First-class Concurrency Testing and Debugging

Thomas Ball, Sebastian Burckhardt, Madan Musuvathi, Shaz Qadeer
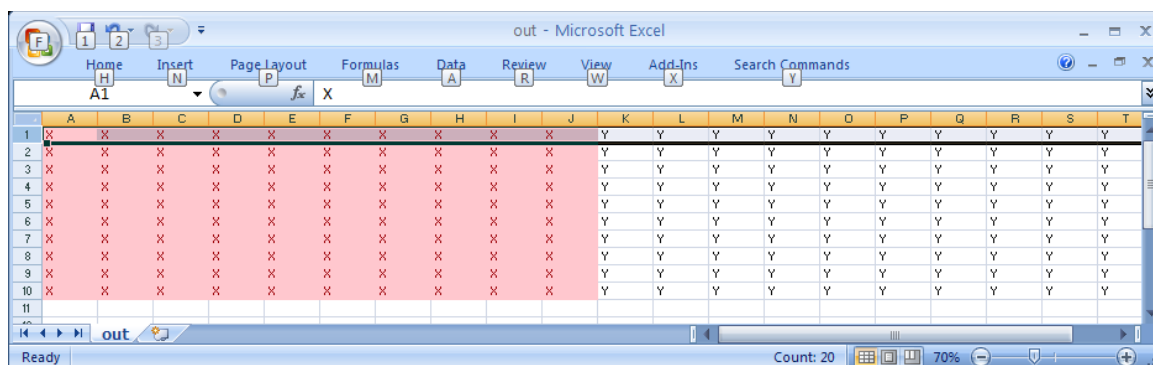Microsoft Research

Introduction

Concurrency is a fundamental attribute of systems software. Asynchronous computation is the norm in important software components such as operating systems, databases, and web servers. As multi-core architectures find their way into desktop computers, we will see an increasing use of multithreading in application software as well.

Unfortunately, the design of concurrent programs is a very challenging task. The main intellectual difficulty of concurrent programming lies in reasoning about the interaction between concurrently executing threads. Nondeterministic thread scheduling makes it extremely difficult to reproduce behavior from one run of a program to another. As a result, the process of testing and debugging concurrent software becomes tedious resulting in a drastic decrease in the productivity of programmers and testers. For example, today testers attempt to induce bad schedules by creating thousands of threads, running tests millions of times, and forcing context switches at special program locations. Such approaches are not systematic or predictable, as there is no guarantee that one execution will be different from another. And, if a bug is found, it may be difficult to reproduce the thread schedule that led to the bug.

Since concurrency is both important and difficult to get right, it is imperative that we develop and invest in techniques and tools to aid in the process of developing, testing and debugging of concurrent programs. We believe that there a few simple yet fundamental concepts that can make testing and debugging of concurrent programs a *predictable* and *systematic* process. The view of a concurrent execution as a *total order* of interleaved thread actions (also called a *thread schedule*) is an essential simplifying abstraction. If we follow the implications of this one idea, both testing and debugging become much more predictable. We provide a glimpse into a future where testing and debugging of concurrent programs is an activity with first-class support from all levels of the software platform. We are exploring these ideas using the CHESS concurrency analysis platform to provide a radically new and improved testing/debugging experience for concurrent programs. CHESS provides systematic, repeatable, and efficient enumeration of thread schedules, in a style of program verification commonly known as model checking.

Test Tools

Sequential programs have many useful testing tools and test methods. For concurrent programs, there are few testing tools and methods to point to. Furthermore, because testers don't have control over the thread schedule, they must resort to hacks like inserting sleep statements, random waits, or loading the system with many hundreds of threads in order to force some schedule variation. To emphasize the point, consider a simple C# program with two threads: the main thread forks off a child thread and then proceeds to write out ten instances of the string "X," with separate calls to Console.Write. Then the main thread waits for the child thread, which is concurrently writing out ten instances of the string "Y,". After the child thread finishes, the main thread writes a newline. This simple program can (potentially) print out $\frac{20!}{10!.10!}$ different strings, depending on the thread schedule. Let's look at the result of executing the program ten times in a row. The result is ten lines of comma-separated values. We import this into Excel, and color each "X" red, resulting in:
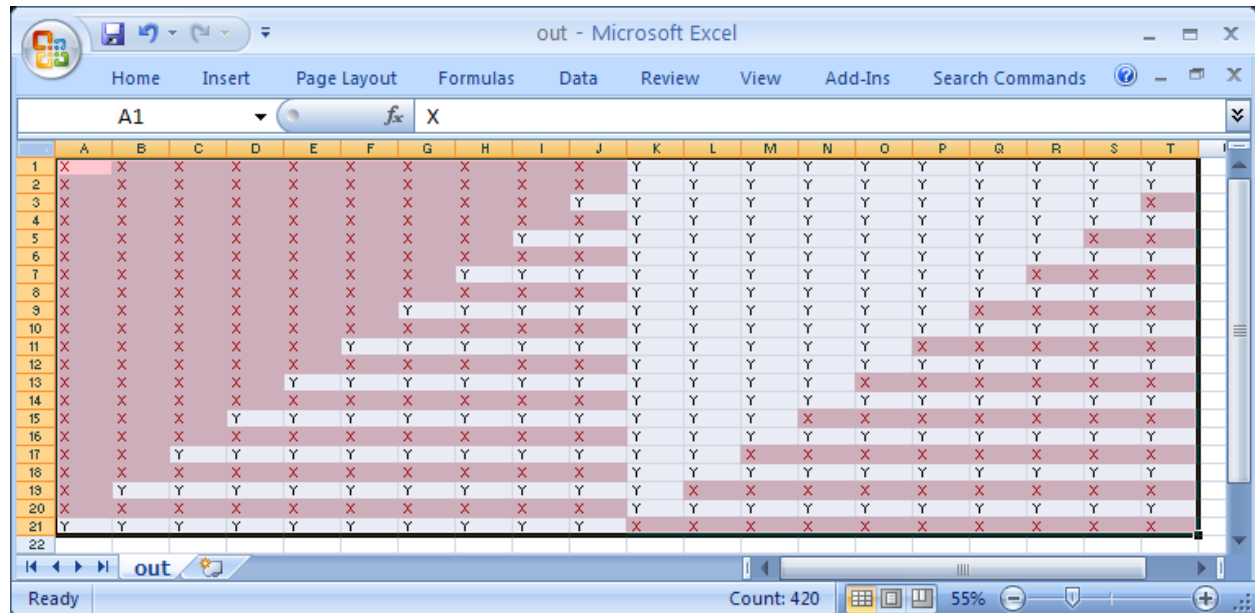

We see that there are ten rows corresponding to the ten runs of the program, each with 10 Xs followed by 10 Ys. Obviously, one is much smaller than $\frac{20!}{10!.10!}$. Running the program longer sometimes gives a few other schedules, but still falls very short of the number of possible thread schedules.

Systematic Concurrency Testing

To address this issue, we developed a concept of systematic concurrency testing and a supporting tool called CHESS (http://research.microsoft.com/projects/chess/). CHESS takes control over thread scheduling via API-level shims (both for Win32 and .NET) and then systematically explores the thread schedules of a concurrent test case. For effectively enumerating the set of thread schedules, which is overwhelmingly large even for simple programs, CHESS uses a new prioritization technique called "preemption bounding". The basic idea is that CHESS explores all thread interleaving in a concurrent test case up to a bounded number of preemptions P, a parameter supplied by the user.

Preemption bounding has three key advantages. First, the number of thread schedules with at most P preemptions is *polynomial* in the number of steps in the schedule (but still exponential in P). This makes it feasible for CHESS to complete its enumeration for reasonably large tests for small values of P. In addition, CHESS employs various model checking techniques (such as partial-order reduction) to further reduce the search space. The second advantage is that preemption bounding enables a very usable notion of coverage. If CHESS terminates without finding any bugs within P preemptions then any bug in the test case requires at least P+1 preemptions. This provides an intuitive estimate of the complexity of the bugs remaining in the system and the probability of their occurrence in practice. Last and not the least, in our experience applying CHESS, we find that many bugs are found with small values of P provided the preemptions take place just at the right point. The systematic exploration of CHESS is good at discovering these places.
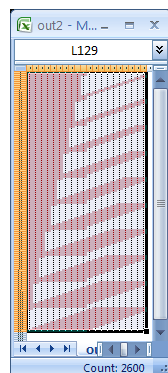
Let's look at the spreadsheet resulting from running CHESS on our example program with a preemption bound of one. The result is 21 different thread schedules yielding 11 different output strings (there are duplicates because there are different underlying thread schedules that yield the same string). We can see that CHESS is systematically preempting the main thread to run the child thread. However, once CHESS has spent its one preemption, it must run the program to completion without further preemptions (except those forced by a blocking operation or thread termination).

On the left is the visualization from running CHESS on the example with a preemption bound of two, which results in 130 thread schedules. In this case, CHESS gets to choose when to interrupt the main thread, then how long to let the child thread run before preempting a second time. This level of control over a concurrent execution is unprecedented and gives testers a big new lever with which to test concurrent programs.

Reproducible Concurrency Bugs

The second difficult part of testing concurrent programs is reproducing a bug. There are numerous horror stories about concurrency bugs that took weeks to track down. CHESS enables replay of a buggy execution by recording a small log file of the current thread schedule it is exploring. If an assert fails or some other catastrophic event occurs during the program execution, one simply re-runs CHESS in replay mode, where it uses the log file to replay the execution. This can be done in the programmer's favorite debugger, single stepping through the buggy execution as often as needed. Also, the user can instruct CHESS to break on each context switch or preemption recorded in the log file or simply use F5 to skip ahead in the trace.

CHESS Applications

We have applied CHESS to many concurrent programs within Microsoft. This section provides a summary of our experience.

1. **Dryad** (http://research.microsoft.com/research/sv/dryad/), an infrastructure developed by researchers at MSR Silicon Valley, allows a programmer to easily use the resources of a data center for running data-parallel programs. A Dryad programmer structures the computation as a directed graph in which sequential programs are graph vertices, while the channels are graph edges. CHESS found nine bugs in the implementation of Dryad channels, all of which were fixed by the Dryad developers.

2. **Singularity** (http://research.microsoft.com/os/Singularity/) is an operating system built at Microsoft Research. We integrated CHESS in Singularity and enabled the testing of Singularity applications. In addition, we tested the Singularity operating system itself using CHESS and found a serious performance error that was fixed within a week of its discovery.

3. **Microsoft Parallel Extensions to the .NET Framework** provides libraries and runtime mechanisms to support high-level high-productivity concurrent programming. CHESS has found four bugs in the implementation of this framework. We have delivered CHESS to the product group responsible for these extensions. The capability of systematic search and the coverage metric based on preemption-bounding has enabled the test team in the product group to think about their test engineering in a deeper and fundamentally more principled way.

The User Interface and Debugger

While CHESS provides an unprecedented level of control over the execution of a concurrent program, much more is needed to make concurrent debugging a first-class activity. In particular, we need to radically rethink the debugging experience. Graphical user interfaces for debugging are tuned for viewing a single thread at a time. Support for viewing concurrent activities has been added as an afterthought, rather than designed in as central to the debugging experience. As a result, it is hard, if not impossible, to understand how different threads in an application are interacting with each other.

The state of a multi-threaded program is the product of the states of each of the threads (plus their shared memory). Each thread has its own program counter (PC) and program stack. Our experience indicates that most concurrency bugs require viewing the interactions of a handful of threads. Figure 1 shows a snapshot of the Concurrency Explorer interface that comes with CHESS, in which it is possible to view the source code and stacks of several threads simultaneously. Note that the view shows a consistent state of the concurrent program execution, displaying the current PC of each thread (green line). Below the source code view for a thread, is list of procedures on the thread's stack (blue line). We believe that such a multi-source/stack view should be central in any concurrency debugger.

Concurrency Explorer improves upon the existing debuggers in its handling of breakpoints as well. Consider how Visual Studio treats breakpoints in concurrent programs. When we set a breakpoint in a source file of a concurrent program, the first thread to reach that breakpoint will cause execution of the entire program to halt. But what happens to the other threads? What do we know about them? Where do they stop? The answer is, it depends on the thread schedule, which is not under the control of the debugger. Moreover, when the user takes a single step in the focus thread, many steps of some other thread (or threads) may execute during that single step. If one of those other threads encounters a breakpoint, the entire GUI shifts focus to that thread. Or worse, one of these other threads might hit the temporary breakpoint inserted for the single step of the focus thread. The result is great unpredictability in the conceptually simple scenario of single stepping execution.

In comparison, as the user single steps a program under the control of CHESS, the focus thread advances one step, the other threads remain at their current PC. When the focus thread blocks, the user simply switches focus to an unblocked thread and continues single stepping. In Concurrency Explorer, the number of mouse clicks to take a single step in the thread schedule of a concurrent program execution is at most two (and usually one). In contrast, emulating this conceptually-simple behavior in existing debuggers, such as Visual Studio or windbg, requires the user to freeze and unfreeze threads repeatedly.

Much more is possible. For example, we might like to step to the next blocking operation in the focus thread, or to the next context switch or preemption in the current thread schedule. Such concurrency-aware debugging operations will give programmers a new vocabulary with which to more easily explore concurrent executions.

Concurrency APIs

Programming language support for concurrency is highly fractured and domain-dependent. Concurrent programming is fundamentally more difficult than sequential programming, but we lack programming abstractions, such block structured programming and object-orientation, which have made sequential programming much easier. As a result, much concurrent programming takes place through the use of APIs/libraries/frameworks that supply concurrency abstractions. Microsoft is

investing heavily in new libraries for concurrent data structures, user-level task schedulers, transactional memory, asynchronous message passing, distributed systems, etc. Each of these libraries builds on lower-level libraries such as Win32, System.Threading, etc, which are non-trivial APIs with complex concurrency semantics.
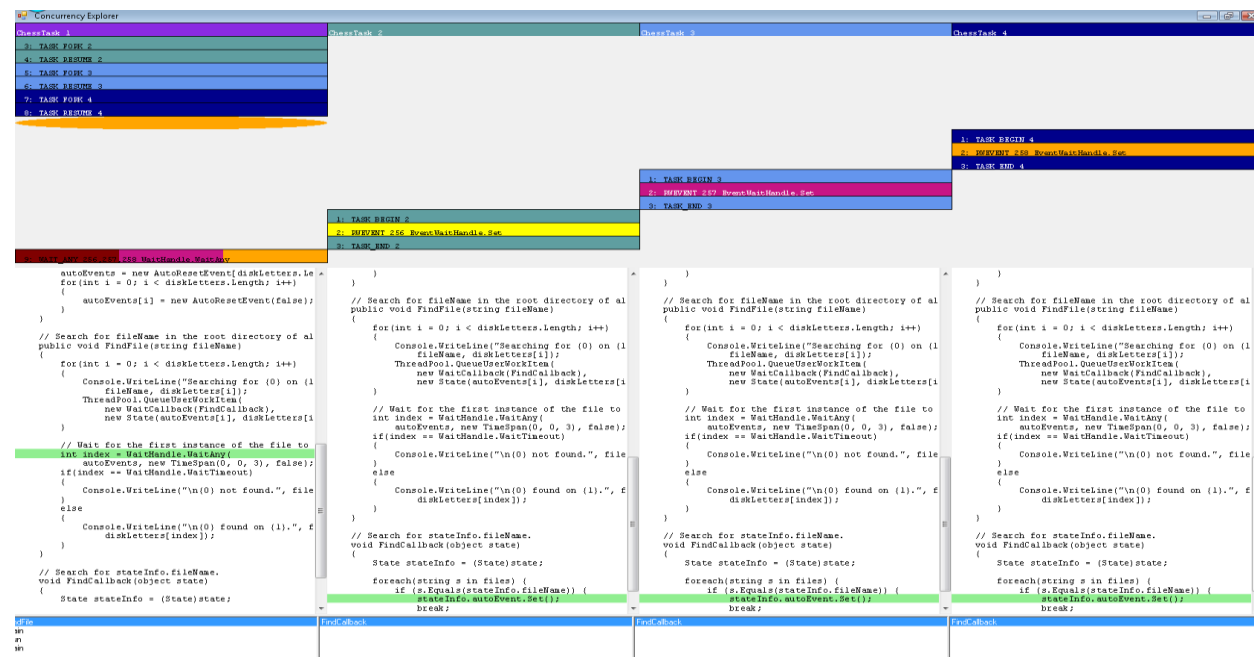

Figure 1. Viewing Four Threads with Concurrency Explorer

Unfortunately, these low-level APIs were not designed with debugging and testing in mind. First, the APIs are very complex, both for developers to use and for testers to test. Second, the APIs are inherently nondeterministic, having many outcomes (depending on whether or not an operation blocks or not, for example). Finally, the APIs are not *observable*. For instance, there is no method to reliably determine when an operation has blocked and under what conditions it can be unblocked. This lack of observability means that it is difficult to *control* the execution of programs using a concurrency API. This has impacts up and down the software stack.

To address this problem we use a simple abstraction of tasks and synchronization variables. A task is an entity that can execute asynchronously, such a thread, thread pool item, or a timer thread. A synchronization variable is an object on which a task performs a synchronization operation. By mapping the operations of a concurrency API into this abstraction of tasks and synchronization variables, CHESS is able to control a diverse set of concurrency primitives. Unfortunately, this mapping must be done for each concurrency API that one wishes to control using CHESS.

We have created CHESS wrappers for Win32 threading, .NET's System.Threading APIs, as well as the concurrency primitives of the Singularity/Singularity operating system. These wrappers allow the CHESS analysis engine to control the non-determinism inherent in concurrency APIs. A huge challenge is to scale this effort to the many other concurrency APIs. Towards this end, we are developing a set of recommendations for developers of concurrent APIs to follow to make their concurrent APIs testable via tools like CHESS.

Conclusion

We have painted a picture of a possible future for testing and debugging of concurrent programs, one that is within our reach but requires coordination at multiple levels of the software stack, from the GUI we use to debug concurrent programs all the way down to the low-level APIs used for concurrent programming. Fundamentally, concurrent programming is a much more difficult problem than sequential programming. However, we believe that simple ideas, such as the abstraction of a concurrent execution as a schedule of thread events, backed up by new analysis techniques such as CHESS, can give us an unprecedented ability to systematically test and debug concurrent programs.