ONTRAC: A system for efficient ONline TRACing for debugging

Vijay Nagarajan, Dennis Jeffrey, Rajiv Gupta and Neelam Gupta
University of Arizona
Department of Computer Science
{vijay,jeffreyd,gupta,ngupta}@cs.arizona.edu

Abstract

Dynamic Slicing [11, 21, 22] is a promising trace based technique that helps programmers in the process of debugging. In order to debug a failed run, dynamic slicing requires the dynamic dependence graph (DDG) information for that particular run. In prior work, address and control-flow traces are collected online and then extensively post-processed offline to yield the DDG, upon which slicing is performed. Unfortunately, the offline post-processing step can be extremely time consuming [21], impeding the use of dynamic slicing as a debugging technique.

In this paper, we present ONTRAC, an efficient online tracing system, that directly computes the dynamic dependences online, thus eliminating the expensive offline post-processing step. To minimize the program slowdown, we make the design decision of not outputting the computed dependences to a file, instead storing them in memory in a specially allocated fixed size circular buffer. The size of the buffer limits the length of the execution history that can be stored. To maximize the execution history that can be maintained, we introduce optimizations to eliminate the storage of most of the generated dependences, at the same time ensuring that those that are stored are sufficient to capture the bug. Our experiments conducted with real bugs confirm the above fact. Other experiments conducted on cpu-intensive programs show that our optimizations are able to reduce the trace-rate from 16 bytes to 0.8 bytes per executed instruction. This enables us to store the dependence trace history for a window of 20 million executed instructions in a 16MB buffer. ONTRAC is also very efficient, only slowing down the execution by a factor of 19, eliminating the slowdown by a factor of 540 due to post-processing.

1 Introduction

It is a well known fact that programmers spend a huge amount of effort in debugging their programs and thus improvement in debugger technology can greatly increase the productivity of programmers. Recently there has been significant research on *Dynamic Slicing* [11, 21, 22] and the indications are that it is a promising debugging technique. The essential idea of debugging using dynamic slicing is quite simple and intuitive. When a programmer observes an erroneous program state, clearly, the cause of the error should have been in one of the several statements that influenced this state. It is precisely these set of statements that the dynamic slicing technique helps us identify. In other words, the dynamic slice of a value computed at an execution point includes all those executed statements which were directly or indirectly involved in the computation of the value. From the definition of the slice, it is quite clear that dynamic slicing requires additional information about the program run to enable the computation of the slice. More specifically, it requires the DDG which is the set of dynamic data and control dependences exercised in the program run. Thus debugging using dynamic slicing consists of two steps:

- **Step 1** Generation of DDG
- **Step 2** Performing dynamic slicing on the DDG.

Previously, it was thought that performing step 2 in reasonable time was impossible due to the huge size of the DDG, even for small program runs. Prior work [21] dispelled this notion by coming up with a highly compact dependence graph representation that made the second step not only practical, but also efficient - in the order of a few seconds. Unfortunately, the generation of this compact dependence representation is expensive. Additional offline post-processing had to be performed on the collected address and control flow traces to yield the compacted representation. In fact, the post-processing[1] in step 1 [21] could take as long as an hour even for short executions of a few seconds (around 5 seconds) of the original program, as seen in Table. 1. Thus the post-processing step causes the program to slowdown by a factor of 540.

Let us now imagine that a programmer uses a dynamic slicing based debugging technique. Debugging code is not

[1] In [21] this step is called the pre-processing step as it precedes slicing. In this work this is called post-processing as it takes place after tracing.

Table 1. Post-processing times in [21]

Benchmark	Post-processing (minutes)
mcf	53.64
bzip	38.36
gzip	23.52
parser	44.06
twolf	65.29
perl	51.12
vortex	44.46

a one step process; it is an iterative process in which the programmer generally makes several changes before arriving at the correct code. Every time the programmer makes a change, the (compacted) dependence graph needs to be generated anew. This means that step 1, in addition to step 2, is in the critical path as far as the programmer is concerned. Clearly, no programmer will want to wait an hour every time he/she makes a change to the code for the new dependence graph to be generated.

In this paper, we address this important issue of building the DDG (step 1) efficiently. Our tracing system, *ONTRAC*, built on top of *DynamoRIO* [13], a dynamic binary instrumentation framework, directly computes the dynamic dependences online, thus eliminating the expensive offline post-processing step. To minimize the program slowdown, we make the design decision of not outputting the dependences to a file, instead storing them in memory in a specially allocated fixed size circular buffer. It is important to observe that the size of the buffer limits the length of the execution history that can be stored, where the execution history is a window of the most recent executed instructions. Since the dependences stored in the trace buffer pertain to the above window of executed instructions, the faulty statement can be found using dynamic slicing only if the fault is exercised within this window. Thus it is important to maximize the length of the execution history stored in the buffer. To accomplish this, we introduce a number of optimizations to eliminate the storage of most of the generated dependences, at the same time we observe from our experiments that those that are stored are sufficient to capture the bug. Besides increasing the length of the execution history that can be stored, our optimizations helps limit the instrumentation overhead, because some of our optimizations identify static dependences for which dynamic instrumentation can be avoided.

The optimizations that we perform to reduce the size of the dependence graph can be classified broadly into two types. While the first kind of optimizations are generic ones, based on program properties, the second kind are exclusively targeted towards debugging. Our generic optimizations are as follows. First, we eliminate the storage of dependences within a basic block that can be directly inferred by static examination of the binary. Second, we extend the same idea to traces of frequently executed code spanning several basic blocks. Third, we detect redundant loads dynamically and exclude the related dependences. Our targeted optimizations are as follows. We first provide support to safely trace only the specified parts of the program, where the programmer expects to find the bug. This is useful because the programmer sometimes has fair knowledge about the approximate location of the bug in the code. For instance, he/she might be modifying a particular function and hence may be relatively sure that the bug is in that function. It is worth noting that a naive solution where the unspecified functions are simply uninstrumented, will not work because this could potentially break the chain of dependences through the user specified functions; this can cause the backward slice to miss some statements from the specified functions that should have been included. Our second targeted optimization is based on the observation that the root cause of the bug is often in the forward slice of the inputs of the program. This observation has been verified in prior work [14, 20]. Thus, by computing the forward slice of the inputs dynamically, we provide support to selectively trace only those dependences that are affected by the input.

Our experiments conducted on cpu-intensive programs from the SPEC 2000 suite show that computing the dependence trace online causes the program to slowdown by a factor of 19 on an average, as opposed to 540 times slowdown caused by extensive post-processing [21]. The optimizations also ensure that we only need to store tracing information at the average rate of 0.8 bytes per executed instruction as opposed to 16 bytes per instruction without them. This enables us to store the dependence trace history for a window of 20 million executed instructions in a 16MB buffer. The rest of the paper is organized as follows. In section 2, we describe the representation of the DDG and describe how the program is instrumented to capture it. In section 3, we briefly explain the implementation of the *ONTRAC* system. In section 4, we discuss the 3 generic optimizations that are based on program properties. Section 5 concerns the two optimizations that are targeted towards debugging. Experimental results are presented in section 6. In section 7 related work is presented and the paper concludes in section 8.

2 Online Computation of DDG

In this section, we discuss the basic representation of our DDG, followed by a description of the instrumentation involved in computing it.

Our representation of the DDG is very similar to the one used in prior work [21]. Each node of the DDG corresponds to a unique static instruction of the program and is identified by its instruction address, the *instr id*. The nodes of the DDG are initially statically linked via static control dependences. As the program executes, the graph is transformed

by introducing edges for the dynamically exercised data dependences. Since the same dependence may be exercised many times, the edge is labeled with additional information to uniquely identify the execution *instances* of the instructions which are involved in the dependence. Execution instances are identified by generating timestamps. A global timestamp value is maintained and each time a basic block is executed, it is incremented. We consider a *dependence* to be either a *data dependence* or a *control dependence*. A data dependence exists between two instructions if one of them *uses* the value *defined* by the other. This is represented by a pair of tuples $(instr\ id_{use}, instance_{use}) \rightarrow (instr\ id_{def}, instance_{def})$. A control dependence exists between two instruction if one of them (the predicate) controls the execution of the other. This is represented by a pair of tuples, $(instr\ id_{use}, instance_{use}) \rightarrow (instr\ id_{predicate}, instance_{predicate})$.

In order to compute the dependences during tracing, we maintain a separate *shadow memory location* for every memory location and register during program execution. The shadow register/memory contains the (instr id, instance) pair of the instruction that writes to the corresponding register/memory location. This enables us to lookup the information when the aforementioned value is subsequently used. It is worth noting that we only compute the data dependences dynamically. However, in this process, the control flow is captured, which allows us to recover the dynamic control dependences.

(a) Original code	(b) Assembly	(c) Instrumentation

int fun (char *input) {
char *array; **int** len,j = -1;
 	
1. len = strlen(input);
*// **root cause***
2. array = malloc(len+2);
 	
3. while (j<len)
{
4. j++;
 	
5. i = 2* j;
 	
6. array[i] = input[j]; ***//crash***
 	
7. if (isupper(array[j]))
{
8. array[i+1] = input[j];
}
else
{
9. array[i+1] =
toupper(array[j]);
}
}
 	
}
	…
	
	//% ebx points to start addr. input
	//% eax points to start addr. array
	//% ecx contains value of i
	//% edx contains value of j
	
	Statement 5:
	
	// i = j
	a) mov %edx, %ecx
	
	*// i = 2*i*
	b) add %ecx, %ecx
	
	Statement 6.
	
	// Load input[j] into %esi
	c) mov (%ebx, %edx), %esi
	
	// Store %esi into array[i]
	d) mov %esi, (%eax, %ecx)
	
	…
		Let:
		i)(pc,inst): (instr addr, instance)
		ii) EA: effective address
		iii) tag_r: shadow for reg 'r'
		iv) $shadow_m$: shadow memory for effective addr 'm'
		
		Instrumentation for instr. a
		i) $tag_{ecx} = (pc_a, inst_a)$ pair
		ii) Output: $(pc_a, inst_a) \rightarrow tag_{edx}$
		
		Instrumentation for instr. b
		i) $tag_{ecx} = (pc_b, inst_b)$ pair
		ii) Output: $(pc_b, inst_b) \rightarrow tag_{ecx}$
		
		Instrumentation for instr. c
		i) $tag_{esi} = (pc_c, inst_c)$ pair
		ii) Output: $(pc_c, inst_c) \rightarrow tag_{ebx}$, tag_{edx}, $shadow_{EA}$
		
		Instrumentation for instr. d
		i) $shadow_{EA} = (pc_d, inst_d)$ pair
		ii) Output: $(pc_d, inst_d) \rightarrow tag_{esi}$, tag_{eax}, tag_{ecx},

Figure 1. Instrumentation for Computing Data Dependences at Runtime.

We explain the computation of data dependences with an example as seen in Fig. 1. The first column shows a simple function in which an input string is processed to obtain a larger string of double the size, such that each character in the original string is duplicated with a capitalized version of the same character. The second column shows two of the statements (5 and 6) from the function in x86 assembly format and the final column shows the instrumentation for instructions corresponding to the statements. There are two main steps involved in the instrumentation for these instructions. First, we store the (instr id,instance) pair of the instruction in the shadow location corresponding to the defined value of the instruction. This is done so that, when the definition is subsequently used, we know where it came from. Second, we output dependency information relating the current instruction with the the shadow locations of the uses. Note that the shadow locations of the uses contain the instr id and instances of those instructions that originally produced the uses. Let us consider instruction a of statement 5, which moves the value of j (stored in ecx register) into i (stored in edx register). The instrumentation for this instruction consists of the following two steps. First, the (instr id, instance) pair for this instruction is stored in the shadow location corresponding to the definition of this instruction, which is the variable i (register ecx). Second, the dynamic dependence relating the current (instr id, instance) pair to the shadow memory (containing the definition information) of the instruction's use. In this case, there is a dependency between the pair $(pc_a, inst_a)$ and the shadow memory of the register edx. The instruction b from statement 5 is similarly instrumented. Let us now consider statement 6, which moves $input[j]$ into $array[i]$. As we see in column 2, this statement consists of two instructions c and d; the former loads $input[j]$ into esi register and latter stores the loaded value into $array[j]$. Let us consider instruction c. The first instrumentation step stores the current instruction information, $(pc_c, inst_c)$ pair into the shadow location for defined register, edx. In the second instrumentation step, we output three dependences to the buffer because the load has three uses: one is the memory location loaded from, and the other two are registers (ebx, edx), which are used in the computation of the effective address of the load. Instrumentation for instruction d, which is a store, is of similar flavor.

As we can see from the first column of Fig. 1, there is a bug in the allocation in statement 2, where the size of the resultant string, array, has not been properly allocated ($len + 2$ has been allocated instead $len * 2$). Let us assume the crash happens in statement 6 (instruction d), because of the overflow of the buffer, array. Recall that we are computing the dependences online during the program run to make sure that a backward slice from instruction d leads to the root cause of the bug, in the statement 2. As we

can see from column 2, one of the uses for instruction d is the start address of array (register eax). Consequently, the shadow location for eax (tag_{eax}) will contain the statement that is responsible for defining it, which is in fact, statement 2. Thus a dynamic slicing based debugger using our computed traces will be able to present the buggy statement to the programmer, in this example.

One-Pass Post-processing. After the program terminates, we consolidate the trace collected in the fixed size buffer, using a fast one-pass post processing step. In this step, we classify the collected dependences instruction-wise in increasing order of instances. This will facilitate the rapid traversal of the dependence trace when dynamic slicing is subsequently performed. In this step, we also compute the dynamic control dependence from the set of multiple potential static control dependences for an instruction, which can happen when unstructured programming statements are used. The dynamic control dependence for an instruction in such a case, is the static control dependence ancestor that was executed the latest. This is easily determined by examining the time stamps of the static control dependence ancestors. Thus, the representation of the DDG at the end of this post-processing step is as indicated in the Fig.2. The figure shows the set of dynamic dependences for one instruction, whose instruction address is pc_1. The dynamic

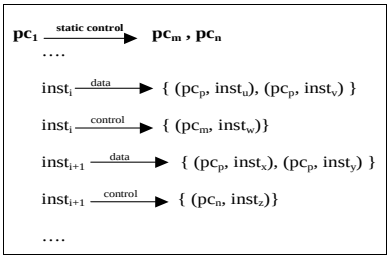

Figure 2. DDG Representation

data and control dependences exercised for each instance of the instruction's execution are represented explicitly. It is important to note that the time taken for this post-processing step is negligible, dominated by the time required for reading the contents of the trace buffer, which has to be done anyway. This is why we claim to have eliminated the expensive post-processing in [21] that involves computation of dependences from address and control flow traces, followed by their compaction. In fact, we found in our experiments that our post-processing step does not exceed 10 seconds for a 16 MB trace buffer.

3 ONTRAC System

In this section, we briefly describe the implementation of ONTRAC, our online tracing system. We built our system on top of *DynamoRIO* a dynamic binary instrumentation framework [13], although other binary instrumentation frameworks like *Pin*[15] could have been used. DynamoRIO supports code transformations on a program,

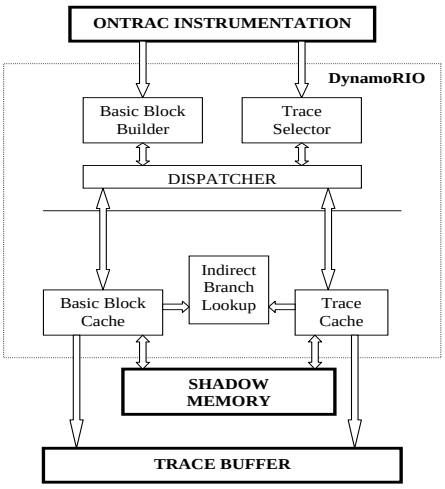

Figure 3. The ONTRAC System

while it executes. It exports an application interface for building dynamic tools for a wide variety of uses: program analysis, profiling, instrumentation, optimization etc. Functionally, DynamoRIO can be viewed as an interpreter which reads/decodes the original instructions from the application and executes it along with the potential instrumentation for that particular instruction. To avoid the emulation overhead, DynamoRIO caches frequently executed code into its *code caches*, from where it is executed natively in the future. This optimization proves to be extremely effective and most programs can be run with very little overhead. The working of dynamoRIO is illustrated in Fig. 3. The top half represents the execution under the control of dynamoRIO while the bottom half represents native execution of instrumented code from the code cache. It is worth noting that very little execution time is actually spent on the top half, which can be considered the instrumentation or the dynamic compilation phase. The *basic block builder* copies the application code one basic block at a time into the basic block code cache. A block that directly targets another block already resident in the code cache is linked to that block to avoid the cost of returning to the DynamoRIO *dispatcher*, through the *indirect branch lookup*. Frequently executed sequences of basic blocks are combined into *traces*, which are placed in a separate code cache. DynamoRIO makes these traces available via its interface for convenient access to hot application code streams. Our main work, the *ONTRAC instrumentation* for computing and outputting the dependence trace, is added after basic blocks/traces are built. Thus the code caches contain our modified instructions, which in-

cludes the instrumentation code.

As we saw in the previous section, we also require additional memory requirements for storing the shadow values for each register and each word of application memory. We use a global array for storing the shadow values for registers. We implemented our shadow memory support into dynamoRIO and made sure that the translation between the effective address and the shadow memory location is efficient. We also made the design decision of maintaining the computed dependence trace in a circular trace buffer instead of writing it to a file. This was done for two reasons. First, we found that writing all the collected information into a file greatly slows down the original program [17]. Secondly, when debugging is performed the trace information has to be read again from the file, which can be avoided if we maintain trace information in a buffer. On the other hand, storing traces in a fixed circular buffer suffers from the obvious disadvantage that trace information is lost when the buffer becomes full. Fortunately, it has been observed that debugging often requires only the tracing information pertaining to a window of recently executed instructions [16]. It was found that a window of 18 million instructions was sufficient to capture the bug for several bugs considered. We later show that for only a 16MB trace buffer, we can store trace information for about 20 millions executed instructions.

4 Generic Optimizations

In this section, we limit the number of dependences that need to be traced, by taking advantage of program properties. Specifically we utilize the fact that several data dependences, especially the dependences between the registers, can be inferred by statically analyzing the code locally within a basic block. Secondly, we extend this idea to *traces* of multiple basic blocks that are executed frequently. Finally we identify redundant loads to eliminate tracing for such loads.

4.1 Basic-block Optimization

Frequently, data dependences between instructions can be inferred statically by performing a simple analysis. For example, consider Fig.4 which is the same function considered as in Fig. 1. We can infer that that statement 5 is data dependent on statement 4 as the former uses the value j defined by the latter. Furthermore, this can be inferred statically by just examining the binary, by observing the fact that statement 5 uses register edx, allocated to variable j, which was defined in statement 4. Here it is important to note that since we use a dynamic instrumentation framework, these static dependences are actually inferred during the instrumentation time. Nevertheless, the benefit of computing the

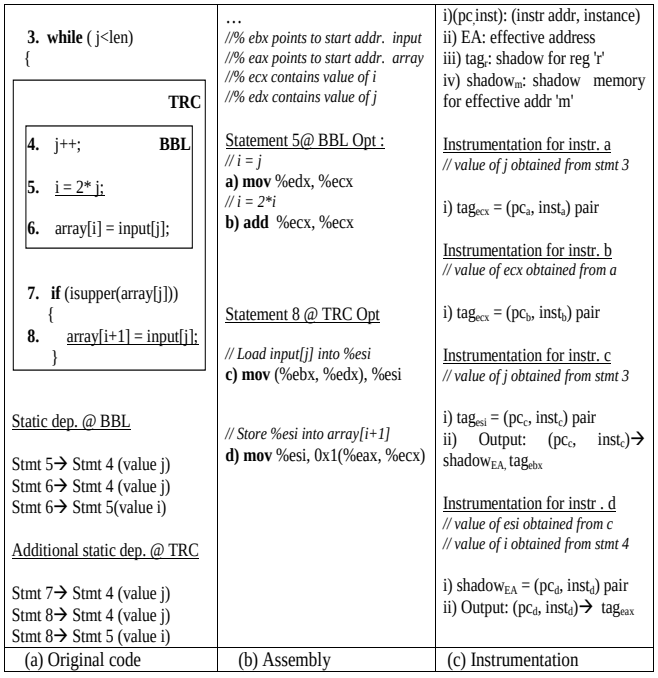

Figure 4. Static Dependences at the Basic block/Trace level

dependency at instrumentation time is equivalent to statically identifying the dependency, since relatively very little time is spent during instrumentation. As we can see from the third column, we do not output dependences for instructions a and b of statement 5, since they were already inferred statically (at instrumentation time). Thus inferring the dependences at instrumentation time, reduces the amount of work that needs to done at runtime and also reduces the traced information.

4.2 Trace Optimization

A *trace* is a sequence of frequently executed basic blocks that do not extend across loop boundaries. Since we use a dynamic instrumentation framework, traces of frequently executed basic blocks can be identified dynamically. In fact, dynamoRIO makes these traces available to the user via its application interface. This makes it possible to apply the optimization discussed in the previous section aggressively across several basic blocks. For example, consider statements 7 and 8 in Fig.4. Let us assume that the input provided is in capitals already, so that the sequence of statements 4,5,6,7 and 8 form a trace. Clearly, we can now infer that the use of j in statement 7 comes from statement 4 and the use of i and j in statement 8 comes from statemented 4 and 5 respectively. As in the basic block optimization these static dependences are output at instrumentation time, when the traces are identified. This obviates the need to output these dependences into the trace buffer during runtime.

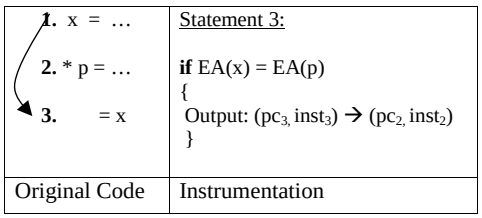

Figure 5. Inferring Memory Dependences in the presence of aliasing

In some situations, even memory dependences can be inferred statically although memory aliasing proves to be a hindrance for this inference. But we can deal with it by dynamically verifying if there has been any memory aliasing. Consider the simple example in Fig. 5. First, we infer that there is a memory dependency between statements 3 and 1; statement 3 uses the global variable x which is defined in statement 1. This inference is performed during instrumentation time and a static dependency between the two statements is speculatively output. Second, we need to make sure that none of the memory instructions between statements 1 and 3 alias with the address of the variable x. Observe that in statement 2, there is a definition to a pointer reference which may or may not point to x. This is done dynamically by instrumenting statement 3 with an additional check, in which the effective addresses of x and p are compared. We output a dynamic dependency between statements 2 and 3, only if the addresses match. It is worth noting that if this memory aliasing happens to occur during program execution, the static dependence edge between statements 1 and 3 becomes superfluous. Thus during dynamic slicing we give precedence to dynamic dependences over the static ones.

4.3 Redundant Load Optimization

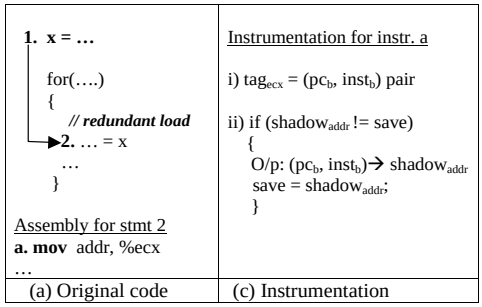

Figure 6. Dealing with Redundant Loads

Prior work [19] has shown that a significant percentage (around 20%) of dynamic loads in program execution are redundant loads. One important reason for such an observation is because of the presence of loads in loops. As we can see in Fig. 6, the load (instruction a) of statement 2 is redundant, since it repeatedly loads the same value across different loop iterations. Thus, all but the load from the first instance is redundant. We detect redundant loads by instrumenting the load under consideration. We check if the current load gets its value from the same dynamic instance of the same store as the load from the previous loop iteration; if so then the current load is redundant. Recall that the (instr id,instance) pair of the store, from which the current load gets it value, is available in the shadow memory of the effective address of the load. This is compared with the corresponding saved pair for the load from the previous iteration. In this example, the saved pair is denoted by the identifier, $save$. In our implementation, we performed the redundant load optimization to selected frequently executed loads inside loops.

5 Targeted Optimizations

The optimizations considered in the previous section are generic in the sense that they are based on program properties and are applicable irrespective of the potential use of the traces. In this section, we discuss two optimizations that are exclusively targeted towards debugging. In our first optimization, we describe how we can selectively trace through functions where the programmer expects to find the bug. In the second optimization, we only trace instructions that are in the *forward slice* of the input.

5.1 Selective Tracing

Programmers often debug large software spread across different functions in various files. More often than not, they have a fair idea of where the bug is, at least at file granularity. Typically, the programmer can isolate a set of few functions, which are likely to contain the bug. In the worst case, it can usually be assumed that the bug is not present in any of the the library functions that the current software uses. In this optimization, we take advantage of the knowledge the programmer has about the bug, to selectively trace only those functions in which the programmer expects the bug to be contained. But this does not mean we can completely ignore all other functions and not instrument them at all. To see why, let us consider a simple example shown in Fig. 7. Let us assume that there is a bug in the original code and the bug lies in the first statement in which an assignment is performed to one of the characters of string a. Further, let us assume there is a call to *strcpy* which copies the string 'a' into another string called ans, which is the output variable. Finally, the program terminates with the value of ans printed out. Since there is a bug in the code, the programmer observes the fact there is a discrepancy with the output and starts a backward slice

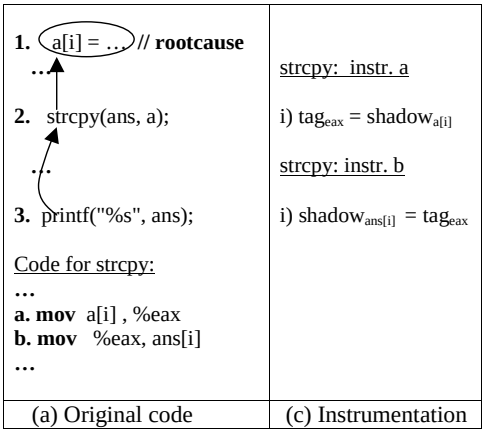

Figure 7. Selective Tracing

based on the value ans, during debugging. Clearly statement 2 will be contained in the backward slice of statement 3 because the latter uses the variable ans defined by the former. Intuitively, statement 1 is expected to be contained in the backward slice of statement 3 for the same reason. It is important to note that this can be inferred only if we can in turn realize that there is a dependency between the strings a and ans via $strcpy$. Now, the programmer surely knows that the root cause is not present in library functions and hence not in $strcpy$. But if we had naively chosen to ignore $strcpy$ completely and not perform any instrumentation for instructions inside $strcpy$, we would have not been able to identify the dependency between the variables a and ans. Hence a backward slice from statement 3 would not reach statement 1, if $strcpy$ was completely ignored. Thus, it is clear that such functions, even if they are not expected to contain the bug, cannot be ignored completely.

In this optimization, we come up with a simple solution, whilst reducing the tracing information collected for functions that are not expected to contain the bug, still ensures that the slicing technique is still effective on the reduced trace collected. The main idea is to not output dependences within such functions, but ensure that we still *propagate* dependences. Suppose there is an instruction with a unique definition and a use; propagation of dependences involves copying the shadow value corresponding to the use into the shadow value for the definition. There is no output of dependences. By propagating the dependency, we are registering the fact that the current instruction alters the existing dependences. By not outputting the the dependence information, we are ensuring the slicing algorithm can safely by-pass the current instruction. Thus, propagation of dependences, is an effective solution for selectively tracing a set of functions. In the above example, consider the two instructions a and b that perform the core function of strcpy. Instruction a, loads a character from the string a into a register and instruction b stores it into ans. The instrumentation involved for performing this propagation is illustrated in the second column and is self explanatory. Since the shadow values of the string a are propagated into the shadow values of ans, statement 1 can still be reached by performing a backward slice from statement 3 and thus the bug will be contained in the slice.

5.2 Forward Slice Optimization

This optimization is based upon the observation that when a software failure occurs during program execution, *the root cause of the failure will generally be contained within the forward slice of the input to the program*. In other words, this leverages the fact that the root cause of a failure will be dependent upon the input to the program. Intuitively, the above fact is not surprising, especially with harder-to-find errors that are only revealed on particular inputs. Moreover, this has been empirically observed in prior work [14] which has shown that forward slices on program input contain the error in most cases. Thus the main idea of this optimization is to *trace only those instructions that are in the forward slice of the input.*

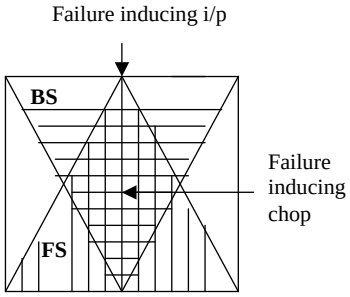

Figure 8. The failure inducing chop

This technique is inspired by the success of *failure-inducing chops* [14] as a technique for debugging. The failure inducing chop is defined as the intersection of backward dynamic slice of the faulty output and forward dynamic slice of the failure inducing input as shown in Fig. 8. It has been found that the failure inducing chop contains the root cause of the error in most cases and it is significantly smaller than the backward dynamic slice, making it an effective technique for debugging. By selectively tracing those instructions that are in the forward slice of the input, we are eliminating the need to perform this intersection later. Moreover, this allows us to trace lesser information and store a larger execution history, for the same buffer size. To implement this, we simply need to propagate an extra forward-slicing bit of data within the shadow contents of variables to indicate whether the variable values are dependent upon the program input or not. Then, we only output dependences involving variables whose forward-slicing bits are set.

6 Experimental Evaluation

We conducted experiments with several goals in mind. First and foremost, we wanted to measure the execution time overhead of computing the dependences online and also observe the effect of our optimizations on the execution time overhead. At the same time, we also wanted to study the rate at which the trace buffer is filled up by executing instructions. We call this the *trace-rate*. We also study the effect of the various optimizations on this trace-rate. Intuitively, this rate should decrease monotonically as and when several optimizations are applied. It is worth noting that the trace-rate is an indirect measure of the length of the tracing history that can be stored in the fixed buffer. The lower the trace-rate, the more slowly the buffer will get filled up and thus the buffer can hence store a longer execution history. We chose a buffer size of 16 MB to store the traces. We made the counter that kept track of the current buffer size to roll over, to avoid repeatedly checking its value during tracing. All experiments were performed on an Intel Pentium 4 - 3GHz machine with 2GB physical memory.

Another important goal of our experiments was to ensure that the optimizations that were targeted towards debugging actually work. In other words, we want to make sure that the optimizations do not accidentally remove important dependences that make it impossible to find the bug through slicing. We present results of this experiment first.

6.1 Efficacy of Targeted Optimizations

In this experiment, we wanted to study whether the reduced trace information collected is still able to capture the bug. For this experiment, we considered 6 real world bugs given in Table 2. Since all the above bugs were memory related, we used only data dependences for the dynamic slicing. For the selective tracing optimization (ST), we performed tracing only in the function in which the bug was present, performing dependence propagation in all other functions. For the forward slice optimization (FS), we only traced the instructions that were in the forward data slice of the input that caused the bug to manifest. As we can see from the last two columns of the table, we were able to find the bug for all benchmarks, even though the two optimizations were performed.

Table 2. Efficacy of targeted optimizations

Benchmark	Bug Type	S.T	F.S
bc-1.06 [1]	heap overflow	Yes	Yes
mc-4.5.55[3]	stack overflow[2]	Yes	Yes
mutt-1.4.2.1i[5]	heap overflow[4]	Yes	Yes
pine-4.44[8]	stack overflow [7]	Yes	Yes
pine-4.44[8]	heap overflow[6]	Yes	Yes
squid-2.3[10]	heap overflow[9]	Yes	Yes

6.2 Overhead of ONTRAC

In this section we evaluate the execution time overhead of online dependence generation relative to the native run of the original program. We considered the SPEC integer programs for this experiment, all of which are CPU intensive programs. We provided the training input set for performing this experiment.

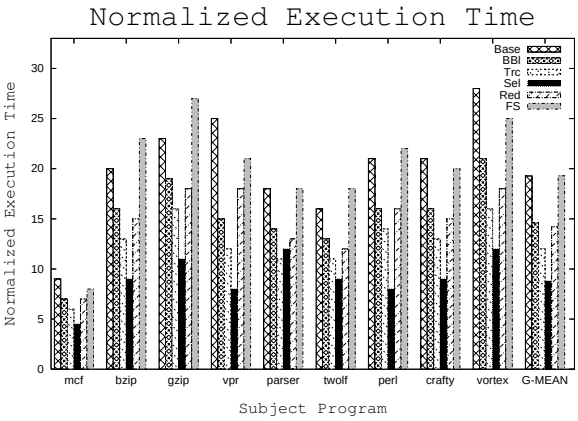

Figure 9. Execution Time Overheads.

As we can see from the Fig. 9, the execution slowdown of the $Base$ configuration in which all dependences are traced is around a factor of 19. Although this is quite high in absolute terms, it is a marked improvement from the slowdown of the post-processing step in [21] which was at least a magnitude higher around a factor of 540 as we saw in Table. 1. The second bar,BBl, shows the execution time overhead after the static dependences present at the basic block level are found at instrumentation time. As expected, this causes a significant drop in the slowdown. The average slowdown experienced after this stage is around a factor of 15. The next optimization, Trc, concerns the deduction of static dependences at the trace level of the program. This further improves the performance of the system by a non-trivial amount. In the selective tracing optimization (Sel), we selected the five statically largest routines from each program and then performed tracing only for those. The choice of the five largest functions, albeit arbitrary, ensures that there is a reasonable size of code we are tracing. As expected, we could save significant execution time by simply performing 'propagation' as opposed to 'tracing' into the non chosen routines. The execution time overhead after this optimization is around a factor of 9. The next bar, Red, refers to the execution time overhead when the load redundancy optimization is implemented online. There is a an increase in the execution time because extra work needs to be done to identify the redundancies. The final bar FS shows the effect of the forward slice optimization. In our experiments we only considered the forward data slice, as

this was sufficient to capture the error for the memory bugs considered. We observe that there is a further increase in the execution time overhead since the forward slice instrumentation has to be performed for most of the executed instructions. The nett program execution slowdown after all optimizations are performed is around a factor of 19. It is important to note that although the last two optimizations causes increases in execution time, they also enable significant savings in the trace rate as we will see in the next section. Finally, we also implemented our system under the Pin instrumentation framework[15], which we used for prototyping due to the ease of implementation. The lack of fine-grained control over instrumentation resulted in ONTRAC running 2-3 times slower in Pin.

6.3 Trace-rate

In this experiment, we measured the rate of trace production, which we define as the number of bytes of tracing data produced per dynamic instruction executed in the original program. As we can see from the Fig. 10, the rate

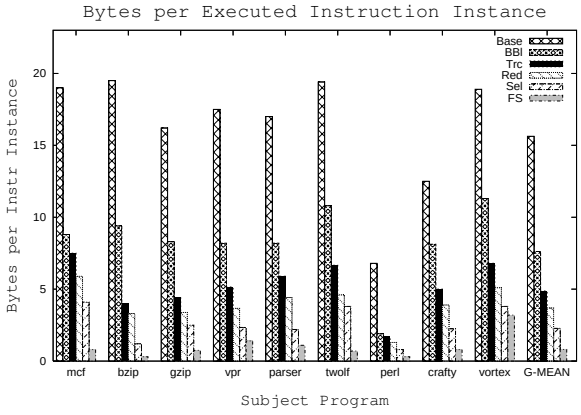

Figure 10. Rate of trace production.

of trace production is quite high without any optimizations, averaging 16 bytes per dynamic instruction. But most of these dependences are static dependences and this can be optimized by examining the dependences within the basic block. There is an almost 50% reduction in the trace production rate if the static dependences within a basic block are identified. There is also a significant reduction in the trace-rate after performing the same optimization at the trace level. At the end of this optimization the trace rate stands at about 5 bytes per instruction. We can further observe that there is about 20% additional reduction in the trace rate when the load redundancies are considered. This confirms with the fact that about 20% of loads even in optimized code are redundant [19].

The final two optimizations are those targeted towards debugging. As we can see from the graphs, each of the final two optimizations results in a significant decrease in the trace rate. There is a about a 50% drop in the trace rate due to selective tracing through special routines. There is a further 4 fold drop in the trace rate when the forward slicing optimization is considered. This is because most of the dependences (about 75%) are surprisingly not in the forward data slice of the input. The average final trace-rate after all optimizations are performed stands at 0.8 bytes per instruction.

6.4 Execution histories stored

One of the goals of the optimizations is to sufficiently reduce the trace-rate so that the execution history of the stored trace can be increased. Accordingly, we measured the execution history stored for a fixed buffer in our implementation that amounts to 16 MB. For this experiment we show the execution histories at the end of three optimizations: the trace level static optimization, the selective tracing debugging optimization and finally after the forward slicing optimization. We did not show the baseline and basic block optimizations for this experiment because they are subsumed by the trace optimization. As we can see from average val-

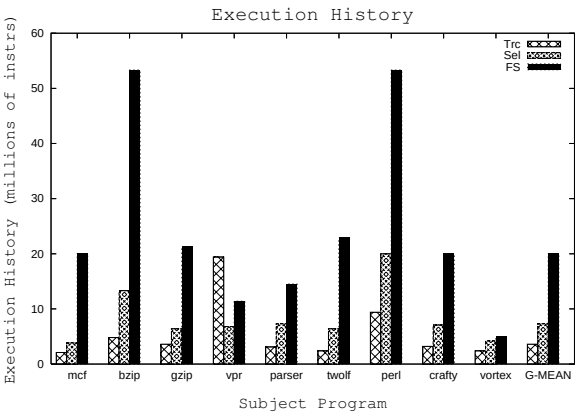

Figure 11. The size of execution history

ues from the graph, the size of the execution history stored after the trace level optimization is about 3.4 million instructions. It increases to about 7 million instructions after the selective tracing optimization is applied and finally reaches the peak value of 20 million instructions when all optimizations are applied together.

7 Related Work

Although there has been several recent work concerning efficient tracing [17, 24, 12], none of them are directly applicable to debugging as they do not directly compute the dynamic dependences. Adapting each of the above approaches to debugging involves an expensive post-processing step, which we avoid in our work. The idea of using a trace buffer was inspired from [24]. Bugnet [16] is a hardware assisted tracing infrastructure that can be used

to replay the program efficiently for debugging. Although, a replay infrastructure is *useful* for performing debugging, the above work does not deal with the problem of actually performing debugging. In contrast, our technique, is first, a fully software based tracing technique. Moreover, we consider the problem of using the collected dependence traces in a dynamic slicing based debugger. There has also been prior work [23] that circumvents the expensive tracing step, for debugging long running programs, by combining tracing along with checkpointing. The main idea of the above paper is based on the repetitive characteristics of long running programs, which is not true for a general class of programs. In our current work we recover control dependences from the data dependences that have been computed online. Recent work [18] shows how to compute the control dependences online; but it does not deal with data dependences.

8 Conclusions

In this paper, we presented ONTRAC, an online tracing infrastructure that is exclusively targeted towards debugging. By performing the dependence tracing step online efficiently, we have now made it more practical for a programmer to use dynamic slicing as a means of debugging. We also evaluated the efficacy of our targeted tracing infrastructure with 6 real world memory errors and found that we were able to capture the error in all cases. For future work, we plan to test this tracing infrastructure with other kinds of bugs.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments. This work was supported by grants from Microsoft and NSF grants CNS-0719791, CNS-0708199, CNS-0614707 and CCF-0541382.

References

[1] Gnu bc. www.gnu.org/software/bc.

[2] Midnight commander exploit. www.securityfocus.com/bid/8658.

[3] Midnight commander. www.ibiblio.org/mc.

[4] Mutt buffer overflow exploit. www.securiteam.com/unixfocus/5fp0t0u9fu.html.

[5] Mutt url. www.mutt.org.

[6] Pine heap buffer overflow. www.securityfocus.com/bid/6120.

[7] Pine stack overflow. www.xatrix.org/advisory.php?s=7408.

[8] Pine website. www.washington.edu/pine/.

[9] Squid buffer overflow exploit. www.securiteam.com/unixfocus/5bp0p2a6ay.html.

[10] Squid. www.squid-cache.org/.

[11] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, New York, NY, USA, 1990. ACM Press.

[12] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drini;, D. Miho, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the second international conference on Virtual execution environments*, pages 154–163, New York, NY, USA, 2006. ACM Press.

[13] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[14] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272, New York, NY, USA, 2005. ACM Press.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[16] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006.

[17] S. Tallam and R. Gupta. Unified control flow and dependence traces. *ACM Trans. Archit. Code Optim.*, To Appear.

[18] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 185–195, New York, NY, USA, 2007. ACM Press.

[19] J. Yang and R. Gupta. Load redundancy removal through instruction reuse. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, page 61, Washington, DC, USA, 2000. IEEE Computer Society.

[20] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM Press.

[21] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 94–106, New York, NY, USA, 2004. ACM Press.

[22] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 33–42, New York, NY, USA, 2005. ACM Press.

[23] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 81–91, New York, NY, USA, 2006. ACM Press.

[24] Q. Zhao, J. E. Sim, W.-F. Wong, and L. Rudolph. Dep: detailed execution profile. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 154–163, New York, NY, USA, 2006. ACM Press.