UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Concolic Testing of Programs with Concurrent Dynamic Data Structures

Permalink
https://escholarship.org/uc/item/5bd9g7sn

Author
Sun, Xiaofan

Publication Date
2023

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Concolic Testing of Programs with Concurrent Dynamic Data Structures

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

Doctor of Philosophy

in

Computer Science

by

Xiaofan Sun

March 2023

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Chengyu Song
Dr. Manu Sridharan
Dr. Zhijia Zhao

Copyright by
Xiaofan Sun
2023

The Dissertation of Xiaofan Sun is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am very grateful to my advisor Rajiv Gupta. Without his help, I would not have reached this milestone. There is not only academic literacy but also life experience learning from him. I want to express my sincere gratitude to my advisor for their careful guidance and assistance over the years. I also appreciate Prof. Zhijia Zhao since he enlightened me on the way of academic thinking and gave me a lot of help in the initial stage of scientific research. Without his help, it would have been hard for me to grasp the methods and abilities of academic research. I also thank Prof. Chengyu Song and Prof. Manu Sridharan for serving on the dissertation committee and providing valuable feedback. Finally, I am grateful to my wife Wei. With her support and encouragement, I can overcome the difficulties, cheer up and complete my Ph.D. program.

I would like to acknowledge the support of National Science Foundation via grants CCF-1813173 and CCF-2226448 to UC Riverside. This thesis included previously published work "DSGEN: concolic testing GPU implementations of concurrent dynamic data structures." at conference International Conference on Supercomputing(ICS), in 2021.

To my family for all the support.

ABSTRACT OF THE DISSERTATION

Concolic Testing of Programs with Concurrent Dynamic Data Structures

by

Xiaofan Sun

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2023
Dr. Rajiv Gupta, Chairperson

Concolic execution combines concrete execution with symbolic execution to automatically generate test inputs that exercise different program paths and deliver high code coverage. However, when this technique is extended to multithreaded programs with concurrent dynamic data structures, the lack of support for exploring data shapes (the skeleton that consists of symbolic pointers in node-based linked data structures) and efficiently exploring thread interleavings makes it hard to expose concurrency bugs that manifest only when certain dynamic data structure shapes, program paths, and thread interleavings are exercised. This thesis presents techniques to effectively explore data shapes for concurrent dynamic data structures and optimize the exploration efficiency.

The approach presented first generates a data shape for a chosen path. By capturing path constraints, we form a shape that satisfies path constraints and exercises the chosen path. In addition, by capturing pointer-pointee relationships, we find how to adjust the shape and find new shapes that also exercise the same path. Finally, using the shapes for individual paths, we find a consistent shape that causes multiple threads to simultaneously follow their chosen paths for exposing concurrency bugs. We generate shapes for each thread separately and provide an integration algorithm to merge the shapes into a consistent shape. This approach does not require the user to write code that constructs data structures to exercise desired paths by individual threads; rather, it automatically collects constraints to generate consistent shapes during the concolic execution.

We also present a summarization-based technique to improve the efficiency of concolic testing further. Via unit testing of key functions that implement a concurrent data structure, function summaries are derived that capture data structure shapes that cause various function paths to be exercised. During the concolic testing of interprocedural paths, these summaries are exploited to eliminate the repeated overhead of handling symbolic pointers and creating dynamic objects by reusing function summaries. The summary also contains symbolic memory accesses and synchronization events that guide application-level concolic testing to identify and confirm potential data races.

To demonstrate effectiveness and efficiency of our approach, we developed two prototypes: DSGEN is built on top of the GKLEE GPU concolic executor; and SSRD is built on top of the Cloud9 concolic executor for multithreaded programs. DSGEN improves the number of races detected from 10 to 25 by automatically generating 1,897 shapes in experiments with concurrent operations on B-Tree, HAMT, RRB-Tree, and Skip List. SSRD, using shape generation and function summaries, outperforms AFL++ and Cloud9 in both effectiveness and efficiency in experiments with Unrolled Linked List, AVL-Tree, Skip List, and Priority Queue. SSRD detects 74 races as opposed to 34 by Cloud9 and 11 by AFL++.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the rapid growth of large scale software systems and the widespread deployment of hardware that supports thread-level parallelism (e.g., multi-core processors and GPUs), automated software testing has been receiving greater attention to enable identification of concurrency bugs in multithreaded programs. Multithreaded programs frequently use concurrent dynamic data structures (e.g., Concurrent Queue, Skip list, Hash Map, etc.) to safely share state between different threads. The use of these concurrent data structures greatly increases the difficulty of automated testing.

Effectively detecting bugs, such as data races, in multithreaded software is a challenging problem. Researchers have developed both static and dynamic techniques for automated testing. The complexity of static methods is often related to the code size, as opposed to program run length, which makes them much more efficient than dynamic methods. Soundness is another advantage of static methods, that is, it can be proved that they find all potential faults. However, these methods can suffer from the problem of false positives, which greatly increases the workload of manual confirmation of potential bugs. Another limitation of static methods is that it is difficult to reproduce bugs and find the conditions under the path errors occur. Typically, the dynamic methods are expensive and can only find bugs that manifest during a program run. However, an exposed bug is real and can be reproduced again via replay techniques. There has also been work done on developing methods that combine static and dynamic techniques to combine the advantages of both.

Symbolic Execution is a powerful technique for automatically generating a symbolic model of a program. Concolic Execution combines symbolic execution with concrete execution to generate symbolic expression along with concrete values for the program inputs. In comparison to other popular automated test generation techniques, e.g. static analyses and fuzzing, symbolic execution has several advantages: (i) it follows executable paths in the program so there are no false positives; (ii) it can generate inputs for hard to exercise paths; and (ii) it finds specific inputs that expose an error. The symbolic execution technique is used in the following chapters to generate a set of path constraints to deliver high test coverage.

1.1 Overview of Challenges and Approaches

Although concolic testing is a powerful technique for testing programs, its applicability for testing of multithreaded programs that utilize concurrent data structures is limited. This thesis addresses the following challenges to overcome this limitation.

Exploring Data Structure Shapes. Although symbolic/concolic execution is effective and powerful for many real applications, programs with pointer-based dynamic data structures are hard to test due to the path constraints that can only be satisfied by certain data structure shapes. Therefore to achieve high path coverage, we need to manually create many different data structure shapes which is time-consuming. In this thesis, we propose a novel technique to automatically generate data structures with different shapes to exercise different program paths.

Exposing Concurrency Bugs in Multithreaded Programs. In a multithreaded program, a concurrency bug manifests when a pair of threads follow certain paths. Therefore, we must generate a data structure shape that causes the threads to simultaneously follow their respective chosen paths. Moreover, if a code segment being executed by two threads contains N paths, then there are N^2 path pairs that must be exercised to uncover concurrency bugs. Therefore the complexity and cost of uncovering concurrency bugs is very high. To achieve the above goal efficiently, we first generate individual shapes that cause each of the N paths to be exercised. Then, to exercise a path pair, we integrate the two shapes corresponding to the two paths such that the resulting shape causes the two threads to exercise the chosen path pair.

Interprocedural Paths. Consider two functions f and g containing N and M paths respectively. Consider an execution in which f and g are called in sequence. Thus, the call sequence gives rise to $M \times N$ interprocedural paths. If we test these paths one by one, a great deal of redundant work will be performed during exploration of data shapes via constraints collection and solving. Therefore, we unit test functions f and g and create *summaries* for the two functions. Interprocedural paths can then be efficiently tested using summaries of paths in f and g. In addition, summaries are also used to identify concurrency bugs and guide exploration of thread interleavings to improve the efficiency of testing.

1.2 Overview of the Thesis

Existing systems for testing C/C++ programs do not support the exploration of data shapes as pointers cannot be made symbolic (e.g., Cloud9 [29], Con2colic [37]) or they do not support multithreading (e.g., CUTE [95]). Also, the cost of concolic testing explodes due to large numbers of possible data structure shapes and interleavings of threads that manipulate the data shape. Consequently, there is a need to design a powerful concolic testing framework that supports symbolic pointers for the systematic exploration of data structure shapes and employs new techniques to guide the exploration of paths and thread schedules both effectively and efficiently. To fulfill the above goals, this thesis presents a testing framework with the following features:

- **Exploring data structure shapes.** Chapter 3 presents our technique that makes concolic testing capable of achieving high coverage of program paths and data shapes. Therefore it increases the possibility of exposing concuurency bugs.

- **Generating consistent shapes for multithreaded programs and GPU programs.** Chapter 4 presents our technique for data shape exploration to exercise path pairs for exposing concurrency bugs. We apply this approach to both multithreaded programs with modest number of threads and GPU programs with thousands of threads.

- **Efficient exploration using summarization and guided search.** Chapter 5 presents a series of techniques to explore data structure shapes efficiently. We provide a shape-aware summary to speed up path exploration with a data shape generation algorithm. A loop-aware summary can handle the infinite exploration tree problem for loops with symbolic latch conditions. Guided search can help thread scheduling to detect data races in a more efficient way.

Before presenting the above results, in Chapter 2 we present background and related work. Finally, in Chapter 6, we present our conclusions.

Chapter 2

Background and Related Work

Testing programs with concurrent dynamic data structures is a challenging problem. There are many known related works in automated testing, including fuzzing, symbolic/concolic execution, model checking, static analysis-based techniques, and others. In this chapter, we will introduce the related works and relevant background knowledge.

2.1 Automated Testing

There is rich literature on generating test inputs [21, 25, 28, 45, 48, 58, 60, 61, 71, 82, 91, 96, 106, 115]. A number of techniques are aimed at generating test input for a given path in a single-threaded program using various approaches: Godzilla [30] and Gotlieb et al. [45] employ constraint solving; Grechanik et al. [48] and Petsios et al. [82] employ feedback-directed fuzz testing; Mansour and Salame [71] developed stochastic search algorithms; and Gupta et al. [49] developed iterative numerical techniques. However, these techniques cannot systematically explore dynamic data structure shapes.

For multithreaded programs the techniques of *fuzzing* [26, 54, 82, 93, 105], *model checking* [78], and *symbolic and concolic execution* [15, 23, 29, 37, 62] are primary options for testing. *Fuzzing* is the most common and practical automatic test generation method, especially for large programs. However, current techniques are not effective in exploring the large space of paths and thread interleavings for exposing concurrency bugs. *Model checking* is another method for exploring thread interleavings. However, the cost is too high to cover all the possible interleavings. Among these, [97] provides a new language that extends a simplified form of sequential Java to support multithreaded program model checking using the counterexample-guided abstraction-refinement framework; and *bounded model checking* via lazy sequentialization [52] is a systematic way to check sequentially consistent C programs that use POSIX threads. However, model checking methods incur cost of modifying current multithreaded programs. Also they do not provide support for concurrent dynamic data structures. Techniques presented in this thesis, based on symbolic execution and concolic testing, require minor changes to the source code, support dynamic data structures, and can be applied to multiple threads. The promise of symbolic/concolic execution is realized for testing concurrent dynamic data structures.

2.2 Symbolic/Concolic Execution

Symbolic Execution uses symbolic input instead concrete input to execute a program, solving the path constraints of a selected path to compute the possible input using an SMT solver. This achieves better coverage during testing because we can explore the path without giving a concrete input which is hard to provide manually or generate via a fuzzing engine. A symbolic execution engine will execute a program with symbolic input, and collect a set of path constraints (PC) by constructing a symbolic expression of each variable which consists of the branch conditions. However, there are several problems that make symbolic execution incapable of exploring all the paths: (i) *solver limitations* - the solver may not be able to solve a complex constraint in reasonable time (e.g., an expression containing a hash function); (ii) *execution environment* may not be symbolically explored (e.g., system calls and shared libraries); and (iii) *accessing data structures* using symbolic addresses.

To address these problems, concolic testing has been proposed by combining the concrete execution along with symbolic execution, so that a concrete variable that is feasible to a set of path constraints can be used to replace the symbolic variable to call system calls, simplify the unsolvable constraints, and access the data structures using the concrete address.

A typical symbolic/concolic execution engine consists of a constraint collector, an SMT solver, and a modeled execution environment (executor). The constraint collector will collect constraints while executing the program for each variable. There are two major kinds of constraints collector: (i) interpreter-based [23, 29, 64]; and (ii) compiler-based [83, 84]. In addition, once we meet branches and thread synchronization points during execution, there is a scheduler in the constraints collector used to determine the exploration order of paths and thread interleavings in presence of multithreading. The SMT solver is used to solve constraints and generate the feasible input. The modeled execution environment is used to model the execution environment of the program, like system calls and standard libraries.

Figure 2.1 presents the relationship between the three major components. Given symbolic inputs S, the constraint collector will select a Path P_i and the execution of path

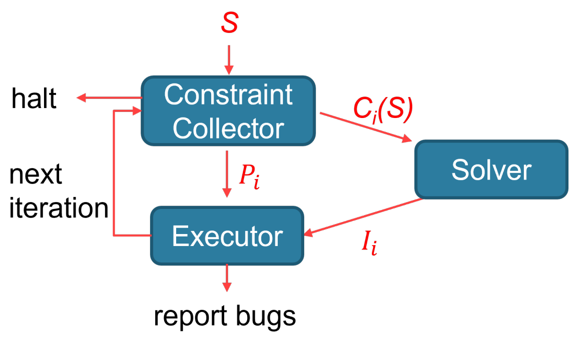


Figure 2.1: The overview of concolic testing.

P_i on S gives symbolic constraints C_i. The constraints are collected from branch conditions along path P_i. Then, the constraints C_i are sent to the solver when we need a concrete value to run some functions or we need to reduce execution complexity. It will produce the concrete inputs I_i for symbolic inputs S and send to the executor. Finally, the bugs are reported if it has and it moves to the next iteration to test path P_{i+1} using random or depth-first order until all user-expected paths are tested.

2.3 Concurrent Symbolic/Concolic Testing

Even though symbolic and concolic execution is a state-of-the-art technique for exploring all paths and generating test inputs; however, they also face path and thread interleaving explosion problems. KLEE [23] is one of the most well-known symbolic execution engines designed for LLVM IR. Cloud9 [29] is an extension of KLEE with added multithreading support for POSIX. GKLEE [64] extends KLEE to GPU programs. COMPI applies concolic testing to test MPI programs [65]. Con2colic [37] is another concolic testing method for multithreaded programs which also supports exploring the contents of a data structure and thread interleavings. However, in comparison to our approach, con2colic is based on a heuristic search rather than guided search, limiting its scalability. Moreover, it cannot explore data shapes for pointer-based data structures. While our focus is on C/C++ programs, jCUTE [94] is a concolic unit testing tool for multithreaded Java programs, and the data shape exploration is based on programming manually using data structure APIs. Thus, unlike the techniques presented in this dissertation, the above techniques do not explore dynamic data structure shapes and are expensive.

2.4 Optimization of Symbolic/Concolic Testing

A number of enhancements have been proposed to improve the efficiency of the above techniques. *Summarization* is one method to deal with path explosion – in [41] it is pointed out that compositional automatic test generation can scale to large programs with many feasible paths. This approach is further extended to interprocedural paths in [5]. *State merging* [62] is another way to solve the path explosion problem by combining similar states during execution. *Chopped symbolic execution* [104] can jump over some unrelated functions during symbolic execution, which can reduce the chance of forking new states. *Path subsumption* [113] proposes an annotation algorithm for branches and statements, which are implied by the current state. The above methods only focus on the path explosion problem but do not address other dimensions of data shape generation or thread interleavings. This dissertation employs summarization to guide and speedup thread interleaving exploration and speedup data shape exploration.

In all, there is no such method that can handle path explosion and thread interleaving explosion while at the same time exposing data races in large-scale dynamic concurrent data structures. Solving this problem is the aim of the proposed research. Our preliminary work on data shape generation via DSGEN [102] achieves good coverage needed to uncover data races by causing concurrent threads to follow selected paths that expose data races. Our symbolic execution-driven summarization of individual functions will further enhance the efficiency of concolic testing.

2.5 Testing of Concurrent Data Structures

Checking concurrent data structures for dynamic data race detection is a challenging task due to the different shape requirements of different paths. CDSChecker [78] provides a model-checking algorithm for modeling concurrent code under the C++ memory model. However, CDSChecker is not aimed for lock-based synchronization, and both the path explosion problem and thread interleaving explosion problem is not addressed by this algorithm. CDSSPEC [80] is a specification checker for the C++11 memory model. However, it requires the use of a specification language to describe the data structure and still has a high overhead. Shoal [6] is a system that extends SharC by grouping objects and providing sharing rules for each group. It can avoid data races for concurrent data structures by turning the data race detection problem into a sharing-rule violation detection problem. Our preliminary work on DSGEN [102] uses a data shape generation method for detecting data races in concurrent data structures of the CUDA platform.

Khurshid et al. [58] use symbolic execution to test library classes with generated set and map data structures in Java but do not consider user-defined data structures. Zhang [115] supports symbolic pointers and symbolic data structures. Burnim et al. [21], in addition, aim to create a worst-case input. Unlike the above techniques, CUTE [96] supports concolic unit testing for C programs with data structure generation support. However, unlike the approach presented in this dissertation, the above methods are not aimed at multithreaded programs and are not able to adequately test implementations of concurrent data structures. Path-based techniques in the presence of pointer-based dynamic data structures have also been developed by Korel and Bogdan [60, 61]. Chung and Bieman [28] generate data shapes using points-to information for statements along a selected path. Visvanathan and Gupta [106] employ a two-phase approach based on branch constraint solving to generate dynamic data structure structures – first, data shapes are generated to meet path constraints, and then values for data fields within data structures are generated. Saingern et al. [91] also handle linked data structures, including homogeneous and heterogeneous recursive structures. These methods are powerful yet they lack support for concurrent dynamic data structures in multithreaded CPU or GPU programs. Also, they are not integrated into a concolic testing framework and thus do not address coverage issues and they lack optimizations enabled via sub-paths sharing across many individual paths.

2.6 Data Race Detection

Both static and dynamic methods for data race detection have been widely studied [16,44,55,75,79,88,93,108]. They have their own advantages and limitations. Static race detection methods include *flow-insensitive type and language based* methods [10, 17, 24, 43] and *flow-sensitive lockset-based methods* [16, 36, 44, 55, 75, 92, 101, 108]. Dynamic race detection methods include *Happens-before* [1,31,39,76] and *Dynamic Locksets* [2,27,35,77,92,107] based analyses. The happens-before analysis does not result in false positives but is expensive. Dynamic lockset analysis relies on collecting memory access information and lock/unlock event tracing for race detection. However, it can lead to false positives. Thus, methods that combine both approaches for efficiency and accuracy have been proposed [32, 51, 79, 85, 114]. However, these methods rely on other techniques to test multiple paths and thread interleavings to expose races. The large search space, especially when we also consider dynamic data structure shapes, makes these methods very expensive.

Chapter 3

Data Shape Generation

For a single-thread program, the data shape influences the branch taken when there are symbolic pointers in the branch condition expressions. In multithreaded programs, data race detection also requires specific data shapes to exercise the path pair for a pair of raced threads. In this chapter, we will introduce the challenges of bug detection in programs with complex data shapes and the basic idea of data shape generation. In general, there are two reasons to use data shape generation: (i) One single shape is not enough to explore all the paths in a program due to the different constraints of symbolic pointers in different paths; and (ii) One single shape is not enough to explore all the potential bugs in a selected path (or a path pair in multithreaded programs) due to the pointer sharing in the data structure which may cause multithreaded bugs, e.g. data races. In the following sections, we will introduce those two problems and give the basic algorithm to generate data shapes to explore the paths and bugs.

In such programs, dynamic data structures with different shapes are needed to exercise different paths and explore more potential bugs. Thus, pointer variables that act as links to construct the dynamic data structure and its shape must also be made symbolic and then used to automatically generate dynamic data structures of suitable shape to exercise a given path by a thread. However, symbolizing pointer-based dynamic data structures is a big challenge for symbolic/concolic executors due to the large memory space and available possibilities of pointers.

In this chapter, we present data shape generation which is the key idea in symbolic/concolic execution to test multiple paths with different pointer constraints. We solve concrete values for those symbolic pointers which satisfied the *Pointer Constraints* (PC) and keep them adjustable for exploration.

3.1 Generating A Data Shape for a Selected Path

The dynamic data structure generation described in this section is inspired by the method proposed in [106] which initially assumes that the pointer variable that provides access to the data structure is simply null. Then, as it scans the code along the desired path, it collects constraints on the shape of the dynamic data structure and solves them to create the data structure of the desired shape. This approach is employed by our prototype DSGEN to create a concolic testing framework capable of exploring different data structure shapes based on a GPU concolic testing framework GKLEE.

During execution, when a memory access to a location marked as being part of a symbolic data structure is encountered, it is intercepted by GKLEE and passed on to DSGEN for handling. DSGEN collects relevant constraints, adapts the dynamic data structure shape to satisfy them, and passes the data structure to GKLEE so it can successfully execute the memory access. To achieve the above, DSGEN needs to collect two kinds of information – *Pointer Constraints* (PC) and *Pointer-Pointee Relations* (PPRs) – described below.

- **Pointer Constraints (PCs)** These are constraints that must be satisfied to ensure that the thread follows its selected path (e.g., branch conditions evaluate appropriately) and successfully executes pointer-based statements along the path (e.g., pointers that are dereferenced must not be null). When new data structure shapes are explored for the same path, each generated shape must continue to satisfy all of the path constraints.

- **Pointer-Pointee Relations (PPRs)** DSGEN must also track pointer-pointee relationships that are created by statements executed on all the selected paths. Each relationship is of the form (p, q) such that pointer p currently points to q. Therefore, pointer-pointee relationships essentially create the shape of the data structure.

Together, PCs and PPRs allow the exploration of shapes to exercise a given path as well as explore different paths. In particular, when generating an input to exercise a given path, PPRs are altered to create different shapes till eventually a shape is found to satisfy all the PCs, that is, paths followed by the threads are preserved in this process. When a new path is to be explored, a branch condition outcome is altered to explore a different path. This results in modifying the corresponding PC and then resumption of shape generation from the point at which branch outcome is altered to exercise the newly chosen path.

The data structure shapes formed by PPRs and the PCs associated with the fields belonging to a symbolic data structure are the result of DSGEN's actions that are determined by the kind of operations encountered: *pointer initialization*, *pointer dereferencing*, *pointer* *assignments*, and *branch conditions*. For example, first-time dereferencing of a pointer typically causes memory allocation that expands the data structure and generates constraints indicating that the pointer is no longer null. Pointer assignments generate constraints causing different symbolic pointers to share the same address and thus contribute to the formation of data structure shape. Branch conditions may themselves involve pointer dereferencing, and branch outcomes may assert that a pointer is null or not null. Next, we demonstrate our approach for capturing pointer constraints and their role in exposing data races using the example of the concurrent skip list data structure.

An Example. Consider the code in Listing 3.1 which presents two operations for a skip list - `insert` and `search` for inserting in an ordered list and searching for a node corresponding to a key value. The function `search_node` is a common function used by both `insert` and `search` functions to find the node which contains key k.

Necessity of Shape Exploration in Path Testing. Some paths require a certain shape to explore due to the branch conditions containing constraints related to the pointers. In the skip list shape in Figure 3.1, assuming passed `N0`, `0`, `0`, and a local memory buffer `pre` to `search_node`, the branch condition h->next[i]!=NULL at line 13 will take false. However, with the same passed parameters, the skip list shape in Figure 3.2 will take true since there is a node `N1` at the `next[0]` field. Since data shapes influence the ability to successfully exercise a given path, more bugs may be exposed while more paths are explored using different shapes.

```c
 1  #define MAXLEVEL 2
 2
 3  typedef struct Node {
 4      key_t key; val_t value;
 5      struct Node* next[MAXLEVEL];
 6      pthread_mutex_t mutex;
 7  } Node;
 8
 9  Node* search_node(Node* h, key_t k,
10      int i, Node** pre) {
11      klee_assume(i >= 0 && i < MAXLEVEL);
12      Node* next = NULL;
13      if (h->next[i]!=NULL && h->next[i]->key<k)
14          next = h->next[i];
15      if (next != NULL)
16          return search_node(next, k, i, pre);
17      pre[i] = h;
18      if (i == 0) return h->next[i];
19      return search_node(h, k, i-1, pre);
20  }
21
22  bool insert(Node* h, key_t k, val_t v) {
23      Node* prev[MAXLEVEL];
24      Node* curr = search_node(h, k, 1, prev);
25      if (curr != NULL && curr->key == k) {
26          if (curr == prev[1]->next[0])
27              prev[0] = curr;  // complex
28          else curr->next[0]->value = v; // simple
29          curr->value = v; return false;
30      }
31      Node* node = create(k, v);
32      int level = rand_level();
33      pthread_mutex_lock(&(prev[0]->mutex));
34      node->next[0] = prev[0]->next[0];
35      prev[0]->next[0] = node;
36      if (level == 1) {
37          node->next[1] = prev[1]->next[1];
38          prev[1]->next[1] = node;
39      }
40      pthread_mutex_unlock(&(prev[0]->mutex));
41      return true;
42  }
43
44  val_t search(Node* head, key_t key) {
45      bool has_node = true;
46      pthread_mutex_lock(&(head->mutex));
47      if (head->next[0] == NULL) has_node = false;
48      pthread_mutex_unlock(&(head->mutex));
49      if (!has_node) return -1;
50      Node* prev[MAXLEVEL];
51      Node* curr = search_node(head, key, 1, prev);
52      if (curr != NULL && curr->key == key)
53          return curr->value;
54      return -1;
55  }
```

Listing 3.1: A Concurrent Skip List Example.

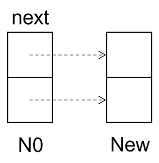


Figure 3.1: A data shape that cannot expose the race.

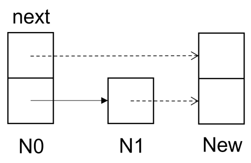


Figure 3.2: A data shape that exposes the race.

line#	Condition	Eval.
25	curr != NULL	T
25	curr→key == k	F
26	level == 1	T

line#	pointer	pointee
25	curr	N0
33	prev[0]	N1
34	node	N2
35	prev[0]→next[0]	N2
37	prev[1]	N3
38	prev[1]→next[1]	N2

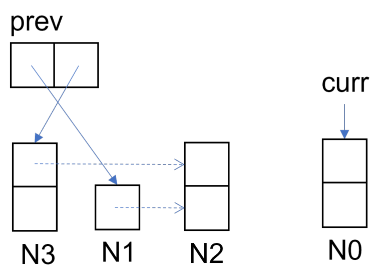


Table 3.1: Pointer-pointee relations (and its visualization) for `insert` function along path with branch conditions values `T F T`.

Capturing Pointer Constraints. Consider the path with branch predicate evaluations of `T F T` in function `insert`. Along this path, the data shape generated is captured by the *pointer-pointee* relationships given in Table 3.1. The corresponding data structure that satisfies these *pointer-pointee* relationships is also shown. All constraints in branch conditions indicate the value of a pointer belongs to *pointer constraints*, so `curr != NULL` will be collected at line 25. Dereferenced pointers can not be empty is another kind of pointer constraint, so `prev[0] != NULL`, `node != NULL`, `prev[1] != NULL` will be collected. These are created by statements along the path as follows: (i) line 25 dereferences pointer `curr`; (ii) line 33 dereferences pointer `prev[0]`; (iii) line 34 dereferences `prev[0]` and assigns the value of `node` to `prev[0]`→`next[0]`; and (iv) line 35 while dereferences `prev[1]` and assigns it to `prev[1]`→`next[1]`.

Solving Pointer-Pointee Relationship (PPRs). Dereferencing implies non-null pointers that point to other nodes as shown in Table 3.1. We first assume all the pointers dereferenced are pointing to different memory if it can not be inferred from branch constraints. `curr`, `prev[0]`, `node`, and `prev[1]` have been assigned `N0`, `N1`, `N2`, and `N3` respectively. And `prev[0]->next[0]` and `prev[1]->next[1]` has been assigned to `node` at line 35 and line 38. The pointer-pointee relationship forms the current shape but is only one case that satisfies the pointer constraints. We can adjust it as needed in the future.

3.2 Exploring Multiple Shapes for a Selected Path

During the exploration of shapes to explore more paths, usually, there are multiple solutions satisfies the selected path. However, a single shape is not enough to expose the bug for multithread programs in some cases. In DSGEN, pointer constraints (PCs) must be satisfied but the pointer-pointee relationship can be adjustable. We can compute different groups of pointer-pointee relationships which satisfied the pointer constraints.

Necessity of Shape Exploration for Concurrent Bugs. Manifestation of the above data race requires a skip list of a certain shape. In the skip list of Figure 3.1, `prev[0]` and `prev[1]` refer to the same node. Therefore the lock at line 33 can protect node `prev[1]` which is being modified and hence the data race between line 38 and line 14 does not occur. However, in Figure 3.2, the skip list shown exposes the data race. Thus we see that it is necessary to explore appropriate data shapes to expose the desired data race. Since data shapes influence the node referenced in memory load/store expressions and lock operations, there are concurrent bugs may be exposed if more shapes are explored. At the same time, shape exploration plays a role in constructing shapes for multithreaded programs. When a shape is suitable for one thread but not the others, we have a chance to construct another shape suitable for both threads.

To help explore the data shapes, we designed a method PREDICT$(attr, s)$, which identifies a new predicted value for $attr$ such that it satisfies all the constraints in s. For example, when making a pointer non-null, predictions considered include setting the pointer to point to: newly allocated memory, itself creating a self-loop, or an existing object of the appropriate type.

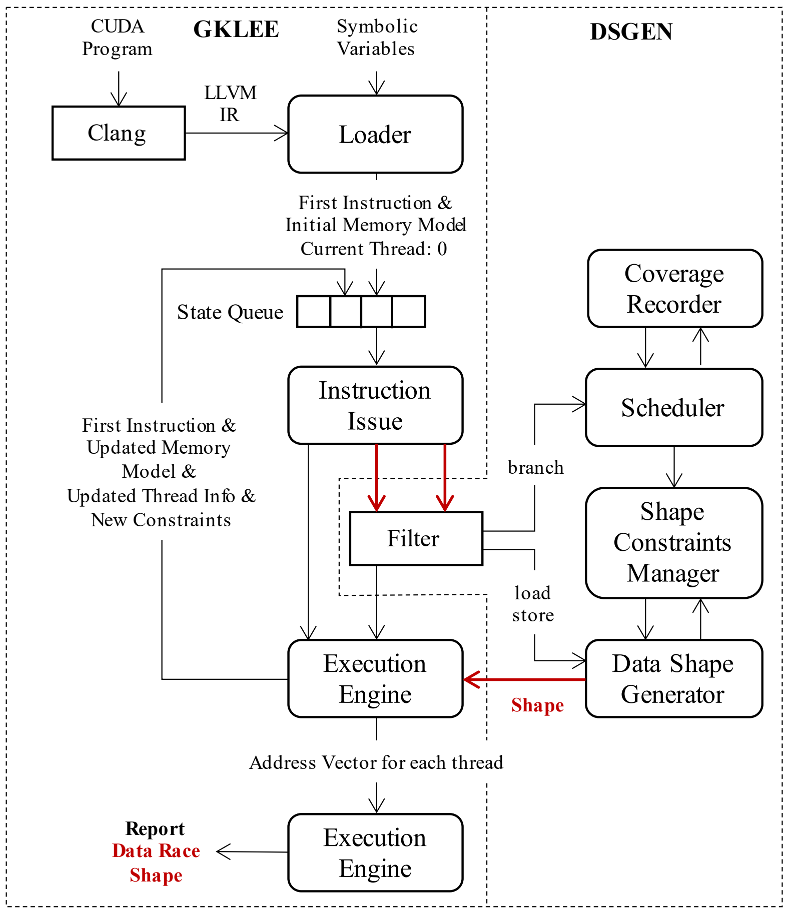


Figure 3.3: DSGEN + GKLEE Prototype.

3.3 Shape Generation and Exploration in DSGEN prototype

Figure 3.3 provides an overview of the DSGEN based on a GPU concolic testing engine - GKLEE system. GKLEE gives instructions to the Filter module that passes on the memory accesses of the symbolic concurrent data structure to the Shape Generator and the branch conditions to the Scheduler. All other instructions are passed directly to GKLEE's execution engine. The Scheduler provides branch coverage information for all threads to the Coverage Recorder and selects new paths to explore. Note that for branch conditions that are symbolic, both true and false outcomes can be explored by path selection. The Scheduler also provides constraints that arise from branch conditions to the Shape Constraints Manager. The selection of alternate paths leads to the modification of these constraints. The Data Shape Generator generates a data structure with a shape that satisfies constraints and passes it on to GKLEE.

The Shape Constraints Manager also performs another important task. It is responsible for ensuring the generation of a non-conflicting data structure. Thus, when data structures produced along paths followed by different threads differ, the constraints manager must detect and resolve conflicts among them to produce a single non-conflicting dynamic data structure shape for all the threads. Resolution of conflicts results in the generation of a test case that exercises the path combination. If conflicts cannot be resolved, the current combination of paths is abandoned. The search then moves on to the next combination of paths that are selected according to the depth-first search strategy.

The functioning of the Data Shape Generator is driven by the memory accesses. Starting from the previously generated shape, this module appropriately modifies the data structure. As an example, a pointer dereferencing operation may lead to the expansion of the data structure via memory allocation. On the other hand, when conflicts are to be resolved, the data structure may need to be compacted. The actions of this module are at the heart of DSGEN function and will be presented in detail in Section 4.

Table 3.2 lists the newly provided APIs that allow the programmer to identify the dynamic data structure that is to be automatically generated and whose shapes are to be explored. The function `klee_make_data_structure` makes x, which is a pointer or an array

Table 3.2: DSGEN API for dynamic data structures.

klee_make_data_structure (*x*, *size*, *name*)
klee_set_data_structure (*x*, *size*, *name*, *function*)
klee_set_double_link (*name*, offset, size, link, link_size)
klee_set_range (*name*, offset, size, min, max)
klee_set_memory_type (name, offset, size, type)

of pointers of given *size*, symbolic and assigns a *name* to the data structure that it provides access to. This indicates to DSGEN that data structure must be automatically generated, its constraints collected, and its shapes explored. *Thus, as the data structure grows, all newly created pointer fields must also be marked as symbolic.*

In certain situations, a data structure that is not automatically generated (e.g., generated by the user by a manually written code) may need to be added to the symbolic data structure and thus requiring that its pointer fields be made symbolic. The function `klee_set_data_structure` provides this functionality. An additional parameter *traverse* is provided by the programmer that fully traverses the data structure to collect the addresses of all contained pointer fields so that `klee_set_data_structure` can mark them also as symbolic. Since doubly-linked data structures are frequently used, the next API function allows user to express their presence which simply guides the shape generation. Finally, the last two APIs simply express a valid range of addresses and the kind of memory where it resides.

Exploring Execution States. The GKLEE's VM creates the state space for exploration as follows. As a thread is being symbolically executed, if the VM determines that based upon the current symbolic values an outcome of a *condition* can be either true or false, it forks off new states for true and false outcomes. Repeated forking creates a tree structure representing a partitioning of all execution states – by exploring different paths in the tree, coverage over program paths is achieved. As is generally the case for concolic testing tools, the features modeled by the symbolic execution model can be exhaustively explored during testing. However, under the constraints of the testing time budget, different strategies may be deployed to prioritize the exploration of state space. GKLEE supports multiple search strategies, and in this work, we relied on *depth-first exploration* of state space to identify data races. Note that the predicates that cause forking of states can be independent of threads or they can depend upon thread and block ids (denoted as tid and bid). In the latter case, forking essentially partitions threads prior to fork into two classes of threads.

When it comes to dynamically linked data structures, GKLEE does not provide any special support. It handles pointer variables using the simple methodology used by the underlying KLEE system. Unfortunately, this makes input generation when testing functions of a library implementing concurrent data structures a problem. To exercise execute states of such a function, the input to the function must be an appropriately shaped and sized dynamic linked data structure. Unfortunately, KLEE is incapable of exploring the space of different shaped and sized data structures. Driven by the API already described, DSGEN, through its special treatment of pointers, is able to explore the execution states that must honor different constraints on these pointers (such as, pointers being null or non-null, shape forming pointer-pointee relations, etc.). Since dynamic data structure can grow arbitrarily large, to constrain the execution space, two configuration parameters are provided that limit *sizes of arrays* used and the number of *levels of links* allowed. Limits on array sizes and the

Table 3.3: DSGEN vs. GKLEE: # of paths explored and data structure shapes generated.

		Shapes Generated	# of Path Combinations
B-Tree	GKLEE	1	126
	DSGEN	1629	2667
HAMT	GKLEE	1	47
	DSGEN	122	282
RRB-Trees	GKLEE	1	6
	DSGEN	16	16
Skip List	GKLEE	1	64
	DSGEN	130	256

number of levels of links limit the size of the dynamic data structure which translates into limits in the lengths of paths that are explored during depth-first exploration of paths.

Experiments of Path Exploration. We use GKLEE to explore paths using only one manually created data structure comparing to our automatically generated data structures. The result shows in Table 3.3, and we can notice using automatic data shape generation explored much more paths in all 4 benchmarks - B-Tree, HAMT, RRB-Trees and Skip List.

Chapter 4

Consistent-Shape Generation for Multithreaded and GPU Programs

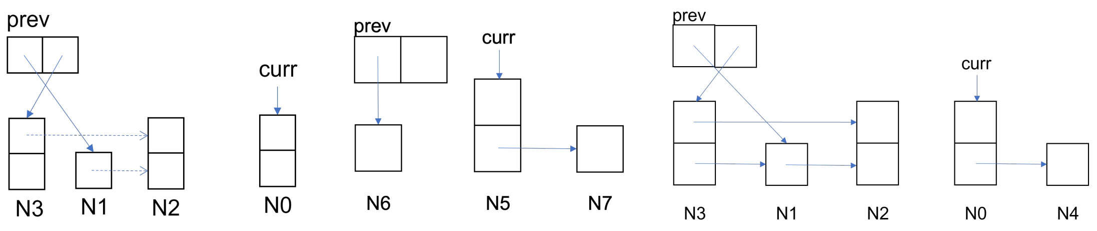
Data shape generation that we present automatically generates suitable dynamic data structures with shapes that exercise desired paths, same or divergent. The algorithm has two steps: generating data structures for each thread separately using the technique presented in Chapter 3; and integrating the generated data structures into one non-conflicting data structure. The latter is the subject of this chapter.

4.1 Integrating Data Structures for Different Threads

We first consider a simpler situation, in which the per-thread data structures can be compacted such that parts of the newly formed data structure come either from one thread's data structure or the other thread's data structure, or they were present in both

line#	**Path Constraints: Thread a**	line#	**Path Constraints: Thread b**
25	curr $\neq$ NULL	25	curr $\neq$ NULL
25	curr$\rightarrow$key $\neq$ k	25	curr$\rightarrow$key == k
36	level == 1	26	curr $\neq$ prev[1]$\rightarrow$next[0]

	PPRs: Thread a			**PPRs: Thread b**	
line#	Pointer	Pointee	line#	Pointer	Pointee
25	curr	N0	25	curr	N5
33	prev[0]	N1	26	prev[1]	N6
34	node	N2	28	N5.next[0]	N7
35	N1.next[0]	N2			
37	prev[1]	N3			
38	N3.next[1]	N2			



(a) Thread a (`T F T`). (b) Thread b (`T T F`). (c) Integrated for Both Threads.

Figure 4.1: A simple example of Skip-List data structure integration.

per-thread data structures. We refer to this as taking the *union* of the data structures. This form of compaction is a simple combining of two data structures without violation of any constraints and it will be illustrated when generating a shape that exposes the first race as shown in Figure 4.5.

We have shown in detail how the constraints for Thread a are collected and the data shape in Figure 4.1(a) is generated in chapter 3. Similar actions for thread b generate the shape in Figure 4.1(b). Next, we will present the algorithm that integrates the two shapes into one data structure that is shown in Figure 4.1(c). The per-thread data structures generated satisfy their respective PCs and now they must be integrated to satisfy PCs for the threads simultaneously.

Given two threads, T_a and T_b, their path constraints $PC(T_a)$ and $PC(T_b)$, and pointer-pointee relations $PPR(T_a)$ and $PPR(T_b)$, we make the following key observations:

- **Feasibility** – Since the integrated data structure must simultaneously satisfy path constraints in $PC(T_a)$ and $PC(T_b)$, the presence of a pair of conflicting constraints in $PC(T_a)$ and $PC(T_b)$ implies that no such integrated data structure exists. That is, the *feasibility* of threads T_a and T_b simultaneously following the chosen paths requires that $PC(T_a)$ and $PC(T_b)$ be *conflict-free.*

- **Adjustment** – The integrated data structure cannot in general be obtained by taking the union of $PPR(T_a)$ with $PPR(T_b)$. This is because corresponding fields in $PPR(T_a)$ and $PPR(T_b)$ may conflict with each other, i.e. have different pointees. Therefore integration essentially involves *adjustment* of PPRs to make them consistent such that the adjustments do not violate any constraints in $PC(T_a)$ and $PC(T_b)$, i.e., paths followed are preserved.

A simple example of data shape integration. Figure 4.1 shows the integration of data shapes for paths `T F T` and `T T F`. Thread a, which follows the path `T F T`, generates the data shape in Figure 4.1(a). Another thread b, which follows the path `T T F`, generates the data shape in Figure 4.1(b). Two shapes are integrated into one such that both threads follow the same respective paths. The integrated data shape that is used to detect potential data races is shown in Figure 4.1(c). The data shape integration consists of three steps: 1) `prev` in thread a's shape contains pointers that do not exist in thread b's shape, so the reachable data shapes from `prev` are merged into the final data shape without any changes;

Algorithm 1: An algorithm for integrating two data shapes.

```
Input: Pointers $x$ and $y$ that point to two data structures that need to be
       compacted into a single non-conflicting data structure.
Output: Pointer $x$ that now points to the compacted non-conflicting data structure
Procedure COMBINE($x$, $y$):
    if test_sets_conflict(cons($x$), cons($y$)) then
        return COMBINING FAILED
    foreach $attr \in x$ do
        if (type($x.attr$)$\neq$pointer) then
            continue ▷ non-pointer details omitted
        if val($x.attr$) $= \tau$ || val($y.attr$) $= \tau$ then
            if val($y.attr$)$\neq\tau$ then
                val($x.attr$)$\leftarrow$val($y.attr$)
                ppr($x.attr$)$\leftarrow$ppr($y.attr$)
            cons($x.attr$)$\leftarrow$cons($x.attr$)$\cup$cons($y.attr$)
        else if val($x.attr$)$\neq$val($y.attr$) then
            $A_x \leftarrow$ adjustable(ppr($x.attr$), cons($x.attr$))
            $A_y \leftarrow$ adjustable(ppr($y.attr$), cons($y.attr$))
            if $\neg A_x$ & $\neg A_y$ then
                return COMBINING FAILED
            else if $A_y$ & $\neg A_x$ then
                RESOLVE($y.attr$, $x.attr$)
            else if $A_x$ & $\neg A_y$ then
                RESOLVE($x.attr$, $y.attr$)
            else
                COMBINE(val($x.attr$), val($y.attr$))
                cons($x.attr$) $\leftarrow$ cons($x.attr$) $\cup$ cons($y.attr$)
    return COMBINING SUCCEEDED
Procedure RESOLVE($\alpha, \beta$):
    foreach $(z.attr, s) \in$ ppr($\alpha$) do
        foreach $pc \in$ cons($\beta$) do
            if not has_conflict(ppr($\alpha$), $pc$) then
                continue
            ppr($\alpha$) $\leftarrow$ ppr($\alpha$) - DEPEND($(z.attr, s)$)
            acons $\leftarrow$ SIMPLIFY($(z.attr)$, cons($\alpha$) $\cup$ cons($\beta$))
            if acons $\notin$ const then
                ($succ$, $c$) $\leftarrow$ PREDICT($(z.attr)$, cons($\alpha$) $\cup$ cons($\beta$))
                if not succ then
                    COMPACTION FAILED
                ppr($\alpha$) $\leftarrow$ ppr($\alpha$) $\cup\, c$
                val($z.attr$) $\leftarrow$ SIMPLIFY($(z.attr)$, cons($\alpha$) $\cup$ cons($\beta$))
                cons($\alpha$) $\leftarrow$ cons($\alpha$) $\cup$ cons($\beta$)
            else
                ppr($\alpha$) $\leftarrow$ ppr($\alpha$) $\cup$ ($z.attr$, acons)
                val($z.attr$) $\leftarrow$ acons
                cons($\alpha$) $\leftarrow$ cons($\alpha$) $\cup$ cons($\beta$)
            break
```

2) pointer `curr` is pointing to different memory objects in shapes for threads a and b, that is, `N0` and `N5` respectively. The integration of memory objects `N0` and `N5` can be done by transferring each field in `N5` to `N0` if there is no conflict, i.e. the path constraints are not violated; and 3) `N5.next[0]` points to the memory object `N6` in thread b. This pointer is transferred to `N0.next[0]`. So the data shape in Figure 4.1(c) is generated and used for confirming the potential race.

Next, we present Algorithm 1 that, guided by the above observations, *explores different adjustments* to PPRs so they can be made consistent without violating PCs. When conflicts among PCs are found, the algorithm reports that no integration is possible. More specifically, COMBINE takes as its inputs two pointers x and y that point to per-thread data structures and modifies the first pointed to by x into an integrated one. for the two threads. In Algorithm 1, given a field fld in a symbolic data structure, $val(fld)$ provides the value of a pointer fld which can be *untouched* (τ), *null*, a *concrete address*, or a *symbolic expression*. The $cons(fld)$ denotes the set of subset of path constraints that involve fld. Note that we only focus on pointer fields because they form the shape of the data structure and mechanisms for data fields are already supported by GKLEE.

Lets us now consider the functioning of COMBINE(x,y) where x and y are pointers that point to the start nodes of the data structure. Lines 2-3 test for conflicts among path constraints of x and y, and if one is found, combining is aborted; otherwise, each attribute field of x and y are considered for combining. Lines 7-11 considers the case where the attribute of x is untouched (i.e., τ) and hence the attribute of y is simply adopted by x as this combining will not violate any path constraints. Lines 12-23 consider cases where attribute values are not equal and not untouched. For combining, they must be made equal by adjusting one or both of them. The *adjustable* returns true or false indicating whether or not an attribute's PPR can be adjusted without violating corresponding PCs. Based upon outcomes A_x and A_y, if possible, search for adjustments is carried out. If attribute of only x or only y is adjustable, then the adjustment of the adjustable one is carried out via call to RESOLVE. If both are adjustable, a recursive call to COMBINE is used to adjust both attributes which will cause them to have the same value.

During integration, COMBINE makes use of the RESOLVE(α, β) procedure that removes those PPRs from α node that conflicts with PCs in the β node. Given a single PPR *ppr*, DEPEND$(lhs(ppr))$ returns all the constraints that can be inferred directly or indirectly from p. Also PREDICT$(attr, s)$ identifies a new predicted value for *attr* such that it satisfies all the constraints in s. For example, when making a pointer non-null, predictions considered include setting the pointer to point to: newly allocated memory, itself creating a self-loop, or an existing object of the appropriate type. Note that SIMPLIFY(c, α) is a GKLEE method that simplifies the constraint c using the set of facts in α and `has_conflict`(p, c) is another method that detects conflicts between a pointer-pointee relation p and a set of path constraints c, if p does not satisfy the PC set c.

Now let us consider an illustration of integration performed by COMBINE. The first set of situations (lines 7-11) arise when at least one of corresponding pointer fields is untouched, i.e. τ. Here the integrated data structure adopts the non τ value if one exists or it is τ when both fields are τ. This situation alone is sufficient for integrating the shapes in Figure 4.1 (c). The following execution call trace shows the steps of integration:

```
COMBINE ($N0$, $N4$)
11  cons($N0$.child[0]) $\leftarrow$ cons($N0$.child[0]) $\cup$ cons($N4$.child[0])
22  COMBINE ( val($N0$.next), val($N4$.next) )
9     val($N3$.child[0]) $\leftarrow$ val($N5$.child[0])
10    ppr($N3$.child[0]) $\leftarrow$ ppr($N5$.child[0])
11    cons($N3$.child[0]) $\leftarrow$ cons($N3$.child[0]) $\cup$ cons($N5$.child[0])
23  cons($N0$.next) $\leftarrow$ cons($N0$.next) $\cup$ cons($N4$.next)
```

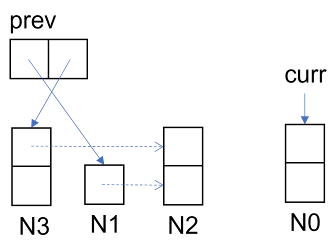
Initially COMBINE is called with parameters $N0(t0)$ and $N4(t1)$. The shapes generated by threads a and b are such that data structures rooted at the two remaining fields, `N0.child[0]` and `N5.child[0]`, are untouched τ in exactly one of the threads. So the trace follows the true branch at line 7, updating cons($N0$.child[0])) (line 11) at first. In the next iteration, it's calling COMBINE (line 22) for their `next` field since both thread contains valid pointer and the $PPR(T0)$ and $PPR(T1)$ are both adjustable. During the second recursive call of COMBINE, val($N3$.child[0]) (line 9) and cons($N3$.child[0]) (line 10) will be updated. After handling the sub-combination in `next` field, cons($N0$.next) will also be updated (line 23). Therefore their integrated data structure adopts the non τ values for these fields leading to the integrated data structure in Figure 4.1(c). The execution of COMBINE and hence the integration is complete. Note that the paths followed by the threads are preserved.

4.2 Creating a Non-Conflicting Data Structure

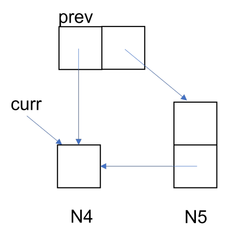
Sometimes, the integration is not always perfectly generate a data shape. A more complex situation is one in which *adjustments* to data structure shapes are made during the compaction process as will be illustrated when generating a shape that exposes the second race of our example. *Note that the compaction of per-thread data structures, always preserves*

line#	**Path Constraints: Thread a**	line#	**Path Constraints: Thread b**
25	curr $\neq$ NULL	25	curr $\neq$ NULL
25	curr$\rightarrow$key $\neq$ k	25	curr$\rightarrow$key == k
36	level == 1	26	curr == prev[1]$\rightarrow$next[0]

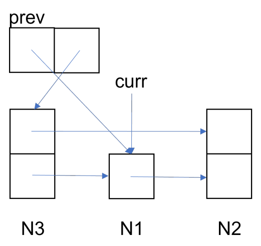
	PPRs: Thread a			**PPRs: Thread b**	
line#	Pointer	Pointee	line#	Pointer	Pointee
25	curr	N0	25	curr	N4
33	prev[0]	N1	26	prev[1]	N5
34	node	N2	26	N5.next[0]	N4
35	N1.next[0]	N2	27	prev[0]	N4
37	prev[1]	N3			
38	N3.next[1]	N2			



(a) Thread a (`T F T`).



(b) Thread b (`T T T`).



(c) Integrated for Both Threads.

Figure 4.2: A complex example of Skip-List data structure integeration.

paths followed by threads. After succeeding or failing to generate a test input that exercises the current path combination, concolic execution considers another path combination.

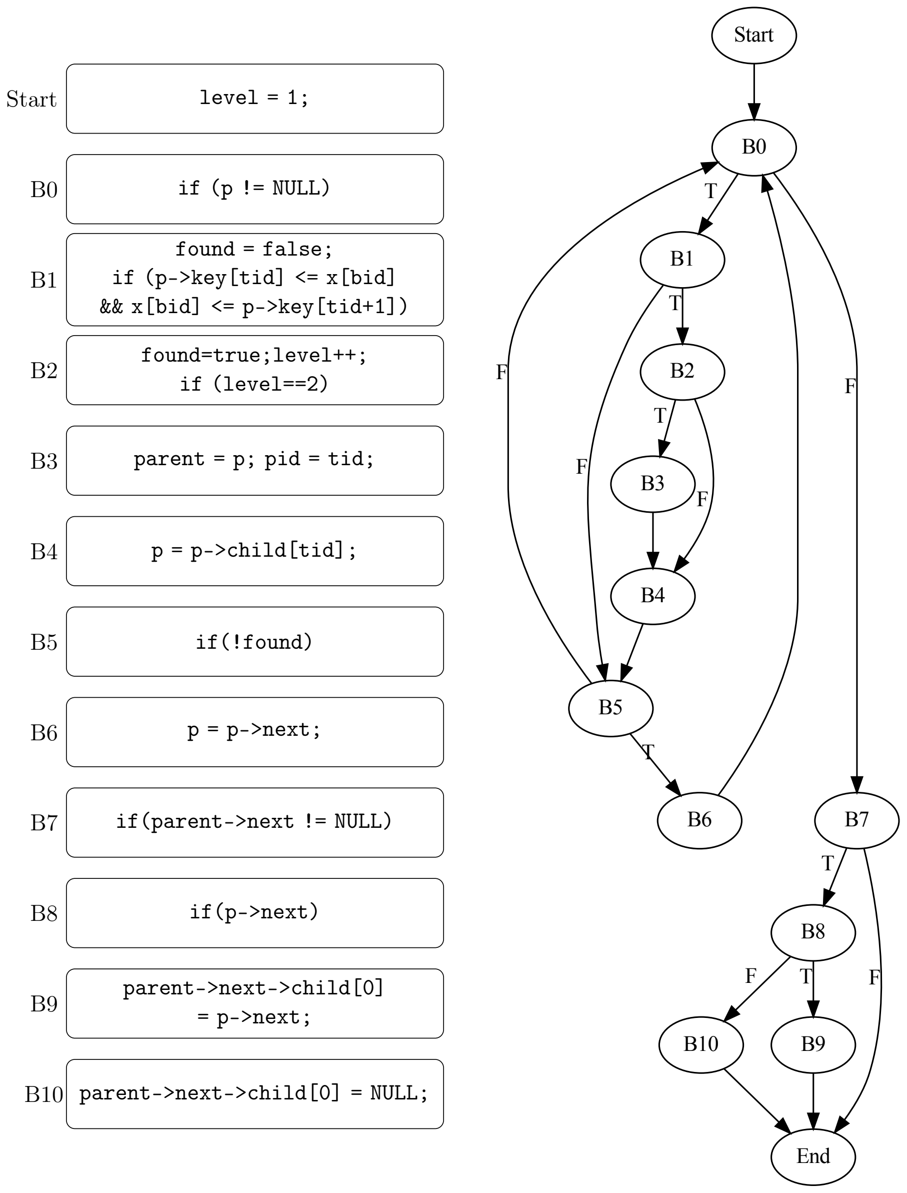
A complex example of data shape integration. Figure 4.2 shows the integration of data shapes for paths `T F T` and `T T T`. The data shapes of threads a and b are shown in Figure 4.2(a) and (b), respectively. In Figure 4.2(c), the integrated shape is created using the following steps: 1) Since both threads contain `prev` and `curr`, the pointer `prev[0]` is first integrated. However, a conflict of `prev[0]` pointing to `N1` for thread a and `N4` for thread b respectively is detected. The conflict is resolved by setting the pointer `prev[0]` is pointing to `N1` in thread b due to its pointee being *modifiable.* Observe there may be multiple ways to resolve the conflict which can be explored if needed. All the pointers pointing to `N4` are modified to `N1` so that the pointer `N5.next[0]` and `curr` are also changed to `N1`; 2) Then, The pointer `prev[1]` is integrated by transferring `N5` to `N3` and the pointer `N5.next[0]` is also transferred so the PPR `N3.next[0]` pointing to `N1` is created; and 3) Because pointer `curr` in thread b points to `N1`, and is not modifiable, the `curr` pointer in integrated result uses the PPR `curr` pointing to `N1`.

Next, we consider a more complex situation if the branch at line 28 in Listing 3.1 takes true path where non-τ values are found in corresponding fields of data structures generated by the two threads and PPRs conflicts are involved. In Figure 4.2, we first show the PCs and PPRs, and then the two data structures generated are given in Figures 4.2(a) and (b). Note that in this example the `N3.next` and `N5.next` are untouched (τ) in both data structures while `child[0]` fields are untouched (τ) in each bottom level node. Thus, their integration of related fields is non-conflicting.

On the other hand, the field `N3.child[0]`, that is non-null in both data structures, requires integration. This integration is carried by lines 12-23 of the Algorithm 1.

4.3 Race Detection in DSGEN prototype for GPU programs

Since making a thread follow a path is dependent upon the dynamic data structure and its shape, the objective of concolic testing is to generate dynamic data structures with different shapes to explore executions along different paths by multiple threads. To accomplish this task, our system enables two key functions. First, it allows pointers that construct the concurrent dynamic data structures to be made symbolic. Second, it allows



```
Start   level = 1;
B0      if (p != NULL)
B1      found = false;
        if (p->key[tid] <= x[bid]
        && x[bid] <= p->key[tid+1])
B2      found=true;level++;
        if (level==2)
B3      parent = p; pid = tid;
B4      p = p->child[tid];
B5      if(!found)
B6      p = p->next;
B7      if(parent->next != NULL)
B8      if(p->next)
B9      parent->next->child[0]
        = p->next;
B10     parent->next->child[0] = NULL;
```

Figure 4.3: The Control Flow Graph of `cu_skiplist_search`.

the collection of constraints on data structure shapes that must be satisfied to cause the selected paths to be followed by respective threads.

SkipList Example Next we illustrate the use of above APIs and the functioning of our system from the user's perspective. For this purpose, we make use of the example that employs concurrent blocked SkipList data structure. The application code in supports searching of a batch of keys in the SkipList by calling `cu_skiplist_search` function and maintains the number of jumps in the data structure. The control flow graph of the `cu_skiplist_search` function is given in Figure 4.3. Our objective is to generate inputs to enable testing of this very function.

To cause automatic generation of the SkipList dynamic data structure, and exploration of different shapes, we mark the `root` node of SkipList as a symbolic pointer using the new API function `klee_make_data_structure` at line 36. The other non-pointer fields in the SkipList nodes are marked symbolic using GKLEE's `klee_make_symbolic` function at line 38. In addition, symbolic `threadIdx` and `blockIdx` are also maintained by GKLEE to generate fewer (typically two) threads.

In implementing the function `cu_skiplist_search`, we have manually introduced data races. We consider the following two data races in this function for illustrating DSGEN.

1. The *first read-write race* arises between read access of the pointer field `p->child[0]` at line 21 and write access of the pointer field `parent->next->child[0]` at line 30 that handles the situation in which a search requires updating of the linked hierarchy of the parent node at line 30, while another thread is reading the child field at the same node concurrently at line 21.

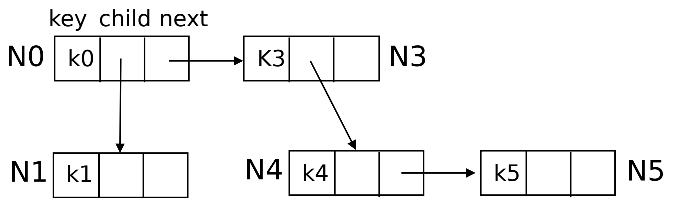
2. The *second read-write race* is between read access of the pointer field `p->child[0]` at line 21 and write access of the pointer field `parent->next->child[0]` at line 29 where

the updating of `child[0]` field at line 29 conflicts with reading of the same node by another thread at line 21.

Note that to test the `cu_skiplist_search` function the main program specifies grid size of 2 and block size of 1, giving us two threads: Thread a with $(bid = 0, tid = 0)$ and Thread b with $(bid = 1, tid = 0)$.

Using two threads and corresponding selected paths, the data races may be exposed by some path pairs, not exposed by other pairs, and different races may be exposed by different path pairs. For example, no data races arise for the path pair shown in Figure 4.4 which takes a false branch at line 27. However, the first race is exposed by another path pair shown in Figure 4.5 where Thread a executes line 21 and Thread b takes true branch at line 27 but the false branch is taken at line 28 causing line 30 to be executed. By exploring path pairs we can uncover data races. The paths taken depend upon the differing shapes of the data structure – for data structure in Figure 4.4 the path taken does not cause a read-write race while for data structure shape in Figure 4.5 data race arises because updating of the parent node is required.

Note that if the user were to write the code to construct the concrete data structure shown in Figure 4.4, then concolic testing performed by GKLEE will not be able to alter the outcomes of these branch conditions and the condition `if (parent->next!=NULL)` in B7 will never be true; thus, parent node update will never occur and the race will not be exposed. Even if, by coincidence, the user constructs a data structure that satisfies the conditions for discovering the data race, it may not be able to use one data structure to find all the races in different paths with different conditions if `(p->next!=NULL)` or not. On



Threads a & b follow the following paths

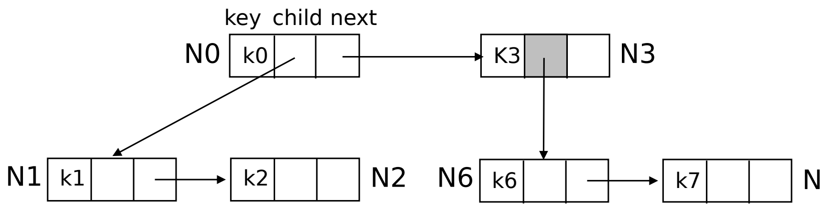
Thread a bid=0, tid=0	Start - B0 T - B1 T - B2 F - B4 - B5 F - B0 T - B1 T - B2 T - B3 - B4 - B5 F - B0 F - B7 F - End
Thread b bid=1, tid=0	Start - B0 T - B1 F - B5 T - B6 - B0 T - B1 T - B2 F - B4 - B5 F - B0 T - B1 F - B5 T - B6 - B0 T - B1 T - B2 T - B3 - B4 - B5 F - B0 - B7 F - End

Figure 4.4: A Skip-List Shape that does not expose either the first or second data race.

the other hand, when the user makes the data structure symbolic using the DSGEN's API, DSGEN is able to generate the new shape shown in Figure 4.5 that causes the desired path to be followed and making condition in B7 to evaluate to true and generate two different shapes depends on the condition in B8. This triggers parent node updating and exposes the data races that are identified using the collected traces.

While concolic testing is meant to explore different paths, it cannot achieve exploration of paths without making pointer-based linked data structure symbolic. This is because the path conditions in basic blocks B7 and B8 depend upon the shape of the dynamic data structure. Only by making the dynamic data structure and its pointer fields symbolic, and exploring different shapes, can the desired paths be exercised.

Next, let us see how path exploration is carried out by a thread in DSGEN. In particular, in Figure 4.6 we show part of the path search space (full space is too large to show) where some of the neighboring paths that will be explored via depth-first search



Threads a & b follow the following paths

Thread a bid=0, tid=0	Start - B0 T - B1 T - B2 F - B4 - B5 F - B0 T - B1 F - B5 T - B6 - B0 T - B1 T - B2 T - B3 - B4 - B5 F - B0 F - B7 T - B8 F - B10 - End
Thread b bid=1, tid=0	Start - B0 T - B1 F - B5 T - B6 - B0 T - B1 T - B2 F - B4 - B5 F - B0 T - B1 F - B5 T - B6 - B0 T - B1 T - B2 T - B3 - B4 - B5 F - B0 - B7 F - End

Figure 4.5: A Skip-List Shape that exposes the first data race but not the second race.

are shown. The predicate outcomes along the path are shown and data structure shapes generated are also given. For example, the symbolic execution of the path highlighted in red corresponds to the path followed by Thread b in Figure 4.5 and it leads to the generation of the data structure shape shown at the bottom of the figure. Note that we mainly focus of predicates on lines 14, 16, 17. This is because these predicates mainly influence the choice of shape for the dynamic data structure while the other omitted predicates have their outcomes determined by the chosen shape.

4.3.1 Experimental Setup

To study the effectiveness of our tool in detecting data races, we implemented four important data structures and used them to compare the effectiveness of DSGEN with original GKLEE (the system DSGEN is bulit on). The comparison shows two advantages of our tool: 1) by automatically creating dynamic data structures of different shapes, it enables

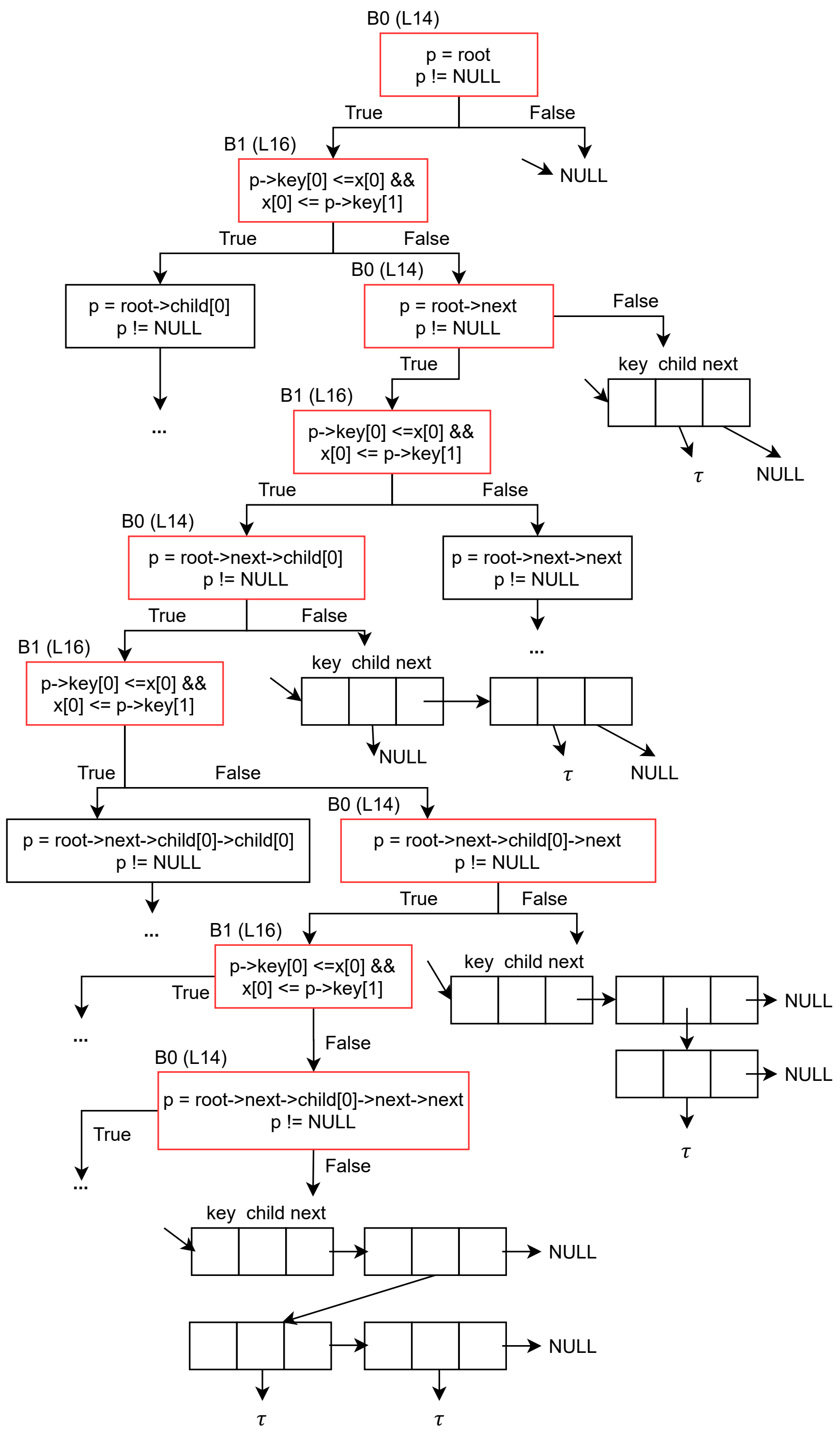


Figure 4.6: The states exploration of dynamic data structure by Thread b of Figure 4.5.

effective concolic testing that explores many program paths; and 2) our tool can uncover hidden data races that cannot be uncovered by GKLEE.

Our evaluation is based upon a diverse set of 25 races shown in Table 4.1. Both read-write (rw) and write-write (ww) races, between threads from the same and different warps, as well as divergent and non-divergent paths are included. To enable execution of GKLEE a simple data structure is manually constructed and provided. While GKLEE's path exploration is based upon this single data structure, DSGEN is able to automatically generate numerous data structure shapes and achieve higher path coverage and superior data race detection. Next, we will discuss the four data structures considered.

Test Concurrent Data Structures We use CUDA implementations of the following four widely used concurrent data structures that are briefly described next:

- B-Tree – Self-Balancing Search Tree B-Tree [8] is widely used in databases. GPU accelerates dynamic queries and batch insertion. In Table 4.1, races 1-10 correspond to B-Tree. The Grid Size and Block Size limits were set to 2 and 16 respectively.

- HAMT – Hash-Array Mapped Trie [11] A HAMT is an array mapped trie where the keys are first hashed to ensure an even distribution of keys and a constant key length. It achieves almost hash table-like speed while using memory much more efficiently. In Table 4.1, races 11-15 correspond to HAMT. The Grid Size and Block Size limits were set to 2 and 8 respectively.

- RRB-Tree – Immutable Radix Balanced Tree [12] The purpose of the RRB Trees is to improve the performance of the standard Immutable Vectors by making the Vector

Concatenation, Insertion as well as Split operation much more performant while not affecting Indexing, Updating and Iteration speeds of the original Immutable Vectors. In Table 4.1, races 16-20 correspond to RRB-Tree. The Grid Size and Block Size limits were set to 2 and 2 respectively.

- Skip List – Probabilistic Ordered Data Structure [73, 87] A skip list is a probabilistic data structure that allows $\mathcal{O}(\log n)$ search and insertion complexity within an ordered sequence of n elements. In Table 4.1, races 21-25 correspond to the Skip List data structure. The Grid Size and Block Size were limited to 2 and 8 respectively.

Since the implementations of above data structures are based upon the correct

Table 4.1: The Data Races of Used in Evaluation.

Line No.: Function Name	RaceId	Data Race Type
`61-98: sort`	1	With Divergence (ww)
`61-61: sort`	2,3	Without Divergence (ww) and Interwarp (rw)
`111-111: split_parent`	4	Interwarp (ww)
`140-235: node_split`	5	With Divergence (ww)
`140-239: node_split`	6	With Divergence (rw)
`271-276: node_insert`	7	With Divergence (rw)
`281-281: node_insert`	8,9	Without Divergence (ww) and Interwarp (rw)
`235-305: search_node`	10	Global Memory (rw)
`53-77: batch_insert`	11	Interwarp (rw)
`84-84: batch_insert`	12	Without Divergence (ww)
`84-108: search_node`	13	Global Memory (rw)
`104-108: search_node`	14	With Divergence (rw)
`77-106: search_node`	15	Global Memory (rw)
`27-27: unref`	16,17	Without Divergence (ww) and Interwarp (rw)
`27-29: unref`	18	Interwarp (ww)
`135-135: modify`	19	Without Divergence (ww)
`135-139: modify`	20	Global Memory (ww)
`32-32: insert`	21	Interwarp (rw)
`39-39: insert`	22	Interwarp (rw)
`78-92: create_node`	23	Global Memory (rw)
`105-105: create_node`	24, 25	Global Memory (rw)

algorithms provided in the noted citations, our implementations did not create any data races. The data races were seeded in these implementations to compare DSGEN with GKLEE.

Metrics for Comparison The comparison will be made in terms of the following:

- The **number of races** found by GKLEE and DSGEN: There are four types of data races that are detected by GKLEE: 1) Intra-warp Races Without Warp Divergence; 2) Intra-warp Races With Warp Divergence; 3) Inter-warp Races; and 4) Global memory races.
- The number of **paths covered**: **path coverage** in these experiments is defined as number of path combinations exercised by the threads for the inputs generated by concolic testing.
- The number **inputs generated** and different **data structures generated** are collected as this compares the power of concolic testing employed by DSGEN vs. GKLEE.
- We also provide the number of **execution steps** and **runtime** for finding the races. An *execution step* is the execution of an LLVM instruction using one of the simulated CUDA threads. Execution time is the running time taken.

Table 4.2: DSGEN vs. GKLEE: Number of Data Races Successfully Detected; and Total Number of Data Structures and Path Combinations Explored during Concolic Testing. For DSGEN the number of data structures that expose data races, the adjustments made, and the corresponding paths explored by them are shown by the first number.

Data Race Type	B-Tree		HAMT		RRB-Trees		Skip List	
	GKLEE	DSGEN	GKLEE	DSGEN	GKLEE	DSGEN	GKLEE	DSGEN
Without Divergence	1	2	0	1	1	2	0	0
With Divergence	0	4	1	1	0	0	0	0
Interwarp	0	3	0	1	1	2	1	2
Global Memory	1	1	1	2	1	1	2	3
Total Data Races Detected	2	10	2	5	3	5	3	5
Data Structure Shapes Generated	1	8+1621	1	4+118	1	4+12	1	4+126
# Adjustments Performed	na	1+35	na	0+7	na	0+0	na	1+4
Path Combinations Covered	126	39+2628	47	13+269	6	4+12	64	7+249

Table 4.3: DSGEN vs. GKLEE: Number of Inputs Generated via Concolic Execution, Execution Steps and Times (seconds) for Exhaustive Exploration.

	B-Tree		HAMT		RRB-Trees		Skip List	
For Detected Races	GKLEE	DSGEN	GKLEE	DSGEN	GKLEE	DSGEN	GKLEE	DSGEN
# Thread Positions	17–33 (64)	17–64 (64)	32–32 (32)	2–32 (32)	8–8 (8)	8–8 (8)	17–32 (32)	17–32 (32)
Exhaustive Exp.	GKLEE	DSGEN	GKLEE	DSGEN	GKLEE	DSGEN	GKLEE	DSGEN
# of Diff. Inputs	61	829	31	72	3	7	64	256
Execution Steps	31,280	67,049	7,582	13,120	516,848	520,668	14,175	52,161
Execution Time(s)	4.905	285.583	5.896	21.042	152.950	173.220	1.55	23.26

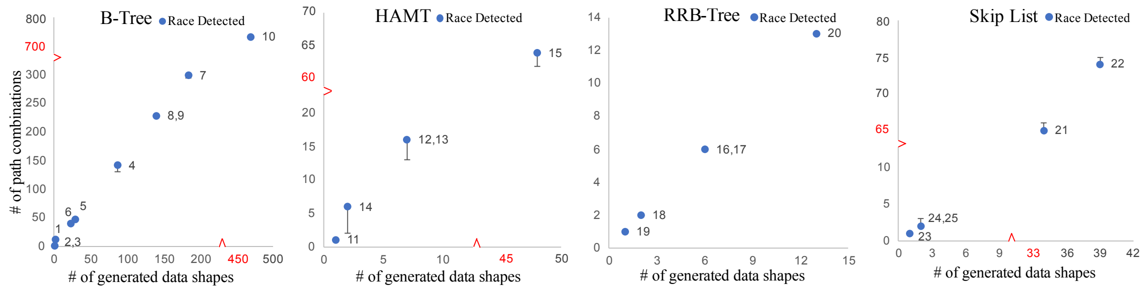


Figure 4.7: DSGEN vs. GKLEE: Data Structures Generated vs. Path Combinations Explored and Data Races Detected.

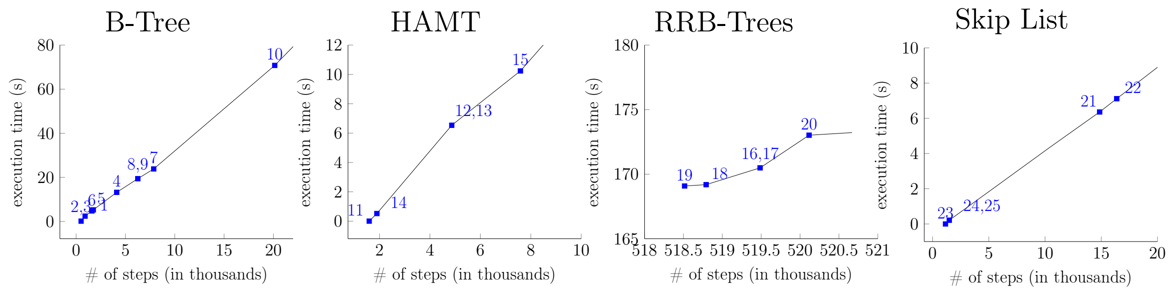


Figure 4.8: DSGEN vs. GKLEE: Cost of Concolic Testing in Terms of Execution Time vs. Number of Execution Steps.

4.3.2 Experimental Results

Effectiveness: Paths Explored and Races Exposed. The effectiveness of race detection is demonstrated by the results presented in Table 4.2 and Figure 4.7. We first note that all 25 races introduced in Table 4.1 were successfully identified by DSGEN, but only 10 were found by GKLEE. In particular, as shown in Table 4.2, GKLEE detected 2 out of 10 races in B-Tree, 2 out of 5 races in HAMT, 3 out of 5 races in RRB-Tree, and 3 out of 5 races in Skip List.

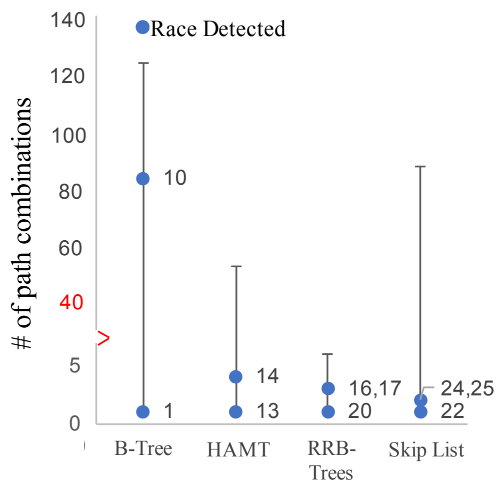


Figure 4.9: Path Combinations Explored by GKLEE using the default data structure shape.

As indicated in Table 4.2, GKLEE could only explore 126, 47, 6, and 64 path combinations using the default data structure shapes provided while DSGEN explored 2667, 282, 16, and 256 path combinations using 1629, 122, 16, and 130 different automatically generated data structures. This shows that manually generating data structure shapes to cover large number of path combinations would require inordinate amount of effort as the programmer would have to manually generate a large number of data structure shapes.

We further note that DSGEN found all the races using a small subset of generated data structures – 8 out of 1629, 4 out of 122, 4 out of 16, and 4 out of 130. We also give #Adjustments which is the total number of adjustments made during integration of per thread data structures. The data shows that integrated data structure cannot always be obtained by the union of per thread data structures. These data structures explored 39, 13, 4, and 7 path combinations in all and can be reported to the user along with the concrete inputs that expose the data race. Figure 4.7 further shows the subset of paths covered by 8, 4, 4, and 4 of the generated data structure shapes that were responsible for uncovering all the data races. The specific data race ids and corresponding path combinations are also marked on the graph. Figure 4.9 gives the corresponding plot for GKLEE.

The above data clearly shows that to detect races, many path combinations need to be explored, and this is only possible by generating different data structures of different shapes. In absence of automatic data structure generation ability, GKLEE requires that the user manually construct different data structure shapes and provide them to GKLEE. However, constructing data structures that can expose data races is difficult for the user, especially without knowing where the race may happen. For example, the race that goes undetected by GKLEE for RRB-Tree involved the reference counting during object destruction. In addition, other races are harder to expose as they involve rare situations requiring data structures of a particular shape and size. For example, when we analyzed the behavior of GKLEE for BTree further, we found that the race conditions are usually hidden by branch conditions that requires specific node size. Furthermore, some races require conflicting race conditions that a single data structure cannot satisfy. For example, Skip List requires a child node size less than 31 to expose race 21 and exactly equal to 31 to expose race 22.

Inputs Generated and Runtime Costs Finally, in Table 4.3 and Figure 4.8 we show the runtime cost of DSGEN and GKLEE. The table shows that concolic testing based upon DSGEN generated far more inputs than GKLEE: 829 vs. 61, 72 vs. 31, 7 vs. 3, and 256 vs. 64. This shows the power of DSGEN as only by generating different data structure shapes can path combinations be explored and thus many different inputs generated.

For detected races, we also report #Thread Positions which is the number of different thread positions within thread blocks that are covered by the concrete inputs that expose data races. The range represents the minimum and maximum thread positions covered across all data races while the number in parenthesis is the maximum number of thread positions available. This shows that although parametric flows represent multiple threads, when data races are successfully exposed, concrete threads in thread blocks that are identified can occupy different positions.

The execution steps and execution time in seconds are also given. As we can see, the runtime cost of using DSGEN was acceptable for cases considered, typically just a few minutes. The plots in Fig. 4.8 show how the execution steps correspond to execution time in seconds for DSGEN. There is no obvious bottleneck data structure observed during the execution in our benchmarks. Since library functions being tested have limited execution space, especially considering the use of parametric flows, we were able to exhaustively test these functions.

4.4 Race Detection via SSRD for Multithreaded Programs

We also developed SSRD to evaluate our methods on multithreaded programs. This prototype is built on top of Cloud9, which is a famous symbolic executor that supports multithreaded programs using `pthread` APIs.

4.4.1 Experimental Setup

To study the effectiveness and efficiency of SSRD, we compared with two representative race detectors, AFL++ (fuzzing+TSAN) and a lock-set based data race detector using Cloud9(concolic execution+lockset). In the experiments, we tested and compared our system with them on lock-based implementations of the following five concurrent dynamic data structures: *Skip List*(SL), *Unrolled Linked List*(ULL), *Priority Queue*(PQ), and *AVL-Tree*(AVL). These codes are modified from open source projects and augmented with statements to trigger read-write and write-write data races. When running AFL++ and Cloud9 data race detector, we provide additional code to create an initial shape for each kind of the data structure and further testing of insertion, deletion, and search is based on the initial shape. For SSRD, we will use a symbolic pointer as an automatic generated data shape to test the data structure, all actions are based on the symbolic pointer. To limit the size of generated shapes, we limit the length of pointer chain allowed.

Our system provides API for exploration of data shapes and thread interleavings. First, we add a new API in cloud9, named `klee_make_data_structure`, to indicate a pointer is pointing to an extensible data structure. Other symbolic values except pointers can be marked using the normal clould9 API `klee_make_symbolic`. For AFL++ fuzzer, we coverted the clould9 API `klee_make_symbolic` into an custom input function which can read data from standard input stream and we provide a file with a few available data as the initial seed. For exploitation of thread interleaving, the posix thread model is already built into Cloud9. User creates a pair of threads using pthread API to execute functions concurrently for testing. The detected concurrency bugs and corresponding input values are recorded in log files.

Table 4.4: Data Race Detection Effectiveness: Comparing Fuzzing(AFL), Concolic Execution without (Cloud9) and with (SSRD) Shape Generation .

Test Case	**Function Name**	**# of Paths Covered**			**# of Data Races Detected**		
		AFL	**Cloud9**	**SSRD**	**AFL**	**Cloud9**	**SSRD**
SL	insert	3	52	3779	2	2	8
	search	6	13	1479	0	0	2
ULL	insert	3	10	321	7	10	15
	delete	16	13	78	2	1	4
	search	16	16	93	0	0	3
PQ	insert	2	22	31	0	5	5
	remove	2	16	1479	0	11	19
AVL	insert	1	7942	16329	0	4	6
	delete	1	285	1322	0	1	12

4.4.2 Experimental Results

We evaluated the effectiveness of our system comparing it with traditional Cloud9 system with lockset based race detecting algorithm. Table 4.4 shows the path covered and the number of data race detected using each approach. The AFL++ shows it can not effectiveless find the data races. Since from all known races, it detects only 2 out of 10 in skiplist, 9 out of 20 races in linked list, 0 out of 24 in priority queue and 0 out of 18 in AVL-tree. We then note that Cloud9 can only detect a subset of data races that are detected by SSRD - 2 out of 10 in skiplist, 9 out of 20 in linked-list, 16 out of 24 in priority queue, and 5 out of 18 in AVL-tree. At the same time, we observed that there is no new data race that is detected by Cloud9 but not found by SSRD. There is only one race in linked list that are finding by fuzzing which not find in SSRD which due to constraints solving problem. We observe that the number of paths explored by SSRD 1.4×(31 vs 22) to 113.7×(1379 vs 13) greater than Cloud9. This greater exploration of search space by SSRD is responsible for uncovering many data races that are missed by Cloud9.

Chapter 5

Improving Efficiency Via Summarization and Guided Search

In this chapter, we provide an optimized approach of concolic testing that is effective and efficient in finding data races in multithreaded C/C++ programs with concurrent dynamic data structures. *Summarization* is the main method to improve efficiency by using *concolic unit testing* to precompute the shared constraints, data shapes and memory access information. It also provides *potential data race* information for *guided search* which can change the exploration order to quickly find the race. The shape exploration and memory accesses can be reused during invoking summaries in concolic testing of the full program.

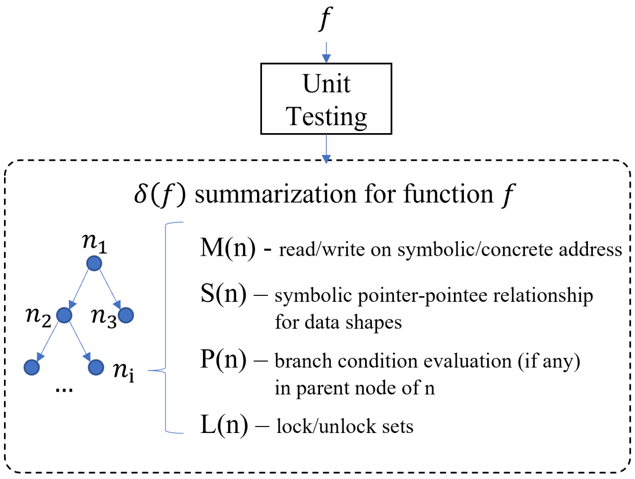
Thus, our testing process consists of two steps, a *concolic unit testing step* followed by the *full program concolic testing step.* In the first step using unit concolic testing summarization of individual functions that implement concurrent dynamic data structures is carried out. At the same time, the memory accesses in summaries are used to identify pairs of paths that contain *potential data races*. In the second program concolic testing step, we start from the main function and test the whole program with the aim of generating inputs that confirm potential data races one by one. During this process, the data structure shapes contained in function summaries are *reused* to direct the exploration of non-conflicting data structure shapes for various paths in multiple threads. This directed search prunes the exploration of paths that cannot realize the potential data race. The reuse of data structure shapes and other information in summaries improves the efficiency with which summarized functions are executed during testing.

5.1 Shape-Aware Summarization

5.1.1 Overview

Summarization via Concolic Unit Testing Given a program P with a set of functions Σ, let F be the subset of functions from Σ that corresponds to concurrent dynamic data structure implementations and are candidates for summarization. For a given function $f \in F$, *concolic unit testing* of f is performed to build the decision tree $\delta(f)$. The constructed decision tree corresponds to the *tested paths* in the function such that each leaf node corresponds to a tested path from the start of the function to a return point. Δ denotes the set of *summaries* of functions in F. The overview of the concolic unit testing of function f and the generated decision tree $\Delta(f)$ is shown in Figure 5.1.

In a *decision tree* $\delta(f)$, each node $n \in \delta(f)$ represents a conditional, a call to a function, a synchronization operation, or a return from a function. All nodes are annotated



f
Unit Testing
$\delta(f)$ summarization for function f
n_1
n_2
n_3
n_i
...
M(n) - read/write on symbolic/concrete address
S(n) – symbolic pointer-pointee relationship for data shapes
P(n) – branch condition evaluation (if any) in parent node of n
L(n) – lock/unlock sets

Figure 5.1: Concolic Testing for Summarization of function f.

with shape $S(n)$ and memory access $M(n)$ summaries defined as follows:

- $S(n)$ – the set of *pointer-pointee relationships* among symbolic pointers representing the *data structure shape* that must be satisfied to enable the execution of $n \in \delta(f)$;
- $M(n)$ – the set of symbolic names (globals and parameters) and concrete addresses (locals) that correspond to the *read/write* memory accesses performed by n;

Also, branch nodes, call nodes, synchronization nodes, and return nodes are annotated with additional information $B(n)$, $C_f(n)$, $L(n)$, $V(n)$ respectively as described below.

- $B(n)$ – is the *branch condition* if n is a branch node;
- $C_{f'}(n)$ – contains name of function f' and the parameters for the call if n is a call node;
- $L(n)$ – contains *lock/unlock* action associated with n; and
- $V(n)$ – is the return value if n is a return node (it is empty if there is no return value).

Here, we briefly describe some key points about the unit testing that computes $\delta(f)$. First, all global variables as well as the parameters of f are treated as symbolic variables. Second, if f contains a function call, the return value of the callee function is treated as a symbolic variable and the testing of paths following the call are explored using the symbolic return value. In addition, since the callee may not be pure function (i.e., it can have side effects), the local variables that are passed as parameters to the callee are also treated as symbolic starting from the call site. In addition, the loops are handled by limiting the number iterations and then enumerate all of paths they can generate.

Concolic Testing of Full Program After concolic unit testing, the function summaries are constructed. We can use those summaries in the concolic testing of full program. Given a list of user specified symbolic input, the target program will execute with both symbolic and concolic input, and different paths will be explored. If the program contains multiple threads, other threads will be executed once the program meets locks or wait signals. During the thread scheduling, different thread interleavings will be explored.

Our approach improves the efficiency of full program concolic testing by taking advantage of summaries in Δ. When a thread encounters a call to a function f for which summary $\delta(f)$ is available in Δ, *summary reuse* is invoked instead of calling f. This approach eliminates overhead of constructing symbolic expressions, gathering and checking constraints, and building data structure shapes that satisfy constraints. That is, some of work performed during unit testing of a function is reused instead of being repeated during each execution of the function during concolic testing.

Given the current state of program ϕ just before node n in a summarized function, ϕ is updated by affecting it using the summary associated with n as follows:

1. *Shape Formation*: Given state ϕ, the shape summary $S(n)$ transforms the shape of the data structure giving state ϕ'.

$$\phi \overset{S}{\underset{n}{\Longmapsto}} \phi'$$

2. *Memory Accesses*: A memory access summary includes reads from locations and writes to locations that copy symbolic or concrete values and changing state to ϕ''.

$$\phi' \overset{M}{\underset{n}{\Longmapsto}} \phi''$$

3. *Updates based upon the type of node n*:

   - *Branch*- The expression $eval(B(n) = true, \phi)$ evaluates branch condition $B(n)$ and checks if it is true on ϕ. By evaluating $eval(B(n) = true, \phi'')$ and $eval(B(n) = false, \phi'')$, and adding appropriate path constraints, new states are represented as:

$$\phi'' \overset{B=true}{\underset{n}{\Longmapsto}} \phi''_t \quad or/and \quad \phi'' \overset{B=false}{\underset{n}{\Longmapsto}} \phi''_f$$

   - *Call to f'*- Update state by invoking callee f', using callee's summary if available, $\phi'' \overset{C_{f'}}{\underset{n}{\Longmapsto}} \phi'''$;

   - *Return node*- Update state by mapping the return value to the caller $\phi'' \overset{V}{\underset{n}{\Longmapsto}} \phi'''$; or

   - *Synchronization*- Applying $L(n)$ to state ϕ'' leads to state ϕ''', $\phi'' \overset{L}{\underset{n}{\Longmapsto}} \phi'''$, where executing threads state changes based upon the synchronization operation.

Finally, updating state ϕ due to a sequence of statements $n_i - n_j$ along a path is performed as follows.

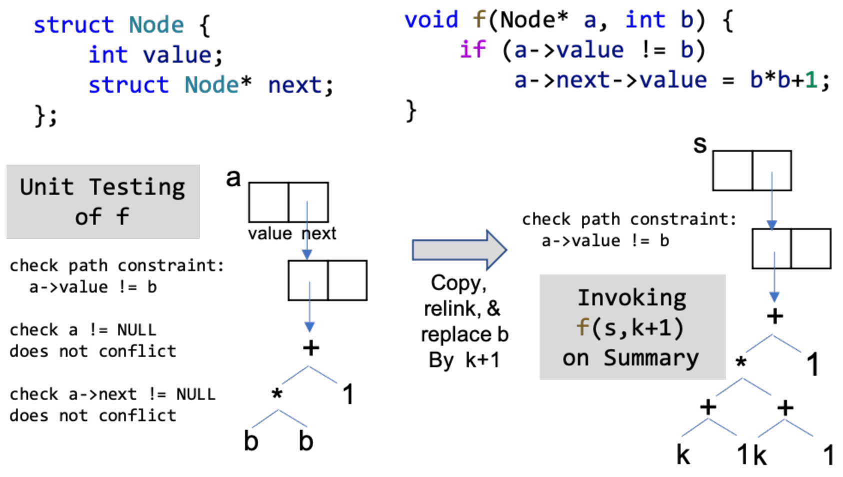
$$\phi \overset{\delta_{i-j}}{\underset{n_i - n_j}{\Longmapsto}} \phi' \quad = \quad \phi \overset{\delta_i}{\underset{n_i}{\Longmapsto}} \phi_i \overset{\delta_{i+1}}{\underset{n_{i+1}}{\Longmapsto}} \phi_{i+1} \cdots \phi_{j-1} \overset{\delta_j}{\underset{n_j}{\Longmapsto}} \phi'$$

Updating state via use of summary is more efficient than the normal function call due to the following reasons:

– *Lightweight Construction* The checking of constraints, creation of data shape, and construction of symbolic expressions that is carried out during concolic *unit testing* of a function is reused during concolic testing of the *full* program. In Figure 5.2, when `f`'s summary is reused, other than replacing formals by actuals and checking path condition, the rest of the work is not repeated.

– *Minimizing Memory Accesses* Once symbolic expressions are simplified, some memory accesses are eliminated – if multiple writes are directed to same address, only the last write is needed. The computation of local variables may also be eliminated. In Figure 5.2, for `g()`, the local variable `b` is eliminated, only last writes to `a[0]` and `a[1]` are performed, and `a[0]` is read only once.

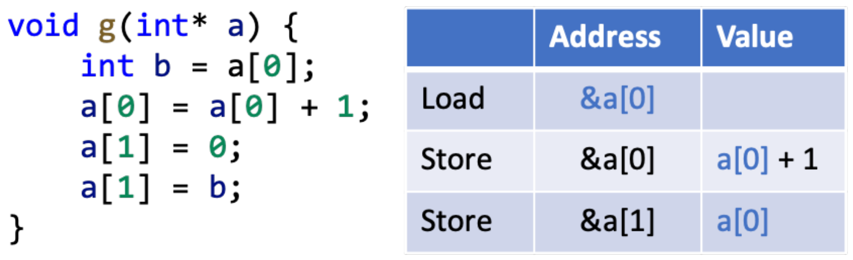
Illustration – Concurrent Skip List. Consider the code in Listing 3.1 which presents two operations for a skip list - `insert` and `search` for inserting in a ordered list and searching for a node corresponding to a key value. The function `search_node` is a common function used by both `insert` and `search` functions to find the node which contains key k. Let us assume that the main function creates two POSIX threads and calls `thread0_main` and `thread1_main` that are the entry functions of the two POISX threads in this example.



```
struct Node {
    int value;
    struct Node* next;
};
```

```
void f(Node* a, int b) {
    if (a->value != b)
        a->next->value = b*b+1;
}
```

Figure 5.2: Benefits of Summaries: Construction Overhead.



```
void g(int* a) {
    int b = a[0];
    a[0] = a[0] + 1;
    a[1] = 0;
    a[1] = b;
}
```

	Address	Value
Load	&a[0]	
Store	&a[0]	a[0] + 1
Store	&a[1]	a[0]

Figure 5.3: Benefits of Summaries: Memory Accesses.

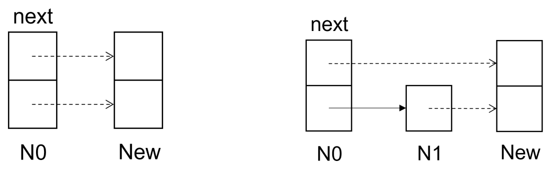


Figure 5.4: A data shape that cannot expose (left) and can expose (right) the race.

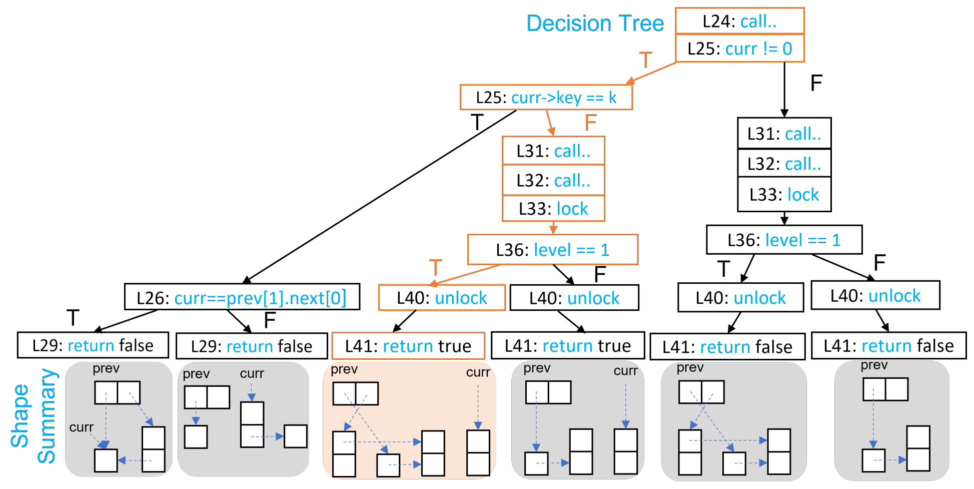


Figure 5.5: The decision tree for `insert` function.

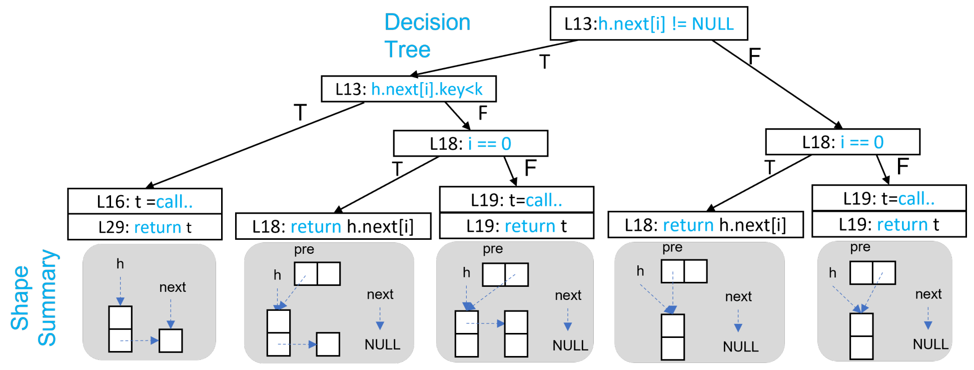


Figure 5.6: The decision tree for `search_node` function.

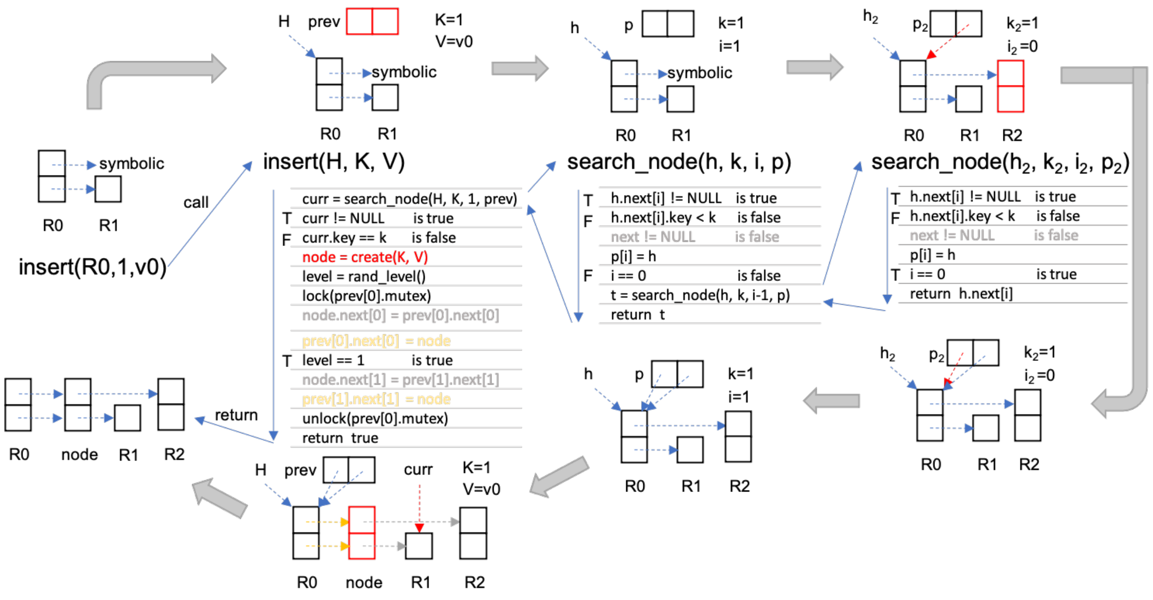
– *Example data race.* To allow an illustration of our method, our implementation includes the following error. During the insertion of a new node in the skip list, `insert` function finds the suitable position for insertion and collects all nodes that need to be modified in the list `prev` (line 24, 34-39). Instead of locking every node in `prev` list, by mistake we have locked only one node (line 33, 40). This leads to many data races in `insert` and `search_node` functions. However, for illustration purposes we consider one race. In function `insert`, since the mutex lock only protects node `prev[0]`, line 38 and line 14 have a data race when `prev[1]->next[1]` is being written and `h->next[1]` is being read at the same time since `prev[1]` and `h` represent the same node.

– *Shape required.* Manifestation of this data race requires a skip list with a certain shape. For the skip list in Figure 5.4 left, `prev[0]` and `prev[1]` refer to the same node. Therefore the lock at line 33 can protect node `prev[1]` which is being modified and hence the data race between line 38 and line 14 does not manifest itself. On the other hand, in Figure 5.4 right, the skip list shape shown exposes the data race. From this example we conclude that it is necessary to explore appropriate data shapes to expose a desired data race. Data shapes also influence the path taken and hence to exercise a given path, we must use an appropriate shape.

– *Summary Representation.* Since functions `insert`, `search`, and `search_node` implement the skip list concurrent data structure, their summaries will be generated. Figures 5.5 and 5.6 show the generated decision trees of functions `insert` and `search_node`. Note that trees include call nodes, branch conditions, synchronization operations, and return nodes. For the highlighted path `T F T` in this decision tree, the data shape generated

line#	address	value	type
25	curr->key		Load
34	prev[0]->next[0]		Load
34	node->next[0]	prev[0]->next[0]	Store
35	prev[0]->next[0]	node	Store
37	prev[1]->next[1]		Load
37	node->next[1]	prev[1]->next[1]	Store
38	prev[1]->next[1]	node	Store

Table 5.1: Memory accesses along path `T F T` in `insert`.



```
  curr = search_node(H, K, 1, prev)
T curr != NULL        is true
F curr.key == k       is false
  node = create(K, V)
  level = rand_level()
  lock(prev[0].mutex)
  node.next[0] = prev[0].next[0]
  prev[0].next[0]  = node
T level == 1          is true
  node.next[1] = prev[1].next[1]
  prev[1].next[1]  = node
  unlock(prev[0].mutex)
  return  true
```

```
T h.next[i]  != NULL   is true
F h.next[i].key < k    is false
  next != NULL         is false
  p[i] = h
F i == 0               is false
  t = search_node(h, k, i-1, p)
  return  t
```

```
T h.next[i]  != NULL   is true
F h.next[i].key < k    is false
  next != NULL         is false
  p[i] = h
T i == 0               is true
  return  h.next[i]
```

Figure 5.7: Data Structure Shapes at entry and return points of function invocations. *Gray/yellow statements are excluded from decision tree as their effect is captured by shape/memory access summaries or they are fully evaluated during unit testing.*

is captured via pointer-pointee relations in Table 3.1 and the memory accesses summary is given in Table 5.1. In all cases pointer dereferencing implies a non null pointer and thus pointers point to other nodes in Table 3.1.

– *Summary invocation.* We illustrate the use of summaries in Figure 5.7 where a thread creates an initial data structure and invokes function `insert` which in turn invokes summarized recursive function `search_node`. All three function invocations use corresponding function summaries. The data structures at function call and return boundaries is shown. The changes involve mapping of symbolic names and also making changes to data structure to reflect the effect of the function via use of summaries of data structure shapes and write memory operations. The statements along the path followed are also shown with statements that are not present in the decision tree are shown in gray and yellow such that the effect of these statements on program state is achieved via use of shape and memory access summaries.

The `insert` function is invoked with arguments `R0, 1, v0` where `R0` and `v0` are set to symbolic by the user. `R0` points to a symbolic object `R0.next[1]` and a concrete object `R0.next[0]` (also `R1`). The invocations of `insert` and `search_node` start with evaluation of their decision trees and lead to actions that affect state as if functions are executed. Upon invocation of each function, the arguments are mapped to symbolic names used during unit testing. For `insert`, the symbolic names `H, K, V` used in unit testing of `insert` are mapped to real arguments `R0, 1, v0`. The evaluation of first decision tree node invokes `search_node`. The arguments `H, K, 1` and local `prev` are passed to `search_node`. Before invocation, local memory object `prev` is allocated and used as the output buffer for search_node. Also symbolic unit testing names h, k, i, p are mapped to reals, concrete or symbolic, in the caller (i.e., H = R0, K = 1, 1, and prev). The recursive call search_node(h, k, i-1, p) maps h_2, k_2, i_2, p_2 to R0, 1, 0, prev.

In the first invocation of search_node, the evaluation of branch conditions in the decision tree uses symbolic arguments (e.g., h.next[i] != NULL becomes R0.next[1] != NULL) that can be true or false, but we choose to explore the true branch first. After evaluating branch conditions, the appropriate data shape is processed to affect the current data structure. Memory object R2 is created to satisfy pointer-pointee relationship in the path. Finally, the memory operations are processed: the write p[i] = h is converted to prev[1] = R0 and prev[0] = R0 in the first and second invocations.

In insert, the branch nodes are evaluated using the return value curr from search_node. Since in the second invocation of search_node returns h.next[i], which is R1, the return value curr refers to R1. The branch conditions become R1 != NULL and R1.key == k. After calling of create and rand_level, the local variable node and level become concrete values. During the evaluation of the subsequent decision tree nodes in Figure 5.5, lock/unlock events are processed, and memory accesses that write to nodes prev[0] and prev[1] (which both refer to R0) are processed. After invocation of insert is complete, all the local variables and names disappear upon the pop action of the current stack frame.

Note that all of the above actions were performed using function summaries which optimizes the work performed.

5.1.2 Summary Invocation Algorithm

Next, we present the key details of our algorithms for summary invocation during full program concolic testing. We assume the summary of each function interested in is available.

During the concolic testing of the full program, a call to a summarized function f is replaced by invocation of its summary. The invocation algorithm maps the symbolic and concrete values, including pointers, obtained via *concolic unit testing* to the values in the current state. The decision tree $\delta(f)$ of function f is used to determine which path is followed and the program state impacted by execution of the path is updated by storing symbolic addresses in memory. The invocation algorithm only handles the summarized functions. Unsummarized functions, or paths whose summaries are unavailable, are executed as they are by standard concolic testing.

The invocation process, presented in Algorithm 2, begins with a list of input parameters (symbolic or concrete values), an execution state ϕ, and the root node of the decision tree of $\delta(f)$. The `InvokeSummary` function presents the actions for different node types. For all node types, at line 2, first $\phi \xmapsto[n]{S} \phi_s$ applies data shapes to the current state ϕ. For a branch node, we evaluate the branch condition $eval(B(n) = true, \phi)$ (line 4-6) and $eval(B(n) = false, \phi)$ (line 7-9) to decide whether branch condition is true, false, or either. Then, we apply the branch condition to the path constraints in the new state(s). We continue to process the child branches based on the evaluation results using new state(s) (at line 6 and 9). For call, synchronization, and return node types we process memory accesses $\phi_s \xmapsto[n_p - n]{M}$ from the last non-branch node (line 11-12) since their side effects must be reflected in the new state ϕ_m. The new state will serve as the start state for a new call, return, or a thread context switch. The state ϕ' represents the state after applying $C'_f(n)$, $L(n)$, or $V(n)$ based on the node type (line 12). Eventually, the final state ϕ' will be pushed into state queue Φ. To implement $\phi \overset{\delta(f)}{\underset{n}{\Longmapsto}} \phi_s$, the key actions when invoking a summary are defined as follows:

(1) *Create object mapping.* Each memory object in the concolic *unit* test summary is mapped to real memory corresponding to the executing program. This mapping is used to convert the memory objects from concolic unit testing to the real memory objects in the current execution state.

(2) *Create value mapping.* The value mapping is used to convert symbolic values used in concolic unit testing to the values in the current execution state.

(3) *Converting objects.* As the decision tree path taken is identified, the memory accesses to the objects in unit testing are converted to real objects by looking up the *object mapping* and calculating the offset if the pointer is not pointing to the beginning of the objects. Observe that multiple pointers in unit testing can map to the same object. Symbolic values in memory are mapped into symbolic or concrete values in the current state using *value mapping.*

For describing details of the above step, some utility functions need to be defined first: `getBase` computes the base address of memory object; `getOffset` computes the offset of a field relative to the base address; `isPointer` determines if the current expression represents a pointer or not; and finally `read(`p`)` can dereference a pointer p and read its content.

Algorithm 2: Summary Invocation – Steps 1, 2 & 3.

```
1  Procedure InvokeSummary($\phi$, $n$):
2      $\phi \overset{S}{\underset{n}{\Longmapsto}} \phi_s$
3      if $n$ is branch then
4          if eval $B(n)$ can be true in $\phi$ then
5              $\phi_s \overset{B=true}{\underset{n}{\Longmapsto}} \phi_t$
6              InvokeSummary($\phi_t$, $n$.true_branch)
7          if eval $B(n)$ can be false in $\phi$ then
8              $\phi_s \overset{B=false}{\underset{n}{\Longmapsto}} \phi_f$
9              InvokeSummary($\phi_f$, $n$.false_branch)
10     else
11         $n_p \leftarrow$ the last non-branch node
12         $\phi_s \overset{M}{\underset{n_p-n}{\Longmapsto}} \phi_m \overset{C/L/V}{\underset{n}{\Longmapsto}} \phi'$ based on the type of $n$
13         if $n$ is return then
14             $\Phi$.push($\phi'$)
15         else
16             InvokeSummary($\phi'$, $n$.child_node)
17 Procedure mapObjectInit(args):
18     obj_map, value_map $\leftarrow$ {}, {}
19     foreach unit_arg, real_arg in args do
20         if isPointer(real_arg) then
21             $b \leftarrow$ getBase(unit_arg)
22             obj_map[$b$].base $\leftarrow$ getBase(real_arg)
23             obj_map[$b$].offset $\leftarrow$ getOffset(real_arg)
24         else
25             value_map[unit_arg] $\leftarrow$ real_arg
26     return obj_map, value_map
27 Procedure mapObject(obj_map, value_mapper, S):
28     foreach pointer, pointee in $S$ do
29         real,_ = convert(obj_map, value_mapper, pointer, _)
30         data = read(real)
31         $b$ = getBase(pointee)
32         $o$ = getOffset(pointee)
33         obj_map[$b$].base $\leftarrow$ getBase(data)
34         obj_map[$b$].offset $\leftarrow$ getOffset(data) - $o$
35         value_map[pointee] $\leftarrow$ real
36 Procedure convert(obj_map, value_map, $a$, $v$):
37     $b$ = getBase($a$)
38     $o$ = getOffset($a$)
39     $b'$ = obj_map[$b$].base
40     $o'$ = obj_map[$b$].offset + exprReplace(value_map, $o$)
41     return ($b' + o'$), exprReplace(value_map, $v$)
```

In Algorithm 2, $\phi \overset{S}{\underset{n}{\Longmapsto}} \phi_s$ can be implemented using `mapObject` and `mapObjectInit`. First, we initialize the object mapping and value mapping by calling `mapObjectInit` with the input parameter pairs, which are the parameters in current state and parameters from the unit testing. Each pair of parameters is processed and added to the object mapping and value mapping as initial information for which objects from unit testing and current state map to each other. Each time during processing $\phi \overset{S}{\underset{n}{\Longmapsto}} \phi_s$, `mapObject` is called by passing the same object mapping, value mapping containers and the shape summary $S(n)$ which is a list of PPRs. First, the pointer is converted into the real memory address *real* and loaded with the value *data* that *real* points to. If *data* is concrete value, which points to a memory object, this object should be mapped to the object that pointee referenced to in the unit testing (line 31-34). For handling address being writen and stored in the memory access, the value mapping is updated (line 35).

The `convert` function converts a memory access with address a and value expression v (for write access) to a real memory object by looking up the object mapping and value mapping. The memory access $\phi \overset{M}{\underset{n}{\Longmapsto}} \phi_m$ is implemented by converting the memory reads and writes in M from unit testing into real memory object reads and writes. Only the first read and last write for each memory address access is needed.

5.2 Summary-Guided Race Detection

Next, we provide a summary guided race detection algorithm based upon the lockset algorithm [92], hybrid race detection [79], and the Cloud9 thread scheduling algorithm [29]. Our race detection method has two phases. In the first phase, before program testing, we iterate over all the possible pairs of paths in a given summary to find potential data races in the function. We use the lockset algorithm [92] to expose potential data races from summaries for each pair of paths. When traversing a pair of different memory accesses along two paths, if the accesses refer to the same address and do not hold the same lock, then we record a potential data race involving the accesses. During program testing, in the second phase, we use the thread scheduler to postpone the thread when it reaches one of potential racing statements. This thread's reactivation is prevented until another thread reaches the corresponding racing statement confirming a race.

If two paths dereference pointers, the data race will only occur when both accesses refer to the same memory object. Therefore, the integration of data structures for the two paths into a non-conflicting consistent shape is required for potential data race detection. Our integration algorithm accepts a pair of paths as the input and generates an integrated data shape for the paths as the output. The integration is achieved by modifying the pointer-pointee relationships for the two paths. If we have two different paths of the same function, we can first start from the root pointer of all the parameters and local/global variables which have been marked as symbolic variables, and combining the pointer-pointee relationship for two paths. We illustrate this process using an example.

In the second phase, while testing the application, the scheduler explores interleaving of threads to confirm a potential data race. An operation called *postpone* is used to exercise thread interleavings. Once the current thread satisfies the conditions for a potential data race, the scheduler postpones the thread to give another thread a chance to be scheduled and progress to a point that realizes the data race. If a schedule is found that realizes the data race, the race is reported and thread is no longer postponed and allowed to be scheduled to search for another data race.

Algorithm 3: An algorithm for data race detection

```
Input: The initial running state $\phi_0$ and a set of potential race set $R$
Output: An optimized scheduling of summary invocations.
1  Procedure SchedulingGen($\phi_0$, $R$):
2      $Q = \phi_0$
3      $\phi = Q$.pop()
4      while $Active(\phi) \neq \emptyset$ do
5          for $t \in Active(\phi)$ do
6              while $t \notin postponed(\phi)$ do
7                  $t$ = nextThread($t, \phi$)
8                  if all threads are in $postponed(\phi)$ then
9                      $postponed(\phi) = postponed(\phi)$ - $t$
10             $\phi'$ = forkThread($\phi$)
11             $Current(\phi') = t$
12             runThreadUntilSync($\phi'$)
13             $Q$.push($\phi'$)
```

Algorithm 3 shows how the thread scheduler explores interleavings to confirm potential data races. It continues exploring all paths and thread interleavings combinations till testing time is exhausted. $Active(\phi)$ maintains a set of running threads. `postponed` is a set of threads which are currently postponed and cannot be scheduled yet. The algorithm checks when there are threads are running ($Active(\phi) \neq \emptyset$). For each running thread t, if t is in $Postponed(\phi)$, we postpone the thread to schedule the next one using `nextThread(t)` to get the next thread. If all threads are postponed, we remove the current thread from the $Postponed(\phi)$ set to make sure there is no dead lock. Then, we fork a new state to explore the thread scheduling for t and execute the thread t using `runThreadUntilSync` until it meets a thread synchronization event (for example lock, unlock, etc). Finally, we push state ϕ' in the state queue Q to schedule the next synchronization event.

5.3 Summarization and Guided Search in SSRD

The SSRD system is implemented under the Cloud9 framework [29] which provided a scalable symbolic execution engine and a POSIX thread model which can be used to detect bugs in multithreaded programs. For concolic testing of the full program that makes use of functions of the concurrent dynamic data structure, we first identify potential data races that may arise when multiple threads execute summarized functions that implement the concurrent dynamic data structure. Given a function $f \in F$, and its summary $\delta(f) \in \Delta$, a set of *potential data races* R is computed. Each data race in R is of the form $r(\rho_i, \rho_j)$ where ρ_i and ρ_j are paths whose simultaneous execution by different threads may cause a data race according to $\delta(f)$. Symbolic variable set I represents all *user defined symbolic variables* for program P. The concolic executor explores paths for confirming data races in R.

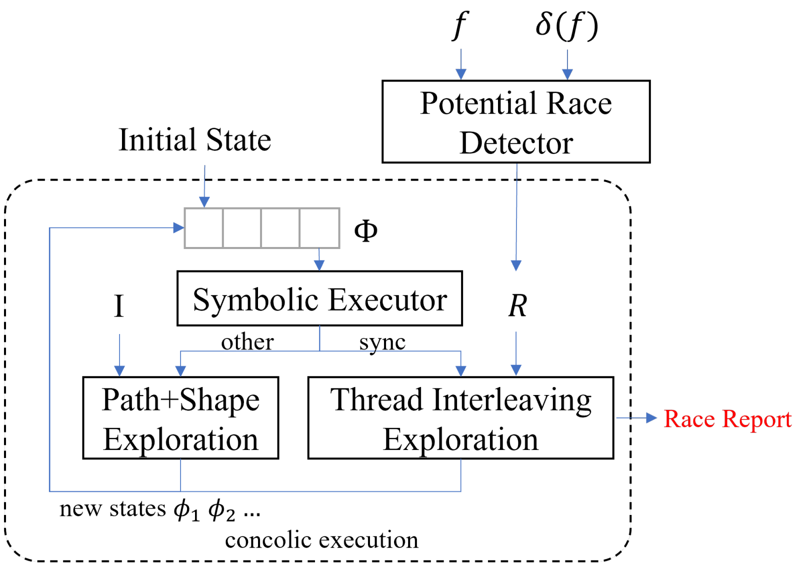


Figure 5.8: Exploring Path, Shape and Thread Interleaving.

Figure 5.8 shows the concolic testing performed to identify realizable data races in R via *search guided* by $\delta(f)$. The set of states maintained by the testing engine is shown as Φ. State set Φ contains all possible execution states of the program P. A state $\phi \in \Phi$ contains the current status of all threads and the complete address space for all memory objects. ϕ_0 is the initial state of the program and $T(\phi)$ denotes all threads in a state ϕ such that $Active(\phi)$ is the subset of threads that are ready to run and $Running(\phi)$ is the thread in $Active(\phi)$ that is currently running under a scheduling policy. Finally, $Postponed(\phi)$ is the subset of threads that have executed a statement involved in a potential data race and are waiting for another thread to exercise the corresponding statement to confirm the data race. This aspect of scheduling direct execution towards exposing a data race.

Let us briefly consider how the search is carried out. Starting from the initial state ϕ_0 for the program, such that the main function serves as the entry point, the concolic executor explores paths and shapes, and thread interleavings when handling branch instructions and synchronization actions. For efficient path exploration using given input values for I, different branch outcomes are forced and at the same time the corresponding states are pushed into the state queue Φ. The thread interleaving exploration is guided by R as follows. A thread is made to execute a path ρ_i involved in a potential data race and another thread is made to explore all paths $\rho_j \in R$ such that $r(\rho_i, \rho_j)$ belongs to R. The data races confirmed during exploration are reported.

Table 5.2: SSRD Execution Time: Without Using (WO-Sum) and With Using (W-Sum) Function Summaries + Summaries Construction Time. Negative percentages indicate reductions, and reduction in parentheses means the reduction if we don't calculate summaries construction time.

Test Program	Function Name	Execution Time (s)		
		WO-Sum	W-Sum (+cons. time)	Reduction (% w/o cons. time)
SL	insert	326.56	172.3 + 0.81	-47.0%(-47.2%)
	search	116.19	98.29 + 0.75	-14.8%(-15.4%)
ULL	insert	1092.27	87.84 + 0.39	-91.9%(-92.0%)
	delete	76.26	21.47 + 0.44	-71.3%(-71.8%)
	search	268.32	66.31 + 0.53	-75.1%(-75.3%)
PQ	insert	15.45	23.77 + 0.88	59.5%(53.9%)
	remove	111.26	8.58 + 31.39	-64.1%(-92.3%)
AVL	insert	41.47	18.42 + 0.76	-53.7%(-55.6%)
	delete	212.06	122.61 + 0.98	-41.7%(-42.2%)

Table 5.3: Analysis of Execution Time: Reduction of Constraint Solving Time Using (W-Sum) Function Summaries. # of Solved represents the number of constraints solved during execution.

Test Program	Function Name	Constraints Solving Time		# of Solved	
		% of Total Time	W-Sum (% Reduction)	WO-Sum	W-Sum (% Reduction)
SL	insert	45.27%	-72.69%	100466	42336(-57.9%)
	search	22.54%	-35.15%	45948	28728(-37.5%)
ULL	insert	85.20%	-96.95%	485212	46655(-90.4%)
	delete	64.55%	-94.38%	75548	3333(-95.6%)
	search	73.56%	-92.32%	164220	7140(-95.7%)
PQ	insert	95.64%	55.72%	5704	6764(18.6%)
	remove	87.20%	-92.27%	9486	527(-94.4%)
AVL	insert	48.66%	-97.92%	19625	333(-98.3%)
	delete	53.39%	-96.40%	49887	3856(-92.3%)

Table 5.4: Analysis of Summarization Benefits. # of Memory Access shows the number of memory read/write during execution and Reduction shows the number of memory read/write reductions once using summarization.

Test Program	Function Name	# of Memory Access W-Sum	
		Read(Reduction)	Write(Reduction)
SL	insert	4460946(-8.3%)	2122027(-5.3%)
	search	2606982(-1.9%)	1204355(-1.4%)
ULL	insert	84180(-88.0%)	158774(-46.9%)
	delete	14564(-86.1%)	14564(-66.5%)
	search	38201(-81.1%)	38201(-52.9%)
PQ	insert	993(-24.6%)	1352(-2.6%)
	remove	6108(-36.7%)	7419(-3.2%)
AVL	insert	28934(-48.0%)	34985(-74.4%)
	delete	1357135(-2.5%)	703537(-23.0%)

5.4 Evaluation

The summaries play an important role in improving the exploration efficiency of SSRD. Table 5.2 compares the performance difference with/without summarization. We observed that with summaries (W-Sum) the execution time is reduced by 14.8% to 91.9% over without summaries (WO-Sum) across the benchmarks. The overall efficiency of SSRD is much better than Cloud9. However, there is one exception - the insert action for priority-queue slows down when summaries are used. This is because the paths contained in summaries are not encountered. Hence summaries do not yield any benefits, while cost is incurred for generating and invoking them.

Table 5.3 provides the reductions in constraints solving time when summarization is used. It shows that constraints solving accounts for a significant portion of the total time, 22.54% to 95.64% across the benchmarks. With summaries, we observed reductions ranging from 35% to 97% in constraints solving time across the benchmarks, while the number of constraints solved is reduced by 37.5% to 98.3%. The constraints solving time reflects the time spent on solving the branch condition and symbolic pointers. This is reduced using summaries since there are solved branch conditions and generated data shapes in the saved path summaries.

Table 5.4 presents the performance improvements of SSRD due to the reductions in memory accesses. The number of read and write operations is reduced via summaries by 1.4% to 88.0% since there are memory read and write operations that are eliminated if their results are pre-computed during concolic unit testing.

Chapter 6

Conclusions and Future Work

In this dissertation, we studied concolic testing of programs with concurrent dynamic data structures, identified issues with effectiveness and efficiency, and developed techniques to address them. First, an automated data structure shape generation algorithm was developed to exercise a selected path. Second, a shape integration algorithm was developed to cause multiple threads to exercise chosen paths simultaneously. Finally, to improve the efficiency of data shape exploration, we presented function summarization via concolic unit testing to enable the reuse of symbolic expressions and the precomputed data shapes when finding a data shape that exercises chosen interprocedural paths.

We developed the DSGEN prototype that expands the applicability of concolic testing to CUDA programs involving concurrent dynamic data structures. Our approach enables the automatic generation of dynamic data structures of different shapes which cause different program paths to be exercised by multiple threads. Our experience shows that DSGEN is effective in testing complex data structures (B-Tree, HAMT, RRB-Tree, and SkipList).

We also developed the SSRD prototype - an efficient and effective approach for concolic testing of multithreaded programs that employ concurrent dynamic data structures with the goal of uncovering data races. The key contributions of this work include summary computation and exploitation for efficiency and non-conflicting shape determination and thread scheduling to guide concolic testing to expose data races. Our evaluation on Skip List, Unrolled Linked List, AVL-Tree, and Priority Queue shows that our approach is significantly more effective in uncovering data races than Cloud9 or AFL++. The reuse of summaries leads to the lightweight creation of objects and elimination of memory accesses during concolic testing thus significantly reducing its cost.

Future Work. The current prototypes that were developed have a few limitations. One is for the loop structure with symbolic iteration numbers. A loop contains many paths but only a few are captured by summarization, for example, we will only capture the first few iterations. Hence reuse of the summary is limited. In future work, the loop structure can be summarized with symbolized loop invariants and number of iterations to increase the reuse of loop structure. Another limitation is with respect to support for atomic operations in SSRD. The current prototype is focusing on pthread API with lock-based synchronization primitives. The support for lock-free data structure and atomic operations should be added to support a wider variety of concurrent programs.

Bibliography

[1] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. *SIGARCH Comput. Archit. News*, 19(3):234–243, apr 1991.

[2] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–242, 2005.

[3] Rahul Agarwal and Scott D Stoller. Type inference for parameterized race-free java. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 149–160. Springer, 2004.

[4] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Building an efficient hash table on the gpu. In *GPU Computing Gems Jade Edition*, pages 39–53. Elsevier, 2012.

[5] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381. Springer, 2008.

[6] Zachary R Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. *ACM Sigplan Notices*, 44(6):98–109, 2009.

[7] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D Owens. Gpu lsm: A dynamic dictionary data structure for the gpu. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, pages 430–440, 2018.

[8] Muhammad A Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D Owens. Engineering a high-performance gpu b-tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 145–157, 2019.

[9] Domagoj Babić and Alan J Hu. Structural abstraction of software verification conditions. In *International Conference on Computer Aided Verification*, pages 366–378. Springer, 2007.

[10] David F Bacon, Robert E Strom, and Ashis Tarafdar. Guava: A dialect of java without data races. *ACM SIGPLAN Notices*, 35(10):382–400, 2000.

[11] Phil Bagwell. Ideal hash trees. epfl. Technical report, 2001.

[12] Philip Bagwell and Tiark Rompf. Rrb-trees: Efficient immutable vectors. epfl. Technical report, 2011.

[13] Mark S Baranowski and Ganesh Gopalakrishnan. Gkleepp: Parallelizing a symbolic gpu race checker.

[14] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Bradley C Kuszmaul. Concurrent cache-oblivious b-trees. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 228–237, 2005.

[15] Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. *ACM SIGPLAN Notices*, 49(10):491–506, 2014.

[16] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[17] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 56–69, 2001.

[18] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated dynamic analysis of cuda programs. In *Proceedings of the Third Workshop on Software Tools for MultiCore Systems*, page 33, 2008.

[19] Alex D Breslow, Dong Ping Zhang, Joseph L Greathouse, Nuwan Jayasena, and Dean M Tullsen. Horton tables: Fast hash tables for in-memory data-intensive computing. In *Proceedings of the USENIX Annual Technical Conference*, ATC '16, pages 281–294, 2016.

[20] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, pages 183–198, 2011.

[21] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the IEEE 31st International Conference on Software Engineering*, ICSE '09, pages 463–473, 2009.

[22] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, 2008.

[23] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 209–224, 2008.

[24] Elias Castegren and Tobias Wrigstad. Reference Capabilities for Concurrency Control. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[25] Florence Charreteur and Arnaud Gotlieb. Constraint-based test input generation for java bytecode. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 131–140, 2010.

[26] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342, 2020.

[27] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, 2002.

[28] Insang Chung and James M Bieman. Generating input data structures for automated program testing. *Software Testing, Verification and Reliability*, 19(1):3–36, 2009.

[29] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[30] Richard A DeMillo, A Jefferson Offutt, et al. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[31] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, 1991.

[32] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *PADD '91*, 1991.

[33] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J Boehm. Ifrit: interference-free regions for dynamic data-race detection. In *Proceedings of the*

ACM international conference on Object oriented programming systems languages and applications, pages 467–484, 2012.

[34] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. Barracuda: binary-level analysis of runtime races in cuda programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, pages 126–140, 2017.

[35] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. *ACM SIGPLAN Notices*, 42(6):245–255, 2007.

[36] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS operating systems review*, 37(5):237–252, 2003.

[37] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 37–47, 2013.

[38] Jordan Fix, Andrew Wilkes, and Kevin Skadron. Accelerating braided b+ tree searches on a gpu with cuda. In *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance*, A4MMC '11, 2011.

[39] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. *ACM Sigplan Notices*, 44(6):121–133, 2009.

[40] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. *ACM Transactions on Graphics (TOG)*, 30(6):1–8, 2011.

[41] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, 2007.

[42] Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In *International Conference on Computer Aided Verification*, pages 68–81. Springer, 2007.

[43] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 21–40, New York, NY, USA, 2012. Association for Computing Machinery.

[44] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. A true positives theorem for a static race detector. 3(POPL), jan 2019.

[45] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Software Engineering Notes*, 23(2):53–62, 1998.

[46] Arnaud Gotlieb, Tristan Denmat, and Bernard Botella. Goal-oriented test data generation for programs with pointer variables. In *Proceedings of the 29th Annual International Computer Software and Applications Conference*, volume 1 of *COMPSAC '05*, pages 449–454, 2005.

[47] Arnaud Gotlieb, Tristan Denmat, and Bernard Botella. Goal-oriented test data generation for pointer programs. *Information and Software Technology*, 49(9-10):1030–1044, 2007.

[48] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 156–166, 2012.

[49] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '98, page 231–244, 1998.

[50] Neelam Gupta, Aditya P Mathur, and ML Soffia. Una based iterative test data generation and its evaluation. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, ASE '99, pages 224–232, 1999.

[51] Jerry J. Harrow. Runtime checking of multithreaded applications with visual threads. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, pages 331–342, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[52] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded c programs via lazy sequentialization. In *International Conference on Computer Aided Verification*, pages 585–602. Springer, 2014.

[53] Dennis Jeffrey, Yan Wang, Chen Tian, and Rajiv Gupta. Isolating bugs in multithreaded programs using execution suppression. *Software: Practice and Experience*, 41(11):1259–1288, 2011.

[54] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2019.

[55] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *International Conference on Computer Aided Verification*, pages 226–239. Springer, 2007.

[56] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, pages 33–37, 2012.

[57] Farzad Khorasani, Mehmet E Belviranli, Rajiv Gupta, and Laxmi N Bhuyan. Stadium hashing: Scalable and flexible hashing on gpus. In *Proceedings of the International Conference on Parallel Architecture and Compilation*, PACT '15, pages 63–74, 2015.

[58] Sarfraz Khurshid and Yuk Lai Suen. Generalizing symbolic execution to library classes. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 103–110, 2005.

[59] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. Parallel multi-dimensional range query processing with r-trees on gpu. *Journal of Parallel and Distributed Computing*, 73(8):1195–1207, 2013.

[60] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[61] Bogdan Korel. A dynamic approach of test data generation. In *Proceedings of the Conference on Software Maintenance*, ICSM '90, pages 311–317, 1990.

[62] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012.

[63] Guodong Li and Ganesh Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE '10, 2010.

[64] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. Gklee: concolic verification and test generation for gpus. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 215–224, 2012.

[65] Hongbo Li, Sihuan Li, Zachary Benavides, Zizhong Chen, and Rajiv Gupta. Compi: Concolic testing for mpi applications. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, pages 865–874, 2018.

[66] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in cuda programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.

[67] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Practical symbolic race checking of gpu programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 179–190, 2014.

[68] Pengcheng Li, Chen Ding, Xiaoyu Hu, and Tolga Soyata. Ldetector: A low overhead race detector for gpu programs. In *Proceedings of the 5th Workshop on Determinism and Correctness in Parallel Programming*, WODET '14, 2014.

[69] Pengcheng Li, Xiaoyu Hu, Dong Chen, Jacob Brock, Hao Luo, Eddy Z Zhang, and Chen Ding. Ld: Low-overhead gpu race detection without access monitoring. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):1–25, 2017.

[70] Lijuan Luo, Martin DF Wong, and Lance Leong. Parallel implementation of r-trees on the gpu. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference*, ASPDAC '12, pages 353–358, 2012.

[71] Nashat Mansour and Miran Salame. Data generation for path testing. *Software Quality Journal*, 12(2):121–136, 2004.

[72] Christophe Meudec. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software testing, verification and reliability*, 11(2):81–96, 2001.

[73] Nurit Moscovici, Nachshon Cohen, and Erez Petrank. A gpu-friendly skiplist algorithm. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*, PACT '17, pages 246–259, 2017.

[74] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. *ACM SIGPLAN Notices*, 42(1):327–338, 2007.

[75] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[76] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. *SIGPLAN Not.*, 42(6):22–31, jun 2007.

[77] Hiroyasu Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium*, pages 127–138, 2004.

[78] Brian Norris and Brian Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. *SIGPLAN Not.*, 48(10):131–150, oct 2013.

[79] Robert O'callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, 2003.

[80] Peizhao Ou and Brian Demsky. Checking concurrent data structures under the c/c++ 11 memory model. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 45–59, 2017.

[81] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. Curd: A dynamic cuda race detector. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, page 390–403, 2018.

[82] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2155–2168, 2017.

[83] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with {SymCC}: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[84] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *NDSS*, 2021.

[85] Eli Pozniansky and Assaf Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[86] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. Gpu-based parallel r-tree construction and querying. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 618–627. IEEE, 2015.

[87] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[88] Jake Roemer, Kaan Genç, and Michael D Bond. Smarttrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 747–762, 2020.

[89] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

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

[91] Sittisak Sai-ngern, Chidchanok Lursinsap, and Peraphon Sophatsathit. An address mapping approach for test data generation of dynamic linked structures. *Information and Software Technology*, 47(3):199–214, 2005.

[92] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[93] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[94] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '06, pages 419–423. Springer, 2006.

[95] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[96] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '05, page 263–272, 2005.

[97] Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *International Conference on Computer Aided Verification*, pages 300–314. Springer, 2006.

[98] Valerio Senni and Fabio Fioravanti. Generation of test data structures using constraint logic programming. In *Proceedings of the International Conference on Tests and Proofs*, pages 115–131. Springer, 2012.

[99] Vineet Singh, Rajiv Gupta, and Iulian Neamtiu. Automatic fault location for data structures. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the International Conference on Compiler Construction*, CC '16, pages 99–109, 2016.

[100] Anastasis A Sofokleous and Andreas S Andreou. Automatic, evolutionary test data generation for dynamic software testing. *Journal of Systems and Software*, 81(11):1883–1898, 2008.

[101] Nicholas Sterling. {WARLOCK}-a static data race analysis tool. In *{USENIX} Winter 1993 Conference ({USENIX} Winter 1993 Conference)*, 1993.

[102] Xiaofan Sun and Rajiv Gupta. Dsgen: Concolic testing gpu implementations of concurrent dynamic data structures. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 75–87, New York, NY, USA, 2021. Association for Computing Machinery.

[103] Xiaofan Sun and Rajiv Gupta. Dsgen: concolic testing gpu implementations of concurrent dynamic data structures. In *Proceedings of the ACM International Conference on Supercomputing*, pages 75–87, 2021.

[104] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, pages 350–360, 2018.

[105] Nischai Vinesh and M Sethumadhavan. Confuzz—a concurrency fuzzer. In *First International Conference on Sustainable Technologies for Computational Intelligence*, pages 667–691. Springer, 2020.

[106] Srinivas Visvanathan and Neelam Gupta. Generating test data for functions with pointer inputs. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, ASE '02, pages 149–160, 2002.

[107] Christoph Von Praun and Thomas R Gross. Object race detection. *Acm Sigplan Notices*, 36(11):70–82, 2001.

[108] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, 2007.

[109] Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 23–32, 2009.

[110] Nicky Williams, Bruno Marre, and Patricia Mouy. On-the-fly generation of k-path tests for c functions. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '04, pages 290–297, 2004.

[111] Mingyuan Wu, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. Characterizing and detecting cuda program bugs. *arXiv preprint arXiv:1905.01833*, 2019.

[112] Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. In *Computer Graphics Forum*, volume 29, pages 1297–1304. Wiley Online Library, 2010.

[113] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. Postconditioned symbolic execution. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[114] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, 2005.

[115] Jian Zhang. Symbolic execution of program paths involving pointer structure variables. In *Proceedings of the Fourth International Conference on Quality Software*, QSIC '04, pages 87–92, 2004.

[116] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

[117] Ruilian Zhao and Qing Li. Automatic test generation for dynamic data structures. In *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management and Applications*, SERA '07, pages 545–549, 2007.

[118] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. Grace: A low-overhead mechanism for detecting data races in gpu programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 135–146, 2011.

[119] Mai Zheng, Vignesh T Ravi, Feng Qin, and Gagan Agrawal. Gmrace: Detecting data races in gpu programs via a low-overhead scheme. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):104–115, 2013.

[120] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):1–11, 2008.