Preemption Sealing for Efficient Concurrency Testing

Thomas Ball[1], Sebastian Burckhardt[1], Katherine E. Coons[2]
Madanlal Musuvathi[1], and Shaz Qadeer[1]

[1] Microsoft Research
[2] University of Texas at Austin

Abstract. The choice of where a thread scheduling algorithm preempts one thread in order to execute another is essential to reveal concurrency errors such as atomicity violations, livelocks, and deadlocks. We present a scheduling strategy called *preemption sealing* that controls where and when a scheduler is *disabled* from preempting threads during program execution. We demonstrate that this strategy is effective in addressing two key problems in testing industrial-scale concurrent programs: (1) tolerating existing errors in order to find more errors, and (2) compositional testing of layered, concurrent systems. We evaluate the effectiveness of preemption sealing, implemented in the CHESS tool, for these two scenarios on newly released concurrency libraries for Microsoft's .NET framework.

1 Introduction

Concurrent programs are difficult to design, implement, test, and debug. Furthermore, analysis and testing tools for concurrent programs lag behind similar tools for sequential programs. As a result, many concurrency bugs remain hidden in programs until the software ships and runs in environments that differ from the test environment.

Systematic concurrency testing offers a promising solution to the problem of identifying and resolving concurrency bugs. We focus on systematic concurrency testing as implemented in CHESS [16], a tool being used to test concurrent programs at Microsoft. A CHESS user provides a collection of tests, each exploring a different concurrency scenario for a program. A concurrency scenario might range from a simple harness that calls into a concurrent data structure, to a web browser starting up and rendering a web page. Given such a scenario, CHESS repeatedly executes the program so that each run of the program explores a different thread schedule, using novel stateless exploration algorithms [14, 15].

Of course, selecting which thread schedules are most useful among the exponentially many possible schedules is a central problem for the effectiveness of a tool like CHESS. We faced the following two related problems when deploying CHESS at Microsoft, which helped motivate this work:

1. Users want the ability to find *multiple distinct bugs* so they can pipeline the testing process and not be blocked waiting for bug fixes.

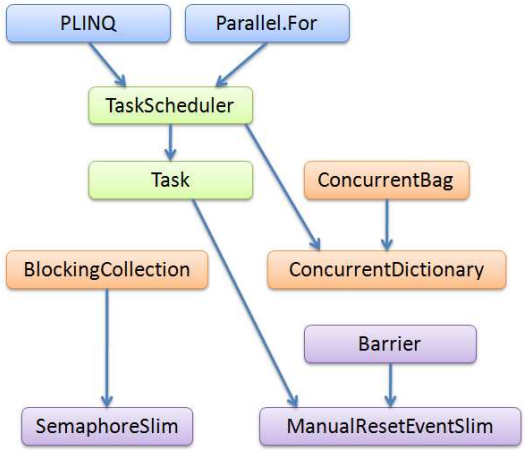

Fig. 1. Dependencies among .NET 4.0 concurrency classes. `SemaphoreSlim`, `Barrier`, and `ManualResetEventSlim` are synchronization primitives (SYN, purple). `BlockingCollection`, `ConcurrentDictionary`, and `ConcurrentBag` are concurrent data structures (CDS, orange). `Task` and `TaskScheduler` are part of a task parallel library (TPL, green). `PLINQ` and `Parallel.For` are parallel versions of LINQ and for-loops (blue).

2. Users want to perform *compositional testing* so they can focus the test on the components they are responsible for.

The first problem arises because many different thread schedules may manifest the same bug. Thus, even if the systematic search continues after finding a bug, that same bug may cause the system to crash repeatedly. This problem is important because large software systems often have a large number of bugs, some known and many unknown. Known bugs can be in various life stages: the tester/developer might be debugging, finding the root cause, designing a fix, or testing the fix. Depending on its severity, a bug may be fixed immediately or the fix may be deferred to a future release. As a result, it may be several weeks or even months before a bug is fixed. Thus, a tool such as CHESS will be most useful if it finds new bugs while avoiding schedules that trigger known bugs.

The second problem arises because a systematic search tests *all* possible schedules, even those that are irrelevant to the part of the system being tested. Well-engineered software consists of layered modules where upper layers depend on the services of lower layers, but not vice versa. Figure 1 shows an example of such a layered system from the .NET 4.0 libraries, which we will return to later in the paper. Usually, different teams are responsible for developing and testing different layers. A testing tool should allow users to "focus" the exploration on specific layers. If a particular layer, such as a low-level concurrency library, has been extensively tested or verified, then repeatedly testing its functionality when called from higher layers is a waste of valuable testing resources.

Preemption sealing is a simple but effective strategy to address these problems. A preemption is an unexpected interruption of a thread's execution caused, for example, by the thread's time slice expiring or a hardware interrupt occurring. A preemption-sealing scheduler disables preemptions in a particular scope of program execution, resorting to non-preemptive scheduling within that scope.

By resorting to non-preemptive scheduling, a preemption-sealing scheduler avoids exposing concurrency bugs that require at least one preemption within a given scope. To identify multiple errors, we seal preemptions in a scope related to the root cause of a bug. For example, if an error-inducing schedule contains a preemption in method m, we can instruct the scheduler to seal preemptions whenever control is within the scope of m in subsequent runs. To enable compositional testing, the user provides a set of methods or types that already have undergone thorough testing. By sealing preemptions in these scopes, the scheduler conserves valuable testing time.

Preemption sealing builds upon prior work on *preemption bounding* [14], a technique that first explores executions containing fewer preemptions. The hypothesis of preemption bounding is that most concurrency errors surface in executions that contain few preemptions. This hypothesis has been validated by various researchers [2, 14, 12]. Accordingly, a preemption-bounded scheduler explores executions with fewer preemptions first. Preemption bounding and preemption sealing are orthogonal scheduling strategies that combine naturally.

We implement preemption sealing in the CHESS concurrency testing tool and evaluate its effectiveness on a set of platform libraries for .NET that provide essential concurrency constructs to programmers. Testers for these libraries have been using CHESS over the past year to more thoroughly test these critical platform layers. We leverage 74 of their concurrency unit tests and use them to demonstrate preemption sealing's effectiveness in finding multiple errors and enabling compositional testing. Our experiments show that CHESS successfully finds multiple errors by sealing methods containing bug-inducing preemptions. Also, on average, compositional testing with preemption sealing cuts the number of executions explored during testing by more than half.

In the remainder of the paper, we formalize preemption-bounded scheduling (Section 2), define preemption sealing (Section 3), justify its use for finding multiple errors (Section 3.1) and compositional testing (Section 3.2), describe our implementation of preemption sealing and evaluate it on a set of .NET concurrency platform libraries (Section 4), discuss related work (Section 5), and conclude (Section 6).

2 Preemption-Bounded Scheduling

We model the execution of a concurrent program as a sequence of events, each corresponding to an operation performed by a thread. We represent an event with a five-tuple (tid, ctx, op, loc, blk), where tid is the thread id, ctx is the context of the thread including its program counter ($ctx.pc$) and its call stack ($ctx.stack$), op is the operation performed, loc is the (shared) memory location or object

on which the operation is performed, and *blk* is a boolean flag that indicates whether the thread is blocked while performing the operation or not. We use $|E|$ to denote the length of execution E and $E[i]$ to denote the event at position i in execution E. We access the components of an event e with '.' notation:

$$(e.tid, e.ctx, e.op, e.loc, e.blk)$$

An event e is blocking if $e.blk$ is true. A *completing event* for a blocking event e is the event $(e.tid, e.ctx, e.op, e.loc, false)$. A sequence is *well-formed* if for every blocking event e in an execution E, the next event performed by thread $e.tid$ in E, if any, is the completing event for e. We only consider executions that are well-formed. Also, we use _ to denote the *op* and *loc* components of events that do not access shared state.

A *context switch* in an execution E is identified by an index c such that $0 \leq c < |E| - 1$ and $E[c].tid \neq E[c+1].tid$. A context switch c is said to be *non-preemptive* if $E[c].blk$ is true or $E[c].op$ is the thread "exit" operation, signaling the end of the execution of thread $E[c].tid$. Otherwise the context switch is said to be *preemptive*. We call a preemptive context switch a *preemption*, for short.

The preemption bound of an execution E is the number of preemptions in E. Preemption-bounded scheduling ensures that each execution contains at most P preemptions, where P is a number chosen by the tester. Note that a preemption bound of zero simply means that the scheduler runs non-preemptively, executing the current thread until it blocks and then switching to a different (enabled) thread. If non-preemptive scheduling is unable make progress (because all threads are blocked), then the program contains a deadlock. Thus, when a preemption-bounded scheduler runs out of preemptions, it simply resorts to non-preemptive scheduling until the end of execution or a deadlock is encountered.

In addition to the choice of where to place preemptive context switches, the scheduler also has the choice of which enabled thread to execute after a context switch. This latter choice is typically constrained by a desire for fair scheduling, but fairness is beyond the scope of this paper (for more details about fair stateless model checking, see [15]). In this paper, we assume the scheduler is free to schedule any enabled thread after a context switch.

Figure 2(a) shows a buggy "bank account" class `Acct` and a test method `TestAcct` containing a test scenario. The test scenario creates three threads that test the class `Acct`. Thread `t1` withdraws from the bank account, thread `t2` reads the account balance, and thread `t3` deposits to the account.

Figure 2(b) shows an execution of this program that exposes an assertion failure. For brevity, we represent the context by the program label and use the string "`acc`" to refer to the single instance of the `Acct` class. For example, the operation at label `L2` is a lock operation on the object `acc`, while the operation at label `L4` is a read operation on the field `acc.bal`. In this execution, the transition from `(t1,L5,_,_,F)` to `(t3,L6,lock,acc,F)` represents a preemption. Thread `t1` is preempted at label `L5` of the `Read` method after reading the account balance, but before acquiring the lock on `acc` at label `L2` of the `Withdraw` method. Next, thread `t3` executes the entire `Deposit` method. Then, because thread `t3` has

(a)

```
  public class Acct {
    volatile int bal;

    public Acct(int n) {
      bal = n;
    }
    public void Withdraw(int n) {
L1:   int tmp = Read();
L2:   lock (this) {
        bal = tmp - n;
L3:   }
    }
    public int Read() {
L4:   return bal;
L5: }
    public void Deposit(int n) {
L6:   lock (this) {
        var tmp = bal;
        bal = 0;
L7:     bal = tmp + n;
LU:   }
L8: }
  }
```

```
 void TestAcct() {
   var acc = new Acct(10);

   var t1 = new Thread(o =>
     { (o as Acct).Withdraw(2);
L9:  });

   var t2 = new Thread(o =>
     { var b = (o as Acct).Read();
LA:    assert(b>=8);
LB:  });

   var t3 = new Thread(o =>
     { (o as Acct).Deposit(1);
LC:  });

   t1.Start(acc); t2.Start(acc);
   t3.Start(acc);
   t1.Join(); t2.Join(); t3.Join();

LD: assert(account.Read() == 9);
LE:
}
```

(b)

```
(t2,L4,read,acc.bal,F) (t2,L5,_,_,F) (t2,LA,_,_,F) (t2,LB,_,_,F)
(t1,L1,_,_,F) (t1,L4,read,acc.bal,F) (t1,L5,_,_,F) (t3,L6,lock,acc,F)
(t3,L7,write,acc.bal,F) (t3,LU,unlock,acc,F) (t3,L8,_,_,F) (t3,LC,_,_,F)
(t1,L2,lock,acc,F) (t1,L3,unlock,acc,F) (t1,LA,_,_,F) (t0,LD,_,_,F)
```

(c)

```
(t3,L6,lock,acc,F) (t3,L7,write,acc.bal,F) (t3,LU,unlock,acc,F)
(t2,L4,read,acc.bal,F) (t2,L5,_,_,F) (t2,LA,_,_,F)
```

Fig. 2. (a) Simple bank account example with two bugs and (b)-(c) two executions demonstrating the two bugs.

completed, a non-preemptive context switch returns control to thread `t1`, which acquires the lock at label `L2` and executes to completion. This execution violates the assertion at label `LD` because thread `t3`'s deposit is lost.

3 Preemption Sealing

Preemption sealing uses information associated with events to determine whether an event meets certain criteria, which we call a "scope". If an event is within scope, preemption sealing prevents the scheduler from performing a preemption prior to that event.

A *scope* is a function F that takes an event as input and returns true if that event is "in scope" and false otherwise. The function F may examine any data

associated with an event e, such as its thread id, $e.tid$, its operation, $e.op$, etc. In this paper, we assume a finite set of scopes, given by a finite set of functions. Thus, a scope F identifies a subsequence of an execution E containing those events $E[i]$ such that $F(E[i])$ is true. Operationally, for each event executed, we can apply the function F to determine if it is in the scope of F or outside it, though we use more efficient means in practice. Preemptions are disabled at events that are "in scope" and are enabled at events that are not in any scope.

By disabling preemptions in certain scopes, the scheduler effectively focuses its search on other parts of the search space. Disabling preemptions does not introduce new deadlocks. As noted in the previous section, when a scheduler has no preemptions to use, it simply resorts to non-preemptive scheduling. Thus, the only way the scheduler cannot make progress in the presence of preemption sealing is if the program deadlocks. Also, it is straightforward to see that disabling preemptions does not introduce additional behaviors in the program and thus does not introduce safety violations.

Preemption sealing can be seen as an extension of previous work that addresses the relationship between data races and the placement of preemptions [14]. In that work, Musuvathi and Qadeer partition the world of all objects into synchronization objects and data objects, as is typical when defining data races. They show that if a program is data-race free then it is possible to disable preemptions at operations on data objects without missing errors in the program.

Preemption sealing builds upon this work by disabling preemptions at operations on synchronization objects when those operations occur within a particular scope. We discuss circumstances under which preemption sealing can be done safely without missing errors. In the two scenarios we consider, finding multiple errors and compositional testing, we find that preemption sealing improves the efficiency and efficacy of systematic search by eliminating thread interleavings that fall within a well-defined scope.

3.1 Detecting Multiple Errors

Detecting multiple errors is a difficult problem because many different thread interleavings may expose the same bug. To alleviate this problem, preemption sealing capitalizes on the observation that during a preemption-bounded search, the preemptions involved in a failure-inducing schedule are good indicators of the *root cause* of the failure. This observation is a consequence of the following two reasons: (1) the scheduler always has a choice regarding whether or not to introduce a preemption prior to a given event and (2) the scheduler carefully exercises this choice to explore executions with fewer preemptions first. Thus, the preemptions in a failure-inducing schedule are crucial to expose the bug. Otherwise, the scheduler would have found the same bug with fewer preemptions.

We return to the bank account example in Figure 2(a) to illustrate the problem of finding multiple errors. Figure 2(b) shows an execution that ends in an assertion failure at label `LD` because the bank account balance is incorrect. This failure occurs because the `Withdraw` method does not contain proper synchronization, which makes its effect appear non-atomic. A preemption at label `L2` in

the `Withdraw` method, followed by complete execution of the `Deposit` method, will cause the assertion failure.

Figure 2(c) shows an execution that fails due to another defect in the class `Acct`. Because the `Read` method does not use synchronization, it may observe an intermediate value of the account balance (after it has been set to zero by the `Deposit` method). This execution leads to an assertion failure at label `LA`.

We wish to find both errors rather than first finding one, asking the programmer to fix it, waiting for the fix, and then running again to find the second error. We would like the search to avoid known errors once they have been identified by "tolerating" the error in a temporary way.

Our idea is inspired by the observation that programmers intend many, if not most, methods to appear atomic in their effect when executed concurrently [6]. Thus, once we find an error that requires a preemption in method m to surface, we wish to seal method m from being preempted in the rest of the search. Effectively, this means that once the scheduler starts executing method m, it executes it to completion (modulo the case where m blocks). Note that we could seal just at the specific program counter where the preemption took place, but there are likely many other preemption points in the same method that will expose the same error. The above observation implies that methods are a natural scope in which to seal preemptions.

We generalize this idea to multiple preemptions. Assume a preemption bounded search that explores all executions with P preemptions before exploring any executions with $P+1$ preemptions. Thus, if no errors were found with P preemptions, then an error found with $P+1$ preemptions could not be found with P or fewer preemptions. If an error surfaces in execution E with preemption set S of size $|S|$, then at most $|S|$ methods must be sealed. The *preemption methods* are the active methods (methods on top of the call stack) in which the preemptions occur: $\{m \mid s \in S, E[s].ctx.stack.top = m\}$. If two different tests fail with the same set of preemption methods, the failures are likely due to the same error.

Note that preemption sealing at the method level may not eliminate the failure. For example, suppose method m calls method n and a preemption in either method leads to the same failure. If the preemption in method n occurs first, then sealing only method n will not prevent the failure. If the preemption in method m occurs first, however, and we use dynamic scope when sealing the preemption in method m, then we will ensure that method n will not be preempted when called from m. Thus, we use dynamic scoping when sealing preemption methods.

3.2 Compositional Testing

Strict layering of software systems is a basic software engineering practice. Upper layers depend on the services of lower layers, but not vice versa. Different teams may develop and test the different layers. The efficiency of testing the entire system depends greatly on eliminating redundant tests. This observation implies that in a layered system, tests for the upper layers need not (indeed, *should not*) perform redundant tests on the functionality of the lower layers.

Complicating matters, each layer of a system may be "thread-aware", protecting its data from concurrent accesses by an upper layer's threads, while explicitly creating threads itself to perform its tasks more efficiently.

However, although one may imagine and craft arbitrarily complicated interactions between layers, in practice, function calls into lower layers are often meant to appear atomic to the upper layers. In fact, several dynamic analysis tools (such as SideTrack [18], Atomizer [6], and Velodrome [8]) rely on this programming practice, as they are designed to check the atomicity of such function calls. What this means for preemption sealing is that

> *if we can establish or trust the lower-level functions to be atomic, it is safe to disable preemptions in the lower layer while testing the upper layer.*

Although this claim may be simple to understand intuitively, it should be understood in the context of prior work on atomicity [6]. This work derives the definition of atomicity from the classic definition of conflict-serializability and treats all function calls into the lower layer as transactions.

The concept of layering means that we partition the code into an upper layer A and a lower layer B such that A calls into B, B never calls into A, and execution starts and ends in A. For an execution E, defined earlier as a sequence of events, we label all events as A-events or B-events. For simplicity, we assume that each thread executes at least one A-event or B-event in between any pair of calls/returns that transition between layers.

For a fixed execution E we define transactions as follows. Let E_t be the sequence of events by thread t. More formally, E_t is the maximal subsequence of E consisting of events by only t. We then define a *transaction* of thread t to be a maximal contiguous subsequence of E_t consisting of only B-events. Atomicity is now characterized as follows, in reverse order of logical dependency:

- The layer B is *atomic* if all executions E are serializable.
- An execution E is *serializable* if it is equivalent to a serial execution.
- Two executions are *equivalent* if one can be obtained from the other by repeatedly swapping adjacent independent events.
- Two events are *dependent* if either (1) they are executed by the same thread, (2) they are memory accesses that target the same location and at least one writes to the location, (3) they are operations on the same synchronization object, and are not both side-effect-free.[3]
- An execution E is called *serial* if there are no context switches within transactions. For any context switch at position c, the event $E[c]$ is either not part of any transaction, or is the last event of a transaction.

Thus, if B is atomic, then for any execution that reveals a bug, there exists an equivalent serial execution that also reveals the bug. Such a serial execution does not contain any preemptions inside B, so the search will still cover this serial execution even when sealing preemptions in B.

[3] An example of a side-effect-free operation is a failed (blocking) lock acquire operation.

4 Implementation and Evaluation

We implemented preemption sealing in CHESS, a tool for concurrency testing [14]. CHESS repeatedly executes a concurrency unit test and guarantees that each execution takes a different thread schedule. CHESS records the current thread schedule so that when it finds an error, it can reproduce the schedule that led to the error. CHESS detects errors such as assertion failures, deadlocks, and livelocks, as well as data races, which are often the cause of other failures. CHESS contains various search strategies, one of which is preemption bounding.

After finding an error, CHESS runs in "repro" mode to reproduce the error by replaying the last stored schedule. During this repro execution CHESS collects extensive context information, such as the current call stack, to produce an attributed execution trace for source-level browsing. During this execution, CHESS also outputs *preemption methods* from the stored schedule. The preemption methods consist of methods in which CHESS placed a preemption.

To implement preemption sealing, we extended CHESS's API with methods to enable and disable preemptions. We implemented a preemption sealing strategy via a CHESS monitor that tracks context information, such as which method is currently on the top of the call stack, and makes calls to the new API to enable/disable preemptions. Command-line parameters to CHESS enable preemption sealing based on assembly name, namespace, class name, or method name. For the purposes of this paper, we use two options: `/dpm:M` for "disable/seal preemptions in method M"; `/dpt:T` for "disable/seal preemptions in all methods in type T"[4]. As currently implemented, we disable preemptions in the dynamic scope of a method, which suits our two applications (as discussed previously). Other scoping strategies are possible within the framework we implemented.

We evaluated preemption sealing's ability to find multiple errors and enable compositional testing on new parallel framework libraries available for .NET. These libraries include:

- *Concurrency and Coordination Runtime* (CCR) provides a highly concurrent programming model based on message-passing with powerful orchestration primitives enabling coordination of data and work without the use of manual threading, locks, semaphores, etc. (http://www.microsoft.com/ccrdss/)
- *New synchronization primitives* (SYN), such as `Barrier`, `CountdownEvent`, `ManuelResetEventSlim`, `SemaphoreSlim`, `SpinLock`, and `SpinWait`;
- *Concurrent data structures* (CDS), such as `BlockingCollection`, `ConcurrentBag`, `ConcurrentDictionary`, etc.
- *Task Parallel Library* (TPL) supports imperative task parallelism.
- *Parallel LINQ* (PLINQ) supports declarative data parallelism.

In all of the experimental results below, we ran CHESS with its default settings: preemptions are possible at all synchronization operations, interlocked operations, and volatile memory accesses; the scheduler can use at most two preemptions per test execution.

[4] The sense for these switches could trivially be switched so that the user could disable preemptions everywhere *except* the specified scope.

Sealed methods/types	Asserts	Timeouts	Livelocks	Deadlocks	Leaks	OK
∅	5	3	40	0	0	5
+ `DQueue.TryDequeue`	6	5	0	1	1	40
+ `TEW.WaitForTask`	5	5	0	2	1	40
+ `Port.RegisterReceiver` + `Port.PostInternal`	5	5	0	0	0	43
`DQueue`	5	5	0	2	0	41

Table 1. Evaluation of preemption sealing for detecting multiple errors (Rows 1-4), and for compositional testing (Row 5).

4.1 Discovering Multiple Unique Errors

We first evaluate preemption sealing's ability to discover multiple unique errors on the CCR code base, which has an accompanying set of concurrency unit tests. Most of these tests ran without modification under CHESS. The only modification we made was to decrease the iteration count for certain loops. Some tests contained high-iteration count loops to increase the likelihood of new thread interleavings. Because CHESS systematically searches the space of possible thread interleavings, this repetition is unnecessary within a single test. We took all of the CCR unit tests from its *CoreSuite, CausalitySuite, SimpleExamples*, and *TaskTest* suites, which resulted in 53 independent concurrency unit tests.

Table 1 shows the results of running CHESS on each of the 53 tests. The first column shows the set of preemption-sealed methods/types (initially empty). The next five columns show the number of tests that failed: **Asserts** occur when a test assertion fails; **Timeouts** occur when a test execution takes longer than ten seconds (CHESS default); **Livelocks** occur when a test executes over 20,000 synchronization operations (CHESS default, most concurrency unit tests, including those in CCR, execute hundreds of synchronization operations); **Deadlocks** are self explanatory; **Leaks** means that the test terminates with child threads alive - CHESS requires that all child threads complete before the test terminates. The final column (**OK**) contains the number of tests for which CHESS successfully explored all schedules within the default preemption bound (of two) without finding an error.

During the first CHESS run (Row 1) we see five assertion failures. All of these failures occurred on the first test execution, which never contains a preemption. These five failures represent errors in the test harness code. The three timeouts also occur on the first execution. These timeouts have a single root cause, which is a loop in the CCR scheduler that contains no synchronization operations, and that does not yield the processor (a violation of the "Good Samaritan" principle [15]). Because these assertion failures and timeouts occurred on the initial execution, which contains no preemptions, they were not candidates for preemption sealing.

The 40 tests that failed with a livelock all failed well into CHESS testing. Each failure was found in a schedule containing a single preemption in the method `DQueue.TryDequeue`, as output by CHESS during the repro phase. To evaluate

preemption sealing, we ran CHESS on the 53 tests again, sealing only the method `DQueue.TryDequeue` (Row 2). The effect of sealing is stark: all 40 of the tests that previously livelocked were able to avoid the livelock.[5] While sealing only one method, CHESS was able to avoid a livelock in 40 tests, verify 35 of those tests correct within the default preemption bound, and detect five new failures: one assertion failure, one deadlock, one thread leak, and two timeouts. The five new failures all have associated preemption methods, output by CHESS (`TEW` = `TaskExecutionWorker`):

- *Assertion failure*: `TEW.WaitForTask`, `TEW.Signal`;
- *Timeouts*: `TEW.WaitForTask`;
- *Deadlock*: `Port.RegisterReceiver`, `Port.PostInternal`;
- *Thread Leak*: `TEW.WaitForTask`, `Port.PostInternal`;

Based on these results, we performed two more runs of CHESS (Rows 3 and 4 of Table 1). In the third run, we sealed the additional method `TEW.WaitForTask`. This converted one test from an assertion failure into a deadlock. In the fourth run, we additionally sealed the methods that contained preemptions leading to the first deadlock: `Port.RegisterReceiver` and `Port.PostInternal`. As seen in Row 4, sealing these methods eliminated both deadlocks and the thread leak, converting both into passing tests.

The results of this experiment show the efficacy of preemption sealing at the method level for the CCR code base. Without any code modification, sealing the method that led to 40 livelocking tests resulted in five new bugs and 35 passing tests. Further sealing exposed an additional deadlock, and enabled more tests to run to completion.

4.2 Compositional Testing

When evaluating preemption sealing for compositional testing, we consider two metrics: (1) what is the bug yield relative to testing without preemption sealing?; (2) for tests that produce the same results with and without sealing, what is the run-time benefit of preemption sealing?

We take another look at CCR before moving to the other .NET libraries. CCR uses a queue (implemented by `DQueue`) containing tasks for the CCR scheduler to run. The scheduler removes tasks from this queue, while other CCR primitives create new tasks that are placed in the queue. Using the terminology from Section 3.2, the class `DQueue` is layer B, and the other components (the scheduler and the CCR primitives) are layer A, which make use of the services of B.

The last row in Table 1 shows the results of running CHESS with preemption sealing on all methods in the class `DQueue`. As expected, preemption sealing at this level will not find the livelock because the method `DQueue.TryDequeue`

[5] An interesting twist to the livelock bug is that while the developer agreed that there was a potential performance problem, he thought it would not occur very often and decided not to address the issue. In this case, the ability to avoid the livelock without requiring a change to the code was crucial to make progress finding more bugs.

Test	Sealed scope	Result N	Result S	Executions N	Executions S	Seconds N	Seconds S	Execs/sec N	Execs/sec S	Speed-up
BlkCol1	SemSlim	D	D	59167	18733	527.0	206.0	112.3	91.0	2.6
BlkCol2	”	P	P	258447	106608	2128.0	1181.0	121.4	90.3	1.8
BlkCol3	”	D	D	265	265	1.8	2.2	147.2	120.5	0.8
BlkColRC1	”	P	P	1114	364	8.8	4.5	126.6	80.8	2.0
BlkColRC2	”	P	P	2406	510	22.8	7.2	105.5	70.8	3.2
BlkColRC3	”	P	P	6084	2391	49.0	33.9	124.1	70.5	1.4
BlkColRC4	”	P	P	5003	1012	36.9	13.7	135.6	73.9	2.7
BlkColRC5	”	D	D	1	1	0.2	0.3	5.0	3.3	0.7
BarrierRC1	MRES	P	P	776	109	5.5	1.5	141.1	72.7	3.7
BarrierRC2	”	P	P	166	92	1.7	1.0	97.6	92.0	1.7
BarrierRaw	”	A	A	96	33	0.7	0.7	137.1	47.1	1.0
CBagRC1	CDict	P	P	559	375	3.7	3.7	151.1	101.4	1.0
CBagRC2	”	P	P	559	375	3.8	3.9	147.1	96.2	1.0
CBagAPC	”	P	P	6639	4168	46.0	39.0	144.3	106.9	1.2
CBagACTA	”	P	P	8230	5727	58.0	57.0	142.9	100.5	1.0
CBagTTE	”	P	P	1212	793	7.6	7.5	159.5	105.7	1.0
CBagTTP	”	P	P	1529	775	12.9	5.6	118.5	138.4	2.3
NQueens1	TSchd	L	L	1145	1318	19.4	61.8	59.0	21.3	0.3
NQueens2	”	T	T	10146	1027	182.1	55.8	55.7	18.4	3.2
NQueens3	”	T	T	9887	1027	181.8	56.4	54.4	18.2	3.2
PLINQ	”	P	P	3668	1031	33.1	12.9	110.8	79.9	2.6

Table 2. Evaluation of preemption sealing for compositional testing. Columns labeled 'S' use preemption sealing and columns labeled 'N' do not. **Abbreviations**: BlkCol (BlockingCollection), CBag (ConcurrentBag), SemSlim (SemaphoreSlim), MRES (ManualResetEventSlim), CDict (ConcurrentDictionary), TSchd (TaskScheduler), P (Pass), D (Deadlock), A (Assert), L (Livelock), T (Thread leak).

is sealed. However, CHESS discovers both deadlocks, which indicates that these deadlocks are due to defects in layer A. The analysis in the previous section confirms this result. For the two deadlocks, CHESS with the `DQueue` class sealed found them in 4,662 schedules (59 seconds) and 142 scheules (2 seconds), respectively. The runs that found the deadlocks without sealing `DQueue` took 9,774 schedules (126 seconds) and 10,525 schedules (330 seconds), respectively.

The other concurrency libraries that we consider include the layers illustrated in Figure 1. This figure shows dependencies among a subset of the classes in these libraries. At the lowest level are the new synchronization primitives (SYN) and the concurrent data structures (CDS, mostly lock-free). On top of these two libraries sits a new task scheduler (TPL), with a set of primitives for task parallelism. Finally, on top of TPL sits the implementation of parallel LINQ (PLINQ) for querying LINQ data providers, and parallel for loops for data parallelism. The test team for these libraries explicitly developed CHESS tests for most of these classes. We used their tests, unmodified, for our experiments.

Table 2 shows the results of these experiments. The first column is the test name, which indicates the class being tested. **Sealed scope** lists the class that we told CHESS to seal based on the dependencies shown in Figure 1 (see caption for abbreviations). The next three columns, **Result**, **Executions**, and **Seconds**, present results for two CHESS runs, one without sealing (columns labeled 'N') and one with sealing (columns labeled 'S'). The column **Execs/sec** shows the executions per second for both runs. Finally, the last column is the speedup in total execution time attained via preemption sealing.

For example, the first row shows that CHESS found a deadlock in the test `BlockCol1` both with and without preemption sealing. With class `SemaphoreSlim` sealed, however, CHESS found the deadlock after exploring one-third as many test executions, and 2.6 times faster.

The **Result** columns validate that preemption sealing at lower layers did not mask errors in higher layers. CHESS reported the same result for all tests both with and without preemption sealing. On average, preemption sealing reduced the number of executions explored by more than half. In all but three tests, preemption sealing reduced the time taken for CHESS to finish or left it the same, resulting in an average speedup of 1.83. We expect these numbers to improve if we optimize the instrumentation required to implement preemption sealing. In particular, our instrumentation results in a prohibitive overhead in the TPL tests, probably due to frequent calls to small methods.

5 Related Work

The main contribution of this paper is the concept of preemption sealing as a solution to two important problems in concurrency testing—finding multiple distinct bugs in a single test run, and compositional testing.

The idea of using preemption sealing to discover multiple distinct errors in concurrent programs can be viewed as a root cause analysis for concurrency errors. For sequential programs, using executions that pass to help localize the cause of failures has been popular [1, 9]. For example, the SLAM software model checker [1] determines which parts of an error trace are unique from passing traces and places **halt** statements at these locations to guide the model checker away from the error trace and towards other errors. This idea is analagous to preemption sealing, but for the sequential rather than the concurrent case.

The idea of using preemption sealing for compositional testing is most closely related to the use of atomicity for simplifying correctness proofs of multithreaded programs (e.g., [7, 4]). However, that work used atomicity only for the purpose of static verification; to the best of our knowledge, ours is the first effort to use this idea in the context of runtime verification. Our use of atomicity for compositional testing is orthogonal to the large body of work on runtime verification techniques for detecting atomicity violations (e.g., [13, 8, 5]). It is also worth noting that while most work on static compositional verification of concurrent programs requires manual specifications, our approach is fully automatic; we use the preemption-sealed version of a component as its specification.

Delta-debugging can be used to identify, from a failing execution, the context switch points that cause a multithreaded program to fail [3]. Our work exploits preemption bounding to make this problem simpler. Since preemptions are the likely causes of bugs and the erroneous execution discovered by CHESS has few preemptions, the problem of discovering the root cause is greatly simplified. Finally, our goal goes beyond root-cause analysis to find multiple qualitatively different bugs.

Apart from improving concurrency testing, preemption sealing can be used to make programs more resilient to concurrency errors in a spirit similar to recent work on tolerating locking-discipline violations [17] and deadlocks [20, 11].

Recent work has investigated techniques for creating real data-races [19] and deadlocks [10] by using feedback from other conservative static or runtime analysis techniques. Our work is orthogonal and complementary to this work; while they focus on where to place preemptions we focus on where not to place preemptions, via preemption sealing.

6 Conclusions

Preemption sealing is a scheduling strategy that increases the efficiency and efficacy of run-time tools for detecting concurrency errors. Preemption sealing has many potential applications and we considered two of them in depth here: tolerating existing errors in order to find more errors; and compositional testing of layered systems. The power of preemption sealing is that it does not require code modifications to the program under test and can be easily implemented in existing schedulers, whether part of model checking, testing, or verification tools. Our evaluation shows that preemption sealing is effective at finding multiple bugs and testing layered concurrent systems more efficiently.

References

1. Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, pages 97–105, 2003.
2. Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Producing scheduling that causes concurrent programs to fail. In *PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 37–40, New York, NY, USA, 2006. ACM.
3. Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *ISSTA 02: International Symposium on Software Testing and Analysis*, pages 210–220, 2002.
4. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL 09: Principles of Programming Languages*, pages 2–15, 2009.
5. Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *CAV 08: Computer Aided Verification*, pages 52–65, 2008.
6. C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL 04: Principles of Programming Languages*, pages 256–267. ACM Press, 2004.

7. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI 03: Programming Language Design and Implementation*, pages 338–349. ACM, 2003.
8. Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI '08*, pages 293–303. ACM, 2008.
9. Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.
10. Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09*, pages 110–120. ACM, 2009.
11. Horatiu Jula, Daniel M. Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI 08: Operating System Design and Implementation*, pages 295–308, 2008.
12. Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS 08: Architectural Support for Programming Languages and Operating Systems*, 2008.
13. Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS 06: Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.
14. Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 07: Programming Language Design and Implementation*, pages 446–455, 2007.
15. Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *PLDI 08: Programming Language Design and Implementation*, 2008.
16. Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.
17. Sriram K. Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS 09: Architectural Support for Programming Languages and Operating Systems*, pages 181–192, 2009.
18. Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP 2009*, LNCS 5502, pages 394–409. Springer-Verlag, 2009.
19. Koushik Sen. Race directed random testing of concurrent programs. In *PLDI '08*, pages 11–21. ACM, 2008.
20. Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott A. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL 09: Principles of Programming Languages*, pages 252–263, 2009.