. By maintaining static def-use information in path vector representation, we exploit the fact that dynamic slicing requires dependence information along execution paths. The hierarchical trace presented is rapidly traversable, and still suitable for compression algorithms to exploit the locality of reference. In future we plan to look at trace compression schemes in more detail, so that space requirements of data trace is also addressed. Since we are using a hierarchical trace representation, one needs to explore if our approach can be adapted for hierarchical interactive slicing [17].

References

1. H. Agarwal and J.Horgan, "Dynamic Program Slicing", *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 246-256, 1990
2. T. Akgul and V. J. Mooney III, "A fast assembly level reverse execution method via dynamic slicing", *International Conference on Software Engineering (ICSE)*, pages 522-531, 2004
3. J. Bharadwaj, K. Menezes, and C. Mckinsey, "Wavefront scheduling - path based data representation and scheduling of subgraphs", *ACM/IEEE Symposium on Micro architecture*, pages 262-271, 1999
4. B.D. Bus, B.D. Sutter, L.V. Put, D. Chanet, and K.D. Bosschere, "Link-Time Optimization of ARM Binaries", *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, pages 211-220, 2004

5. E. Duesterwald, R. Gupta, and M.L. Soffa, “Rigorous Data Flow Testing through Output Influences”, *Irvine Software Symposium*, pages 131-145, 1992.
6. , N. Gupta and P. Rao, “Program Execution Based Module Cohesion Measurement”, *IEEE International Conf. on Automated Software Engineering*, pages 144-153, 2001.
7. R. Jhala and R. Majumdar, “Path Slicing”, *ACM Conference on Programming Language Design and Implementation*, 2005.
8. M. Kamkar, “Interprocedural Dynamic Slicing with Applications to Debugging and Testing”, *PhD Thesis, Linkoping University*, 1993.
9. B.Korel and J.Laski, “Dynamic Program Slicing” *Information Processing Letters (IPL)*, Vol. 29, No. 3, pages 155-163, 1988
10. G.B. Mund, R. Mall, and S. Sarkar, “Computation of intra-procedural dynamic program slices”, *Information and Software Technology*, Vol. 45, pages 499-512, 2003
11. PIN Tool, *http://rogue.colorado.edu/Pin*
12. Vishal B, Raghavan R and P. Shankar, “Instruction Scheduling and Regression Testing using Path Vectors”, *MSc Thesis, IISc*, 2005
13. , Y. Sazeides, “Instruction-Isomorphism in Program Execution”, *Value Predicition Workshop*, 2003
14. H. Tahir and P. Shankar, “Region base Regression Testing using Path Vectors”, *MSc Thesis, IISC*, 2006
15. G.Venkatesh, “Experimental Results from Dynamic Slicing of C Programs”, *ACM Transactions on Programming Languages (TOPLAS)*, Vol. 17, No. 2, pages 197-216, 1995
16. T. Wang and A.Roychoudhury, “Dynamic Slicing on Java Bytecode Traces”, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2007
17. T. Wang and A.Roychoudhury, “Hierarchical Dynamic Slicing”, *International symposium on Software Testing and Analysis (ISATA)*, 2007
18. M. Weiser, “Program Slicing”, *IEEE Transactions on Software Engineering (TSE)*, Vol. SE-10, No. 4, pages 439–449, 1981
19. X. Zhang, R. Gupta, and Y. Zhang, “Precise Dynamic Slicing Algorithms”, *IEEE Conference on Software Engineering (ICSE)*, pages 319-329, 2003
20. X. Zhang, and R. Gupta, “Cost Effective Dynamic Program Slicing”, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004
21. X. Zhang, H. He, R. Gupta, and N.Gupta, “Experimental Evaluation of using dynamic slices for fault location”, *ACM Symposium on Automated analysis-driven debugging (AADEBUG)*, pages 33-42, 2005
22. C.B. Zilles and G. Sohi, “Understanding the Backward Slices of Peformance Degrading Instructions”, *ACM/IEEE International Symposium on Computer Architecture*, 2000