Finding and Reproducing Heisenbugs in Concurrent Programs

Madanlal Musuvathi
Microsoft Research

Shaz Qadeer
Microsoft Research

Thomas Ball
Microsoft Research

Gerard Basler
ETH Zurich

Piramanayagam Arumuga Nainar
University of Wisconsin, Madison

Iulian Neamtiu
University of California, Riverside

Abstract

Concurrency is pervasive in large systems. Unexpected interference among threads often results in "Heisenbugs" that are extremely difficult to reproduce and eliminate. We have implemented a tool called CHESS for finding and reproducing such bugs. When attached to a program, CHESS takes control of thread scheduling and uses efficient search techniques to drive the program through possible thread interleavings. This systematic exploration of program behavior enables CHESS to quickly uncover bugs that might otherwise have remained hidden for a long time. For each bug, CHESS consistently reproduces an erroneous execution manifesting the bug, thereby making it significantly easier to debug the problem. CHESS scales to large concurrent programs and has found numerous bugs in existing systems that had been tested extensively prior to being tested by CHESS. CHESS has been integrated into the test frameworks of many code bases inside Microsoft and is used by testers on a daily basis.

1 Introduction

Building concurrent systems is hard. Subtle interactions among threads and the timing of asynchronous events can result in concurrency errors that are hard to find, reproduce, and debug. Stories are legend of so-called "Heisenbugs" [18] that occasionally surface in systems that have otherwise been running reliably for months. Slight changes to a program, such as the addition of debugging statements, sometimes drastically reduce the likelihood of erroneous interleavings, adding frustration to the debugging process.

The main contribution of this paper is a new tool called CHESS for systematic and deterministic testing of concurrent programs. When attached to a concurrent program, CHESS takes complete control over the scheduling of threads and asynchronous events, thereby capturing all the interleaving nondeterminism in the program. This provides two important benefits. First, if an execution results in an error, CHESS has the capability to reproduce the erroneous thread interleaving. This substantially improves the debugging experience. Second, CHESS uses systematic enumeration techniques [10, 37, 17, 31, 45, 22] to force every run of the program along a different thread interleaving. Such a systematic exploration greatly increases the chances of finding errors in existing tests. More importantly, there is no longer a need to artificially "stress" the system, such as increasing the number of threads, in order to get interleaving coverage — a common and recommended practice in testing concurrent systems. As a result, CHESS can find in simple configurations errors that would otherwise only show up in more complex configurations.

To build a systematic testing tool for real-world concurrent programs, several challenges must be addressed. First, such a tool should avoid perturbing the system under test, and be able to test the code as is. Testers often do not have the luxury of changing code. More importantly, whenever code is changed for the benefit of testing, the deployed bits are not being tested. Similarly, we cannot change the operating system or expect testers to run their programs in a specialized virtual machine. Therefore, testing tools should easily integrate with existing test infrastructure with no modification to the system under test and little modification to the test harness.

Second, a systematic testing tool must accomplish the nontrivial task of capturing and exploring all interleaving nondeterminism. Concurrency is enabled in most systems via complex concurrency APIs. For instance, the Win32 API [30] used by most user-mode Windows programs contains more than 200 threading and synchronization functions, many with different options and parameters. The tool must understand the precise semantics of these functions to capture and explore the nondeterminism inherent in them. Failure to do so may result in lack of reproducibility and the introduction of false

behaviors.

Finally, a systematic testing tool must explore the space of thread interleavings intelligently, as the set of interleavings grows exponentially with the number of threads and the number of steps executed by each thread. To effectively search large state spaces, a testing tool must not only reduce the search space by avoiding redundant search [16] but also prioritize the search towards potentially erroneous interleavings [32].

The CHESS tool addresses the aforementioned challenges with a suite of innovative techniques. The *only* perturbation introduced by CHESS is a thin wrapper layer, introduced with binary instrumentation, between the program under test and the concurrency API (Figure 1). This layer allows CHESS 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 [30], .NET [29], and Singularity [19].

CHESS uses a variety of techniques to address the state-explosion problem inherent in analyzing concurrent programs. 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 will not model the behavior 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* [32], 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 improve upon preemption bounding in several important ways. First, in addition to introducing preemptions at calls to synchronization primitives in the concurrency API, we also allow preemptions at accesses to volatile variables that participate in a data race. Second, we provide 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.

Today, CHESS works on three platforms and has been integrated into the test frameworks of several product teams. CHESS has been used to find numerous bugs, of which more than half 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. We have reproduced all stress test crashes reported to us so far; as an additional benefit, these crashes were reproduced in small configurations, thereby greatly reducing the debugging effort. 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 [21] and Singularity, a research operating system [19].

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

- a tool for taking control of scheduling through API-level shimming, allowing the systematic testing of concurrent programs with minimal perturbation;

- techniques for systematic exploration of systems code for fine-grained concurrency with shared memory and multithreading;

- validation of the tool and its architecture and wrapper methodology on three different platforms;

- demonstrating that the tool can find a substantial number of previously unknown bugs, even in well-tested systems;

- the ability to consistently reproduce crashing bugs with unknown cause.

The paper is organized as follows. Section 2 gives an example of how we used CHESS to quickly reproduce a Heisenbug in production code. Section 3 explains the design decisions behind the CHESS scheduler, its basic abstractions, and the wrappers that allow CHESS to effectively control interleaving nondeterminism. Section 4 describes the search strategies that CHESS employs in order to systematically test a concurrent program. Section 5 provides an experimental evaluation of CHESS on several concurrent systems from Microsoft. Finally, section 6 reviews related work.

2 Example

In this section, we describe how we used CHESS to deterministically reproduce a Heisenbug in CCR [8], a .NET library for efficient asynchronous concurrent programming. The code is a client of .NET's System.Threading library, from which it primarily uses monitors, events, and interlocked operations.

The creator of the library reported that a nightly test run had failed. The entire test run consists of many
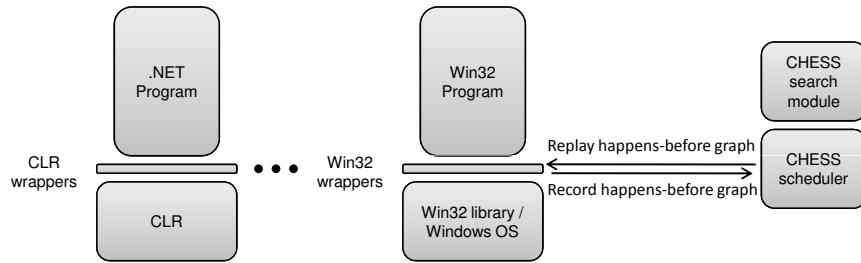
Figure 1: CHESS architecture

smaller unit concurrency tests, each testing some concurrency scenario and each repeatedly executed hundreds to millions of times. The failing test (which did not terminate) previously had not failed for many months, indicating the possibility of a rare race condition.

Our first step was to isolate the offending unit test, which was simply a matter of commenting out the other (passing) tests. The offending test was run several hundred times by the test harness. We changed the harness so it ran the test just once, since CHESS will perform the task of running the test repeatedly anyway. Apart from this change, we did not make any other changes to the test. The entire process took less than thirty minutes, including the time to understand the test framework. As expected, when we ran the modified test repeatedly under the control of the .NET CLR scheduler, it always terminated with success.

We first ran the test under CHESS. In just over twenty *seconds*, CHESS reported a deadlock after exploring 6737 different thread interleavings. It is worth noting that running CHESS a second time produces exactly the same result (deadlock after 6737 interleavings), as the CHESS search strategy is deterministic. However, by simply telling CHESS to use the last recorded interleaving, written out to disk during the execution, CHESS reproduces the deadlock scenario immediately.

We then ran CHESS on the offending schedule under the control of a standard debugger. When CHESS is in control of thread scheduling, it executes the schedule one thread at a time; consequently, single-stepping in the debugger was completely predictable (which is not true in general for a multithreaded program). Furthermore, CHESS provides hooks that allow us to run a thread till it gets preempted. By examining these preemption points, we easily identified the source of the bug, described below.

In this program, the worker threads communicate with each other by exchanging tasks through CCR ports [8], a generic message-passing mechanism in CCR. The bug is due to a race condition in the implementation of a two-

phase commit protocol required to atomically process a set of tasks belonging to a port. The bug manifests when a worker A registers to process a set of tasks in a port while a worker B cancels all the tasks in the port. In the erroneous schedule, worker A is preempted after it successfully registers on the port, but before it receives the tasks in the port. Worker B proceeds with the cancellation but gets preempted *after* processing the first task. However, the code fails to set an important field in the port to indicate that cancellation is in progress. As a result, when worker A resumes, it erroneously proceeds to process both tasks, violating the atomicity of the cancellation. This violation results in an exception that leads to a deadlock.

To summarize, with CHESS we were able to reproduce in thirty seconds a Heisenbug that appeared just once in months of testing of a fairly robust concurrency library. This bug requires a specific interleaving between two threads when accessing a port with two or more tasks. Without CHESS, such a complicated concurrency error would have taken several days to weeks to consistently reproduce and debug. It took one of the authors less than an hour to integrate CHESS with the existing test framework, isolate the test that failed from a larger test suite, and deterministically reproduce the Heisenbug. In addition to successfully reproducing many such stress-test crashes, CHESS has also found new concurrency errors in existing, unfailing stress tests.

3 The CHESS scheduler

This section describes the CHESS scheduler and explains how it obtains control over the scheduling of all concurrent activity in a program.

Execution of a concurrent program is highly nondeterministic. We distinguish between two classes of nondeterminism — input and interleaving nondeterminism. The former consists of values provided by the environment that can affect the program execution. For a user-mode program, this consists of return values of all system

calls and the state of memory when the program starts. Interleaving nondeterminism includes the interleaving of threads running concurrently on a multi-processor and the timing of events such as preemptions, asynchronous callbacks, and timers.

The primary goal of the CHESS scheduler is to *capture* all the nondeterminism during a program execution. This allows the scheduler to later reproduce a chosen concurrent execution by replaying these nondeterministic choices. Another equally important goal is to *expose* these nondeterministic choices to a search engine that can systematically enumerate possible executions of a concurrent program. As a concrete example, consider two threads that are blocked on a lock that is currently unavailable. On a subsequent release, the lock implementation is allowed to wake up any of the two blocked threads. Capturing this nondeterminism involves logging which thread was woken up by the system and ensuring that the same happens on a subsequent replay. Exposing this nondeterminism involves notifying the search engine about the two choices available, thereby allowing the engine to explore both possible futures.

Capturing and exposing *all* nondeterministic choices is a nontrivial task. Programs use a large number of concurrency primitives and system calls. The CHESS scheduler needs to understand the semantics of these functions accurately. Failing to capture some nondeterminism can impede CHESS' ability to replay an execution. Similarly, failing to expose nondeterminism can reduce the coverage achieved by CHESS and possibly reduce the number of bugs found. On the other hand, the scheduler has to be careful to not introduce new behaviors that are otherwise not possible in the program. Any such perturbation can result in false error reports that drastically reduce the usability of the tool. Also, the scheduler should not adversely slow the execution of the program. Any slowdown directly affects the number of interleavings CHESS can explore in a given amount of time. Finally, the techniques used to implement the scheduler should allow CHESS to be easily integrated with existing test frameworks. In particular, requiring the program to run under a modified kernel or a specialized virtual machine is not acceptable.

The rest of this section describes how CHESS addresses the aforementioned challenges. We first describe how CHESS handles input nondeterminism in the next subsection and focus on interleaving nondeterminism in following subsections.

3.1 Handling input nondeterminism

Handling input nondeterminism is necessary for deterministic replay. The standard technique for dealing with input nondeterminism involves log and replay [44, 13, 3, 23, 14, 27]. In CHESS, we decided not to implement a complete log and replay mechanism, and shifted the onus of generating deterministic inputs to the user. Most test frameworks already run in a controlled environment, which clean up the state of the memory, the disk, and the network between test runs. We considered the overhead of an elaborate log and replay mechanism unnecessary and expensive in this setting. By using these cleanup functions, CHESS ensures that every run of the program runs from the same initial state.

On the other hand, the scheduler logs and replays input values that are not easily controlled by the user, including functions that read the current time, query the process or thread ids, and generate random numbers. In addition, the scheduler logs and replays any error values returned from system calls. For instance, the scheduler logs the error value returned when a call to read a file fails. However, when the call succeeds CHESS does not log the contents of the file read. While such a log and replay mechanism is easier to implement, it cannot guarantee deterministic replay in all cases. Section 4.2 describes how the search recovers from any unhandled input nondeterminism at the cost of search coverage. Also, CHESS can be easily combined with tools that guarantee deterministic replay of inputs.

3.2 Choosing the right abstraction layer

A program typically uses concurrency primitives provided by different abstraction layers in the system. For Win32 programs, primitives such as locks and thread-pools are implemented in a user-mode library. Threading and blocking primitives are implemented in the Windows kernel. In addition, a program can use primitives, such as interlocked operations, that are directly provided by the hardware.

The scheduler is implemented by redirecting calls to concurrency primitives to alternate implementations provided in a "wrapper" library (Figure 1). These alternate implementations need to sufficiently understand the semantics of these primitives in order to capture and expose the nondeterminism in them. For complex primitives, an acceptable design choice is to include an implementation of these primitives as part of the program. For example, if a user-mode library implements locks, the scheduler can treat this library as part of the program under test. Now, the scheduler only needs to understand the simpler system primitives that the lock implementation uses. While this choice makes the scheduler implementation easier, it also prevents the scheduler from exposing all the nondeterminism in the lock implementation. For instance, the library could implement a particular queuing policy that determines the order in which threads acquire locks. Including this implementation as

part of the program prevents the scheduler from emulating other queuing policies that future versions of this library might implement. In the extreme, one can include the entire operating system with the program and implement the scheduler at the virtual machine monitor layer. While this choice might be acceptable for a replay tool like ReVirt [13], it is unacceptable for CHESS because such a scheduler cannot interleave the set of all enabled threads in the program. Rather, it can interleave only those threads that are simultaneously scheduled by the operating system scheduler.

In CHESS, we implement the scheduler for well-documented, standard APIs such as WIN32 and .NET's System.Threading. While these APIs are complex, the work involved in building the scheduler can be reused across many programs.

3.3 The *happens-before* graph

To simplify the scheduler implementation, we abstract the execution of the concurrent program using Lamport's *happens-before* graph [24]. Intuitively, this graph captures the relative execution order of the threads in a concurrent execution. Nodes in this graph represent instructions executed by threads in the system. Edges in the graph form a partial-order that determine the execution order of these instructions. Building the happens-before graph has two important benefits. First, it provides a common framework for reasoning about *all* the different synchronization primitives used by a program. Second, the happens-before graph *abstracts* the timing of instructions in the execution. Two executions that result in the same happens-before graph but otherwise execute at different speeds are behaviorally equivalent. In particular, CHESS can reproduce a particular execution by running the program again on the same inputs and enforcing the same happens-before graph as the original execution.

Each node in the happens-before graph is annotated with a triple—a task, a synchronization variable, and an operation. A task, in most cases, corresponds to the thread executing an instruction. But other schedulable entities, such as threadpool work items, asynchronous callbacks, and timer callbacks are also mapped as tasks. While such entities could possibly execute under the context of the same thread, the CHESS scheduler treats them as logically independent tasks. A synchronization variable represents a resource used by the tasks for synchronizing and communicating with each other, e.g., locks, semaphores, variables accessed using atomic operations, and queues. A fresh synchronization variable is created for a resource the first time it is accessed.

Each resource, depending on its type, allows a set of operations with potentially complex semantics. However, CHESS only needs to understand two bits of information, `isWrite` and `isRelease`, for each of these operations. The bit `isWrite` is used to encode the edges in the happens-before graph of the execution. Intuitively, this bit is true for those operations that change the state of the resource being accessed. If this bit is *true* for a node n, then two sets of happens-before edges are created: (1) edges to n from all preceding nodes labeled with the same synchronization variable as n, and (2) edges from n to all subsequent nodes labeled with the same synchronization variable as n.

The bit `isRelease` is *true* for those operations that unblock tasks waiting on the resource being accessed. The search module in CHESS needs to know the set of enabled tasks at any point in the execution. To maintain this information efficiently, CHESS not only maintains the set of enabled tasks but also the set of tasks waiting for each resource. Upon the execution of an operation whose `isRelease` bit is *true*, CHESS adds these waiting tasks to the set of enabled tasks.

For example, a `CriticalSection` resource provided by WIN32 allows three operations— `EnterCriticalSection`, `ReleaseCritical-Section`, and `TryEnterCriticalSection`. The `isWrite` bit is *true* only for the first two operations and for the last operation whenever it succeeds. The `isRelease` bit is *true* only for `ReleaseCriticalSection`.

Next, we describe how the CHESS scheduler captures the happens-before graph of a concurrent execution. We distinguish between two kinds of inter-thread communication that create edges in the graph. Section 3.4 describes how the scheduler captures communication through synchronization operations, while Section 3.5 describes how the scheduler addresses communication through shared memory.

3.4 Capturing the happens-before graph

During the program execution, CHESS redirects all calls to synchronization operations to a library of wrappers as shown in Figure 1. These wrappers capture sufficient semantics of the concurrency API to provide the abstraction described in Section 3.3 to CHESS. To create this abstraction, the wrappers must: (1) determine whether a task may be disabled by executing a potentially blocking API call, (2) label each call to the API with an appropriate triple of task, synchronization variable, and operation, and (3) inform the CHESS scheduler about the creation and termination of a task. Ideally, in order to minimize perturbation to the system, we would like to meet these three goals without reimplementing the API. To achieve these goals, CHESS maintains state variables for the currently-executing task, the mapping from resource handles to synchronization variables, the set of

enabled threads, and the set of threads waiting on each synchronization variable.

Achieving the first goal requires knowing which API functions can potentially block. We have observed that all such functions invariably have a non-blocking "try" version that indicates the non-availability of the resource using a return value. For example, if the critical section `cs` is not available, then `EnterCriticalSection(cs)` blocks whereas `TryEnterCriticalSection` returns *false*. The wrapper for a blocking function calls the "try" version first, and if this call fails (as indicated by the appropriate return value), then it adds the currently-executing task to the set of tasks waiting on this resource and removes it from the set of enabled tasks. Later, when a release operation is performed on this resource, each task in the set of waiting tasks is moved to the set of enabled tasks.

For the second goal, creating the task and synchronization variable for a particular API call is easily done using the state maintained by CHESS. The first element of the triple is simply the currently-executing task. The second element is obtained by looking up the address of the resource being accessed in the map from resource handles to synchronization variables. Setting the `isWrite` and `isRelease` bits in the third element of the triple requires understanding the semantics of the API call. Note that this understanding does not have to be precise; when in doubt it is always correct, at the cost of efficiency, to set either of those bits to *true*. Conservatively setting the `isWrite` bit to *true* only adds extra edges in the happens-before graph of the execution, adversely affecting the state-caching optimization described later (Section 4.4.2). Conservatively setting the `isRelease` bit to true might unnecessarily move all tasks waiting on a resource to the set of enabled tasks. Later, when the CHESS scheduler attempts to schedule these tasks, they will be moved back into the set of waiting tasks, thus creating some wasteful work. This robustness in the design of our scheduler is really important in practice because the exact behavior of an API function is often unclear from its English documentation. Also, this design allows us to refine the wrappers gradually as our understanding of a particular API improves.

For the third goal, it is important to identify the API functions that can create fresh tasks. The most common mechanism for the creation of a new task are functions such as `CreateThread` and `QueueUserWorkItem`, each of which takes a closure as input. While the former creates a new thread to execute the closure, the latter queues the closure to a threadpool which might potentially multiplex many closures onto the same thread. The CHESS wrappers for both of them are identical. The wrapper first informs CHESS that a new task is being created, creates another closure wrapping the input closure, and then calls the real API function on the new closure. The new closure simply brackets the old closure with calls to the CHESS scheduler indicating the beginning and the end of the task.

A task may be created when a timer is created using `CreateTimerQueueTimer`. 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 each expiration of the 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 is 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 for 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.

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 redirect all calls to the synchronization library by overwriting the import address table of the program under test. In addition, we use binary instrumentation to insert a call to the wrapper before instructions that use hardware synchronization mechanisms, such as interlocked operations. For .NET programs, we used an extended CLR profiler [11] that replaces calls to API functions with calls to the wrappers at JIT time. Finally, for the Singularity API, we use a static IL rewriter [1] to make the modifications. Table 1 shows the complexity and the

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.

amount of effort required to write the wrappers.

3.5 Capturing data-races by single-threaded execution

Most concurrent programs communicate through shared memory and it is important to capture this communication in the happens-before graph of an execution. If the program is data-race free, then any two conflicting accesses to a shared memory location are ordered by synchronization operations. In this case, the happens-before graph of synchronization operations captured by the wrappers, as described in Section 3.4, is sufficient to order all accesses to shared memory.

Unfortunately, our experience suggests that most concurrent programs contain data-races, many of which are intentional [46]. In this case, the happens-before graph should include additional edges that determine the order of racy accesses. Neglecting these edges may result in inability to replay a given execution. One possible solution is to use a dynamic data-race detection tool [46, 9] that captures the outcome of each data-race at runtime. The main disadvantage of this approach is the performance overhead—current data-race detection slow the execution of the program by an order of magnitude. Therefore, we considered this solution too expensive.

Instead, the CHESS scheduler controls the outcome of data-races indirectly by *enforcing* single-threaded execution [13, 39, 25]. By enabling only one thread at a time, CHESS ensures that two threads cannot concurrently access memory locations. Hence, all data-races occur in the order in which CHESS schedules the threads. While this solves the replay problem, there are two potential downsides of this approach. First, depending on the parallelism in the program, running one thread at a time can slow down the execution of the program. However, this lost performance can be recovered by running multiple CHESS instances in parallel, each exploring a different part of the interleaving state-space. We intend to explore this promising idea in the future. Second, CHESS may not be able to explore both of the possible outcomes of a data-race. This loss of coverage can either result in missed bugs or impact the ability of CHESS to reproduce a Heisenbug that occurs in the wild. We address this problem to an extent, by running the data-race detector on the first few runs of CHESS and then instrumenting the binary with calls to CHESS at the set of data-races found in these runs. This trick has helped us to successfully reproduce all Heisenbugs reported to us (§5), our most important criterian for the success of CHESS.

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 search operation

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.

During replay, 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 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 a yielding operation 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 [43, 31] 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 [13, 23, 27, 4], 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 starvation-free schedules

Many concurrent programming primitives implicitly assume that the underlying OS schedulers are *strongly fair* [2], that is, no thread is starved forever. 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 until they receive a signal from another thread. An unfair scheduler is not required to eventually schedule the signaling 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. 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 because CHESS relies on the termination of the 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, since any schedule that unrolls a spin-loop an arbitrary but finite number of times is 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 [34], 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 [34].

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 essential 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 [32], 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 [32] for a more formal argument.) On the other hand, we expect that most concurrency bugs happen [28] 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 two 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 [5] (see also [41]) — 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 [43, 31] is their ability to cache visited program states. This allows them to avoid 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 [31, 45]. 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 along each of 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 [16] but combines well with preemption bounding [33].

4.5 Monitoring executions

The main goal of CHESS is to systematically drive a concurrent program through possible thread interleavings. Monitoring an interleaving for possible errors, either safety violations or performance problems, is a orthogonal but important problem. Since CHESS executes the program being tested, it can easily catch standard assertion failures such as null dereferences, segmentation faults, and crashes due to memory corruption. The user can also attach other monitors such as memory-leak or

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

use-after-free detectors [36].

In addition to assertion failures, CHESS also checks each interleaving for deadlocks and livelocks. The wrappers maintain the set of enabled tasks (Section 3.4), allowing CHESS to report a deadlock whenever this set becomes empty. Detecting liveness violations is significantly more difficult and fundamentally requires the fair scheduling capability of CHESS (Section 4.3). Any liveness property is reducible to the problem of fair-termination [42], which is to check whether a program terminates under all fair interleavings. Then, 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 interleaving 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.

We have also implemented monitors for detecting data-races and for detecting whether an execution could result in behavior that is not sequentially consistent [6]. These monitors require trapping all memory accesses and consequently impose a significant runtime overhead. Therefore, we keep these monitors off by default but allow the user to turn them on as needed.

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.

PLINQ [12] 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 [8], which is part of Microsoft Robotics Studio Runtime. Dryad is a distributed execution engine for coarse-grained data-parallel applications [21]. Finally, Singularity [19] is a research operating system.

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 `sleep` and other yielding 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.

Finally, we used CHESS to systematically test the *entire* boot and shutdown sequence of the Singularity operating system [19]. The 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 imme-

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.

diately after the boot without forking the login process. These changes involved ~300 lines in four files.

5.3 Validating CHESS against stress-testing

Many believe 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 [28] 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 nondeterministic, 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. CHESS was able to reproduce the failure, within minutes in some cases, with less than ten threads and two preemptions. Similarly, CHESS was able to reproduce two failures that the tester had previ-

```
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 2: A race condition exposed when LiteEvents are used with multiple waiters.

ously (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

In this section, we describe two bugs that CHESS was able to find.

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 2 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 [34]: every

```
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 3: Violation of the good Samaritan property. Under certain cases, this loop results in the thread spinning idly till time-slice expiration.

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 3 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.

6 Related work

The systematic exploration of the behaviors of executable concurrent programs is not a new idea 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 [7] proposed repeatable deterministic testing by running a program with an input and explicit thread schedule. The idea of systematic generation of thread schedules came later under the rubric of reachability testing [20]. Recent work in this area includes RichTest [26] which performs efficient search using on-the-fly partial-order reduction techniques, and ExitBlock [5] 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 [17], 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 [43], Bogor [38], CMC [31], and MaceMC [22], 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—preemption bounding, and fair scheduling—absent from the previous work.

ConTest [15] 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 [25, 39, 44, 13, 3, 23, 14]. 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., [40, 35, 46], 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.

7 Conclusions

In this paper, we presented CHESS, a systematic testing tool for finding and reproducing Heisenbugs in concurrent programs. The systematic testing capability

of CHESS is achieved by carefully exposing, control-
ling, and searching all interleaving nondeterminism in
a concurrent system. CHESS works for three different
platforms—Win32, .NET, and Singularity. It has been
integrated into the test frameworks of many code bases
inside Microsoft and is used by testers on a daily basis.
CHESS has helped us and many other users find and re-
produce numerous concurrency errors in large applica-
tions. Our experience with CHESS indicates that such
a tool can be extremely valuable for the development
and testing of concurrent programs. We believe that all
concurrency-enabling platforms should be designed to
enable the style of testing espoused by CHESS.

Acknowledgments

References

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

[2] APT, K. R., FRANCEZ, N., AND KATZ, S. Appraising fairness
in languages for distributed programming. In *POPL 87: Princi-
ples of Programming Languages* (1987), pp. 189–198.

[3] 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: Virtual Execution Environments* (2006),
ACM, pp. 154–163.

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

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

[6] BUCKHARDT, S., AND MUSUVATHI, M. Effective program veri-
fication for relaxed memory models. In *CAV 08: Computer-Aided
Verification* (2008), pp. 107–120.

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

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

[9] CHOI, J.-D., LEE, K., LOGINOV, A., O'CALLAHAN, R.,
SARKAR, V., AND SRIDHARAN, M. Efficient and precise
datarace detection for multithreaded object-oriented programs. In
PLDI 02: Programming Language Design and Implementation
(2002), pp. 258–269.

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

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

[12] DUFFY, J. A query language for data parallel programming: in-
vited talk. In *DAMP* (2007), p. 50.

[13] 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).

[14] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A.,
AND CHEN, P. M. Execution replay of multiprocessor virtual
machines. In *VEE 08: Virtual Execution Environments* (2008),
ACM, pp. 121–130.

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

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

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

[18] GRAY, J. Why do computers stop and what can be done about it?
In *Büroautomation* (1985), pp. 128–145.

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

[20] 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.

[21] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FET-
TERLY, D. Dryad: distributed data-parallel programs from se-
quential building blocks. In *Proceedings of the EuroSys Confer-
ence* (2007), pp. 59–72.

[22] 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.

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

[24] LAMPORT, L. Time, clocks, and the ordering of events in a dis-
tributed system. *Communications of the ACM 21*, 7 (1978), 558–
565.

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

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

[27] 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), pp. 257–270.

[28] 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 Pro-
gramming Languages and Operating Systems* (2008).

[29] .NET Framework 3.5—http://msdn.microsoft.com/en-
us/library/w0x726c2.aspx.

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

[31] 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.

[32] 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.

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

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

[35] 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.

[36] Rational Purify—http://www-01.ibm.com/software/awdtools/purify.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

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

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

[44] 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.

[45] 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.

[46] 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.