UNIVERSITY OF CALIFORNIA
RIVERSIDE

User Assisted Data Structure Debugging and Verification

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Vineet Singh

December 2016

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Iulian Neamtiu
Dr. Zhijia Zhao
Dr. Vassilis Tsotras

The Dissertation of Vineet Singh is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This dissertation would not have been possible without all people who have supported and inspired me during my Ph.D. study and my life.

I owe my deepest gratitude to **Dr. Rajiv Gupta** for his continuous support of my research and study. His never ending enthusiasm for research, hard working nature and insightful vision have influenced me right from the beginning. He has given my research a direction and a push at every step. Throughout these 5 years, he has been supportive of me in every situation, academic or otherwise. *Thank you Prof Gupta!*

I would also like to express my gratitude to **Dr. Iulian Neamtiu**. His technical and editorial advice has been invaluable to my research. He has taught me innumerable lessons and insights on the workings of academic research in general. I have always felt comfortable in approaching him with silliest of my doubts. *Thank you Prof Neamtiu!*

I would like to thank my other dissertation committee members, Dr. Vassilis Tsotras and Dr. Zhijia Zhao for taking their time to review this dissertation.

I gratefully acknowledge the funding received towards my PhD from National Science Foundation grants CCF-0963996, CCF-1149632, CCF-1318103 and CCF-1524852.

I was fortunate enough to do my internship at Intel under the mentorship of Dr. Harish Patil. I would like to sincerely thank him for making my internship a valuable research experience.

I am thankful to all my lab-mates. You have been a family to me during my time at UCR. Thank you Amlan Kusum, Sai Charan Koduru, Yan Wang, Keval Vora, Zach Benavides, Tanzirul Azim, Farzad Khorasani, Bo Zhou and YongJian Hu for helping me in

many ways during these years.

I would also like to thank all my teachers I have had throughout my life. I would specially like to thank Dr. Uday P. Khedker and Dr. Amitabha Sanyal for introducing me to the amazing field of compilers.

Last but not least, I would like to thank my family who supported me throughout this endeavor. *Mummy, Papa, Aai, Baba, Aaji* and *Aajoba*, thanks for all your prayers and blessings. In particular, I wish to thank my wife, *Prerna*, for being my best buddy and later my life partner.

To my Parents for making me who I am.

To my wife, *Prerna*, for being my source of motivation.

ABSTRACT OF THE DISSERTATION

User Assisted Data Structure Debugging and Verification

by

Vineet Singh

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2016
Dr. Rajiv Gupta, Chairperson

Data structures are critical to the correct functioning of most programs. Corruption of runtime data structures, e.g., due to program faults, can lead to a program crash or even worse, wrong output. Such bugs are hard to analyze using traditional debugging techniques. Specification-based debugging techniques have been used for addressing such bugs, albeit with limited effectiveness. Concurrent data structures make correctness verification even harder; for example, linearizability, the standard correctness criterion for concurrent data structure implementations, is notoriously hard to prove. Consequently, current verification techniques can only prove linearizability for certain classes of data structures. Therefore, there is a pressing need for data structure debugging and verification — the centerpiece of this dissertation.

First, this dissertation presents a precise fault location framework for debugging sequential programs. The framework combines specification-based runtime data structure verification with automatic detection of faulty program statements that corrupted the data structures. The framework consists of a data structure constraint specification language, a compact memory graph representation *MG++*, and an efficient fault location module. Experiments show the precision of our technique: while Tarantula [50] statistical debugging technique narrows the fault to 10 statements, our technique narrows it to about 4 statements. Experiments studying the time and space efficiency for real-world programs show that MG++ is space-efficient and the time overhead for MG++ construction is acceptable for debugging purposes. The dissertation also presents an efficient dynamic backward slicing algorithm to assist the user with further debugging.

Second, this dissertation introduces a *generic, sound*, and *practical* technique to statically check the linearizability of concurrent data structure implementations. Our technique requires specifying the concurrent operations as a list of sub-operations and passing this specification on to an automated checker that automatically verifies linearizability using relationships between individual sub-operations. We have proven the soundness of our technique. Our approach is highly expressive – we have successfully verified the linearizability of 12 popular concurrent data structure implementations, including algorithms that are considered to be challenging to prove linearizable such as elimination back-off stack, lazy linked list, and time-stamped stack. Our checker is efficient, as it verified these specifications in less than a second.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data structures are critical to the correct functioning of most programs. Program faults often lead to data structure corruption. The compromised structural integrity of runtime data structures is hard to detect unless it manifests as a program failure. Traditional debugging techniques have limited applicabiliy in detecting such faults. Specification-based debugging techniques are useful in detecting such faults but are limited to fault detection. The manual localization of such faults is a difficult – tedious and time consuming.

In the case of concurrent data structures, this problem is even worse, as parallelism complicates design, implementation, and verification. First, concurrent implementations of abstract data structures (stacks, queues, sets, etc.) are becoming more and more complex as implementations that increase the degree of concurrency are identified; this complexity in turn is making correctness verification harder. Second, myriad thread interactions at runtime seriously hinder make program understanding and debugging. To address these issues, researchers have proposed correctness proofs for concurrent data structures. Linearizability, introduced by Herlihy and Wing [47], is the standard form of correctness for concurrent data structure implementations. Even the recent state-of-the-art techniques (e.g., [63,116]) for proving linearizability lack generality as they are limited to specific classes of concurrent data structures — so far no technique (manual or automatic) for proving linearizability has been proposed that is both sound and generic. Therefore, researchers often rely on custom proofs of linearizability which is error-prone and highly time-consuming.

To address the above problems, this dissertation presents a fault location framework for sequential data structure implementations and a linearizability verification technique for concurrent data structure implementations. Our techniques harness the power of user specifications and provide the user with the following capabilities:

1. **Easy to Use Input Specification Language.** The fault location framework takes data structure consistency constrains as input. To enable the user to input the data structure consistency constraint with ease, a specification language is provided. The user expresses the constraints in terms of relationship between allocated memory regions and memory stores. The specification language enables the user to express a wide range of data structure constraints with ease.

2. **Precise Fault Location.** The fault location framework uses the input constraint specification to detect and locate faults that violate data structure constraints. The input constraints enable the framework to increase the precision of fault location.

3. **Runtime Efficiency.** The dynamic analysis required for precise fault location is costly, both in terms of time and memory. An efficient representation for storing data structure evolution history has been introduced to limit our fault location framework'

memory usage. We have also introduced a number of optimizations (incremental constraint matching, modification prediction) to make fault location faster.

4. **Sound and Generic Linearizablity Verification Technique.** The linearizability verification technique presented in the dissertation is generic, i.e., it does not rely on specific properties of concurrent data structure implementations and can be applied to a wide range of implementations. We have proven the soundness of our technique.

The remainder of the chapter provides an overview of the data structure **fault location framework** and the **linearizability verification technique** and also presents the organization of the remainder of the thesis.

1.1 Locating Data Structure Faults in Sequential Programs

Heap-allocated data structures can be easily modeled using Memory Graphs: heap allocations and pointers between heap elements correspond to nodes and edges. The structural definition of a data structure can then be expressed via *consistency constraints* that describe the allowed relationships among the nodes of the memory graph. The *data structure errors* that violate these constraints can be detected and located by evaluating the constraints. As an example, consider the widely-used memory allocator in the GNU C library (glibc) that maintains a doubly-linked list to track free memory chunks. The structural *consistency constraints* of this list are often violated by bugs in client programs (e.g., heap overflow bugs) leading to a program crash. Examples of bugs in popular software that do exactly the above include: Kate (KDE bug #124496), Kdelibs (KDE#116176), Kooka

(KDE#111609), Open office (Open office#77015), GStreamer (GNOME#343652), Doxygen (GNOME#625051), Rhythmbox (GNOME#636322), Evolution (GNOME#338994, GNOME#579145) [3, 5, 7]. Previous work in debugging data structures has been directed towards automatic repair of data structures (Malik et al. [71], Juzi [35], Demsky et al. [28, 29]) or were limited to locating the data structure fault to a region of program execution (Gopinath et al. [42], Demsky et al. [27]).

Motivated by the above observations, this dissertation presents a system that (1) allows the user to specify the consistency constraints for dynamic data structures as relationship rules among the nodes of a memory graph, (2) automatically detects any violation of these rules at runtime, and (3) locates the faulty code. The design of this system addresses the following challenges:

1. The system includes an *expressive* and *concise* language to specify constraints.

2. The system supports a novel representation to store data structure state after each mutation, i.e., the *data structure evolution history.* Also, mapping from dynamic (runtime) data structures to source code is maintained.

3. Data structure invariants are routinely broken, albeit temporarily, during operations on the data structure. For example, the structural invariant for a doubly-linked list, *if element **e** points to element **e'** then **e'** should point back to **e***, is violated temporarily during the insertion of a new node in the list. Due to the presence of temporary constraint violations, our fault location system is designed to differentiate between a constraint violation caused by a fault and a temporary legitimate violation. Moreover, the fault location system delivers ease of use and runtime efficiency.

Our fault location framework provides support for specification-based fault location via two main constructs. First, a simple yet effective specification language for writing consistency constraints for data structures. Second, a directive for specifying *C-points*, i.e., program points where our system will check at runtime whether the constraints are satisfied; at such points, the data structure is supposed to be in a consistent state with respect to the provided specification. *C-points* are akin to transactions hence emerge naturally, e.g., at the beginning and end of functions that modify the data structure. *C-points* allow us to detect data structure corruption early, before it gets a chance to turn into further state corruption or crash. In addition, if the program crashes then the crash point is used as a *C-point*. Our technique can work even with a single *C-point*, while more *C-points* imply greater precision.

The fault location framework employs a user-provided specification of the data structure constraints to analyze a buggy execution. As the program executes, our system traces the evolution history of the data structures. The constraints are matched over the program data structure states at *C-points*. Once a constraint violation is detected, our approach identifies the corrupted data structures and the set of inconsistencies. Then, it traces back through the evolution history, searching for program points where inconsistencies were introduced and collecting a list of faulty program statements.

This dissertation presents a unified memory graph representation (MG++). Given a MG++ at an execution point, the memory graph at any prior execution point can be extracted using the timestamps associated with nodes and edges present in the MG++ representation. The unified representation is built by incrementally incorporating changes as the program executes. This makes the representation space-efficient and allows constraint checks to be performed incrementally.

Our proposed fault location technique narrows down the faults to responsible data structure mutating statements. We further advocate the use of Dynamic Backward Slicing [56] for investigating the program fault using the fault data structure mutation. The dynamic backward slice of a computed value is defined to include the executed statements that played a role in the computation of the value. Dynamic backward slice is computed by taking the transitive closure over data and control dependences starting from the computed value and going backwards over the execution trace. For our technique, the resulting faulty data structure mutations from the previous step form the slicing criterion. Dynamic Backward slicing is a classic debugging technique but it still faces the challenge of huge time overhead. This dissertation presents an improved dynamic backward slicing algorithm.

1.2 Verifying Linearizability for Concurrent Data Structures

Linearizability, introduced by Herlihy and Wing [47], is the standard form of correctness for concurrent data structure implementations. Linearizability means that the entire observable effect of each operation on a concurrent data structure happens instantly, i.e., the effect of each operation is atomic.

A concurrent data structure *implementation* consists of a shared state (defined by shared variables) and methods which operate on the shared state. An *execution* consists of a variable number of threads, each executing one of the defined methods. An *operation* is a successful execution of a method. The definition of linearizability given by Herlihy and

Wing in [47] says that, for a linearizable concurrent data structure implementation, each concurrent execution must be equivalent to some sequential execution of operations of the abstract data structure while preserving the order of non-overlapping operation.

This dissertation presents a sound linearizability verification technique which can be applied to a wide range of concurrent data structure implementations. In this technique the concurrent execution of operations can be modeled as interleaved sequences of corresponding atomic sub-operations. The novelty of our approach is that:

1. We can express the set of all sequences allowed by an implementation in terms of static relationships between pairs of individual sub-operations.

2. Given the properties of sub-operation pairs, we can statically verify if all the sequences of sub-operations allowed by the implementation can be mapped to an equivalent non-interleaved sequence of sub-operations while maintaining the order of non-overlapping operations, i.e., linearizability.

Our technique consists of (1) a specification language that allows concurrent data operations to be specified simply as sequences of atomic sub-operations and (2) a static checker that, given the relationship between the sub-operations, determines if the implementation is linearizable. If the linearizability proof fails, the static checker returns a sequence of sub-operations that could not be linearized.

We have applied our technique to 13 popular concurrent data structure implementations and were able to verify 12 of them. As our approach is sound, the inability to verify the remaining data structure represents a false positive. The evaluation shows that our technique is generic, practical, and efficient.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents the language to support data structure constraint specification. Next, Chapter 3 presents, our new memory graph representation and memory graph construction technique. The complete process of fault location for scalar data structure is covered in Chapter 4 and Chapter 5. Chapter 6 presents our linearizability verification technique for concurrent data structure implementations. Chapter 7 surveys existing literature in related areas and Chapter 8 concludes the thesis with a summary of our work and presents a brief future outlook.

Chapter 2

Constraint Specification Language

Program data structures follow consistency constraints, i.e., data structures have structural properties that must hold during execution. Even the simplest of data structure have certain consistency constraints based on the implementations.

We show the importance of data structue consistency constraints using *Mozilla BugID 588187*. Due to this bug, Mozilla crashes while traversing a linked list. The linked list in the program has a simple constraint: each entry→next field must point to another linked list node or NULL. Figure 2.1 shows an excerpt of the traversal code. If entry→next contains a value which is neither NULL nor a heap address then the execution will enter the loop (line 1) and crash at line 2.

```
1   while (entry) {
2       if (entry→Key == key) { <−−− crash here
3           return entry→Data;
4       }
5       entry = entry→Next;
6   }
```

Figure 2.1: Linked list traversal.

This chapter introduces the language support for specifying the data structure consistency constraints. Before introducing our constraint specification language, it is important to mention the following apparent alternatives and explain why we have not used them for our system:

Archie [27] uses constraint-based specification for data structure repair. The specification contains a model the data structure must satisfy. When the model is violated, Archie repairs the data structure to satisfy the model. In our system, the model of the constraints is already fixed in terms of the memory graph. Specifying the model again puts significant extra burden on the programmer.

Alloy [48] is a rich object modeling language for expressing high-level design properties. In comparison, our language is centered around logical, arithmetic, layout and graph constraints at the data structure level.

Programming languages can be used to specify constraints (e.g., repOK [67]). The constraints written need to be checked (hence executed) along with the program and are not useful to us as we need to match constraints during the trace back. Writing constraints in programming languages is verbose and error-prone.

Our language is based on the same principles as the aforementioned ones but is designed specifically for the purpose of specifying data structure constraints for debugging — it is a *domain-specific language*, in other words. Taking on this specific problem makes our language simpler. In our approach, the relevant program state at an execution point is captured by the *memory graph*, and *consistency constraints* for a data structure are specified in terms of relationships among nodes and edges of the memory graph. We provide the user with a C-like syntax so the language is easy to use with minimum learning requirement. In this section we first define our constraint specification language and demonstrate that our language is both expressive and simple to use. That is, we can handle a variety data structures with equal or less burden (in comparison to other languages) on the programmer using our specification language.

2.1 Constraint Model and Language Syntax

The *memory graph*, at each point in the execution, consists of nodes corresponding to allocated memory regions and the edges are formed by pointers between the allocated memory regions. Each node (representing an allocation) has fields corresponding to the fields of the data structure for which the memory was allocated. The structure of the memory graph corresponds to the shape of the data structure; hence violations in data structure constraints can be detected by evaluating those constraints for the memory graph.

Execution Trace	**Memory graph at execution point x**

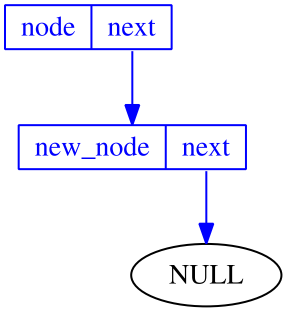
11.node = malloc(**sizeof**(snode));
12.node−>next = NULL;
13.new_node=malloc(**sizeof**(snode));
14.new_node−>next = NULL;
15.node−>next = new_node;
∗∗∗ Execution Point x
	

Figure 2.2: Memory graph representation of runtime data structures.

Figure 2.2 gives the memory graph for a sample execution. The first column shows a sample execution trace. The program allocates memory for structure of type *snode* in lines 11 and 13. The memory graph shows two nodes corresponding to the allocated memory chunks. The *next* pointer is written to point to *new_node* in line 15 which is represented by an edge from *node* to *new_node* in the memory graph.

Our specification language is designed to provide an easy way to express the *structural* form of the memory graph for a data structure. In other words, the programmer simply expresses how the data structure can be visualized in the memory, which makes specification writing very intuitive. Specifying data structure constraints involves three steps. The first step is specifying the types of nodes in the memory graph. The second (optional) step is specifying any special node attributes which may be involved in the constraints. The third step is to specify the constraint using variables of declared types. The grammar of our specification language consists of corresponding three components: *structure*, *model*, and *constraints* (Figure 2.3).

2.2 Structure Specification

The nodes in a memory graph can correspond to an *array* or a *structure*. Structures are defined in terms of the number of *fields* and *edges* present in the memory graph nodes. The structure specification declares the types of the memory graph nodes present in the specifications. The specification of the *quad tree* in Figure 4.2 shows that each node has 9 fields and 5 edges.

Example 1. The structure specifications of *B-tree* and *AVL-tree* are:
– struct btree{ int count; int key[2]; struct btree * child[3];}

btree FIELD 6 EDGE 3;

– struct avltree{ int val; struct avltree * right, * left;}

avltree FIELD 3 EDGE 2;

Table 2.1: *Standard* Attributes.

Attribute	Type	Represents
n.INDEGREE	INT	Indegree of the node n
n.OUTDEGREE	INT	Outdegree of the node n
n.EXTERNAL	BOOL	(n.OUTDEGREE == 0) ∨ (n.INDEGREE ==0)
n.INTERNAL	BOOL	(n.INDEGREE != 0) ∧ (n.OUTDEGREE != 0)
n.ISROOT	BOOL	(n.INDEGREE == 0)
n.ISLEAF	BOOL	(n.OUTDEGREE == 0)

2.3 Attributes and Model Specification

We provide several node attributes (shorthands) that simplify the task of writing the specifications and making them concise. Table 2.1 contains the list of provided node attributes along with their meaning. Standard attributes are valid for any type declared in the structure specification.

When the standard attributes are not adequate, user-defined node attributes are introduced via the *model* part of the specification language in Figure 2.3(b). User-defined node attributes are specific to a node type. Specifying a custom node attribute involves declaring the name (h) of the node attribute along with the node type it is associated with (f). The declaration is followed by the rules for assigning the attribute value for each node. Assignment rules (r) consist of *guard* (g), *terminal assignment* (a), and *non-terminal assignment*(a). A *guard* is a precondition that, when true, leads to *terminal assignment* otherwise *non-terminal assignment* is followed. Assignment statement is assignment of an arithmetic expression to the node attribute. Note that the user-defined attributes can only be used for acyclic data structures. Therefore, when such attributes are used, our implementation performs an acyclicity check because bugs may lead to formation of cycles in data structures that are supposed to be acyclic. The model specification allows the user to create node attributes corresponding to real world node properties and constraints can be specified in terms of these node properties.

Example 2. The height of an *AVL-tree* node is specified as:

```
avltree.HEIGHT; avltree X;
X.ISLEAF == true ⇒ X.HEIGHT = 0 ||
X.HEIGHT = (X[2].HEIGHT≥X[3].HEIGHT) ?
X[2].HEIGHT+1 : X[3].HEIGHT+1;
```

Similarly, for a red black tree node with structure

– struct rbtree{ int color; struct avltree * right, * left;}

the black height derived from the right child is specified as:

```
rbtree.BHEIGHT; rbtree X;
X.ISLEAF == true ⇒ X.BHEIGHT = 0 ||
X.BHEIGHT = (X[2].(1)!= BLACK) ?
X[2].BHEIGHT : X[2].BHEIGHT+1;
```

2.4 Constraint Specification

Our language allows the user to write both inter-node and intra-node constraints. Inter-node constraints refer to constraints on relationship among memory graph nodes. This

Types	t	::=	f FIELD n EDGE n ;
		$\mid$	ARRAY f ;

(a) Structure Specification

User defined node attribute	m	::=	$name\ r$
	$name$	::=	$f\ .\ h$;
Rules	r	::=	$d\ r \mid g$ => $a \mid\mid a$;
Assigment	a	::=	$x.h\ =\ ve$
	ve	::=	$ve + ve \mid ve - ve$
		$\mid$	$ve * ve \mid ve/ve$
		$\mid$	$\lvert ve \rvert \mid (ve)$
		$\mid$	$(ae)?ve : ve$
		$\mid$	av

(b) Model Specification

Intra-node constraint	c'	::=	$d\ q\ ,\ b$;
	q	::=	for I in n to n
	b	::=	$x[e]\ op\ e$
	e	::=	$e + e \mid e - e \mid e * e$
		$\mid$	$e/e \mid (e) \mid \lvert e \rvert$
		$\mid$	$x[e] \mid$ I $\mid n$
Inter-node constrant	c	::=	$d\ c \mid d\ g$ => g ; $\mid d\ g$;
Constraint expression	g	::=	g and $g \mid g$ or g
		$\mid$	$ae \mid be \mid ce$
	ae	::=	$av\ op\ xe$
	xe	::=	$xe + xe \mid xe - xe$
		$\mid$	$xe * xe \mid xe/xe$
		$\mid$	$(xe) \mid \lvert xe \rvert$
		$\mid$	av
	be	::=	bv == true
		$\mid$	bv == false
Edge	ce	::=	$v \rightarrow x$
		$\mid$	$v \not\rightarrow x$
Path		$\mid$	$v \twoheadrightarrow x$
		$\mid$	$v \not\twoheadrightarrow x$

(c) Constraint Specification

Variable declaration	d	::=	$f\ x$;
Boolean Value	bv	::=	v.EXTERNAL
		$\mid$	v.INTERNAL
		$\mid$	v.ISLEAF
		$\mid$	v.ISROOT
Arthmatic Value	av	::=	v.INDEGREE
		$\mid$	v.OUTDEGREE
		$\mid$	$v[n] \mid v.h \mid n$
Vertex	v	::=	$x \mid (x[n])$
Rel. Ops.	op	::=	$== \mid \neq \mid \leq \mid \geq \mid < \mid >$
String	f, h		
Integer	n		
Variable	x		

Figure 2.3: Specification language

kind of constraints are common in pointer based data structures. Intra-node constaints refer to relationships between fields of a memory graph node. Array-based data structures have intra-node consistency constraints where the complete data structure is a big memory graph node (a single memory chunk).

2.4.1 Inter-node Constraints

Inter-node constraints defined via the grammar in Figure 2.3(c) are composed of *declarations*(d), *guard* (optional), and *body*(g). A *guard* is a precondition that must be true in order for the constraint to be applicable. The *body* is composed of one or more constraint statements joined by the boolean operator AND. Three types of constraint statements are allowed: boolean(be), arithmetic(ae), and connection(ce). Connection statements indicate the following: $X \rightarrow Y$ (edge allowed), $X \not\rightarrow Y$ (edge not allowed), $X \twoheadrightarrow Y$ (path allowed), and $X \not\twoheadrightarrow Y$ (path not allowed).

Let us consider constraint specifications for the *B-tree* data structure, shown in Figure 2.4 (top). The first two constraints ensure that the structure represents a tree and are thus the same for all trees (including *Quad-tree* shown earlier). Constraint 1 uses a guard (X.ISROOT == FALSE) to identify non-root nodes and indicate that their indegree must be 1. Constraint 2 uses a guard to indicate that if there is a path from X to Y then there is no path from Y to X. The additional constraints in the specification of *B-tree* follow. Constraint 3 and 4 restrict the outdegrees of internal nodes in *B-tree* while constraint 5 ensures that the number of children is 1+ number of stored keys (value stored in the first field of the *B-tree* structure).

Example 3. The balanced height constraint for *AVL-tree* is expressed using the user-declared node attribute *HEIGHT* below.

avltree X;

(X[2]).HEIGHT - (X[3]).HEIGHT $\leq$ 1 *and*

(X[2]).HEIGHT - (X[3]).HEIGHT $\geq$ -1;

The above examples illustrate that our language is powerful enough to express the constraints embodied by commonly used data structures and at the same time it is intuitive for the programmer to use. While we have shown only non-nested data structures, nested structures can be handled by flattening of structure fields.

btree FIELD 6 EDGE 3;

– *Data structure is a tree*

1. btree X; btree Y;
X.ISROOT == FALSE $\Rightarrow$ X.INDEGREE == 1;

2. btree X; btree Y;
X $\twoheadrightarrow$ Y $\Rightarrow$ Y $\not\twoheadrightarrow$ X;

– *Internal nodes have 2 or 3 children*

3. btree X; btree Y;
X $\rightarrow$ Y $\Rightarrow$ 2 $\leq$ X.OUTDEGREE;

4. btree X; btree Y;
X $\rightarrow$ Y $\Rightarrow$ X.OUTDEGREE $\leq$ 3;

– *Number of children is 1 + number of stored keys*

5. btree X;
X.OUTDEGREE == X.[1] + 1;

Figure 2.4: Specification for *2-3/B-Tree*

2.4.2 Intra-node Constraints

Intra-node constraint specifications are aimed at handling array-based implementations of data structures. Our language supports expressing relationships between array elements. *Intra-node constraints*, defined via the grammar given in Figure 2.3(c) are composed of *declarations*(d), *range*(q), and *body*(b). A *range* gives the *min* to *max* values of node field index (I) on which the constraint will be applicable. The body of the constraint is a relational expression in terms of the value of the field in question.

Example 4. Consider the shard graph representation [57], implemented as an array, storing 8 entries (each entry has source node, source value, edge value, and destination node). The constraint that the source nodes should be ordered is represented as:

ARRAY shard;

shard X;

for I in 0 to 8, X[I*4 +1] < X[(I+1)*4 +1];

Note that we do not specify the types of structure fields in our specification language. The user can specify constraint to check the type of a field specific to the implementation. The following is an example of type checking for a linked list.

Example 5. Consider a linked list implementation using the following structure

– struct node{ int value; struct node * next;}

The next example is the specification for checking the type-safety of the *next* field.

node FIELD 2 EDGE 1;

node X; node Y;

X[2] $\neq$ NULL$\Rightarrow$ (X[2])==Y;

The constraint statement states that if the *next* field of node X is not equal to NULL, if must point to another node Y.

```
Structure Specification
Node* root;
structure Node {
int count;
int key[2];
Node * child[3];
}
Set Nodes(Node);
child0 : Nodes → Nodes;
child1 : Nodes→ Nodes;
child2 : Nodes→ Nodes;
count : Nodes→ int;

Model Specification
[forall node in Nodes ], !(node.child[0]=0) ⇒ node.child[0] in Nodes;
[forall node in Nodes ], !(node.child[1]=0) ⇒ node.child[1] in Nodes;
[forall node in Nodes ], !(node.child[2]=0) ⇒ node.child[2] in Nodes;
[forall node in Nodes ], !(node.child[0]=0) ⇒ <node, node.child[0]> in child0;
[forall node in Nodes ], !(node.child[1]=0) ⇒ <node, node.child[1]> in child1;
[forall node in Nodes ], !(node.child[2]=0) ⇒ <node, node.child[2]> in child2;

Constraint Specification
– Data structure is a tree
[], sizeof(Nodes) >= literal(1);
[forall node in Nodes], sizeof(node. child0) <= 1;
[forall node in Nodes], sizeof(node. child1) <= 1;
[forall node in Nodes], sizeof(node. child2) <= 1;
[forall node in Nodes],
sizeof(child0.node) + sizeof(child1.node)+ sizeof(child2.node) <= 1;
[], sizeof(child0.root) + sizeof(child1.root) + sizeof(child2.root) = 0;

– Internal nodes have 2 or 3 children
[forall node in Nodes],
2<= sizeof(node.child0) + sizeof(node.child1)+ sizeof(node.child2) and sizeof(node.child0) +
sizeof(node.child1)+ sizeof(node.child2) <=3;

– Number of children is 1 + number of stored keys
[forall node in Nodes],
sizeof(node.child0) + sizeof(node.child1)+ sizeof(node.child2) = node.count +1;
```

Figure 2.5: Archie [27] specification for *2-3/B-Tree*.

2.5 Comparison with Archie [27], An Example

Let us compare our specification language with that of Archie [27]. Figure 2.5, we illustrate the use of Archie for specifying a *B-tree* (2-3 tree). As shown, in Archie the developer has to first define a model for the data structure (*Structure Specification and Model Specification* in Figure 2.4). The constraints are defined in terms of this model.

Specification for *B-tree* (2-3 tree) in our specification language is presented in Figure 2.4. In our specification the constraints are defined directly in terms of the relationship of memory graph nodes which is more intuitive and thus the specification is concise and intuitive.

Table 2.2: Specification size comparison.

Data Structure	Number of Statements	
	Ours	**Archie [27]**
Circular Linked List	3	7
Doubly-linked List	2	8
Binary Tree	3	13
Binary Heap	4	14
B-tree	6	21
Quad Tree	4	23
AVL Tree	6	–
Red-Black Tree	9	–
Leftist Heap	5	–
Full K-ary Tree	4	4*K+7

2.6 Evaluation: Size of Specification

We compared specification in our language with that written in Archie [27] for several data structures and summarize the results, i.e., the number of statements required to express various data structures, in Table 2.2. The table shows the compact and expressive nature of our specification in comparison to Archie. The '–' rows indicate that Archie is not able to express three data structures. One of the data structures that Archie cannot specify is the AVL tree for which our specification was shown earlier. Archie cannot handle *global* *constraints*, e.g., that the difference in heights between the left subtree and right subtree of an AVL tree node should not be more than 1. Our specification allows the user to specify global constraints via user-defined node attributes.

2.7 Summary

This chapter presented the language support for fault location framework for scalar data strcutures. The language offers a simple and concise way of expressing data structure consistency constraints in terms of relationship among memory graph nodes and edges. The language provides support for both inter-node and intra-node constraints. Standard node attributes are supported in the language to make specification concise while the user can define custom node attributes to express complicated data structure consistency constraints.

Chapter 3

MG++: Memory Graph Construction and Representation

A memory graph, where nodes represent allocated memory chunks and edges represent links between them created by memory stores, is effective in visualizing the shapes of heap-allocated data structures constructed at runtime. Memory graphs are useful in program understanding [74], or identifying data structures used by a program to replace them with more efficient ones [54]. In programs with bugs, execution of faulty code often results in anomalies that can be observed in the memory graph. Thus memory graphs are useful for helping locate memory bugs (e.g., memory leaks and illegal memory access patterns [14, 80, 88]) as well as in general-purpose debugging [112]. However, prior representations [54, 74, 87] fail to capture important information and their construction algorithms make assumptions that limit their utility:

- *Lack evolution history.* Existing representations [54, 74, 87] are a snapshot of the heap at a program point but do not capture the runtime evolution of the memory graph. This deprives the user of critical information useful in verifying data structure properties and understanding how anomalies were introduced in the memory graph [111].

- *Lack mapping to source code.* Memory graphs used in prior work do not capture the program statements whose execution constructs and modifies the memory graph. This makes it hard for the user to relate memory graph anomalies to faulty source code statements.
- *Lack memory allocator history.* Since existing memory graphs do not capture the behavior of memory allocators, they are not effective when understanding program's (faulty) behavior requires examining the internal actions of the memory allocators (e.g., updates to the internally-maintained free list). This limitation is particularly problematic when programs use custom memory allocators.
- *Allocator information requirement.* Existing methods for constructing memory graphs [54, 74,87] must know what functions allocate/free memory—information from these functions (e.g., starting address and size of allocated memory chunk) is required during graph construction. The allocator-based approaches can only be applied when allocator information is available.

Keeping our focus on dynamic data structures, to overcome all of the above shortcomings, this chapter presents *MG++*, a new representation of heap memory graphs, and a novel approach to construct them. In addition to information traditionally captured by memory graphs, MG++ also captures the runtime evolution history of data structures and its mapping to the source code (Section 3.1.1). Intuitively, MG++ compactly represents the memory graph at the end of the execution, as well as the graph's evolution history; from this history, the memory graph at any earlier program execution point can be extracted. MG++ also captures the internal actions of the memory allocator (Section 3.1.2). This is useful in debugging programs that internally manage the storage or where understanding program behavior requires examining the interaction between program actions and memory allocator actions. We provide examples of real bugs where this information is critical for understanding faulty behavior. We also found the additional information available in MG++ representation useful for manually analyzing program data structures when coupled with Graphviz [37] to visualize the memory graph.

Our novel technique for MG++ construction is based on binary instrumentation and captures memory allocator behavior without requiring knowledge of the allocator function. The technique is based on the key observation that each field within an allocated chunk of memory is accessed via an address computed as an offset from the starting address of the allocated chunk. This enables us to construct the memory graph without assuming that the allocator functions will supply us with the starting address and size information for each newly-allocated chunk. Rather, we are able to construct the memory graph by simply monitoring heap references and operations involving them. Runtime information is analyzed to construct the graph by grouping heap references together to form nodes and using stores in memory to create edges between graphs nodes (Section 3.2.2).

We have implemented our memory graph construction technique using the PIN dynamic binary instrumentation framework [68] for Linux executables running on the IA-32 architecture. We have evaluated the efficiency and effectiveness of our techniques on various real-world programs; we now highlight the results. The space required for storing the complete memory graph evolution history of a large real-world program (the CPython interpreter) using the MG++ representation is less than 150 MB; using prior memory graph representations would require about 100 GB (capturing snapshots after each memory graph change). For the benchmarks evaluated, our MG++ construction approach manages to keep the average slowdown of the execution time for instrumented code to 1.7x in comparison to an allocator-based approach while the worst case slowdown is less than 5x. This shows that our approach provides a practical method for constructing memory graphs in scenarios where allocator information is not available. We illustrate the benefits of our representation in locating faults in GNOME and Mozilla and detecting heap buffer overflows using the RIPE test suite [110].

3.1 MG++ Representation

We first present the MG++ representation that captures the evolution of heap data structures as well as the mapping to relevant source code. Next we present the additions to MG++ which capture the behavior of the memory allocator functions as well as splitting and merging of allocated memory chunks. Finally, we show how the memory graph at any program execution point can be extracted from MG++.

3.1.1 MG++ for Heap Data Structures

A straightforward approach for tracking the evolution of heap data structures is to capture the traditional memory graph at each program execution point where it is modified.

For example, Figure 3.2 shows the execution of a sequence of statements from a C program that creates a singly-linked list by creating two nodes (statements 11 and 13) and linking them to form the list (statement 15). The last statement (19) is faulty, mistakenly breaking the linked list via the NULL assignment. The programmer can examine the corresponding series of traditional memory graphs [87] and understand how the link list grows and is finally broken by the execution of the faulty statement (19). While examining the sequence of memory graphs allows the programmer to observe the evolution of the link list, including its corruption, this approach is impractical due to its memory cost.

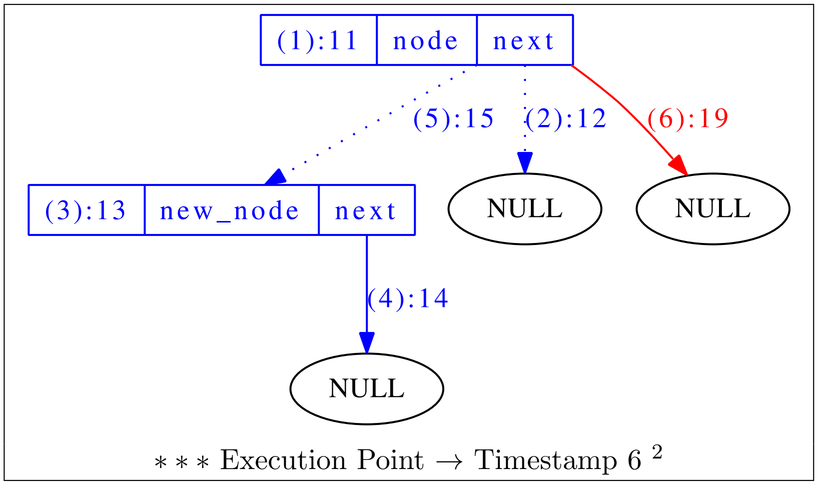


Figure 3.1: The compact MG++ representation.

To efficiently capture the memory graph's evolution we introduce a compact representation, MG++, from which the memory graph at any execution point can be extracted. As we can see in Figure 3.1, MG++ is compact because, by construction, MG++ eliminates redundancy across the series of memory graphs corresponding to the six execution points uniquely identified by timestamps 1 through 6. The additional annotations in MG++ rep-

[2] Throughout the paper, *Execution Point* → *Timestamp t* stands for "at execution point corresponding to timestamp *t*".

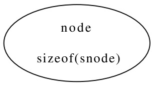
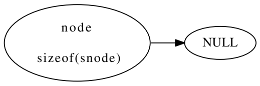
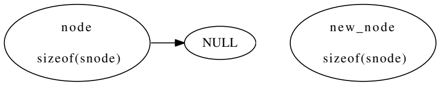
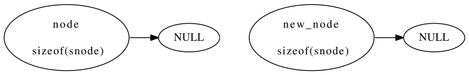
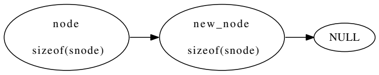
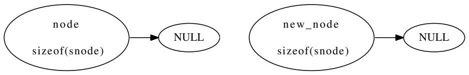
Execution trace	**Traditional Memory Graph**
11.node = malloc(**sizeof**(snode)); ∗ ∗ ∗ Execution Point → Timestamp 1	
12.node−>next = NULL; ∗ ∗ ∗ Execution Point → Timestamp 2	
13.new_node=malloc(**sizeof**(snode)); ∗ ∗ ∗ Execution Point → Timestamp 3	
14.new_node−>next = NULL; ∗ ∗ ∗ Execution Point → Timestamp 4	 node sizeof(snode) NULL new_node sizeof(snode) NULL
15.node−>next = new_node; ∗ ∗ ∗ Execution Point → Timestamp 5	
19.node−>next = NULL; ∗ ∗ ∗ Execution Point → Timestamp 6	 node sizeof(snode) NULL new_node sizeof(snode) NULL

Figure 3.2: Executed statements and corresponding traditional Memory Graphs.

resent *timestamps* for capturing the order in which nodes/edges are created/deleted and identities of *source code statements* responsible for changes to the memory graph. In particular, in Figure 3.1:

- The two non-NULL nodes are labeled with (1):11 and (3):13 indicating their creation at timestamps (1) and (3) by execution of statements 11 and 13, respectively;
- The outgoing edge from new_node−>next to NULL is labeled (4):14 as it was created at timestamp (4) by execution of statement 14; and
- Since node−>next is assigned at timestamps (2), (5), and (6) by statements 12, 15, and 19, it has three outgoing edges labeled (2):12, (5):15, and (6):19. The lifetimes of these edges can be inferred from the timestamps – the (solid) edge labeled with timestamp (6) is the most recent edge; the earlier (dashed) edges exist from the time of their creation to when the next edge is created.

We observe that the use of timestamps prevents *redundancy* across multiple memory graphs and thus makes the MG++ compact. In particular, if a node or an edge is created at timestamp t, and it remains unchanged until the end of execution, represented by timestamp T, then the MG++ will have a single copy of the node or edge labeled with t implying that it has remained unchanged until T.

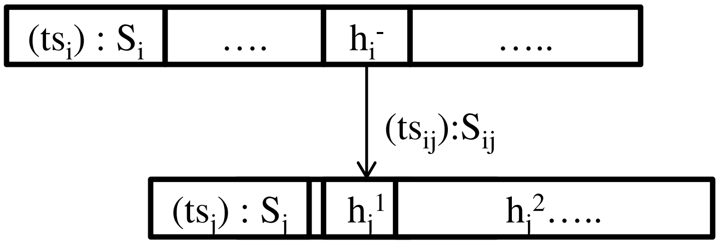
Given a MG++, the memory graphs corresponding to series of execution points can be extracted and shown to the user. The user can then observe the evolution of the dynamic data structure, identifying steps in execution at which the data structure appears to get corrupted, and then, using the statement numbers contained in MG++, identify the faulty code. We now provide a formal definition of MG++.

The MG++ is defined as a tuple (V, E) such that:

- V is a set of nodes such that each node v_i consists of $\langle (ts_i) : S_i; H_i \rangle$, where H_i is a set of heap addresses $\{h_i^1, h_i^2, \ldots\}$ that the node represents, ts_i is the timestamp at which the node was created, and S_i is the source code statement which led to creation of the node v_i. The statement is identified by its location in the source code, i.e., $\langle$file name:line number$\rangle$.

$(ts_i) : S_i$	$h_i{}^1$	$h_i{}^2$…..

- E is a set of directed edges $H_i.h_i^k \rightarrow H_j.h_j^1$ ($H_i.h_i^k$ represents the heap address that contains a pointer to the heap address $H_j.h_j^1$ where $H_j.h_j^1$ is the first heap address of node v_j); each edge has a label $\langle (ts_{ij}) : S_{ij} \rangle$, where ts_{ij} is the timestamp at which the edge was created and S_{ij} is the source code statement that created the edge. There can be multiple edges corresponding to the same heap address. The edge with the highest timestamp is marked as the current edge.



3.1.2 Modeling the Memory Allocator

The MG++ representation presented so far does not capture the behavior of the memory allocator itself. Therefore it may be ineffective in cases where understanding program behavior requires allocator information, or when the program has a custom memory allocator for dynamic data structures. In such cases, a memory graph node can no longer be simply defined as an allocated chunk of memory, since the allocator's actions may split a big memory chunk into smaller chunks (during allocation) or join two smaller chunks into a bigger one (following a free).

To capture the history of splitting and merging of memory chunks, we introduce two new kinds of nodes and edges, called *cluster nodes* and *merge edges*, in the MG++ representation. A *cluster node* marks a big consolidated memory chunk formed by joining multiple smaller memory chunks. Representing the node as smaller nodes joined by merge edges enables us to track the history of memory allocation and deallocation operations. This action is captured in the memory graph by joining the two nodes using a *merge edge*. For the purpose of interaction with other nodes, a cluster node is a single node although it internally stores multiple nodes corresponding to earlier smaller chunks.

Execution trace

```
1.tmp1 = malloc(sizeof(struct snode));      ***1
2.tmp2 = malloc(sizeof(struct snode));      ***2
3.tmp3 = malloc(sizeof(struct snode));      ***3
4.tmp4 = malloc(sizeof(struct snode));      ***4
5.tmp5 = malloc(sizeof(struct snode));      ***5
..
6.tmp1->next = tmp2;                        ***8
7.tmp2->next = tmp3;                        ***9
8.tmp3->next = tmp4;                        ***10
9.tmp4->next = tmp5;                        ***11
10.tmp1->next = tmp3;                       ***12
..
11. free(tmp2);                             ***15
..
12. free(tmp4);                             ***18
..
13.tmp1->next = tmp3;                       ***21
..
14.tmp3->next = NULL;                       ***25
15. free(tmp5);                             ***26
```

MG++ Representation

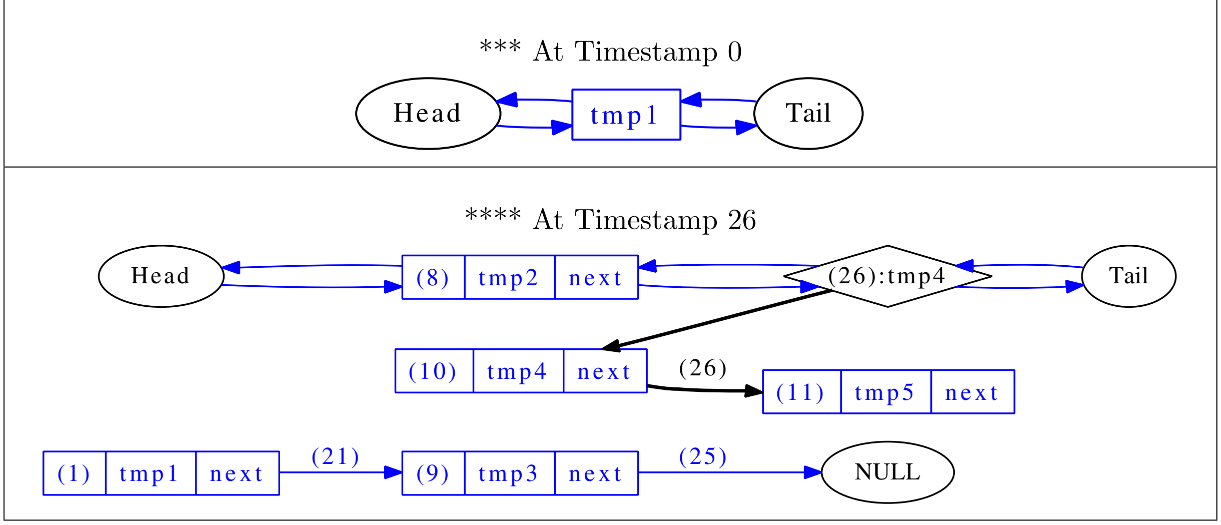
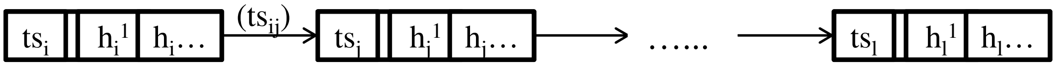


Figure 3.3: MG++ capturing the actions of the memory allocator.

Figure 3.3 shows a sample execution trace of a C program that uses an allocator based on Lea's *dlmalloc* allocator [58] along with corresponding timestamps. In addition, we also show the MG++ immediately before the execution and at the end of the execution. *Dlmalloc* maintains the free memory chunks in a doubly-linked list. The oval *head* and *tail* nodes have been shown in the figure for clarity. The MG++ at timestamp 0 shows such a free list with a big memory chunk having starting address `tmp1`. *Dlmalloc* serves different memory requests by splitting this big chunk into smaller chunks, and stores back the freed memory chunks in the same doubly-linked list. When two contiguous memory chunks are freed, we consolidate them to form a bigger memory chunk. Such a chunk is formed in this example when adjacent memory chunks corresponding to `tmp4` and `tmp5` are freed (lines 12 and 15) and are consolidated via internal malloc actions. MG++ stores this information using a cluster node – the diamond-shaped node shown in Figure 3.3. The cluster node has timestamp 26 and points to the two smaller chunks joined by a *merge edge* (edge corresponding to timestamp 26). A cluster node enables the MG++ to retrieve the earlier heap snapshot using the timestamp information.

The formal definitions of the set of *cluster nodes* and *merge edges* follow:

- V' is a set of *cluster nodes* such that each cluster node v'_i is defined as $< (ts_i) : starting\_address; N_i >$ where N_i is an ordered list of nodes $v_i \in V$ joined together by *merge edges.*



- A *merge edge* m connects two nodes $v_i, v_j \in V$ inside a cluster node and has a label

$< (ts_{ij}) >$ such that the timestamp marks the merging of the node v_j in the cluster node.

- Each node $v_i \in V$ carries a *sourceID* which marks the parent nodeID corresponding to the node out of which the node v_i is formed after a split.

The definitions of the set of nodes V and the set of edges E are similar to those in Section 3.1.

Algorithm 1 Memory graph retrieval algorithm

```
/* $n_i$: node in MG++; $n_j$: node in memory graph; MG++$_{target}$: MG++ at target
timestamp; MG$_{target}$: Memory Graph at target time stamp */
INPUT: MG++$_{final}$ - the MG++ at final timestamp $ts_{final}$; target timestamp $ts_{target}$
where $ts_{target} \leq ts_{final}$
function GRAPH_RETRIEVE()
    Step 1: /* retrieve MG++$_{target}$ */
        Remove all the nodes created after $ts_{target}$
        Remove all the edged created after $ts_{target}$
        Join all the nodes split after $ts_{target}$
        Separate all the nodes merged after $ts_{target}$
    for all Heap addresses $h_i$ in MG++$_{target}$ do
        Set the outgoing edge with highest timestamp as the current Edge
    end for
    Step 2: /* retrieve MG$_{target}$ */
    for all nodes $n_i$ in MG++$_{target}$ do
        starting_address($n_j$) ← Head($n_i$)
        Size($n_j$) ← Size(addrList($n_i$))
        MG$_{target}$ ← $MG_{target} + n_j$
    end for
    for All edges $e_i$ in MG++$_{target}$ do
        add corresponding edges in MG$_{target}$
    end for
return MG$_{target}$
end function
```

3.1.3 MG++ Rollback and Retrieval

Given the MG++ at timestamp ts_{final}, the memory graph MG for any time stamp $t \leq ts_{final}$ can be efficiently reconstructed by selecting appropriate subsets of nodes and

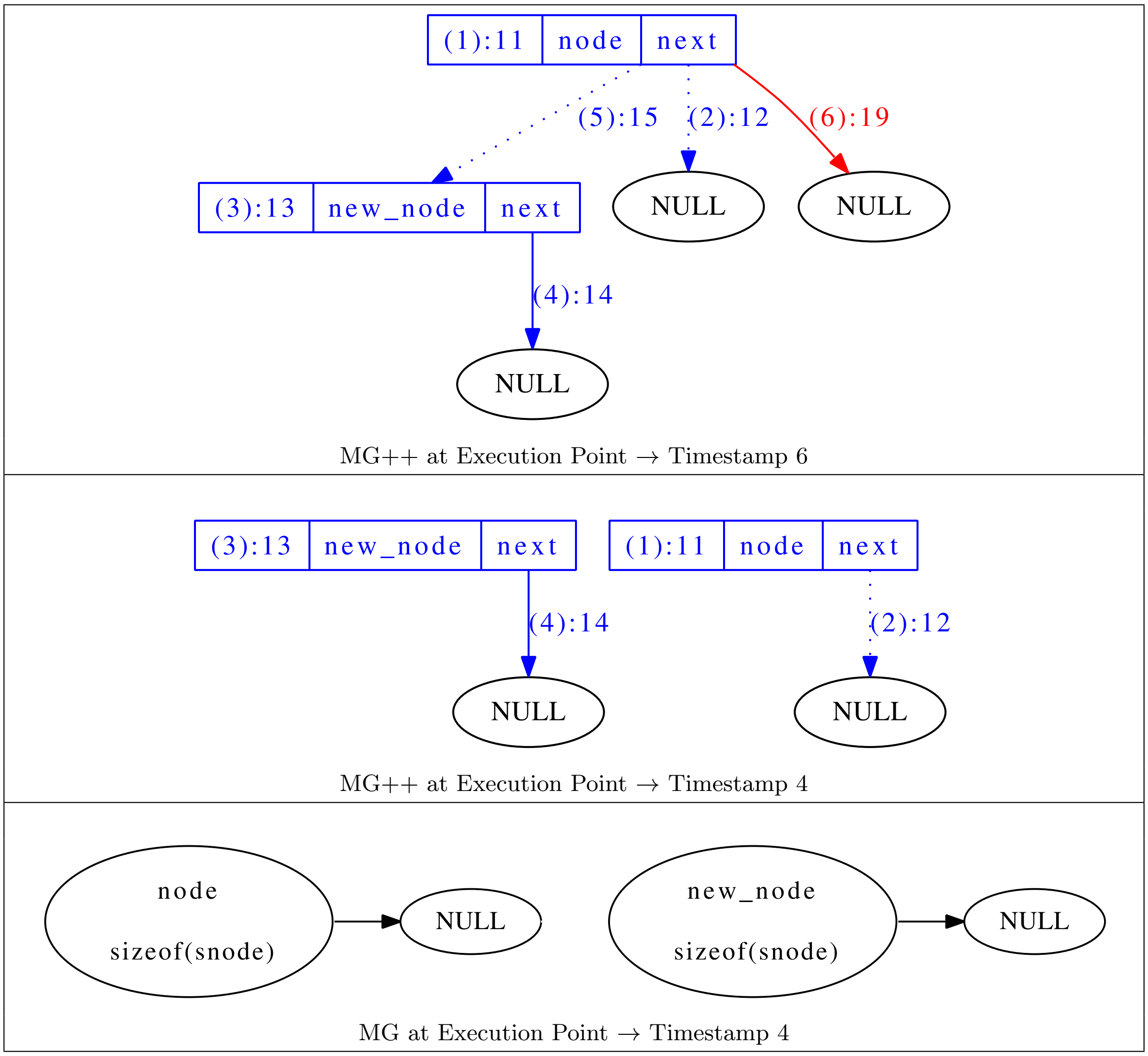


Figure 3.4: Memory graph rollback and retrieval.

edges. We can reconstruct the step-by-step evolution snapshots of the memory graph enabling us to navigate back and forth over the changes in memory graph during the execution.

Algorithm 1 shows how we retrieve MG$_{target}$, the memory graph at target timestamp ts_{target} from MG++$_{final}$, the MG++ corresponding to final timestamp ts_{final}, such that $ts_{target} \leq ts_{final}$. The retrieval takes place in two steps. In the first step, we retrieve MG++$_{target}$, the MG++ at the target timestamp ts_{target}. For this, all the nodes, edges, and merge edges having timestamp greater than ts_{target} are removed from the graph. Also, the addresses of any nodes that were split after the target timestamp are joined together. Removal of nodes may result in isolated data nodes, which are removed. Edges which were overwritten by a store executed after the target timestamp are restored as follows. For each of the heap addresses, the edge with the highest timestamp is set as current edge. Similarly, for each node, merge edge with the highest timestamp is set as current merge edge. In the second step, MG_{target}, the memory graph at the target timestamp is constructed from $\mathrm{MG}{+}{+}_{target}$. This is done by creating nodes and edges in the memory graph corresponding to the nodes and edges in MG++. The starting address of a node is the same as the head of the address list in the corresponding MG++ node. The size of a memory graph node is calculated by joining the sizes of addresses in the address list of the corresponding MG++ node.

Figure 3.4 illustrates retrieval of the memory graph at time stamp 4 from a MG++ at timestamp 6. In the first step, the node timestamps are examined. Since both nodes have timestamps less then 4, they are retained. The edges with timestamps ≥ 4, i.e., edges with timestamps 5 and 6, are deleted. This leads to an isolated data node which is removed, yielding the MG++ at program point corresponding to timestamp 4. The starting addresses of the two nodes are node and new_node, respectively. The sizes of these nodes are equal to the size of snode, i.e., size of head address + size of next.

3.2 Portable Memory Graph Construction

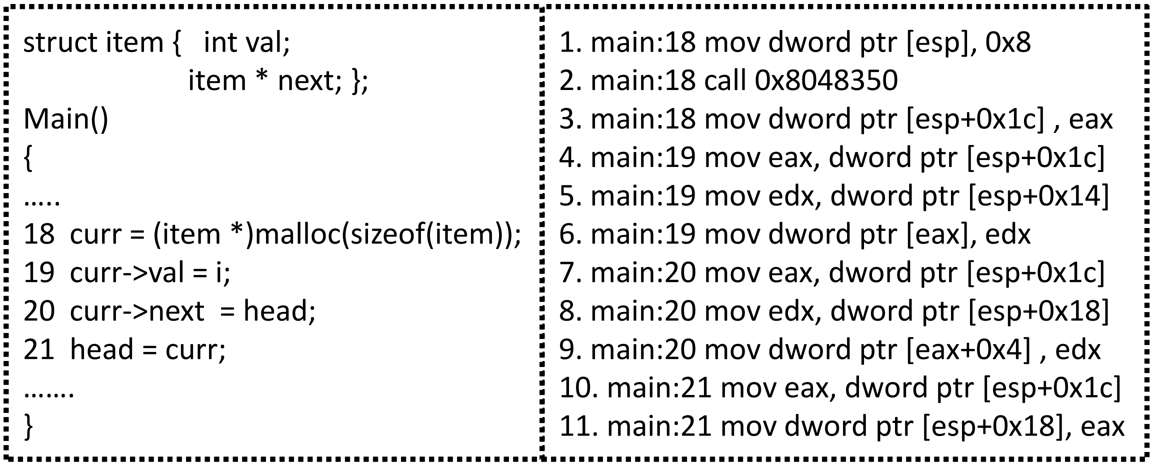
We have developed a novel MG++ construction algorithm based on binary instrumentation. By not relying on allocator function information, we develop a portable algorithm that can build the MG++ for programs using different memory allocators, custom allocators, and in-program memory managers. The implementation does not rely on source code or symbol table information. To our knowledge, this is the first attempt to capture the memory usage of a program without relying on source code or symbol table information.

3.2.1 Key Observations

The MG++ construction is based on the following two observations.

(1) *Accesses to fields within a memory node.* Each address inside an allocation site (i.e., node in the memory graph) is always derived from the starting address. The fields can be accessed as an offset from the starting address or the field address is explicitly calculated by adding the offset to the starting address. For example, in Figure 3.5 the starting address of the memory allocated is stored in register *eax*, the node field *val* in instruction number 6 is accessed as $eax + 0$, and in instruction number 9 the node field *next* is accessed as $eax + 4$. We observed this behavior in a variety of compilers: GCC, LLVM, Microsoft VC++, and Intel's C compiler.

Even when using registers to pass pointers to fields, or to pass the contents of a **char** array from within a structure as a string, an address inside the allocated memory chunk is accessed transitively from the starting address (address C is accessed as an offset from address B, while B is accessed as an offset from starting address A). This observation can be understood in terms of allocator behavior: the allocator returns the starting address of an allocated memory chunk to the program and the program can access the internal addresses of an allocated memory chunk only through the starting address of the memory chunk. *The above observation lets us join all the addresses being derived from the same starting address into a memory graph node, i.e., nodes can be identified without knowledge of memory allocator functions used.*



```
struct item {   int val;
                item * next; };
Main()
{
…..
18  curr = (item *)malloc(sizeof(item));
19  curr->val = i;
20  curr->next  = head;
21  head = curr;
…….
}
```

```
1. main:18 mov dword ptr [esp], 0x8
2. main:18 call 0x8048350
3. main:18 mov dword ptr [esp+0x1c] , eax
4. main:19 mov eax, dword ptr [esp+0x1c]
5. main:19 mov edx, dword ptr [esp+0x14]
6. main:19 mov dword ptr [eax], edx
7. main:20 mov eax, dword ptr [esp+0x1c]
8. main:20 mov edx, dword ptr [esp+0x18]
9. main:20 mov dword ptr [eax+0x4] , edx
10. main:21 mov eax, dword ptr [esp+0x1c]
11. main:21 mov dword ptr [esp+0x18], eax
```

Figure 3.5: Memory access example.

(2) *Pointers point to the head address of a memory graph node.* When the internal actions of a memory allocator are also being considered, we cannot rely only on the first observation for constructing a memory graph node. The allocator gets the starting address of the memory space from the system (using the `brk()` or `mmap()` system calls) and derives all the internal addresses from this starting address. Using only the first observation we will end up with a single node (multiple nodes in case of `mmap()`) for the whole program.

Therefore we form memory graph nodes using the observation that all pointers point to the head address of a memory graph node. Any address being pointed to becomes the starting address of a new memory graph node.

In prior works memory graph nodes correspond to allocated memory chunks and the construction techniques rely on the knowledge of calls to the allocator function. Moreover, if the program uses a custom memory allocator or it manages memory internally, then traditional memory graph representations fail to provide any useful information because they will simply show a single memory graph node.

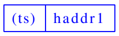
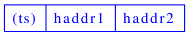
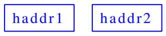
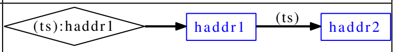
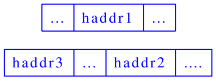
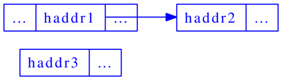
Operation	**Instruction**	**MG++ Before**	**MG++ After**
Join_Addresses $(haddr_1, haddr_2)$	$haddr1 + \text{offset}$ $(= haddr2)$		
Merge_Nodes $(haddr_1,\ haddr_2)$	$haddr1 + \text{offset}$ $(= haddr2)$		
Split_Node$(haddr_2)$	$haddr_1 \leftarrow haddr_2$		

Figure 3.6: MG++ construction operations.

3.2.2 Construction Algorithm

Given a program execution trace which captures the operations on heap references during the program execution, our algorithm builds the MG++ by grouping together heap references to form a memory graph node. The technique for identifying heap references is described in the implementation description (Section 4.4). Whenever a heap address-handling instruction is encountered, it is analyzed for its effects on the graph. The *timestamp* attached to each node and edge marks the order of their creation during the execution of the program. For example, a node will always have a lower timestamp than an edge pointing to it because the node was formed earlier in program execution and the edge was added later. The timestamp is initialized at the start of memory graph construction and is incremented with each change to the graph.

Note that we do not capture information about node deallocation, as the MG++ maintains information about nodes even after their deallocation (complete evolution history) so deallocation does not require any special treatment.

Algorithm 2 summarizes memory graph construction, changing the memory graph according to the instructions being executed. For a heap address *haddr*, in Algorithm 2, *node*(*haddr*) denotes the node corresponding to *haddr*. Given an instruction i, the following four cases arise:

Case 1) If the current instruction i operates only on *one heap address* which has never been encountered before the current execution point, then a new node is created (call to Create_Node, line 8). In Create_Node, the address is added to the address list of the new node. The node is also given a new timestamp which marks its creation.

Case 2) If instruction i operates on *heap address + offset*, the algorithm proceeds as follows. If both the base heap address and offset heap address have not been encountered earlier, then a new node is created and both addresses are added to the address list of the node (omitted from the algorithm for simplicity).

– If the base heap address has been encountered earlier (i.e., it corresponds to a node) and the offset heap address has not been encountered until that execution point, then the offset heap address is added to the address list of the node corresponding to the base address (Figure 3.6, row 1).

Algorithm 2 Memory Graph construction

```
 1: /* $haddr$: heap address; node($haddr$): node corresponding to $haddr$; $data$: non heap address
    value */
 2: INPUT: Execution Trace
 3: OUTPUT: Memory Graph (MG)
 4: function GRAPH_CONSTRUCTION()
 5:     switch $instruction\ i$ :
 6:         case $i$ has $haddr_1$
 7:             if $haddr_1 \notin$ MG then
 8:                 Create_Node($haddr_1$)
 9:             end if
10:         case $i$ has $haddr_1 + offset (= haddr_2)$
11:             if $haddr_2 \notin$ MG then
12:                 Join_Addresses($haddr_1, haddr_2$)
13:             else if node($haddr_2$) $\neq$ node($haddr_1$)  then
14:                 Merge_Nodes($haddr_1$, $haddr_2$)
15:             end if
16:         case $i$ is $haddr_1 \leftarrow data$
17:             Create_Edge($haddr_1, data$)
18:         case $i$ is $haddr_1 \leftarrow haddr_2$
19:             if $haddr_2 \notin$ MG then
20:                 Create_Node($haddr_2$)
21:             else if $haddr_2$ is not head of node then
22:                 Split_Node($haddr_2$)
23:             end if
24:             Create_Edge($haddr_1$, $haddr_2$)
25: end function
```

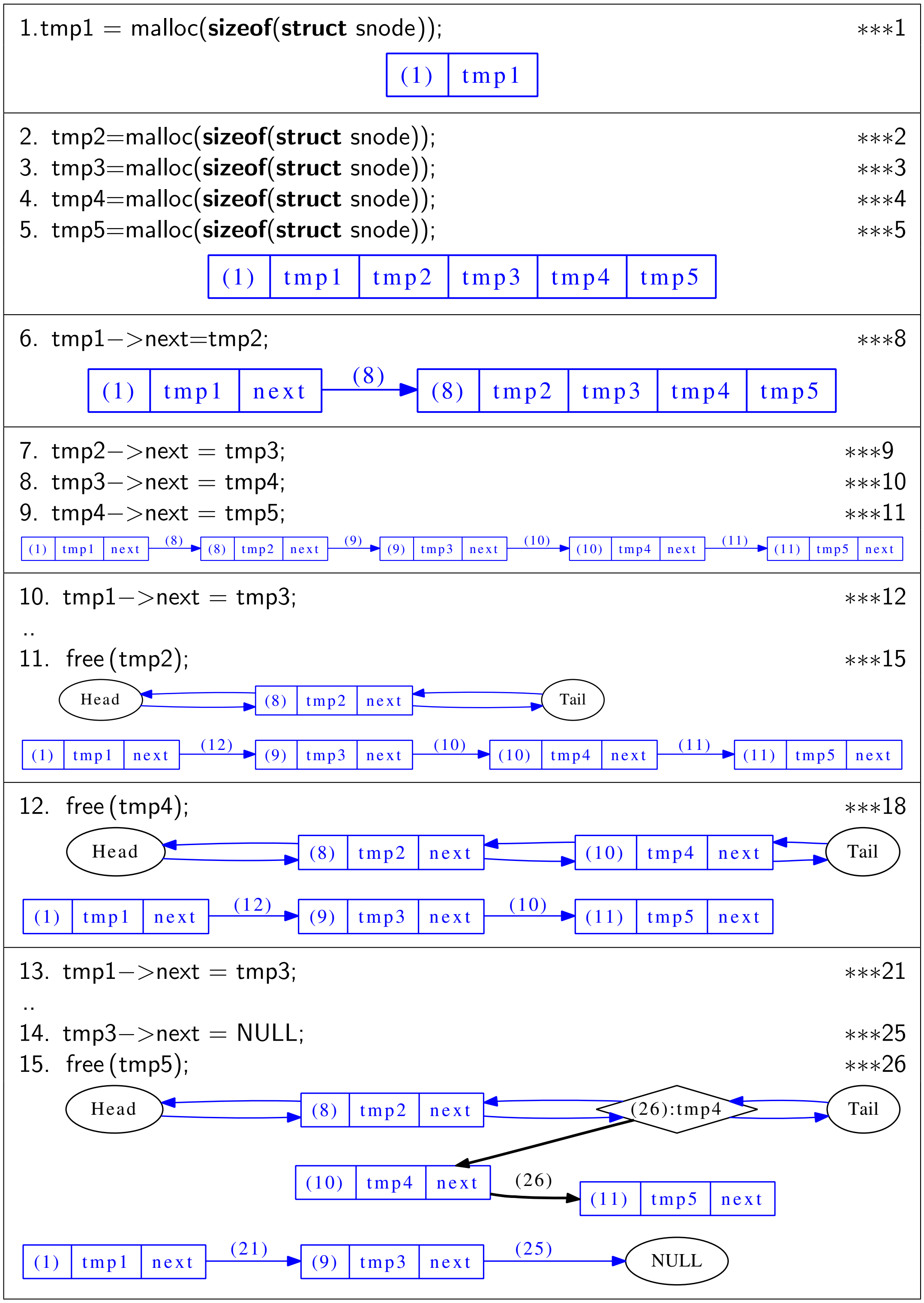
– If both the base heap address and the offset address have already been encountered before, and correspond to different nodes, then the nodes are merged together using the merge edge (call to Merge_Nodes, line 14). This leads to the creation of a cluster node (Figure 3.6, row 2). An example of this situation is the memory allocator consolidating two adjacent free memory chunks into one bigger chunk.

Case 3) If the instruction i is a *memory write*, A←B where A is a *heap address* and B is a *data value* (e.g., tmp3−>next = NULL in Figure 3.3) then a data edge from the address is created (call to Create_Edge, line 17). The edge is assigned a new timestamp that marks its creation.

Case 4) If instruction i is a *memory write*, A←B where both A and B are *heap addresses* (e.g., tmp1−>next = tmp2 in Figure 3.3), an edge is created, from the address written to, to the node corresponding to the address value written (call to Create_Edge, line 24). If the address value written is not the starting address of a memory graph node then the node is split such that the address value written forms the head of the newly created node (Figure 3.6, row 3). A common example of such a situation is when the memory allocator allocates a smaller chunk out of a bigger free memory chunk. If the address value written is a new address, a new node is created as in step 1 (call to Create_Node, line 20).

Node merging and splitting operations come into play only when the internal actions of the memory allocator are included in the analysis. If such internal actions are not considered in the analysis then accessing an address from a node as an offset from the address of a different node (line 13: $haddr_1 + offset (= haddr_2)$ and node($haddr_2$) $\neq$ node($haddr_1$)) is considered to be suspicious program behavior, e.g., a potential heap buffer overflow.

An Example. Figure 3.7 illustrates our approach when run on the example given in Figure 3.3. The first column shows the sample C source code statements executed at each step during construction of a linked list. The second column shows the memory graph formed once the source code in the same row is executed. When the execution starts on line 1, a new heap address tmp1 is encountered because of the call to malloc. A new node is created with initial time stamp of 1 (call to Create_Node, line 8 in Algorithm 2). Executing statements up to line 5, heap addresses tmp2, tmp3, tmp4 and tmp5 are encountered inside malloc. All these memory addresses are derived as an offset of the initial address inside



```
1.tmp1 = malloc(sizeof(struct snode));                 ***1

2. tmp2=malloc(sizeof(struct snode));                  ***2
3. tmp3=malloc(sizeof(struct snode));                  ***3
4. tmp4=malloc(sizeof(struct snode));                  ***4
5. tmp5=malloc(sizeof(struct snode));                  ***5

6. tmp1−>next=tmp2;                                    ***8

7. tmp2−>next = tmp3;                                  ***9
8. tmp3−>next = tmp4;                                  ***10
9. tmp4−>next = tmp5;                                  ***11

10. tmp1−>next = tmp3;                                 ***12
..
11. free(tmp2);                                        ***15

12. free(tmp4);                                        ***18

13. tmp1−>next = tmp3;                                 ***21
..
14. tmp3−>next = NULL;                                 ***25
15. free(tmp5);                                        ***26
```

Figure 3.7: An illustration of MG++ construction.

malloc so they are added inside the same MG++ node (call to Join_Addresses, line 12 in Algorithm 2). When the next field is written with value tmp2, the original node is *split* into two nodes (Split_Node, line 22 in Algorithm 2) to form tmp2 as the starting address of the new node. Similarly, other nodes are created due to memory writes in the lines 7, 8 and 9. The free operation on tmp2 leads to the node's addition to the doubly-linked list of free chunks internally maintained by malloc. We have represented the head and tail of the doubly linked list of free chunks with oval nodes (not present in the C code) for clarity. Similarly, free(tmp4) adds the node to the free list. When tmp5 which is adjacent to tmp4 is freed, malloc consolidates the two chunks. A *merge edge* is created to join the nodes corresponding to tmp4 and tmp5 and a *cluster node* is added to the MG++ (Merge_Nodes, line 14 in Algorithm 2). Note that the algorithm performs all of the above actions without knowledge of the memory allocator.

3.3 Implementation and Evaluation

Implementation overview. Our system takes a program binary as input. The binary is instrumented to capture the heap references and operations on them. The instrumented program is executed to generate a trace, containing the information about the heap references, operations on heap references and timing information. If debugging information is present in the binary then information about mapping to source code is also generated. The trace is analyzed off-line to generate the memory graph.

Our binary instrumentation is based on PIN-2.6 [68]. We only instrument loads, stores and a set of heap reference-handling instructions (ADD, LEA, SUB, INC, XCHG).

Other instructions may have to be instrumented depending on the compiler. The instrumented binary is then executed, generating the execution trace. At runtime, PIN determines the heap address range for the program by monitoring the system calls (brk() and mmap()) as well as the address space of the program, and outputs the valid heap address range and those instructions which handle heap references. These optimizations help reduce the size of our execution trace. The execution trace contains the timestamp information and the statement identifier for each of the executed instructions along with the address references involved in the instruction. An automated analysis of the execution trace constructs the memory graph. The valid heap address range is used to identify heap references in the trace.

Experimental setup. All measurements were performed on an Intel Core 2 6700 @ 2.66GHz with 4 GB RAM, running Linux kernel version 2.6.32. The experiments indicate that our memory graph representation and construction technique are efficient and effective in real-world scenarios.

Table 3.1: Overview of benchmarks.

Program Description	**Data Structure**	**Details**
ls (GNU core utilities [4])	Array & Linked list	ls -R coreutils-8.0/
Tidy (HTML check & clean [86])	Binary tree variant	Version 1.46
Bison (GNU parser generator [1])	Array & Linked list	Version 2.4.1
CPython [9]	Linked lists & Trees	Version 2.7.8
Perl [8]	Arrays & Linked lists	Version 5.20.0
Graph Coloring	Linked list of arrays	1,589 nodes; 2,742 edges
Independent Set	Linked list	2,361 nodes; 7,182 edges
Shortest Path	Shard	7,115 nodes; 103,689 edges

Table 3.2: Benchmarks' input description.

Program Description	**Input**
ls (GNU core utilities [4])	coreutils-8.0 source dir
Tidy (HTML check & clean [86])	Tidy Project Page.html
Bison (GNU parser generator [1])	ANSI C grammar
CPython [9]	Page rank program
Perl [8]	File search program
Graph Coloring	NetScience [16, 76]
Independent Set	YeastL (protein interaction network) [16, 98]
Shortest Path	Wiki-vote (Wikipedia network) [11]

Benchmarks. We measured the cost of using MG++ in terms of space and execution time to show that our approach is practical. A summary of benchmarks used in our evaluation is shown in Table 3.2. Our benchmarks are divided into three categories. First, widely-used programs: the **ls** GNU core utility, **Tidy** HTML checking&cleanup, and the GNU **bison** parser generator. Second, programs that perform internal memory management: the commonly-used **CPython** and **Perl** interpreters. In the third category, we have graph applications (graph coloring, independent set, and shortest path) with small real-world graphs as input, which we believe reflect common debugging scenarios. The benchmarks in the last category are read-intensive, "build-then-traverse data structure" style programs.

3.3.1 Cost of Constructing MG++

Space Required: Table 3.4's columns 2 and 3 show the space required for MG++, without and with allocator analysis, respectively. In order to show the space-efficient nature of MG++, we compared the space requirement of MG++ with that of a snapshot approach for capturing the data structure evolution history. The snapshot approach stores snapshots of memory graphs after each change using representations similar to the ones mentioned in [54, 74, 87]. The memory required to capture the sequences snapshots of memory graph is approximated by assuming a snapshot of the average number of nodes stored upon each change to the memory graph (Table 3.4, column 4). The size of each node is assumed to be 8 bytes (a single field). The space required by MG++, excluding (including) the allocator ranges from 3 MB to 138 MB (7 MB to 141 MB, respectively). This is two orders of magnitude less than that of capturing sequences of memory graphs using prior representations. In fact, this data shows that capturing sequences of memory graphs is impractical, while MG++ is highly space-efficient. The memory required for MG++ depends on the number of objects allocated (MG++ nodes) and manipulated (leading to creation of MG++ edges).

Program Execution Time: Table 3.4's columns 8 and 9 show the execution time for running the instrumented code for our technique, without and with allocator analysis, respectively. Table 3.4's column 5 (Original) shows the cost of running the original program; column 6 (Null PIN) shows the cost of running the program under PIN without any instrumentation; column 7 shows the cost of collecting trace for memory graph using allocator information [54, 74, 87], i.e., only calls to allocator functions and memory writes are instrumented. Although program execution time increases significantly due to instrumentation, it is a necessary cost of dynamic analysis. Column 10 shows the relative time overhead for our approach in comparison to the allocator-based approach. The average slowdown for the read-intensive benchmarks (graph-coloring, independent-set, shortest-path) is 3.16x while for the other benchmarks the average slowdown is 1.18x. The slowdown is higher in case of read-intensive programs because our MG++ construction algorithm has to monitor all the heap reference-handling instructions, including reads. The total average slowdown is

Table 3.3: Time overhead of capturing Memory Graphs.

1	2	3	4	5	6	7
Program Description	**MG++ Construction Method**					
	Program Execution Time (seconds)					**Slowdown**
			Instrumented			(col 8/ col 7)
	Original	**Null PIN**	Allocator approach	MG++ w/o allocator	MG++ with allocator	
ls	0.02	1.25	1.97	2.14	5.41	1.08
Tidy	0.01	1.46	6.46	8.90	9.63	1.37
Bison	0.06	2.04	2.23	2.68	9.82	1.20
CPython	0.09	7.84	24.60	30.74	32.85	1.25
Perl	1.11	11.73	26.09	26.41	32.52	1.01
Graph Coloring	0.05	0.68	8.69	37.33	38.08	4.29
Independent Set	0.05	0.53	13.44	47.27	50.79	3.51
Shortest Path	0.06	0.57	8.25	13.68	45.84	1.68

Table 3.4: Memory costs of capturing Memory Graphs.

1	2	3	4
Program Description	**MG++ Representation Space Required (MB)**		
	MG++		**Snapshot Approach**
	without allocator	with allocator	
ls	3.8	6.9	≈ 1,895
Tidy	22.4	32.1	≈ 2,484
Bison	2.5	32.3	≈ 283
CPython	137.8	140.5	≈110,242
Perl	48.1	50.2	≈ 65,652
Graph Coloring	82.3	83.5	≈ 311
Independent Set	6.3	10.2	≈ 451
Shortest Path	47.9	84.7	≈ 50,827

1.7x. These results show that the MG++ construction algorithm gives a feasible method of memory graph construction for cases where allocator information is not available. Instrumenting the memory allocator increases the execution times by 2x to 4x in comparison to when only application code is instrumented.

3.3.2 Fault Location using MG++

There are cases where the internal actions of the memory allocator are required to be a part of analysis and therefore MG++ can enable fault location. The Glibc memory allocator keeps track of free chunks via a doubly-linked list. Numerous real-world bugs *(GNOME BugIDs 697397, 449433, 318401; KDE BugIDs 119108, 281770, 224877, 237913)* are associated with the corruption of the free chunks list. Figure 3.8 shows a simple C program which causes Glibc corruption (**** glibc detected *** corrupted double-linked list*) later in the program. The programmer accidentally passes the address instead of the value of `tmp→target` in line 4. This overwrites the free chunks metadata, in turn corrupting the doubly-linked list. The program will crash later in the execution when `malloc` tries to access the elements of the doubly-linked list. We detect the violation of the doubly-linked list invariant by matching the invariant *if element **e** points to element **e'** then **e'** should point back to **e*** over MG++. Upon detecting corruption, the MG++ is rolled back and the statement responsible for corruption is identified.

```
1 struct node * tmp;
2 tmp = (struct node *)malloc(sizeof(struct node));
3 tmp→target = malloc(value_size);
4 memcpy(&tmp→target, value, value_size);
```

Figure 3.8: Sample code that corrupts Glibc's free chunks list.

3.3.3 Detecting Buffer Overflow Attacks using MG++

In this experiment we tested the use of MG++ to detect heap buffer overflows on the basis of Observation (1) in Section 3.2.1. Our assumption was that if an address of a memory node X is being accessed as an offset from an address inside memory graph node

Y then it is a heap buffer overflow. We tested our technique against 12 attack benchmarks from the RIPE buffer overflow testbed [110]. For this experiment we did not include allocator internals in the analysis. These benchmarks exercise various types of heap buffer overflows including return address, function pointers, and vulnerable structs, and have been commonly used in prior work to test the effectiveness of attack detection systems. As shown in Table 3.5, our technique was able to detect heap buffer overflow for all vulnerabilities, except for vulnerable structs. In the case of vulnerable structs, an attack-prone buffer resides along with a target function pointer inside the same memory graph node. There is no buffer overflow across memory graph nodes in this case and thus our technique fails to capture it. Valgrind's memcheck [75] can also detect other attacks except for vulnerable structs; it uses a valid bit for each byte of allocated memory and reports an error in case unallocated memory is written. Memcheck cannot be applied in the absence of allocator information while our technique will be able to detect buffer overflow without allocator information.

Table 3.5: Heap buffer overflow detection results.

Vulnerabilities	**# of benchmarks**	**Exception raised**
Return Address	1	Yes
Old base Pointer	2	Yes
Function Pointers	7	Yes
Vulnerable Structs	2	No

3.4 Limitations

Our approach has two limitations.

1. Our algorithm identifies a heap address by checking if the value falls in the valid heap address range. If a non-heap address operand with value in the valid heap address

range is encountered, our approach creates a one-field node in the memory graph for it. Such a node is unnecessary and can be removed using a post-construction analysis.

2. The size of a memory node constructed by our algorithm is different from the allocated size in cases where trailing fields of an allocated node are never read or written.

3.5 Summary

This chapter has presented a novel memory graph representation, MG++, along with MG++'s construction algorithm. The novelty of MG++ is that it carries dynamic data structure evolution history as well as a mapping to the source code modifying the dynamic data structure. The extended MG++ representation supports modelling the memory allocator internals as well. The construction algorithm does not require modeling the memory allocator semantic for memory graph construction, which makes the approach general and portable. The chapter has also presented the application of the construction algorithm for detection of heap buffer overflows.

Chapter 4

Fault Location Framework

This chapter presents the details of our fault location framework. Data structure fault location faces the problem of *temporary legitimate violation* of data structure consistency constraints. Fault location is a costly operation as it involves tracing back through the data structure evolution history to search for faulty data structure mutations. This chapter tackles the problem of fault location and introduces optimizations to make fault location more efficient and practical.

4.1 Overview of Our Approach

In this section we provide an overview of our approach and highlight its key features using an example. The example is centered around quad trees, a widely-used data structure for spatial indexing. A quad tree is a tree with internal nodes having four children and the data stored at the leaf nodes. Thus one of the key structural consistency constraints for this data structure is: *for any internal element* ***e****, the number of children is four.* In Figure 4.1

```
struct pt { int x, y };
struct qdtree {
    int posX, posY, width, height;
    struct pt * point;
    struct qdtree * child [4];
}
struct qdtree * root;
int main() {
     ...
    while ( fscanf( file ,"%d%d",&x,&y)!=EOF)
        insert (x,y, root);
}
void insert (int x, int y, struct qdtree * root) {
    ##C−POINT(qdtree)
    if (root == NULL) {
        root = (struct qdtree *)malloc(sizeof(qdtree));
        temp = create_point(x, y);
        root→point = temp;
    }
    else {
        n = search(x, y, root);
        if (n→pt != NULL)
            split (x, y, n);
    }
    ##C−POINT(qdtree)
}
void split (int x, int y, struct qdtree * node){
    struct qdtree * temp;
    for (i = 0; i < 4; i++) {
        temp = (struct qdtree *)malloc(sizeof(qdtree));
        set_node_fields (temp, i , node);
        node→child[i] = temp;
        if (x==node[i]→posX && y==node[i]→posY) {
        ...
            parent_node→child[0]= NULL;
        }
        else {
            move_value(node, node→child[i]);
            assign_value (x, y, node→child[i]);
        }
    }
}
```

Figure 4.1: Faulty *Quad Tree* implementation.

we show example code that creates and manipulates quad trees, and contains a bug which leads to a violation of the consistency constraint. The quad tree definition (lines 2–7) contains nine fields: the first five fields store data about the node, while the next four fields, child [4], point to children in the quad tree. The main function reads coordinates (x,y) from a file and populates the tree by calling the insert function. In insert, we first search for an existing node that is suitable for coordinates (x,y). If the resulting node n already has a point stored in it, then four children of n are created, one for each quadrant. The point at node n and the newly-read point are then inserted into the quad tree rooted at n. The statement at line 35 in function split which sets an edge to NULL is faulty.

qdtree FIELD 9 EDGE 5;
/*# total fields and # pointer fields */
qdtree X; ..(c_1)
X.ISROOT == FALSE ⇒ X.INDEGREE == 1;

qdtree X; *qdtree* Y; ..(c_2)
X $\twoheadrightarrow$ Y ⇒ Y $\not\twoheadrightarrow$ X;

qdtree X; *qdtree* Y; ..(c_3)
X→ Y ⇒ X.OUTDEGREE == 4;

Figure 4.2: Consistency constraints of a *Quad Tree.*

4.1.1 Specification of Consistency Constraints

Our specification language enables the developer to express the data structure constraints directly in terms of the relationships among heap elements.

Figure 4.2 shows the consistency constraint specification for a quad tree in our language. The user specifies the structure of each node in the memory graph by declaring a node type (*qdtree*) that contains 9 data fields and 5 pointer fields – (FIELD 9) and

(EDGE 5); constraints are specified next. The constraint specification involves declaring node variables and specifying the relationships among them. For quad tree we specify three constraints in terms of variables X and Y of node type *qdtree*. The constraint c_1 indicates that all nodes besides the root have an indegree of 1. The constraint c_2 indicates that if there is a path from node X to node Y, then there cannot be a path from node Y to node X. The constraint c_3 specifies that, if any node X points to another node Y (i.e., the node X is internal) then the outdegree of X is 4. While we have chosen a simple constraint for the purpose of understanding, our language can handle a variety of complex constraints (as explained in Chapter 2).

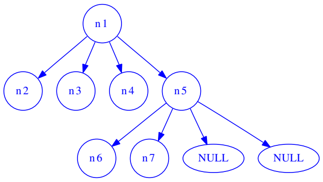
4.1.2 Tracing Data Structure Evolution History

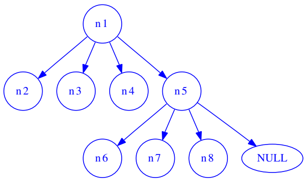
We execute the program and trace the evolution history of the program data structure using binary level dynamic instrumentation. The instrumentation is independent of the constraints specified. We only instrument the allocation/deallocation calls and memory writes to the allocated memory. Once the program execution completes; normally or because of a program crash, the traced information is used to construct memory graph.

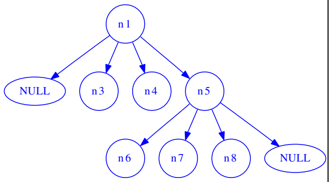
4.1.3 Fault Location

We match the specified constraints over the memory graph at program points corresponding to *C-points*. The memory graph must be in a consistent state with respect to the specified constraints at these program points. Once we encounter a consistency constraint violation, the process of fault location begins. The process of fault location involves tracing back the program execution. We analyze the effect of each operation (on

Execution trace	**Inconsistencies**	**Memory Graph**

temp = (**struct** qdtree ∗)
malloc(**sizeof**(qdtree));
set_node_fields (temp, 1, node);
node−>child[1] = temp;
move_value(node, node−>child[1]);
assign_value (x, y, node−>child[1]);
i++;
∗ ∗ ∗∗ Execution Point 1
	Inconsistences
	@Point 1
	$S_1 = \{$
	$\langle c_3, \{n5 \rightarrow n6\}\rangle,$
	$\langle c_3, \{n5 \rightarrow n7\}\rangle$
	$\}$
		

temp = (**struct** qdtree ∗)
malloc(**sizeof**(qdtree));
set_node_fields (temp, 3, node);
node−>child[3] = temp;
∗ ∗ ∗∗ Execution Point 2
	Inconsistencies
	@Point 2
	$S_2 = \{$
	$\langle c_3, \{n5 \rightarrow n6\}\rangle,$
	$\langle c_3, \{n5 \rightarrow n7\}\rangle,$
	$\langle c_3, \{n5 \rightarrow n8\}\rangle$
	$\}$
		

parent_node−>child[0]= NULL;
i++;
∗ ∗ ∗∗ Execution Point 3
	Inconsistencies
	@Point 3
	$S_3 = \{$
	$\langle c_3, \{n1 \rightarrow n3\}\rangle,$
	$\langle c_3, \{n1 \rightarrow n4\}\rangle,$
	$\langle c_3, \{n1 \rightarrow n5\}\rangle,$
	$\langle c_3, \{n5 \rightarrow n6\}\rangle,$
	$\langle c_3, \{n5 \rightarrow n7\}\rangle,$
	$\langle c_3, \{n5 \rightarrow n8\}\rangle$
	$\}$
		

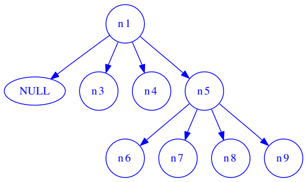
temp = (**struct** qdtree ∗)
malloc(**sizeof**(qdtree));
set_node_fields (temp, 3, node);
node−>child[3] = temp;
move_value(node, node−>child[3]);
assign_value (x, y, node−>child[3]);
∗ ∗ ∗∗ Execution Point 4 - C-POINT
	Inconsistencies
	@Point 4
	- C-POINT
	$S_c = \{$
	$\langle c_3, \{n1 \rightarrow n3\}\rangle,$
	$\langle c_3, \{n1 \rightarrow n4\}\rangle,$
	$\langle c_3, \{n1 \rightarrow n5\}\rangle$
	$\}$
		

Figure 4.3: Memory graph at different program points.

Execution Point	**Inconsistencies**	**Pending Inconsistencies**	**Relation**
Execution Point 4	$S_4 = \{$ $e_1 = \langle c_3, \{n1 \rightarrow n3\}\rangle$, $e_2 = \langle c_3, \{n1 \rightarrow n4\}\rangle$, $e_3 = \langle c_3, \{n1 \rightarrow n5\}\rangle\ \}$	@4 $P = \{e_1, e_2, e_3\}$ $(Faulty\ Statements)$ $FS = \phi$	
Execution Point 3 O_3: $n5 \rightarrow$child[3] $= n9$; Execution Point 4	$S_3 = \{$ $e_1 = \langle c_3, \{n1 \rightarrow n3\}\rangle$, $e_2 = \langle c_3, \{n1 \rightarrow n4\}\rangle$, $e_3 = \langle c_3, \{n1 \rightarrow n5\}\rangle$, $e_4 = \langle c_3, \{n5 \rightarrow n6\}\rangle$, $e_5 = \langle c_3, \{n5 \rightarrow n7\}\rangle$, $e_6 = \langle c_3, \{n5 \rightarrow n8\}\rangle\ \}$	@3 $P = \{e_1, e_2, e_3\}$ $FS = \phi$	$\{e_4, e_5, e_6\}$ $\wr\ \{e_1, e_2, e_3\}$ $=$ FALSE $O_3 \rightsquigarrow e_1 =$ FALSE $O_3 \rightsquigarrow e_2 =$ FALSE $O_3 \rightsquigarrow e_3 =$ FALSE
Execution Point 2 O_2: $n1 \rightarrow$child[0] $=$ NULL; Execution Point 3	$S_2 = \{$ $e_4 = \langle c_3, \{n5 \rightarrow n6\}\rangle$, $e_5 = \langle c_3, \{n5 \rightarrow n7\}\rangle$, $e_6 = \langle c_3, \{n5 \rightarrow n8\}\rangle\ \}$	@2 $P = \phi$ $FS = \{O_2\}$	$\{e_4, e_5, e_6\}$ $\wr\ \{e_1, e_2, e_3\}$ $=$ FALSE $O_2 \rightsquigarrow e_1 =$ TRUE $O_2 \rightsquigarrow e_2 =$ TRUE $O_2 \rightsquigarrow e_3 =$ TRUE

Figure 4.4: Fault location on Figure 4.1.

the memory graph) on the inconsistencies present in the memory graph. The statements corresponding to the operations contributing to the inconsistencies are added to list of potentially faulty statements.

Our system performs consistency checks on the program memory graph at the beginning and at the end of function insert (lines 17 and 28) indicated by *C-points*. In Figure 4.3, the first column shows an execution trace of the code in Figure 4.1, while the second and third columns contain the set of constraints violated at selected program execution points and the corresponding quad tree. Note that execution point 4 is a *C-point* and we find that there are multiple violations of constraint c_3 from Figure 4.2 at this point. The set of violations is represented by the set S_c and each violation is described in form of a tuple $\langle C, G\rangle$, where G is the sub-graph over which constraint C is violated. In the

Figure 4.3 example, the constraint violations include: $\langle c_3, \{n1 \rightarrow n3\}\rangle$, $\langle c_3, \{n1 \rightarrow n4\}\rangle$, and $\langle c_3, \{n1 \rightarrow n5\}\rangle$. We perform consistency checks at earlier execution points, computing the violations into set S_i where i is the execution point. The fault location algorithm finds the operations contributing to inconsistencies in S_c by examining the S_i's.

In our example, the fault location algorithm determines that at program point 3, the operation of setting $n1 \rightarrow$child[0] to NULL introduces all the inconsistencies present in the set S_c. The statement corresponding to the operation is output as a faulty statement. Our algorithm also determines that none of the inconsistencies in S_2 are related to inconsistencies in S_c and therefore there is no need to further search for faulty statements. Note that there are other inconsistencies present at different program points that are not related to the inconsistencies at *C-points*. For example, at program point 3, the inconsistency $\langle c_3, \{n5 \rightarrow n8\}\rangle$ is temporary and ignored in the search for faulty statements.

4.1.4 Optimizations

We have optimized our fault location system in terms of both memory and time costs. For fault location, we need the memory graph at each execution point so that the constraint violations at each execution point can be detected. To avoid saving memory graphs at all execution points we employ a unified memory graph representation (given in Chapter 3) that combines memory graphs at all execution points into one and distinguishes subgraphs via association of timestamps with graph components (nodes and edges). Thus, portions of the graph that do not change across many execution points are stored only once. Given a unified memory graph representation at program point P, the memory graph for any earlier program point can be reconstructed using the process of *rollback* (explained in section 4.3). We employ an incremental algorithm which avoids redundant constraint evaluations over unchanged parts of the memory graph.

4.2 Fault Location Algorithm

Our location algorithm is based on the observation that there are two kinds of inconsistencies in the memory graph (MG): a *temporary* kind of inconsistencies that are removed prior to reaching *C-points*, and an *error* kind of inconsistencies, which are present at *C-points* and are caused by faulty statements. The algorithm identifies the statements responsible for the *error* kind of inconsistencies. We first define the key concepts and then present the algorithm.

Definition 1. An *inconsistency* e at execution point i is a tuple $\langle c_j, G_i \rangle$ where c_j is a constraint that is violated when evaluated over G_i, a subgraph of the memory graph at execution point i.

Example 6. Consider $\langle c_3, G_4 \rangle$ at execution point 4 in Figure 4.3 where c_3 is

qdtree X; qdtree Y;

X→Y ⇒ X.OUTDEGREE == 4;

and G_4 is $\langle \{n1 \to n3\} \rangle$. Then $\langle c_3, G_4 \rangle$ is an *inconsistency* at execution point 4 because c_3 is violated when $X = n1$ and $Y = n3$.

Note that there can be multiple inconsistencies corresponding to the same constraint violation as the constraint check can fail over multiple sub-graphs.

Consider the execution of operation O such that *beforeO* and *afterO* denote the execution points just before and after execution of O. Next we provide the conditions under which inconsistencies at *beforeO* and *afterO* are related (denoted by ‘$\wr$’) to each other and the conditions under which O is considered to be a potentially faulty operation.

Definition 2. An inconsistency $\langle c_j, G_b \rangle$ at execution point *beforeO* is said to be *related to* (denoted by ‘$\wr$’) an inconsistency $\langle c_k, G_a \rangle$ at execution point *afterO* **iff** the operation O has modified the arguments to the constraint check of c_j over G_a as well as the arguments to check c_k over G_b.

Example 7. In Figure 4.3, the inconsistency $e_1 = \langle c_3, \{n5 \rightarrow n8\} \rangle$ at program point 2 (after) is related to $e_2 = \langle c_3, \{n5 \rightarrow n6\} \rangle$ at program point 1 (before) because the operation of setting of edge for $n5$ to $n8$ modifies $n5$.OUTDEGREE which is an argument to check c_3 over the subgraph for both e_1 and e_2. In other words, $e_1 \wr e_2$ is TRUE.

Definition 3. Operation O is said to have *contributed to* (denoted by ‘$\leadsto$’) an inconsistency $\langle c_j, G_a \rangle$ present at program point *afterO* if the arguments to the constraint check of c_j over subgraph G_a at program point *beforeO* are not equal to the arguments to the constraint check of c_j over subgraph G_a at program point *afterO*.

Example 8. The operation $n1 \rightarrow \mathsf{child[0]} = \mathsf{NULL}$ (statement 10) in Figure 4.3, contributes to the inconsistency $e = \langle c_3, \{n1 \rightarrow n3\} \rangle$ present in row 3 because it has modified $n1$.OUTDEGREE which is an argument to check c_3.

Our fault location algorithm is presented in Algorithm 3. It begins by initializing the set of pending inconsistencies with inconsistencies at *C-point*(line 4). The system rolls

Algorithm 3 Fault Location

```
 1: $P$: Set of pending inconsistencies;
 2: $S_i$ denotes inconsistencies in $MG_i$ at execution point $i$;
 3: $O_i$: Operation on $MG$ performed at execution point $i$;
 4: $Check\_Constraints(MG, C)$: checks constraints in $C$ for $MG$ and returns the set of
    inconsistencies found;
 5: $Roll\_Back(MG_{i+1})$: rolls back $MG_{i+1}$ by one operation.
 6: **INPUT:** Memory Graph $MG_c$ at execution point $c$ for a *C-point* and constraint specification C $\langle c_1, ...c_n \rangle$.
 7: **function** FAULT_LOCATION()( )
 8:     $i \leftarrow c$; $P = Check\_Constraints(MG_c, \text{C})$
 9:     **repeat**
10:         $i \leftarrow i - 1$
11:         $MG_i = Roll\_Back(MG_{i+1})$
12:         $S_i = Check\_Constraints(MG_i, \text{C})$
13:         **for** each $e \in P$ **do**
14:             **if** $e \rightsquigarrow O_i == true$ **then**
15:                 Output($statement(O_i)$)
16:                 **for** each $e' \in S_i$ **do**
17:                     **if** $(e \wr e')\&\&(e! = e')$ **then**
18:                         $P = P \cup \{e'\}$
19:                     **end if**
20:                 **end for**
21:                 **if** $!(e \in S_i)$ **then**
22:                     $P = P - \{e\}$
23:                 **end if**
24:             **end if**
25:         **end for**
26:     **until** $P! = \{\phi\}$
27: **end function**
```

back memory graph one operation at a time (line 7) and checks if the rolled back operation has contributed to any of the pending inconsistencies (line 10). The statement corresponding to the contributing operation is output as faulty (line 11). Inconsistencies in the new memory graph which are related to the pending inconsistencies are added to the pending set (line 14). Any pending inconsistencies no longer present in the memory graph are removed from the pending set (line 18). When the set of pending inconsistencies reduces to $\emptyset$, the algorithm stops. Note that our algorithm only considers operations contributing to inconsistencies as faulty, rather than marking every statement that modifies the inconsistency subgraph as faulty; this helps increase precision.

Figure 4.4 illustrates the fault localization algorithm for the fault in Figure 4.1. The set of pending inconsistencies P is initialized with inconsistencies at execution point 4 (C-point) ($P = \{e_1, e_2, e_3\}$). The operation O_3: $n5 \rightarrow \mathsf{child[3]} = n9$ is not faulty because it does not contribute to any of the inconsistencies in P, i.e, it does not modify arguments to any of the inconsistencies present in P. The next operation $O_2 : n1 \rightarrow \mathsf{child[0]} = \mathsf{NULL}$ contributes to inconsistencies $\{e_1, e_2, e_3\}$ in P and hence is a faulty statement. Thus, O_2 is added to the FS set. None of the inconsistencies in S_2 are related to those in P; hence no new inconsistencies are added to P. The inconsistencies $\{e_1, e_2, e_3\}$ are removed from P causing it to become empty and the search for faulty statements terminates. In other words, O_2 is identified as faulty.

4.2.1 Identifying Corrupted Data Structures

Matching a constraint with the entire program memory graph will lead to false positives due to violations of a data structure constraint when it is evaluated for other unrelated data structures. To avoid this problem we track the identity of the data structure associated with each memory graph node during program execution. Using the data structure identity, we only evaluate constraints relevant to the memory graph node involved avoiding false positives. Furthermore, knowing the identity of the corrupted data structure helps us during trace back for faults as we limit our search only to the corrupted data structure instead of the whole program memory graph.

4.3 Optimizations

4.3.1 Incremental Constraint Checking

Constraint checking is a critical operation, and containing the cost of checks on large data structures allows our approach to scale well. We reduce the cost of checking via the use of incremental on-demand checks: we keep a mapping between constraint atoms and dependent nodes (nodes involved in the constraint); when the memory graph is modified, we map modified nodes to affected constraint atoms and invalidate those atoms.

4.3.2 Efficient Traceback

When tracing back for faults, performing rollback and constraint checking in a naïve way would significantly affect scalability due to the high cost of constraint checking. We use two techniques to make trace back efficient. First, knowing which data structure is corrupted, we limit our constraint matching (during trace back) to the data structure in question. Second, we use *modification prediction* to check if a rolled back operation can contribute to the inconsistencies present in the pending inconsistency list. We perform the rollback and consistency check on the memory graph only when the prediction for the operation returns true. Modification prediction is based on spatial locality. We can predict that an operation O operating on a sub-graph G_o will not affect an inconsistency $\langle c, G \rangle$, based on the properties of constraint c and the relationship between sub-graphs G_o and G. For example, an operation O of creating edge $n_1 \rightarrow n_2$ will not affect inconsistency $\langle c, n_3 \rangle$, where constraint c checks the value of a node field. We create a list of nodes (dependency list) for each inconsistency $\langle c, G \rangle \in P$ (set P in the algorithm 3) based on constraint c.

Modifications to a node present in the dependency list can affect the inconsistency. A check is performed if any of the inconsistent nodes for the pending inconsistency list (set P) is modified by the operation or is dependent on any of the nodes (i.e., in its dependency list) modified by operation O. Algorithm 3 is modified to directly roll back before an operation only when the modification prediction returns true for the operation.

4.4 Evaluating Fault Location

Next we evaluate the precision and cost of our fault location technique. Our implementation consists of a binary instrumenter, constraint matcher generator, memory graph constructor and a fault locator. The *binary instrumentation* is based on Pin-2.6 [68] – only allocation calls and memory writes are instrumented. To reduce the size of the execution trace, at runtime, Pin keeps track of allocated heap addresses and outputs only the instructions which write allocated heap addresses. The execution trace contains the timestamp information, and the statement identifier for each allocation and memory writes. We have used the source code location of memory allocation as a unique identifier of the data structure type for each memory graph node as each allocation site belongs to a unique data structure. The execution trace drives the construction of the memory graph. The *constraint matcher generator* produces the constraint matcher based on the input constraint specifications. If an error is detected during constraint matching, the *fault locator* searches for the root cause and outputs a list of candidate faulty statements. Measurements were performed on an Intel Core 2 6700 @ 2.66GHz with 4 GB RAM, running Linux kernel version 2.6.32. All benchmarks were written in C.

Table 4.1: Precision of fault location.[1]

1	2	3	4	5
Data structure	**Lines of code**	**Statements examined**		
		Ours	**Dyn. Slice**	**Tarantula**
Circular linked list	160	6	7	7
Ordered list	172	2	20	7
Doubly-linked list	203	5	38	17
Quad tree	294	1	58	9
AVL tree	243	4	9	10
B tree	405	6	8	8
Red-Black tree	395	10	24	13
Leftist heap	274	1	28	11
Bipartite graph	284	2	32	8

4.4.1 Precision of Fault Location

The strength of our technique lies in its ability to carry out highly precise fault location using a single test case during which constraints are violated – note that program execution may or may not lead to a program crash. Table 4.2 presents the results of our fault location technique for implementations of several data structures whose program sizes are given in the second column. In each case, the data structure was first initialized to a base size of 1,000 nodes. Next, 500 operations (inserts and deletes) were performed on the data structure along with random injection of 10 faults. In each case, the injected fault leads to violation of the data structure constraint. The program crash point was used as *C-point* in cases where program crashed. The end of execution was used as *C-point* in cases where program terminated normally with wrong output. Column 3 shows the number of faulty statements our technique detected. The numbers of faulty statements range from 1 to 10 while program sizes range from 160 to 405 lines of code. This indicates that our technique narrows down the fault to a very small code region. In all cases the fault was captured by the identified faulty statements.

We also computed the dynamic slice of the faulty statement instances. The fourth column shows the number of distinct program statements in the largest dynamic slice among faulty instances' slices. These numbers show that without the knowledge of data structure constraints, the set of potentially faulty statements identified can be quite large (≥ 20 for 6 programs). The fifth column reports the number of statements that must be examined to find the faulty statement using the ranking produced by Tarantula [50]. Tarantula is a statistical technique that uses information from multiple runs on different inputs – we used 1 failing run and 9 successful runs in this experiment. The results show that our approach requires fewer statements to be examined although it is based upon a single run as opposed to Tarantula that used 10 runs.

Table 4.2: Time overhead of fault location.[1]

1	2	3	4	5	6
Data structure	**Program execution time (ms)**			**Const. Match.**	**Fault Loc.**
	Original	Null Pin	Instrume.	**time** (ms)	**time** (sec)
Circular linked list	0.5	645	664	1	0.75
Ordered list	0.9	638	677	1	0.02
Doubly-linked list	0.7	653	673	1	0.25
Quad tree	2.6	713	885	36	9.22
AVL tree	1.6	739	864	25	1.18
B tree	1.5	722	767	4	73.39
Red-Black tree	0.4	661	711	31	72.21
Leftist heap	0.4	664	720	27	2.21
Bipartite graph	8.2	659	748	2	3.32

4.4.2 Overhead of Fault Location

In Table 4.2, columns 6–10 show the overhead in terms of time and space for using our fault location implementation. The 6th column shows the execution time of the buggy program. The 7th and 8th columns show the execution times when the benchmarks run under Pin without instrumentation ("Null Pin" column) and with instrumentation ("Instrumented" column). Although instrumentation overhead is significant, it is acceptable for debugging purposes. The 9th column shows that the time to perform constraint matching (after error detection) is typically a second or less. The 10th column shows the time for fault location—this is very efficient, 0.2–73 seconds in our tests.

4.5 Experience with Real Programs

For each application we defined consistency constraints for the main data structures. Next, we used fault injection to simulate common programming errors that lead to data structure corruption. Then, using our technique, we identified the buggy statements in the program. Table 4.4 shows our findings. The execution times are for the buggy version of the programs. Column 4 gives the number of faulty statements our technique found. Our fault location technique captured the faulty statement precisely (less than 5 statements) in all cases. In all applications, the program crash point was used as the single, automatically-inserted *C-point*, hence programmer effort was limited to specifying constraints and indicating allocation sites for the data structures.

Table 4.3: Experience with real world programs.[1](Precision)

1	2	3	4
Program	**Lines of code**	**Data Structure**	**Statements Examined**
ls	3.5K	Linked List	4
403.gcc	365K	Splay Tree	1
464.h264ref	32K	Multidimen. Array	1
Bison	17K	Graph	3

[1] First row in Table 4.2 and Table 4.4 is column numbering.

Table 4.4: Experience with real world programs.[1](Time overhead)

1	2	3	4	5	6
Program	**Program execution time (sec)**			**Const. Match.** **time** (sec)	**Fault Loc.** **time** (sec)
	Original	Null Pin	Instrume.		
ls	0.19	1.10	1.36	0.003	0.05
403.gcc	0.04	5.83	10.76	0.028	0.23
464.h264ref	15.08	38.17	2703.69	0.008	0.29
Bison	0.18	1.48	4.99	0.426	16.48

ls. GNU *ls* [4] lists information about files including directories. The program source code consists of 4,000 lines of C code. It uses a linked list internally to store information about the remaining directories when run in recursive mode. We inserted a bug in the code where the next pointer's value is a non-NULL non-heap address. Due to this bug the program crashes. The violated constraint here is that the next pointer needs to be either a heap address or NULL. As input, we used the GNU coreutils-8.0 source code directory. Our technique traced this fault to 4 statements.

403.gcc. A benchmark program from SPEC CINT 2006 benchmark suite [10], 403.gcc is based on Gcc version 3.2 set to generate code for an AMD Opteron processor. The program uses splay trees, a form of binary search trees optimized to access to recently-accessed elements (splaying is the process of rotating the tree to put a key to the tree root).

To inject a bug, we replaced the right rotate function of the tree by left rotate function. We used the SPEC training input for this experiment, which leads to program crash. The violated constraint here is the binary search tree invariant: *Key(root)* > *Key(left child)* and *Key(root)* < *Key(right child)*. Our method traced the fault to 1 statement.

464.h264ref. This benchmark program from the SPEC CINT 2006 benchmark suite [10] is a reference implementation of Advanced Video Coding, a video compression standard. The benchmark uses a pointer-based implementation of a multidimensional array, with each dimension being a level of a full and complete tree.

To inject a bug we set an internal pointer to NULL which caused a crash. The SPEC test input was used in this experiment. The violated constraint here, based on the OUTDEGREE attribute, is that the tree is full and complete. Our fault location method traced the fault to 1 statement.

GNU Bison. Bison(3.0.4) [1] is a general-purpose parser generator that converts an annotated context-free grammar into a parser. The application uses a graph to store the symbol table where each node is a token or a grammar non-terminal and the node attributes are assigned accordingly.

Our bug injection simulates a programming error where wrong attributes are assigned to nodes, leading to a crash. We used the C language grammar file as input. The violated constraint here is the correctness of the node attributes. Our method traced back the fault to 3 faulty statements.

4.6 Scalability of the Technique

There are two sources of time overhead incurred by our technique: collecting the execution trace and trace back. We now explain why both these slowdowns are unavoidable and we believe the overhead is acceptable. First, dynamic analysis (collecting the execution trace) is inherently slow. Table 4.2 column 7 and Table 4.4 column 6 list the cost of running application under Pin without any instrumentation which itself is a significant slowdown. We only instrument allocation/deallocation calls and references to allocated regions (section 4.4, paragraph 1). This limits instrumentation cost while still capturing data structure faults.

Second, the alternative to automatically search the program execution trace is manually narrowing down the fault by running the application multiple times. We have introduced a number of optimizations to speed up the trace back as explained in section 4.3.

Programmers can use our tool for debugging larger programs by capturing the execution trace for just the relevant sections of program execution hence limiting the search space. As shown in the evaluation, our tool can handle large enough search spaces for practical debugging purposes.

Our technique is easily applicable to parallel programs, as that would only require modifications to the Pin-based tracing mechanism. While tracing a multi-threaded program can potentially increase the overhead, this problem can be abated using selective record and replay, e.g., PinPlay [79].

4.7 Summary

The chapter has presented our fault location framework. The framework provides the user with precise and efficient fault location for bugs violating data structure consistency constraints. We have showed optimizations of *incremental constraint matching, identifying corrupted data structure* and *modification prediction.* These optimization make the process of fault location more efficient. We have compared the precision of our fault location framework with that of *dynamic slicing* and *tarantula*. We have explored the time overhead for fault location and found it to be with in practical limits.

Chapter 5

Efficient Backward Slicing

The fault location approach presented in Chapter 4 tracks the data structure fault back to the faulty data structure mutation. The actual source of the fault in the program may have been an earlier data structure unrelated statement whose execution eventually led to the faulty data structure mutation. We propose the use of dynamic backward slicing for finding this fault statement starting from the faulty data structure mutation.

Dynamic program slicing, introduced by Korel and Laski [56], is widely used for debugging purposes. Other applications of dynamic slicing include software testing and software maintenance. The dynamic backward slice of a computed value is defined to include the executed statements that played a role in the computation of the value. It is computed by taking the transitive closure over data and control dependences starting from the computed value and going backwards over the execution trace. In our technique, the resulting faulty statements from the previous step, i.e., “locate the faulty mutation”, forms the slicing criterion.

The problem with dynamic slicing is its high time overhead. This chapter presents an improved dynamic backward slicing algorithm to address this problem. The improved version compares favorably with the performance of the state of the art dynamic backward slicing presented as a part of *DrDebug* [109] debugging framework.

5.1 Background and Overview

A *dynamic backward slice* is the set of statements that did affect a computed value of a variable at a program point for one specific execution. Given a program execution, the dynamic slice of a variable v at the execution point i_j, is the set of statements that contributed to the value of at that point. An execution point $i_j$is defined by the execution instance j of a statement i. The tuple $< v, i_j >$ is known as the slicing criterion.

5.1.1 Computing the Backward Dynamic Slice

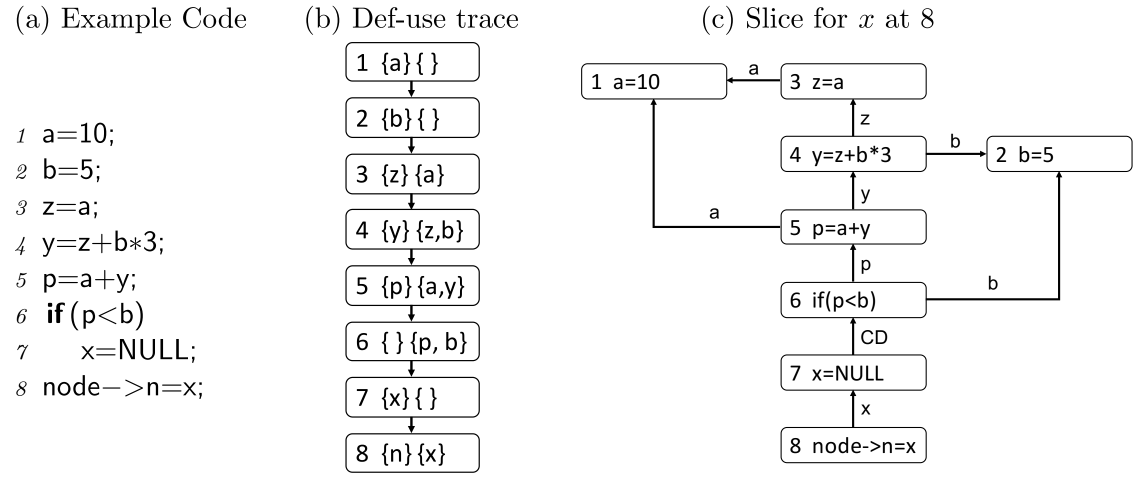
Computing the dynamic backward slice involves two steps. First, the program is executed to collect the *execution trace*. The execution trace contains the complete information about control flow and data flow and has an entry for each executed instruction. In case of multithreaded programs, additional preprocessing is required to create a single global trace which combines the information about all the executed threads. In the second step, a backwards traversal of the trace is performed to recover dependences which should be included in the slice. The traversal starts by initializing a *dependency list* where each entry in the list is a memory reference (address & size, or register). The dependency list is initialized with the variable in the slicing criterion. The information of each instruction in the trace processed in the backward order. The processing of each instruction involved deciding if the instruction is the part of the slice and updating the dependency list. Algorithm 4 gives the processing performed for each instruction. The dependency list is matched against the memory addresses (or registers) defined by the instruction. If any of the memory regions present in the dependency list are defined by the instruction then the instruction is added to the slice. The dependency list is updated according to the def-use information of the instruction. A similar dependency list is maintained for detecting control dependencies. Each entry in the control dependency list is a global instruction id. The check is performed if any instruction present in the control dependency list if control dependent on the current instruction.

Algorithm 4 Backward Traversal of Execution Trace

```
Def[i]: Memory reference and registers defined in instruction $i$
Use[i]: Memory reference and registers used in instruction $i$
MemDepList: Memory Dependency List
ControlDepList: Control Dependency List

function Process(instruction $i$)
    for each definition $d$ in $Def[i]$ do
        if $d \cap$ MemDepList $\neq \phi$ then
            MemDepList = MemDepList $\cap Def[i]$
            MemDepList = MemDepList $\cup Use[i]$
            ControlDepList = ControlDepList $\cup i$
            Slice = Slice $\cup i$
            Break
        end if
    end for
end function
```

We illustrate the process of computing dynamic backward slices using the example from Figure 5.1. The code snippet is shown in Figure 5.1(a). Line 8 sets field `n` of `node` to `NULL`, introducing a fault in the program data structure. Figure 5.1(b) shows the def-use



(a) Example Code

```
1 a=10;
2 b=5;
3 z=a;
4 y=z+b*3;
5 p=a+y;
6 if (p<b)
7     x=NULL;
8 node->n=x;
```

(b) Def-use trace

(c) Slice for x at 8

Figure 5.1: Dynamic backward slicing

information trace for the execution. For example, line 4 defines y by using z (defined on line 3) and using b (defined on line 2). We compute the backward slice by doing a backward traversal over the execution to determine which executed statements have to included in the slice. The slice for x at line 8 is shown in Figure 5.1(c). All the executed statements which have affected the value of x on line 8 are part of the slice.

5.2 Complexity Analysis for Slicing

The backward traversal of the trace is the most significant part of backward slicing in terms of time cost. We are concentrating here on the complexity of the backward traversal of the trace. Figure 5.2 gives the complexity analysis for the backward traversal of execution trace. The cost of processing each instruction is matching each memory reference defined in the instruction with the elements in the dependency list (line 1). The cost of the complete backward traversal is the sum of the cost of processing individual instructions. In the worst case, the size of the dependency list can get incremented for every precessed instruction, i.e., a $x = x + y$ type of instruction will increment the size of dependency list. In such a case, the cost of complete traversal will be linear summation over i where i varies from 1 to n (n is the total number of instructions in the trace). This makes the worst case complexity O(n^2). The complexity of traversal can be improved by maintaining a sorted dependency list and preforming binary search. The modified complexity of traversal will be O(n log n).

1. Cost of processing 1 instruction = no. of memory references in instruction * length of dependency list
2. Cost of processing n instruction = $\sum_{i=1}^{i=n}$ (cost of processing single instruction i)
   = $\sum_{i=1}^{i=n}$ ($M*$ length of dependency list for i)
   (M is a constant denoting average number of memory references defined per instruction)

In worst case, each instruction can add an entry to dependency list

3. Cost of processing n instruction = $M * \sum_{i=1}^{i=n} i$
   = $M * n(n+1)/2$
   = O(n^2)

Figure 5.2: Complexity analysis for backward traversal.

5.3 Improved Slicing Algorithm

In this section we introduce an improved backward traversal which reduces the average case complexity. We observed that during the backward traversal most of the instructions we encounter are not a part of the slice. Based on this observation we designed a new stage in the backward traversal where we perform a lightweight check to determine if an instruction is part of the slice. If the instruction passes the test then we perform the heavyweight check against the complete dependency list for the instruction.

The lightweight filtering stage is based on using the minimum information required for determining if an instruction is part of the slice. In case of data dependency, the filter is a binary search tree with memory references as node. We also coalesce contiguous memory ranges which reduces the number of elements in the filter. Figure 5.3 shows an example source code. If we start with the slicing criterion of z@4, by the time the backward traversal reaches line 1, the length of dependency list will increase to a million elements while the number of elements in the filter will remain 1.

```
1 x = u;
2 for(i = 0; i <= 1000000; i++)
3         y = y + a[i];
4 z = y;
```

Figure 5.3: Example code for comparison of dependency list size and filter size.

In case of control dependency, the filter is a hash table with instruction ids as elements. The cost of determining if an instruction is a part of the slice due to control dependency comes out to be constant.

Algorithm 5 gives the modified algorithm for the backward traversal of the execution trace. The filter and the dependency lists are initialized with the variable in the slicing criterion. For each instruction, the filter test is performed. The filtering out of instruction which are not a part of the slice reduces the cost of slicing.

5.4 Evaluation

We compared the updated backward slicing algorithm to that presented in DrDebug [109]. DrDebug presents the state of the art implementation for computing dynamic

Algorithm 5 Updated Backward Traversal

```
Slicing Criterion: $v@j$
$MemFilter$ $=\{v\}$
$ControlFilter$ $=\{j\}$

function Filter(instruction $i$)
    for each definition $d$ in $Def[i]$ do
        if $d \cap MemFilter \neq \phi$ then
            Process($i$)
            Update the filters
            Goto: END
        end if
    end for
    if $i \cap ControlFilter \neq \phi$ then
        Process($i$)
        Update the filters
    end if
    END
end function
```

backward slice. Both algorithms were implemented as a part of same framework. For evaluation we use 7 PARSEC [18] benchmarks version 2.1 run of the *native* input. The goal of the evaluation is to compare the performance of the two algorithm under different workloads. The evaluations were done on a pool of machines with 16 Intel Xeon ("Sandy Bridge E") processors (hyper-threading OFF) and 128GB of physical memory running SUSE Linux Enterprise Server 10.

Table 5.1 compares the two slicing algorithms for a trace formed by regions of length 1 million instructions in the main thread. The total instructions in the trace (column 2) from all threads were 3–4 times more than the length in the main thread. The last 5 memory reads have been used as the slicing criterion for the evaluation. Column 3 presents the slicing criterion number in reverse order i.e., criterion no. 1 is the last memory read in the trace. Column 4 shows the time taken (in seconds) by backward slicing algorithm

1	2	3	4	5	6
Benchmark	Instruction	Criterion	Slicing Time(seconds)		Speed-up
	Count	#	DrDebug	Improved Slicing	(4/5)
blackscholes	4117570	1	95.7	21.7	4.41
		2	17.3	18.7	0.92
		3	19.3	20.0	0.96
		4	17.2	33.6	0.51
		5	19.4	19.2	1.01
bodytrack	4439187	1	3888.5	128.8	30.19
		2	3550.6	564.9	6.28
		3	2412.7	184.7	13.06
		4	4924.7	135.9	36.20
		5	4459.5	122.0	36.55
fluidanimate	4296790	1	459.5	772.4	0.59
		2	480.2	637.0	0.75
		3	416.0	753.8	0.55
		4	488.1	694.5	0.70
		5	462.1	777.9	0.59
swaptions	4850013	1	23.3	34.1	0.68
		2	92.5	17.7	5.22
		3	8.6	14.8	0.58
		4	1023.6	255.5	4.00
		5	1542.0	291.2	5.29
vips	999998	1	109.0	5.8	18.79
		2	106.1	5.1	20.80
		3	124.5	5.4	23.05
		4	109.0	4.7	23.19
		5	106.5	5.3	20.09
dedup	26756781	1	1429.5	1966.1	0.72
		2	65.3	132.7	0.49
		3	67.7	120.4	0.56
		4	75.6	256.0	0.29
		5	49168.5	12710.4	3.86
streamcluster	3917850	1	72.3	24.5	2.95
		2	66.2	41.9	1.57
		3	81.7	43.1	1.89
		4	64.5	32.2	2.00
		5	59.6	24.0	2.48

Table 5.1: Comparison of slicing time for **1 million instruction per thread** program runs, PARSEC benchmarks.

1	2	3	4	5	6
Benchmark	Instruction	Criterion	Slicing Time(seconds)		Speed-up
	Count	#	DrDebug	Improved Slicing	(4/5)
blackscholes	42688922	1	13940.0	3069.3	4.54
		2	2365.3	1135.6	2.08
		3	2196.2	1319.6	1.66
		4	2494.7	1389.3	1.79
		5	2441.1	1394.6	1.75
bodytrack	41400603	1	228979.8	54432.3	4.20
		2	–	51883.6	–
		3	–	65629.1	–
		4	–	48718.9	–
fluidanimate	32613170	1	–	6886.8	–
		2	–	12747.7	–
		3	–	6776.6	–
		4	–	8207.5	–
		5	–	7009.5	–
swaptions	40719219	1	–	101.7	–
vips	10000791	1	–	5827.2	–
		2	–	2273.9	–
		3	–	5712.2	–
		4	–	2259.4	–
		5	–	6068.4	–
dedup	172615758	1	–	5250.8	–
streamcluster	40148501	1	123.6	192.2	0.64
		2	8090.0	4074.7	1.98
		3	8821.2	4037.4	2.18
		4	8725.9	3742.7	2.33
		5	8463.4	4424.9	1.91

Table 5.2: Comparison of slicing time for **10 million instruction per thread** program runs, PARSEC benchmarks.

present in DrDebug while column 5 shows the time taken by the improved slicing algorithm. The last column shows the relative speedup.

The results show that the new algorithm performs significantly better in case of longer running timings. The unoptimized algorithm performs better in case of shorter slicing timings. The explanation for this observation is that the cost of maintaining and updating the filter more than offsets the performance gains of using the filter in case of shorter runs. The improved algorithm consistently performs better for longer runs. This argument is further supported by the results in Table 5.2.

Table 5.1 compares the two slicing algorithms for a trace formed by regions of length 1 million instructions in the main thread. The blank cells represent that the corresponding algorithm was unable to complete slicing due to time or memory overrun. The results clearly show that the improved algorithm is able to handle longer traces. The improved slicing algorithm gives a speed up of 1.5x to 4.5x for 10M instruction runs.

5.5 Summary

This chapter has presented an improved backward slicing algorithm which optimizes the backward execution trace traversal for better performance. The algorithm is based on the idea of adding a relatively low-cost filtering stage to backward traversal. The low cost filtering stage discards the instructions which are not a part of slicing without performing the actual slicing on the instructions. We have evaluated our technique on 7 PARSEC [18] benchmarks and found that the algorithm significantly outperforms the unoptimized version for longer program runs.

Chapter 6

Linearizability Verification of Concurrent Data Structures

Linearizability [47] is the standard form of correctness for concurrent data structure implementations. Linearizability means that the effect of each operation on the concurrent data structure is atomic. A more technical definition of linearizability states that each concurrent execution must be equivalent to some sequential execution of operations of the abstract data structure while preserving the order of non-overlapping operation.

This chapter presents a technique for linearizablity verification that consists of (1) a specification language that allows concurrent data operations to be specified simply as sequences of sub-operations where each sub-operation may or may not carry a precondition, and (2) a static checker that, given the relationship between the sub-operations, determines if the implementation is linearizable. The user specifies the operations of concurrent data structure implementation as a sequence of atomic sub-operations in our specification language. Our static checker returns true if the implementation is linearizable.

6.1 System Model and Linearizability

A concurrent data structure *implementation* consists of a shared state (defined by shared variables) and methods which operate on the shared state. An *execution* consists of a variable number of threads, each executing one of the defined methods. An *operation* is a successful execution of a method. The concept of operations is native to linearizability of concurrent implementations. Most of the prior work on concurrent data structures with fixed linearization points establish the linearizability of the data structure by locating the linearization point of the operations present in the implementation. For example, Michael and Scott [73] have proven their queue's linearizability by locating the linearization points of enqueue, dequeue_empty, and dequeue_non_empty operations. Hendler et al. [46] have also used the operations' linearization points for proving linearizability.

Operations are sequential compositions of atomic sub-operations. Each sub-operation (atomic) α has the form $< g > t$, where g is a pre-condition to the sub-operation. A sub-operation can execute only at a state which satisfies g, while t is the set of reads and writes which get executed when α gets executed. Sub-operations can be *global* or *local.* A global sub-operation involves reading or writing shared variables (or references). Local sub-operations are used just for the sake of completeness and do not play any role in proving linearizability. Section 6.3 introduces a detailed language along with examples to help users express concurrent data structures implementations as a sequence of sub-operations.

6.1.1 Execution Model

We assume a sequentially consistent memory model. The program state s is the current valuation of the variables (both shared and local) present. The program state changes with execution of sub-operations belonging to operation instances. Each operation instance is a unique invocation of one of the operations defined in the specification. Each operation instance has a unique operation instance id from the set O_{id}. We represent sub-operation $\alpha(< g > t)$ being executed as a part of operation instance v as $\alpha[v]$; $g[v]$ and $t[v]$ represent the corresponding pre-condition and sub-operation body. Note that we do not use *thread id* instead of operation instance id because more complicated data structures ([46,73]) can have an operation execution distributed among multiple threads. The atomicity and error conditions are given in Figure 6.1. We use the notation $(s_1, H_1, \alpha) \rightarrow (s_2, H_2)$ to indicate that sub-operation α executed at program state s_1 takes the program state to s_2 while the execution history changes from H_1 to H_2. The definition `ATOMIC` states that if the current program state s_1 satisfies the sub-operation pre-condition $g[v]$, $(v \in O_{id})$, represented by $s_1 \models g[v]$, the program state is modified with transition $t[v]$. The execution history H_1 gets appended with sub-operation $\alpha[v]$. The `ERROR` definition states that the sub-operation α cannot successfully execute at a program state where the corresponding pre-condition is not satisfied.

$$\boxed{(s_1, H_1, \alpha) \rightarrow (s_2, H_2)}$$

$$\texttt{ATOMIC}\ \frac{s_1 \models g[v] \qquad (s_1, t[v]) \rightarrow s_2}{(s_1, H, \alpha[v]) \rightarrow (s_2, H.\alpha[v])}$$

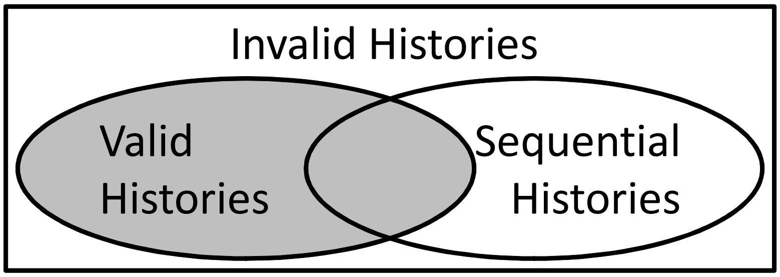
$$\texttt{ERROR}\ \frac{s_1 \models \neg g[v]}{(s_1, H, \alpha[v]) \rightarrow (\text{error}, H)}$$

Figure 6.1: Atomicity and error definitions.

6.1.2 Histories

Definition 1. A *history* H is a finite sequence of sub-operations where the sub-operations corresponding to the same operation instance follow the program order.

A history H is *sequential* if the sub-operations from the same operation instance occur together. A sequential history is *legal* if it follows the abstract data structure behavior. A history H in which the preconditions for all the sub-operation instances present in the history are satisfied is *valid* with respect to the implementation (naturally, we call "invalid" a history that is not valid). The following Venn diagram shows the relationship between valid, invalid and sequential histories.



Two valid execution histories are *equivalent* if they contain the same sub-operation instances and the program state visible to a sub-operation instance is the same in both the histories.

6.1.3 Linearizability

Linearizability requires every execution history to be equivalent to some legal sequential history that preserves the order of non-overlapping operations in the original history.

Many techniques are available for checking the correctness of a sequential implementation with respect to the abstract data structure (e.g., [85]). Taking advantage of this we make the following safe assumption in our technique:

Assumption 1. Every valid sequential history with respect to the implementation is a legal sequential history.

In other words, we assume that all sequential executions of the implementation are correct. With this assumption, our linearizability definition is:

Definition 2. An implementation is *linearizable* if for any valid history there exists some equivalent valid sequential history such that the order of non-overlapping operations in the two histories is the same.

6.2 Overview and Example

We illustrate our technique on the MS non-blocking queue [73]: a singly linked list-based queue with three operations, initialize , enqueue, and dequeue (Figure 6.2). Each queue node has next and value fields. Enqueue first allocates a new node then reads the tail pointer and sets the next pointer of the end node to point to the newly allocated node; the final step in the process is to set the tail pointer to the new node. Any thread operating on the list can set the trailing tail pointer. Dequeue reads head and tail pointers then checks if both point to the same node. If head and tail point to the same node, i.e., the queue is empty, dequeue returns false; otherwise, dequeue reads the head node's value and updates the head pointer.

```
typedef struct Node_t * Node
struct Node_t{
int value;
Node next;
}
struct Queue{
Node Head, Tail;
} * Q;

function INITIALIZE(Q: pointer to queue)
    node = new_node()
    node→next = NULL
     Q→Head = Q→Tail = node
end function

function ENQUEUE(Q:pointer to queue, value:int)
    node = new_node();
    node→value = value;
    node→next = NULL;
    loop
    tail = Q→Tail;
    next = tail→next;
    if tail == Q→Tail then
        if next == NULL then
            if CAS(&tail→next, next, node) then
                break;
            end if
        else
            CAS(&Q→Tail, tail, next);
        end if
    end if
    endloop
    CAS(&Q→Tail, tail, node);
end function
```

```
function DEQUEUE(Q:pointer to queue, pvalue:int)
     loop
    head = Q→Head;
     tail = Q→Tail;
    next = head→next;
    if  head == Q→Head then
        if head == tail then
            if next == NULL then
                return FALSE;
            end if
             CAS(&Q→Tail, tail, next);
        else
            pvalue = next→value;
            if CAS(&Q→Head, head, next) then
                break;
            end if
        end if
    end if
    endloop
    free(head)
    return TRUE;
end function
```

Figure 6.2: Michael and Scott non-blocking concurrent queue [73].

1. node X = new_node(); 2. node tail = Q→tail; 3. node next = tail→next; 4. assume (tail == Q→tail); 5. assume (next == NULL); 6. CAS(tail→next, next, X); 7. CAS(Q→tail, tail, X);	2. node tail = Q→tail; 3. node next = tail→next; 4. assume (tail == Q→tail); 5. assume (next == NULL); 6. <tail→next == next> tail→next = X; 7. <Q→tail == tail> Q→tail = X;

node tail = Q→tail;
assume (tail == Q→tail);
<tail→next == NULL>
tail→next = X;
<Q→tail == tail>
Q→tail = X;
	<Q→tail = tail; tail→next == NULL>
	tail→next = X;
	<Q→tail == tail>
	Q→tail = X;

Figure 6.3: Expressing an operation as a sequence of atomic sub-operations.

6.2.1 Specifying Concurrent Operations

The user specifies the concurrent data structure's operations as a sequence of atomic sub-operations. Let us consider the *enqueue* method; the `loop` in the method leads to an unbounded number of execution paths. We leverage the notion of pure loops ([40,107]) to transform the loop to its last iteration. Figure 6.3 column 1 shows one of the possible loop free execution paths of the *enqueue* method. Figure 6.3 column 2 shows CAS (Compare and Swap) replaced with corresponding sub-operation $< g > t$ format. The *next* var in line 5 and 6 is a local variable i.e., cannot be modified by other threads. Replacing the value of *next* with NULL gives us column 3. The only possible modification to Q→tail in the program is setting the *next* node in the list. The value of Q→tail does not change between 4 and 6 because that would violate the pre-condition tail→next == NULL. Adding this pre-condition to the sub-operation gives us column 4. Previous works on atomicity verification ([39, 40, 43, 59, 107]) have detailed discussion about program transformations required to express operations as sequences of atomic sub-operations.

```
T1.    struct node{int value, node next};
T2.    struct queue{node head, node tail};
G1.    queue Q ;
Enqueue (int val)
E1.    node X;
E2.    node Z;
E3.    1 X = malloc node;                                        ...a
E4.    1 X.value = val;                                          ...b
E5.    1 X.next = NULL;                                          ...c
E6.    1 < Q.tail == Z;  Z.next == NULL;>
             Z.next = X;                                         ...d
E7.    * < Q.tail == Z>
             Q.tail = X;                                         ...e
Dequeue_nonempty()
D1.    node X;
D2.    1 < Q.head == X;  Q.tail ≠ X;   X.next≠ NULL;>
             Q.head = X.next;                                    ...f
D3.    1 free X;                                                 ...g
Dequeue_empty()
DE1.   node X;
DE3.   1 < Q.head == X;   Q.tail == X;   X.next == NULL;>
```

Figure 6.4: MS non-blocking queue [73] specification.

Figure 6.4 contains the MS queue specification in our language. The user specifies data structures by declaring their type name along with the fields. For example, line T1 declares a type *node* which has two fields and line G1 declares a shared variable **Q** of type *queue*. There are two different *dequeue* specifications, *Dequeue_nonempty* and *Dequeue_empty*, which correspond to the sequence of sub-operations for dequeue on a non-empty queue and an empty queue, respectively. Figure 6.5 shows the global sub-operations that involve shared variables, i.e., the global sub-operations of MS queue.

6.2.2 Pairwise Ordering and Reversibility

An ordered pair of sub-operations cannot be part of a valid history if the execution of the first sub-operation destroys the precondition for the second one. For example, sub-operation d in Figure 6.4 line E6 cannot be followed by another instance of d because the

Global sub-ops for Enqueue and Dequeue

```
Enqueue()
1 < Q.tail == Z;   Z.next == NULL;>
    Z.next = X;                          ...d
* < Q.tail == Z;>
    Q.tail = X;                          ...e
```

```
Dequeue_nonempty()
1 < Q.head == X;   Q.tail ≠ X;   X.next≠ NULL;>
    Q.head = X.next                      ...f

Dequeue_empty()
1 < Q.head == X;  Q.tail == X;  X.next == NULL;>      ...h
```

	Pair-wise ordering	Non-Boundary Cases	Pair-wise reversibility
$\neg d \wr d$	$d \wr f$ [if($Q.head \neq Q.tail$)]	d, f	$d \ominus f$
$\neg d \wr e$	$e \wr d$ [Boundary case]	f, e	$f \ominus e$
$\neg d \wr h$	$e \wr f$ [Boundary case]		
$\neg e \wr e$	$f \wr d$ [Boundary case]		
$\neg e \wr h$	$f \wr f$ [Boundary case]		
$\neg h \wr e$	$f \wr h$ [Boundary case]		
$f \wr e$	$h \wr d$ [Boundary case]		
$\neg h \wr f$	$h \wr h$ [Boundary case]		

Figure 6.5: Proving the MS queue linearizable.

first instance sets the tail→next pointer to a non-null value, invalidating the precondition for the second instance. Figure 6.5 shows all the pairs of sub-operations and their feasibility. In our approach, $a \wr b$ means sub-operation a does not destroy the precondition for b. The sub-operation execution order in a feasible (pair-wise orderable) pair can be changed if changing the order will not affect the value of shared or local variables. We call this *reversibility*, denoted as $a \ominus b$. The properties are explained in detail in Section 6.4.

A boundary case refers to a pair of sub-operations that includes the last sub-operation of an operation followed by the first sub-operation of an operation. Pair-wise orderable non-boundary pairs lead to interleaving operations. Boundary pairs are always defined as non-reversible. Pairwise reversibility for non-boundary sub-operation pairs for MS queue is also defined in Figure 6.5.

6.2.3 Trace Transformation

An execution history formed by moving sub-operations using the reversibility property is equivalent to the original history. A valid concurrent history can be mapped to a valid sequential history using this trace transformation. The order of non-overlapping operations always remains the same during this process because the boundary pairs are defined to be non-reversible. For example, consider a valid history for the MS queue:

$$\begin{gathered} ..., d_1, f_2, f_3, e_1, ... \\ \equiv ..., d_1, f_2, e_1, f_3 ... \text{ (using } f_3, e_1 \equiv e_1, f_3 \text{)} \\ \equiv ..., d_1, e_1, f_2, f_3 ... \text{ (using } f_2, e_1 \equiv e_1, f_2 \text{)} \end{gathered}$$

For a linearizable implementation, every valid concurrent history can be mapped to an equivalent valid sequential history using trace transformation. Our linearizability checker in Algorithm 6 answers the question: "Given an ordering and reversiblity specification, can all the valid histories (for unbounded number of concurrent operation instances) be mapped to an equivalent sequential history using trace transformation?"

6.3 Specification Language

We have designed a language to assist the user in expressing concurrent data structure operations in terms of sub-operations. Note that the language is not central to our technique — our technique will work as long as concurrent operations can be expressed as sub-operations with ordering and reversibility properties. That said, we found that the language made the task of expressing sub-operations and finding the pair-wise ordering and reversibility very easy. In this section we first define the language and then demonstrate its use for expressing common features in concurrent implementations.

Specification	sp	$::=$	$stList\ g\ opList \mid g\ opList$
Struct Decl	$stList$	$::=$	$stList\ st \mid st$
	st	$::=$	$\mathsf{struct}\ sname\ \{l\}\ ;$
	l	$::=$	$l, type\ fname \mid type\ fname$
Type	$type$	$::=$	$\mathsf{int} \mid sname$
Global Decl	g	$::=$	$dList$
Operation	$opList$	$::=$	$opList\ op \mid op$
	op	$::=$	$opName\ opBody$
	$opName$	$::=$	$oname(dList\ ) \mid oname$
	$opBody$	$::=$	$dList\ sList$
	$sList$	$::=$	$sList\ subOp \mid subOp$
	$dList$	$::=$	$dList\ decl \mid decl$
Declaration	$decl$	$::=$	$type\ var\ ; \mid type\ var[\,]\ ;$
	$subOp$	$::=$	$tmark\ subOpBody$
		$\mid$	$tmark\ < cList >\ subOpBody$
		$\mid$	$tmark\ < cList >$
	$tmark$	$::=$	$\mathsf{1} \mid *$
Pre-condition	$cList$	$::=$	$cList\ condition\ ; \mid condition\ ;$
	$condition$	$::=$	$lhs\ rop\ rhs$
	$subOpBody$	$::=$	$subOpBody\ stmnt\ ; \mid stmnt\ ;$
	$stmnt$	$::=$	$var\ = \mathsf{malloc}\ type \mid \mathsf{free}\ var$
		$\mid$	$lhs\ =\ rhs$
		$\mid$	$\mathsf{READ}\ lhs$
LHS	lhs	$::=$	$var \mid var\ .\ fname \mid var[index]$
RHS	rhs	$::=$	$\mathsf{ID} \mid n \mid \mathsf{NULL} \mid lhs$
	$index$	$::=$	$n \mid var$
Operators	rop	$::=$	$== \mid \neq \mid \leq \mid \geq \mid < \mid >$
Integers	n		
Variable	var		
Struct Name	$sname$		
Field Name	$fname$		
Op Name	$oname$		

Figure 6.6: Syntax of specification language.

6.3.1 Syntax

We provide the user with a simple C-like syntax, shown in Figure 6.6. Specifications consist of an optional list of structure declarations *st*, global declarations *g* and a list of operations *op*. Each global declaration consists of a variable name *var* and its type. Types can be **int** or **struct**, where **struct**s have a name *sname* and consist of a list of fields; each fields has a name *fname* and a *type*.

Operations. Each operation *op* has a name *oname*, and an optional argument (*type var*). Operation bodies *opBody* consist of local variables, *dList*, and sub-operations, *sList*.

Sub-operation. A sub-operation specification has a **tmark** followed by a pre-condition and a sub-operation body. *tmark* marks the thread which executes the statement. A *tmark* of **1** means that the thread invoking the operation instance will perform the sub-operation. If a *tmark* is *, it indicates that any thread can perform the sub-operation. Specifying *tmark* for each sub-operation allows specifying an operation which is distributed across multiple threads.

– An optional precondition to the sub-operation is composed of a list of *condition*s, where each condition is a relational expression involving a variable **var**, or field **var.fname** on lhs and null, constant, variable, or field on rhs.

– A statement can be allocation or deallocation; malloc and free statements are used for specifying local sub-operations. Other possible statement forms involve assigning values to a variable or writing a field. **READ** refers to reading of a field or a variable.

– We use **ID** to identify the operation instance; ID is useful in modeling locks.

6.3.2 Modeling Synchronization Primitives

We now show how to model compare-and-swap (CAS), fetch-and-increment (F&I), and locks, using our language.

Pseudocode	him=collision[pos];
	while(!CAS(&collision[pos],him,mypid))
	  him=collision[pos];

Specification	1 him = collision[pos];
	  collision[pos] = mypid;

Figure 6.7: Sample CAS specification from [46].

Modeling Compare-and-swap. CAS can be modeled in two different ways, depending on the implementation.

Case 1 - When the memory location being compared was read or written before CAS statement and execution of the CAS statement is conditionally controlled by an *if statement*. In this case the CAS statement can be modeled as a sub-operation with the *if condition* as the precondition and the memory write as the body of the sub-operation. For example, in Figure 6.2, the execution of the CAS statement on line 10 is dependent on the *if conditions* in lines 8 and 9. The specification for the CAS statement is as follows:

```
1 < Q.tail == Z;   Z.next == NULL;>
            Z.next = X;
```

Case 2 - When the memory location being compared was read or written before CAS statement and execution of CAS statement is unconditional. In this case CAS statement can be modeled as a sub-operation with no precondition and two-statement body: the first statement reads/writes the location being compared while second statement writes the memory location. Figure 6.7 shows a pseudocode excerpt from the implementation of elimination back-off stack by Hendler et al. [46]. The second row shows corresponding sub-operation specification.

Modeling Fetch-and-Increment. F&I on a shared variable x is modeled by a sub-operation with no precondition. The body of the sub-operation consists of two statements, first reading shared variable x in a local variable and second incrementing x. Figure 6.8 left shows the pseudocode from the *enqueue* operation in Herlihy and Wing's queue [47] which uses fetch-and-increment. The right side shows the specification in our language — declaration of variables *back* and *AR* and the *enqueue* operation.

```
enqueue(lv){
/*(Fetch and Increment)*/
(k, back) := (back, back
+1);
AR[k] := lv;}
```

Pseudocode

```
int AR[ ];
int back;

Enqueue(int lv)
int k;

1 k = back;
  back = back + 1;

1 AR[k] = lv;
```

Specification

Figure 6.8: Sample Fetch-and-increment specification from [47].

Modeling Locks. A lock can be modeled using our specification language as a shared variable with a default value. The locking sub-operations will look like:

```
1 < lock == default_value; >

      lock = ID;
```

where operation ID refers to a unique value identifying the operation instance. Any sub-operation performed with the lock acquired will have a precondition of the form:

$$< \text{lock} == \text{ID};>$$

Unlocking takes the form:

$$1<\text{lock} == \text{ID};>$$

$$\text{lock} = \text{default\_value};$$

6.4 Proving Linearizability

The number of possible valid histories for an implementation is directly proportional to the number of executing concurrent operation instances. The number of valid histories becomes intractable with the increase in the number of concurrent operation instances executing on the data structure. We solve the problem of intractable number of histories by breaking the history down into basic building blocks, *sub-sequences of two sub-operations.* We then argue about all the histories in terms of properties on these building blocks. We call (a, b) a *sub-operation pair*, where a and b *belong to different operation instances*; for brevity, we will heretofore drop the parentheses when referring to pairs. Note that a and b can be the same sub-operation. The number of possible pairs for an implementation with n sub-operations is n^2.

We define the property of pair-wise ordering as follows:

Definition 3. Figure 6.9 shows how we determine if a sub-operation pair is orderable. The rule states that a sub-operation pair α_1, α_2 is *not pairwise orderable* (denoted by $\neg\alpha_1 \wr \alpha_2$) if after executing α_1, the pre-condition for α_2 is not met; otherwise $\alpha_1 \wr \alpha_2$.

$$\frac{\forall s, \forall u, v \in O_{id}, u \neq v \qquad (s, H, \alpha_1[u]) \rightarrow (s_2, H_2) \qquad (s_2, H_2, \alpha_2[v]) \rightarrow (\text{error}, H_2)}{\neg \alpha_1 \wr \alpha_2}$$

Figure 6.9: Pair-wise ordering.

$$\begin{array}{c}\forall s, \forall u, v \in O_{id}, u \neq v \\ \dfrac{\dfrac{\alpha_1 \wr \alpha_2}{\begin{array}{c}(s, H, \alpha_1[u]) \rightarrow (s_2, H_2) \\ (s_2, H_2, \alpha_2[v]) \rightarrow (s_3, H_3)\end{array}} \qquad \dfrac{\alpha_2 \wr \alpha_1}{\begin{array}{c}(s, H, \alpha_2[v]) \rightarrow (s_2', H_2') \\ (s_2', H_2', \alpha_1[u]) \rightarrow (s_3', H_3')\end{array}} \qquad s_3 = s_3'}{\alpha_1 \ominus \alpha_2}\end{array}$$

Figure 6.10: Pair-wise reversibility.

For example, consider the sub-operation d of MS queue's *enqueue* (Figure 6.5):

```
1 < Q.tail == Z;    Z.next == NULL;>

Z.next = X;
```

The pre-condition states that Q→tail→next should be NULL. Now consider the pair d, d (d_1, d_2 for clarity). The execution of d_1 sets the value of Q→tail→next to a non-NULL value. This invalidates the pre-condition for sub-operation instance d_2. Hence d, d is not pair-wise orderable; i.e., $\neg d \wr d$. Figure 6.10 states the rule for determining sub-operation pair's reversibility.

Definition 4. A sub-operation pair α_1, α_2 is *reversible* (denoted by $\alpha_1 \ominus \alpha_2$) iff

1. $\alpha_1 \wr \alpha_2$ and $\alpha_2 \wr \alpha_1$ and

2. Given a program state, the final program state is the same irrespective of the order of execution of α_1 and α_2.

Note that pair reversibility is different from *moverness* properties (left movers and right movers). The left and right mover properties of an atomic sub-operation are very restrictive as the sub-operation should be commutative with respect to every sub-operation present in the program. Reversibility on the other hand is a property on a single sub-operation pair. [59] has discussed that *movers* fail to prove atomicity in the presence of the ABA problem [25]; reversibility, on the other hand, enables our technique to handle the ABA problem.

Pairs can also be *conditionally orderable*. For example, consider the pair d, f, (Figure 6.5) where sub-operation d is:

$$1\!< \text{Q[tail]} == \text{Z}; \quad \text{Z.next} == \text{NULL};\!>$$

$$\text{Z.next} = \text{X};$$

and sub-operation f is:

$$*\!< \text{Q.head} == \text{X}; \quad \text{Q.tail} \neq \text{X}; \quad \text{X.next}\neq \text{NULL}\!>$$

$$\text{Q.head} = \text{X.next};$$

$d \wr f$ only if Q.head≠Q.tail.

We consider a pair orderable, if it is orderable for any possible values of the variables involved.

The reversibility of a *conditionally orderable pair* is defined under the same conditions for which it is orderable. For example, reversibility for pair d, f (mentioned above), will be calculated with the premise that Q.head≠Q.tail. Reversibility of a pair is decided conservatively. If the reverse order of execution is not equivalent to the original pair under any possible condition (which is satisfied by the ordering condition), we deem the pair to be non-reversible. We found that the conservative definition of reversibility is not sufficient to handle complex interactions of operation instances. We explain in Section 6.5 how to handle such complex cases.

In order to preserve the order of non-overlapping operations during trace transformation, we have used the following simple technique. A pair formed by last sub-operation of any operation and the first sub-operation of any operation (a.k.a boundary pair) is always set to be non-reversible. This ensures that the order of non-overlapping operations in a history will not change when moving around the sub-operations using reversibility property. Given the pairwise ordering and pairwise reversibility of all possible pairs of sub-operations, we can now define valid histories.

Definition 5. A *valid history H* (sequence of sub-operations following program order) is defined as follows:

1. For a sub-sequence a, b of H, either a, b belong to same operation instances, or $a \wr b$ and

2. For any sub-sequence a, b of H, where a and b belong to different operation instances, if $a \ominus b$ then the history formed by reversing the order of a and b in H is also a valid history.

The equivalence of histories is defined in terms of pairwise reversibility:

Definition 6. Two valid histories H and H' are *equivalent*, denoted H $\equiv$ H', iff H' can be formed from H by reversing the pairs present in H using pairwise reversibility.

Using this definition of equivalence of history, we redefine our problem of checking linearizability as follows:

Definition 7. An implementation is *linearizable* iff all valid histories with respect to the implementation can be mapped to some sequential history by changing the order of pairwise reversible sub-operations.

Note that the order of non-overlapping operations will always be preserved because boundary pairs are not reversible.

Algorithm 6 Checking Linearizability

```
Input: $S$: Set of Operations
Sub(S):Set of all sub-operations
first(S): set of first sub-operations for each operation in S
I is the set of all possible prefix sequences for operations
next(x), x$\in$ I : next sub operation in the operation after x
x$\in$ I, y$\in$I$\cup$Sub(S), x.y: sequence formed by concatenating y after x
x$\in$ I, y$\in$I$\cup$Sub(S), x$\wr$y iff x.y is a valid history
x$\in$ I, x is a proper prefix, y$\in$I$\cup$Sub(S), x$\ominus$y iff x.y $\equiv$ y.x
x$\in$ I, x is a complete operation, y$\in$Sub(S), y$\notin$first(S), x$\ominus$y iff x.y $\equiv$ y.x
x$\in$ I, x is a complete operation, y$\in$I, $\neg$x$\ominus$y
x$\in$ I, x is a complete operation, y$\in$Sub(S), y$\in$first(S), $\neg$x$\ominus$y
function CHECK_LINEARIZABILITY()( )
    ret = TRUE
    for all x$\in$ I, x is a proper prefix do
        ret = ret && Check(x)
    end for
    RETURN ret
    }
end function
function CHECK(X)( )
    for all y $\in$ Closure(x) do
        if y$\wr$next(x) && $\neg$y$\ominus$next(x) then
            RETURN FALSE
        end if
    end for
    RETURN TRUE
end function
function CLOSURE(U)( )
    $C = \{\phi\}$
    for all w$\in$ I; st $u \wr w$ && $\neg u \ominus w$ do
        $C = C \cup w$
    end for
    for all $z \in C$ do
        for all w$\in$ I; st $z \wr w$ && $z \ominus w$ do
            $C = C \cup w$
        end for
    end for
    RETURN $C$
end function
```

Algorithm 6 presents our linearizability checking approach. The input to the algorithm is the set of operations, corresponding sub-operation sequences, pair-wise ordering as well as reversibility for each possible pair. We start by initializing set **I** with all prefixes of operations (line 2). The prefix of an operation o is a partial sequence of sub-operations starting from the first sub-operation of o following program order. The prefix set for an operation represented by a, b, c (where a, b and c are the sub-operations) is $\{a$, ab, $abc\}$.

We define the ordering for a prefix pair (x, y — x and $y \in \mathbf{I}$) simply by checking if the concatenated sequence of x and y is a valid history with respect to the input ordering and reversibility specifications (line 5). Reversibility for a prefix pair (x, y — x and $y \in \mathbf{I}$) is defined by the equivalence of two histories formed by reversing the order of x and y (line 6). The ordering and reversibility between a prefix sequence and single sub-operations is defined in a similar manner (lines 5,6,7). Lines 8 and 9 state that two non-overlapping operations are non-reversible.

The closure of a prefix u (Closure(u)) is the set of all prefixes v that can follow u in a valid history (lines 25-35) and the order of execution of u and v cannot be reversed. Complexity of calculating Clousure(u) is O(n^2), where n is the total number of prefixes. Our algorithm checks if for each proper prefix x, for each prefix y that can occur in-between x and next(x), the prefix y can be moved before x or after next (x) in the sequence using the trace transformation (using property of reversibility) (lines 18-24). If the check fails for any prefix $x \in \mathbf{I}$, the algorithm returns false otherwise it returns true.

The algorithm returning true means all possible valid histories (for the input specification) are equivalent to some valid sequential history with the same order of non-overlapping operations. This in turn implies linearizability of the implementation using Definition 7.

In case of failure, our checker returns a valid history (sequence of sub-operations) which cannot be mapped to a sequential history. If the check on line 19 fails for prefix x and $y \in$ Closure(x) then the failing valid history is the sequence x,...,y, next(x). The dotted sequence is a sequence of prefixes such that for any subsequence u, v u and v are in Closure(x), $u \wr v$ and $\neg u \ominus v$.

6.5 Handling Complex Operation Interactions

We found that for complex concurrent operations, the ordering and reversibility of sub-operation pairs varies depending on the relative values of the involved variables. There are only finite number of ways in which two operations can interact with each other, i.e., how values of involved variables can relate to each other. Linearizability of an implementation can be checked by applying our technique using all possible interactions of the involved operations. We illustrate this process using O'Hearn et al.'s Lazy set [78] (ORVYY set).

Figure 6.11 shows the pseudocode for the ORVYY set. There are three concurrent methods *contains*, *remove*, and *add*. The specification for the ORVYY set, written in our language, is in Figure 6.12. There are three operations: *contains*, *remove*, and *add*. The operation *contains (key not present)* is equivalent to the operation *contains(key present)*. The operations *add (key present)* and *remove (key not present)* are trivial extensions and have been omitted for simplicity.

```
type E{
int mark;
int key;
E next;
}
E H, T;

function ExE locate(int k)
    E p = H;
    E c = p.next;
    while(c.key < k)
    p = c;
    c = p.next;
    endwhile
    return p,c;
end function
function bool contains(int k)
    ExE p,c = locate(k);
    bool b = (c.key == k);
     return b;
end function
```

```
function bool remove(int k)
     bool restart = true, retval;
     while (restart)
    ExE p,c = locate(k);
    atomic{
    if p.next == c && !p.mark then
        restart = false;
        if c.key == k then
            c.mark = true;
            p.next = c.next;
            retval = true;
        end if
    else
        retval = false;
    end if
    }
    endwhile
    return retval;
end function
```

```
function bool add(int k)
     bool restart = true, retval;
     while (restart)
    ExE p,c = locate(k);
    atomic{
    if p.next == c && !p.mark then
        restart = false;
        if c.key ≠ k then
            E t = alloc(E);
            t.mark = false;
            t.key = k;
            t.next = c;
            p.next = t;
            retval = true;
        end if
    else
        retval = false;
    end if
    }
    endwhile
    return retval;
end function
```

Figure 6.11: ORVYY set [78].

The interaction between operations depends on the relationships among the shared variables involved in the operations. For example, consider the interaction of operation *add* and operation *contains* from Figure 6.11. The *add* operations involves three variables (P_a, C_a, T_a). The *contains* operation involves two variables (P_c, C_c). The two operations can interact with each other in several ways. Each possible interaction is a result of a different relation between the involved variables. The list of possible iterations of *contains* and *add* is:

1. $P_c = P_a$ and $C_c = C_a$
2. $P_c = P_a$ and $C_c = T_a$
3. $P_c = T_a$ and $C_c = C_a$
4. P_c, C_c and P_a, C_a, T_a are not related

```
T1. struct E{int mark, int key, E next};

contains()
C1.   E P;
C2.   E C;
C3.   1 READ P;
C4.   1 C = P.next;
C5.   1 READ C.key;

remove()
R1.   E P;
R2.   E C;
R3.   1 READ P;
R4.   1 C = P.next;
R5.   1 < P[next] == C; P.mark == 0; >
             READ C.key;
              C.mark = 1;
              P.next = C.next;

add()
A1.   E P;
A2.   E C;
A3.   E T;
A4.   1 READ P;
A5.   1 C = P.next;
A6.   1 < P.next == C; P.mark == 0; >
             READ C.key;
              T.next = C;
              P.next = T;
```

Figure 6.12: ORVYY set [78] simplified specification.

Note that other cases are either infeasible or equivalent to one of the aforementioned cases. For example, the case where $C_c = P_a$ is equivalent to case 4. All the interactions between operations for the ORVYY set are listed in Figure 6.13. We use multiple versions of the same operation to cover every possible combination of interactions between operations (each version is considered as a new operation). The ordering and reversibility properties of the pairs corresponding to each pair of operations are defined according to the premises. The resulting ordering and reversibility definitions are fed to the checker to verify if the implementation is linearizable.

6.6 Soundness Proof

Now we show that our linearizability check is sound. First we show that for any input which passes the linearizability check, all valid histories with respect to the input

Contains()(P_c, C_c)-Remove()(P_r, C_r) 1. $P_c = P_r$, $C_c = C_r$ 2. $P_c = C_r$ 3. P_c, C_c and P_r, C_r are not related
Remove()(P_{r1}, C_{r1})-Remove()(P_{r2}, C_{r2}) 1. $C_{r1} = P_{r2}$ 2. P_{r1}, C_{r1} and P_{r2}, C_{r2} are not related
Add()(P_{a1}, C_{a1}, T_{a1})-Add()(P_{a2}, C_{a2}, T_{a2}) 1. $C_{a1} = P_{a2}$ 2. $P_{a1} = P_{a2}$, $C_{a1} = T_{a2}$ 3. $P_{a1} = T_{a2}$, $C_{a1} = C_{a2}$ 4. P_{a1}, C_{a1}, T_{a1} and P_{a2}, C_{a2}, T_{a2} are not related
Remove(P_r, C_r)-Add()(P_a, C_a, T_a) 1. $P_r = P_a, C_r = T_a$ 2. $P_r = T_a, C_r = C_a$ 3. $P_r = C_a$ 4. P_r, C_r and P_a, C_a, T_a are not related
Contains()(P_c, C_c) -Add()(P_a, C_a, T_a) 1. $P_c = P_a, C_c = C_a$ 2. $P_c = P_a, C_c = T_a$ 3. $P_c = T_a, C_c = C_a$ 4. P_c, C_c and P_a, C_a, T_a are not related

Figure 6.13: Operation interactions for the ORVYY set.

specification are equivalent to some valid sequential history with the same order of non-overlapping operations. We prove this by induction over the length of valid histories, where length of history refers to number of sub-operations in the history.

Base case: A sequence of a single sub-operation is a trivially valid sequential history.

Given: A history of length k maps to a valid sequential history with order of non-overlapping operations preserved.

To Prove: Any valid history formed by appending a new sub-operation to the history can also be mapped to a valid sequential history with the order of non-overlapping operations preserved.

In Figure 6.14 we start with a history of length $k + 1$ formed by appending sub-operation S_{n+1} from operation instance S to a history of length k. The history of length k can be

1.	$[\underbrace{........................}_{\text{Valid history of length } k}]S_{n+1}$	$\equiv$	$[....\underbrace{S_1S_2...S_n}_{\text{Prefix S}}\underbrace{.....................}_{\text{Valid sequence of prefixes}}]S_{n+1}$	Replacing history of length k by corresponding sequential history. $S_1S_2...S_n$ denoted by P_s
2.	$...P_s\underbrace{XY...............}_{\text{Valid sequence of prefixes}}S_{n+1}$	$\equiv$	$....XP_sY........S_{n+1}$	if($P_s \ominus X$).
3.	$....P_sXY........S_{n+1}$	$\equiv$	$....P_sYX........S_{n+1}$	if($\neg P_s \ominus X$ and $X \ominus Y$).
4.	$........P_s............S_{n+1}$	$\equiv$	$.........P_sZ_1Z_2..........Z_uS_{n+1}$	where $Z_i \in$ Closure(P_s) for $1 \leq i \leq u$
5.		$\equiv$	$.........P_sS_{n+1}Z_1Z_2........Z_u$	$\forall Z \in$ Closure(P_s), $Z \wr S_{n+1} \Rightarrow Z \ominus S_{n+1}$

Figure 6.14: Proving soundness of linearizability check by induction.

mapped to a valid sequential history. The valid sequential history will be a sequence of prefixes. The prefix $S_1S_2 \ldots S_n$ of operation instance S denoted by P_s will be present in the sequential history. The history formed by replacing the history of length k with its sequential counterpart falls under one of two cases:

- Case 1: P_s is immediately followed by prefix X such that $P_s \ominus X$. In this case, we reverse the order of P_s and X in the history (line 2).

- Case 2: P_s is immediately followed by X such that $\neg P_s \ominus X$ and X is immediately followed by Y such that $X \ominus Y$. In this case, we reverse the order of X and Y in the history (line 3).

We apply case 2 for prefixes down the sequence until no further order change is possible. The result is a history where P_s if followed by prefixes, each of which is an element of Closure(P_s) (line 4). Since our linearizability check guarantees that for every element Z in Closure(P_s) which can occur before S_{n+1}, $Z \ominus S_{n+1}$. Using this property, we reverse the oder of Z and S_{n+1}. The final result will be a history where sequence P_sS_{n+1} is followed by a sequence of prefixes, which is a valid sequential history (line 5). The order of non-overlapping operations remains same because boundary pairs are always non-reversible i.e., their order cannot be changed.

Using Definition 7 we can say that any implementation which holds Assumption 1 and passes the linearizability test is linearizable with respect to the abstract data structure.

6.7 Incompleteness

Our technique is not complete, i.e., an implementation which fails the linearizability test may or may not be non-linearizable. Specifically, a linearizable implementation can fail our linearizability test for two reasons:

1. Algorithms which do not preserve internal data structure state. There are linearizable algorithms which do not preserve the state of internal data structures when mapping a history to a sequential history. An example of this case is the Herlihy-Wing queue [47]. The queue is implemented using an unbounded length array and a pointer storing the upper end of the array. Let H' be the sequential history corresponding to a concurrent history H; then the execution of H and H' may leave the array with elements at different indexes.

Since our technique conserves the state of internal data structures while mapping a history to sequential a history, it returns False for the Herlihy-Wing queue.

2. Conservative definition of history (Definition 5). A history H, as defined in Definition 5, is the superset of all possible sets of histories allowed by an implementation. Let S be the set of all possible histories categorized as valid according to Definition 5, for a given implementation. It is theoretically possible to design an implementation for which S will include histories which can never be executed. If the linearizability test fails for such histories then our technique will result in a false positive. In such a case, the non-linearizable sequence of sub-operations returned by the linearizability checker can be manually verified to be non-executable, i.e., impossible at runtime.

6.8 Evaluation

We have applied our technique on a number of popular implementations of concurrent stacks, queues, and sets. Our static checker is a C++ implementation of Algorithm 6 running on an Intel(R) Xeon(R) CPU E5607 @ 2.27GHz with 16 GB RAM, Linux kernel version 2.6.32. Table 6.2 presents our findings. For each benchmark, the table reports operations we considered for the implementation (column 2) and the total number of sub-operations across all operations (column 3). Column 4 gives the time taken by our static checker took (in milliseconds). Column 5 indicates if the benchmark passed or failed the check. We have kept the granularity of sub-operations limited to single reads, writes, and synchronization primitives. The granularity can be easily increased for trivial cases (by combining consecutive sub-operations).

6.8.1 Benchmarks

The **MS non-blocking queue** was our running example. **MS two-lock queue** is the two-lock based queue from the same paper [73]. There are two methods described in the algorithm, *enqueue* and *dequeue*. The *dequeue* method corresponds to two operations,

Data Structure	**Operations**	**# Sub-ops**
MS non-blocking queue [73]	Enqueue, Dequeue(empty), Dequeue(non-empty)	4
MS two-lock queue [73]	Enqueue, Dequeue(empty), Dequeue(non-empty)	11
DGLM non-block. queue [32]	Enqueue, Dequeue(empty), Dequeue(non-empty)	4
Herlihy-Wing Queue [47]	Enqueue, Dequeue	5
Treiber's stack [101]	Push, Pop(empty), Pop(non-empty)	5
Elimination back-off stack [46]	Push(eliminating), Push(eliminated), Pop(eliminating), Pop(eliminated), Push (normal), Pop (normal)	20
Time-stamped Stack [31]	Push(normal), Push(eliminated), Pop(eliminating), Pop (normal)	10
HHLMSS Lazy set [45]	Contains, Remove(key present), Add (key not present)	23
VY CAS set [105]	Contains, Remove(key present), Remove(key not present), Add(key present), Add(key not present)	20
VY DCAS set [105]	Contains, Remove(key present), Remove(key not present), Add(key present), Add(key not present)	19
ORVYY set [78]	Contains, Remove(key present), Remove(key not present), Add(key present), Add(key not present)	15
Pair snapshot [84]	Read-pair, write	5
RDCSS [44]	RDCSS, RDCSS_Read, CAS_Write	5

Table 6.1: Specification details of concurrent data structure implementations.

one for the successful dequeue and the other for the empty-queue dequeue.

DGLM non-blocking queue [32] is a modified version of the MS non-blocking queue. The specification for the DGLM non-blocking queue varies from the MS non-blocking queue in terms of pre-condition for the sub-operations. The sub-operation ordering and reversibility remain the same for both the benchmarks.

Herlihy-Wing queue is an array-based queue described in the original linearizability paper [47]. The *Dequeue* method for an empty queue never terminates (that is why we have considered only the successful dequeue operation in our check). As described in Section 6.7, our technique fails to prove the Herlihy-Wing queue linearizable.

Treber's stack [101] is the simplest form of a non-blocking concurrent stack algorithm. It is a linked-list based implementation, and the operations — *Push*, *Pop(empty)*, and *Pop(non-empty)* — are performed using CAS.

Elimination back-off stack [46] is an elimination-based lock-free stack. Elimination refers to canceling out concurrent push and pop operations without modifying the central data structure. The elimination process uses two auxiliary arrays. A pair of concurrently executing push and pop are eliminated. There is no sequential execution equivalent for such a case. We handled this case by distributing the sub-operations between eliminated and eliminating operations. This way there exists a sequential execution equivalent of the two eliminated operations. The implementation supports *push* and *pop* methods. The elimination parts leads to four operations: *push (eliminating)*, *push (eliminated)*, *pop (eliminating)*, and *pop (eliminated)*.

Time-stamped stack [31] is a linked-list based stack where each thread has its own linked list, using timestamps to avoid total ordering in concurrent stack operations; it also uses elimination to increase performance. The stack supports *push* and *pop* methods. The implementation has four operations, *push(normal)*, *push (eliminated)*, *pop (eliminating)*, and *pop (normal)*. We have not considered the stack empty check for this implementation. In addition, the *pop* operation is limited to setting the *deleted marker* which is associated with each node. Finally, we have not considered the removal of the node from the linked list.

HHLMSS Lazy set [45] is a linked-list based set which uses locks. Each node has a lock associated with it. The implementation supports three methods: *contains*, *remove* and *add*. The *contains* method is wait-free. The operations in the implementation that we have considered are *contains*, *remove (key present)*, and *add(key not present)*.

VY CAS set and **VY DCAS set** [105] are linked-list based set algorithms which use Compare-and-swap (CAS) and Double Compare-and-swap (DCAS) primitives for synchronization. The *contains* method is wait-free. The operations involved in the implementation are *contains*, *remove (key present)*, *remove (key not present)*, *add (key-present)* and *add(key not present)*.

ORVYY set [78] is also a linked-list based set which uses a marked bit for marking deleted nodes; it has been discussed in detail in Section 6.5.

Pair snapshot [84] reads two variables atomically in the presence of concurrent writes.

RDCSS [44] is an atomic multiword compare-and-swap. It works in the presence of RDCSS read and CAS based writes.

6.8.2 Discussion

The evaluation shows that our technique is applicable to a variety of data structure implementations, regardless of which synchronization techniques they use. Our technique is very efficient (running time is at most 29 ms) because the static checker explores a very small search space compared to other techniques. Tools like CAVE [103] and Poling [116] take several seconds to several hundred seconds for the benchmarks they can handle. Note

Data Structure	**Time (ms)**	**Passes Check**
MS non-blocking queue [73]	5	Yes
MS two-lock queue [73]	9	Yes
DGLM non-block. queue [32]	5	Yes
Herlihy-Wing Queue [47]	3	No
Treiber's stack [101]	3	Yes
Elimination back-off stack [46]	14	Yes
Time-stamped Stack [31]	8	Yes
HHLMSS Lazy set [45]	29	Yes
VY CAS set [105]	23	Yes
VY DCAS set [105]	23	Yes
ORVYY set [78]	19	Yes
Pair snapshot [84]	3	Yes
RDCSS [44]	4	Yes

Table 6.2: Checking linearizability of different concurrent data structure implementations.

that we are not comparing our verification time to theirs — it would be inappropriate to do so, as their techniques are fully automatic. The main strength of our technique is that it is applicable to any concurrent data structure, i.e., it is generic. The *Time-stamped stack* has never been handled by any linearizability verification technique. The paper presenting the data structure provides a very customized linearizability proof for the algorithm. The main overhead of our technique is specifying the operations in terms of sub-operations. We found that the method of specifying operations varies with the technique used for synchronization. The elimination technique has been used in elimination back-off stack and time-stamped stack. Elimination leads to different versions of operations depending upon whether the operation is being eliminated or is eliminating. The set algorithms, the time-stamped stack, and the pair-snapshot benchmarks had complex interactions among the operations. We handled these benchmarks by using multiple versions of some operations (as described in Section 6.5). The MS non-blocking queue, the elimination back-off stack, and the RDCSS benchmarks had operations distributed across multiple threads.

6.9 Summary

This chapter has presented our linearizability verification technique. The technique is based on modeling a concurrent data structure operation as a sequential combination of atomic sub-operations. The execution history is expressed in terms of static properties of *ordering* and *reversibility* of sub-operation pairs. Our static checks verifies if based on pair properties, all the histories allowed by an implementation can be mapped to a sequential history. We have applied our technique to 13 popular concurrent data structures. We have also proved the soundness of our technique. The evaluation shows that our verification technique is applicable to a wide range of concurrent data structures.

Chapter 7

Related Work

This chapter summarizes research works in domains and problems addressed by this thesis. We first summarize prior techniques used for general purpose debugging as well as specilized debugging for data structure related bugs. Next, we address various memory graph represetations and techniques using memory graphs. Finally, we summarize work on verification of linearizability of concurrent data structure implementations and how our techniques stand out.

7.1 Location Data Structure Faults for Scalar Data Structures

Fault location. Various general approaches for fault location that do not rely on data structure information have been developed (e.g., statistical techniques [13, 64, 89], dynamic slicing [56, 115], or combinations of the two [114]). Statistical techniques require running the program on a suite of test cases. Our approach is more comparable with dynamic slicing as they both perform debugging by analyzing a single program run during which a fault is encountered. Jose and Majumdar [52] use MAX-SAT solvers while Sahoo et al. [92] use dynamic backward slicing and filtering heuristics for software fault location. The focus of our work is specifically on data structure errors. Taylor and Black [100] examine a number of structural correction algorithms for list and tree data structures. Bond et al. [19] track back undefined values and NULL pointers to their origin. In contrast, our system concentrates on fault location for violation of high-level data structure properties. We consider the concept of temporary violation of high-level data structure constraints. This helps us locate errors early and trace them to faults precisely.

Specification-based error detection. A wide range of specification techniques have been used to specify correctness properties from which monitors for runtime verification of those properties are generated. For example, MOP [23] allows correctness properties to be specified in LTL, MTL, FSM, or CFG; though MOP is geared more towards verifying protocols or API sequences rather than data structures. Similarly, a specification technique for easily handling memory bugs has also been developed [115]. Our work focuses on data structure correctness and hence is closest to Archie [28] and Alloy [48]. Our data structure specification language differs from languages used by Archie and Alloy: their modeling languages let the developer specify high-level design properties in terms of a model while our language enables developers to express high-level data structure properties in terms of the memory graph of the program. This makes specification writing in our language easy and concise. Berdine et al. [17] and Chang and Rival [21] have introduced predicate-based specification languages for shape analysis. Customized versions of these languages can be used for our technique. Malik et al. [71], Juzi [35] and Demsky et al. [28, 29] use constraint-based error detection for data structure repair while Gopinath et al. [42] combine spectra-based localization with specification matching to iteratively localize faults. Malik et al. [70] proposed the idea of using using data structure repair for repairing faulty code. Jung and Clark [54] applied invariant detection on memory graphs to identify the data structures used in the program. Our system uses a similar concept of matching constraints over program state to detect violations. Our approach goes one step further and maps the dynamic data structures constraint violations at runtime back to the source code. Also, we give a method for incremental matching based on the timing information which makes it feasible for the developers to perform constraint checks more frequently. This leads to *early* detection of errors. Zaeem et al. [77] use history of program execution (field reads and writes) for data structure repair but do not capture structural information.

DITTO [96] performs incremental structure invariant checks for JAVA. It incurs additional overhead for storing all the computations from the last check, as these computations are later used for the incremental checking. Our system reuses the time stamp information from the memory graph and stores only the timestamp of the previous check for incremental constraint matching.

7.2 Memory Graphs

7.2.1 Memory Graphs

Prior heap analyses fall into two main categories: *static* and *dynamic*. Static techniques include shape analysis [41, 91] and other techniques that are aimed at deriving a compile-time approximation of the heap structure [72]. However, static analysis does not give a clear picture of the runtime heap activity in a particular execution. Dynamic analysis techniques like ours collect data from program runs and analyze it either online or offline. Several prior dynamic techniques are aimed at visualizing and navigating a single snapshot or a series of snapshots of the heap [14,81,99,117]. These works are orthogonal to our work and can make use of MG++ internally for memory efficiency. DDD [113] and Zimmermann & Zeller [117] use symbol table information for constructing memory graph while Raman & August [87] use allocator function information for memory graph construction. Work on visualizing memory management behavior [22, 82] has focused on analyzing memory allocator's behavior and performance. Unlike our work, these approaches are not aimed at capturing the memory graph for program understanding or debugging applications. The technique presented in [53] is specifically aimed at detecting recursive data structures and dynamic degree invariants. Our approach captures all pointer mutations of the heap and the resulting MG++ can be used for detecting data structures and their links. Finally, MG++ representation is similar to that of persistent data structures [33].

7.2.2 Applications of Memory Graphs

Memory graphs are at the heart of a number of different applications aimed at program understanding and debugging.

Memory debugging and general-purpose debugging. A number of approaches have aimed at displaying memory uses of the program using memory graphs thereby enabling programmers to detect memory leaks or identify memory corruption patterns [14, 80, 88]. In Zeller's work [112], program state is captured as a memory graph and state differences between passing and failing runs are used to isolate cause-effect chains for a program failure.

Program Understanding. Myers and Duke [74] extract design abstractions from memory graphs to provide users an intuitive representation. Malik [69] analyzes spectra of heap graphs to extract dynamic invariants.

Data Structure Extraction. Jung and Clark [54] apply invariant detection on a memory graph to identify the data structure used by the program. Lin et al. [65] extract type information from program binary and provide an abstract representation of the variables used in the program. They rely on standard library function calls for the typing information.

All of the above techniques rely on memory graphs and therefore can benefit from our improved representation as it captures program behavior with greater precision, including the behavior of the memory allocator. Moreover, the compact nature of our representation will allow the above techniques to scale to longer program runs.

7.3 Linearizability Verification

We presented a general and practical technique for checking the linearizability of concurrent data structure implementations. We now discuss other techniques used for verifying linearizability; our focus is on generic techniques that can be applied to more than one concurrent data structure.

Model-checking linearizability [20, 106] aims at exploring all possible linearization points and finding a counter example; this does not guarantee soundness. There are linearization-point based proof techniques for which the linearization points are user specified or automatically inferred by the techniques. Such techniques fail to handle more complicated algorithms where an operation's linearization point depends upon other concurrently executing operations.

Most of the techniques for proving linearizability are tied to a particular class of concurrent data structures. There are techniques which work only for the algorithms which have the linearization point inside the operation code [12]. Other techniques work for external linearization points only for read only operations [30]. Reduction based technique presented in [38] requires *moverness* which is too strong a criterion limiting the application of the approach. The backward-simulation based technique in [95] claims to be applicable to all concurrent data structure; it handles the Herlihy-Wing queue as well, which we cannot. According to the paper the authors had to write 500 proof rules in the KIV theorem prover just for the specific Herlihy-Wing queue. Authors have applied their technique only on the Herlihy-Wing queue and extending the technique to other data structures is not trivial. Liang and Feng [63] use instrumentation and rely-guarantee reasoning. The technique is specifically built to handle concurrent data structure implementations which have *helping mechanisms* and *future dependent linearization points*. Vafeiadis's approach [103] works on a number of data structures but fails in verifying complex set algorithms. Zhu et al. [116] handle data structures implementations which follow the patterns of *thread helping* and *hindsight*. In contrast to these techniques, our method is not dependent on any property of the data structure implementation. Another advantage of our technique is that when the check fails, our method provides the user with a sequence of sub-operations which cannot be linearized.

Chapter 8

Conclusions and Future Work

8.1 Contributions

This dissertation contributes to enabling precise fault location for data structure faults. It presents a data structure constraint specification language that enables the user to specify constraints easily, thus simplifying the task of debugging data structure faults. The fault location framework also comprises of various optimizations which reduce the runtime overhead of fault location. A new memory graph representation is introduced to decrease the memory used for storing the data structure evolution history of a program run. The improved backward slicing algorithm reduces the runtime overhead of slicing enabling the user to perform slicing for much longer program runs.

Our sound and generic technique for linearizability verification is independent of the synchronisation technique used in the concurrent data structure. The technique only depends on the static properties of sub-operation pairs. our technique is applicable to a variety of data structure implementations, regardless of which synchronization techniques they use. The only criterion for application of out technique is that the concurrent operation should be expressed as a sequence atomic sub-operations.

Specifically, the key contributions of this dissertation are as follows:

Exploiting User Input to Improve Precision of Automatic Fault Location. Automatic Fault Location techniques have to explore a vast search space in order to find locate the program faults. This not only increase the time overhead of the search but also affects the precision of the technique. This dissertation presented our fault location framework for data structure corruption bugs where we utilized the user specification of data structure consistency constraints to narrow down the faulty statements precisely and efficiently. We have provided the user with an easy-to-use constraint specification language, thus making the technique more practical.

Increasing Efficiency of Dynamic Post-mortem Analysis. Dynamic Analysis in general is very costly in terms of time overhead. This dissertation introduces a number of optimizations to make dynamic post-morterm analysis more efficient and practical. Incremental Constraint matching introduced as a part of the fault location framework uses information from previous constraint checks to reduce the cost of constraint matching. Identifying corrupted data structures limits the search space making the search for faulty program statements more efficient. We have introduced an improved version of backward slicing which harnesses the idea to using a relatively lightweight check to discard big portion of search space.

Improved Representation for Run-time Data Structures. Memory graphs have been popular to capture the run-time data structure representation. This dissertation presented MG++, an improved memory graph representation which not only captures the run-time data structure but also captures data structure evolution history. MG++ also captures mapping from run-time data structure to source code. We have also presented a way of constructing memory graphs in absence of allocator function information. This construction method is especially useful in security applications where allocator function information is generally not available. We have shown the application of the construction technique in detecting heap buffer overflows.

Sound and Generic Linearizability Verification. Linearizability verification for concurrent data structures is a hard problem. It becomes harder in case of introduction of newer synchronization techniques. There are no techniques which can be used for a newly identified concurrent data structure implementations. Our technique for linearizability verification works for any implementation as long as the concurrent operations can be expressed as a sequence of atomic sub-operations. This is crucial as our technique will theoretically not only work for existing concurrent data structure implementations but also for new algorithms. We have supported this claim by evaluating our technique against a number of state of the art and complex concurrent data structure implementations such as such as elimination back-off stack, lazy linked list, and time-stamped stack. Thus our linearizability verification technique helps the research community come up with better concurrent data structure implementations.

8.2 Future Directions

Dependence Analysis using Dynamic Slicing. With the ubiquity of multicore hardware, finding parallelism in programs has become an important application. Detecting loop-carried dependencies is an important step towards detecting the amount of parallelism present in the program. We propose to detect loop-carried dependencies by combining *Dynamic Control-Flow Graph(DCFG) Generation* with *Dynamic Forward Slicing*. The role of DCFG is to detect source level loops. Once the loops are identified, we use dynamic forward slicing to detect loop-carried dependencies by using the first iteration of the loop as slicing criterion. If the later iterations of the loop are found to be part of the calculated forward slice, we report a loop-carried dependency. This work is being pursued in collaboration with Dr. Harish Patil at Intel. A working demo of the tool was presented at PLDI 2016 as part of the *Pinplay* tutorial.

Automatic Linearizability Verification with Source Code as Input. Our current linearizability verification technique is based on user input. The user expresses the concurrent data structure operations as a sequence of atomic sub-operations using our specification language. This poses the danger of user specifications being erroneous. In addition, writing the specification requires some degree of familiarity with the concurrent data structure algorithm. In future work we plan to overcome these limitations by completely automating the verification process. The verification tool will analyze the source code and automatically extract the sequence of sub-operations along with *ordering* and *reversibility* properties for each sub-operation pair. The results will be fed to the static checker to automatically verify linearizability.

Persistent Memory Analysis using Backward Slicing Persistent memory is an intermediate stage in the memory/storage hierarchy between DRAM and storage. It provides nonvolatile, low-latency memory closer to the processor. We aim at developing a tool for automatically determining the commit points in a program for persistent storage. The persistent analysis is based on determining the data and control dependency between memory loads. Backward slicing will be used for determining the memory load pairs with dependencies. The challenges are performing the analysis at run time for long running executions.

Automatic Extraction of Constraint Specification Our current fault location framework is based on getting the data structure constraints as input from the user. In future work we plan to automatically extract the data structure constraint from a given correct execution of the program. The modified fault location framework can facilitate software development, e.g., regression testing.

Bibliography

[1] Bison-gnu parser generator. http://www.gnu.org/software/bison/.

[2] flex: The fast lexical analyzer. http://flex.sourceforge.net/.

[3] Gnome bug tracker. `https://bugzilla.gnome.org/`. Accessed: 03/2014.

[4] Gnu core utilities. http://www.gnu.org/software/coreutils/.

[5] Kde bug tracker. `https://bugs.kde.org/`. Accessed: 03/2014.

[6] Mozilla bug tracker. `https://bugzilla.mozilla.org/`. Accessed: 03/2014.

[7] Open office bug tracker. `https://issues.apache.org/ooo/`. Accessed: 03/2014.

[8] Perl programming language. http://www.perl.org/.

[9] Python interpreter. https://www.python.org/.

[10] Spec cint2006 benchmarks. https://www.spec.org/cpu2006/cint2006/.

[11] Stanford large network dataset collection. http://snap.stanford.edu/data/index.html.

[12] Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. An integrated specification and verification technique for highly concurrent data structures. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2013.

[13] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim International Symposium on Dependable Computing*, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

[14] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. Heapviz: interactive heap visualization for program understanding and debugging. In *SoftVis'10*, pages 53–62, 2010.

[15] Daphna Amit, Noam Rinetzky, Thomas Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer Berlin Heidelberg, 2007.

[16] Vladimir Batagelj and Andrej Mrvar. Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/data/, 2006.

[17] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings*, chapter Shape Analysis for Composite Data Structures, pages 178–192. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[18] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[19] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. In *OOPSLA*, pages 405–422, 2007.

[20] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 330–340, New York, NY, USA, 2010. ACM.

[21] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In George C. Necula and Philip Wadler, editors, *POPL*, pages 247–260. ACM, 2008.

[22] A. M. Cheadle, A. J. Field, J. W. Ayres, N. Dunn, R. A. Hayden, and J. Nystrom-Persson. Visualising dynamic memory allocators. In *ISMM '06*, pages 115–125.

[23] F. Chen and G. Rosu. Mop: An efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.

[24] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[25] I. B. M. Corporation. *IBM System/370 Extended Architecture, Principles of Operation.* IBM Publication No. SA22-7085, 1983.

[26] Sandeep Dasgupta and Amey Karkare. Precise shape analysis using field sensitivity. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1300–1307, 2012.

[27] Brian Demsky, Cristian Cadar, Daniel Roy, and Martin Rinard. Efficient specification-assisted error localization. In *WODA'04*.

[28] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.

[29] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *ICSE*, pages 176–185, 2005.

[30] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Verifying linearisability with potential linearisation points. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 323–337. Springer Berlin Heidelberg, 2011.

[31] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 233–246, New York, NY, USA, 2015. ACM.

[32] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In David de Frutos-Escrig and Manuel Núñez, editors, *FORTE*, volume 3235 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2004.

[33] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, February 1989.

[34] Cezara Drăgoi, Ashutosh Gupta, and ThomasA. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 174–190. Springer Berlin Heidelberg, 2013.

[35] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 855–858, New York, NY, USA, 2008. ACM.

[36] Bassem H. Elkarablieh. *Assertion-based Repair of Complex Data Structures.* PhD thesis, Austin, TX, USA, 2009. AAI3517771.

[37] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. Graphviz — open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[38] Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer Verlag, April 2010.

[39] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 2–15, New York, NY, USA, 2009. ACM.

[40] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. *IEEE Trans. Softw. Eng.*, 31(4):275–291, April 2005.

[41] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 1–15, 1996.

[42] D. Gopinath, R.N. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *ASE'12*.

[43] Lindsay Groves. Verifying michael and scott's lock-free queue algorithm using trace reduction. In *Proceedings of the Fourteenth Symposium on Computing: The Australasian Theory - Volume 77*, CATS '08, pages 133–142, Darlinghurst, Australia, 2008. Australian Computer Society, Inc.

[44] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, London, UK, UK, 2002. Springer-Verlag.

[45] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS'05, pages 3–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[46] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

[47] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[48] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[49] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.

[50] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM ASE*, ASE '05, pages 273–282, New York.

[51] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[52] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. PLDI'11, pages 437–446.

[53] Maria Jump and Kathryn S. McKinley. Dynamic shape analysis via degree metrics. In *Proceedings of the International Symposium on Memory Management*, ISMM '09, pages 119–128, 2009.

[54] Changhee Jung and Nathan Clark. Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage. In *IEEE/ACM MICRO*, pages 56–66, 2009.

[55] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM.

[56] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, October 1988.

[57] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley.

[58] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 1987.

[59] Mohsen Lesani, Todd Millstein, and Jens Palsberg. *Automatic Atomicity Verification for Clients of Concurrent Data Structures*, pages 550–567. Springer International Publishing, Cham, 2014.

[60] M. E. Lesk and E. Schmidt. Unix vol. ii. chapter Lex - a lexical analyzer generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[61] Kfir Lev-Ari, Gregory Chockler, and Idit Keidar. A constructive approach for proving data structures' linearizability. In Yoram Moses, editor, *Distributed Computing*, volume 9363 of *Lecture Notes in Computer Science*, pages 356–370. Springer Berlin Heidelberg, 2015.

[62] Chung-Chi J. Li, Paul P. Chen, and W. K. Fuchs. Local concurrent error detection and correction in data structures using virtual backpointers. *IEEE Trans. Comput.*, 38(11):1481–1492, November 1989.

[63] Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 459–470, New York, NY, USA, 2013. ACM.

[64] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan. Scalable statistical bug location. In *PLDI*, 2005.

[65] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS*, 2010.

[66] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[67] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

[68] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[69] Muhammad Zubair Malik. Dynamic shape analysis of program heap using graph spectra (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 952–955, 2011.

[70] Muhammad Zubair Malik, Khalid Ghori, Bassem Elkarablieh, and Sarfraz Khurshid. A case for automated debugging using data structure repair. In *ASE*, pages 620–624, 2009.

[71] M.Z. Malik, J.H. Siddiqi, and S Khurshid. Constraint-based program debugging using data structure repair. ICST, pages 190–199, 2011.

[72] Mark Marron, Deepak Kapur, and Manuel Hermenegildo. Identification of logically related heap regions. In *Proceedings of the International Symposium on Memory Management*, ISMM '09, pages 89–98, 2009.

[73] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[74] Colin Myers and David Duke. A map of the heap: revealing design abstractions in runtime structures. ISSV'10, pages 63–72.

[75] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, 2007.

[76] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *pre*, 74(3):036104, September 2006.

[77] Razieh Nokhbeh Zaeem, Divya Gopinath, Sarfraz Khurshid, and Kathryn S. McKinley. History-aware data structure repair using sat. TACAS'12, pages 2–17.

[78] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 85–94, New York, NY, USA, 2010. ACM.

[79] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[80] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 116–134. Springer-Verlag, 1999.

[81] S. Pheng and C. Verbrugge. Dynamic data structure analysis for java programs. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 191–201, 2006.

[82] Tony Printezis and Richard Jones. Gcspy: An adaptable heap visualisation framework. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 343–358, 2002.

[83] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *AADEBUG*, pages 273–276, Ghent, Belgium, September 8–10, 2003.

[84] Shaz Qadeer, Ali Sezgin, and Serdar Tasiran. Back and forth: Prophecy variables for static verification of concurrent programs. Technical Report MSR-TR-2009-142, October 2009.

[85] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. In *Proceedings of the 34th Conference on Programming Language Design and Implementation (PLDI'13)*, pages 231–242. ACM, June 2013.

[86] Dave Raggett. Html tidy program. http://tidy.sourceforge.net/, 2009.

[87] Easwaran Raman and David I. August. Recursive data structure profiling. In *MSP Workshop*, pages 5–14, 2005.

[88] Steven P. Reiss. Visualizing the java heap to detect memory problems. In *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 73–80, 2009.

[89] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *IEEE ASE*, pages 30–39, 2003.

[90] Shounak Roychowdhury and Sarfraz Khurshid. Software fault localization using feature selection. In *International Workshop on Machine Learning Technologies in Software Engineering*, MALETS '11, pages 11–18, 2011.

[91] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, 1999.

[92] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *ASPLOS'13.*

[93] Gerhard Schellhorn, John Derrick, and Heike Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Logic*, 15(4):31:1–31:37, September 2014.

[94] Gerhard Schellhorn, John Derrick, and Heike Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Logic*, 15(4):31:1–31:37, September 2014.

[95] Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to prove algorithms linearisable. In P. Madhusudan and SanjitA. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 243–259. Springer Berlin Heidelberg, 2012.

[96] Ajeet Shankar and Rastislav Bodík. Ditto: automatic incrementalization of data structure invariant checks (in java). In *PLDI*, 2007.

[97] Vineet Singh, Rajiv Gupta, and Iulian Neamtiu. Mg++: Memory graphs for analyzing dynamic data structures. In *IEEE SANER*, 2015.

[98] Shiwei Sun, Lunjiang Ling, Nan Zhang, Guojie Li, and Runsheng Chen. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucleic Acids Research*, 31(9), 2003.

[99] Jaishankar Sundararaman and Godmar Back. Hdpv: Interactive, faithful, in-vivo runtime state visualization for c/c++ and java. In *Proceedings of the 4th ACM Symposium on Software Visualization*, pages 47–56, 2008.

[100] D.J. Taylor and J.P. Black. Principles of data structure error correction. *IEEE Trans. on Computers*, C-31(7):602–608, 1982.

[101] R.Kent Treiber. Systems programming : coping with parallelism. Technical Report RJ 5118, IBM US Research Centers (Yorktown,San Jose,Almaden, US), 1986.

[102] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07.

[103] Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[104] Viktor Vafeiadis. Automatically proving linearizability. In *In CAV*, 2010.

[105] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 125–135, New York, NY, USA, 2008. ACM.

[106] Martin Vechev, Eran Yahav, and Greta Yorsh. Experience with model checking linearizability. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 261–278, Berlin, Heidelberg, 2009. Springer-Verlag.

[107] Liqiang Wang and Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 61–71, New York, NY, USA, 2005. ACM.

[108] Y. Wang, I. Neamtiu, and R. Gupta. Generating sound and effective memory debuggers. In *ISMM*, 2013.

[109] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 98:98–98:108, New York, NY, USA, 2014. ACM.

[110] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*, 2011.

[111] Xiao Xiao, Jinguo Zhou, and Charles Zhang. Tracking data structures for postmortem analysis (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 896–899, 2011.

[112] Andreas Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT FSE*, pages 1–10, 2002.

[113] Andreas Zeller and Dorothea Lütkehaus. Ddd&mdash;a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, January 1996.

[114] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, 2006.

[115] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. AADEBUG'05, pages 33–42.

[116] He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: Smt aided linearizability proofs. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer International Publishing, 2015.

[117] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Revised Lectures on Software Visualization, International Seminar*, pages 191–204, London, UK, UK, 2002. Springer-Verlag.