Memory Slicing

Bin Xin and Xiangyu Zhang
Dept. of Computer Science, Purdue University
West Lafayette, Indiana 47906
{xinb, xyzhang}@cs.purdue.edu

ABSTRACT

Traditional dynamic program slicing techniques are code-centric, meaning dependences are introduced between executed statement instances, which gives rise to various problems such as space requirement is decided by execution length; dependence graphs are highly redundant so that inspecting them is labor intensive. In this paper, we propose a data-centric dynamic slicing technique, in which dependences are introduced between memory locations. Doing so, the space complexity is bounded by memory footprint instead of execution length. Moreover, presenting dependences between memory locations is often more desirable for human inspection during debugging as redundant dependences are suppressed. Our evaluation shows that the proposed technique supersedes traditional dynamic slicing techniques in terms of effectiveness and efficiency.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*, *Tracing*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms

Algorithms, Measurement, Reliability

Keywords

dynamic program slicing, memory dependence graph, data-centric slicing, fault localization.

1. INTRODUCTION

Dynamic slicing was introduced in [9] as a debugging aid. It isolates executed statements that have contributed to a runtime failure through program dependences [7]. Despite its recent progress [22], dynamic slicing still has limitations such as space requirement is tied to execution length and slices contain too much information for a human to explore. These issues arise because existing techniques are code-centric, i.e., dependences are introduced between statement instances. In this paper, we propose a data-centric dynamic slicing technique, in which dependences are introduced between memory locations.

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.

Code

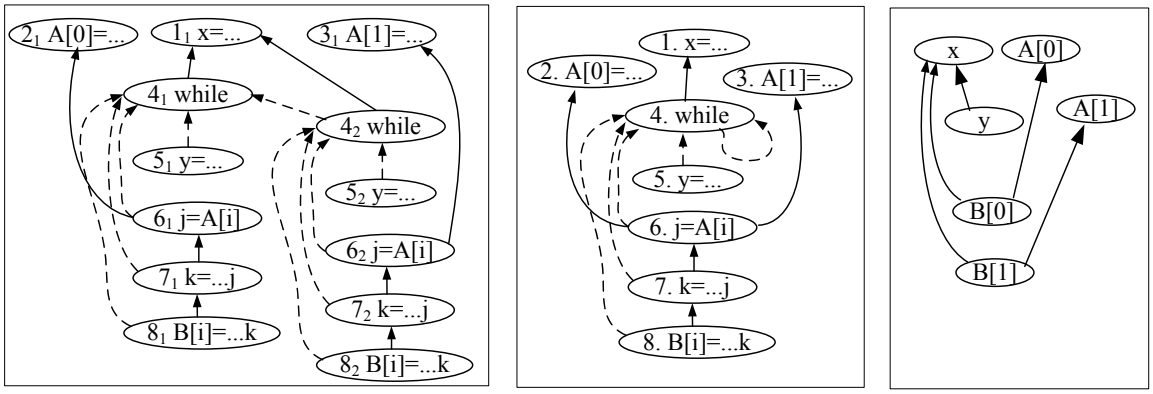
```
   int x, y;
   int A[…], B[…];
   int main () {
     register int i, j, k;
1.   x=…;
2.   A[0]=…;
3.   A[1]=…;
4.   while (i<x) {
5.     y = … ;
6.     j = A[i];
7.     k = … j … ;
8.     B[i]= … k …;
9.     i++;
     }
   }
```

Trace

```
1₁.  x=…;
2₁.  A[0]=…;
3₁.  A[1]=…;
4₁.  while (i<x) {
5₁.    y = … ;
6₁.    j = A[i] ;
7₁.    k = … j … ;
8₁     B[i]= … k …;
9₁     …
4₂.  while (i<x) {
5₂.    y = … ;
6₂.    j = … A[i]... ;
7₂.    k = … j … ;
8₂     B[i]= … k …;
9₂     …
```

Figure 1: Sample code and a sample execution.

The idea can be illustrated by an example. Fig. 1 shows a code snippet and an execution trace. Variables x and y are global variables and thus they reside in memory as arrays $A[]$ and $B[]$ do. Variables i, j and k reside in registers. Traditionally, the dynamic program dependence graph for the sample trace is like the one presented in Fig. 2 (a), in which a node is created for each executed statement, a data dependence edge is introduced between two executed statements if the head statement instance defined a variable and the variable is later used by the tail instance (e.g., there is a data dependence edge between 6_1 and 7_1). A control dependence edge is introduced between a predicate instance and a statement instance if the predicate instance decided the execution of the statement instance (e.g., there is a control dependence edge between 6_1 and 4_1). One can observe that the size of the graph is proportional to the execution length. Earlier study [22] shows that an execution with one hundred million instructions could require 1.5GB space to store the graph without any compression. Furthermore, such a graph is often highly redundant. In the sample code snippet, statement 4 may get executed multiple times so that the dependence between 4 and 1 is exercised repeatedly, resulting in a large number of edges that reveal the same information. For instance, the two data dependences from 4_1 to 1_1 and from 4_2 to 1_1 contain the same



(a) Traditional Dynamic Dep. Graph (b) Statement Based (c) Memory Dep. Graph

Figure 2: Different types of dependence graphs for the sample trace in Fig. 1. For graphs (a) and (b), solid edges and dashed edges represent data dependences and control dependences, respectively. A statement instance is represented as s_i with s being the statement and i being the instance counter.

information. The compression techniques proposed in [22] reduce the space requirement by orders of magnitude, but do not change the space complexity. Besides, they entail heavy-weight program analysis and instrumentation.

A simple idea to reduce space complexity is to discard instance information. More particularly, a node is created for one statement instead of one statement instance. An edge is introduced if there is ever a dependence between instances of two statements. Applying the idea to the sample run results in the graph in Fig. 2 (b). It is easy to infer that the graph size is now decided by the number of statements instead of the execution length. Unfortunately, the graph quality also significantly degrades. For example, suppose it is observed that a wrong value is defined to $B[0]$ at 8_1 due to a fault at statement 2. Using graph (a), the dynamic slice of 8_1, i.e., the set of executed statements that are reachable from 8_1, contains the faulty statement 2. However, when graph (b) is used, the slice also contains statement 3. If the loop iterates for 1000 times, statement 6 depends on the 1000 definition points of the array A elements and the slice contains all those definitions. This bloat is caused by discarding necessary instance information.

We observe that the root cause of these problems is that dynamic dependences are defined between statement instances, i.e., they are *code-centric*. We argue that it is more natural to define dynamic dependences between variables in memory, i.e., making them *data-centric*. The intuition is that memory variables give a strong hint about what information should be kept and what should be discarded. For example, Fig. 2 (c) presents the dependences between variables at the end of the execution. It shows that y is dependent on x because the definition of y is control dependent on the predicate on x at 4. $B[0]$ depends on both $A[0]$ and x. Similarly, $B[1]$ depends on both $A[1]$ and x. Now, even though statement 5 is executed multiple times, only one dependence edge is present between x and y as the multiple instances of 5 operate on the same variable. Furthermore, the memory dependence graph does not undesirably coalesce the dependences caused by $A[i]$ at statement 6 as various instances operate on different memory locations. The first benefit of constructing data-centric dependence graphs is that graph size is no longer tied to the length of an execution but the memory footprint of the execution, which is a much better bound. The second benefit is that it often presents a better view of failure causality and reduces the workload in inspecting a slice. Statements that assign values to registers, e.g., statements 6 and 7, will not be present in a memory dependence graph. It fits the intuition that computation on registers is often temporary so that it is not necessary to present them to the programmer. For instance, loop induction variables mostly reside in registers. They often give rise to very deep dependence chains because of iterative updates. However, inspecting these chains step by step is often fruitless. Such chains are prevented in memory dependence graphs. In Fig. 2, the memory graph (c) is much smaller than the traditional graph (a). Inspecting it requires much less effort.

The proposed technique can be naturally applied to classic debugging. For example, when a core dump occurs due to a segment fault, the current memory graph reflects the dependences between memory locations in the core dump. It is also very convenient, during debugging, for the programmer to pause an execution and investigate relations between memory variables. The technique is also able to identify dependences between inputs and outputs, which are nothing but memory regions. Such a capability is very desirable in scientific computing in the presence of uncertain data. More precisely, the confidence of an output value hinges on the uncertainties of the set of inputs it depends on. Expensive wet-bench experiments can be spared if a positive output unfortunately has a low confidence.

The contributions of this paper are highlighted as follows:

- We propose the novel idea of data-centric slicing, in which dependences are introduced between memory locations instead of statement executions.
- We study a few possible designs of data-centric slicing and identify the one that features effectiveness and efficiency. We formally define the concept of memory dependence graphs and study properties such as space complexity.
- We develop an online set-based graph construction algorithm that is amenable to an efficient implementation using *binary decision diagrams* (BDDs), which exploit the significant overlap among memory dependence graphs.
- We evaluate the technique on a set of medium-sized programs. The results indicate that the proposed data-centric slicing technique supersedes traditional dynamic slicing techniques in terms of effectiveness and scalability.

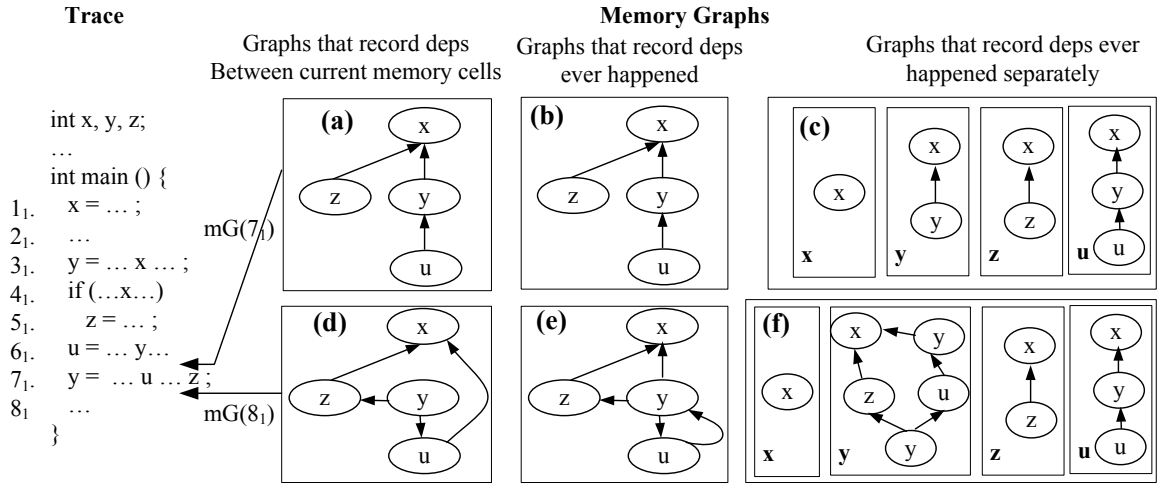


Figure 3: Possible designs for memory dependence graph.

2. MEMORY DEPENDENCE GRAPH

A key concept in our data-centric slicing technique is *memory dependence graph* (mG). Given a program execution point, usually the point where a failure happens, mG describes dependence relations between memory locations. It is constructed and maintained online during program execution. Even though the idea is intuitive, there exist various possible designs.

Design One. In this design, mG reflects dependences between memory locations at the *current* virtual space snapshot. More specifically, given a program execution point s_i, denoting the ith execution instance of statement s [1], a memory location m is said to be dependent on another location n if and only if the current value at m was affected by the current value at n without going through any other memory locations.

Fig. 3 shows examples for different designs. An execution trace is presented on the left. Here we assume that all variables are memory resident. Graphs (a) and (d) represent the mGs for design one when the execution reaches points 7_1 and 8_1, respectively. Right before 7_1, variable y depends on x, denoted as $y \xrightarrow{md} x$, because the current value of y was defined using the current value of x at 3_1. Similarly, $u \xrightarrow{md} y$. Variable z depends on x because the definition of the current value of z was the result of the branch outcome regarding the current x. In other words, x controls the value of z.

Such a design is less effective when memory locations are overwritten. Consider the same example, at statement 7_1, the value of y is overwritten, which implies now the current value of u is no longer dependent on the current value of y, but rather the current value of x. More particularly, upon y being overwritten, the dependence between u and y has to be discarded and a dependence between u and x, on which the obsolete y value is dependent, is introduced. The resulting graph is shown in Fig. 3 (d). One can infer that if u were to be overwritten, a variable that is dependent on u, assuming it to be v, would be re-connected to x. As a result, when such a graph is presented to a user, one has to figure out why there is a dependence between v and x, which is often hard as the corresponding chain $v \xrightarrow{md} u \xrightarrow{md} y \xrightarrow{md} x$ is no longer visible. We call such a problem the *over abstraction* problem.

Design Two. In order to avoid over abstraction, the second possible design is not to discard any dependences that ever occurred. More specifically, overwriting a variable does not induce any reconnections and dependence edges can only be added to the graph without being discarded. Fig. 3 (b) and (e) present the mGs for the second design. The graph before 7_1 is the same as that in the first design. The difference lies in the graph after 7_1. The overwriting of y at 7_1 introduces two new dependence edges $y \xrightarrow{md} u$ and $y \xrightarrow{md} z$ without discarding the original dependence $u \xrightarrow{md} y$. Therefore, at 8_1, the computation of the current value of u, i.e. the chain $u \xrightarrow{md} y \xrightarrow{md} x$, is explicit in the graph. However, such a chain is not distinguishable from other bogus dependence chains starting from u, e.g., $u \xrightarrow{md} y \xrightarrow{md} z \xrightarrow{md} x$. In other words, the second design solves the over abstraction problem, but introduces a new problem, which we call the *over aggregation* problem. One may suggest distinguishing different writes to y using an instance counter or timestamps. Such a solution will make the space complexity of the resulting graph to be $O(\textit{execution length})$.

To overcome the problems in the previous designs, we propose to maintain memory dependence graphs *independently* for each variable. Informally, graphs for different variables are separated so that writing to a variable x only updates the graph of x without changing the graphs of other variables. Fig. 3 (c) and (f) demonstrate such a design. Before 7_1, each variable has its own dependence graph, delimited by a box annotated with the variable. After y is overwritten at 7_1 using u and z, the new graph of y is constituted by copying the graphs of u and z and connecting u to y and z to y, resulting in the graph (f) at 8_1. Note that the graphs of u and z remain untouched. More particularly, overwriting y at 7_1 does not change the node y in u's graph. As a result, the dependence between the old value of y and the current value of u is not obscured by dependences introduced for the new value of y.. Therefore, it overcomes the over abstraction problem by not discarding dependence edges. It also greatly mitigates the over aggregation problem by separating the graphs of individual variables. For example, the graph for u precisely explains the computation of its current value, without being obscured by bogus dependences caused by other variables as discussed earlier.

Next, we formally define memory dependence graphs.

DEFINITION 1. *Given an execution point and a memory location, the memory dependence graph regarding the location at the execution point is a pair* (N, E) *with* N *being a set of nodes and* E *being a set of edges. A node is represented as a pair* $(\textit{vAddr}, \textit{dAddr})$

[1] A statement can execute multiple times during an execution.

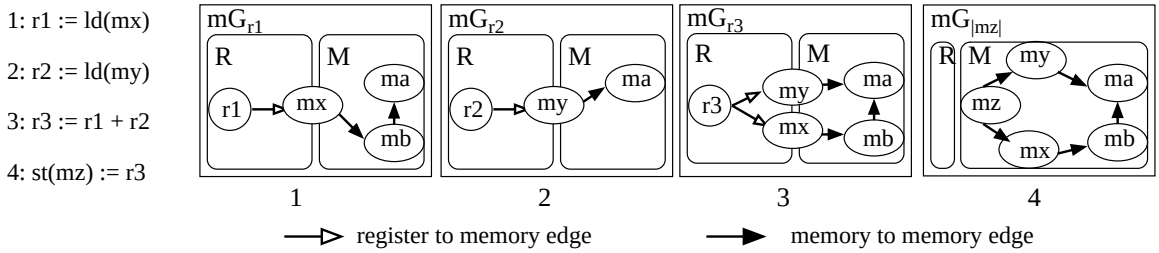


Figure 4: A memory dependence graph construction example. The trace is on the left, and the four graphs represent the mG snapshots after each step, with R representing mG^R, M representing mG^M.

with vAddr being a memory location and dAddr being the program counter of the definition point of the memory location. There is a dependence edge between two nodes if and only if there is a dynamic program dependence path (in terms of dynamic data and control dependences) between them and along the path there is not a definition to any memory locations.

A memory dependence graph is defined regarding a memory location at an execution point and all the nodes in the graph are memory locations that have contributed to the current value of the given location. In fact, the mG of a memory location is similar to the concept of the dynamic slice of a variable. Note that according to the definition, a node is created for a unique pair of a memory location and the definition point to that location. The reason is that there may be multiple definition points to the same memory location. It is needed to distinguish these definition points because they have distinct semantics.

PROPERTY 1. *The space requirement for the mGs of an execution with the memory footprint of M bytes and N unique statements is $O(M^3 \cdot N^2)$.*

The proof is straightforward. For each of the M bytes, the number of nodes of its graph is $O(M \cdot N)$ and thus the number of edges is $O(M^2 \cdot N^2)$. As a result, the total space requirement is $O(M^3 \cdot N^2)$. In fact, the practical bound is significantly better because: (1). many programs consume memory at the scale of megabytes and a particular byte often only depends on a number of other bytes; (2). memory graphs have significant redundancy that can be exploited to achieve space efficiency. We will discuss how to use *binary decision diagrams* (BDDs) to exploit such redundancy in Section 4.

3. GRAPH CONSTRUCTION

In order to facilitate efficient graph construction, we change the definition of memory dependence graph to an equivalent form.

DEFINITION 2. *Given an execution point and a memory location, the memory dependence graph regarding the location at the execution point is a set of four-tuples $\langle vAddr_h, dAddr_h, vAddr_t, dAddr_t \rangle$. Each tuple represents a memory dependence edge with $(vAddr_h, dAddr_h)$ being the head node, including the memory location and its definition, and $(vAddr_t, dAddr_t)$ being the tail node.*

This definition formulates memory graph as a set of tuples (relations). It is trivial to see that it is equivalent to the previous definition. The reason we use a set based definition is that, as it will be illustrated in Section 4, sets can be represented as BDDs to suppress redundancy and achieve efficiency. Given the modified definition, graph construction can be defined with set operations. To better illustrate graph construction, we first give a simplified trace grammar that describes the forms of instructions executed at runtime. Note that one should treat it as a grammar for execution traces instead of a grammar for programs. The grammar is shown in Listing 1.

In this grammar, *r* denotes registers, *const* denotes constants. Together, registers and constants form atoms, denoted by *a*. Expressions, e, can be atoms, memory loads, unary or binary expressions. A value is loaded from memory location a by *ld(a)*. Due to the characteristics of traces, we design a simple grammar that requires addresses to be atoms (Similarly, we require the operands of *uop* and *bop* expressions to be atoms). While in a high level language or a high level IR, an address could be an expression, in a low level IR like the one we encounter in our implementation, an expression is flattened into multiple IR statements. In other words, the address is computed and stored into a temporary register before being used by a load/store statement. *Uop* and *bop* represent unary and binary operations, respectively. Statements, s, can be in the forms of register assignments, memory assignments, conditionals, and unconditional jumps. One may notice that the conditional statement does not have the `else` branch, since the grammar is designed for execution traces, and at runtime, only one branch is executed. For simplicity of presentation, we omit the definition PCs of an dependence edge in the rest of this section, that is to say, an edge is a pair of memory locations instead of a four-tuple.

Listing 1: Simplified Trace Grammar

```
a = const | r
e =   a | ld(a) | uop(a) | bop(a,a)
s =   r := e
    | st(a) := e
    | if(a) goto a
    | jmp a
t = ϵ | s; t
```

Although we are only interested in memory to memory edges when constructing memory dependence graphs, we need to temporarily maintain register to memory dependence edges. However such dependences have very short life times, and they will be translated into real memory dependences once values in registers are written to memories. For the purpose of presentation, we divide the set of dependence edges into two parts:

$$mG = mG^M \cup mG^R,$$

with mG^M subgraph containing memory to memory (*m2m*) dependences, and mG^R subgraph containing register to memory (*r2m*) dependences. We use mG_x denotes the dependence graph of an entity x, which could be a memory location or a register.

Figure 4 gives an overall idea of how the two subgraphs are constructed and propagated for a simple trace segment, assuming that initially, the memory graphs for locations mx and my contain

edges: $\{\langle mx, mb\rangle, \langle mb, ma\rangle\}$ and $\{\langle my, ma\rangle\}$, respectively. After the first step, a r2m edge is added between $r1$ and mx, representing the dependence of the current value in $r1$. Similarly, an edge is added between $r2$ and my after step 2. After step 3, the graphs of $r1$ and $r2$ are first combined, and the temporary edges between $r1$ and mx and between $r2$ and my are replaced by the two edges from $r3$ to mx and my, reflecting the dependences of $r3$. The r2m edges from $r3$ to mx and my are replaced by two m2m edges from mz to mx and my after the final step. The important observation is that we only maintain sufficient r2m edges to ensure the final m2m edges can be correctly identified, which explains why the r2m edges of $r1$ and $r2$ can be discarded.

We use the following helper functions to access these subgraphs for atoms and expressions:

$$rm(atom) = \begin{cases} \emptyset, & atom \doteq const \\ mG_r^R, & atom \doteq r \end{cases}$$

$$mm(atom) = \begin{cases} \emptyset, & atom \doteq const \\ mG_r^M, & atom \doteq r \end{cases}$$

$$rm(e) = \begin{cases} rm(a), & e \doteq a \ \underline{or} \ uop(a) \\ rm(a_1) \cup rm(a_2), & e \doteq bop(a_1, a_2) \\ rm(a) \cup \{\langle \_, |a|\rangle\}, & e \doteq ld(a) \end{cases}$$

$$mm(e) = \begin{cases} mm(a), & e \doteq a \ \underline{or} \ uop(a) \\ mm(a_1) \cup mm(a_2), & e \doteq bop(a_1, a_2) \\ mm(a) \cup mG_{|a|}^M, & e \doteq ld(a) \end{cases}$$

$$replace(G^R, x) = \{\langle x, m\rangle \mid \langle \_, m\rangle \in G^R\}$$

The function $rm()$ is used to extract the register-to-memory subgraph from the dependence graph of an atom or an expression. Similarly, function $mm()$ is used to extract the memory-to-memory subgraph of an atom or an expression. For example, $rm(r1) = \{\langle r1, mx\rangle\}$ and $mm(r1) = \{\langle mx, mb\rangle, \langle mb, ma\rangle\}$ for $r1$ in Figure 4. Function $replace(G^R, x)$ is used to replace the from node (the register) in a register-to-memory graph G^R with node x. More particularly, when an expression e is a binary operation, $rm(e)$ and $mm(e)$ compute the union of the subgraphs of the two operands, reflecting the newly computed value depends on whatever in the dependence graphs of the two operands. When e is a load instruction, $mm(e)$ is the union of the address' dependence graph $mm(a)$ and the dependence graph of the value stored at the address, denoted by $mG_{|a|}^M$; $rm(e)$ is the union of the region-to-memory graph of a and the new edge of $\langle \_, |a|\rangle$, which represents a temporary (don't care) register is associated with e and the register depends on the address $|a|$. Note that such registers will be replaced eventually.

The following discussion of graph construction is driven by the grammar.

***Case I*: r := e**. These are register assignment statements. With the helper functions defined above, the computation of mG_r can be specified as:

$$mG_r = mG_r^R \cup mG_r^M \tag{1}$$

$$mG_r^R = replace(rm(e), r) \tag{2}$$

$$mG_r^M = mm(e) \tag{3}$$

Intuitively, it replaces all the from nodes (temporary registers) in $rm(e)$ with r and then unions with the memory-to-memory dependence graph.

***Case II*: st(a) := e**. The computation of mG for the memory location $|a|$, denoted by $mG_{|a|}$, is defined as:

$$mG_{|a|} = mG_{|a|}^R \cup mG_{|a|}^M \tag{4}$$

$$mG_{|a|}^R = \emptyset \tag{5}$$

$$mG_{|a|}^M = mm(a) \cup mm(e) \cup replace(rm(a) \cup rm(e), |a|) \tag{6}$$

In particular, the last equation says that the memory-to-memory graph is the union of the memory-to-memory graphs of a and e, and the new edges introduced between $|a|$ and memory locations that were loaded to compute a and e.

***Case III*: if(a) goto label**. Handling conditional statements is key to including control dependence into memory dependence graphs. Since a statement sequence t that follows this *if* statement is control dependent on predicate a, the memory dependence graph of the predicate should be propagated to each statement in t. We use a pseudo register cd to hold the value of the current branch condition. The current control dependence graph is denoted as mG_{cd}.

Before the execution of t, the old mG_{cd} is saved to a stack as follows:

$$stack.push(mG_{cd})$$

Then, the new mG_{cd} is defined as:

$$mG_{cd}^R = mG_{cd}^R \cup replace(rm(a), cd) \tag{7}$$

$$mG_{cd}^M = mG_{cd}^M \cup mm(a) \tag{8}$$

The two equations reflect the current control dependence is a union of the previous control dependence and the dependence of the predicate a. For instance, in the case that one if statement nests in another if statement, a statement s guarded by the inner predicate should be transitively control dependent on the outer predicate. In other words, the dependences of outer predicate need to be propagated to s.

After the execution of t, mG_{cd} is restored to its original state through the following:

$$mG_{cd} = stack.pop()$$

It means the execution after this point is no longer control dependent on the predicate a. With mG_{cd}, we can include control dependence into the memory graphs of registers and memory locations in *cases I* and *II*, by performing the following at the end:

$$mG_r = mG_r \cup mG_{cd}^M \cup replace(mG_{cd}^R, r) \tag{9}$$

$$mG_{|a|} = mG_{|a|} \cup mG_{cd}^M \cup replace(mG_{cd}^R, |a|) \tag{10}$$

In practice, program execution is often not well-structured because of `continue`, `break`, or `setjmp`/`longjmp` constructs. Due to space limitations, handling such cases is omitted. The basic idea is to rely on post-dominator computation instead of syntactic structure [20].

No special action is required for the remaining case of unconditional jumps.

4. IMPLEMENTATION

The system is implemented on Valgrind [12]. Control flow analysis on binaries is implemented on Diablo [6]. *Binary decision diagrams* (BDDs) [10] are used to represent graphs, which are indeed sets of dependence edges. As mentioned earlier, each memory location has its own graph. The graphs of different memory locations have significant overlap. Such redundancy can be exploited by BDDs. BDDs are data structures to represent Boolean functions

`t5 = GET(20)`	*//See Equation 1: case* $e \doteq r20$ $mG_{t5} = mG^M_{r20} \cup replace(mG^R_{r20}, t5)$ *//See Equation 9* $mG_{t5} = mG_{t5} \cup mG^M_{cd} \cup replace(mG^R_{cd}, t5)$ `t5 = GET(20)`	`t5 = GET(20)`
`t4 = Add32(t5,-8)`	*//See Equation 1: case* $e \doteq bop(t5, const)$ $mG_{t4} = mG^M_{t5} \cup replace(mG^R_{t5}, t4)$ *//See Equation 9* $mG_{t4} = mG_{t4} \cup mG^M_{cd} \cup replace(mG^R_{cd}, t4)$ `t4 = Add32(t5,-8)`	*//T4 depends only on %ebp, thus ignored.* `t4 = Add32(t5,-8)`
`t6 = GET(0)`	*//See Equation 1: case* $e \doteq r0$ $mG_{t6} = mG^M_{r0} \cup replace(mG^R_{r0}, t6)$ *//See Equation 9* $mG_{t6} = mG_{t6} \cup mG^M_{cd} \cup replace(mG^R_{cd}, t6)$ `t6 = GET(0)`	*//Save the dependence of t6 to r0 at instrument time.* `t6 = GET(0)`
`STle(t4) = t6`	*//See Equation 4: case* $a \doteq t4, e \doteq t6$ $mG_{\|t4\|} = mG^M_{t4} \cup replace(mG^R_{t4}, \|t4\|)$ $mG_{\|t4\|} = mG_{\|t4\|} \cup mG^M_{t6} \cup replace(mG^R_{t6}, \|t4\|)$ *//See Equation 10* $mG_{\|t4\|} = mG_{\|t4\|} \cup mG^M_{cd} \cup replace(mG^R_{cd}, \|t4\|)$ `STle(t4) = t6`	*//T6's dependence on r0 forwarded here.* $mG_{\|t4\|} = mG^M_{r0} \cup replace(mG^R_{r0}, \|t4\|)$ $mG_{\|t4\|} = mG_{\|t4\|} \cup mG^M_{cd} \cup replace(mG^R_{cd}, \|t4\|)$ `STle(t4) = t6`
(a)	(b)	(c)

Figure 5: Examples of the Valgrind IR for binary instruction `mov %eax, -8(%ebp)` (a), instrumentation for the graph construction algorithm (b), and the effect of optimizations (c).

(or truth tables) [4], and thus they can be used to represent relations (sets) with finite domains. For example, an mG is represented with a 4-tuple relation in this paper. BDDs use a global data structure, which is a binary graph, to represent a very large number of relations. A relation can be accessed through an index (an integer) to the global data structure. Different relations have different indices. Equality of two relations can be decided by comparing their indices. Operations on relations can be translated to function calls to the BDD package with relation indices as the parameters. Relation operations used in this paper such as union and element replacing can be straightforwardly implemented with BDDs. Details are elided due to space limitations.

Figure 5(a) shows some examples of Valgrind IR, on top of which our algorithm is implemented. (For presentation purpose, some non-essential parts of the IR are omitted, like the size annotations on operands.) Here it uses 4 IR statements to translate the native instruction `mov %eax, -8(%ebp)`. Native registers are assigned indices. They are read/written with the IR primitives `GET/PUT`, for example, the index for %ebp is 20 and that for %eax is 0. Valgrind IR also introduces its own class of temporary variables (IR Temps), like $t4$, $t5$, and $t6$ in the example. The rest is self-explained: `Add32` constructs an address expression by adding two 32-bit integers, and the `STle` statement stores a value to an address in the memory.

Figure 5(b) shows the instrumentation of memory graph construction for the IR in (a). The first IR instruction `t5=GET(20)` belongs to case I as described in Section 3, i.e. the **r:=e** category with r being $t5$ and e being a register $r20$(%ebp). The first line of instrumentation is to instantiate the memory graph of $t5$ with that of $r20$ following Equation 1. More specifically, it first introduces new edges from $t5$ to the memory locations that were used to compute the value of $r20$ by calling *replace()*. It then further unions the set of new edges with the memory-to-memory graph of $r20$. The second line of instrumentation is to union the current control dependence graph with the computed $t5$ graph, following Equation 9. The instrumentation for the second IR statement `t4=Addr32(t5,-8)` is similar. For the store statement, which falls into case II, the first two lines of the instrumentation correspond to Equation 4. They first compute the memory graph for $t4$ and then union it with the memory graph of $t6$. The third line is to include control dependence according to Equation 10.

4.1 Optimization

Strictly applying the instrumentation rules in Section 3 unfortunately leads to substantial runtime overhead. We develop two optimizations that can significantly improve efficiency.

OPT-TEMP. One of the major sources of inefficiency comes from our treatment of IR Temps such as $t4$ and $t5$ in Figure 5. The frequency of definitions and references of IR Temps is much higher than those of registers and memory values. In our base implementation, many function calls are used for manipulating the dependences of IR Temps. However, two observations help us alleviate this problem.

Firstly, IR Temps are often used to compute stack addresses. Without compromising the important feature of supporting dependence tracing for pointers, we can largely reduce the instrumentation by precluding dependence tracing for stack addresses. More particularly, we can skip dependence tracking for all IR Temps that only depend on values of register %ebp and %esp, or constants. Thus, we do not need to instrument the assignments to $t4$ and $t5$, as shown in Figure 5(c). The dependence on $t4$ in the store statement can be removed too.

Secondly, Valgrind IR Temps have the SSA property, i.e. each IR Temp is guaranteed to be defined only once. Also, IR Temps are only visible in a super block (a translation unit in Valgrind). These two properties allow us to move the handling of IR Temp dependences from runtime to instrumentation time. Specifically, the dependences and definitions of IR Temps are analyzed and identified at instrumentation time and merged directly into functions handling dependences for register or memory values. Only when the definition of an IR Temp depends on a register or memory that is later rewritten in the same block, do we need to use separate function calls for the IR Temp. Thus in Figure 5(c), we see that for the store statement, no explicit calls are needed to propagate the dependences from $t6$. Its dependences are directly forwarded and merged with the memory graph computation for $|t4|$.

OPT-CACHE. The second source of inefficiency comes from the usage of BDDs. One of the basic operations in our algorithm is to create an edge from two addresses. This involves six function calls to various BDD library functions. Since we are building slices for all live values as the program executes, the same dependence edge might appear in the slices of more than one value. We found that a caching mechanism that avoids repeated use of expensive BDD operations to create the same edge can greatly reduce runtime overhead. This also means that once an edge is created, it cannot be reclaimed by the BDD kernel, suggesting a higher space overhead. However, our experience shows that the increase in memory consumption is insignificant.

5. EVALUATION

In this section, we study the characteristics of data-centric slices and their effectiveness in debugging. We also show the runtime cost of our algorithm. Table 1 shows an overview of the set of benchmarks used in the experiments. They are collected from the BugBench suite [11]. They all contain real known bugs that are summarized in the last column.

5.1 Effectiveness

The first experiment is about the effectiveness of data-centric slicing in debugging. As we discussed in Section 3, mGs are computed and stored as BDDs for every register and memory values as a program executes. When the program crashes, the memory graph corresponding to the wrong observable output is dumped, for example, a wrong address value that caused the SIGSEGV. The effectiveness study involves how this memory graph (i.e. memory-data-centric slice) can be used to find the root cause in the source program. We assume that a breadth-first search strategy is adopted in inspecting the graph, starting from the failure node. For comparison, we also applied the conventional code-centric dynamic slicing and constructed dependence graphs from instruction instances. Similarly, a breath-first search is used to reach the root cause nodes.

Table 2 shows that for each benchmark, how many nodes in the graph have been visited before reaching the root cause node. If we assume that bugs are likely in the user code instead of library code, then programmers may only want to inspect those nodes in the user code. We also show the distinct number of source line locations of these nodes, since they are better estimations of the amount of code that programmers have to examine.

5.1.1 Bc-1.06

Bc is an arbitrary precision calculator utility tool commonly found in Unix-based systems. In *bc-1.06*, a coding mistake leads to a variable being confused with another variable of a similar name. As a result, an incorrect boundary condition is used when filling a buffer, causing an overflow. Parts of the code for this benchmark are shown in Fig. 6. The root cause is at line 4706: variable *v_count* should be *a_count*. In our experiment, *bc* eventually crashes in a C library function. This type of failure can be difficult to debug.

With the data-centric slice, 63 nodes have to be visited before the root cause is identified. Among them, 20 nodes are in user code. They correspond to 49 and 16 source line locations. With the conventional code slice, 93 nodes have to be traversed before the root cause is reached, with 41 of them in the user code, corresponding to 37 and 13 source lines. The number of nodes visited in the data slice is 32.3% smaller than that in the code slice, while the numbers of user-level source code locations are comparable.

We know the direct cause for this *bc* bug is the overflow at line 4707, with the buffer and its size being defined at line 4699 and 4698. Fig. 7 shows how the causing nodes are reached in the memory dependence graph (Note: for case study purpose, we have merged all source files into a single file *bc_1.06-noLn.c*). The number on an edge indicates how many edges are used in the code-centric slice to establish the dependence that is represented by a single edge in our memory graph, i.e., it indicates how many instruction instances are along the data/control dependence path between two memory locations. It clearly demonstrates that memory dependence graph can suppress less useful information and presents a better view for debugging. This figure also explains why the size of a memory graph can be much smaller than that of a code-centric slice.

```
void more_variables(void)                                    4659
{                                                            4660


  v_count += 4;                                              4669
  variables = (bc_var **)bc_malloc(...                       4670
              v_count * sizeof(bc_var*)));                   4671


}                                                            4687
void more_arrays(void)                                       4688
{                                                            4689


  a_count += 4;                                              4698
  arrays = (bc_var_array **)bc_malloc(...                    4699
           a_count * sizeof(bc_var_array *)));               4700


  while (indx < v_count) {                                   4706
    (*(arrays + indx)) = (bc_var_array *)(0);                4707
    indx ++;                                                 4708
  }                                                          4709
```

Figure 6: Parts of bc-1.06 code, from the source file *bc_1.06-noLn.c*.

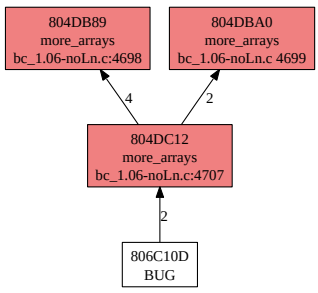


Figure 7: Part of the mG for the *bc* crash point. The number on an edge represents the length of the data/control dependence path in the code-centric dependence graph that corresponds to the edge in the mG. Highlighted nodes are memory locations defined in user code.

5.1.2 Squid-2.3

Squid is an Internet object caching proxy, supporting common Internet protocols like HTTP and FTP. It can be used to develop web proxies and content serving applications. *Squid-2.3* has a bug that causes it to crash when serving requested URLs that contain escaped characters. Fig. 8 shows parts of the code from *Squid-2.3* that explain the bug. The buffer length is computed at line 1004, which is used to allocate a buffer at line 1019. An overflow can happen in the call to `strcat` at line 1022. In our experiment, *squid* eventually crashes inside a C library function.

In the mG of the crashing point, the number of nodes to be examined before reaching the root cause is 245, compared to 29269 in

Benchmarks	LOC	Description	Bug type
squid-2.3	93.5k	Web proxy cache server	Heap buffer overflow
bc-1.06	17.0k	Arbitrary precision calculator	Heap buffer overflow
man-1.5	4.7k	Redhat documentation tools	Global buffer overflow
gzip-1.2.4	8.2k	File (de)compression	Global buffer overflow
ncompress-4.2	1.9k	File (de)compression	Stack smash

Table 1: Benchmark overview.

	Memory-data slice		Code-centric slice	
Benchmark	All	User-Level	All	User-Level
squid	245/78	145/50	29269/2207	9177/1141
man	182/107	75/48	149/57	139/54
bc	63/49	20/16	93/37	41/13
gzip	4/3	3/2	73/18	13/5
ncompress	2/2	1/1	1866/364	36/9

Table 2: The number of nodes/lines visited before reaching the root cause in both our data slices and conventional code slices.

```
ftpBuildTitleUrl(FtpStateData * ftpState)                    999
{                                                            1000
    request_t *request = ftpState->request;                  1001
    size_t len;                                              1002
    char *t;                                                 1003
    len = 64 + ...;                                          1004
    ...                                                      1005

    t = ftpState->base_href = xcalloc(len, 1);               1019
    strcat(t, "ftp://");                                     1020
    if (strcmp(ftpState->user, "anonymous")) {               1021
    strcat(t, rfc1738_escape_part(ftpState->user));          1022
    ...                                                      1023
```

Figure 8: Parts of squid-2.3 code, from the source file *ftp.c*.

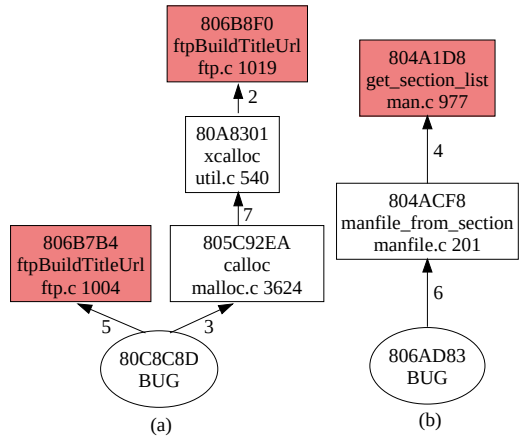


Figure 9: Bug memory graph for: *squid* (a) and *man* (b). The number on the edge represents the length of the corresponding dependence path in the code slice.

the code-centric slice. They correspond to 78 and 2207 source line locations, respectively. The number of user-level source locations is 50 in the memory data slice and 1141 in the code-centric slice, representing a 95.7% reduction. The root cause of this bug is that the buffer length at line 1019 is computed incorrectly at line 1004. Fig. 9 (a) shows how the root cause node is reached in the memory graph, and similar to Fig. 7, the numbers on the edges represent the corresponding dependence chain lengths in the code-centric slice.

5.1.3 Man-1.5

Man is a common utility command for formatting and displaying on-line manual pages on Unix-based systems. The version used in this experiment, *man-1.5*, has a bug that leads *man* to a crash in a C library function. Fig. 10 shows parts of the code from *man-1.5* that illustrate the bug. The buffer overflow happens at line 977.

For this benchmark, the number of nodes visited before reaching the root cause node is 182 in the memory graph, compared with 149 in the code-centric slice. Similarly, if we only consider user-level source locations, the number is 48 in data slice and 54 in code slice, representing a reduction of 11.2%. The root cause of this benchmark is at line 977. Fig. 9 (b) shows how it can be reached.

5.1.4 Other Benchmarks

For *gzip-1.2.4*, the root cause can be reached after visiting 4 nodes, compared to 73 in the code slice. The number of use level source locations is 2 in the data slice and 5 in the code slice. For *ncompress-4.2*, the root cause can be reached after traversing 2 nodes, whereas the number is 1866 in the code-centric slice. The number of user level source locations has been reduced from 9 in code slice to 1 in data slice.

5.2 Efficiency

The second experiment is about the time and space cost of the technique. Table 3 lists the runtime of our technique (column *Slicing*). For comparison purpose, we also list the time of running the programs with the Valgrind engine (column *Valgrind*) to separate the slow-down caused by Valgrind. We also collect the number of dynamic instructions (column *#Instr exec.*) and the total numbers of memory writes and the unique addresses that have been written to (column *#Writes/unique*). From the table, we can see that the average slowdown is about 26.18X.

Figure 13 shows the effectiveness of the optimizations that we described in Section 4.1. It shows runtimes normalized to the base implementation of our slicing algorithm. In the *OPT-TEMP* series, the OPT-TEMP optimization is turned on; In the *BOTH OPT* series, both OPT-TEMP and OPT-CACHE optimizations are turned on. From the table, we can see that on average, with both optimizations, the runtimes have been reduced to 32.8% of those of the base implementation.

To evaluate the scalability of our technique, we ran *ncompress*

			Runtime (s)		
Benchmarks	#Instr exec.	#Writes/unique	Valgrind	Slicing	Slowdown
squid	11,443K	2,713K/614K	6.5s	350.5	53.9X
man	6,374K	1,121K/49K	1.0s	56.4	56.4X
bc	194K	38K/3K	0.5s	4.4	8.8X
gzip	28K	5K/2K	0.4s	2.4	6.0X
ncompress	15K	8K/3K	0.4s	2.3	5.8X

Table 3: Time cost of data-centric slicing algorithm.

```
get_section_list (void) {                                    960

  i = 0;                                                     972
  for (p = colon_sep_section_list; ; p = end+1) {            973
    if ((end = strchr (p, ':')) != NULL)                     974
      *end = '\0';                                           975
                                                             976
    tmp_section_list[i++] = my_strdup (p);                   977
                                                             978
    if (end == NULL||i+1 == sizeof(tmp_section_list))        979
      break;                                                 980
  }                                                          981
```

Figure 10: Parts of man-1.5 code, from the source file *man.c*.

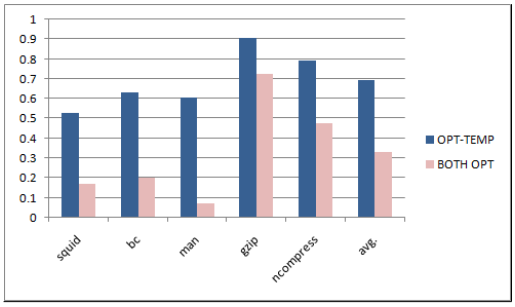


Figure 13: The runtimes with OPT-TEMP turned on, and with both OPT-TEMP and OPT-CACHE turned on, normalized to the base implementation.

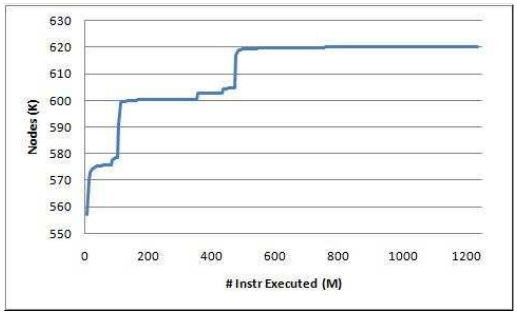


Figure 11: BDD nodes used for memory slicing as *ncompress* process a 30M file.

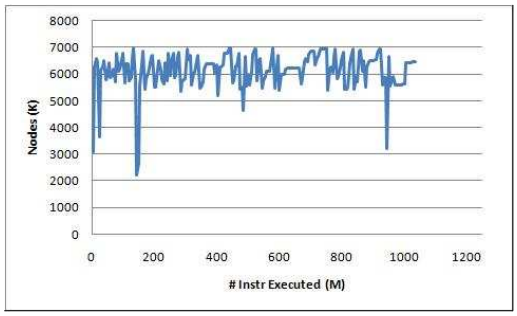


Figure 12: BDD nodes used for memory slicing as *squid* serves 5K requests.

and *squid* with large inputs and observe the memory consumption during the execution. The results are shown in Fig. 11 and Fig. 12. The X axis represents the number of dynamic instructions. The Y axis represents the memory consumption, denoted by the number of BDD nodes created. Note that a BDD node is different from an mG node. All the mGs are indeed represented in BDDs by a unified large binary diagram. The space needed by the BDDs is decided by the number of nodes in the binary diagram and multiple mGs share many common BDD nodes to achieve compression. One node takes 20 bytes memory. From the figures, we observe that the memory consumption becomes stable as the execution proceeds. The deep drops in Fig. 12 correspond to BDD garbage collections.

5.3 Checkpoint

In code-centric slices, a node is created for an instruction instance. Therefore, if a memory address is written multiple times during execution, multiple nodes will be created with each representing a unique write instance. As a consequence, the dependences exercised by various write instances can be represented separately. In contrast, our technique creates one node for each memory address in order to achieve a space complexity bounded by active memory size. One side effect is that the dependences associated with the various write instances are aggregated on the node representing the address. The implication is that if the user is following the breadth first search strategy, one may have to inspect nodes that are distant from the failure point in the corresponding code slice. To mitigate this effect, we propose to periodically checkpoint all live memory cells' slices. Once a memory address's slice is checkpointed, it is reset to empty. Doing so, less dependences are aggregated. When debugging, we first examine the latest slice of the wrong output, if the root cause is not found, we continue to search into the checkpointed slices.

Table 4 shows some results on checkpointing the squid benchmark, with the checkpoint period set to *Never*, *50s*, *10s* and *5s*.

Period	Runtime	Size	All	User
Never	350.5s	49030	245/78	145/50
50s	174.0s	20765	124/45	99/38
10s	179.4s	10840	82/32	68/28
5s	187.8s	2826	34/23	31/20

Table 4: The effect of checkpointing on the slice for the bug in *squid*.

The table shows data of the last slices, which are the memory slices computed on the failure points up to the last check points, i.e. we do not need to chase into the checkpointed slices. The root cause node can always be found in these slices. The *Size* column shows how many edges the last slice contains; As is expected, the smaller the checkpoint period, the smaller the slice size. The *Runtime* column shows how long the benchmark runs including the checkpointing time. The shorter runtime when checkpoint is enabled can be explained by that when the BDDs are smaller (having fewer internal nodes), resulting in BDD operations being faster. The *All* and *User* columns, similar to Table 2, show how many nodes and source line locations are examined before reaching the root cause node, and how many of them are in user-level code. As we can see from the table, the smaller the checkpoint period, the smaller number of source code locations needing to be inspected due to the less aggregation. In practice, choosing the checkpoint period should balance the need to reduce the aggregation effect in our slices and the costs of checkpoint operations.

6. RELATED WORK

Our work is closely related to traditional program slicing [18, 14], especially dynamic slicing [9, 1, 16, 22]. However, existing slicing techniques are code-centric, meaning that they identify dependences between statements or statement executions. The dynamic versions of these techniques have undesirable effects because the space required to store dynamic dependences is proportional to execution length. Existing techniques either resort to building approximate dependence graphs [1] or compression [22, 17]. However, approximate graphs are often too conservative. Compression does not change space complexity. Furthermore, traditional code-centric dynamic slices are hard to inspect by humans because of too many details are presented. The root cause of these limitations is that traditional techniques are code-centric. The proposed data-centric technique in this paper overcomes these limitations. It is also worth mentioning the various techniques proposed in [22] that improved efficiency and effectiveness of dynamic slicing are orthogonal to our technique. There has been work on detecting memory dependence [5]. However, their memory dependences are dependences between load and store instructions. Beszedes et al. proposed an offline algorithm to compute dynamic slices for C programs [3]. By using a D/U representation to represent static must-dependences and a trace to disambiguate pointers, they claim that the space requirement for dynamic slicing is smaller and is bounded by the number of dynamic memory accesses that needs to be disambiguated, instead of the number of executed instructions. No evaluation was presented in that work. In comparison, no trace is used in our online algorithm and our space overhead is bounded by the number of distinct addresses used.

We are not the first one to have observed that data-centric techniques can be more efficient than the corresponding code-centric techniques. In [15], Vaziri et al. observed that atomicity can be more effectively represented as data properties rather than code properties.

BDDs have been used in various static program analysis [2, 19, 8, 21] and dynamic program analysis [23] to deliver efficiency. More particularly, dynamic slices are computed using BDDs in [23]. However, their slices are set of statements and our slices are graphs between memory locations. Their algorithm is offline and ours is online. The number of slices produced by their technique is a function of execution length as it is still a code-centric technique.

7. FUTURE WORK

From the discussion of Section 5.3, we can see that memory graphs are amenable to online parallel construction. With the advancement of multicore platforms, we can exploit idle cores to further reduce the runtime overhead. More specifically, graph construction can be off-loaded from application program execution. Traces of checkpoint intervals can be communicated to separate graph construction processes, which build subgraphs independently.

8. CONCLUSION

We propose a data-centric dynamic slicing technique. Dependences are introduced between memory locations instead of executed instructions in traditional dynamic slicing techniques. The technique is space-efficient as the required space to store a memory dependence graph is no longer bound to the execution length but the memory footprint, which is often well bounded. The technique is very effective in debugging as it excludes redundant information and is able to illustrate causality with much shorter chains than traditional code-centric dynamic slicing. It incurs reasonable overhead.

9. ACKNOWLEDGMENTS

This work is supported by NSF grants CNS-0720516 and CNS-0834529 awarded to Purdue University.

10. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 246–256, 1990.

[2] M. Berndl and O. Lhotak and F. Qian and L. Hendren and N. Umanee. Points-to Analysis using BDDs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 103–114, 2003.

[3] A. Beszedes, T. Gergely, Z. M. Szabo, J. C., and T. Gyimothy. Dynamic slicing method for maintenance of large c programs. In *CSMR '01: Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pages 105–113, Lisbon, Portugal, March 2001.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 677–691, 1986.

[5] G. Z. Chrysos and J. S. Emer. Memory Dependence Prediction using Store Sets. In *ISCA: Proceedings of the 25th Annual International Symposium on Computer Architecture*, 142–153, 1998.

[6] Diablo binary rewriting framework. http://www.elis.ugent.be/diablo/.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *ACM*

Transactions on Programmming Languages and System (TOPLAS), 9(3):319–349, July 1987.

[8] M. Gabel and Z. Su. Symbolic Mining of Temporal Specifications. In *Proceeding of the International Conference on Software Engineering (ICSE)*, 51-60, 2008.

[9] B. Korel and J. Laski. Dynamic program slicing, In *Information Processing Letters*, 29(3), 1988.

[10] J. Lind-Nielsen. BuDDy, a binary decision diagram package. http://buddy.sourceforge.net.

[11] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: a benchmark for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, Chicago, Illinois, 2005.

[12] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 89–100, 2007.

[13] F. Qin and C. Wang and Z. Li and H. Kim and Y. Zhou and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO: Proceedings of the 38th International Symposium on Microarchitecture*, 135-148,2006.

[14] F. Tip. A Survey of Program Slicing Techniques. In *Journal of Program Languages*, 121–189, 1995.

[15] M. Vaziri and F. Tip and J. Dolby. Associating Synchronization Constraints with Data in An Object-Oriented Language. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 334-345, 2006.

[16] T. Wang and A. Roychoudhury. Hierarchical Dynamic Slicing. In *ISSTA: International Symposium on Software Testing and Analysis*, 228-238,2007.

[17] T. Wang and A. Roychoudhury. Using Compressed Bytecode Traces for Slicing Java Programs. In *Proceeding of the International Conference on Software Engineering (ICSE)*, 512–521, 2004.

[18] M. Weiser. Program Slicing. In *Proceeding of the International Conference on Software Engineering (ICSE)*, 439–449, San Diego, CA, 1981.

[19] J. Whaley and D. Avots and M. Carbin and M. S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of Programming Languages and Systems: Third Asian Symposium*, 2005.

[20] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 185–195. ACM, July 2007.

[21] L. Yuan and J. Mai and Z. Su and H. Chen and C.-N. Chuah and P. Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *IEEE Symposium on Security and Privacy*, 199-213, 2006.

[22] X. Zhang. Fault Localization via Precise Dynamic Slicing. *Ph.D Dissertation*, University of Arizona, 2006.

[23] X. Zhang and R. Gupta and Y. Zhang. Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams. In *Proceeding of the International Conference on Software Engineering (ICSE)*, 502-511, 2004.