Nekara: Generalized Concurrency Testing

Udit Agarwal[*], Pantazis Deligiannis[+], Cheng Huang[++], Kumseok Jung[**], Akash Lal[+], Immad Naseer[++]
Matthew Parkinson[+], Arun Thangamani[+], Jyothi Vedurada[#], Yunpeng Xiao[++]
[*]*IIIT Delhi*, [+]*Microsoft Research*, [++]*Microsoft*, [**]*University of British Columbia*, [#]*IIT Hyderabad*

***Abstract*—Testing concurrent systems remains an uncomfortable problem for developers. The common industrial practice is to stress-test a system against large workloads, with the hope of triggering enough corner-case interleavings that reveal bugs. However, stress testing is often inefficient and its ability to get coverage of interleavings is unclear. In reaction, the research community has proposed the idea of *systematic testing*, where a tool takes over the scheduling of concurrent actions so that it can perform an algorithmic search over the space of interleavings.**

We present an *experience paper* on the application of systematic testing to several case studies. We separate the algorithmic advancements in prior work (on searching the large space of interleavings) from the engineering of their tools. The latter was unsatisfactory; often the tools were limited to a small domain, hard to maintain, and hard to extend to other domains. We designed Nekara, an open-source cross-platform library for easily building custom systematic testing solutions.

We show that (1) Nekara can effectively encapsulate state-of-the-art exploration algorithms by evaluating on prior benchmarks, and (2) Nekara can be applied to a wide variety of scenarios, including existing open-source systems as well as production distributed services of Microsoft Azure. Nekara was easy to use, improved testing, and found multiple new bugs.

I. Introduction

Exploiting concurrency is fundamental to building scalable systems. It can range from desktop applications with multiple threads that exploit a multi-core CPU or giant distributed systems with multiple processes spanning many VMs. In either case, getting the concurrency right is challenging. The combinatorial explosion of possible interleavings between concurrent actions makes it hard to find bugs, or even reproduce known ones [1], [2]. Prior work argues for *systematic testing*, which hooks on to the concurrency in a program to reliably control the interleaving of concurrent actions, and then orchestrates a search (exhaustive or randomized) within the space of all interleavings. This idea manifests in several tools, such as CHESS [2], [3] and Cuzz [4] for multi-threaded applications, dBug [5], Modist [6] and SAMC [7] for distributed message-passing systems, and many others [8], [9], [10], [11], [12].

This paper summarizes our experience in applying systematic testing in practice. Concurrency comes in many forms; it changes with the programming language (C#, Go, Rust, etc.), programming model (e.g., threads, tasks, actors, async-await, etc.), framework (e.g., Service Fabric [13], libevent [14], Trio [15], etc.) and so on. Our immediate learning in this space was that each of the tools mentioned above only address a specific class of programs, and are difficult or impossible to apply to other classes of programs.

To understand this shortcoming, we can examine the design of a typical solution. It consists of three parts. The first is algorithmic search: a collection of heuristics that guide the search to interesting parts where bugs can be found, for instance, fewer context switches first [3], or priority-based scheduling [4], etc. Second is the *modeling* of supported concurrency primitives that specify their semantics, for instance, the behavior of spawning a thread or acquiring a semaphore, etc. Last is the *injection* of these models into a program so that calls to the original primitives can be replaced by calls to their corresponding models, helping take over scheduling decisions at runtime.

Prior work attempts to offer integrated solutions that address all three aspects for a particular domain. For instance, the CHESS tool supports a subset of C# threading APIs (from the `System.Threading` namespace). Their models are build into the tool itself, and they are injected automatically into the program binary via binary instrumentation. This offers a seamless experience for supported programs. However, there is no easy way to extend the tool to programs outside this supported class. Even C# `Task`-based or async-await programs, which are very commonplace, are not supported. Furthermore, use of complex technology like binary instrumentation makes the tool hard to maintain. Other tools do not fare any better. SAMC, for instance, only supports `Sockets`-based communication that too only for `ZooKeeper`-like distributed systems. Supporting other systems was arguably never a goal of that work.

This paper aims to democratize systematic testing by allowing users (i.e., developers) to build their own systematic testing solutions. We propose Nekara, a cross-platform open-source library that provides a simple, yet expressive API that a developer can use to model the set of concurrency primitives that they have used in their program. Nekara encapsulates various search heuristics from prior work, freeing the developer from having to design these themselves. Nekara can record scheduling decisions and provide full repro for bugs.

Unlike an integrated solution, Nekara delegates modeling, as well as injection of these models, to the developer. We argue, through experience presented in this paper, that the burden on developers is small, and is out-weighted by many engineering benefits. In most cases, applications depend on a well-defined framework or library to provide concurrency support (e.g., `pthreads`). The user can then only design models for these framework APIs (only ones they used) with Nekara hooks, and the rest of the system remains unchanged.

Importantly, these Nekara-enabled models are just code; they live and evolve alongside the rest of the system, and benefit from standard engineering practices. The models can be easily injected for testing, say through macros for C code or

mocking utilities for higher-level languages like C#. (Binary instrumentation is an overkill.) They can be shared between multiple teams that use the same form of concurrency, or a community can contribute to support models for a popular framework like `pthreads`. Developers can even choose to limit their code to only those APIs for which they have these models, given the clarity that Nekara provides.

Nekara also helps with other forms of *non-determinism* as well (such as failure injection), and provides the same benefits of reproducibility. In all, with Nekara, systematic testing of non-determinism is just another programming exercise, completely in the hands of the developer.

We show the effectiveness of Nekara by documenting several case studies, each time showcasing that modeling concurrency requires minimal effort (less than a few weeks, easy to share) and provides significant value for testing (more bugs found). Nekara is just 3K lines of C++ code.[1] We believe that a solution like Nekara unlocks the potential of systematic testing that has so far been siloed in individual tools.

The main contributions of this paper are as follows.

- The design of the Nekara library that allows building custom systematic-testing solutions (Sections II and III).
- Experience from several case studies that cover a range of different forms of concurrency and non-determinism (Sections IV to VI). Nekara has been adopted by Microsoft Azure to test distributed services (Sections VI-A and VI-B).
- The search algorithms in Nekara are inspired from previous work; we show that the generality of Nekara does not limit its bug-hunting abilities (Section VII).

Section IX discusses related work and Section X concludes.

II. Nekara Library

This section illustrates how Nekara can be used to build a systematic testing solution. Nekara is only meant for testing. Production code need not take a dependency on Nekara.

The core Nekara APIs are shown in Figure 1. Nekara *operations* generalize over threads and *resources* generalize over synchronization. Operations and resources are uninterpreted, each is just an integer. Nekara understands that each operation, once started and before it ends, executes concurrently with respect to other operations, unless it *blocks* on a resource. Operations while executing can additionally *signal* a resource, which unblocks *all* operations that may be blocked on it.

Nekara implements a co-operative scheduler (§III). It ensures that at most one operation executes at any point in time. The current operation continues running until it calls `schedule_next` or `wait_resource`. At that point, Nekara can switch to execute some other enabled operation.

It is up to the programmer to map the concurrency in their program to Nekara operations and resources. We explain this exercise using the simple concurrent program shown in Figure 2. The program contains a test case, which executes the method `foo` using multiple threads and does an assertion

[1]Available open source at https://github.com/microsoft/coyote-scheduler

```
// Starting/stopping the scheduler
void attach();
void detach();

// Operations
void create_operation();
void start_operation(int op);
void end_operation(int op);

// Resources
void wait_resource(int rid);
void signal_resource(int rid);

// Co-operative context switch
void schedule_next();

// Nondeterministic choices
int next_integer(int min, int max);
```

Fig. 1: Core APIs of the Nekara library.

at the end. Clearly, the assertion can fail, but only in a specific interleaving where thread `t1` updates the shared variable `x` last. For this program, it makes sense to map threads to operations and locks to resources. Our goal will be to simply mock the concurrency-related APIs and leave the rest of the program unchanged as much as possible.

Figure 3 shows the mocks. The `create_thread_wrapper` function calls `create_operation` from the parent thread and then `start_operation` in the child thread. The former informs Nekara that a new operation is about to be created, whereas the latter allows Nekara to take control of the child thread. The actual thread is still spawned via `create_thread`; Nekara does not provide any runtime of its own.

Next step is to *implement* the synchronization via resources. We map each lock to a unique resource, and then `acquire` and `release` methods can be mocked as shown in Figure 3. For ease of explanation, we have assumed that the resource id corresponding to a lock object `x` is stored in the field `x->id`, and a Boolean variable representing the status of the lock (locked/unlocked) is stored in the field `x->acquired`. Because Nekara performs co-operative scheduling, implementing synchronization is typically easy. The procedure `acquire_wrapper` first does a context switch to give other operations the chance to acquire the lock, and then goes ahead to grab the lock if it is available. When the lock is not available, then the operation blocks on the corresponding resource. In `release_wrapper`, we signal the resource, so that blocked operations can try to grab the lock again. (Note that signalling a resource unblocks all waiting operations, but Nekara ensures that only one unblocked operation executes at any point in time.) Any other way of implementing a lock is acceptable, as long as it ensures that the only blocking operation is `wait_resource`.

The final step is to run the test in a loop for a desired number of iterations, under the control of Nekara's scheduler, as shown in Figure 4. Nekara uses search heuristics to explore the space of interleavings of the underlying test. Nekara records the set of scheduling decisions that it makes. When a test iteration fails, the sequence of scheduling decisions can be supplied back to the Nekara scheduler to directly reproduce the buggy iteration. The default set of search heuristics in Nekara are randomized, so we measure the effectiveness of Nekara as the percentage of buggy iterations, i.e., what fraction of

Project	LoC	Operations	Resources	Modeling LoC	Modeling #PW
Memcached (§IV)	21K	`pthread_t`	`pthread_mutex_t`, `pthread_cond_t`, `libevent::event_loop`	1K	2
Verona (§V)	8K	`std::thread`	`std::condition_variable`, `std::atomic`	302	n/a
CSCS (§VI-A)	56K	`Task`	`TaskCompletionSource<T>`	3K	4
ECSS (§VI-B)	44K	`Task`	`TaskCompletionSource<T>`, `lock`	3K	4
Coyote (§VII)	27K	`Actor`	`EventQueue`	16	<1

TABLE I: Systems integrated with Nekara.

```
int x; // Shared variable
LCK lock; // Mutex

void test() {
  // Initialization
  x = 0; lock = new LCK();
  // Run workload
  t1 = create_thread(&foo, 1);
  t2 = create_thread(&foo, 2);
  t3 = create_thread(&foo, 3);
  // Continues in right column.
  thread_ join(t1);
  thread_ join(t2);
  thread_ join(t3);
  assert(x != 1);
}
void foo(int arg) {
  acquire(lock);
  x = arg;
  release(lock);
}
```

Fig. 2: A simple concurrent program.

```
int create_thread_wrapper(
    FUNC_PTR foo,
    ARGS_PTR args) {
  int op = create_new_op();
  scheduler.create_operation(op);
  create_thread(starter, (op,
    foo, args));
}
void starter(int op, FUNC_PTR foo,
    ARGS_PTR args) {
  scheduler.start_operation(op);
  foo(args);
  scheduler.end_operation(op);
}
void acquire_wrapper(LCK lock) {
  scheduler.schedule_next();
  while (true) {
    if (lock.acquired == true) {
      scheduler.wait_resource(lock->id);
    } else {
      lock.acquired = true; break;
    }
  } }
void release_wrapper(LCK lock) {
    assert(lock.acquired);
    lock.acquired = false;
    scheduler.signal_resource(lock->id);
  }
```

Fig. 3: Mocks for thread creation, and lock acquire-release.

```
void nekara_test() {
  Scheduler scheduler(options);
  for (int i = 0; i < 100; i++) {
    scheduler.attach(); // Start the scheduler
    test(); // Run the test for iteration i
    scheduler.detach(); // Stop the scheduler
} }
```

Fig. 4: A typical Nekara test running for 100 iterations.

invocations of the underlying test fail.

Completeness, Limitations: A natural question is if Nekara's exploration is *complete*, i.e., if it is possible to explore all behaviors of a given test in the limit. With Nekara, we leave this decision to the developer. Completeness can be achieved by inserting a context switch just before each *non-commuting action* [16]. For message-passing programs, non-commuting actions are typically just message transfers, thus instrumenting them is enough to get completeness.

In shared-memory programs, an access to shared memory is potentially non-commuting. One option is for a developer to implement their own methodology for inserting context switches at memory accesses in their code. A second simpler (and common [2]) solution is to only insert at synchronization APIs. Then the instrumentation remains limited to mocks of the synchronization APIs, and does not pollute the code. One case study (§V) uses the former strategy whereas rest use the latter strategy. With Nekara, the aim is not necessarily to find all bugs (it's a testing solution after all), but rather to significantly improve existing practice. Behaviors induced by weak memory models [17] are also outside the scope of this paper (although interesting future work).

Nondeterminism: The call to `next_integer` returns a randomly-generated integer within the supplied range. The returned value is recorded to allow for replay. This is handy for modeling non-determinism such as failure injection (§VI-A) or abstracting branches (§V).

Case Studies: We demonstrate Nekara on various systems (Table I). Memcached [18] is a popular open-source in-memory key-value store with cache management. Verona [19] is a language runtime written in C++. CSCS and ECSS are production cloud services of Microsoft Azure, written in C#. Coyote [20] provides a C# actor-based programming framework for building distributed systems. The third and fourth columns of Table I list what is mapped to Nekara operations and resources, respectively. The last two columns show the modeling effort: lines of code (LoC) as well as the number of person weeks (#PW) spent for modeling. Verona used custom Nekara modeling from the outset, so we cannot quantify the effort. Each of CSCS and ECSS use the same mocks, which were developed once in 4 person weeks. It is worth noting that Nekara testing is an integral part of the engineering process for Verona, CSCS and ECSS.

III. Nekara Implementation

The Nekara scheduler must be *attached* before it takes any action; once *detached*, all Nekara APIs are no-ops. A simplified implementation of the core Nekara APIs is shown in Algorithm 1. Nekara also includes APIs for joining on an operation, blocking on a conjunction or a disjunction of resources, as well as signalling a particular operation. These additional APIs are not discussed here.

Nekara must ensure that only one operation executes at any point in time, and it must block the rest, to give it precise control of the program's execution. Nekara maintains: the current operation o_{cur}, a map $M : O \rightarrow R$ that maps

Algorithm 1: Nekara Scheduling APIs

```
   State: int cnt_pending, o_cur; M : O → R
1  Procedure create_operation(o)
2      O ← O ∪ {o};  M[o] ← ∅;  atomic { cnt_pending ++ }
3  Procedure start_operation(o)
4      atomic { cnt_pending −− }
5      o.cv.wait()  // cv is a condition variable
6  Procedure schedule_next()
7      while cnt_pending > 0 do wait()
8      E ← {o | M[o] = ∅} // collect enabled operations
9      if E = ∅ then raise deadlock
10     o_nxt ← S.next(E)  // choose next operation
11     if o_cur = o_nxt then return
12     o_prev ← o_cur;  o_cur ← o_nxt
13     o_nxt.cv.notify()  // resume next operation
14     if o_prev ∈ E then
15         o_prev.cv.wait()  // pause previous operation
16 Procedure wait_resource(r)
17     M[o_cur] ← {r};  schedule_next()
18 Procedure signal_resource(r)
19     foreach o ∈ O do
20         M[o] ← M[o] \ {r}
```

an operation to a set of resources that it's currently blocked on, and a counter $\texttt{cnt}_{\texttt{pending}}$ that is initialized to zero. An operation o is *enabled* if $M[\texttt{o}]$ is empty, and is *disabled* otherwise. Nekara creates a condition variable o.cv for each operation o that it uses to block the operation.

Calling `create_operation` atomically increments $\texttt{cnt}_{\texttt{pending}}$ to keep track of operations that Nekara should expect to be spawned. No other action is taken, which means that the current operation $\texttt{o}_{\texttt{cur}}$ will continue executing. `start_operation` decrements $\texttt{cnt}_{\texttt{pending}}$ and immediately blocks the caller. Having these two calls decorating the spawn of a new operation was important to keep Nekara independent of the underlying runtime. A call to `start_operation` can happen concurrently with respect to other Nekara API.

The executing operation $\texttt{o}_{\texttt{cur}}$ continues until it calls either `schedule_next` or `wait_resource`. The latter just calls the former, so we just describe `schedule_next`. It first waits until $\texttt{cnt}_{\texttt{pending}}$ goes to zero. (Once this happens, it implies that all operations must be inside Nekara.) Nekara then uses a search heuristic $\mathcal{S}$ to decide the next operation to schedule from the set of all enabled operations (or raise "Deadlock" is none is enabled). If the next operation is the same as the current, no action is necessary. Otherwise, the current operation is blocked and the next operation is scheduled.

Search heuristics: Several search heuristics have been proposed in prior work; Thomson et al. [12] provides a survey. These heuristics are based around empirical observations of where most bugs lie in practice: for instance, prioritize few context switches [3], few delays [21], few priority-exchange points [4], [9], etc.; even selecting the next operation uniformly at random works well in practice [12]. Nekara, by default, has several heuristics implemented and uses all of them in a round-robin fashion (across test iterations).

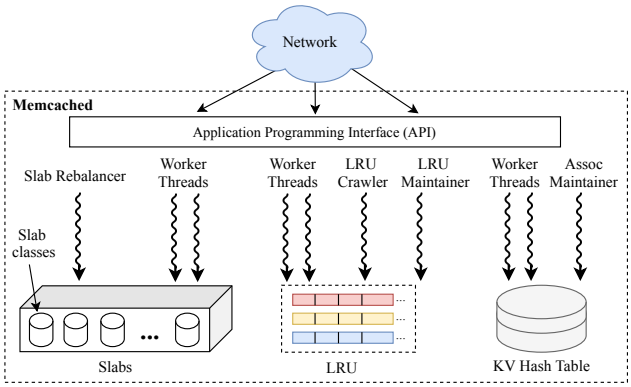

Fig. 5: Architecture of Memcached.

Nekara captures the crux of systematic testing in a small, simple C++ library. It separates search heuristics from the modeling and specialization of testing to a particular system.

IV. Case Study: Memcached

Memcached [18] (MC) is an open-source, in-memory key-value store commonly used as a cache between an application and a database. It is mostly written in C with approximately 21 KLOC. Being a popular multi-threaded application, it has been used for benchmarking bug-detection tools in prior work [22], [23], [24], [25].

Figure 5 illustrates the high-level architecture of MC, showing the key data structures at the bottom and the various kinds of threads that access them. MC maintains an in-memory, chained hash table for indexing key-value (KV) pairs. *Worker* threads update the hash table and the *assoc maintainer* thread is responsible for expanding or shrinking the hash table when its load crosses above or below certain threshold.

To reduce internal fragmentation, MC uses a slab-based allocator for storing KV pairs. KV pairs of different sizes are mapped to different slab classes, and the *slab rebalancer thread* redistributes memory among slab classes as needed. For every slab class, MC maintains three linked lists, named *hot*, *warm* and *cold* LRUs, to keep track of the least recently used (LRU) KV pairs. When a slab class runs out of memory, some memory is reclaimed by evicting the least recently used KV pair. Two background threads, called *LRU maintainer* and *LRU crawler*, update these linked lists and performs evictions whenever required.

The *dispatcher thread* is the main thread that starts and stops all other threads. Its primary function is to look for incoming network connections and dispatch them to the available worker threads, which then serve the network client's requests. MC relies on an event-driven, asynchronous model based on `libevent` [14]; worker threads do not block on network I/O, but instead switch to processing another request.

Integrating Nekara: We first needed to make MC more modular and unit-testable. We mocked system calls like `socket`, `sendmsg`, `recvmsg`, `getpeername`, `poll`, `read`, `write`, `etc.` so that we could imi-

	Id	Bug Type	Description	Reference
Known	1	Misuse of pthread API	Double initialization of mutex and conditional variables	PR#566
	2	Data Race	*Dispatcher* and *worker thread* race on a shared field named `stats`	PR#573
	3	Misuse of pthread API	Deadlock due to recursive locking of a non-recursive mutex	PR#560
	4	Misuse of pthread API	Deadlock due to an attempt to `pthread_join` a non-existing thread	Issue#685
	5	Atomicity Violation	Two *worker threads* simultaneously increment the value of the same KV pair	Issue#127
New	6	Atomicity violation	Null pointer dereference when one *worker thread* deletes a global variable while another *worker thread* was updating it	Issue#728
	7	Misuse of pthread API	Attempt to join a non-existant thread	Issue#733
	8	Deadlock	*Slab rebalancer* and the *dispatcher* thread deadlock due to simultaneous invocation of `pthread_cond_signal` by a *worker* and the *dispatcher thread*.	Issue#738
	9	Misuse of pthread API	This bug can lead to either unlocking an unlocked mutex or unlocking a mutex that is held by some other thread (resulting in data races).	Issue#741

TABLE II: List of previously known bugs, as well as new bugs, that Nekara found in Memcached.

	Bug#2	Bug#5	Bug#6	Bug#8
Uncontrolled	✗	0.12%	✗	✗
Nekara	4%	20%	0.8%	0.01%

TABLE III: Percentage of buggy iterations, for Memcached tests. Each test was executed for 10K iterations.

tate network traffic coming from one or multiple MC clients. Nekara-specific work was limited to the mocking of `libevent` and `pthread` APIs. These totalled around 1 KLOC and took a nominal effort of around two weeks.

Existing MC tests did not exercise concurrency, so we wrote a test ourselves. It concurrently invoked several MC operations; the workload itself was a combination of workloads from existing tests. We then compared the response returned by MC with the expected response. Because Nekara tests run in a loop, we needed to ensure that every iteration of the test was independent of the previous one. This required resetting of all the global, static variables and clearing the cache after every iteration.

Bugs: We picked five previously-known concurrency-related bugs, listed in Table II, and injected them back in the latest version. We picked MC's latest stable version, 1.6.8 for experimentation. Some of these bugs (Bugs 1, 2, 5) were either used or found by previous work [25], [23], and others were obtained from GitHub. In the course of testing, we also found four previously unknown bugs (Bugs 6 to 9), which have been confirmed by MC developers.

Bugs related to misuse of pthread APIs (Bug 1, 3, 4, 7, 9) were easy: they were caught in the first test iteration. The only reason they escaped MC's existing tests is that pthreads does not fail on invalid invocations; we found them simply because our pthread API mocks checked for them.

Table III shows results for the remaining bugs where using Nekara was crucial. Most of these bugs could not be caught without Nekara despite running the test several (10K) times. All of previously-unknown bugs were present in MC from at least the past four years, and even prior tools [23], [26] did not find them.

State coverage: In addition to finding bugs, we wanted to check if Nekara indeed provides more coverage of concurrent

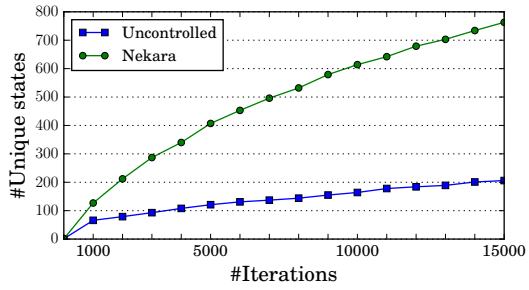

Fig. 6: State coverage for Memcached.

behaviors. We took a hash of all key data-structures of MC (LRU linked lists, slab classes, and the KV table) at the end of a test iteration, and counted the number of unique hashes seen over all test iterations. We ran the Nekara test for 15K iterations and the results are shown in Figure 6.

The results show a roughly four-fold increase in the total number of hashes with Nekara, clearly indicating that it was able to exercise many more behaviors of MC. Deeper inspection revealed that Nekara was able to trigger corner-case behaviors more often, for instance, a slab class running out of memory, or a *get* operation observing a cache miss. Overall, our evaluation demonstrates that a system like MC can benefit greatly from systematic testing, and using Nekara requires little effort.

V. Case Study: Verona

Project Verona [19] is a prototype implementation of a new programming language that explores the interaction between concurrency and ownership to guarantee race freedom. The runtime of the language has several complex concurrent protocols that use shared memory. The runtime has scheduling including work stealing, back pressure (to slow message queue growth), fairness, memory management using atomic reference counting, and global leak detection. These concepts interact in subtle ways that are difficult to debug.

The designers of Verona decided to use systematic testing from the start, motivated by prior struggle with concurrency bugs that took hours to days to resolve, and that had been a significant barrier to making quick progress.

	Uncontrolled	Nekara	Commit Hash
Bug#1	✗	0.049%	25bb324
Bug#2	✗	0.091%	c087803

TABLE IV: Percentage of buggy iterations, for Verona tests. Each test was executed repeatedly for 5 mins.

During development of the Verona runtime, the scheduler, concurrent queues, and memory management protocols were carefully reviewed for uses of shared memory concurrency. Threads were mapped to operations (like in Section II). Synchronization happened in two forms. The first was the use of `std:atomic`; these remain unchanged except that before every store, and after every load, a call to `schedule_next` is inserted, because they mark a potentially racy access to shared memory. The second was the use of condition variables; these directly map to Nekara resources.

Verona uses Nekara's `next_integer` to control other sources of non-determinism. For instance, object identity, in systematic testing builds, was made deterministic, so features that sort based on identity can be tested reproducibly. The runtime has numerous heuristics to postpone expensive operations until there is sufficient work to justify their cost. This implies that certain bugs, which require the expensive operation to execute, can take a long time to manifest. The postponement-heuristic was replaced with systematic choice, i.e., `if next_integer(0,1) == 0`. This had two benefits: (1) it shortened the trace length to find a bug, and (2) it removed any dependency on a specific heuristic's behaviour.

Verona uses systematic testing as part of its code-review and CI process. Any change to the Verona runtime has an additional twenty minutes of CI time for running systematic testing. There is a small amount of stress testing, but most runtime bugs are found using systematic testing. The project's ethos is such that any failure found which did not exhibit during systematic testing is treated as two bugs: the underlying bug, and a bug in the use of systematic testing. The latter is fixed first by writing a systematic test that reveals the bug. Moreover, users of Verona get systematic testing for free because the runtime is already instrumented with Nekara.

Anecdotal evidence from the Verona team has said that the use of systematic testing has given greater confidence for new members of the team to modify the runtime primarily due to the debugging experience of replayable crashes. Detailed logging and replayable crashes provide a way to understand the subtle interactions in concurrent systems that would normally be a significant barrier to entry for new programmers. Most runtime bugs do not make it off the developer's machine as the local systematic testing catches them before CI. Two recent bugs that we investigated are listed in Table IV. Both bugs would not have been found without systematic testing. Bug #1 was a failure to correctly protect memory from being deallocated by another thread. The window for this to occur was a few hundred cycles, hence almost impossible to reproduce reliably without systematic testing. The second bug was due to not considering a corner case of a new feature.

```
class Task{
  static Task Run(Func<Task> func);
  static Task Delay(TimeSpan delay);
  static Task WhenAll(params
    Task[] tasks);
  static Task<Task> WhenAny(
    params Task[] tasks);
    ...
  TaskAwaiter GetAwaiter();
}
```

```
class TaskCompletionSource<T> {
  Task<T> Task { get; }
    ...
  void SetResult(T result);
  void SetException(Exception ex);
  void SetCanceled(
    CancellationToken ct);
}
```

Fig. 7: `Task` and `TaskCompletionSource<T>` APIs.

VI. Case Study: Task Parallel Library

The Task Parallel Library [27] (TPL) is a popular, open-source, cross-platform library provided by .NET for building concurrent applications. TPL exposes several key types, such as `Task` and `TaskCompletionSource<T>` that interoperate with the `async` and `await` keywords in the C# language, enabling writing asynchronous code without the complexity of managing callbacks. A developer writes code using these high-level APIs and TPL takes care of the hard job of partitioning the work, scheduling tasks to execute on the thread pool, canceling and timing out tasks, managing state and invoking asynchronous callbacks.

TPL is pervasive in the .NET ecosystem. We designed TPL^N as a drop-in-replacement library for TPL. TPL^N provides stubs that replace the original TPL APIs, as well as subclasses that override original TPL types. These stubs and subclasses call into Nekara via a C++ to C# foreign-function interface. Any C# application that uses TPL for its concurrency simply needs to replace it with TPL^N to get systematic testing. We now explain core TPL^N design.

The `Task` type (Figure 7) provides several public-facing APIs including: `Task.Run` for queueing a function to execute on the thread pool and returning a task that can be used as a handle to asynchronously await for the function to complete; `Task.WhenAll` and `Task.WhenAny` for waiting one or more tasks to complete; `Task.Delay` for creating an awaitable task that completes after some time passes; and compiler-only APIs, such as `Task.GetAwaiter`, which the C# compiler uses in conjunction with the `async` and `await` keywords to generate state machines that manage asynchronous callbacks [28].

Figure 8 shows the Nekara-instrumented version of `Task.Run`. For simplicity, we have omitted low-level details such as task cancellation and exception handling. `Task.Run` of TPL^N creates a new Nekara operation via `create_operation`, and then invokes the original `Task.Run` to spawn a task. This new task first calls `start_operation`, then invokes the user function, followed by `end_operation`. The parent task calls `schedule_next` to give the child task a chance to execute.

`TaskCompletionSource<T>` (TCS), shown in Figure 7, allows developers to asynchronously produce and consume results. This type exposes a `Task` get-only property

```
using SystemTask = System.Threading.Tasks.Task;
static Task Run(Func<Task> func) {
  // Create a new Nekara operation id for the new task.
  int op = GetUniqueOperationId();
  Nekara.create_operation(op);
  var task = SystemTask.Run(async () => {
    Nekara.start_operation(op);
    // Execute the user-specified asynchronous function.
    // The await logic is instrumented with Nekara.
    await func();
    Nekara.end_operation(op);
  });

  Nekara.schedule_next();
  return task;
}
```

Fig. 8: Instrumentation of `Task.Run` with Nekara.

```
// TCS context used for Nekara testing.
class TCSContext<T> {
  // Nekara resource id for the TCS.
  int Id = GetUniqueResourceId();
  // The result set by the TCS.
  T Result = default;
  bool IsCompleted = false;
}
void SetResult(T result) {
  var context = GetCurrentContext();
  if (!context.IsCompleted) {
    // Set the TCS result.
    context.Result = result;
    context.IsCompleted = true;
    // Signal the TCS consumer.
    Nekara.signal_resource(context.Id);
  }
}
```

```
Task<T> Task => {
  // Get the current TCS context.
  var context = GetCurrentContext();
  if (context.IsCompleted) {
    // If TCS is completed, return the result.
    return Task.FromResult(context.Result);
  }

  // Return a Nekara-controlled task
  // that will be completed by the producer.
  return Task.Run(() => {
    // Wait the producer to set the result.
    Nekara.wait_resource(context.Id);
    // Exception/cancellation logic.
    ...
    return context.Result;
  });
};
```

Fig. 9: Instrumentation of TCS APIs with Nekara.

that a consumer can await to receive a result of type `T` asynchronously. This task remains uncompleted until the producer completes it with a result by invoking `SetResult`.

Instrumentation of TCS `Task` and `SetResult` is shown in Figure 9. We designed `TCSContext<T>`, a simple test-only data structure in TPL^N that contains information needed to model a TCS. `TCSContext<T>` contains a Nekara resource id associated with the TCS, the result of the TCS, and a Boolean value that is set to true once the TCS completes. `TCSContext<T>` is set to the TCS state upon initialization by TPL^N, so that it can be accessed when invoking one of the stub TCS APIs. The producer (`SetResult`) is modeled as follows: it first accesses the context, then checks if the TCS has completed and, if not, it sets the result and `IsCompleted` to `true` and signals the resource associated with the TCS to unblock any task that might be waiting on the TCS `Task` getter property. The consumer (`Task` getter) is modeled as follows: it first accesses the context, then checks if the TCS has completed and, if it has, it simply returns a completed task with the result. If the TCS has not completed yet, it will create a new task by invoking the `Task.Run` TPL^N API (which we described above). This asynchronous task immediately waits on the resource, and once the producer signals, it returns the completed result.

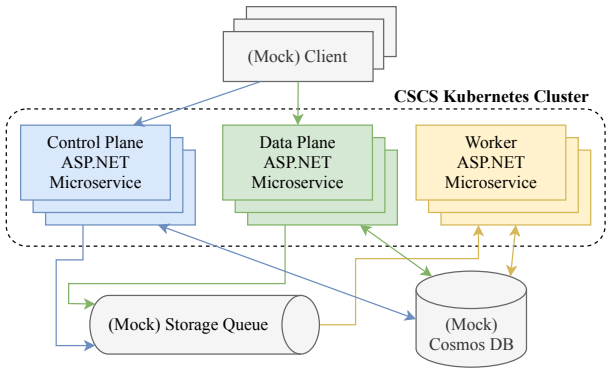

Fig. 10: The high-level architecture of CSCS.

We also modeled two other TPL types. First is the `Monitor` type that implements a reentrant lock in C# (along similar lines to Figure 3). Second is the type `AsyncTaskMethodBuilder` that the C# compiler uses to generate state machines to manage asynchronous callbacks in methods that use `async` and `await`.

Creating TPL^N took roughly one person month, with most time spent in ensuring that we support each TPL API and maintain details such as exception propagation, cancellation, etc. This is largely a one-time effort (unless TPL itself changes significantly). Several engineering teams in Microsoft were able to use TPL^N to test their services without needing any changes to their production code. Two such systems are summarized next in Sections VI-A and VI-B. They use conditional compilation to automatically replace the original TPL library with TPL^N during testing.

A. *Testing CSCS with TPL^N*

Cloud Supply Chain Service (CSCS) exposes a set of HTTP REST APIs that clients can invoke to create supply-chain entities and orchestrate supply-chain processes. CSCS is designed with a typical microservice-based architecture (Figure 10) where multiple stateless microservices coordinate with each other through shared state maintained in backend storage systems. CSCS consists of roughly 56K lines of C# code. It is built using ASP.NET [29] and achieves horizontal scalability through Kubernetes [30].

The CSCS microservices, although stateless, concurrently access the backend storage, including Cosmos DB [31] (a globally-distributed database service) and Azure Queue Storage [32] (a durable distributed queue). Some requests in CSCS follow a simple request/response pattern, while others trigger background jobs, making the client periodically poll to check on the completion of the job. This requires complex synchronization logic between the microservices.

CSCS is written against storage interfaces, so they can be easily mocked during testing using techniques such as dependency injection. Multiple concurrent client calls are simulated by spawning concurrent tasks that invoke the relevant ASP.NET controller actions.

```
// Instantiates a Cosmos DB Mock, an instance of a CSCS
// ASP.NET microservice, and a CSCS client.
var cosmosDbMock = new CosmosDb_Mock(...);
var factory = new ServiceFactory(cosmosDbMock, ...);
var client = factory.CreateClient(...);

// Invokes a concurrent Create and Update client request.
Task req1 = Task.Run(() => client.Create("id",
  { Content: "payload1", Timestamp: 7 }));
Task req2 = Task.Run(() => client.Update("id",
  { Content: "payload2", Timestamp: 8 }));

await Task.WhenAll(req1, req2);

// Gets the resource (stored in Cosmos DB) and asserts
// it contains the expected payload and timestamp.
var resource = await client.Get("id");
Assert.IsTrue(resource.Content == "payload2" &&
  resource.Timestamp == 8);
```

Fig. 11: A simple Nekara concurrent test in CSCS.

	Bug#1	Bug#2	Bug#3	Bug#4
Uncontrolled	✗	✗	✗	✗
Nekara	11%	7%	1%	1%

TABLE V: Percentage of buggy iterations on CSCS tests.

CSCS concurrency unit tests range from simple concurrency patterns where the test calls an API with different inputs but the same key (to exercise interference in Cosmos DB) to more complex tests that randomly fail certain mock invocations (using the `next_integer` API) to simulate intermittent network failures. Figure 11 shows a simple CSCS test. These tests, when exercised by Nekara, were able to uncover subtle bugs. Table V lists some of the bugs, comparing testing with and without Nekara. Each test was run multiple times for a total of 5 minutes and the table shows percentage of buggy runs. Without Nekara, none of these bugs would have been found. Following is a description of these bugs.

a) Data loss due to concurrent requests (Bug#1): CSCS requires that upon two concurrent `Create` or `Update` requests, only the request with the latest modified timestamp succeeds. To achieve this, CSCS uses Cosmos DB ETags functionality for optimistic concurrency control, but a bug in the handling of ETags led to a stale `Create` overwriting fresher data. This bug was missed by both stress testing and manual code review, but found quickly with Nekara (with the test shown in Figure 11). This bug could have lead to customer data loss.

b) Inconsistent entity state (Bug#2): CSCS manages two sets of related entities, which are stored in different tables and partitions of Cosmos DB. Rejecting an update on one of the entities, must lead to rejection of updating the other entity too. However, Cosmos DB does not support transactions between entities stored in different partitions, and the developers had to implement custom synchronization logic to get around this limitation. When the team wrote a concurrent test that tries to cancel and reject an entity at the same time, Nekara uncovered an issue where the system got into an inconsistent state. This bug had escaped stress testing and manual code review.

c) Resource creation liveness issue (Bug#3): Certain requests trigger a background task requiring the client to periodically poll its completion. This functionality is implemented by storing a record indicating the request status in Cosmos DB and then submitting the job to a worker queue to trigger the asynchronous work. There was a bug where the submission to the worker queue could fail due to network connectivity issues. However, as the record was created in the database, the user would find the status to be `pending-creation` upon a `GET` request and would erroneously assume the resource will be eventually created. This *liveness* bug was caught with a test that simulated potential network failures.

d) Race condition in test logic (Bug#4): Interestingly, Nekara found a bug in the CSCS test code itself. The buggy test performed two concurrent `PUT` requests to provision a resource, then waited for the provisioning to complete and then deleted the resource. The test was failing because two concurrent `Create` requests led to two asynchronous workers for the same resource. The test then deleted the resource as soon as one of the workers transitioned the resource state to `created`. However, there was a race between the second asynchronous worker and the `Delete` request, which caused the test to fail. The developers were initially not sure if this bug was in their production logic or test logic, but due to Nekara's reproducible traces they were able to understand the issue and fix it.

B. Testing ECSS with TPL^N

Cloud storage uses geo-redundancy to protect against catastrophic data center failures. The Erasure Coding Storage Service (ECSS) offers an economical solution to this problem by applying erasure coding to blob objects across geographically distributed regions. Figure 12 shows a partial view of the high-level architecture of ECSS. The system consists of a data service and a metadata service. The data service is responsible for striping data and generating parities across regions, as well as reconstructing data in the event of failures. The metadata service manages erasure coding stripes and handles dynamic updates to the stripes (due to object creation, update, and deletion). ECSS consists of roughly 44K lines of C# code.

To achieve high-throughput, ECSS was designed to be highly-concurrent. The data service implements a component called Syncer that periodically synchronizes metadata between individual regions and the metadata service using Azure Queue Storage. Syncer is sharded and the ECSS developers implemented a lease-based mechanism to assign different Syncer partitions to different metadata service nodes. The metadata service executes two long running TPL tasks: a table updater and a table scanner. The updater asynchronously dequeues Syncer messages from Azure Queue Storage and uses their contents to update Azure Table Storage. The scanner periodically scans Azure Table Storage to check different types of metadata. Based on the metadata state, the scanner will

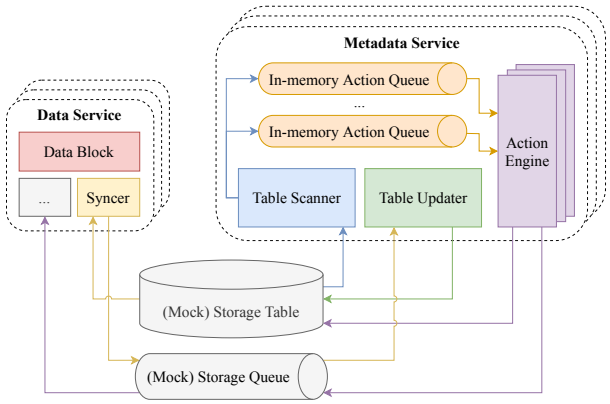

Fig. 12: Parts of the high-level architecture of ECSS.

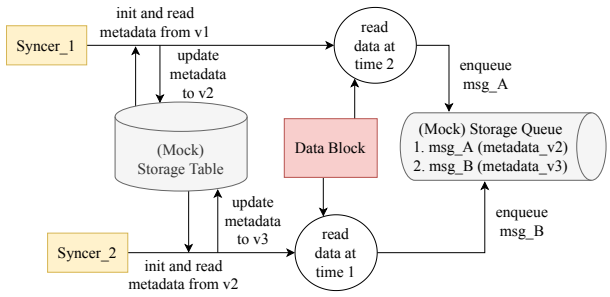

Fig. 13: Race condition in the ECSS data service.

enqueue actions on a set of in-memory queues. Long-running tasks execute action engines that drain these queues, perform actions on the state and update the metadata in Azure Table Storage, as well as sending action messages to the data service through Azure Queue Storage to do erasure coding among other operations.

Eventually, the state of the data service and Azure Table Storage must be consistent. ECSS manages exabytes of customer data, and correctness is absolutely vital, which is why the team used TPL^N for thorough testing. No changes were required to the production code. The main investment was in writing the tests: the tests instantiate the Syncer and the metadata service in-memory, use dependency injection for inserting their mocks of Azure Table Storage and Azure Queue Storage, write input data and finally assert that the Azure Table Storage state is consistent.

Nekara helped find several critical bugs that could have resulted in data loss. Figure 13 illustrates one of these bugs, a race condition between two Syncer instances from the same partition. The first Syncer instance (`Syncer_1`) starts and reads metadata with version 1 from Azure Table Storage, updates it and writes it back to the table with version 2. On the same partition, a new Syncer instance (`Syncer_2`) starts, reads the same metadata (which now has version 2), and updates it to version 3. Immediately after, `Syncer_2` reads from the data block, but just before it continues its execution, `Syncer_1` reads the latest data from the same block and generates msg_A containing version 2 of the metadata and enqueues this message to Azure Queue Storage. Next, `Syncer_2` continues executing and generates msg_B containing version 3 of the metadata and also enqueues this message. Finally, the metadata service dequeues conflicting messages from the same partition, which could result in a later version of the metadata being overwritten by an outdated version. To fix the bug, the developers had to write logic just before enqueueing the messages to check if the metadata and the version are consistent, and if not then discard the message.

This bug had escaped continuous unit and stress tests. Nekara was able to find it in under 10 seconds and just 8 test runs (on average). The ECSS team (as well as the CSCS team) routinely run Nekara tests as part of their CI.

	TPL^N		Coyote^N		Coyote	
Benchmarks	**BI%**	**Time**	**BI%**	**Time**	**BI%**	**Time**
Ch. Replication	✗	594	**0.01%**	**197**	**0.01%**	224
Fail. Detector	✗	39	**0.08%**	**19**	0.07%	23
Paxos	0.05%	254	**0.07%**	**56**	0.06%	104
Raft	0.35%	789	0.29%	**151**	**0.38%**	156

TABLE VI: Comparing systematic testing with Nekara against the original Coyote. Tests run for 10k iterations. BI% denotes percentage of buggy iterations. Time is in seconds.

VII. Reproducing Known Bugs

We evaluate Nekara against three prior tools on their own set of benchmarks. The purpose of this evaluation is to show that one can get state-of-the-art systematic testing with Nekara.

a) Coyote: The first comparison is against Coyote [20] that provides an open-source .NET actor library, used by teams in Azure [33]. Coyote provides a runtime for executing actors. All actors execute concurrently and communicate by sending messages to each other. Coyote also provides a systematic testing solution for testing these actor-based programs [11], [34].

We directly instrumented the Coyote runtime using Nekara to build our own systematic testing solution. We had two options. In the first approach, we took advantage of the Coyote runtime being implemented entirely on top of TPL, which we simply replaced with TPL^N. In the second approach, we instead instrumented at the level of actor semantics: an actor is mapped to an operation, and the actor's inbox is mapped to a resource. We skip the details of this instrumentation for lack of space; the high-level summary is that it only required 16 lines of changes to the Coyote runtime. We refer to the second approach as Coyote^N. Note that in both approaches, all changes were limited to the Coyote runtime; the user program remains unchanged.

Table VI shows results on prior Coyote benchmarks, which consist of buggy protocol implementations. The two Nekara approaches have different advantages. TPL^N has the benefit of being able to test the Coyote runtime itself (that it correctly implements actor semantics). Moreover, the instrumentation was mechanical and required no knowledge of Coyote itself.

```
class MockDictionary<K, V> : Dictionary<K, V> {
  int SharedVar; bool IsWrite = false; ...
  bool override ContainsKey(K key) {
    var tid = Task.CurrentId;
    SharedVar = tid;
    Nekara.schedule_next();
    // Check for race.
    assert(!(SharedVar != tid && IsWrite));
    return base.ContainsKey(key);
  }
  void override Add(K key, V value) {
    var tid = Task.CurrentId;
    SharedVar = tid; IsWrite = true;
    Nekara.schedule_next();
    // Check for race.
    assert(SharedVar == tid);
    IsWrite = false;
    base.Add(key, value);
  }
}
```

Fig. 14: Mock dictionary instrumented to detect races.

	Applications	LoC	#BF	Nekara BR?	Nekara BI%
TSVD	DataTimeExtention	3.2K	3	✓	89.3%
	FluentAssertion	78.3K	2	✓	51.3%
	K8s-client	332.3K	1	✓	11.7%
	Radical	96.9K	3	✓	28.3%
	System.Linq.Dynamic	1.2K	1	✓	99.0%
	Thunderstruck	1.1K	2	✓	48.3%
Maple	SpiderMonkey*	200K	2	✓	0.5%
	Memcached-1.4.4	10K	1	✓	20.0%
	StreamCluster*	2.5K	3	✓	41.7%
	Pbzip2	1.5K	1	✓	50.0%

TABLE VII: Comparison with TSVD and Maple. #BF is number of bugs found by these tools. The ✓(under BR?) denotes that Nekara reproduced all these previously found bugs. BI% denotes percentage of buggy iterations.

However, bugs in the Coyote runtime was not in question here, and we found this approach to have worse performance than the other systematic testing solutions. Instrumenting at the TPL level significantly increased the number of operations and scheduling points in the program, which decreased bug-finding ability and increased test time. Coyote^N, however, was comparable to Coyote because both directly leverage actor semantics. We also compared against uncontrolled testing, which unsurprisingly, could not find any of the bugs.

We make a note on the relationship of Coyote to Nekara because both projects have influenced each other. Coyote had earlier only supported an actor-based programming model. Nekara inherited many of the search techniques used in Coyote, but applied them to a generalized setting. Given the success of Nekara's TPL^N model in testing Task-based programs, the Coyote team has since incorporated that work natively to support Task-based programs as well. Coyote uses bytecode-level instrumentation to automatically inject hooks into an unmodified C# program, making the end-user experience more seamless than with using TPL^N.

b) TSVD: In the second experiment, we reproduce bugs found by TSVD [35], a tool that looks for *thread-safety violations* (TSVs), which are concurrent invocations of operations on a non-thread-safe data structure (e.g., a `Dictionary`). TSVD's open-source evaluation [35] covered nine .NET applications; here we consider six of those (one was removed from GitHub and in two others we were unable to locate a TSV bug from the cited GitHub issue). In each application, we replaced TPL with TPL^N and made one additional change. To capture TSV as an assertion, we implemented a `MockDictionary` type as a subclass of `Dictionary`, and modified the application to use this type. Figure 14 shows this mock: an assertion failure in the mock corresponds to a TSV. Table VII shows the results: we were able to find all TSVs, taking at most 8 iterations on average. In the DataTimeExtension benchmark, we found four additional TSVs (not reported by TSVD) by running the test for 100 iterations.

c) Maple: The third comparison is against Maple [36], a systematic-testing tool that uses dynamic instrumentation and provides coverage-driven testing through online profiling and dynamic analyses. We picked a set of real-world benchmarks from SCTBench [12]. These include: an older version of Memcached, SpiderMonkey (a JavaScript runtime engine), StreamCluster (online clustering of input streams), and pbzip2 (a parallel implementation of bzip2). Bugs in these applications include atomicity violations, order-violation, deadlocks and livelocks. All these benchmarks use `pthreads` so we reused the models from Section IV. Table VII shows the results. Nekara found all the reported bugs; most in a small number of iterations. These numbers are either comparable or smaller than those reported by Maple, although a direct comparison is not possible because of the different set of techniques and instrumentation used. For some benchmarks, marked in the table with a *, we also inserted `schedule_next` just before global variables accesses, without which we were unable to find some of the bugs (due to the incompleteness issue mentioned in Section II).

VIII. Experience Summary

This section summarizes our experience in integration Nekara with the various case studies presented in this paper.

Integration: We were able to integrate Nekara testing with several systems with no changes to the production code in most cases. Only in the case of Memcached, we had to modify the code in order to mock system calls to sockets so that we could write unit tests with multiple concurrent clients. The fact that production code need not take a dependency on Nekara was important was acceptance with development teams.

Modeling effort: Table I shows that the modeling effort was small to moderate. For systems like Verona, systematic testing was always an integral part of its development process. Systems like ECSS and CSCS fully shared their models (TPL^N) demonstrating amortization of effort across teams.

Development of the models does require some expertise in reasoning about concurrency, especially for understanding the semantics of synchronization APIs and encoding it correctly using Nekara resources. Correctness of these models is a concern, because mistakes can lead to deadlocks when running Nekara tests. We leave the problem of validating models as future work. In general, it would be interesting to explore a community-driven effort that maintains models of common concurrency frameworks or APIs.

Bugs: Nekara helped catch several bugs, including liveness bugs, deadlocks, memory leaks as well as functionality bugs like data corruption. Many of these would not have been caught with existing practices like stress testing or code review.

Writing test cases: Developing good tests that exercise concurrency in the system, and assert something meaningful, are crucial to get benefits from systematic testing. For Memcached, we wrote a generic test by simply combining existing test cases. For other systems, the developers of those systems were able to write tests themselves without our help.

Debugging: Repro capabilities of Nekara was well appreciated by developers. Furthermore, instrumentation of a runtime (like Verona or Coyote) provides systematic testing to users of the runtime for free.

IX. Related Work

The systematic testing approach has its roots in *stateless model checking*, popularized first by VeriSoft [10]. *Stateful* approaches require capturing the entire state of a program in order to avoid visiting the same state again, because of which these techniques are typically applied on an abstract model of an implementation [37], [38]. Stateless approaches, on the other hand, only control program actions. They do not inspect program state at all, consequently are directly applicable to testing the implementation itself. All these techniques were initially focussed on verification. CHESS [39] shifted the focus to bug finding; it prioritized exploration of a subset of behaviors, one with a few number of context switches, and found many bugs. Several randomized search techniques [4], [9], [12], [40], [21] have followed since, showcasing very effective bug-finding abilities. Our focus has been on making systematic testing easy to use, to that end we capture all these search techniques inside Nekara. One class of techniques that we are unable to capture in Nekara's interface currently is partial-order-reduction (POR) based techniques [10], [41]. POR requires knowing if the *next* steps of two different operations are *independent* of each other or not. Supporting POR require more information to be supplied to Nekara, which is interesting future work.

Instantiation of systematic testing exists for multi-threaded [42], [2], [3], [4], [41] as well as message-passing programs [5], [6], [7], [8], [9], however their combined reach is still small. For instance, most C# applications that we considered are built on TPL Tasks. These Tasks eventually execute on the .NET threadpool, so one can consider them to be multi-threaded applications, but instrumenting the .NET runtime is challenging, and we did not find any readily applicable tool. Moreover, even if this was possible, instrumenting at a higher-level is typically much easier (§VI) and more efficient (§VII). Our goal is to unlock systematic testing; modeling is not hidden inside a tool, but available as code that can be readily adopted by others.

Complementary to systematic testing is the work on finding low-level concurrency bugs such as data races and atomicity violations [43], [44], [45], [46], [47]. These techniques examine a given concurrent execution and evaluate if there were potentially racy accesses. Data race detection typically requires monitoring memory accesses at runtime. Nekara can help generate a diverse set of concurrent executions for these tools, or be used directly for specific races (§VII).

Another related problem is deterministic replay of processes, even virtual machines [48], [49], [50], [51], [52]. These techniques seek generality to support arbitrary systems, and require very detailed tracing. The problem is simpler in our setting, as we are focussed on a single test written by a developer aware of the concurrency used in their application.

X. Conclusions

Systematic testing holds promise in changing the way we test concurrent systems. In this paper, we present Nekara, a simple library that allows developers to build their own systematic testing solutions. Integration of Nekara is a simple programming exercise that only requires modeling of key concurrency APIs. The model code is easy to share to further amortize the cost across multiple projects. We report several case studies where the use of Nekara made considerable impact, both in existing systems, as well as systems designed from scratch with systematic testing.

References

[1] J. Gray, "Why do computers stop and what can be done about it?" in *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*. IEEE, 1986, pp. 3–12.

[2] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 267–280.

[3] M. Musuvathi and S. Qadeer, "Fair stateless model checking," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 362–371. [Online]. Available: https://doi.org/10.1145/1375581.1375625

[4] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, 2010, pp. 167–178. [Online]. Available: https://doi.org/10.1145/1736020.1736040

[5] J. Simsa, R. Bryant, and G. A. Gibson, "dbug: Systematic testing of unmodified distributed and multi-threaded systems," in *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, ser. Lecture Notes in Computer Science, A. Groce and M. Musuvathi, Eds., vol. 6823. Springer, 2011, pp. 188–193. [Online]. Available: https://doi.org/10.1007/978-3-642-22306-8_14

[6] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MODIST: transparent model checking of unmodified distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009, pp. 213–228.

[7] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 399–414.

[8] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, A. Gupta, S. Lu, and H. S. Gunawi, "Flymc: Highly scalable testing of complex interleavings in distributed systems," in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, 2019, pp. 20:1–20:16.

[9] B. K. Ozkan, R. Majumdar, F. Niksic, M. T. Befrouei, and G. Weissenbacher, "Randomized testing of distributed systems with probabilistic guarantees," *PACMPL*, vol. 2, no. OOPSLA, pp. 160:1–160:28, 2018.

[10] P. Godefroid, "Software model checking: The verisoft approach," *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, 2005. [Online]. Available: https://doi.org/10.1007/s10703-005-1489-x

[11] P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal, R. Mudduluru, S. Qadeer, and W. Schulte, "Uncovering bugs in distributed storage systems during testing (not in production!)," in *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.*, 2016, pp. 249–262.

[12] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using controlled schedulers: An empirical study," *ACM Transactions on Parallel Computing*, vol. 2, no. 4, pp. 1–37, 2016.

[13] Microsoft, "Azure Service Fabric," https://azure.microsoft.com/services/service-fabric/.

[14] libevent, "An event notification library," https://libevent.org/.

[15] Trio, "A friendly Python library for async concurrency and I/O," https://trio.readthedocs.io/en/stable/.

[16] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717–721, Dec. 1975.

[17] B. Norris and B. Demsky, "Cdschecker: checking concurrent data structures written with C/C++ atomics," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, 2013, pp. 131–150.

[18] Memcached, "An in-memory key-value store," https://www.memcached.org/, 2020.

[19] Microsoft Research, "Verona: Research programming language for concurrent ownership," https://github.com/microsoft/verona, 2021.

[20] Microsoft Coyote, "Fearless coding for reliable asynchronous software," https://github.com/microsoft/coyote, 2020.

[21] M. Emmi, S. Qadeer, and Z. Rakamaric, "Delay-bounded scheduling," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, 2011, pp. 411–422.

[22] T. Elmas, J. Burnim, G. Necula, and K. Sen, "CONCURRIT: a domain specific language for reproducing concurrency bugs," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 153–164.

[23] D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, and K. Wehrle, "Symbolic partial-order execution for testing multi-threaded programs," *arXiv preprint arXiv:2005.06688*, 2020.

[24] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi, "Multicore acceleration of priority-based schedulers for concurrency bug detection," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 543–554.

[25] N. Jalbert, C. Pereira, G. Pokam, and K. Sen, "RADBench: A concurrency bug benchmark suite." *HotPar*, vol. 11, pp. 2–2, 2011.

[26] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*, 2009, pp. 62–71.

[27] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2009, pp. 227–242.

[28] Microsoft, "Asynchronous programming with async and await," https://docs.microsoft.com/dotnet/csharp/programming-guide/concepts/async/, 2020.

[29] ——, "ASP.NET: A framework for building web apps and services with .NET and C#," https://dotnet.microsoft.com/apps/aspnet, 2020.

[30] The Kubernetes Authors, "Kubernetes," https://kubernetes.io/, 2020.

[31] Microsoft, "Cosmos DB: Fast NoSQL database with open APIs for any scale," https://azure.microsoft.com/en-us/services/cosmos-db/, 2020.

[32] ——, "Queue Storage: Durable queues for large-volume cloud services," https://azure.microsoft.com/en-us/services/storage/queues/, 2020.

[33] P. Deligiannis, N. Ganapathy, A. Lal, and S. Qadeer, "Building reliable cloud services using P# (experience report)," *CoRR*, vol. abs/2002.04903, 2020. [Online]. Available: https://arxiv.org/abs/2002.04903

[34] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson, "Asynchronous programming, analysis and testing with state machines," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 154–164.

[35] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 162180, https://doi.org/10.1145/3341301.3359638.

[36] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 485502. [Online]. Available: https://doi.org/10.1145/2384616.2384651

[37] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2011.

[38] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: A model checker for concurrent software," in *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, 2004, pp. 484–487.

[39] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, 2007, pp. 446–455.

[40] A. Desai, S. Qadeer, and S. A. Seshia, "Systematic testing of asynchronous reactive systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 73–83.

[41] J. Huang, "Stateless model checking concurrent programs with maximal causality reduction," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 165–174.

[42] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A pragmatic approach to model checking real code," in *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002.

[43] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, 2004, pp. 256–267.

[44] S. Park, S. Lu, and Y. Zhou, "Ctrigger: exposing atomicity violation bugs from their hiding places," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, 2009, pp. 25–36.

[45] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, 2009, pp. 121–133.

[46] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 2008, pp. 11–21.

[47] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs," in *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, 1997, pp. 27–37.

[48] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," in *Proceedings of the 2nd International Conference on Virtual Execution Environments*, ser. VEE '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 154–163. [Online]. Available: https://doi.org/10.1145/1134760.1220164

[49] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 211–224, Dec. 2003. [Online]. Available: https://doi.org/10.1145/844128.844148

[50] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA:

Association for Computing Machinery, 2008, pp. 121–130. [Online]. Available: https://doi.org/10.1145/1346256.1346273

[51] J.-D. Choi and H. Srinivasan, "Deterministic replay of java multithreaded applications," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, ser. SPDT '98. New York, NY, USA: Association for Computing Machinery, 1998, pp. 48–59. [Online]. Available: https://doi.org/10.1145/281035.281041

[52] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA '03. New York, NY, USA: Association for Computing Machinery, 2003, pp. 122–135. [Online]. Available: https://doi.org/10.1145/859618.859633