DSGEN: Concolic Testing GPU Implementations of Concurrent Dynamic Data Structures

Xiaofan Sun
xsun042@ucr.edu
University of California, Riverside
Riverside, CA, USA

Rajiv Gupta
gupta@cs.ucr.edu
University of California, Riverside
Riverside, CA, USA

ABSTRACT

Concolic testing combines concrete execution with symbolic execution along a path to automatically generate new test inputs that exercise program paths and deliver high code coverage during testing. The GKLEE tool uses this approach to expose data races in CUDA programs written for execution on GPUs. In programs employing concurrent dynamic data structures, automatic generation of a data structure with appropriate shape is necessary to cause threads to follow selected, possibly divergent, paths. In addition, a single non-conflicting data structure shape must be generated that simultaneously causes multiple threads to follow their respective chosen paths. When an execution exposes a bug (e.g., a data race), the generated data structure shape helps the programmer understand the cause of the bug. Because GKLEE does not permit pointers that form the shape of the dynamic data structure to be made symbolic, it cannot automatically generate data structures of different shapes and must rely on the user to write code that constructs them to exercise desired paths. We have developed DSGEN for automatically generating non-conflicting dynamic data structures with different shapes and integrated it with GKLEE to facilitate uncovering of data races in programs that employ complex concurrent dynamic data structures. In comparison to GKLEE, DSGEN increases the number of races detected from 10 to 25 by automatically generating a total of 1,897 shapes in implementations of four complex concurrent dynamic data structures – B-Tree, Hash-Array Mapped Trie, RRB-Tree, and Skip List.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Concolic Testing, Data Structure Shapes, Symbolic Linking Pointers, CUDA Programs, Data Races.

ACM Reference Format:
Xiaofan Sun and Rajiv Gupta. 2021. DSGEN: Concolic Testing GPU Implementations of Concurrent Dynamic Data Structures . In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3447818.3460962

1 INTRODUCTION

Today Graphics Processing Units (GPU) are being extensively used in high-performance computing due to their ability to efficiently support high degree of parallelism. Increasingly general-purpose parallel applications are being implemented on GPUs, including those employing variety of concurrent dynamic data structures [1–3, 23, 26]. The concurrent operations on such structures can lead to bugs (e.g., data races). Therefore their thorough testing is very important. Data races in GPU programs can be detected using static symbolic evaluation based methods [18, 21], dynamic methods [4, 11, 24], and combined static and dynamic methods [35, 36]. Since static methods are conservative, they lead to false positives. Dynamic methods typically report true races, assuming dynamic synchronizations are accurately captured and not missed. However, uncovering of races is dependent upon selection of suitable program input and many races present may not be exposed by a given input. The combined analysis techniques use static analysis to identify potential races and then limit program instrumentation to dynamically verify manifestation of some potential races. The above techniques are effective for programs with simple control flow and branch predicates that can be readily analyzed using static analysis or exercised using manually selected inputs.

As GPUs are increasingly being used by general-purpose parallel applications with complex control flow, branch conditions, and concurrency structure, there is a need for employing more powerful automated techniques. Concolic execution is such a technique that performs symbolic evaluation along a concrete execution path, systematically and automatically generating new inputs to exercise and thus test different execution paths for improved code coverage [31]. This approach is used by GKLEE [19] to detect data races in multithreaded CUDA programs. It is built using the KLEE [6] concolic testing system for single-threaded programs. The concolic testing loop that generates new inputs is driven by user identified variables that are made symbolic, leading to derivation of symbolic constraints for branch conditions that drive computation of new inputs to exercise different branch outcomes.

While GKLEE is a superior state-of-the-art system, it also has limitations. In particular, it cannot adequately exercise code implementing and using *concurrent dynamic data structures* [1–3, 23, 26]. Overcoming this limitation requires addressing the following two challenges related to the shapes of dynamic data structures.

- **Symbolic Pointer-based Dynamic Data Structures** – In such programs, dynamic data structures with *different shapes* are needed to exercise different paths. 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 structure of suitable shape to exercise a given path by a thread. GKLEE does not support symbolic linked dynamic data structures.

- **Non-Conflicting Shape** – Since multiple threads are involved in data races, different dynamic data structures generated to cause multiple threads to traverse their respective selected paths must be integrated to generate a single *non-conflicting* dynamic data structure whose shape simultaneously causes all threads to follow their chosen paths.

In this paper, we present DSGEN that addresses the above challenges and automatically generates dynamic data structures of suitable shapes that cause multiple threads to exercise GPU kernels implementing concurrent dynamic data structures. GKLEE's symbolic VM considers threads one by one and interacts with DSGEN to continually construct their dynamic data structures to satisfy constraints involving symbolic linking pointers. The satisfaction of constraints for individual threads generates data structures with shapes that cause threads to follow their respective paths. The satisfaction of constraints for multiple threads requires integrating the dynamic data structures with different shapes into a single non-conflicting data structure with the proper shape that simultaneously causes all threads to follow their respective paths. Generating dynamic data structures of suitable shapes causes threads to follow different combinations of frequently executed convergent paths and infrequently executed divergent paths, DSGEN enables higher path coverage and exposes data races that are then detected using GKLEE's ability to collect and analyze execution traces.

We have developed a prototype of DSGEN which is integrated with GKLEE. To demonstrate its capabilities, we use it to generate dynamic data structures of suitable shapes that uncover data races in highly parallel GPU implementations of B-Tree [1], Hash-Array Mapped Trie (HAMT) [2], RRB-Tree [3], and Skip List [23, 26]. In comparison to GKLEE, DSGEN increases the number of races detected from 10 to 25 by automatically generating a total of 1,897 shapes in the implementations of the above four complex concurrent dynamic data structures. Though we demonstrate the use of DSGEN in expanding the applicability of GKLEE's data race detection capability to a new class of programs, the shape exploration based concolic testing introduced can also be used to uncover other kinds of input shape sensitive bugs.

2 GKLEE: CAPABILITIES AND LIMITATIONS

GKLEE [19] is a symbolic execution engine for CUDA programs. The loader packages the symbolic variables, the memory model, current instruction, threads information, and empty constraints set into the *running state* of all GPU threads. A State Queue contains all the possible running states of all the threads to allow exploration of all possible paths. The emulation loop of GKLEE execution engine, during each of its iteration, takes a state from the queue and issues an instruction from the current running thread resulting in update of the memory model and addition of new constraints in terms of symbolic variables. Variables such as `blockIdx` and `threadIdx` are symbolic and in addition user can make program variables symbolic using GKLEE's API as follows:

```
__device__ int a;
klee_make_symbolic(&a, sizeof(int), "a");
```

Upon reaching a barrier (e.g., a function call to "`__syncthreads()`" or the end of a function), the state switches to running the next thread. After all threads reaching the barrier, race detection is performed and data race report is generated for the user.

GKLEE efficiently explores a potentially large space of execution states, involving a large number of threads, by leveraging the power of symbolic analysis. In particular, it embodies two important ideas: *canonical schedule* [19] and *parametric flows* [21]. As demonstrated by authors of GKLEE, these features greatly contribute to GKLEE's scalability. Moreover, these techniques preserve the soundness of data race detection [19, 21].

The *canonical schedule* fully executes each thread till it reaches a barrier, before switching execution to another thread. If the program is race free, the canonical schedule is equivalent to any other schedule. When potentially conflicting memory accesses (read-write or write-write) among threads are possible, the SMT solver picks concrete values for thread ids and data values that cause the conflict to manifest itself. The precision of the SMT solver ensures the absence of false alarms.

The idea of *parametric flows* allows efficient handling of a large number of threads by partitioning the space of executions into parametric flow equivalence (PFE) classes such that all intra-warp races can be detected by considering a pair of threads from the same parametric flow while all inter-warp races can be detected by considering one thread each from two different parametric flows. The SMT solver identifies the specific threads and data values that causes the data race to manifest. Note that the Grid Size and Block Size determine the number of thread blocks and threads within each block. Limits on their sizes play a role when the solver identifies the values of bid and tid for the racing threads.

Typically in SIMD programs, different threads operate on different data items while following the same execution path. In contrast, when considering concurrent dynamic data structures, multiple threads frequently follow divergent paths that may access the same data fields giving rise to data races. Generation of inputs that exercise the complex control flow due to divergent paths, requires the power of concolic testing. Since GKLEE does not support symbolic pointer-based linked data structures, it cannot generate different dynamic data structure shapes. Thus, to enable the complex control flow to be thoroughly exercised, the burden of creating suitably shaped data dynamic structures falls on the user.

3 DSGEN OVERVIEW

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

Figure 1 provides an overview of the integrated DSGEN and 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

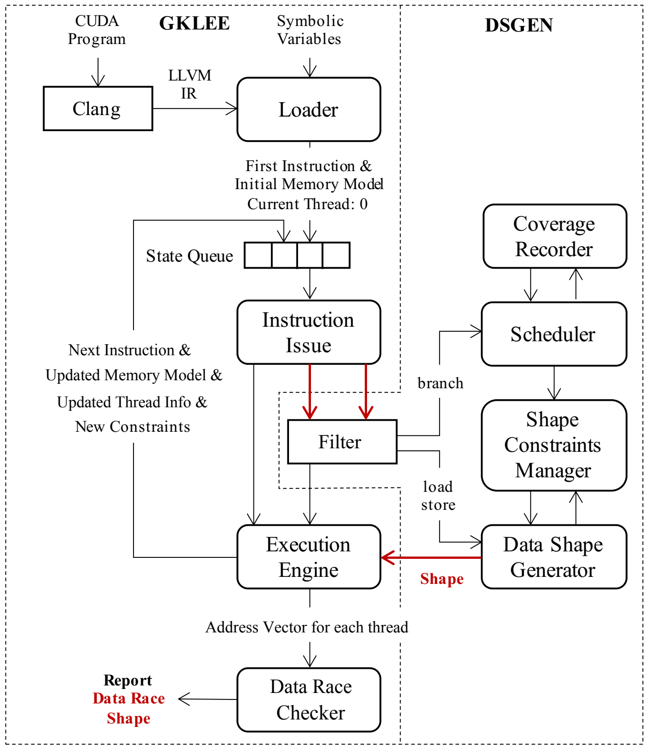


Figure 1: DSGEN + GKLEE Prototype.

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. Selection of alternate paths leads to 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 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 generation of a test case that exercise 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 is 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.

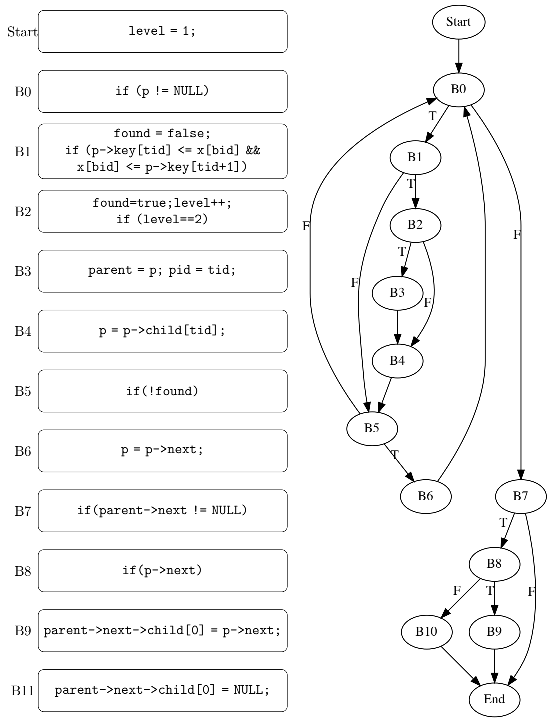
Table 1: DSGEN API for dynamic data structures.

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)

Table 1 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 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 modelled 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 dynamic linked data structures, GKLEE does not provide any special support. It handles pointer variables using the simple methodology used by underlying KLEE system. Unfortunately, this makes input generation when testing functions of a library implementing concurrent data structures a problem. To



```
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;
B11    parent->next->child[0] = NULL;
```

Figure 2: The Control Flow Graph of cu_skiplist_search.

exercise execution 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, pointer 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 number of *levels of links* allowed. Limits on array sizes and number of levels of links limit the size of the dynamic data structure which translates into limits in lengths of paths that are explored during depth first exploration of paths.

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 Listing 1 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 2. 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

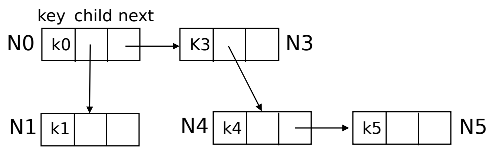
```
1  #define SIZE 1
2  struct Node {
3    int key[SIZE+1];
4    Node* child[SIZE];
5    Node* next;
6  };
7
8  __global__ void
9  cu_skiplist_search(Node* p, int* x) {
10   int bid = blockIdx.x, tid = threadIdx.x;
11   Node* parent = NULL; int pid;
12   int level = 1;
13   __shared__ bool found;
14   while(p != NULL) {
15     found = false;
16     if(p->key[tid]<=x[bid] &&
17        x[bid]<=p->key[tid+1]) {
18       found = true;
19       level++;
20       if (level == 2) { parent = p; pid = tid; }
21       p = p->child[tid];  // find in its child
22     }
23     if (!found) {
24       p = p->next; // find in the same level
25     }
26   }
27   if (parent->next!=NULL) {
28     if (p->next)
29         parent->next->child[0] = p->next;
30     else parent->next->child[0] = NULL;
31   }
32 }
33
34 int main() {
35   Node* root; int k[2];
36   klee_make_data_structure(
37     &root, sizeof(root), "root");
38   klee_make_symbolic(&k, sizeof(int)*2, "k");
39   // grid size=2 and block size=1
40   cu_skiplist_search<<<2, 1>>>(root, k);
41   return 0;
42 }
```

Listing 1: Skip-List CUDA Implementation.

`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.



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

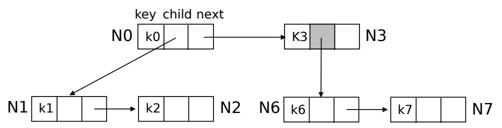
(2) The *second read-write race* is between read access of the pointer field `p->child[0]` at line 21 and write access of the pointer field `parent->next->child[0]` at line 29 where the updating of `child[0]` field at line 29 conflicts with reading of the same node by another thread at line 21.

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 two 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, neither data race arises for the path pair shown in Figure 3 which takes a false branch at line 27. However, the first race is exposed by another path pair shown in Figure 4 where Thread a executes line 21 and Thread b takes true branch at line 27 but 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 3 the path taken does not cause a read-write race while for data structure shape in Figure 4 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 3, 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 such as if `(p->next!=NULL)` or not. On 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 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



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

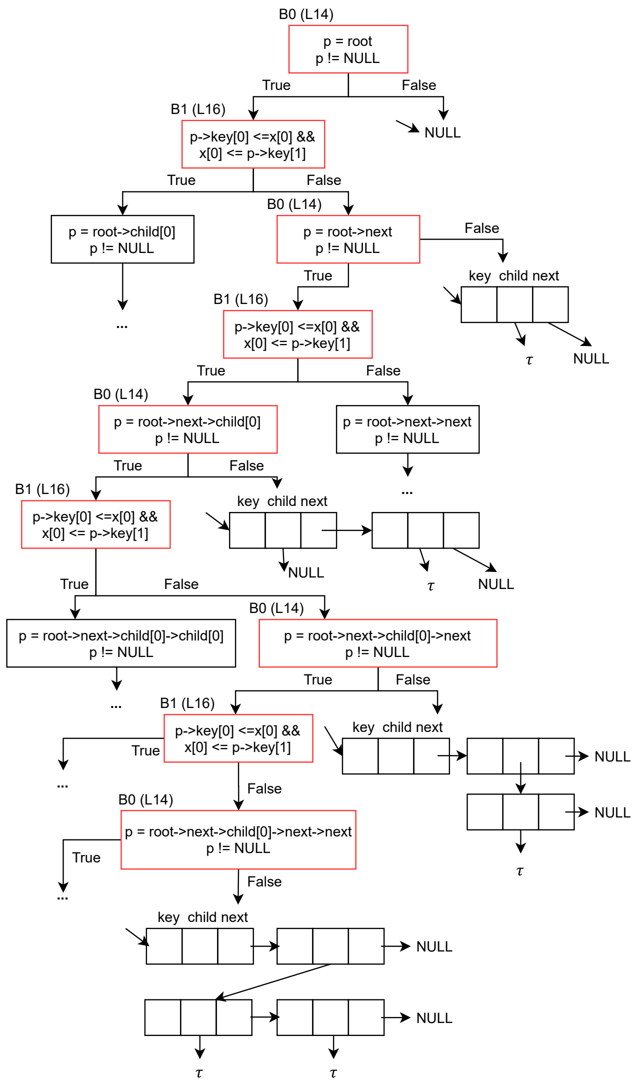


Figure 5: The states exploration of dynamic data structure by Thread b of Figure 4.

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 5 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 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 and it leads to 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 DATA SHAPE GENERATION ALGORITHM

Data shape generation algorithms that we present automatically generate 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 (Section 4.1); and compacting the generated data structures into one non-conflicting data structure (Section 4.2).

There are two types of situations encountered during compaction. The first and simpler situation is one 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 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. The second and 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 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.

4.1 Generating a Data Structure that Causes a Given Thread to Follow a Selected Path

The dynamic data structure generation described in this section is inspired by the method proposed in [33] 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 DSGEN to create a concolic testing framework capable of exploring different data structure shapes and handling execution of multiple threads.

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 data structure shape to satisfy them, and passes the temporary data structure to GKLEE so it can successfully execute the memory access. To achieve the above, DSGEN collects two kinds of information – *Path Constraints* (PC) and *Pointer-Pointee Relations* (PPRs) – described below.

- **Path 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 all the selected path. 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 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 also 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 selected path.

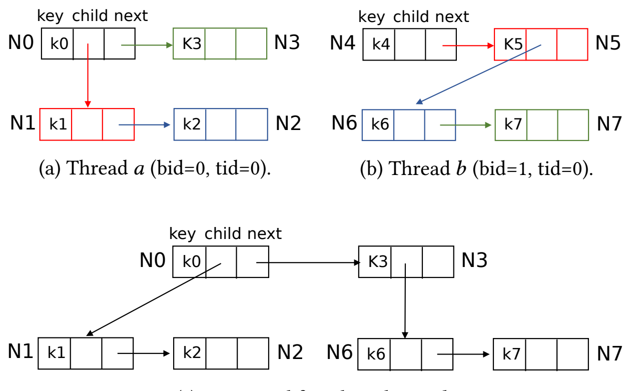
The data structure shapes formed by PPRs and the PCs associated with the fields belonging to a symbolic data structure, are the result of DSGEN's actions that are determined by the kind of operations encountered: *pointer initialization*, *pointer dereferencing*, *pointer assignments*, and *branch conditions*. For example, first time dereferencing of a pointer typically causes memory allocation that expands the data structure and generates constraint 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.

An Example. Next we illustrate the above actions using the execution of two threads along paths from Figure 4 that take the true branch at line 27 and false branch at line 28 (recall that this execution exposes the first data race). In Figure 6 the table at the top gives the PCs and PPRs corresponding to the data structures generated by the two threads, the generated data structures are shown next in (a) and (b), and finally (c) shows the integrated data structure. The matching colors (**black**, **red**, **blue**, **green**) indicate the correspondence between PCs+PPRs and relevant portion of the data structure. The PCs+PPRs are generated as follows:

– **(Black)** – The subset of constraints and portion of data structure shown in black represents the status starting with the execution of line 16 of function `cu_skiplist_search` in Listing 1. For Thread a path constraints `N0.key[0] <= x[bid]` and `x[bid] <= N0->key[1]` are added by the branch condition at lines 16-17 where `N0` is the local instance of Thread a's variable p. The black part of the shape is created. In $B6$, the loop goes into the next iteration by updating the pointer `p` with its children.

BB (LN)	**PCs: Thread *a***	BB (LN)	**PCs: Thread *b***
B1 (L16)	N0.key[0] <= x[bid]	B1 (L17)	x[bid] > N4.key[1]
B1 (L17)	x[bid] <= N0.key[1]	B0 (L14)	N4.next ≠ null
B0 (L14)	N0.child[0] ≠ null	B1 (L16)	N5.key[0] <= x[bid]
B1 (L17)	x[bid] > N1.key[1]	B1 (L17)	x[bid] <= N5.key[1]
B0 (L14)	N1.next ≠ null	B0 (L14)	N5.child[0] ≠ null
B1 (L16)	N2.key[0] <= x[bid]	B1 (L17)	x[bid] >= N6.key[1]
B1 (L16)	x[bid] <= N2.key[1]	B0 (L14)	N6.next ≠ null
B7 (L27)	N0.next ≠ null	B1 (L16)	N7.key[0] <= x[bid]
B10 (L30)	N3.child[0] = null	B1 (L17)	x[bid] <= N7.key[1]

	PPRs: Thread *a*			**PPRs: Thread *b***	
BB (LN)	Pointer	Pointee	BB (LN)	Pointer	Pointee
B1 (L16)	N0.child[0]	N1	B1 (L16)	N4.next	N5
B1 (L16)	N1.next	N2	B1 (L16)	N5.child[0]	N6
B10 (L30)	N0.next	N3	B1 (L16)	N6.next	N7



(a) Thread *a* (bid=0, tid=0).

(b) Thread *b* (bid=1, tid=0).

(c) Integrated for Threads *a* & *b*.

Figure 6: A generated shape of Skip-List data structure

– **(Red)** – After the next iteration's update of the pointer `p` at line 21, the condition `p != NULL` implies that pointer must not be null; thus, the path constraint `N0.child[0]` $\neq$ `null` is generated. Due to dereferencing by access `N0.child[0]` in branch condition at line 16, `N1` is allocated and PPR between `N0.child[0]` and `N1` is created. All pointers that do not have concrete addresses are initially assigned unique integer values as identifiers. The red part of the shape is created when first dereference of $N1$ is encountered in $B1$. After creating child node `N1`, path constraint `x[bid] > N1.key[1]` is added due to the branch condition's result.

– **(Blue)** – In next iteration that explores `N1.next`, a similar process generates path constraint `N1.next` $\neq$ `null` and PPR between `N1.next` and `N2` where latter is allocated memory due to dereferencing via access `N1.next` in B1. After creating the new shape N2, the path constraints `N2.key[0] <= x[bid]` and `x[bid] <= N2.key[1]` are generated by the corresponding branch condition.

– **(Green)** – These constraints are created upon entry to $B9$ that accesses, i.e. dereferences, `parent->next->child[0]`. The branch condition in $B7$ evaluates to true and $B8$ to false in this example. The path constraint `N0.next` $\neq$ `null` is generated and `N0.next` is assigned by dereferencing in $B7$ and $B10$. A PPR between `N0.next` and `N3` is created. Then `N3.child[0]` is assigned with `null` by memory write in $B10$ creating the path constraint `N3.child[0]` = `null`. The execution of thread a has created PCs, PPRs, and corresponding data structure. Similarly Thread b creates its data structure. Finally, both data structures are integrated as described in the next section.

4.2 Integrating Data Structures for Different Threads to Create a Single Non-Conflicting Data Structure

We have shown in detail how the constraints for Thread a are collected and the data shape in Figure 6(a) is generated. Similar actions for thread b generate the shape in Figure 6(b). Next we will present the algorithm implemented by DSGEN's data shape generator that integrates the two shapes into one non-conflicting data structure that is shown in Figure 6(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, Ta and Tb, their path constraints $PC(Ta)$ and $PC(Tb)$, and pointer-pointee relations $PPR(Ta)$ and $PPR(Tb)$, we make the following key observations:

- **Feasibility** – Since the integrated data structure must simultaneously satisfy path constraints in $PC(Ta)$ and $PC(Tb)$, presence of a pair of conflicting constraints in $PC(Ta)$ and $PC(Tb)$ implies that no such integrated data structure exists. That is, the *feasibility* of threads Ta and Tb simultaneously following the chosen paths requires that $PC(Ta)$ and $PC(Tb)$ be *conflict-free*.
- **Adjustment** – The integrated data structure cannot in general be obtained by taking the union of $PPR(Ta)$ with $PPR(Tb)$. This is because corresponding fields in $PPR(Ta)$ and $PPR(Tb)$ may conflict with each other, i.e. have different pointees. Therefore integration essentially involves *adjustment* of *PPR*s to make them consistent such that the adjustments do not violate any constraints in $PC(Ta)$ and $PC(Tb)$, i.e., paths followed are preserved.

Next we present Algorithm 1 that, guided by above observations, *explores different adjustments* to *PPR*s so they can be made consistent without violating *PC*s. When conflicts among *PC*s 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 (e.g., those in Figure 6(a) and (b)) and converts the first pointed to by x into an integrated one (e.g., the one in Figure 6(c)) 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 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 the attribute of only x or only y has adjustable PPRs, then the adjustment procedure of the adjustable PPRs is carried out via a 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.

Algorithm 1: An algorithm for combining two data shapes generated by different threads

```
   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
1  Procedure COMBINE($x$, $y$):
2      if test_sets_conflict(cons($x$), cons($y$)) then
3          return COMBINING FAILED
4      foreach $attr \in x$ do
5          if (type($x.attr$)≠pointer) then
6              continue ▷ non-pointer details omitted
7          if val($x.attr$) = $\tau$ || val($y.attr$) = $\tau$ then
8              if val($y.attr$)≠$\tau$ then
9                  val($x.attr$)←val($y.attr$)
10                 ppr($x.attr$)←ppr($y.attr$)
11             cons($x.attr$)←cons($x.attr$)∪cons($y.attr$)
12         else if val($x.attr$)≠val($y.attr$) then
13             $A_x$ ← adjustable(ppr($x.attr$), cons($x.attr$))
14             $A_y$ ← adjustable(ppr($y.attr$), cons($y.attr$))
15             if $\neg A_x$ & $\neg A_y$ then
16                 return COMBINING FAILED
17             else if $A_y$ & $\neg A_x$ then
18                 RESOLVE($y.attr$, $x.attr$)
19             else if $A_x$ & $\neg A_y$ then
20                 RESOLVE($x.attr$, $y.attr$)
21             else
22                 COMBINE(val($x.attr$), val($y.attr$))
23                 cons($x.attr$) ← cons($x.attr$) ∪ cons($y.attr$)
24     return COMBINING SUCCEEDED

25 Procedure RESOLVE($\alpha$, $\beta$):
26     foreach ($z.attr, s$) ∈ ppr($\alpha$) do
27         foreach $pc$ ∈ cons($\beta$) do
28             if not has_conflict(ppr($\alpha$), $pc$) then
29                 continue
30             $ppr(\alpha)$ ← $ppr(\alpha)$ - DEPEND(($z.attr, s$))
31             $acons$ ← SIMPLIFY(($z.attr$), cons($\alpha$) ∪ cons($\beta$))
32             if $acons \notin const$ then
33                 ($succ$, $c$) ← PREDICT(($z.attr$), cons($\alpha$) ∪ cons($\beta$))
34                 if not $succ$ then
35                     COMPACTION FAILED
36                 $ppr(\alpha)$ ← $ppr(\alpha)$ ∪ $c$
37                 val($z.attr$)←SIMPLIFY(($z.attr$),cons($\alpha$) ∪ cons($\beta$))
38                 $cons(\alpha)$ ← cons($\alpha$) ∪ cons($\beta$)
39             else
40                 $ppr(\alpha)$ ← $ppr(\alpha)$ ∪ ($z.attr, acons$)
41                 val($z.attr$) ← $acons$
42                 $cons(\alpha)$ ← cons($\alpha$) ∪ cons($\beta$)
43             break
```

During integration, COMBINE makes use of the RESOLVE(α, β) procedure that removes those *PPR*s from α node that conflict with *PC*s in the β node. Given a single PPR *ppr*, DEPEND(*ppr*) returns all the constraints that can be inferred directly or indirectly from the left hand side of *ppr*. 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 method that solves c as the left hand side of an adjustable PPR using the set of inviolable constraints α to get the new result of right hand side of the PPR, and `has_conflict`(p, c) is another method that detects conflicts between a PPR p and a set of PCs c, if p does not satisfy PCs in c.

Now let us consider an illustration of integration performed by COMBINE. 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 6. The following execution call trace shows the steps of integration:

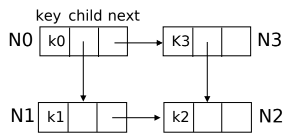
```
COMBINE ($N0$, $N4$)
11  cons($N0$.child[0]) ← cons($N0$.child[0]) ∪ cons($N4$.child[0])
22  COMBINE ( val($N0$.next), val($N4$.next) )
9     val($N3$.child[0]) ← val($N5$.child[0])
10    ppr($N3$.child[0]) ← ppr($N5$.child[0])
11    cons($N3$.child[0]) ← cons($N3$.child[0]) ∪ cons($N5$.child[0])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

Initially COMBINE is called with parameters $N0(Ta)$ and $N4(Tb)$. The shapes generated by threads Ta and Tb 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, COMBINE (line 22) is called for attribute `next` as data structures for both threads contain valid pointers and both $PPR(Ta)$ and $PPR(Tb)$ are adjustable. During the second recursive call to 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 6(c). The execution of COMBINE and hence the integration is complete. Note that the paths followed by the threads are preserved.

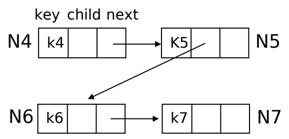
Next we consider a more complex situation if the branch at line 28 in Listing 1 takes true path where non-τ values are found in corresponding fields of data structures generated by the two threads and PPRs conflicts are involved. In Figure 7, we first show the PCs and PPRs, and then the two data structures generated are given in Figures 7(a) and (b). Note that in this example `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.

BB (LN)	**PCs: Thread *a***	BB (LN)	**PCs: Thread *b***
B0 (L14)	N0.child[0] ≠ null	B0 (L14)	N4.next ≠ null
B0 (L14)	N1.next ≠ null	B0 (L14)	N5.child[0] ≠ null
B7 (L27)	N0.next ≠ null	B0 (L14)	N6.next ≠ null
B9 (L29)	N3.child[0] = N1.next		

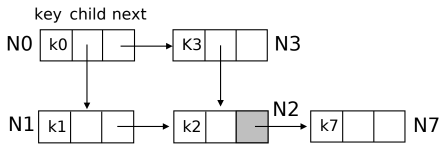
	PPRs: Thread *a*			**PPRs: Thread *b***	
BB (LN)	Pointer	Pointee	BB (LN)	Pointer	Pointee
B1 (L16)	N0.child[0]	N1	B1 (L16)	N4.next	N5
B1 (L16)	N1.next	N2	B1 (L16)	N5.child[0]	N6
B9 (L29)	N0.next	N3	B1 (L16)	N6.next	N7
B9 (L29)	N3.child[0]	N2			



(a) Thread *a* (bid=0, tid=0)



(b) Thread *b* (bid=1, tid=0)



(c) Integrated for Threads *a* & *b*

```
COMBINE ( N0, N4 )
  . . .
22 COMBINE ( val(N0.next), val(N4.next) )
20   RESOLVE ( N5, N3 )
30     α ← α − DEPEND ( (N5.child[0], N6) )
30     N2 ← SIMPLIFY ( (N5.child[0]), cons(N3) ∪ cons(N5) )
40     α ← α ∪ ( N5.child[0], N2 )
23 cons(N0.next) ← cons(N0.next) ∪ cons(N4.next)
  . . .
```

(d) Execution Call Trace of Algorithm 1.

Figure 7: Example of conflicts solving

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.

In Figure 7(d), we show a portion of the execution call trace for integration of $N3.child[0]$ for two threads. The fields `N0.next` and `N4.next`, in both threads *a* and *b*, point to different symbolic names $N3$ and $N5$ with different addresses. The condition at line 12 evaluates to true as both are valid pointers. Since `(N5.child[0], N6)` is adjustable while `(N3.child[0], N2)` is not, the condition at line 17 is true and RESOLVE is executed to merge `N5.child[0]` into $N2$ by removing PPRs of $N5$.

In RESOLVE, for each PPR(Tb) in the α node (e.g., `(N5.child[0], N6)` in the current function call), each path constraint is tested for conflicts at line 28. We set `N3 = N5` causing PCs and PPRs to be merged by eliminating the conflicting PPR `(N5.child[0], N6)`. In general, inferred PPRs, if any, must also be eliminated. The latter are identified by calling DEPEND function. Next SIMPLIFY function, at line 31, solves newly composed set of constraints. Using the set $cons(\alpha) \cup cons(\beta)$ which means the new PPR needs to satisfy both constraints from two threads, we simplify the constructed constraint for $(z.attr)$ which leads to the inference the new pointee of `N5.child[0]`. At the line 40, the result `(N5.child[0], N2)` is appended to the $ppr(\alpha)$ set. Finally, the data structure in Figure 7(c) is obtained. Note that the paths followed by the threads are preserved.

5 CASE STUDIES

5.1 Experimental Setup

To study the effectiveness of our tool in detecting data races, we implemented four important data structures and used them to compare DSGEN with original GKLEE. The comparison shows two advantages of our tool: 1) by automatically creating dynamic data structures of different shapes, it enables 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 2. Both read-write (rw) and write-write (ww) races, between threads from 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 discuss the four data structures considered.

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

- B-Tree – Self-Balancing Search Tree B-Tree [1] is a widely used data structure in databases. GPU accelerates dynamic queries and batch insertion. In Table 2, races 1-10 correspond to B-Tree. In the experiments, the Grid Size and Block Size limits were set to 2 and 16 respectively.
- HAMT – Hash-Array Mapped Trie [2] is an array mapped trie where the keys are hashed to ensure an even distribution of keys and have a constant key length. It achieves almost hash table-like speed while using memory much more efficiently. In Table 2, 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 [3]. The purpose of 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 2, 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 [23, 26]. A skip list is a probabilistic data structure that allows $O(\log n)$ search and insertion complexity within an ordered sequence of n elements. In Table 2, races 21-25 correspond to the Skip List data structure. The Grid Size and Block Size were limited to 2 and 8 respectively in the experiments.

Table 2: 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)

Since the implementations of above data structures are based upon the correct algorithms provided in the noted citations, our implementations did not create any data races. The data races were seeded in these implementations to enable comparison of 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. The number of adjustments performed in generating these data structures are also reported.
- 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. When racing threads are found, they occupy different positions in thread blocks. We also present the number of **thread positions** exercised by the generated concrete inputs.

5.2 Experimental Results

Effectiveness: Paths Explored and Races Exposed. The effectiveness of race detection is demonstrated by the results presented in Table 3 and Figure 8. We first note that all 25 races introduced in Table 2 were successfully identified by DSGEN, while only 10 were found by GKLEE. In particular, as shown in Table 3, 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.

As indicated in Table 3, 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. 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 8 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 10 gives the corresponding plot for GKLEE. We also give #Adjustments which is the total number of adjustments performed during the 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 and thus adjustments are needed to produce additional data structure shapes that can help uncover data races.

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 7 to expose race 21 and exactly equal to 7 to expose race 22.

Inputs Generated and Runtime Costs. Finally, in Table 4 and Figure 9 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

Table 3: 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

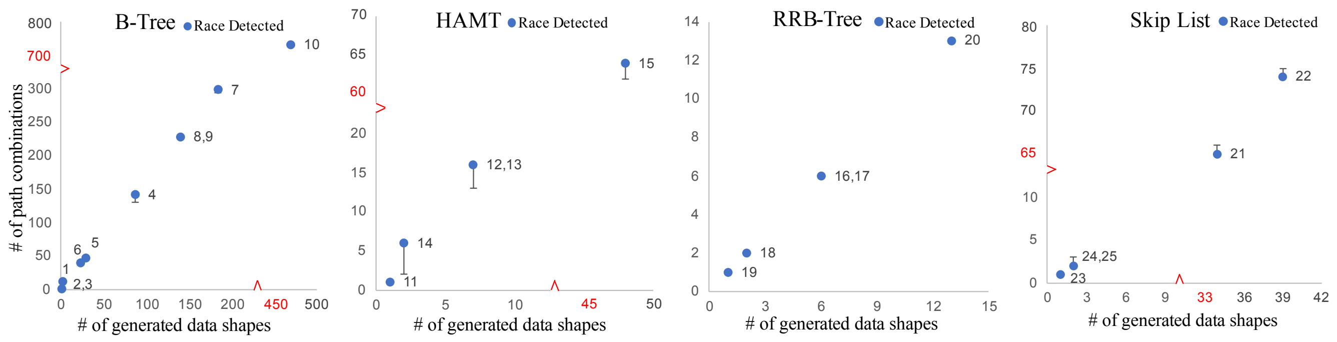


Figure 8: DSGEN vs. GKLEE: Data Structures Generated vs. Path Combinations Explored and Data Races Detected.

Table 4: DSGEN vs. GKLEE: Number of Inputs Generated via Concolic Execution, Execution Steps and Times (seconds).

	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)
For Exhaustive Exploration	GKLEE	DSGEN	GKLEE	DSGEN	GKLEE	DSGEN	GKLEE	DSGEN
# of Diff. Inputs	61	829	31	72	3	7	64	256
Total Execution Steps	31,280	67,049	7,582	13,120	516,848	520,668	14,175	52,161
Total Execution Time(s)	4.905	285.583	5.896	21.042	152.950	173.220	1.55	23.26

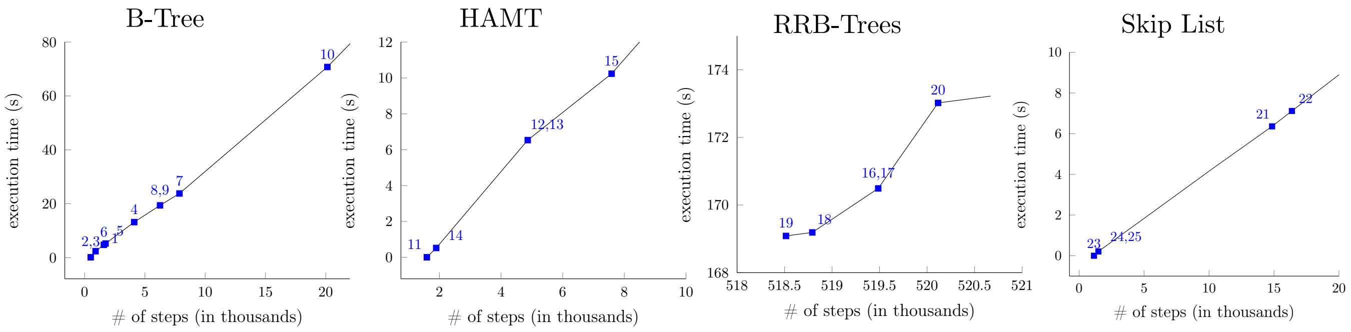


Figure 9: DSGEN vs. GKLEE: Cost of Concolic Testing in Terms of Execution Time vs. Number of Execution Steps.

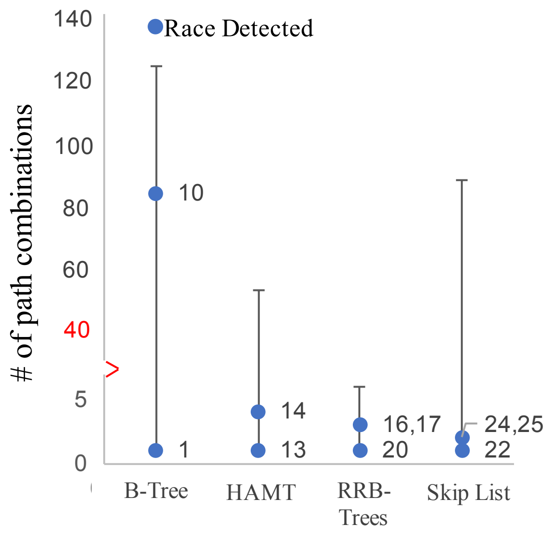


Figure 10: Path Combinations Explored by GKLEE using the provided default data structure shape.

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. 9 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.

6 RELATED WORK

There is rich literature on generating test inputs [5, 7, 8, 12, 13, 15–17, 22, 25, 29, 31, 33, 34]. A number of techniques are aimed at generating test input for a given path in a single-threaded program running on a CPU. Godzilla [10] and Gotlieb et al. [12] presented techniques based on constraint solving, Grechanik et al. [13] and Petsios et al. [25] employ feedback-directed fuzz testing, Mansour and Salame [22] developed stochastic search algorithms, and Gupta et al. [14] developed iterative numerical techniques. However, none of these approaches deal with dynamic data structures.

Path-based techniques in the presence of pointer-based dynamic data structures have also been developed by Korel and Bogdan [16, 17]. Chung and Bieman [8] generate data shapes using points-to information for statements along a selected path. Visvanathan and Gupta [33] 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. Sai-ngern et al. [29] also handle linked data structures, including homogeneous and heterogeneous recursive structures. Those 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 subpaths sharing across many individual paths.

Symbolic execution and concolic testing. Khurshid et al. [15] use symbolic execution to test library classes with generated set and map data structures in Java but does not consider user-defined data structures. Zhang [34] support symbolic pointers and symbolic data structures. Burnim et al. [5], in addition, aim to create a worst-case input. Unlike the above techniques, CUTE [31] supports concolic unit testing for C programs with data structure generation support. However, in comparison to DSGEN, the above methods are not aimed at multithreaded programs and more specifically are not able to adequately test implementations of concurrent data structures.

Concolic testing of multithreaded programs. jCUTE [30] is a concolic unit testing tool for multithreaded Java programs while Cloud9 [9] is an extension of KLEE with multithreading support for POSIX system. However, they require the user to manually create the data structures. COMPI applies concolic testing to efficiently test MPI programs [20]. GKLEE extends the above to GPU programs. As DSGEN's comparison with GKLEE shows, for good coverage needed to uncover data races it is essential to automatically search for suitable data structure shapes that cause concurrent threads to follow selected paths that expose data races.

7 CONCLUSIONS

We presented DSGEN that expands applicability of concolic testing to CUDA programs involving implementations of concurrent dynamic data structures. Our approach enables 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 such as B-Tree, HAMT, RRB-Tree, and SkipList. Though in this work the constraints on data structure shape are derived only from executed code, in the future we will consider augmenting the constraint set with shape specifications obtained via annotations and/or analysis [27, 28, 32]. Finally, while in this work we incorporate data structure shape generation in concolic testing framework for CUDA programs, the same ideas can also be developed to debug multithreaded programs running on multicores that access shared dynamic data structures.

ACKNOWLEDGEMENTS

We thank our shepherd and the reviewers for their detailed and constructive feedback for improving the presentation of this work. This work is supported in part by National Science Foundation grants CCF-1813173, CCF-2002554, and CCF-2028714 to the University of California Riverside.

REFERENCES

[1] Muhammad A Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D Owens. 2019. Engineering a high-performance GPU B-Tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. 145–157.
[2] Phil Bagwell. 2001. *Ideal hash trees. EPFL.* Technical Report.

[3] Philip Bagwell and Tiark Rompf. 2011. *RRB-Trees: Efficient Immutable Vectors. EPFL.* Technical Report.

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

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

[6] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. 2008. 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)*. 209–224.

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

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

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

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

[11] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. 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)*. 126–140.

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

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

[14] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. 1998. 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)*. 231–244.

[15] Sarfraz Khurshid and Yuk Lai Suen. 2005. 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)*. 103–110.

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

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

[18] Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '10)*.

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

[20] Hongbo Li, Sihuan Li, Zachary Benavides, Zizhong Chen, and Rajiv Gupta. 2018. COMPI: Concolic Testing for MPI Applications. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '18)*. 865–874.

[21] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2012. 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* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 29, 10 pages.

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

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

[24] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A Dynamic CUDA Race Detector. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. 390–403.

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

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

[27] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 1998. Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM Transactions on Programming Languages and Systems* 20, 1 (1998), 1–50.

[28] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 1999. Parametric Shape Analysis via 3-Valued Logic. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. 105–118.

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

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

[31] Koushik Sen, Darko Marinov, and Gul Agha. 2005. 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)*. 263–272.

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

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

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

[35] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. 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)*. 135–146.

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