CHESS: A Systematic Testing Tool for Concurrent Software

Madanlal Musuvathi
Microsoft Research

Shaz Qadeer
Microsoft Research

Thomas Ball
Microsoft Research

Technical Report
MSR-TR-2007-149

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

CHESS: A Systematic Testing Tool for Concurrent Software

Madanlal Musuvathi
Microsoft Research

Shaz Qadeer
Microsoft Research

Thomas Ball
Microsoft Research

Abstract

Concurrency is used pervasively in the development of large systems programs. However, concurrent programming is difficult because of the possibility of unexpected interference among concurrently executing tasks. Such interference often results in "Heisenbugs" that appear rarely and are extremely difficult to reproduce and debug. Stress testing, in which the system is run under heavy load for a long time, is the method commonly employed to flush out such concurrency bugs. This form of testing provides inadequate coverage and has unpredictable results. This paper proposes an alternative called concurrency scenario testing which relies on systematic and exhaustive testing We have implemented a tool called CHESS for performing concurrency scenario testing of systems programs. CHESS uses model checking techniques to systematically generate all interleaving of a given scenario. CHESS scales to large concurrent programs and has found numerous previously unknown bugs in systems that had been stress tested for many months prior to being tested by CHESS. For each bug, CHESS is able to consistently reproduce an erroneous execution manifesting the bug, thereby making it significantly easier to debug the problem. CHESS has been integrated into the test frameworks of many code bases inside Microsoft and is being used by testers on a daily basis.

1 Introduction

Given the importance and prevalence of concurrent systems,[1] it is unfortunate that there exists no first-class notion of concurrency testing. What is "concurrency testing"? In theory, concurrency testing is the process of testing a concurrent system for correct behavior under all possible schedules. In practice, people almost always identify concurrency testing with stress testing, which evaluates the behavior of a concurrent system under load. While stress testing does indirectly increase the variety of thread schedules, such testing is far from sufficient. Stress testing does not cover enough different thread schedules and, as a result, yields unpredictable results. A bug may surface one week, when stress testing happens to cover a low-probability schedule, and then disappear for months. Stories are legend of the so-called "Heisenbugs" that rarely surface and are hard to reproduce. Still, the mark of reliability of a system remains its ability to run for a long time under heavy load without crashing.

We introduce a new concept called concurrency *scenario* testing that is applicable to large systems.The system designer thinks of interesting concurrency *scenarios*. For instance, an interesting scenario for a device driver is concurrently receiving both an I/O message and a shutdown message. These scenarios are system specific and intuitively correspond to unit-tests for sequential programs, though we expect concurrent scenarios to be more complex.

Given these scenarios, we use model checking [7, 31] techniques to systematically cover all thread schedules. A model checker essentially captures the nondeterminism of a system and then systematically enumerates all possible choices. For a multithreaded process, this approach is tantamount to running the system under a demonic scheduler. We identify and address three key challenges in making model checking applicable to large shared-memory multithreaded programs.

First, existing model checkers requires the programmer to do a huge amount of work just to get started. We call this the "perturbation problem." Traditional methods for model checking are meant to formally verify abstract models and do not work on real code. Other direct-execution methods for checking multithreaded code require significant modifications to the system under test. For instance, CMC [26, 39] requires running the Linux kernel in user-space in order to analyze it, a non-trivial engineering effort. Tools like JPF [37] and ExitBlock [4] force the user to use a specialized JVM. Other recent approaches to model checking systems code, such as MaceMC [19], require writing the program in a new programming language.

The perturbation problem makes model checking a non-starter for testers who want better control over the execution of concurrent programs. Whenever code is changed for the benefit of testing, the tester knows that the real bits are not being tested. Moreover, testers do not have the luxury of changing code. Changing code takes time and is risky. Finally, testers already have extensive and varied test infrastructure in place and are unwilling to "buy into" a different testing tool. Therefore, a concurrency test tool should easily integrate with existing test infrastructure with no modification to the system under test and little modification to the test harness.

Second, concurrency is enabled in most systems via rich and complex concurrency APIs.[2] For instance, the Win32 API [25] used by most user-mode Windows programs contains more than 200 threading and synchronization functions, many with different options and parameters. We wish to wrap the concurrency APIs to capture and control the nondeterminism inherent in the API, without changing the underlying OS scheduler or reimplementing the synchronization primitives of the API. Again, the principle of minimal perturbation is key to making model checking an effective test tool.

Finally, we have the classic problem of state-space explosion. The number of thread interleavings even for small systems can be astronomically large. Scaling model checking to such large systems essentially requires various techniques to focus the systematic enumeration to interesting parts of the state space that are more likely to contain bugs. Recent work [14, 26, 39, 19] has successfully applied model checking to coarse-grained message-passing systems. However, it is not clear how to adapt such techniques for shared-memory multithreaded programs with fine-grained concurrency.

In this paper, we address all these challenges with a tool called CHESS. CHESS enables systematic testing and debugging of three classes of multithreaded programs—user-mode Win32 programs, .NET programs, and Singularity [16] applications. The main contribution of CHESS is that it eliminates the nondeterminism inherent in multithreaded execution, systematically exploring thread schedules in a deterministic manner and, when a bug is discovered, reproducing the exact thread schedule that led to the bug.

To solve the perturbation problem, we first observe that most concurrency tests have been designed to run repeatedly millions of times for the purpose of stress testing. This means that at the end of a concurrency test, all allocated resources are freed and any global state is reset. CHESS leverages this *idempotency* of concurrency tests in its testing methodology (see Figure 1). Given an idempotent test, CHESS repeatedly executes the test in a loop, exploring a different schedule in each iteration. In particular, CHESS does not need to track any program state, including the initial state, making it relatively easy to attach CHESS to an existing test. The test harness either exposes the `TestStartup`, `RunTestScenario`, `TestShutdown` functions to CHESS, or the tester introduces appropriate calls to CHESS in an existing test framework.

The *only* perturbation introduced by CHESS is a thin wrapper layer between the program under test and the concurrency API (see Figure 2). This is required to capture and explore the nondeterminism inherent in the API. We have developed a methodology for writing wrappers that provides enough hooks to CHESS to control the thread scheduling without changing the semantics of the API functions and without modifying the underlying OS, the API implementation, or the system under test. We are also able to map complex synchronization primitives into simpler operations that greatly simplify the process of writing these wrappers. We have validated our methodology by building wrappers for three different platforms—Win32, .NET, and Singularity.

Finally, CHESS uses a variety of techniques, discussed in Section 4 to address the state-explosion problem. The CHESS scheduler is non-preemptive by default, giving it the ability to execute large bodies of code atomically. Of course, a non-preemptive scheduler is at odds with the fact that a real scheduler may preempt a thread at just about any point in its execution. Pragmatically, CHESS explores thread schedules giving priority to schedules with *fewer* preemptions. The intuition behind this search strategy, called *preemption bounding* [27], is that many bugs are exposed in multithreaded programs by a few preemptions occurring in particular places in program execution. To scale to large systems, we had to improve upon preemption bounding in several important ways. First, we not only restrict preemptions at synchronization points (calls to synchronization primitives in the concurrency API) we also only preempt at volatile variables that participate in a data race. Second, we give the tester the ability to control the components to which preemptions are added (and conversely the components which are treated atomically). This is critical for trusted libraries that are known to be thread-safe.

This paper is the culmination of a three-year effort in applying model checking techniques to comprehensively test concurrent programs. Our testing methodology has been validated by the successful integration of CHESS into the test frameworks of several codebases inside Microsoft (§5). Using CHESS, we have found 25 unknown bugs; each bug was found by running an *existing* stress tests, modified to run with smaller number of threads.

Moreover, CHESS produced a consistently reproducible execution for each bug, greatly simplifying the debugging process. Interestingly, one bug found by CHESS was the root cause of more than 30 previously hard-to-debug Heisenbugs.

More importantly, more than half of the bugs were found by Microsoft testers—people other than the authors of this paper. We emphasize this point because there is a huge difference between the robustness and usability of a tool that researchers apply to a code base of their choice and a tool that has been released ”into the wild” to be used daily by testers. CHESS has made this transition successfully. Furthermore, we have demonstrated that CHESS scales to large code bases. It has been applied to Dryad, a distributed execution engine for coarse-grained data-parallel applications [18] and Singularity, a research operating system.

We would like to emphasize that although CHESS is a testing tool dependent on the concurrent test scenario that a tester creates, it has the capability to explore all thread schedules of that test scenario (up to a given a preemption bound). Testers who use CHESS appreciate this concept of *quantified soundness* and, indeed, have themselves pushed us in this direction. In particular, the soundness guarantee is critically important to the use of CHESS in testing a software transactional memory (STM) library. The test harness of this library uses CHESS to compare the behaviors of the STM implementation under different memory consistency models.

To summarize, the main contributions of this paper are the following:

- the first system for integrating model checking into concurrent systems and test frameworks with minimal perturbation;
- techniques for systematic exploration of systems code for fine-grained concurrency with shared memory and multithreading;
- validation of the CHESS tool and its accompany testing and wrapper methodology on three different platforms;
- a substantial number of previously unknown bugs, even in well-tested systems;
- the ability to consistently reproduce crashing bugs with unknown cause.

2 Using CHESS

In this section, we describe how CHESS is used to systematically test a concurrent program. From the point of view of CHESS, a concurrent program is an isolated code module that exports three functions—`TestStartup`, `RunTestScenario`, and `TestShutdown`. The platform on which such a module is expected to run typically exposes a concurrency API for two purposes—for creating a new task that will execute in parallel with existing tasks, and for creating and accessing primitive objects that will be used for synchronization among the tasks. For example, a user-mode Windows application may use the function `CreateThread` to create a new thread and the functions `InitializeCriticalSection`, `EnterCriticalSection`, and `LeaveCriticalSection` to respectively create, acquire, and release a lock object. Intuitively, `TestStartup` is expected to prepare the input for the test by allocating and initializing needed data structures, `RunTestScenario` creates and executes the tasks participating in the concurrency scenario being tested, and `TestShutdown` disposes any allocated resources after the test is finished.

```
TestStartup();
Chess.Quiesce();

while(true){
  RunTestScenario();
  Chess.Quiesce();
  if(Chess.Done()) break;
}

TestShutdown();
```

Figure 1: The CHESS testing methodology. CHESS repeatedly executes the `RunTestScenario` exploring a different schedule in each iteration.

Testing scenarios for real-world concurrent programs invariably have functions that are analogous to `TestStartup`, `RunTestScenario`, and `TestShutdown`. The essence of traditional testing is captured by the following code fragment, whose net effect is to call `TestStartup` once, followed by repeated calls to `RunTestScenario`, followed by a final call to `TestShutdown`.

```
TestStartup();
while(true){
  RunTestScenario();
  if(*) break;
}
TestShutdown();
```

There are two disadvantages of this traditional testing method. First, there is little scheduling variance among the various calls to `RunTestScenario` resulting in poor coverage and possibility of latent bugs that are manifested later possibly after the software is deployed. Second, even if a bug is revealed during a call to `RunTestScenario`, the buggy execution might not replay on a subsequent call making such errors extremely difficult to debug. On the other hand, testing the scenario with CHESS is tantamount to executing the code shown in Figure 1; this is identical to the code executed with traditional testing but for the callbacks `Chess.Quiesce` and `Chess.Done` whose purpose will be explained later in this section. As with traditional testing, each execution of `RunTestScenario` is a legal interleaving of the tasks in the scenario. In addition, CHESS guarantees that every execution of `RunTestScenario` generates a new interleaving and that each such interleaving can be replayed.

To provide these guarantees, CHESS must control the scheduling of the tasks in the test scenario. One way to achieve this control is to modify the scheduler in the runtime platform of the test module. We avoided this option for two main reasons. First, this option contradicts our philosophical goal of perturbing the system under test as little as possible. Second, the deployment of CHESS in practical testing situations would be severely limited if it required its own modified version of the runtime platform. Instead, CHESS controls the scheduling of tasks by instrumenting all functions in the concurrency API of the platform that create tasks and access synchronization objects (Figure 2). The instrumentation wrappers have to be written once for each platform, e.g. WIN32 or CLR, and are reused for all programs written for that platform. We explain the implementation of these wrappers in Section 3.

The callbacks `Chess.Quiesce` and `Chess.Done` are part of the contract between CHESS and the test program that allows CHESS to control the task scheduling using instrumentation wrappers. CHESS requires that every call to `RunTestScenario` is performed only when the system being tested has reached quiescence. Let us refer to the task executing the code in Figure 1 as the main task. Informally, quiescence means that the computation in the system has ceased temporarily. The computation restarts as soon as the main task calls `RunTestScenario`. Thus, CHESS takes the test scenario from one quiescent state to another quiescent state via interleavings, each of which is guaranteed to be distinct. The callback `Chess.Quiesce` asks CHESS to wait until the system under test has reached quiescence. CHESS supports several different modes for reaching quiescence, as described in Section 2.1. In order to determine when to terminate the testing, the callback `Chess.Done` simply asks CHESS if there are any interleavings that remain to be explored.

CHESS makes two important assumptions about the test program. First, the test program should be isolated from any other computation running on the test platform. For testing a user-mode application on the WIN32 platform, this is easily achieved if the application has no tasks other than those created in `TestStartup` and `RunTestScenario`. If such tasks exist, then it is up to the tester to make sure they do not interfere with the tasks in the program being tested. Second, each execution of `RunTestScenario` must be idempotent, that is, the quiescent states at the beginning and end of `RunTestScenario` must be behaviorally equivalent. Idempotence of the test guarantees that there is a unique quiescent state of the program. An interleaving of tasks in `RunTestScenario` then corresponds to a unique non-quiescent state obtained by executing that interleaving from the unique quiescent state. Thus, CHESS to systematically generate all interleavings of `RunTestScenario` without capturing *any* state of the program including the quiescent state. Section 4 describes the details of the search algorithm. The idempotence requirement is usually simple to satisfy; it means that `RunTestScenario` must free all allocated resources and reset any global state accessed by the computation. We have observed that most real-world test programs already satisfy this requirement because they are designed to run repeatedly for a long time.

2.1 Quiescence

The simplest kind of quiescent state is one in which the main task is the only task in the system and it is about to call `RunTestScenario`. In such a test case, a call to `TestStartup` simply initializes data structures without creating any tasks, while a call to `RunTestScenario` creates tasks but waits for all of them to terminate before returning. Detecting such a quiescent state is quite simple.

A more general quiescent state is one in which there are several tasks in the system including the main task. In this state, the main task is about to call `RunTestScenario` and each of the other tasks is blocked on a synchronization object. In such a test case, a call to `TestStartup` not only initializes data structures but also creates worker tasks that typically block on some resource waiting for work. A call to `RunTestScenario` creates a few work items which are processed by the workers until the next quiescent state in which they again block waiting for work. Such a quiescent state is also simple to detect; it is similar to a deadlock state except that the main task is enabled and about to call `RunTestScenario`.

The most general quiescent state handled by CHESS is one in which the main task is about to call `RunTestScenario` and the other tasks are either blocked on a synchronization object or in a livelock. In such a test case, a call to `TestStartup`, in addi-

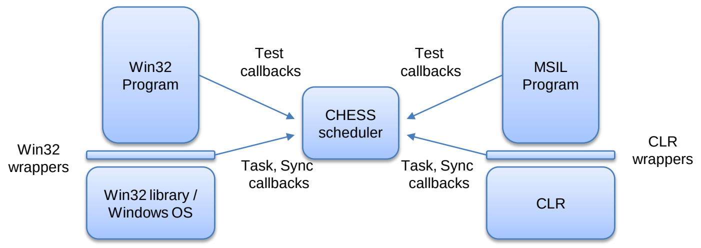

Figure 2: CHESS architecture

tion to initializing data structures and creating worker threads, also creates a few background tasks that wake up periodically to perform some activity such as flushing dirty buffers. These background tasks are different from blocking worker tasks because in the quiescent state, they usually sleep for some fixed time rather that blocking on a synchronization object. CHESS detects such a state by keeping track of elapsed time accumulated by sleep statements. When the total elapsed time reaches a large value, CHESS considers it to be a livelock. Note that this strategy for detecting livelocks works only for fair schedules in which all background threads get opportunity to make progress. Fortunately, fairness is designed into the CHESS scheduler [29]; more details are provided in Section 4.3.

3 Capturing nondeterminism

Large concurrent programs contain a variety of asynchronously executing entities such as threads, threadpool work items, timers, and asynchronous I/O. To enable systematic search, CHESS creates a simple abstraction for each such entity called a task. CHESS abstracts a program as a nondeterministic state transition system in which each transition is executed by a task. Given a state and task enabled in it, executing the task results in a unique new state; nondeterminism arises because in each state more than one task may be enabled and any one of them may be scheduled. The initial state of this system is the unique quiescent state described in Section 2. Starting from this initial state, an execution is obtained by iteratively picking an enabled task and executing it for one step. Given the task abstraction and knowledge of the set of tasks enabled in a state, all such execution can be systematically generated in a straightforward fashion.

The goal of the wrappers is to capture sufficient semantics of the concurrency API to provide the abstraction mentioned above to CHESS. In this section, we described the design of these wrappers for the Win32 API.

Let us first look at the wrapper for a function that creates a new task (Figure 3). The function `CreateThread` creates a new thread that starts the ex-

```
Wrapper_CreateThread(f,arg){
    tid = Chess.Fork();
    Closure c = <f, arg, tid>;
    return Real_CreateThread(
        ChessTaskWrapper, c);
}

ChessTaskWrapper(Closure c){
    Chess.TaskBegin(c.tid);
    DWORD retVal = c.f(c.arg);
    Chess.TaskEnd(c.tid);

    return retVal;
}
```

Figure 3: Task wrappers

ecution by calling the function pointer `f` with the argument `arg`. The wrapper for `CreateThread` informs the CHESS scheduler that a new task is being forked by calling `Chess.Fork`. It also creates a closure containing `f`, `arg`, and the task identifier returned by `Chess.Fork`. Then instead of `f` and `arg`, it passes a wrapper function `ChessTaskWrapper` and the closure to `CreateThread`. `ChessTaskWrapper` retrieves the original function and argument from the closure and brackets the call to the function by calls to `Chess.TaskBegin` and `Chess.TaskEnd`. `Chess.TaskBegin` adds the child task to the set of enabled threads. `Chess.TaskEnd` removes the child task from the set of enabled threads and then schedules another task. Other API functions that create tasks behave similarly by creating closures that bracket appropriate calls to `TaskBegin` and TaskEnd.

We now show the wrappers for typical synchronization functions (Figure 4). The simplest wrappers are for non-blocking synchronization operations such as `InterlockedIncrement`. The wrapper simply introduces a call to the CHESS scheduler indicating that a synchronization variable is about to be accessed. CHESS treats each access of a synchronization

```
Wrapper_InterlockedIncrement(v) {
  Chess.SyncVarAccess(v, RW);
  return InterlockedIncrement(v);
}

Wrapper_EnterCriticalSection(cs) {
  while (true) {
    Chess.SyncVarAccess(cs, ACQUIRE);
    if (TryEnterCriticalSection(cs))
      return;
    Chess.LocalBacktrack();
  }
}

Wrapper_LeaveCriticalSection(cs) {
  Chess.SyncVarAccess(cs, RELEASE);
  LeaveCriticalSection(cs);
}

Wrapper_WaitForSingleObject(
               handle, timeout) {
  while (true) {
    Chess.SyncVarAccess(handle, ACQUIRE);
    ret = WaitForSingleObject(
              handle, 0);
    if !(ret == WAIT_TIMEOUT &&
          timeout == INFINITE)
      return ret;
    Chess.LocalBacktrack();
  }
}
```

Figure 4: Synchronization wrappers

variable as a potential point for introducing a context switch. The wrappers for `EnterCriticalSection` `LeaveCriticalSection` are more interesting; they illustrate how CHESS solves the problem of keeping track of the set of enabled threads in the presence of potentially blocking operations. To illustrate the problem, note that if the wrapper simply called `EnterCriticalSection` directly, then the CHESS scheduler will deadlock if the lock is currently held by another thread. Fortunately, the WIN32 API provides a non-blocking version of `EnterCriticalSection` called `TryEnterCriticalSection` that acquires the lock if available and returns true, and otherwise returns false without blocking. The wrapper calls `TryEnterCriticalSection` and then `Chess.LocalBacktrack` if the lock is not available. This call to `Chess.LocalBacktrack` is matched with the preceding call to `Chess.SyncVarAccess` and indicates that the current task is blocked on the lock being accessed. The CHESS scheduler then removes the current task from the set of enabled tasks, adds it to the set of tasks waiting on this lock, and schedules a different task. Later, when the task holding this lock calls `LeaveCriticalSection`, the call to `Chess.SyncVarAccess` inside the wrapper moves every task waiting on this lock to the set of enabled tasks. In our experience, concurrency APIs invariably provide a "try" version for every blocking operation allowing us to treat every other potentially-blocking operation in the same fashion. For example, the wrapper for `WaitForSingleObject` uses a timeout of 0 milliseconds to create the non-blocking version of the operation.

The implementation of the task abstraction, while mostly straightforward, has a few subtle cases, one of which we describe here. The CHESS scheduler performs cooperative scheduling of the tasks maintaining the invariant that only one task in the program is executing at any given moment. When the currently executing task calls `Chess.TaskEnd`, the CHESS scheduler must not only schedule another task but also let the current task continue to termination. Without proper care, a small window is created where the aforementioned invariant does not hold. This window might create an undesirable race condition if the newly scheduled task attempts to join with the terminating task. The wrapper of the join operation will attempt to join using the "try" version of the operation as described earlier. The attempt will succeed or fail depending on the relative timing of the joining and terminating thread, thereby corrupting the enabled set in the scheduler. To solve this problem, `Chess.TaskEnd` maintains a variable whose value is either null or the identifier of the terminating task. Whenever a task is woken up by the CHESS scheduler, it first checks this variable and waits for the terminating thread to finish if the value is non-null.

A task may also be created when a work item in a threadpool is created using `CreateUserWorkItem` or a timer is created using `CreateTimerQueueTimer`. While threadpools are handled in the same way as threads, timers are more complicated for several reasons. First, a timer is expected to start after a time period specified in the call to create the timer. The CHESS scheduler abstracts real-time and therefore creates a schedulable task immediately. We feel this is justified because programs written for commodity operating systems usually do not depend for their correctness on real-time guarantees. Second, a timer may be periodic in which case it must execute repeatedly after every elapse of a time interval. Continuing our strategy of abstracting away real-time, We handle periodic timers by converting them into aperiodic timers executing the timer function in a loop. Finally, a timer may be cancelled any time after it has been created. We handle cancellation by introducing a canceled bit per timer. This bit is checked just before the timer function starts executing; the bit checked once for an aperiodic timer and repeatedly at the beginning of each loop iteration for a periodic timer. If the bit is set, the timer task is terminated.

In addition to the synchronization operations discussed above, CHESS also handles communication primitives involving FIFO queues such as asynchronous procedure calls (APC) and IO completion ports. Each WIN32 thread has a queue of APCs associated with it. A thread can enqueue a closure to the queue of another thread by using the function `QueueUserAPC`. The APCs in the queue of a thread are executed when the thread enters an alertable wait function such as `SleepEx`. Since the operating system guarantees FIFO execution of the APCs, it suffices in the wrappers to pass on the call to the actual function. The treatment of IO completion ports is similar again because the completion packets in the queue associated with an IO completion port are delivered in FIFO order.

Asynchronous IO, usually in conjunction with events, APCs, or IO completion ports, also introduces nondeterminism in concurrent programs. To understand the problem, consider the function

```
BOOL WINAPI ReadFileEx(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPOVERLAPPED lpOverlapped,
  LPOVERLAPPED_COMPLETION_ROUTINE routine);
```

This function reads a file asynchronously and enqueues a closure to the queue of APCs of the calling thread upon completion. This behavior is inherently nondeterministic and dependent upon the relative speed of the program and the IO operation executing in the kernel. To expose this nondeterminism, we create a separate task mapped onto a freshly created thread for each call to `ReadFileEx`. This task performs the read operation synchronously and when the read is finished enqueues the completion routine to the caller of `ReadFileEx` using the function `QueueUserAPC` mentioned earlier.

API	No. of wrappers	LOC
Win32	134	2512
.NET	64	1270
Singularity	37	876

Table 1: Complexity of writing wrappers for the APIs. The LOC does not count the boiler-plate code that is automatically generated from API specifications.

3.1 Hooking the wrappers

Once the wrappers are defined, we use various mechanisms to dynamically intercept calls to the real API functions and forward them to the wrappers. For Win32 programs, we use dll shimming techniques to overwrite the import address table of the program under test. For .NET programs, we used a generalized CLR profiler [8] that replaces calls to API functions with calls to the wrappers at JIT time. Finally, for the Singularity API, we used static IL rewriter [1] to make the modifications. Table 1 shows the complexity and the amount of effort required to write the wrappers.

4 Exploring nondeterminism

The previous section describes how CHESS obtains control at scheduling points before the synchronization operations of the program and how CHESS determines the set of enabled threads at each scheduling point. This section describes how CHESS systematically drives the test along different schedules

4.1 Basic scheduler operation

To maintain absolute control of the thread interleaving, CHESS allows only one thread to execute at a time, essentially emulating the execution of the test on a uniprocessor. The reason for this design decision is that two or more threads running simultaneously could create data-races whose outcome CHESS cannot control. By systematically exploring the different interleavings, CHESS still has the capability of driving the test along any sequentially-consistent execution of the test possible on a multi-processor.

CHESS repeatedly executes the same test driving each iteration of the test through a different schedule. In each iteration, the scheduler works in three phases: replay, record, and search.

In the replay phase, the scheduler replays a sequence of scheduling choices from a trace file. This trace file is empty in the first iteration, and contains a partial schedule generated by the search phase from the previous iteration. Once the replay is done, CHESS switches to the record phase. In this phase, the scheduler behaves as a fair, nonpreemptive scheduler. It schedules a thread till the thread yields the processor either by completing its execution, or blocking on a synchronization operation, or calling one of the yielding operations such as `sleep()`. On a yield, the scheduler picks the next thread to execute based on priorities that the scheduler maintains to guarantee fairness (§4.3). Also, the scheduler extends the partial schedule in the trace file by recording the thread scheduled at each schedule point together with the set of threads enabled at each point. The latter provides the set of choices that are available but not taken in this test iteration.

When the test terminates, the scheduler switches to the search phase. In this phase, the scheduler uses the enabled information at each schedule point to determine the schedule for the next iteration. Picking the next interesting schedule among the myriad choices available is a challenging problem, and the algorithms in this phase are the most complicated and computationally expensive components of CHESS (§4.4).

The subsequent three subsections describe the key challenges in each of the three phases.

4.2 Dealing with imperfect replay

Unlike stateful model checkers [37, 26] that are able to checkpoint and restore program state, CHESS relies on its ability to replay a test iteration from the beginning to bring a program to particular state. As has been amply demonstrated in previous work [10, 20, 23, 3], perfect replay is impossible without significant engineering and heavy-weight techniques that capture *all* sources of nondeterminism. In CHESS, we have made a conscious decision to not rely on perfect replay capability. Instead, CHESS can robustly handle extraneous nondeterminism in the system, albeit at the cost of the exhaustiveness of the search.

The CHESS scheduler can fail to replay a trace in the following two cases. First, the thread to schedule at a scheduling point is disabled. This happens when a particular resource, such as a lock, was available at this scheduling point in the previous iteration but is currently unavailable. Second, a scheduled thread performs a different sequence of synchronization operations than the one present in the trace. This can happen due to a change in the program control flow resulting from a program state not reset at the end of the previous iteration.

When the scheduler detects such extraneous nondeterminism, the default strategy is to give up replay and immediately switch to the record phase. This ensures that the current test runs to completion. The scheduler then tries to replay the same trace once again, in the hope that the nondeterminism is transient. On a failure, the scheduler continues the search beyond the current trace. This essentially prunes the search space at the point of nondeterminism. To alleviate this loss of coverage, CHESS has special handling for the most common sources of nondeterminism that we encountered in practice.

Lazy-initialization: Almost all systems we have encountered perform some sort of lazy-initialization, where the program initializes a data-structure the first time the structure is accessed. If the initialization performs synchronization operations, CHESS would fail to see these operations in subsequent iterations. To avoid this nondeterminism, CHESS "primes the pump" by running a few iterations of the tests as part of the startup in the hope of initializing all data-structures before the systematic exploration. The downside, of course, is that CHESS loses the capability to interleave the lazy-initialization operations with other threads, potentially missing some bugs.

Interference from environment: The system under test is usually part of a bigger environment that could be concurrently performing computations during a CHESS run. For instance, when we run CHESS on Dryad we bring up the entire Cosmos system (of which Dryad is a part) as part of the startup. While we do expect the tester to provide sufficient isolation between the system under test and its environment, it is impractical to require complete isolation. As a simple example, both Dryad and Cosmos share the same logging module, which uses a lock to protect a shared log buffer. When a Dryad thread calls into the logging module, it could potentially interfere with a Cosmos thread that is currently holding the lock. CHESS handles this as follows. In the replay phase, the interference will result in the current thread being disabled unexpectedly. When this happens, the scheduler simply retries scheduling the current thread a few times before resorting to the default solution mentioned above. If the interference happens in record mode, the scheduler might falsely think that the current thread is disabled when it can actually make progress. In the extreme case, this can result in a false deadlock if no other threads are enabled. To distinguish this from a real deadlock, the CHESS scheduler repeatedly tries scheduling the threads in a deadlock state to ensure that they are indeed unable to make progress.

Nondeterministic calls: The final source of nondeterminism arises from calls to `random()` and `gettimeofday()`, which can return different values at different iterations of the test. We expect the tester to avoid making such calls in the test code. However, such calls might still be present in the system under test that the tester has no control over. We determinize calls to `random()` by simply reseeding the random number generate to a predefined constant at the beginning of each test iteration. On the other hand, we do not determinize time functions such as `gettimeofday()`. Most of the calls to time functions do not affect the control flow of the program. Even when they do, it is to periodically refresh some state in the program. In this case, the default strategy of retrying the execution works well in practice.

4.3 Ensuring fair schedules

All reasonable operating systems schedulers are *fair* — they use various queueing policies and priority schemes to ensure that no thread is starved forever. Moreover, most concurrent programs implicitly assume scheduler fairness for correct behavior. For instance, spin-loops are very common in programs. Such loops would not terminate if the scheduler continuously starves the thread that is supposed to set the condition of the loop. Similarly, some threads perform computation till they receive a signal from another thread. An unfair scheduler is not required to eventually schedule the signalling thread.

On such programs, it is essential to restrict systematic enumeration to only fair schedules. Otherwise, a simplistic enumeration strategy will spend a significant amount of time exploring unfair schedules. [3] Moreover, errors found on these interleavings will appear uninteresting to the user as she would consider these interleavings impossible or unlikely in practice. Finally, fair scheduling is essential for the CHESS testing methodology (§2) for a subtle reason. CHESS relies on the termination of the idempotent test scenario to bring the system to the initial state. Most tests will not terminate on unfair schedules, and with no other checkpointing capability, CHESS will not be able to bring the system to the initial state.

Of course, it is unreasonable to expect CHESS to enumerate *all* fair schedules. For most programs, there are infinitely many fair interleavings. To see this, any schedule that unrolls a spin-loop arbitrary but finite number of times is still fair. Instead, CHESS makes a pragmatic choice to focus on interleavings that are likely to occur in practice. The fair scheduler, described in detail in a simultaneous publication [29], gives lower priority to threads that yield the processor, either by calling a yielding function such as `Thread.yield` or by sleeping for a finite time. This immediately restricts the CHESS scheduler to only schedule enabled threads with a higher priority, if any. Under the condition that a thread yields only when it is unable to make progress, this fair scheduler is guaranteed to not miss any safety error [29].

As an interesting side-effect of fair scheduling, CHESS automatically gets the ability to detect liveness violations as well. This is because all liveness properties can be reduced to fair-termination [36]. Essentially, to check if "something good eventually happens", the user writes a test that terminates only when the "good" condition happens. If the program violates this property, the CHESS scheduler will eventually produce a fair schedule under which the test does not terminate. The user identifies such nonterminating behavior by setting an abnormally high bound on the length of the execution.

4.4 Tackling state-space explosion

State-space explosion is the bane of model checking. Given a program with n threads that execute k atomic steps in total, it is very easy to show that the number of thread interleavings grows astronomically as n^k. The exponential in k is particularly troublesome. It is normal for realistic tests to perform thousands (if not more) synchronization operations in a single run of the test. To be effective in such large state spaces, it is essentially to focus on interesting and potentially bug-yielding interleavings. In the previous section, we described how fairness helps CHESS to focus only on fair schedules. We discuss other key strategies below.

4.4.1 Inserting preemptions prudently

In recent work [27], we showed that bounding the number of preemptions is a very good search strategy when systematically enumerating thread schedules. Given a program with n threads that execute k steps in total, the number of interleavings with c preemptions grows with k^c. Informally, this is because, once the scheduler has picked c out of the k possible places to preempt, the scheduler is forced to schedule the resulting chunks of execution atomically. (See [27] for a more formal argument.) On the other hand, we expect that most concurrency bugs happen [24] because of few preemptions happening at the right places. In our experience with CHESS, we have been able to reproduce very serious bugs using just 2 preemptions.

On applying to large systems, however, we found that preemption bounding alone was not sufficient to reasonably reduce the size of the state space. To solve this problem, we had to *scope* preemptions to code regions of interest, essentially reducing k. First, we realized that a significant portion of the synchronization operations occur in system functions, such as the C run time. Similarly, many of the programs use underlying base libraries which can be safely assumed to be thread-safe. CHESS does not insert preemptions in these modules, thereby gaining scalability at the cost of missing bugs resulting from adverse interactions between these modules and the rest of the system.

Second, we observed that a large number of the synchronizations are due to accesses to volatile variables. Here we borrow a crucial insight from Bruening and Chapin [4] (see also [35]) — if accesses to a particular volatile variable are always ordered by accesses through other synchronization, then it is not necessary to interleave at these points.

4.4.2 Capturing states

One advantage of stateful model checkers [37, 26] is their ability to cache visited program states. This prevents them from redundantly exploring the same program state more than once, a huge gain in efficiency. The downside is that precisely capturing the state of a large system is onerous [26, 39]. Avoiding this complication was the main reason for designing CHESS to be stateless.

However, we obtain some advantages of state-caching by observing that we can use the trace used to reach the current state from the initial state as a representation of the current state. Specifically, CHESS maintains for each execution a partially-ordered *happens-before* graph over the set of synchronization operations in the execution. Two executions that generate the same happens-before graph only differ in the order of independent synchronizations operations. Thus, a program that is data-race free will be at the same program state on the two executions. By caching the happens-before graphs of visited states, CHESS avoids exploring the same state redundantly. This reduction has the same reduction as a partial-order reduction method called sleep-sets [13] but combines well with preemption bounding [28].

5 Evaluation

In this section, we describe our experience in applying CHESS to several large industry-scale systems.

5.1 Brief description of benchmarks

Table 2 describes the systems on which CHESS has been run on. We briefly describe each of these systems to emphasize the range of systems CHESS is applicable to. Also, the integration of CHESS with the first five systems in Table 2 was done by the users of CHESS, with some help from the authors.[4]

PLINQ [9] is an implementation of the declarative data-parallel extensions to the .NET framework. CDS (Concurrent Data Structures) is a library that implements efficient concurrent data structures. STM is an implementation of software transactional memory inside Microsoft. TPL and ConcRT are two libraries that provide efficient work-stealing implementations of user-level tasks, the former for .NET programs and the latter for C and C++ programs. CCR is the concurrency and coordination runtime [6], which is part of Microsoft Robotics Studio Runtime. Dryad is a distributed execution engine for coarse-grained data-parallel applications [18]. Finally, Singularity [16] is a research operating system.

Programs	LOC	max Threads	max Synch.	max Preemp.
PLINQ	23750	8	23930	2
CDS	6243	3	143	2
STM	20176	2	75	4
TPL	24134	8	31200	2
ConcRT	16494	4	486	3
CCR	9305	3	226	2
Dryad	18093	25	4892	2
Singularity	174601	14	167924	1

Table 2: Characteristics of input programs to CHESS

5.2 Test scenarios

In all these programs, except Singularity, we took existing stress tests and modified them to run with CHESS. Most of the stress tests were originally written to create large number of threads. We modified them to run with fewer threads, for two reasons. Due to the systematic exploration of CHESS, one no longer needs a large number of threads to create scheduling variety. Also, CHESS scales much better when there are few threads. We validate this reduction in the next section.

Other modifications were required to “undo” code meant to create scheduling variety. We found that testers pepper the code with random calls to yield functions. With CHESS, such tricks are no longer necessary. On the other hand, such calls impede the coverage achieved with CHESS as the scheduler (§4.3) assumes that a yielding thread is not able to make progress, and accordingly assigns it a low scheduling priority. Another common paradigm in the stress tests was to randomly choose between a variety of inputs with probabilities to mimic real executions. In this case we had to refactor the test to concurrently generate the inputs so that CHESS interleaved their processing. These modifications would not be required if system developers and testers were only concerned about creating interesting concurrency scenarios, and relied on a tool like CHESS to create scheduling variety.

Many of the stress tests were idempotent by design, once CHESS was able to handle lazy-initialization of

Programs	Total	Failure / Bug		
		Unk/Unk	Kn/Unk	Kn/Kn
PLINQ	1		1	
CDS	1		1	
STM	2			2
TPL	9	9		
ConcRT	4	4		
CCR	2	1	1	
Dryad	7	7		
Singularity	1		1	
Total	27	21	4	2

Table 3: Bugs found with CHESS, classified on whether the failure and the bug were known or unknown.

datastructures (§4.2). Common failures of idempotence were bugs in test harnesses, such as handle leaks, that were easy to fix. These bugs are not reported in Table 3. There was one case where a test was not idempotent by design — the tester left the state "corrupted" for subsequent tests. While CHESS could handle this scenario, by running the corrupting test once as part of the startup, we have not tried this at the time of writing this paper.

Finally, we used CHESS to systematically test the *entire* boot and shutdown sequence of the Singularity operating system [16]. Singularity OS has the ability to run as a user process on top of the Win32 API. This functionality is essentially provided by a thin software layer that emulates necessary hardware abstractions on the host. This mechanism alone was sufficient to run the entire Singularity OS on CHESS with little modification. The only changes required were to expose certain low-level synchronization primitives at the hardware abstraction layer to CHESS and to call the shutdown immediately after the boot without forking the login process. These changes involved ˜300 lines in 4 files.[5]

5.3 Validating CHESS against stress-testing

A common objection to the CHESS methodology is the belief that one cannot find concurrency errors with a small number of threads. This belief is a direct consequence of the painful experience people have of their concurrent systems failing under stress. A central hypothesis of CHESS is that errors in complex systems occur due to *complex* interleavings of *simple* scenarios. In this section, we validate this hypothesis. Recent work [24] also suggests a similar hypothesis.

Table 3 shows the bugs that CHESS has found so far on the systems described in Table 2. Table 3 distinguishes between *bugs* and *failures*. A bug is an error in the program and is associated with the specific line(s) in the code containing the error. A failure is a, possibly non-deterministic, manifestation of the bug in a test run, and is associated with the test case that fails. Thus, a single bug can cause multiple failures. Also, as is common with concurrency bugs, a known failure does not necessarily imply a known bug — the failure might be too hard to reproduce and debug.

Table 3 only reports the number of distinct bugs found by CHESS. The number of failures exceeds this number. As an extreme case, the PLINQ bug is a race-condition in a core primitive library that was the root-cause for over 30, apparently unrelated, test failures. CHESS found a total of 27 bugs in all of the programs, of which 25 were previously unknown. Of these 25 bugs, 21 did not manifest in existing stress runs over many months. The other 4 bugs were those with known failures but unknown cause. In these cases, the tester pointed us to existing stress tests that failed occasionally but she was not able to debug the problem. CHESS was able to reproduce the failure, within minutes in some cases, with less than 10 threads and two preemptions. Similarly, CHESS was able to reproduce two failures that the tester had previously (and painfully) debugged on the STM library. So far, CHESS has succeeded in reproducing every stress-test failure reported to us.

5.4 Description of two bugs

We describe two bugs that CHESS was able to find in this section.

5.4.1 PLINQ bug

CHESS discovered a bug in PLINQ that is due to an incorrect use of `LiteEvents`, a concurrency primitive implemented in the library. A LiteEvent is an optimization over kernel events that does not require a kernel transition in the common case. It was originally designed to work between exactly two threads — one that calls `Set` and one that calls `Wait` followed by a `Dispose`. Figure 5 contains a simplified version of the code. A lock protects the subtle race that occurs between a `Set` that gets preempted right after an update to `state` and `Dispose`. However, the lock does not protect a similar race between a `Wait` and a `Dispose`. This is because, the designer did not intend to use LiteEvents with multiple waiters. However, the PLINQ code did not obey this restriction and CHESS promptly reported this error with just one preemption. As with many concurrency bugs, the fix is easy once the bug is identified.

5.4.2 Singularity bug

CHESS is able to check for liveness properties and it checks the following property by default [29]: every

```
LiteEvent::Set(){
   state = SIGNALLED;
   if(kevent)
      lock_and_set_kevent();
}

LiteEvent::Wait(){
   if(state == SIGNALLED)
     return;
   alloc_kevent();
   //BUG:kevent can be 0 here
   kevent.wait();
}

LiteEvent::Dispose(){
   lock_and_delete_kevent();
}
```

Figure 5: A race condition exposed when LiteEvents are used with multiple waiters.

thread either makes progress towards completing its function or yields the processor. This property detected an incorrect spin loops that never yields in the boot process of Singularity. Figure 6 shows the relevant code snippet. A thread spins in a loop till it receives a particular notification from the underlying hardware. If the notification has a different `id` and thus does not match what it is waiting for, the thread retries without dispatching the notification. On first sight, it appears that this loop yields by calling the `Halt` function. However, `Halt` will return without yielding, if *any* notification is pending. The boot thread, thus, needlessly spins in the loop till its time-slice expires, starving other threads, potentially including the one that is responsible for receiving the current notification.

The developers responsible for this code immediately recognized this scenario as a serious bug. In practice, this bug resulted in "sluggish I/O behavior" during the boot process, a behavior that was previously known to occur but very hard to debug. This bug was fixed within a week. The fix involved changing the entire notification dispatch mechanism in the hardware abstraction layer.

```
Dispatch(id){
  while (true) {
     Platform.Halt();
     // We wake up on any notification.
     // Dispatch only our id

     if ( PendingNotification(id) ) {
        DispatchNotification(id);
        break;
     }
  }
}

Platform.Halt(){
  if(AnyNotification())
    return;
  Sleep();
}
```

Figure 6: Violation of the good samaritan property. Under certain cases, this loop results in the thread spinning idly till time-slice expiration.

6 Related work

This paper is concerned with systematic exploration of the behaviors of executable concurrent programs. This basic idea is not new and has previously occurred in the research areas of software testing and model checking. In contrast to this previous work, this paper is the first to demonstrate the applicability of such systematic exploration to large systems with little perturbation to the program, the runtime, and the test infrastructure.

Carver and Tai [5] proposed repeatable deterministic testing by running the program with an input and explicit thread schedule. The idea of systematic generation of thread schedules came later under the rubric of teachability testing [17]. Recent work in this area includes RichTest [22] which performs efficient search using on-the-fly partial-order reduction techniques, and ExitBlock [4] which observes that context switches only need to be introduced at synchronization accesses, an idea we borrow.

In the model checking community, the idea of applying state exploration directly to executing concurrent programs can be traced back to the Verisoft model checker [15], which is similar to the testing approach in that it enumerates thread schedules rather than states. There are a number of other model checkers, such as Java Pathfinder [37], Bogor [32], CMC [26], and MaceMC [19], that attempt to capture and cache the visited states of the program. CHESS is designed to be stateless; hence it is similar in spirit to the work on systematic testing and stateless model checking. What sets CHESS apart from the previous work in this area is its focus on detecting both safety and liveness violations on large multithreaded systems programs. Effective safety and liveness testing of such programs requires novel techniques—quiescence detection, preemption bounding, and fair scheduling—absent from the previous work.

ConTest [12] is a lightweight testing tool that attempts to create scheduling variance without resorting to systematic generation of all executions. In contrast, CHESS obtains greater control over thread scheduling to offer higher coverage guarantees and better reproducibility.

The ability to replay a concurrent execution is a fundamental building block for CHESS. The problem of deterministic replay has been well-studied [21, 33, 38, 10, 2, 20, 11]. The goal of CHESS is to not only capture the nondeterminism but also to systematically explore it. Also, to avoid the inherent cost of deterministic replay, we have designed CHESS to robustly handle some nondeterminism at the cost of test coverage.

The work on dynamic data-race detection, e.g., [34, 30, 40], is orthogonal and complementary to the work on systematic enumeration of concurrent behaviors. A tool like CHESS can be used to systematically generate dynamic executions, each of which can then be analyzed by a data-race detector.

References

[1] Abstract IL — http://research.microsoft.com/projects/ilx/absil.aspx.

[2] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *VEE 06: International Conference on Virtual Execution Environments* (2006), ACM, pp. 154–163.

[3] BOOTHE, B. Efficient algorithms for bidirectional debugging. In *PLDI 00: Programming Language Design and Implementation* (2000), pp. 299–310.

[4] BRUENING, D., AND CHAPIN, J. Systematic testing of multithreaded Java programs. Tech. Rep. LCS-TM-607, MIT/LCS, 2000.

[5] CARVER, R. H., AND TAI, K.-C. Replay and testing for concurrent programs. *IEEE Softw. 8*, 2 (1991), 66–74.

[6] Concurrency and Coordination Runtime — http://msdn.microsoft.com/en-us/library/bb648752.aspx.

[7] CLARKE, E., AND EMERSON, E. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs* (1981), LNCS 131, Springer-Verlag, pp. 52–71.

[8] The CLR profiler — http://msdn.microsoft.com/en-us/library/ms979205.aspx.

[9] DUFFY, J. A query language for data parallel programming: invited talk. In *DAMP* (2007), p. 50.

[10] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI 02: Operating Systems Design and Implementation* (2002).

[11] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *VEE '08* (New York, NY, USA, 2008), ACM, pp. 121–130.

[12] EDELSTEIN, O., FARCHI, E., GOLDIN, E., NIR, Y., RATSABY, G., AND UR, S. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience 15*, 3–5 (2003), 485–499.

[13] GODEFROID, P. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* LNCS 1032. Springer-Verlag, 1996.

[14] GODEFROID, P. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages* (1997), ACM Press, pp. 174–186.

[15] GODEFROID, P. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages* (1997), ACM Press, pp. 174–186.

[16] HUNT, G. C., AIKEN, M., FHNDRICH, M., HODSON, C. H. O., LARUS, J. R., LEVI, S., STEENSGAARD, B., TARDITI, D., AND WOBBER, T. Sealing OS processes to improve dependability and safety. In *Proceedings of the EuroSys Conference* (2007), pp. 341–354.

[17] HWANG, G., TAI, K., AND HUNAG, T. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering 5*, 4 (1995), 493–510.

[18] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the EuroSys Conference* (2007), pp. 59–72.

[19] KILLIAN, C. E., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI 07: Networked Systems Design and Implementation* (2007), pp. 243–256.

[20] KONURU, R. B., SRINIVASAN, H., AND CHOI, J.-D. Deterministic replay of distributed java applications. In *IPDPS 00: International Parallel and Distributed Processing Symposium* (2000), pp. 219–228.

[21] LEBLANC, T. J., AND MELLOR-CRUMMEY, J. M. Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36*, 4 (1987), 471–482.

[22] LEI, Y., AND CARVER, R. H. Reachability testing of concurrent programs. *IEEE Trans. Software Eng. 32*, 6 (2006), 382–403.

[23] LIU, X., LIN, W., PAN, A., AND ZHANG, Z. Wids checker: Combating bugs in distributed systems. In *NSDI 07: Networked Systems Design and Implementation* (2007).

[24] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS 08: Architectural Support for Programming Languages and Operating Systems* (2008).

[25] Windows API reference — http://msdn.microsoft.com/en-us/library/aa383749(vs.85).aspx.

[26] MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D., AND DILL, D. L. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation* (2002), pp. 75–88.

[27] MUSUVATHI, M., AND QADEER, S. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 07: Programming Language Design and Implementation* (2007), pp. 446–455.

[28] MUSUVATHI, M., AND QADEER, S. Partial-order reduction for context-bounded state exploration. Tech. Rep. MSR-TR-2007-12, Microsoft Research, 2007.

[29] MUSUVATHI, M., AND QADEER, S. Fair stateless model checking. In *PLDI 08: Programming Language Design and Implementation* (2008).

[30] O'CALLAHAN, R., AND CHOI, J.-D. Hybrid dynamic data race detection. In *PPOPP 03: Principles and Practice of Parallel Programming* (2003), pp. 167–178.

[31] QUEILLE, J., AND SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In *Fifth International Symposium on Programming*, LNCS 137. Springer-Verlag, 1981, pp. 337–351.

[32] ROBBY, DWYER, M., AND HATCLIFF, J. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering* (2003), ACM, pp. 267–276.

[33] RUSSINOVICH, M., AND COGSWELL, B. Replay for concurrent non-deterministic shared-memory applications. In *PLDI '96: Programming language design and implementation* (1996), pp. 258–266.

[34] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems 15*, 4 (1997), 391–411.

[35] STOLLER, S. D., AND COHEN, E. Optimistic synchronization-based state-space reduction. In *TACAS 03* (2003), LNCS 2619, Springer-Verlag, pp. 489–504.

[36] VARDI, M. Y. Verification of concurrent programs: The automata-theoretic framework. *Annals of Pure and Applied Logic 51*, 1-2 (1991), 79–98.

[37] VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. Model checking programs. In *ASE 00: Automated Software Engineering* (2000), pp. 3–12.

[38] XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News 31*, 2 (2003), 122–135.

[39] YANG, J., TWOHEY, P., ENGLER, D. R., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems 24*, 4 (2006), 393–423.

[40] YU, Y., RODEHEFFER, T., AND CHEN, W. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP 05: Symposium on Operating Systems Principles* (2005), pp. 221–234.

Notes

[1]In this paper, we restrict our attention to single process programs consisting of multiple threads.

[2]This is even true for programs written in languages such as Java and C#, which simply provide syntactic sugar over an existing concurrency API.

[3]The set of unfair schedules far outnumber the set of fair schedules for reasonable programs. Theoretically, for a program that terminates on fair schedules, the set of fair schedulers is countably infinite while the set of unfair schedules is uncountably infinite.

[4]The authors would like to thank Chris Dern, Rahul Patil, Susan Wo, Raghu Simha, Pooja Nagpal, and Roy Tan for being immensely patient first users of CHESS.

[5]The authors would like to thank Chris Hawblitzel for these changes