Accelerating Schedule Space Exploration of Multi-threaded Programs with GPUs

Prakhar Banga*, Atul Pai†, Subhajit Roy‡ and Mainak Chaudhuri§
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, India.
*prakhar345banga@gmail.com, †atulramanathpai@ymail.com, ‡subhajit@iitk.ac.in, §mainakc@iitk.ac.in

***Abstract*— Given an input that can trigger a concurrency bug, only a subset of possible thread schedules satisfying certain constraints can actually cause such a bug to manifest. Recent proposals on controlled randomization of thread schedules with concrete guarantees on bug detection probabilities have opened promising avenues in this direction. However, to boost the bug detection probability, these techniques typically require a significant number of schedules to be explored. As a result, it is, in general, beneficial to accelerate the schedule space exploration of the multi-threaded programs.**

In this paper, we introduce Simultaneous Interleaving Exploration with Controlled Sequencing (SINECOSEQ), a generic framework that leverages the high-performance graphics processing units (GPUs) to significantly accelerate schedule space navigation of general-purpose multi-threaded programs. The SINE framework accepts POSIX compliant multi-threaded programs, instruments them to intercept all shared memory accesses, and automatically generates CUDA (Compute Unified Device Architecture) compliant code that navigates the schedule space of the input multi-threaded program on an NVIDIA GPU. Each GPU thread typically explores one schedule of the input program. The COSEQ framework decides how the schedule space is navigated by architecting the schedules on the fly. While it is straightforward to construct and navigate a different schedule on each GPU thread, the performance of the resulting technique can be very poor due to disparate pieces of codes executed by each GPU thread leading to full control divergence. In this paper, we demonstrate one application of SINECOSEQ by proposing a new GPU-friendly scheduler for accelerated concurrency testing (ACT), which is inspired by the recently proposed randomized scheduler of probabilistic concurrency testing (PCT). Compared to the state-of-the-art parallel PCT (PPCT) implementation on a twelve-core CPU, our proposal implemented on an NVIDIA Kepler K20c GPU card significantly speeds up schedule space exploration for eight multi-threaded applications and kernels drawn from the Phoenix and the PARSEC suites.

I. Introduction

In this era of multi-cores, efficiently testing applications for concurrency bugs has gained enormous importance. Concurrency bugs are elusive: these bugs require a specific sequence of decisions from a scheduler to manifest. As schedulers are non-deterministic, both detecting and reproducing these faults are hard. Traditionally, concurrent programs are heavily *stress-tested* — the application is run many times hoping for the right set of decisions from the scheduler that unearth latent bugs.

Recent proposals on controlled randomization of thread schedules with concrete guarantees on bug detection probabilities have opened promising avenues in this direction. An interesting approach in this direction attempts to classify concurrency bugs using the notion of *bug depth* [5]: the "depth" of a concurrency bug is the least number of constraints on the ordering of thread instructions that is required to manifest the concurrency bug in a thread-schedule. The PCT algorithm, proposed in the same article, uses a priority-based scheduler that assigns priorities to each thread in a manner that each run of the program tests one such possible ordering at a user-provided bug-depth. Hence, overall, the algorithm guarantees a *lower-bound* on the probability of catching bugs at a certain bug-depth. However, due to complete serialization of the multi-threaded execution, the PCT runs are slow. The parallel PCT algorithm (PPCT) [11] restricts serialized execution to a small number of threads (bounded by the bug depth) while allowing the other threads to execute in parallel. The PPCT algorithm achieves significant speedup (5× on an eight-core machine) over the PCT algorithm.

However, the lower-bound guarantee on catching a concurrency bug is small (dictated by the number of dynamic instructions), while the search space increases exponentially with the bug depth. Hence, an application still needs to be executed a large number of times to gain enough *coverage* of feasible thread schedules to elicit reasonable confidence.

Modern graphics processing units (GPUs) support massive parallelism, thus providing an exciting opportunity for concurrency testing. However, the GPU hardware, being tuned for the graphics pipelines, needs carefully designed algorithms to extract acceptable parallelism. In NVIDIA GPUs, the computation is divided into blocks of threads, where each block executes threads in warps of 32 threads. The blocks are scheduled on multiple streaming multiprocessors (SM's) by the GPU scheduler. Each warp is executed by a single instruction multiple thread (SIMT)-style vector unit and, thus, the threads in the warp must be data parallel; any control-flow divergence among these threads results in serialization, thus hurting performance. Efficient utilization of the memory hierarchy (consisting of the large but slow global memory, small shared memory, and fast but minuscule local memory) also turns out to be a challenge for GPU algorithms.

A naive port of multi-threaded concurrency testing algorithms (like PPCT) to GPUs is not a viable approach to achieving high performance, as such an implementation would face significant control divergence. The PPCT algorithm attempts to run most of the program threads in parallel (while serializing a small number

* Now with Rocketfuel Inc. † Now with ServiceNow

978-1-5090-2791-0/16/$31.00 ©2016 IEEE

of them); on a GPU, such an implementation will see almost no data-parallelism and high divergence (as the different program threads would execute completely different instructions) leading to an abysmally poor performance.

In this paper, we introduce Simultaneous Interleaving Exploration with Controlled Sequencing (SINECOSEQ), a generic framework that leverages the high-performance GPUs to significantly accelerate schedule space navigation of general-purpose multi-threaded programs (Section III). The SINE framework accepts POSIX compliant multi-threaded programs, instruments them to intercept all shared memory accesses, and automatically generates CUDA (Compute Unified Device Architecture) compliant code that navigates the schedule space of the input multi-threaded program on an NVIDIA GPU. The COSEQ framework constructs an efficient orchestration by controlling and navigating the explored interleavings on the fly with the goal of maximizing exploration coverage while minimizing control-flow divergence. Overall, SINECOSEQ provides an extensible scheduler that allows implementation of different schedule-space exploration strategies while drawing support from the framework for their efficient GPU execution.

Our framework (SINECOSEQ) is capable of model-checking unmodified POSIX-complaint multi-threaded applications on a GPU. SINECOSEQ includes support for identifying shared locations, provides a compiler (C-to-COSEQ) to transform the source program for interleaved execution by "flattening" it, instruments preemption points to allow the scheduler to take control, models POSIX thread calls on the GPU and provides stubs for common libc methods that enable running thousands of interleaved executions of realistic programs in parallel on the GPU (Section III). Our framework is generic; an algorithm developer interested in experimenting with new scheduling algorithms for discovering bugs would only need to implement a new thread orchestration strategy within the scheduler, while inheriting the rest of the functionality for free.

In this paper, we demonstrate the utility of our framework by designing a GPU-friendly randomized scheduler for accelerated concurrency testing (ACT) and implementing it on our SINECOSEQ framework. ACT is inspired by the PCT scheduler: ACT carries out PCT across the warps (using unbiased sampling of preemption points across warps) but exploits a biased sampling technique to gain in performance within a warp. In particular, we show how our *Delayed Divergence* algorithm carefully constructs and navigates the schedules so that control divergence among the GPU threads is contained (Section IV). Compared to the state-of-the-art parallel PCT (PPCT) implementation on a twelve-core CPU, our proposal implemented on an NVIDIA Kepler K20c GPU card significantly speeds up schedule space exploration for eight multi-threaded applications and kernels drawn from the Phoenix and the PARSEC suites (Section V). Figure 1 provides a high-level summary of our results by enumerating the best speedup achieved by our implementation at each bug depth for all the applications. The best speedup numbers vary from 1.2× to 365.3×. These impressive improvements over the state-of-the-art can significantly accelerate concurrency testing.
The contributions of this paper are as follows:

- Our primary contribution is in the design of the SINECOSEQ framework that allows accelerated schedule space exploration on GPUs.
  - We introduce the notion of Controlled Sequencing (COSEQ) and demonstrate its effectiveness at exploring thread schedules on a GPU.
  - We discuss the Simultaneous Interleaving Exploration (SINE) scheme that involves executing the COSEQ on multiple GPU threads, effectively exploring multiple interleavings concurrently.
- We demonstrate the capabilities of our SINECOSEQ framework by building a GPU-friendly scheduler, ACT, on the SINECOSEQ framework. Our proposal speeds up schedule space exploration by 1.2× to 365.3× over the state-of-the-art multi-threaded PPCT implementation for eight multi-threaded applications.

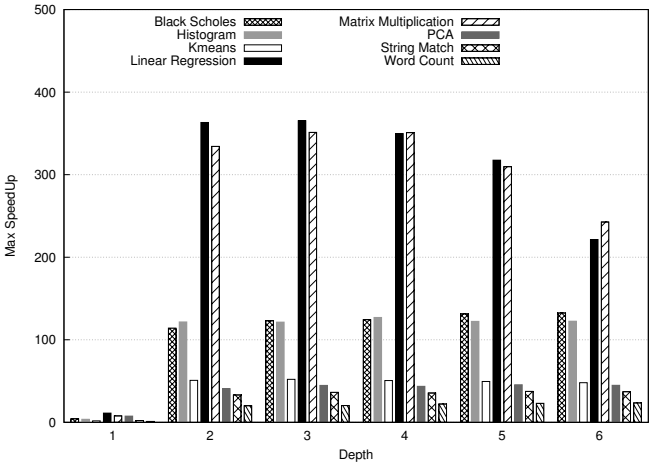


Fig. 1: Speedup achieved at various bug depths

II. Background

A. Probabilistic Concurrency Testing

The major challenge in testing multi-threaded programs is the non-determinism in their thread schedules. As the space of all possible thread schedules is exponential in the number of dynamic instructions executed by the program, an exhaustive coverage of all such schedules is not practical.

Burckhardt et al. [5] propose to classify a bug based on the least number of constraints on the ordering of thread instructions that are required to manifest the concurrency bug in a thread schedule; they refer to this as the *bug depth*. For example, if a thread T_1 acquires two locks, a lock `x` followed by another lock `y`, while another thread T_2 acquires the same locks but in the different order, we would need *two* ordering constraints to elicit a deadlock: T_1's `acquire(x)` must happen before T_2's `acquire(x)`, and T_1's `acquire(y)` must happen after T_2's `acquire(y)`. Probabilistic Concurrency Testing (PCT) [5] proposes a scheduling policy that provides a probabilistic lower bound of $\frac{1}{tn^{d-1}}$ for detecting a concurrency bug at a given depth d for a program with t threads that executes a maximum of n instructions.

To explore bugs at a depth d, the PCT scheduler starts off by assigning distinct random priorities between d and $(t + d - 1)$ to each of the t threads in a given program. Assuming that an execution of the program retires n dynamic instructions, it picks $(d - 1)$ out of these n instructions as the *priority change points* uniformly at random. Till a priority change point is encountered (the scheduler maintains a counter to count the number of dynamic instructions executed), it follows a strict priority based

scheduling scheme, picking the enabled thread with the highest priority every time. At the i^{th} priority change point, the priority of the running thread is lowered to $(d - i)$.

The intuition behind PCT is that at the i^{th} priority change point, the scheduler, by lowering the priority of the current thread to a priority of $d - i$, establishes an ordering constraint between the current dynamic instruction and the instruction at the $(i-1)^{th}$ priority change point (as the priority of the current thread is lower than the thread that was running when the $(i - 1)^{th}$ priority change point was encountered). However, this ordering can get violated in priority inversion scenarios; it is a demon that the algorithm lives with.

The parallel PCT (PPCT) [11] algorithm attempts to accelerate the PCT algorithm on multi-core CPU platforms. This Parallel PCT (PPCT) scheduler makes use of the important observation that it is not necessary to initially assign distinct priorities to the threads as the ordering constraints are only established once the thread priorities are lowered. The PPCT scheduler initially assigns a high priority to each thread and allows them to run in parallel. At a priority change point i, it lowers the priority of one of the executing threads to $d-i$, thereby establishing an ordering constraint by sequentializing the execution. As the high priority threads run in parallel while serial execution is only limited to at most d threads, PPCT offers substantial speedup (about $5\times$ on eight-core machines) while providing the same probabilistic assurance of hitting concurrency bugs as the PCT scheduler.

As both the PCT and the PPCT schedulers explore only a single schedule in a run, to ensure higher probabilities of hitting concurrency bugs, one has to repeatedly explore the schedule space with these schedulers. In this paper, we propose strategies for simultaneously exploring multiple runs on the GPU in a symbiotic manner. Also, while the parallelism in the PPCT scheduler is limited by the number of program threads, the parallelism of our framework is independent of the number of program threads and is only bounded by the resource limits of the GPU hardware.

The NeedlePoint [11] framework facilitates systematic schedule space exploration of multi-threaded programs; it provides implementations of the PCT and the PPCT algorithms (besides others) on traditional CPUs.

B. Overview of GPU Architecture

The GPU architecture that we use in this study contains an array of streaming multiprocessors (SM's or SMX's) with each SM containing an array of execution units. The CUDA (Compute Unified Device Architecture) parallel programming model divides the threads of a program into a number of thread blocks and a thread block is scheduled on an SM. The threads in a thread block are further grouped into warps. The GPU architecture we use in this study has a warp size of 32 threads. All threads in a warp are scheduled together and execute the same instructions in lock-step, except when the threads in a warp diverge on a branch condition. The lock-step execution of the threads in a warp leads to the single instruction multiple thread (SIMT) execution model, which is central to high-throughput vector computation; the same instruction of the threads in a warp operate on a vector data in parallel. If the threads in a warp diverge due to a branch condition, the divergent subsets of the threads within the warp execute sequentially severely hurting the performance. One of the primary goals of a good GPU algorithm is to minimize control divergence within a warp.

III. SINECOSEQ Framework

We present the details of the SINECOSEQ framework in this section. We begin by introducing the concept of controlled sequencing (COSEQ) of concurrent programs (Section III-A). Next, we discuss how the SINE component simultaneously explores different interleavings of a concurrent program on the GPU threads (Section III-B). Finally, we briefly discuss the details of our implementation of SINECOSEQ (Section III-C).

A. Controlled Sequencing (COSEQ)

A concurrent program P can exhibit one of many possible thread interleavings during an execution. We denote the possible interleavings by I_1, I_2, I_3, ..., I_n. Let the execution of P with the interleaving I_k be represented as $\langle P, I_k \rangle$. Consider a sequential program P'_k that is designed to simulate $\langle P, I_k \rangle$ for some $k \in [1, n]$. The sequential program P'_k will be referred to as one of the n possible controlled sequencing of P, or COSEQ(P, k). Intuitively, given a schedule of a concurrent program P, there exists a COSEQ that is equivalent in execution to the given schedule, since each execution $\langle P, I_k \rangle$ has a one-to-one correspondence with the sequence of instructions of the COSEQ. The COSEQ is constructed by stitching together the set of executed instructions in the order they are scheduled in the given interleaving.

We refer to a program thread as *sthread* (sequential thread) to distinguish it from the thread that executes the COSEQ corresponding to a given schedule of the concurrent program P. Each sthread executes the same sequence of dynamic instructions as its corresponding thread in the original program running under the schedule simulated by the COSEQ. However, instead of executing concurrently, they execute sequentially in an interleaved manner on the COSEQ thread.

The COSEQ framework includes a scheduler which is invoked at each identified scheduling point in P. While any program point can potentially be a scheduling point, for studies on concurrency testing, we mark the shared memory access points as the scheduling points. On encountering a scheduling point in the currently running sthread, the COSEQ thread transfers control to the scheduler. The scheduler's job is to pick an sthread among the active sthreads to run on the COSEQ thread. By designing different types of schedulers, the COSEQ can explore different families of schedules. Since a COSEQ thread interleaves the execution of the sthreads, it must keep track of individual sthread states and any global states of the original program. This is the responsibility of the sthread manager. Each COSEQ thread maintains a data region with space for holding the global variables of the original program. All global variable accesses from sthreads are converted to accesses to the corresponding segment in the global data region. Additionally, each COSEQ thread maintains the stack region for each sthread. All private variables of an sthread are stored in this region. The COSEQ framework also includes a *mutex manager* that maintains the states of the mutexes in the original concurrent program and the set of sthreads blocked on each mutex (implemented in the form of a mutex hash table).

B. Simultaneous Interleaving Exploration (SINE)

The SINE technique uses the COSEQ framework to perform simultaneous exploration of different interleavings of a concurrent program. Given a concurrent program, we generate its COSEQ

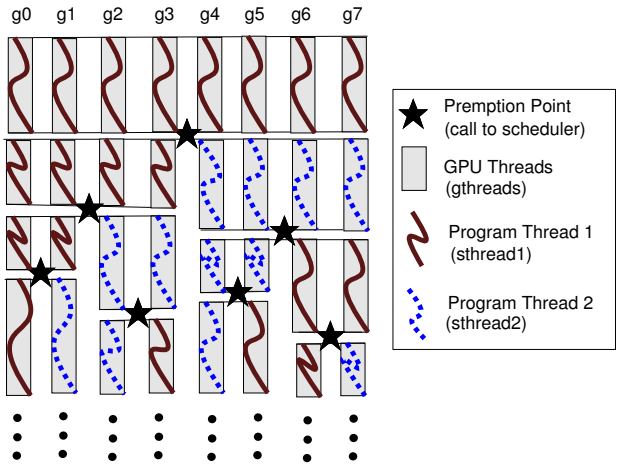


Fig. 2: Prefixes of explored interleavings of a sample SINE with eight GPU threads (gray boxes corresponding to `g0` to `g7`) and two program threads of the concurrent program, `sthread1` and `sthread2`, in the COSEQ are executed by the `gthreads`. At each preemption point, the GPU threads split in halves to pick a different `sthread`, thereby exploring different interleavings. After hitting three preemptions, all `gthreads` are exploring a distinct COSEQ.

by instrumenting all potential shared memory accesses (the schedule points). Then, we emit a GPU kernel to execute a different COSEQ on each GPU thread (gthread). Hence, each gthread executes the sthreads in an interleaved fashion, transferring control to the scheduler on encountering a schedule point. Ideally, the scheduler's job is to pick sthreads in such a manner that the COSEQ executing on each gthread explores a different interleaving, thereby offering the maximum possible coverage in one run of the GPU kernel. Figure 2 shows the interleaving explored by a sample SINE execution with eight GPU threads (`gthreads`) and a concurrent program with two program threads (`sthreads`).

The scheduler's job is complicated as the GPU performance heavily depends on the amount of control divergence experienced by a warp. The gthreads in a warp should ideally execute in lock-step conforming to the single-instruction-multiple-thread (SIMT) execution model, which is central to the efficiency of the vector operations in a GPU. Control divergence refers to the situation where a subset of threads in a warp executes instructions that are different from the remaining subset. The lock-step SIMT execution model can get significantly disturbed if the gthreads that form a warp explore completely disparate schedules leading to full control divergence. Since the amount of divergence is directly proportional to the number of different interleavings explored by the threads within a warp, we face two contradictory requirements when designing an efficient SINECOSEQ framework for concurrency testing. First, different gthreads must explore a different interleaving. Second, the amount of control divergence within a warp must be minimized, which can be achieved only when all threads within a warp explore the same interleaving. Since these contradictory goals cannot be achieved simultaneously, our proposal takes a middle ground where full divergence is *delayed* as much as possible while achieving maximum possible schedule space coverage. The central idea is to guarantee that the COSEQ threads within a warp share a significant common prefix while gradually (as opposed to at the same time) diverging to explore different interleavings.

To begin with, the GPU threads in a warp start off by exploring the same interleaving while different warps are assigned different interleavings to explore. Within a warp, at certain scheduling points, the threads are made to diverge to explore different interleavings as shown in Figure 2. These scheduling points will be referred to as the splitting points. At every splitting point for a group of GPU threads, the group is halved; for one half, the scheduler picks sthread X (say) for execution on the COSEQ, while for the other half, it picks sthread Y. The splitting points need to be picked carefully as delaying the splitting points too far can prevent full divergence within a warp; while this is good for performance, this can lead to a suboptimal coverage of the schedule space. We employ a statistical technique for choosing the splitting points.

Since all threads in a warp execute a common prefix of instructions before reaching the first divergence point, this may seem wastage of resources. A possible alternative could be to run one thread through the common prefix and then gradually "fork" new threads as the splitting points are encountered. On a GPU, there are multiple objections to this scheme. First, as a GPU does not support a "fork" call, we would need to stop execution of the complete warp to emulate what a fork call would have done i.e., copy the complete state of the parent thread to the child; this would be prohibitively expensive. Second, as each thread in the warp is executing a completely different set of instructions, there would be complete divergence. In terms of the GPU hardware, it is equivalent to running 32 different warps, with one active thread in each warp. To avoid these problems, it is necessary to execute all threads in a warp through a common prefix so that each thread builds up its own context. As a result, we incur zero performance penalty at a splitting point. This is faster than executing a costly (and inherently sequential) context copy at each splitting point. Even on traditional multiprocessors with OS support for fork calls, most model checkers commence the exploration of each interleaving from scratch; each interleaving, thus, builds up its own context, thereby duplicating executions of prior explorations.

While our SINECOSEQ framework provides the complete machinery around it, the orchestration of the interleavings to be explored by each gthread is central to a good concurrent model checker that must balance between providing a lower-bound probability of eliciting a buggy schedule (if possible) and providing good performance (in terms of coverage attained per unit time). Our framework allows for writing such "schedulers" for experimenting with different algorithms. In Section IV, we propose a new GPU-friendly scheduling algorithm, the ACT scheduler, to demonstrate a concrete application of the SINECOSEQ framework for conducting a directed exploration of the schedule space with the aim of identifying concurrency bugs.

C. Implementation

Figure 3 shows the architecture of our tool. Our tool accepts a multi-threaded C program (we currently support only the POSIX library). The important components of the tool-chain are:

- **Identify shared accesses**: We use the Inspect tool [18] to identify the shared memory accesses. Inspect uses escape analysis to statically over-approximate the shared memory accesses.
- **C-to-COSEQ transformation**: Our C-to-COSEQ transformer instruments the shared memory accesses to record

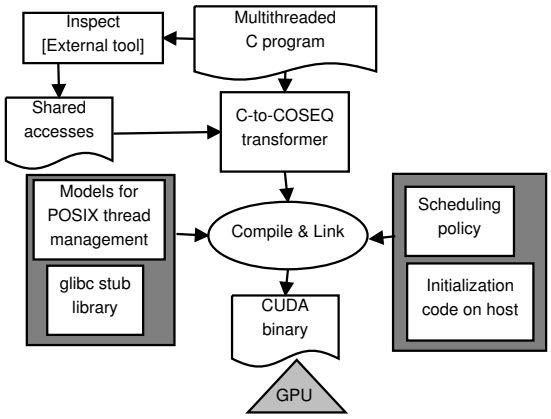


Fig. 3: The SINECOSEQ tool chain

the scheduling or preemption points. Further, this pass prepares the program for controlled sequencing on a GPU by flattening the function hierarchy (similar to what exists in assembly code) and inserting conditional jumps to enable control transfer across preemption points (via the scheduler); when the source and target jumps correspond to locations in two different threads, then this control transfer amounts to establishing a preemption. Thus, selecting a sequence of (valid) jump targets would enable a sequentialized run of a multi-threaded execution of the program corresponding to one particular thread interleaving. The scheduler controls these jump targets, thereby dictating the sequence of preemptions. Our C-to-COSEQ transformer is a source-to-source transformation pass implemented on the CIL compiler framework [1]. It handles complex programming constructs like recursion and dynamic memory allocation.

- **Enable GPU execution**: Execution of this transformed program on the GPU is enabled by our models for POSIX thread creation and synchronization primitives. We model many of the POSIX library calls as CUDA `__device__` methods. These include thread management calls such as `pthread_create()` and `pthread_join()`, and synchronization primitives such as `pthread_mutex_lock()` and `pthread_mutex_unlock()`. We also provide stub methods for many useful *glibc* functions, again implemented as CUDA `__device__` methods[1]. The GPU main memory is used to store the state of the programs. As the CUDA code executing on a GPU cannot handle any system calls, we approximate the behavior of some of the important calls; one noteworthy mention are the file system calls: any input file is loaded into memory by the CPU-side code and transferred to the GPU memory. Our GPU stubs for the file system calls replace the file system operations by accesses to the GPU memory. Note that all this is done under-the-hood by the stub library (implemented by us), thereby not requiring any modifications of the source code. Our current implementation contains enough support to allow for execution of real-life programs (as demonstrated by our benchmark applications).

[1] We also wrote the CPU-side methods of these calls and use them with Needlepoint for a fair comparison.

- **Scheduler**: The scheduler is the central component of the framework. The scheduler operates by constructing a valid sequentialization of the COSEQ-transformed code. Our framework allows for the implemention of various scheduling policies without any change to any of the other components in the framework. For instance, our implementation of the ACT scheduler (discussed in the next section) controls the sequencing with an objective to achieve an efficient exploration of the interleavings.
  Designing a scheduler for the SINECOSEQ framework not only entails sequencing the program threads in a manner that the thread scheduling policy is obeyed, but it also requires arbitration of the valid sequences among the GPU threads so that the schedule space exploration is performed efficiently on the GPU hardware. We discuss how our implementation of the ACT scheduler allows a fast exploration of different schedules in the next section.

The above components are compiled and linked using the NVIDIA CUDA compiler (nvcc) to produce a CUDA-compliant binary.

IV. Accelerated Concurrency Testing (ACT)

In this section, we discuss the design of our Accelerated Concurrency Testing (ACT) scheduler implemented within SINECOSEQ. In particular, we discuss the *Delayed Divergence (DD)* algorithm that allows us to achieve efficient performance on the GPU architecture.

A. Design Alternatives

We considered multiple design alternatives while attempting to implement our scheduler on GPU. Some of the prominent ones are discussed below.

- **PPCT on GPU**: The first option that one may have is to consider implementing the PPCT algorithm directly on GPU. This option poses a difficulty: the PPCT scheduler allows multiple program threads to execute in parallel. Due to the SIMT constraints on GPU, such a control-parallel design would manifest full divergence, causing complete serialization among the GPU threads in a warp. The second objection to this design is that PPCT's parallelism is limited by the number of program threads spawned by the program under test. Hence, such a design would not be able to exploit the enormous parallelism offered by GPUs.
- **PCT runs independently on each GPU thread**: This design addresses the second objection on directly running PPCT on GPU — that of exploiting the massive parallelism offered by GPUs. By executing as many PCT runs as allowed by the hardware limits of the GPU, the GPU hardware can be exploited to the fullest. However, such a design will still have a large control divergence, causing serialization among the threads in a warp.
- **Delayed Divergence**: Our *Delayed Divergence* algorithm attempts to delay the warp divergence as much as possible. We discuss this algorithm in the rest of this section.

B. The ACT Scheduler

The goal of the *Delayed Divergence* algorithm is to delay control divergence within a warp as much as possible while attempting to cover as many schedule interleavings as the GPU threads employed. Using the Delayed Divergence algorithm, we propose

the Accelerated Concurrency Testing (ACT) scheduler that attempts to minimize control divergence within a warp during the initial phase of the parallel exploration, while full divergence is eventually, and inevitably, manifested in the later stages.

The ACT scheduling algorithm operates as follows:

- As control divergence only manifests inside a warp, each group of GPU threads (gthreads) in a warp is assigned the task of exploring schedules corresponding to:
  - an initial assignment of priorities for program threads (sthreads): each gthread within a group share the initial assignment of priorities for sthreads;
  - a selection of *priority change points* (spoints): at a bug depth of d, the PCT scheduler selects $(d-1)$ spoints; in contrast, the ACT scheduler selects $(d-1)$ *tuples* of spoints for exploring interleavings executed by the gthreads in a warp. The first $(d-2)$ tuples are pairs of spoints, while the last tuple is of length $\frac{warpsize}{2^{d-2}}$.
- Initially, the gthreads are grouped by the warp they belong to. As the exploration progresses, the gthread groups are split into smaller groups: each gthread in a group would have explored exactly the same interleaving before hitting the next split.
- After encountering the $(k-1)^{th}$ spoint, a group fetches the k^{th} spoint-tuple. For an spoint-tuple of size n, the group further splits its members into n equal groups; the i^{th} subgroup selects the i^{th} component in the tuple as its next spoint.
- After $d-2$ such splits, each group will contain a sole gthread; thus, each gthread would have selected a unique sequence of spoints among the initial group of the gthreads in the warp. Also, as each warp is initialized with a different random assignment of priorities for the sthreads and a different assignment of the spoint-tuples, each thread in the GPU would have exercised a unique interleaving.

Note that the ACT scheduler attempts to delay full divergence all the way to the last spoint-tuple. At any point in the algorithm, the divergence is bounded by the number of groups formed so far, as the gthreads within a group have no divergence. We discuss the detailed algorithm below.

1) gthread Group Management

The ACT scheduler chooses a 'master' for each warp. As each group is initially formed from a GPU warp, the master is chosen as the first gthread of each warp (Algorithm 3, lines 4-5). The master generates random priorities for the sthreads and random spoint-tuples for its group (lines 6-9). All GPU threads in a warp use the same assignment of random priorities for their sthreads (see Algorithm 1).

2) Generation of the spoint-tuples

While attempting to hit concurrency bugs at depth d, PCT needs $(d-1)$ spoints in each thread schedule. The ACT scheduler needs $d-1$ spoint *tuples* for each warp so as to efficiently assign a unique sequence of spoints to each of its member gthread.

For spoint-tuple generation (Algorithm 2), a warp requires exactly $2(d-2)+(\frac{warpsize}{2^{d-2}})$ spoints for generating the $d-1$ spoint-tuples: two spoints for the spoint-pairs corresponding to spoint indices till $d-1$ and $\frac{warpsize}{2^{d-2}}$ spoints for the final spoint-tuple.

```
1 var Π_ρ ← RandomPermute(t) ;                    // t is #sthreads
2 for i ← 1 to t do
3 |   ρ_#[i] ← d + Π_ρ(i) − 1
4 end
```

Algorithm 1: A warp's master storing random priorities in $\rho_{\#}$.

```
1  numSPoints ← 2 ∗ (d − 2) + (warpsize/2^{d−2})
2  var Π_k ← distinctRandomPoints({1 . . . n}, numSPoints) ;
   // n is estimated number of preemption points
3  sort-ascending(Π_k)
4  for i ← 1 to d − 1 do
5  |   if i < d − 1 then
6  |   |   k_#[i] ← ⟨getNextInSeq(Π_k), getNextInSeq(Π_k)⟩
7  |   else
8  |   |   k_#[i] ← ⟨getNextInSeq(Π_k), . . . ⟩
   |   |   ;          // the above tuple will have size warpsize/2^{d−2}
9  |   end
10 end
```

Algorithm 2: A warp's master storing random spoints in $k_{\#}$.

Given an estimated length n for an interleaving sequence (in terms of its preemption points), the master generates `numSPoints` distinct points in the range $[1, n]$, in the increasing order. It, then, creates respective spoint tuples by extracting points from this sorted random list. We estimate the length of interleaving sequences by counting the number of spoints hit in a few random executions on the CPU (similar to [5]).

3) Parallel Exploration of Thread-Schedules

After the master assigns the sthread priorities and spoint tuples for its group (as discussed above), each gthread in the warp initializes the groupsize as the warpsize, computes its *id* within the group (which is currently the whole warp), and the *splitId* which helps compute whether this respective gthread would belong to the first or the second half if this group was split into two equal halves (Algorithm 3, lines 12-14).

The scheduler makes progress by picking the highest priority enabled program thread t_ρ and extending the (currently empty) execution sequence S by executing a step on the thread t_ρ. All gthreads within a warp start as a single group exploring the same thread schedule before they split into equal halves, each half picking a different first spoint to form two groups. Each of these two groups continues exploring a unique thread-schedule before each further splits into equal halves, each half picking a different second spoint to form four groups and so on, as shown in Algorithm 3 (lines 21-25).

Till the first $(d-2)$ spoints, every group picks its i^{th} spoint from the i^{th} spoint pair, each half of the split selecting the respective component from the tuple. After $(d-2)$ splits, the size of each group is reduced to $\frac{warpsize}{2^{d-1}}$. Once the $d-1^{th}$ spoint needs to be chosen (Algorithm 3, lines 28-30), each gthread picks the $(d-1)^{th}$ spoint from this array based on its own index within the group. Figures 4a, 4b, 4c, and 4d illustrate this for a warp of 32 GPU threads exploring the schedule space at concurrency bug depths of 1, 2, 3, and 6, respectively.

After $(d-1)$ spoints, each gthread continues exploring its unique thread-schedule. Note that each gthread within the warp

```
1  var blockid ← cuda-blockid()
2  var threadid ← cuda-threadid()
3  var blocksize ← cuda-blocksize()
4  var master ← blockid ∗ ⌈blocksize/warpsize⌉ + ⌈threadid/warpsize⌉
5  var me ← master + threadid % warpsize
6  if me = master then
7  |   ρ# ← makepriority() ;                    // warp-shared
8  |   k# ← makeschedule() ;                    // warp-shared
9  end
10 var S ← ϵ ;                                  // begin null thread-schedule
11 var di ← 1 ;                                 // index of next switch-point
12 var groupsize ← warpsize
13 var id ← threadid % groupsize ;              // index in current group
14 var splitId ← ⌈2∗id/groupsize⌉ ;             // first/second half on next split
15 while enabledthreads(S) ≠ ∅ do
16 |   tρ ← enabled and ρ#[tρ] is maximum
17 |   S ← S + ||tρ|| ;                         // extend S by advancing thread tρ
18 |   if di < d then
19 |   |   if di < d − 1 then
   |   |   |   ; // spoint 2-tuple
20 |   |   |   if length(S) == k#[di][splitId] then
21 |   |   |   |   ρ#[tρ] ← d − di
22 |   |   |   |   di++
23 |   |   |   |   groupsize ← (groupsize/2)
24 |   |   |   |   id ← id % groupsize
25 |   |   |   |   splitId ← ⌈2∗id/groupsize⌉
26 |   |   |   end
27 |   |   else
   |   |   |   ; // spoint (warpsize/2^(d−2))-tuple
28 |   |   |   if length(S) == k#[di][id] then
29 |   |   |   |   ρ#[tρ] ← d − di
30 |   |   |   |   di++
31 |   |   |   end
32 |   |   end
33 |   end
34 end
```

Algorithm 3: The ACT scheduler used by gthreads within a warp.

is assigned its own unique set of $(d-1)$ spoints, forcing the thread-schedule explored by each gthread to be unique. At the same time, as the gthreads within a warp share the spoints, all the preemption points (marked as $\star$ in Figure 2) would occur in a synchronized manner (see Figure 4d).

C. Discussion

1) ACT versus PCT

ACT carries out PCT across the warps (using unbiased sampling across warps) but exploits a biased sampling technique to gain performance within a warp (at the cost of a lower worst-case probability of catching a bug). The overall effectiveness of concurrency testing depends on the product of two parameters, namely, the expected number of schedule explorations needed to uncover a bug and the time taken by each of these schedule explorations (or coverage achieved per unit time). The first term depends on the probability of uncovering a bug. A biased sampling within a warp trades the probability of finding a bug for performance. Overall, we show in Figure 6 that compared to an unbiased sampling technique that offers the same probabilistic guarantee as PCT across as well as within warps, our delayed divergence scheme offers better schedule exploration performance. Moreover, (as also mentioned in [5]), often a bug is triggered by multiple schedules, which makes it important to be able to explore a large number of schedules quickly.

2) Addressing Control Divergence

Before passing the first spoint, all the gthreads in the warp perform the same operation. Control divergence is triggered only after the first spoint but it is controlled carefully. Full control divergence is established only after all the GPU threads in a warp pass their last spoint. This method of controlled and delayed divergence achieves good performance on GPUs.

3) Exploring deeper bug depths

As the current GPU hardware supports a warp size of 32 gthreads, which we use as our initial groups, the above scheme is adept at exploring bug depths up to six (as a bug depth of six requires five spoints). For exploring deeper bug depths, we need to form our initial groups consisting of gthreads from multiple warps. For instance, grouping two warps will enable an exploration at bug depth of seven. However, note that the performance at bug depth of seven will be similar to that at bug depth of six, as divergence only manifests within a warp (i.e., the first split will cause no divergence as it would only detach the two warps).

V. Experiments and Results

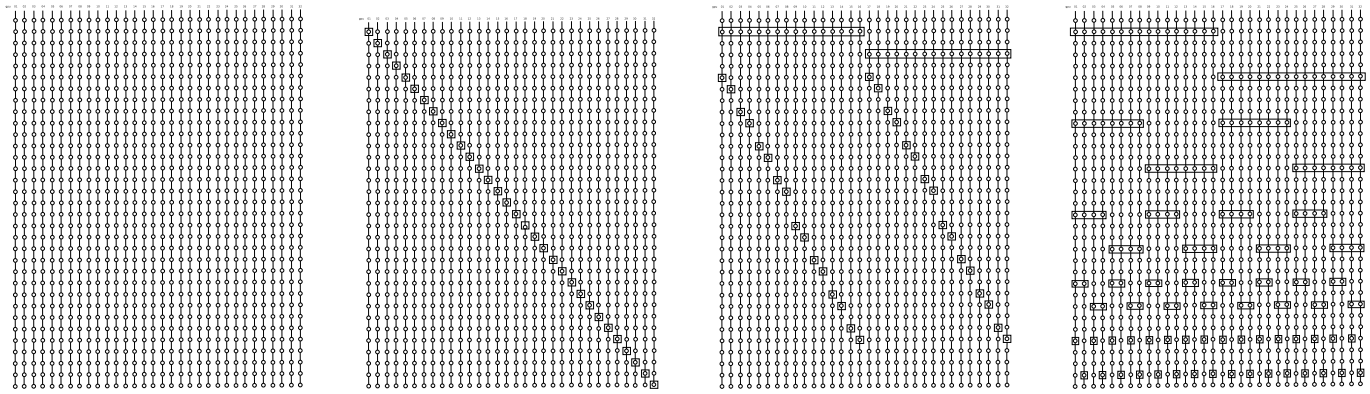
This section presents the performance comparison between NeedlePoint with PPCT and SINECOSEQ with ACT. The former is a multi-threaded schedule space exploration tool designed for multi-core CPU platforms, while the latter is our proposal running on the GPU hardware.

A. Benchmark Applications and Experimental Setup

We selected seven concurrent programs (Histogram, K-Means, Linear Regression, Matrix Multiplication, PCA, String Match, Word Count) from the Phoenix benchmarks [15] and one (Black-Scholes) from the PARSEC suite [3]. These concurrent programs use the POSIX thread library for multi-threading. Each benchmark application creates fifteen POSIX threads. We do not use any buggy multi-threaded program in our evaluation because the primary focus of this study is to accelerate the schedule space exploration of multi-threaded programs as opposed to uncovering known or unknown bugs.

We evaluate NeedlePoint PPCT (N-PCT) on a dual CPU Intel hardware platform, where each CPU has six cores and runs at 2 GHz. The machine has 32 GB DDR3 memory. We evaluate our proposal SINECOSEQ-assisted ACT (S-ACT) on an NVIDIA Kepler K20c GPU card having a GK110 graphics processor clocked at 706 MHz. The graphics processor has 2496 CUDA cores. The card is equipped with 5 GB GDDRx memory. Since this GPU has a warp size of 32, we explore bug depths up to six, which is often sufficient to uncover most of the common concurrency bugs.

We use speedup of S-ACT over N-PCT as the performance metric, defined as the ratio of average run-time per explored schedule of N-PCT to that of S-ACT. We report the achieved speedup for each of the possible six bug depths separately. For each bug depth, the time for N-PCT is calculated by taking the average time of 20000 sample runs. For S-ACT, we fix the



(a) Depth 1: There are no spoints. All GPU threads (gthreads) run the same thread-schedule. There is absolutely no warp-divergence.

(b) Depth 2: There are 32 spoints, one for each GPU thread (gthread). Here control divergence manifests quickly as each gthread passes its spoint.

(c) Depth 3: There are 18 spoints. Control divergence partially manifests as the odd half of 16 GPU threads (gthreads) passes its first spoint. Full control divergence is manifested gradually as each of the 16 GPU threads in the two groups pass their corresponding second spoints.

(d) Depth 6: There are only 10 spoints. Full warp-divergence manifestation is slowest. Full warp-divergence is manifested when each of the two GPU threads in the sixteen groups pass their corresponding fifth spoints.

Fig. 4: The figures illustrate how control divergence manifests in a warp of 32 GPU threads (gthreads) exploring program thread schedules at bug depths 1, 2, 3, and 6. At depth 1, as there are no spoints, each warp executes the same schedule (the hardware is underutilized). At depths of 2, 3 and 6, the warp hardware is fully utilized wherein 32 different program thread schedules are explored by each warp, each GPU thread running a unique interleaving of the program threads.

total number of CUDA threads at 65536 and vary the number of threads per block (the block size) from 32 to 512 while keeping it a power of two. As the block size doubles, the number of blocks is halved to keep the total number of CUDA threads constant. For each bug depth, we show the results for all these configurations. The time of execution for each of these configurations is calculated as the average time of fifteen sample runs.

B. Performance Analysis

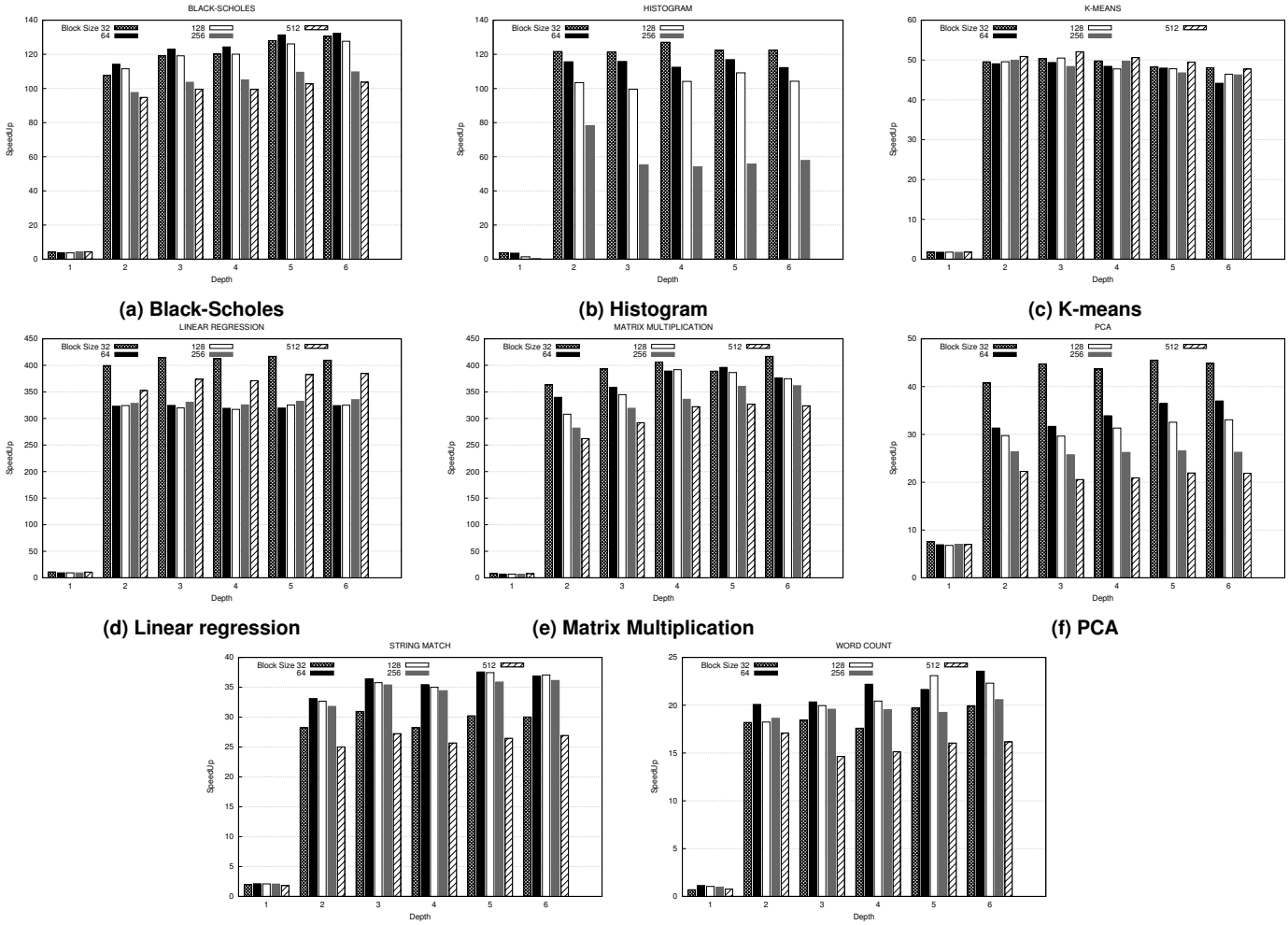
Figure 5 shows the detailed speedup achieved by our proposal on the benchmark applications. For each application, we present six groups of bar, each group corresponding to one bug depth value. For each bug depth, we show how the speedup varies with the CUDA kernel configurations, namely, the block size and the number of blocks (recall that the total number of CUDA threads is kept constant at 65536).

We observe two general trends from these results. First, the speedup achieved by our proposal at bug depth one is small and the speedup significantly increases when bug depth becomes two. Increasing the bug depth further either offers slightly better speedup or leaves the speedup more or less same. For example, at bug depth one, our proposal speeds up the schedule space exploration of Black-Scholes, Histogram, K-means, Linear Regression, Matrix Multiplication, PCA, String Match, and Word Count by up to 4.3×, 3.7×, 1.8×, 11.2×, 7.8×, 7.6×, 2.1×, and 1.2×, respectively. On the other hand, the best speedup figures offered by our proposal on these applications are 132.3× (at bug depth six, block size 64), 127.1× (at bug depth four, block size 32), 52.1× (at bug depth three, block size 512), 365.3× (at bug depth three, block size 32), 351.2× (at bug depth three, block size 32), 45.5× (at bug depth five, block size 32), 37.6× (at bug depth five, block size 64), and 23.6× (at bug depth six, block size 64), respectively. The reason for relatively low speedup at bug depth one is that all the GPU threads within a warp explore the same interleaving, even though different warps explore different interleavings (please refer to Figure 4a). As the bug depth increases, our proposal explores different interleavings in different GPU threads while employing the delayed divergence technique (please refer to Figures 4b, 4c, and 4d). This leads to significantly higher speedup.

The second observation is that, in general, the speedup achieved by our proposal is better when the COSEQ kernel is launched with smaller block sizes. This is primarily because smaller block sizes allow the run-time to have more thread blocks, given that the total number of GPU threads is kept constant. A larger number of thread blocks offer a better load-balance overall, since the execution of the slower and/or heavier blocks can get overlapped with many fast and/or shorter blocks. A larger thread block is more likely to have work-imbalance across its warps, causing the larger blocks to hold up GPU resources longer preventing other thread blocks from getting scheduled. So, it is beneficial to have many small thread blocks when the work distribution across the warps may be uneven or unpredictable, which can very well be the case when carrying out schedule space exploration of an arbitrary multi-threaded program.

In Figure 1, we have already summarized the best speedup achieved by our proposal at each bug depth for all the applications. These best speedup numbers vary from 1.2× to 365.3×. These impressive improvements over the state-of-the-art can significantly accelerate concurrency testing.

Finally, before closing this section, we quantify the importance of the delayed divergence technique. With the same experimental setup, we implement the PCT scheduler on our framework. This algorithm is a completely divergent scheduling algorithm, paying no attention to control divergence and causes full divergence to happen very early in a warp's life. Figure 6 shows the percentage reduction in run-time achieved by our proposal compared to

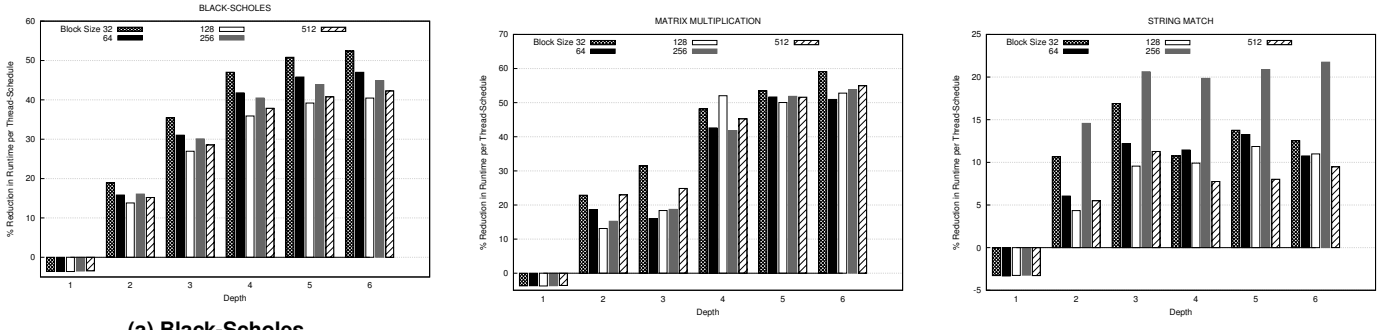


(a) Black-Scholes (b) Histogram (c) K-means

(d) Linear regression (e) Matrix Multiplication (f) PCA

(g) String Match (h) Word Count

Fig. 5: Summary of speedup achieved by our proposal.



(a) Black-Scholes (b) Matrix Multiplication (c) String Match

Fig. 6: Percent reduction in run-time over full divergence.

the PCT scheduling algorithm (on our framework) for three applications. Only at depth one, the delayed divergence scheme suffers a slowdown (by at most 4% for these three applications). This is because the fully divergent scheduler explores 32 different interleavings in a warp leading to better average exploration time per schedule (recall that our scheduler explores only one schedule per warp at bug depth one). As the bug depth increases, our proposal achieves significantly better run-times than the fully divergent scheduler. These results for the rest of the applications are not shown in the interest of space, but they follow similar trends as shown in Figure 6. We would like to emphasize that this should not be mistaken as the overall winnings of our framework as this implementation of PCT also benefits from all the heavy machinery provided by our GPU framework; it only lacks the ability to orchestrate the thread schedules well.

Finally, it should be noted that single-threaded execution of the applications on a GPU thread runs much slower compared to the CPU as the execution units in a GPU are much simpler.

GPUs, however, provide hardware to run thousands of threads in parallel. To quote one of our raw data-points from Black-Scholes, the Needlepoint implementation takes 0.44 seconds (on an average) to complete, while our framework with the ACT scheduler takes about 215 seconds to complete an execution. However, our framework employing 65536 GPU threads (each GPU thread handles one interleaving) achieves an overall speedup of (0.44)/(215/65536) i.e. about 134x per interleaving.

VI. Related Work

An interesting direction of testing for concurrency bugs was in the design of *frameworks* that allowed for writing new scheduling strategies easily [7], [19], [11]. These frameworks identified that writing a concurrency tester has two components: one is a *generic* implementation that encompasses identification of shared accesses and synchronization operations, a generic dispatcher to allow for dispatching threads, and, often, a high-level strategy (like Fuzzing). The second component is a specialized component that implements a specific strategy, and one that is able to ride on top of the generic machinery. Such frameworks have made the design and development of new ideas much easier than developing a complete tool from scratch. For instance, the NeedlePoint framework was reported to be around 6K lines of code, while exploration strategies like preemption bounding [10], AtomFuzzer [16] and PCT [5] were implemented quite easily, each taking less than 300 lines of code. The CalFuzzer framework [7] uses the high-level strategy of fuzzing-based testing for Java applications. It has seen quite impressive tools built on it for detecting deadlocks [8], atomicity violations [12] and races [17]. Maple [19] identifies a set of interleaving idioms to define coverage of multithreaded programs, and uses this coverage metric to guide testing. Maple employs the dynamic instrumentation capabilities of PIN [9] to control the execution of the program. NeedlePoint builds a similar, PIN based framework, providing a stable framework for implementing different analysis. In this work, we also attempt to build SINECOSEQ as a generic framework that would ease and encourage implementing concurrent program testing strategies on GPU architectures. SINECOSEQ provides an end-to-end support for transforming a POSIX-complaint parallel program to enable it to run on a CUDA-enabled GPU, while taking care of static instrumentation of shared-memory, modeling of synchronization operations and an extensive stub library for common glibc functions to enable running of realistic programs on a GPU. To the best of our knowledge, this is the first such framework that enables testing of concurrent programs on GPU.

A popular direction in testing concurrent programs has been in designing policies that would indicate a certain guaranteed *coverage*. Based on a crucial observation that many concurrency bugs are triggered by a small number of context-switches, Qadeer et al. [13] built a tool that would test all behaviors of a multithreaded program with a small number of context-switches. Many more such coverage metrics were invented and successfully employed like bounded preemptions [10], bounded delay [6], bounded exploration at a certain bug-depth [5], coverage of inter-thread dependencies (iRoots) [19], and bounding the number of variables modeled and the number of participating threads [4]. Our SINECOSEQ framework is capable of supporting these scheduling policies with a careful design of a good arbitration policy so that the schedule space exploration is performed efficiently on the GPU hardware.

Rajan et. al [14] use GPUs to test applications by simultaneously executing the program with one test-case per GPU thread. CUDA accelerated LTL Model Checking [2] uses Nvidia GPU cards along with CUDA technology to accelerate automata theoretic LTL model-checking; in particular, the work formalizes a GPU based *fast* algorithm for *accepting cycle detection* in an automata. This work, however, is orthogonal to our proposal in its goal, scope and approach.

VII. Summary

In this paper, we propose a generic framework (SINECOSEQ) for accelerating schedule space exploration of multi-threaded applications on GPUs. Our experiments demonstrate that SINECOSEQ (with our efficient ACT scheduler) can often extract one to two orders of magnitude speedup over the optimized state-of-the-art multi-core CPU implementations.

References

[1] CIL - Infrastructure for C Program Analysis and Transformation, `https://www.cs.berkeley.edu/ necula/cil/`.

[2] Jirí Barnat, Luboš Brim, Milan Ceška, and Tomáš Lamr. CUDA accelerated LTL model checking. In *ICPADS '09*, 2009.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08*, 2008.

[4] Sandeep Bindal, Sorav Bansal, and Akash Lal. Variable and thread bounding for systematic testing of multithreaded programs. In *ISSTA 2013*, 2013.

[5] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS XV*, 2010.

[6] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *POPL '11*, 2011.

[7] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *CAV '09*, 2009.

[8] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09*, 2009.

[9] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, 2005.

[10] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07*, 2007.

[11] Santosh Nagarakatte, Sebastian Burckhardt, Milo M.K. Martin, and Madanlal Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI '12*, 2012.

[12] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08/FSE-16*, 2008.

[13] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS '05*. 2005.

[14] Ajitha Rajan, Subodh Sharma, Peter Schrammel, and Daniel Kroening. Accelerated test execution using gpus. In *ASE '14*, 2014.

[15] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA '07*, 2007.

[16] Koushik Sen. Effective random testing of concurrent programs. In *ASE '07*, 2007.

[17] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI '08*, 2008.

[18] Yu Yang. *Efficient Dynamic Verification of Concurrent Programs*. PhD thesis, University of Utah, 2009.

[19] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *OOPSLA '12*, 2012.