2010

Memory Indexing and its Use in Automated Debugging

William Summer
Purdue University, wsummer@cs.purdue.edu

Xiangyu Zhang
Purdue University, xyzhang@cs.purdue.edu

Report Number:
10-010

Memory Indexing and its Use in Automated Debugging

William N. Sumner   Xiangyu Zhang
Department of Computer Science, Purdue University
{wsumner,xyzhang}@cs.purdue.edu

Abstract

Execution comparison examines different executions generated by different program versions, different inputs, or by perturbations. It has a wide range of applications in debugging, regression testing, program comprehension, and security. Meaningful comparison demands that executions are aligned before they are compared, otherwise the resulting differences do not reflect semantic differences. Prior work has focused on aligning executions along the control flow dimension. In this paper, we observe that the memory dimension is also critical and propose a novel solution to align memory locations across different executions. We introduce a canonical representation for memory locations and pointer values called memory indexing. Aligned memory locations across runs share the same index. We formally define the semantics of memory indexing and present a cost-effective design. We also show that memory indexing overcomes an important challenge in automated debugging by enabling robust state replacement across runs.

1 Introduction

Comparing executions is a fundamental challenge in dynamic program analysis with a wide range of applications. For instance, comparing executions of two versions of a program with the same input can be used to isolate regression faults [10], and analyze the impact of code changes [17]. It helps identify implementation differences between the two versions, which can be exploited by attackers. Comparing program state at different points within executions can also be used to normalize and cluster execution traces, simplifying analyses that use those traces as input [6]. Execution comparison also provides unique advantages in program deobfuscation [7] and debugging compiler optimizations, where aggressive transformations make static comparison less effective. Two executions from the same concurrent program can be generated with schedule perturbations to confirm harmful data races [19], and real deadlocks [15]. In computing cause transitions for failures [20, 18], a failing execution and a passing execution are compared to isolate instructions or program states relevant to the failure.

Recently, a technique called *structural indexing* [19] was proposed to align the dynamic control flow of executions at the granularity of instruction execution such that comparison can be carried out at aligned places. This substantially improves accuracy in race detection [19], deadlock detection [15], and computing cause transitions for failures [18]. However, structural indexing only solves one dimension of the problem – control flow. The other unsolved dimension, orthogonal to control flow, is memory. In the presence of program differences, input differences, or non-determinism, corresponding heap memory regions are allocated in different places across runs. Therefore, although executions are aligned along control flow paths, if memory regions are not properly aligned, comparing values is hardly meaningful.

Existing techniques rely on sub-optimal solutions [20, 17, 18], such as identifying memory using symbolic names. In particular, to compare memory states of two executions, reference graphs [1] are first constructed in which global and local variables are roots, and memory regions, especially heap regions, are connected by reference edges. Roots align by their symbolic names; other memory regions align by their reference paths, which consist of variable and field names. We call it *symbolic alignment*. However, symbolic alignment is problematic in the presence of aliasing. Detailed discussion can be found in Section 2.

In this paper, we propose a technique called *memory indexing* (MI). The central idea is to canonicalize memory addresses such that each memory location is associated with a canonical value called its *memory index*. Memory locations across multiple executions align according to their indices. Pointers are compared by comparing the indices of their values. Memory indices are maintained along an execution such that they can be directly accessible or computable.

Overall, we make the following contributions.

- We formally present the memory indexing problem. We identify key properties of valid solutions.
- We discuss two semantics for memory indexing. The first one is an online semantics that computes indices on the fly during execution and handles pointer arithmetic. The second is a lazy semantics that computes indices on demand. It has lower cost and is more suitable for languages without pointer arithmetic.
- We introduce a practical design that uses a tree to allow multiple indices to share their common parts. Optimizations remove redundant tree construction and maintenance.
- We illustrate how memory indexing facilitates computing cause transitions for failures. Novel memory comparison and substitution primitives resolve limitations of existing solutions. They allow robust mutation of a passing run to a failing run by copying state across runs.
- We evaluate the proposed MI scheme. It causes a 41% slow down and 213% space overhead on average. The results of two client studies show that MI is able to canonicalize address traces across runs, and it scales cause transition computation to programs with complex heap structures.

2 Motivation

Execution comparison not only requires alignment on the control flow dimension, but also on the memory dimension. Aligning and comparing memory snapshots across runs is thus a key challenge. Existing techniques do not provide satisfactory solutions to the following challenges.

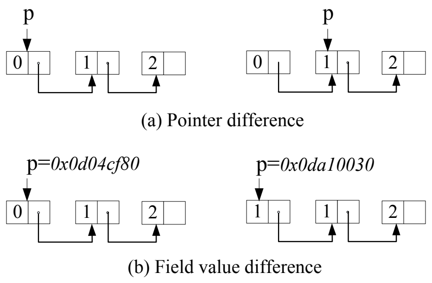


Figure 1: Pointer Comparison. Linked lists represent the snapshots of different executions. Each node has two fields: *val* and *next*.

Support for Pointer Comparison. Many applications require the ability to compare pointers across runs. For example, regression debugging [10] and computing cause transitions [20] rely on contrasting variable values in a passing run and a failing run to identify faulty values. For pointer related failures, it is critical to identify when a pointer contains a faulty value. However, due to semantic differences or non-determinism, even pointers that point to the same (heap) data structure can have different values across runs, and hence they are not directly comparable. Most existing techniques do not support pointer comparison. Instead, non-pointer field values, such as $p \rightarrow val$ and $p \rightarrow next \rightarrow val$ in Figure 1, are compared following their symbolic reference paths. For the case in Figure 1 (b), such comparison yields the right result. That is, only $p \rightarrow val$ has different values across executions. Whereas in case (a), the conclusion is that $p \rightarrow val$ and $p \rightarrow next \rightarrow val$ have different values, implying the definitions to these fields are faulty in a debugging application, which is not true. A more appropriate conclusion is that *only* the pointer p has different values, all other differences are manifestations of the pointer difference.

Destructive State Mutation. Applications such as computing cause transitions [20] compare memory snapshots from a passing run and a failing run. A variable having different values in the two respective runs is called a *difference.* In order to reason about the causal relevance of differences with the failure, values of

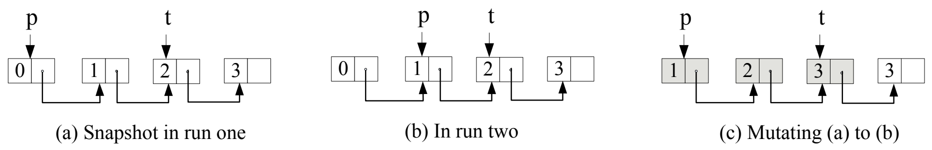


Figure 2: Destructive State Mutation. (a) Snapshot in run one. (b) In run two. (c) Mutating (a) to (b).

difference subsets are copied from the failing run to the passing run to see if the failure is eventually triggered in the mutated passing run. However, using symbolic alignment causes a *destructive mutation* problem in the presence of aliasing. In particular, a memory location may have multiple reference paths. It may be classified as a difference when it is compared under one path but not along another path. Mutation along one path destroys the semantic constraints along other paths and often leads to undesirable effects. Consider the example in Figure 2. Two snapshots are shown in (a) and (b) with p pointing to different locations in each. With symbolic alignment, the root p is aligned first, followed by nodes along paths from p. As a result, the first node in (a) is aligned with the second node in (b), the second node in (a) with the third node in (b), and so on. Comparing the non-pointer fields of the aligned nodes yields the following reference paths denoting differences: $p \rightarrow val$, $p \rightarrow next \rightarrow val$ and $p \rightarrow next \rightarrow next \rightarrow val$. They are fields in (b) having values different than those in (a). Suppose we try to mutate (a) to (b) by copying values from (b) to (a) following the paths of differences. The resulting state is shown in (c). Observe that t's value is undesirably destroyed.

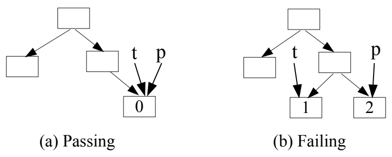


Figure 3: Lost Mutation.

Lost Mutation. If multiple differences alias, they may result in *lost mutation* when they are applied together. Specifically, the differences applied earlier may be overwritten undesirably by differences applied later. This may lead to incorrect conclusions about the relevance of differences. Consider the example in Figure 3. Assume the failure is that $(p \rightarrow val) + (t \rightarrow val)$ has the wrong value. Pointer t points to the wrong place, and the node pointed to by p has the wrong value. These together cause the failure. Symbolic alignment and non-pointer value comparison identifies two differences denoted by their reference paths: $p \rightarrow val$ and $t \rightarrow val$. However, as p and t alias in (a), when the differences are applied to (a) in the order of p difference first and then t difference, the rightmost leaf first has the value 2 and then 1. The mutated state does not lead to the expected failure. Hence, we mistakenly conclude that the two differences are irrelevant to the failure.

3 Problem Statement and Overview

To overcome the aforementioned problems and provide robust support for memory comparison and mutation, we propose a novel technique called *memory indexing*. The basic challenge is to *associate each memory location with a canonical value such that locations across runs are aligned by their canonical values; pointers can be compared by their canonical values.* Such values are also called *memory indices* because they essentially provide an indexing structure for memory.

The idea is illustrated by Figure 4, which revisits the example in Figure 3. Focus on the parts inside the boxes for now. Each node is associated with a canonical value (index) circled at a corner. Nodes are aligned by their indices. Hence, we can see the root nodes align as they have the same index α. The node with index δ on the right does not align with any node on the left. Besides its concrete value, pointer p also has a canonical value π in both runs. Pointer t has π on the left but δ on the right. With memory indexing, the differences of the two states can be correctly identified: pointer t has a different pointer value and the

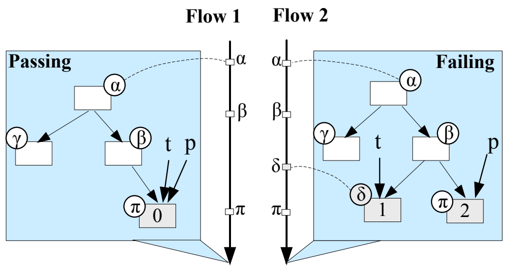


Figure 4: Overview of Memory Indexing.

nodes pointed to by p have different field values. When mutation occurs, t is set to the location with index δ, which is not present in the passing run and thus entails allocation. p's field value is changed to 2. Such mutation properly induces the failure.

A valid memory indexing scheme should have the following property: *at any execution point, each live memory location must have a unique index.* We call this the *uniqueness property.* If this property is not satisfied, multiple locations may share the same index or one location may have multiple indices, which makes proper alignment across runs impossible. Symbolic alignment does not always satisfy this property and is thus not a good indexing scheme.

A good indexing scheme should have the following additional feature: *locations across runs that semantically correspond to each other should share the same index.* We call this the *alignment feature.* Using the concrete address of a memory location is an indexing scheme satisfying the uniqueness property, but it does not deliver good alignment.

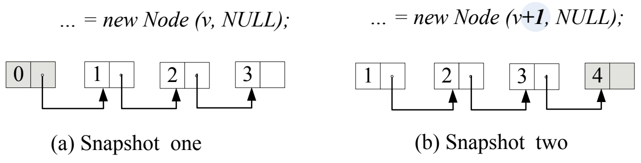


Figure 5: Graph Matching may be Undesirable.

Inadequacy of Graph Matching. Finding the most appropriate memory alignment concerns program semantics and is thus, in general, not a concretely knowable problem. As pointed out in [20], one possible approximate solution in the general case is to represent the memory snapshots under comparison as reference graphs and formulate the alignment problem as a graph matching problem [2]; the goal of which is to produce a match with the minimal number of graph differences. However, this solution is too expensive (NP complexity) to be practical [20]. More important, we observe that it fails to deliver desirable alignment in many cases because it does not capture semantic differences. Consider the example in Figure 5. The failure in (b) occurs because the value field passed to the node constructor is incremented by one. With a graph matching algorithm, to minimize graph differences, the second node in (a) aligns with the first node in (b), the third node in (a) aligns with the the second node in (b), and so on. As highlighted in the figure, the graph differences, namely the graph operations needed to mutate (a) to (b), are: add (b)'s tail to (a); add the edge to the added node; remove the head in (a). However, such differences imply that the shape of the linked list is faulty, which is not true. The most appropriate alignment matches the corresponding nodes in the lists, resulting in four field value differences that precisely reflect the semantic differences.
Our Indexing Scheme. We propose to use *the execution point where the allocation of a memory region occurs as the index of the region.* We leverage the observation that semantic equivalence between executions often manifests itself as control flow equivalence. Hence, semantically equivalent memory regions are often allocated at corresponding execution points. Figure 4 presents an overview. The two lines in the middle represent the control flows. The memory indices are essentially canonical flow representations of the allocation points. For instance, the root nodes share the memory index α, indicating they are allocated at the same point α. In contrast, index δ's presence in only the failing run means that the allocation does not occur in the passing run. Our indexing scheme satisfies the uniqueness property and provides high quality alignment

in practice.

4 Semantics

In this section, we present two semantics for memory indexing. The first is for low level languages such as C. It supports pointer arithmetic by updating indices on the fly. This is called the *online semantics*. The other semantics computes indices on demand and does not need interpretation of pointer arithmetic. We call this the *lazy semantics*.

In our semantics, each memory location is canonically represented by a pair (*region*, *offset*), with *region* as the canonicalized representation of its containing allocated region and *offset* as its offset inside the region. The canonical representation of a region is generated when the region is allocated and serves as a birthmark of the region during its lifetime. We provide a function MI() that maps a concrete address to its index. We also maintain a function PV() that maps a pointer variable to the index of the value stored in the pointer. In the lazy semantics, PV(p) is lazily computed from MI(p), whereas in the online semantics PV(p) is updated on the fly through pointer manipulations on pointer p. Hence, PV(p) may be different than MI(p) in the online semantics. As we will later show, separating PV from MI allows us to precisely handle pointer arithmetic, which is desirable for certain applications.

Rule	Event	Instrumentation
(5)	Prog. starts	for each global variable g: MI(&g)= (nil, `global_offset`(g))
(6)	Enter proc. X	for each local variable lv of X: MI(&lv)= (CS, `local_offset`(lv))

(7)	pc: $p =$ `malloc`(s)	for i=0 to s-1:
		MI($p+i$)= ([SI, pc], i)
		PV(p)= MI(p)

(8)	$p = \&v$	PV(p)= MI(&v)
(9)	$p = q$	PV(p)= PV(q)
(10)	$p = q \pm offset$	PV(p)=(PV(q).first, PV(q).second$\pm$ $offset$)

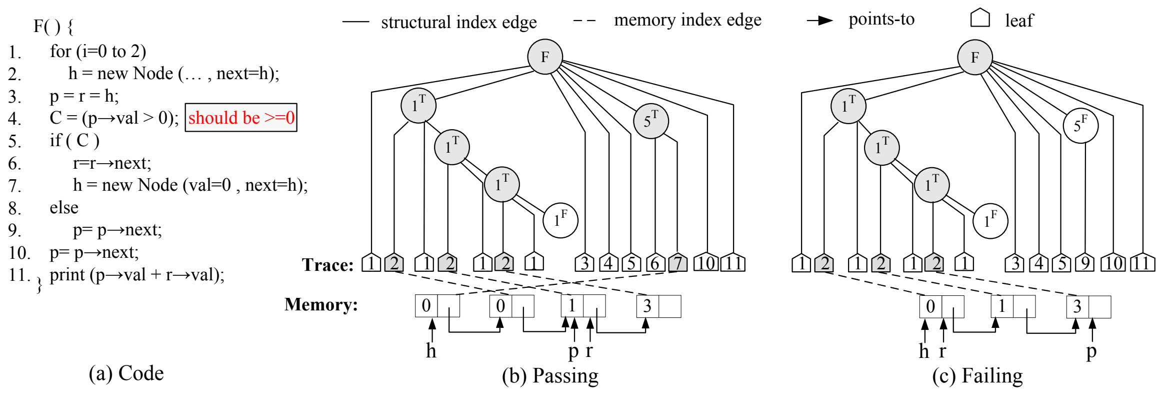
Figure 6: Online semantics. A memory index MI(a) represents the memory index of an address a, which is a pair comprising a region identifier and an offset. CS represents the current call stack. pc represents the program counter. SI represents the current structural index. PV(p) represents the memory index of the address value stored in p.

4.1 Online Semantics

In this subsection, we discuss the online semantics.

Indexing Global Memory. We consider global memory locations as part of a *global region*. Hence, the memory index of a global location is its offset in the global region (Rule 5 of Figure 6). In our terminology, &g denotes the concrete address of a variable g. If executions from different program versions are considered, e.g. in comparing regressing executions, symbolic names of variables are used instead of their offsets. It is easy to see the uniqueness property is satisfied.

Indexing Stack Memory. We consider stack memory to be allocated upon function entry. The allocated region is the stack frame of the function. Hence, we use a stack frame identifier and the stack frame offset of a location to represent its index. Recursive calls allow multiple instances of the same function to exist in the call stack at an execution point so that we have to use the call path of a stack frame as its id. Such stack indices trivially satisfy the uniqueness property and provide meaningful alignment. This is presented in Rule 6 of Figure 6. Some programs perform dynamic allocation on the stack, which makes stack variables have varying offsets. We identify such variables through static analysis and use our own IDs to replace the offsets.



```
F( ) {
1.   for (i=0 to 2)
2.       h = new Node (… , next=h);
3.   p = r = h;
4.   C = (p→val > 0);  should be >=0
5.   if ( C )
6.       r=r→next;
7.       h = new Node (val=0 , next=h);
8.   else
9.       p= p→next;
10.  p= p→next;
11.  print (p→val + r→val);
}
```

Figure 7: Example for heap indexing. The code constructs a linked list of three nodes with values of 0, 1 and 3. Initially, the three pointers h, p, and r all point to the head of list. There is a regression bug at line 4 in computing the predicate. As a result, the failing run takes the false branch, making p point to its second node. Pointer p further advances to the third node at line 10. In contrast, the passing run takes the true branch, eventually resulting in both p and r pointing to its third node. The failure is observably wrong output. The memory snapshots are before the failure at statement 11.

4.1.1 Indexing Heap Memory.

The essence of our technique is to create a birthmark of a memory location as its canonical representation. The birthmarks of heap locations are more tricky. Using the program counter (PC) of the allocation point is not sufficient because multiple live heap regions may be allocated at the same PC. The calling context of the allocation point is not sufficient either. For example, the code in Figure. 7 (a) creates a linked list in the loop on lines 1 and 2. All allocations occur in the same context (statement 2 inside `F()`). Adding an instance count does not help either because different executions may take different paths so that the same count does not imply correspondence. In this paper, we utilize *structural indexing.*

Background: Structural Indexing [19] is a technique that provides a canonical representation for *execution points within control flow* such that points across runs can be aligned. Readers familiar with structural indexing can skip to the ending □ symbol.

Conceptually, an execution is indexed by a tree representing its nesting execution structure. A leaf node denotes an execution point, and internal nodes represent control structures such as branches, loop bodies, and method invocations.

Consider the program in Figure 7 (a). The index trees for the two runs are shown in Figure 7 (b) and (c). The two control flow traces are displayed horizontally from left to right. Individual trace points are also the leaves. Consider the (b) tree. The root represents the whole execution, which consists of six statement executions presented as the leaf children of the root, namely statements 1, 3, 4, 5, 10, and 11. Since the `for` statement has substructure, an internal node 1^T, pronounced as "the true branch of statement 1", is created to represent the loop body. The remainder is constructed similarly. Traces are aligned by the index trees in a top-down fashion. First, the two roots align. Then the six leaf children align, dictated by the aligned parents. *Note that their alignment is independent of the branch outcomes of 1 and 5.* Internal nodes may or may not align, depending on their labels (i.e., the branch outcomes). If they do, recursive alignment is performed. The two trees in Figure 7 (b) and (c) align except for the subtree rooted at 5^T.

The structural index of an execution point is the path leading from the root to the leaf node representing the point. For instance, the index of the shaded 7 in the left tree is $[\mathtt{F}, \mathtt{5}^{\mathtt{T}}, \mathtt{7}]$. Deciding the presence of an execution point in other runs is equivalent to deciding the presence of its index in the corresponding trees. An important property is that *each dynamic point in an execution has a unique index.*

Structural indices are computed using control dependence analysis [8]. Figure 8 defines the semantics of structural indexing. Each internal node in an index tree represents an execution region delimited by a function entry and its exit or by a branch point (including loop predicates) and its immediate postdominator. Regions are either disjoint or nesting, analogous to function invocations. Hence, a stack similar to a call stack, named the *structural indexing stack* (SI), is used to maintain the structural nesting relation. An entry

Rule	Event	Instrumentation
(1)	invoke function F at call cite c	SI.push(F^c)
(2)	Exit proc. F	SI.pop()
(3)	Predicate p takes branch B	SI.push(p^B)
(4)	Statement s	while (s is the immediate post-dominator of SI.top()) SI.pop()

*SI is the structural index represented as a stack.

Figure 8: Semantics for Structural Indexing.

is pushed upon function entries (Rule 1) or predicate executions (Rule 3). The top entry is popped upon the exit of a function (Rule 2) or when the immediate postdominator of the predicate on top is encountered (Rule 4). The SI stack always contains the structural index of the current execution point. More details and examples can be found in [19]. □

To index heap memory, we use the structural index of the allocation point as the id of an allocated region to compose the memory index. The uniqueness property of structural indexing ensures the uniqueness of heap indices. The alignment feature of the memory indexing scheme also originates from the fact that structural indexing identifies equivalent allocation points across executions. In particular, heap indices are set when a region is allocated (Rule 7 in Figure 6). A heap index consists of the current SI and the allocation site *pc*. Besides setting the memory indices, the rule also sets the canonical value of the pointer variable, i.e. PV(p), to the memory index of the head of the region. Such a canonical value will be used in pointer manipulation. For example, in the second iteration of the loop in Figure 7, after the allocation in statement 2, PV(h)=([$\mathtt{F}, \mathtt{1}^{\mathtt{T}}, \mathtt{1}^{\mathtt{T}}, 2$], 0).

Memory locations across multiple runs are aligned by their indices. By this criterion, in Figure 7, the head of the list in (b) does not have alignment while the remaining three nodes align with the list in (c).

A key feature of memory indexing is pointer value comparison across runs. Besides a concrete memory address, a pointer variable is also associated with a canonical value. Canonical pointer values are updated on the fly in the online semantics, as specified by Rules 8-10. For brevity, we assume a simple syntax for pointer operations. In particular, if the address of a variable v is retrieved and assigned to a pointer, the canonical value of the pointer is the memory index of v's address (Rule 8). If a pointer variable is copied to another variable, the canonical value gets copied too (Rule 9). For pointer arithmetic expressions $p = q \pm offset$, variable p's canonical value is computed by copying the region identifier of q and adding r to the offset of q (Rule 10). For brevity, our semantics assumes type information has been processed so that offset variables are identified at the unit of bytes.

4.2 Lazy Semantics

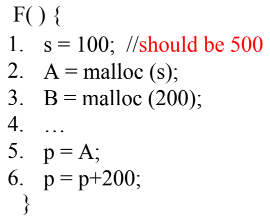
For high level languages in which pointer arithmetic is not permitted, or when client applications do not require considering the effects of pointer arithmetic, we can derive PV values on demand and hence allow a more efficient implementation. The semantics is called the *lazy semantics*. The observation is that canonical values of pointers can be lazily inferred from their concrete values. That is, given a pointer p, PV(p)=MI(p). Recall that in the online semantics, PV is computed by interpreting pointer arithmetic (Rule 10) and hence PV(p) is not necessarily equivalent to MI(p).

Rule	Event	Instrumentation
(11)	*pc*: p = `malloc`(s)	MI(p)=([SI, *pc*], 0)
(12)	Query the index of heap address a	$t = a$ `while` (MI(t)$\equiv$ nil) t=t-1 `return` (MI(t).first, a-t)

Figure 9: Lazy Semantics.

The new rules are presented in Figure 9. On the fly computation is only needed upon heap allocation (Rule 11): the current SI is assigned to the region base address, but not to the other cells in the region. When the MI value of a heap address is queried (Rule 12), the algorithm scans backwards from the given address to find the first address with a non-empty index. The memory index of the given address consists of the region denoted by the non-empty index and the offset inside the region. For large heap regions, we set the MI for a number of addresses at set intervals besides the base address such that the linear scan can quickly encounter a non-empty MI. No on-the-fly computation is needed for global and stack memory. The MI values of global and stack addresses can similarly be computed on demand.

Precision Lost in the Lazy Semantics. In languages with pointer arithmetic, the lazy semantics does not instrument pointer operations or track the original regions of pointers. The looser coupling with program semantics may lead to undesirable imprecision in certain applications.



```
F( ) {
1.   s = 100;  //should be 500
2.   A = malloc (s);
3.   B = malloc (200);
4.   …
5.   p = A;
6.   p = p+200;
  }
```

Figure 10: The Advantage of the Online Semantics.

Consider the example in Figure 10. It is a simplification of a real bug in `bc` version `1.06`. Assume there is a regression error in the program such that variable s should be 500 whereas it is 100 in the faulty version. For simplicity, we also assume the A and B regions are positioned consecutively in memory. In the failing run, buffer A has size 100, and pointer p points to a location in B due to overflow, despite the fact that it originally pointed to A. According to the lazy semantics, at the end of the failing execution, PV(p)= MI(p)=([$\mathtt{F}, 3$], 100), which is offset 100 in the B region. In the passing run, PV(p)= ([$\mathtt{F}, 2$], 200), which is offset 200 in the A region. Thus, pointer p is considered a difference. In contrast, following the online semantics, both the passing and the failing runs have PV(p)= ([$\mathtt{F}, 2$], 200), so p is not considered a difference. Instead, the online semantics only reports variable s to be a difference, which precisely reflects that the program has a faulty allocation size instead of faulty pointer arithmetic.

In practice, one can choose the right semantics based on the application. For instance, the online semantics is more desirable when out-of-bound accesses are involved, such as when debugging a segmentation fault. It also handles dangling pointers better because a PV value has the same lifetime as the pointer regardless of the status of the deallocated memory, whereas in the lazy semantics a dangling pointer is no longer dangling when the memory is re-allocated.

5 Design and Optimizations

Rule	Event	Instrumentation
(15)	pc: $p =$ `malloc`(s)	l=new Leaf(pc, p, s) *Tree_Insert* (SI, l) MI(p)=(l, 0)
(16)	`free` (p)	*Tree_Remove* (MI(p).first)

Figure 11: Tree based Indexing in Lazy Semantics.

The semantics in the previous section are conceptual. They model an index as a sequence of symbols (the region) and an integer (the offset). This is too expensive to operate with in practice. In our design, we explicitly maintain an index tree for heap memory and represent a heap region as a reference to some leaf in the tree. The full index of a heap location can be acquired by traversing bottom-up from the leaf. Rules 15

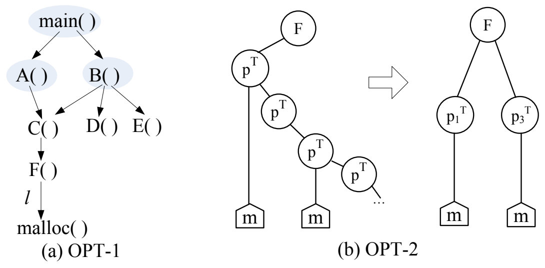


Figure 12: (a) A call graph with only the highlighted functions instrumented. Label l denotes the call site. (b) Heap index trees before and after OPT-2.

and 16 show the tree based instrumentation for the lazy semantics. Upon heap allocation (Rule 15), a leaf node representing the allocation is created and inserted into the tree by calling *Tree_Insert()*. The function first checks if the current SI is part of the tree. If not, it adds the SI to the tree before it inserts the leaf node. At the end, the instrumentation sets the MI of the region base address to the leaf. Upon deallocation (Rule 16), *Tree_Remove()* is called with the leaf node corresponding to the to-be-freed region. Recursive tree elimination is performed, meaning that removing a leaf node may lead to removing its ancestors if they have no children. Shaded subtrees in Figure 7 are example heap index trees. Dotted edges link leaf nodes to memory regions. We have the following optimizations to make our design practical.

OPT-1: Removing Redundant Instrumentation. We have two observations that help remove redundant instrumentation. The first one is that we only need a partial structural tree to index heap allocations. Hence we can *avoid instrumentation that maintains irrelevant structural indices.* Figure 12a shows a sample call graph. Functions `D()` and `E()` do not allocate heap memory, and hence If a function does not allocate heap memory, it is not necessary to compute structural indices inside that functions. More formally, a function or a predicate branch is *relevant* to heap allocation if and only if a heap allocation can directly or transitively occur in its body. Irrelevant functions and predicates are not instrumented with pushes and pops.

The second observation is that we do not need to instrument all relevant functions or predicate branches. More specifically, given a relevant function other than `main` (predicate) n, if all index paths from any of n's parents to a heap allocation inside n's body have to go through n, we don't need to instrument n. We call n a *dominant* function (predicate). Intuitively, we don't need to instrument if we can infer the presence of n on an index path given the allocation site and the parent node. In Figure 12a, function `C()` does not need to be instrumented although it is relevant, because there is only one index path from its parent `A()` or `B()` to the `malloc()` function. In contrast, `A()` and `B()` need to be instrumented. With the optimized instrumentation, the possible heap indices are $[\texttt{main}, \texttt{A}/\texttt{B}, l]$. We have developed static analyses to identify relevant but not dominant functions and predicates. They are analyses on call graphs and control dependence graphs. Details are elided. Note that such optimizations are not applicable to general structural indexing because they leverage heap allocation information.

OPT-2: Handling Loop Nodes. From the semantics, loops require pushing multiple entries of the same predicate, which can be optimized as follows. As in [19, 15], we add a counter field to the stack entry; then upon encountering a loop predicate, it is first checked if the top entry is the same predicate. If so, the counter is incremented instead of pushing. When the current stack is materialized to the tree due to heap allocation, a new node is inserted to the tree if there is not an existing node with the same counter value. Consider Figure 12b. On the left, a sample index tree is shown with multiple instances of the loop predicate `p`. With the optimized instrumentation, only two nodes are generated to denote the first and the third instances of the loop predicate, in which allocations occur. The optimization does not affect the uniqueness and alignment properties of indexing.

The space consumption is dominated by the tree, whose size depends on its shape and the number of live heap regions. A pessimistic bound is O(*maximum tree depth* × *maximum live heap regions*). In theory, the tree depth is unbounded because it is tied with loop counts and the depth of recursion. In practice, because we are only interested in the partial tree for allocations and we optimize loop predicates using counters, tree depth is often well bounded such that the space overhead is feasible (see Section 7).

6 Robust Memory Comparison and Replacement

Cause transition computation [5, 20] produces a causal explanation for a software failure. The technique takes two executions: one failing and the other passing that closely resembles the failing. The passing run can be generated by selecting an input similar to the failing input. The overall idea is to compare memory snapshots of the two runs at selected execution points. A reference graph [1] is constructed to represent a snapshot. Memory comparison is reduced to graph comparison driven by symbolic reference paths. Causality testing is conducted to isolate a minimal subset of graph differences relevant to the failure. More specifically, subsets of graph differences are enumerated through the delta debugging algorithm. A subset is considered relevant if *replacing the program state specified by the subset in the passing run with the corresponding values in the failing run produces the failure.* The minimal subsets computed at the selected execution points are chained together to form an explanation. In [18], the technique is improved by automatically aligning two execution traces using structural indexing before comparison. However, in both [5] and [18], memory comparison and replacement is driven by symbolic paths, and hence has the issues mentioned in Section 2.

Consider the example in Figure 7. Using symbolic alignment and comparing only non-pointer values, if only heap memory is considered, the set of differences (failing - passing) Δ=$\{p \rightarrow \mathit{val}$, $r \rightarrow \mathit{val}$, $r \rightarrow \mathit{next} \rightarrow \mathit{val}$, $r \rightarrow \mathit{next} \rightarrow \mathit{next} \rightarrow \mathit{val}$, $h \rightarrow \mathit{next} \rightarrow \mathit{val}$, $h \rightarrow \mathit{next} \rightarrow \mathit{next} \rightarrow \mathit{val}\}$. None of the subsets, including the Δ set itself, can induce the same failure. For instance, applying subset $\{p \rightarrow \mathit{val}$, $r \rightarrow \mathit{val}$ $\}$ does not work due to the lost mutation problem. As a result, the delta debugging algorithm terminates without finding the minimal failure inducing subset. Since aliasing is very common in general programs, we need to perform robust memory comparison and replacement.

With MI, we are able to develop two robust primitives: comparison of memory snapshots *Mem_Comp()* and application of a memory difference *Diff_Apply()*, i.e., copy a value from one memory snapshot to the other (across runs).

For the comparison primitive, snapshots are first aligned by their indices and then comparison is conducted on aligned locations. Memory locations with non-pointer types are compared by their concrete values. Locations with pointer types are compared by their canonical values. Differences are presented as a set of indices, denoting that the corresponding locations are different.

Consider the two snapshots in Figure 7. Global variables C, h, p, and r are aligned. Since C is of boolean type, its values are compared and classified as differences. In contrast, canonical value comparison is conducted for pointer variables h, p and r. It is easy to see that they are different. Heap memory is compared by the index trees. The region pointed to by h in (b) is identified as the only tree difference. Hence, if we compute the difference set (passing - failing), the result is {(nil, offset(C)), (nil, offset(h)), (nil, offset(p)), (nil, offset(r)), ($[\mathtt{F}, 5^{\mathtt{T}}, 7]$, *)}. The symbol '*' in the last index signifies that the entire region is different. It is much smaller than the symbolic results.

The second primitive is the application of a unit difference[1] represented as an index, from which the corresponding concrete memory location in both snapshots can be identified. The value is copied from the source snapshot to the target snapshot. If the value is a pointer, we cannot simply copy the concrete address. Instead, we identify the proper concrete address in the target snapshot following the canonical value of the pointer. If the region is not present in the target snapshot, it is first allocated.

Function *Diff_Apply()* in Algorithm 1 describes how to apply a heap unit difference. In the algorithm, the source and target heaps are indexed by trees rooted at T' and T, respectively. Variable δ represents the unit difference. Lines 3 and 4 identify the heap region denoted by δ in T' and T. In lines 5 and 6, the concrete addresses are computed. At line 7, the algorithm tests if the computed address is a pointer (the superscript specifies where the dereference occurs). If not, the algorithm copies the value (line 8). If so, it tests if the region pointed-to is present in T (line 11). If not, it copies the region (line 12). Finally at line 14, the concrete address stored to the pointer is set to a location in the region (in T) aligned with the source region (in T').

Function *Region_Copy()* copies a region denoted by the parameter *path*$'$ from T' to T. It first locates the region in T' (line 2) and allocates a region of the same size in T (line 3). The *path*$'$ is inserted to T and a leaf node is created to represent the allocated region (lines 4-5). This avoids allocating the same region again. Finally, individual fields are copied from T' to T by calling *Diff_Apply()* (lines 6-7). Note, it may transitively copy more regions from T' to T.

[1]A unit difference is a difference regarding a specific memory location instead of a region.

Algorithm 1 Apply a heap difference.

Description: Copy the value in location δ from T' to T. Leaf node is of the type (*pc*, *base*, *size*).

```
1: Diff_Apply ($T$, $T'$, $\delta$):
2:    let $\delta$ be ($path$, $offset$)
3:    let (-, $base'$, -) be the leaf node in $T'$ along $path$
4:    let (-, $base$, -) be the leaf node in $T$ along $path$
5:    $a \leftarrow base + offset$
6:    $a' \leftarrow base' + offset$
7:    if ($*(a')^{T'}$ is NOT a pointer) then:
8:       $*(a)^{T} \leftarrow *(a')^{T'}$
9:    else
10:      let $\mathrm{PV}(a')^{T'}$ be ($p'$, $f'$)
11:      if ($T$ does not have path $p'$) then:
12:         Region_Copy ($T$, $T'$, $p'$)
13:      let (-, $b$, -) be the leaf node in $T$ following $p'$
14:      $*(a)^{T} \leftarrow b + f'$
```

Description: copy region *path*$'$ in T' to T.

```
1: Region_Copy ($T$, $T'$, $path'$):
2:    let (-, $base'$, $size'$) be the leaf in $T'$ along $path'$
3:    $r \leftarrow$ allocate($size'$) in the run denoted by $T$
4:    insert $path'$ to $T$
5:    set the leaf node following $path'$ in $T$ to (-, $r$, $size'$)
6:    for ($i$=0 to $size' - 1$) do:
7:       Diff_Apply($T$, $T'$, ($path'$,$i$))
```

Applying stack and global differences is similarly defined.

Example. Consider the example in Figure 7. Assume we want to apply the differences of p and r to the passing run. Observe that p points to the third node in the failing run and $\mathrm{PV}(p)^{fail}$= $([\mathtt{F}, \mathtt{1}^{\mathtt{T}}, \mathtt{1}^{\mathtt{T}}, \mathtt{1}^{\mathtt{T}}, \mathtt{2}], 0)$. During the p difference application, following the path, the concrete address of the fourth node *in the passing* run is identified and assigned to p. Similarly, after applying the r difference, r holds the concrete address of the second node in the passing run. Note that, by applying these two differences, the same failure can be produced. Applying other differences, such as C, at this point (before statement 11) has no impact on the failure. The minimal failure inducing difference subset including p and r is emitted as one cause transition.

The same memory comparison and difference minimization is further performed at aligned instructions 10 and 5; it stops at 4 as no state difference is identified. The chain of cause transitions is: C has the incorrect value false at 5, then p and r point to the wrong places at 10 and 11, and finally the failure. These transitions compose a failure explanation.

Next, we define the composability property and show that it holds for the proposed primitive.

Definition 1 *A scheme for memory differencing and replacement is composable iff given a set of unit differences* Δ=$\{\delta_1, \delta_2, ..., \delta_n\}$ *and the universal set* $\mathcal{U}$ *of all differences, after applying the differences in* Δ *from* T' *to* T*, the differences between* T' *and the mutated* T *is* $\mathcal{U} - \Delta$.

Composability is very important for cause transition computation, it ensures that the delta debugging algorithm is able to make progress, because it mandates that the effect of applying a set of differences must subsume the effect of applying a subset of the differences [20]. If a replacement scheme is not composable, applying the universal set of differences may even fail to convert T to T'. The symbolic path based scheme is not necessarily composable. As shown in Figure 7, applying the two differences of $p \rightarrow val$ and $r \rightarrow val$ from the failing run to the passing run results in a state in which $p \rightarrow val$ still manifests itself as a difference.

Property 1 *The proposed MI based memory differencing and replacement primitive is composable.*

The proof is elided due to space limits. However from Algorithm 1, we observe that for non-pointers, the primitive faithfully copies values; hence the property is trivially true. For pointers, the primitive either

Table 1: Instrumentation and allocation.

program	instmt. func	instmt. branch	# of alloc stat/ dyn.	avg. alloc size	tree dep.
164.gzip	11 (12%)	17 (1.8%)	5 / 436k	28 KB	130
175.vpr	100 (37%)	202 (7.5%)	3 / 107k	481 B	59
176.gcc	1282 (57%)	17774 (24.9%)	236 / 10.2m	5 KB	700
181.mcf	5 (19%)	6 (2%)	4 / 3	33 MB	8
186.crafty	8 (7.3%)	158 (2.9%)	12 / 37	23 KB	6
197.parser	2 (0.6%)	41(1.3%)	1 / 1	31 MB	292
254.gap	596 (70%)	4109 (19%)	2 / 2	100 MB	10
255.vortex	672 (73%)	3400 (19%)	9 / 258k	399 B	365
256.bzip2	8 (11%)	7 (1.1%)	10 / 36	16 MB	51
300.twolf	59 (31%)	218 (3.5%)	3 / 574k	31 B	28

allocates a region when it is not present in the index tree or simply assigns the address if the region is present. Such behavior does not lead to additional differences that were not present in the original difference set or mask any other existing differences.

7 Evaluation

The implementation consists of both semantics and two client studies. It is based on the CIL infrastructure and has 3500 lines OCaml, 3500 lines C and 3000 lines Python.

7.1 Efficiency

The first experiment focuses on cost. The evaluation is on SPECint 2000 benchmarks. We excluded `252.eon` and `253.perlbmk` because they were not compatible with CIL. All experiments were executed on an Intel Core 2 2.1GHz machine with 2 GB RAM and running Ubuntu 9.04.

Table 1 shows the instrumentation needed and characteristics of allocations. All executions are acquired on reference inputs. The second column shows the number of instrumented functions (after optimizations) and their percentage over all functions. The third column shows the same data for predicates. The fourth column shows the numbers of static allocation sites and dynamic allocations. The fifth column shows the average size of each allocation. The last column shows the maximum depth of the memory index tree. We observe that some programs make a lot of allocations with various sizes (`gcc` and `twolf`) and some make very few but large allocations (`mcf` and `bzip2`). They have different impacts on the performance. Programs `parser` and `gap` allocate a memory pool at the beginning and then rely on their own memory management systems. Our current system does not trace into memory pool management. We leave it for future work. Observe, the maximum tree depth is not high with respect to the structural complexity of programs. Recall that we collapse consecutive instances of a loop predicate, so the depths are largely decoupled from loop counts.

The overhead can be seen in Figure 13, in which `Full` represents implementation without removing redundant instrumentation; `Part` removing redundant instrumentation; `Flow` the online semantics; and `Lazy` the lazy semantics. The figure presents the performance overhead for a number of combinations. In practice, `Part+Lazy` is desirable for most applications, as illustrated by later client studies. `Space` represents the space overhead for `Part+Lazy`. All data is normalized against original runs without instrumentation.

We observe first that `Full+Lazy` has substantially more runtime overhead than `Part+Lazy`. `Part+Flow` is slightly more expensive than `Part+Lazy` due to instrumentation on pointer operations. The overhead of `Part+Lazy` is low (41%). Next, observe that in half the benchmarks, there is little space overhead. This is because the number of allocations and the tree depth are relatively low regarding the size of each allocation. In contrast, `300.twolf` had the most overhead. It performs a large number of very small allocations, ¡32

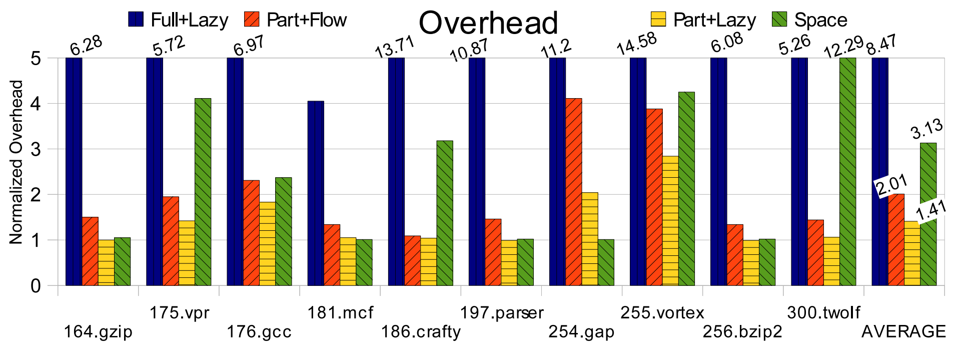


Figure 13: Normalized runtime and space overheads of memory indexing with and without optimizations.

bytes on average, so on average maintaining the index for each allocation is more costly[2]. Nonetheless, the average space overhead is 213% (111% without `twolf`). The conclusion is that the cost of MI is feasible for many applications.

7.2 Trace Canonicalization

Trace canonicalization is the alignment of control flow and memory accesses across traces from two executions. It plays a part in debugging and regression analyses [17, 10, 7], among others. With MI, an important question can be answered, *given two address entries in two respective traces, should they be considered differences?* Note, two accesses at the corresponding points in the two traces do not mean that they operate on the same data; the accessed addresses being different does not mean they do not semantically correspond.

The study is on three common, open source programs, `make`, `gawk`, and `dot`. We reported the number of address differences before and after MI canonicalization. We turned off all memory layout randomization. To avoid comparing trace entries that do not correspond, we used structural indexing to establish which memory accesses occurred at the same point in both traces and only compare those accesses.

The traces were generated from the programs' provided test suites or, in the case of `dot`, the provided examples in the documentation. Each full trace was compared with traces generated by a fixed percentage of the input, i.e., removing part of the inputs. The results are shown in Fig. 14. For each percentage of input similarity, we present the percentage of matched stack and heap memory accesses before and after canonicalization. For MI, these are '`MI locals`' and '`MI allocs`' respectively, while for the addresses without canonicalization, they are '`Addr allocs`' and '`Addr locals`'. We furthermore present the percentage of control flow correspondence ('`Control Flow`').

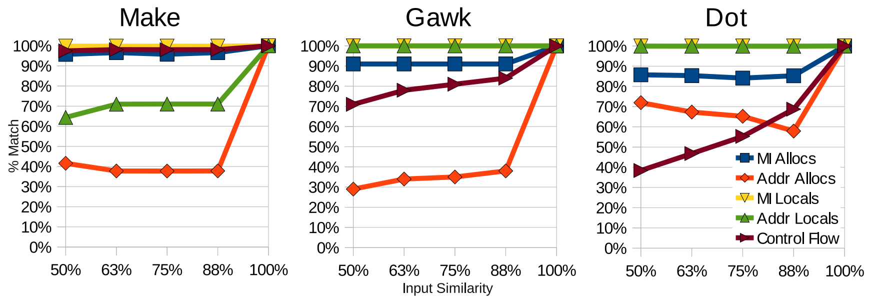


Figure 14: Percent of corresponding memory accesses w. and w/o MI.

Observe first that MI provides a substantially higher level of heap access correspondence (50% more for `make` and 50-60% more for `gawk`, and 15-30% more for `dot`). Less benefit was observed in `dot` because `dot`'s dynamic allocations are largely on fixed buffers that do not change according to inputs. MI was able to find

[2]In our implementation, we use 22 bytes for each tree node.

more corresponding addresses for local variables too. Observe that stack local allocation and variable sized objects on the stack make it more difficult to find correspondences without MI (e.g. the `make` case).

The control flow similarity increases as the input similarity increases. Note that the address correspondence without canonicalization stays roughly the same or even decreases as the control flow similarity increases (like in `dot`). The decrease happened because greater control flow similarity allows more (different) addresses to be compared. In contrast, the correspondence found by MI is roughly consistent.

7.3 Cause Transition Computation

This experiment evaluates the impact of MI on computing cause transitions. The algorithm in [18] was implemented as a platform on which we tested two versions of the memory comparison and replacement primitive: one is symbolic path based, used in [5, 18]; the other is the new MI-based. The study is on several real bugs in open source programs, including `gcc`, `make`, and `gawk`, that have non-trivial heap behavior and aliasing. The failing runs are generated according to the bug reports. The passing runs are acquired from the correct inputs in the reports if provided; previous non-regressing versions; or using an automated patching technique [21]. Note that acquiring passing runs is an orthogonal problem out of our scope. Other patching techniques such as [12] can also be used.

Results are summarized in Table 2. The `Program` column contains the buggy programs. `Bug ID` presents the bug id, through which one can identify the report online, or the publication date on the mailing list. `Bug` describes each bug. `Passing Run` shows the sources of the passing runs: inputs provided in reports (`correct input`), non-regressing versions (`non-regressing`), and dynamic patching (`predicate switch`). The maximum number of differences (present in failing and absent in passing) found using symbolic differencing is presented in `Sym Diffs`, and the maximum when using MI is in `MI Diffs`. In `Sym Diffs`, we report one memory cell only once although it may be a difference along multiple paths. Of further interest is the number of differences with aliases (in column `Aliases`), or multiple symbolic paths. They can cause issues as discussed in Section 2. Column `Issue` presents the exhibited problems when using the symbolic path based primitive. We also present the number of transitions and the average number of differences included in each transition in `Trans/Diffs`, along with time required (in seconds) in `Time` when using the MI version.

Table 2: Cause Transition Computation for Failures.

Program	Bug ID	Bug	Passing	Sym. Diffs	MI Diffs	Aliases	Issue	Trans/Diffs	Time (s)
gcc 2.95.2	529	-Wshadow warns on functions	predicate switch	8365	233	8105	>12h	8/1	4559
gcc 2.95.2	776	Large array size causes abort	predicate switch	10101	230	9027	>12h	2/1	379
gcc 2.95.2	2771	-O1 breaks strength-reduce	provided input	11095	284	10254	>12h	4/1	1797
make 3.81	16958	.PHONY targets are unrecognized	non-regressing	2699	184	33	>12h	9/1	740
make 3.81	18435	Parentheses break make targets	provided input	3301	336	356	>12h	29/2	645
make 3.81	19133	./ prevents self remake	provided input	3728	550	187	>12h	5/2	235
make 3.80	112	Rules cannot handle colons	provided input	3309	645	217	>12h	11/6	685
gawk 3.1.5	1/20/06	Deallocate bad pointer	provided input	630	22	509	early term.	8/1	56

Observe that in every case, the number of symbolic differences is substantially, 2-50 times, larger than the number of differences when using MI, because the proper memory correspondence cannot be found. Furthermore, the `Aliases` column shows that substantial aliasing is common, creating a lot of difficulties for the symbolic method. As seen in the `Issue` column, in most cases, symbolic path based computation would not terminate within 12 hours. The main reason is that lost mutation and destructive mutation caused (see Section 2) by aliasing prevent the relevant difference subset from being computed, so the algorithm ends up searching subsets of the already significant difference sets. For example, in the first `gcc` case, it may be possible that all the enumerated subsets of the 8365 differences need to be tested. In `gawk`, the algorithm simply terminated early, unable to produce relevant transitions for the failure.
A Case Study on Detailed Comparison. We performed a separate test focusing on `make` bug 18435 from Table 2. We selected 10 sample points at 10% intervals along the part of the passing run that is beyond the first divergence of the two runs. At each sample point, we compared the memory snapshots across the two runs and then mutated the memory in the passing to that in the failing by applying the universal set of differences, alternatively using the symbolic path based primitive [5, 18] and the MI based primitive. Then we collected the trace after the mutation and compared it to that of the failing run. Trace comparison

is conducted using control flow canonicalization and MI based memory canonicalization. According to the discussion in Section 6, the traces should be identical if the primitives are composable.

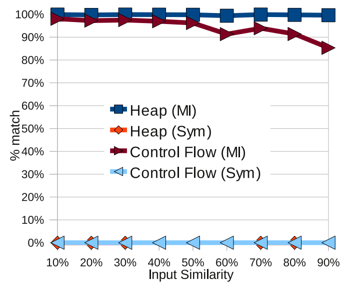


Figure 15: Heap accesses and control flow trace similarity after state mutation in the execution of `make`.

The results are shown in Figure 15. '`Heap (MI)`' and '`Control Flow (MI)`' represent the similarities of heap access and control flow traces using the MI based primitive, and '`Heap (Sym)`' and '`Control Flow (Sym)`' represent those using the symbolic primitive. Observe, the access similarity when using MI is consistently near 100%, and the control flow similarity is consistently above 90% until the end. This means the mutation is mostly successful in turning the passing run to the failing run. The similarity is not 100% because we currently do not model external state such as file IDs, process IDs, etc. Thus, such states are not eligible for meaningful comparison and replacement. In contrast, when the symbolic primitive is used, the execution quickly diverges from the expected control flow, having near 0% similarity, and it has near 0% similarity for accessing the heap. In fact, the mutated run often quickly crashes due to destructive mutation (Section 2). This supports that the MI primitive is composable, but the symbolic primitive is not.

A Sample Chain of Cause Transitions. For the bugs considered, using the MI primitive not only captures the root causes as described in the reports but also the precise cause transitions. Let's use one case to illustrate the results. `Make` is a tool for executing a set of rules that specify actions for different 'target's or commands along with dependencies on other targets whose actions must be performed first. With bug 18435 from Table 2, target dependencies will not always be resolved, and thus the target will be considered invalid.

A simplification of the code involved is presented in Fig. 16 on the left and the reported relevant states at transitions are on the right. The high level problem is that targets with parentheses in their names are incorrectly considered archives, thus the rules for them cannot be correctly executed. We used one such input from the bug report where the target had parentheses and another input without parentheses, also from the report.

In lines 7-14, the `pattern_search` function discovers the rules and dependencies for a target by scanning through all rules and determining if they apply. For each file in the dependency list, `make` checks to see if the file exists. In this process, observe that on lines 1-3, `ar_name` returns `True` if a file name has matching parentheses and false otherwise. Thus the dependency is considered an archive. This is returned to line 4 in `file_exists_p`, which overrides normal behavior and returns `False` even if a file of the given name exists. As a result, line 10 of `pattern_search` is incorrectly false, and the required dependency of the rule is never added to the list of target dependencies. Further, on lines 13-14, the commands for the rule are not added to the target, effectively making the rule unresolved. This error then propagates to line 15 in `update_file_1`, which takes the false branch in the incorrect execution, not setting `must_make`. Thus, line 17 branches false, leading to the failure. That is, desired output is not observed as line 18 is not executed.

The computed transitions are presented on the right with the reported memory differences. Note that the precise root cause is captured, where `ar_name` incorrectly returns `True`. The remainder of the transitions precisely explain the propagation. Note that some of the steps are simplified for the sake of presentation, as opposed to the 29 steps in Table 2. The average minimal set at each transition has size 2, indicating this is a very thin line of propagation. The symbolic approach failed to make progress in 12 hours.

In summary, MI allows the strength of cause transition computation to be fully realized, reflected by our success of scaling to programs like `gcc` with full automation. Note, although a `gcc` case was presented in [20]. It was conducted with human intervention. The later automated system [5] works well for small

```
ar_name(name):
1   if (pos = findchar(name, '(')):                  condition = True
2       return findchar(pos, ')')                    return value = True
3   return False

file_exists_p(file):
4   if (ar_name (name)):                             condition = True
5       return False                                 return value = False
6   return {True iff file exists}

pattern_search(file):
7   file->deps = file->cmds = NULL
8   for each rule:
9       for each dep in rule->deps:
10          if (file_exists_p(dep)):                 condition = False
11              file->deps = add_dep(dep, file_deps) (skipped)
12      if (file->deps):                             file->deps = NULL, condition = False
13          file->cmds = rule->cmds                  (skipped)
14          break

update_file_1:
15  if (file->deps != NULL):                         file->deps = NULL, condition=False
16      must_make = True                             (skipped)
17  if (must_make):                                  must_make = False, condition=False
18      {execute file->cmds}
                                                     ERROR
```

Figure 16: The cause transitions for a `make` bug

programs without much aliasing.

8 Related Work

Trace normalization [6] divides traces into segments. Segments with the same starting and ending state are considered equivalent. Client applications using such traces only need to look at a consistent representative segment from each equivalence class. The outcome is reduced workload and increased precision. Memory indexing is complementary in that it provides a robust way of comparing program state across executions and hence helps identify equivalent trace segments.

There has been recent work on comparing executions for debugging regression faults [10], analyzing impact of code changes [17], and finding matching statements across program versions [7]. These techniques are able to construct a symbolic mapping of variables across program versions through profiling, such as pointer x in version one being renamed to y in version two. The constructed mapping is static. In comparison, we focus on comparing dynamic (address) values of corresponding variables, answering questions like "does x point to the corresponding address in the two executions". Furthermore, trace canonicalization facilitated by MI would improve the precision of these analyses.

Many debugging techniques [14, 16, 3, 13, 4, 11] compute fault candidates by looking at a large number of executions, both passing and failing. In these techniques, execution profiles are collected and analyzed statistically. Some debugging techniques compare a simple profile of a failure with a small number of correct runs (usually one) [9]. They use control flow paths and code coverage. MI is complementary to these techniques by providing a way to canonicalize profiles before they are analyzed to achieve better precision, especially for pointer related bugs. We have demonstrated in this paper that MI is able to drive cause transition computation that is highly sensitive to memory alignment.

Compared to the recent advances on generating causal explanations of failures [13, 4], The proposed robust, fine-grained memory differencing and substitution primitives make it feasible to extract succinct and in-depth information about failures. For instance, it is relatively easier for us to reason about whether a value at a given execution point is relevant to a failure. Furthermore, MI improves cause transition computation [5, 18] by allowing alignment along the memory dimension, which substantially improves robustness in the presence of aliasing.

9 Conclusions

We present a novel challenge in dynamic program analysis: aligning memory locations across executions. We propose a solution called memory indexing (MI), which provides a canonical representation for memory addresses such that memory locations across runs can be aligned by their indices. Pointer values can be compared across runs by their indices. The index of a memory region is derived from the canonical control flow representation of its dynamic allocation site such that control flow correspondence is projected to memory correspondence. Enabled by MI, we also propose a novel memory substitution primitive that allows robustly copying states across runs. We evaluate the efficiency of two memory indexing semantics. Our results show that the technique has 41% runtime overhead and 213% space overhead on average. We evaluate effectiveness through two client studies: one is trace canonicalization and the other is cause transition computation on failures. The studies show that MI reduces address trace differences by 15-60%. It also scales cause transition computation to programs with complex heap structures.

References

[1] H. Boehm. Space efficient conservative garbage collection. In *PLDI'93*.

[2] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *CACM*, 16(9), 1973.

[3] Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04*.

[4] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. *ICSE'09*.

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE'05*.

[6] M. Diep, S. Elbaum, and M. Dwyer. Trace normalization. In *ISSRE'08*.

[7] M. Feng and R. Gupta. Detecting virus mutations via dynamic matching. In *ICSM'09*.

[8] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[9] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *CC'06*.

[10] K. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *PLDI'09*.

[11] H.Y. Hsu, J. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *ASE'08*.

[12] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA'08*.

[13] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE '07*.

[14] J. Jones and M.J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE'05*.

[15] P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI'09*.

[16] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *PLDI'03*.

[17] M.K. Ramanathan, A. Grama, and S. Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *ASE'06*.

[18] W.N. Sumner and X. Zhang. Algorithms for automatically computing the causal paths of failures. In *FASE*, 2009.

[19] B. Xin, N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI'08*.

[20] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE'02*.

[21] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE'06*.