

CHES: Systematic Testing of Multithreaded Software

Shaz Qadeer

Microsoft Research

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 mainstream desktop computers, we are likely to 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 this task lies in reasoning about the interaction between concurrently executing threads. Nondeterministic thread scheduling makes it extremely difficult to reproduce behavior from one run of the program to another. As a result, the process of debugging concurrent software becomes tedious resulting in a drastic decrease in the productivity of programmers. Since concurrency is both important and difficult to get right, it is imperative that we develop techniques and tools to automatically detect and pinpoint errors in concurrent programs.

The current state-of-the-art in testing concurrent software is unsatisfactory for two reasons. The first problem is that testing is not systematic. A concurrent test scenario is executed repeatedly in the hope that a bad thread schedule will eventually happen. Testers attempt to induce bad schedules by creating thousands of threads, running the test millions of times, and forcing context switches at special program locations. Clearly, these approaches are not systematic because there is no guarantee that one execution will be different from another. The second problem is that a bad schedule, if found, is not repeatable. Consequently, the programmer gets little debugging help once a bug has been detected.

The CHES project at Microsoft Research attempts to address these limitations by providing systematic, repeatable, and efficient enumeration of thread schedules, in a style of program verification commonly known as *model checking* [1, 8]. CHES instruments the program execution in order to get control of the scheduling. The instrumentation allocates a semaphore for each thread that is created, and preserves the invariant that at any time every thread but one is blocked on its semaphore. Thus, the underlying operating system scheduler is forced to schedule the one thread that is not blocked. When this thread reaches the next point in its execution that is instrumented, a different thread can be scheduled by performing appropriate operations on their respective semaphores. This mechanism allows CHES to implement a simple depth-first search of thread schedules, thereby providing the guarantee that each thread schedule generated is different. Moreover, if a schedule results in an error, it can be replayed ad infinitum.

Clearly, the number of possible schedules for realistic concurrent programs is huge. For example, the number of executions for a program with n threads, each

of which executes k steps, can be as large as $\Omega(n^k)$. This asymptotic complexity is the essence of the so-called *state-explosion problem* that has confounded attempts to scale model checking to large programs.

To combat the state-explosion problem, researchers have investigated reduction techniques such as partial-order reduction [4] and symmetry reduction [6, 5]. Although these reduction techniques help in controlling the state explosion, it remains practically impossible for model checkers to fully explore the behaviors of large programs within reasonable resources of memory and time. For such large programs, model checkers typically resort to heuristics to maximize the number of errors found before running out of resources. One such heuristic is *depth-bounding* [9], in which the search is limited to executions with a bounded number of steps. If the search with a particular bound terminates, then it is repeated with an increased bound. Unlike other heuristics for partial state-space search, depth-bounded search provides a valuable coverage metric—if search with depth-bound d terminates then there are no errors in executions with at most d steps.

Since the number of possible behaviors of a program usually grows exponentially with the depth-bound, iterative depth-bounding runs out of resources quickly as the depth is increased. Hence, depth-bounding is most useful when interesting behaviors of the program, and therefore bugs, manifest in small number of steps from the initial state. The state space of message-passing software has this property which accounts for the success of model checking on such systems [3, 7]. In contrast, depth-bounding does not work well for multithreaded programs, where the threads in the program have fine-grained interaction through shared memory. While a step in a message-passing system is the send or receive of a message, a step in a multithreaded system is a read or write of a shared variable. Typically, several orders of magnitude more steps are required to get interesting behavior in a multithreaded program than in a message-passing program.

CHES implements a novel algorithm called *iterative context-bounding* for effectively searching the state space of a multithreaded program. In an execution of a multithreaded program, a *context switch* occurs when a thread temporarily stops execution and a different thread starts. The iterative context-bounding algorithm bounds the number of context switches in an execution. However, a thread in the program can execute an arbitrary number of steps between context switches, leaving the execution depth unbounded.

Furthermore, the iterative context-bounding algorithm distinguishes between two kinds of context switches — preempting and nonpreempting. A *preempting* context switch, or simply a *preemption*, occurs when the scheduler suspends the execution of the running thread at an arbitrary point. This can happen, for instance, at the expiration of a time slice. On the other hand, a *nonpreempting* context switch occurs when the running thread voluntarily yields its execution, either at termination or when it blocks on an unavailable resource. The algorithm bounds the number of preemptions while leaving the number of nonpreempting context switches unbounded.

Limiting the number of preemptions has many powerful and desirable consequences for systematic state-space exploration of multithreaded programs. First, bounding the number of preemptions does not restrict the ability of the model checker to explore deep in the state space. This is due to the fact that, starting from any state, it is always possible to drive a terminating program to completion (or to a deadlock state) without incurring a preemption. As a result, a model checker is able to explore interesting program behaviors, even with a bound of zero!

Second, for a fixed number of preemptions, the total number of executions in a program is *polynomial* in the number of steps taken by each thread. This theoretical upper bound makes it practically feasible to scale systematic exploration to large programs without sacrificing the ability to go deep in the state space.

Finally, iterative context-bounding finds a trace exposing the error with the smallest number of preemptions. As most of the complexity of analyzing a concurrent error-trace arises from the interactions between the threads, the algorithm naturally seeks to provide the simplest explanation for the error. Moreover, when the search runs out of resources after exploring all executions with c preemptions, the algorithm guarantees that any error in the program requires at least $c+1$ preemptions. In addition to providing a valuable coverage metric, it also provides the programmer with an estimate of the complexity of bugs remaining in the system and the probability of their occurrence in practice.

An important aspect of the CHES implementation is its dynamic partitioning of the set of program variables into data and synchronization variables. Typical programs use synchronization variables, such as locks, events, and semaphores, to ensure that there are no data-races on the data variables. Motivated by this observation, CHES introduces context switches only at accesses to synchronization variables and verifies that accesses to data variables are ordered by accesses to synchronization variables in each explored execution using the Goldilocks [2] data-race detection algorithm.

Our evaluation provides empirical evidence that a small number of preemptions is sufficient to expose nontrivial concurrency bugs. CHES has uncovered 9 previously unknown bugs in several real-world multithreaded programs. Each of these bugs was exposed by an execution with at most 2 preemptions. Also, for a set of programs for which complete search is possible, few preemptions are sufficient to cover most of the state space. This empirical evidence strongly suggests that when faced with limited resources, which is invariably the case with model checkers, focusing on the polynomially-bounded and potentially bug-yielding executions with a small preemption bound is a productive search strategy.

References

1. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

2. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *FATES/RV 06: Formal Approaches to Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 193–208. Springer-Verlag, 2006.
3. P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186, 1997.
4. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. LNCS 1032. Springer-Verlag, 1996.
5. Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE 01: Automated Software Engineering*, pages 254–261, 2001.
6. C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.
7. Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation*, pages 75–88, 2002.
8. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Fifth International Symposium on Programming*, Lecture Notes in Computer Science 137, pages 337–351. Springer-Verlag, 1981.
9. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2002.