

A Method for Providing Complete Access to the Concurrent Programming Model

Jerry James
james@ittc.ku.edu

Doug Niehaus
niehaus@ittc.ku.edu

Information and Telecommunication Technology Center
EECS Department
University of Kansas

Abstract

We describe an environment within which the programmer has access to and control over concurrency. This control can be manifested by requiring a program to reproduce a previously manifested interleaving of instructions in concurrent threads, or by exploring the space of possible interleavings. Scientifically valid experimentation with concurrent software requires such control to make the experiments reproducible. Formal methods for showing properties of programs can also make use of this capability to force the program to exhibit its behavior under a range of interleaving scenarios.

1 Introduction

Concurrent computations are vital components of an ever-widening range of activities, from the embedded control systems in automobiles to transactions in financial markets. Software implementing concurrent computations carries with it a danger because the most widely used concurrency models, programming frameworks, and execution environments provide insufficient support to conduct scientifically valid experiments evaluating the behavior of concurrent software. Currently popular evaluation methods are scientifically invalid because they do not provide adequate access to the programming model to enable a developer to conduct reproducible experiments.

Existing concurrent programming models do not support reproducible experiments because they do not permit adequate control over how components of a multithreaded computation are interleaved, which can affect the computation results. Current programming models and execution environments conceal several aspects of the environment affecting interleaving, making the behavior of a concurrent computation vary in ways that appear random to the user.

The set of correct interleaving scenarios is a small fraction of all possible scenarios for most realistic concurrent computations. This is why concurrent programming models provide concurrency control methods with which the developer can constrain the possible interleaving scenarios to only those producing correct results. Incorrect use of these concurrency control methods can result in a computation which will produce incorrect results (faults) under some scenarios. Such computations will, however, produce these faults intermittently and irreproducibly for two reasons. First, there is no mechanism for knowing the exact scenario that produced the fault. Second, even if a developer is confident that a hypothesized scenario produced the fault in question, there is no support for reproducing a specified interleaving scenario. Faults

are intermittent because the execution environment chooses scenarios independently and under a number of randomizing influences.

This paper describes a method we have used to create an environment supporting reproducible experiments with concurrency. The properties of this environment include accurate recording of concurrency scenarios and correct replaying of recorded scenarios. The existing work is a simple proof of concept for simple computation types. Future work will include methods for constructing specific scenarios and expanding the set of computation types for which we can support reproducibility.

Our method is complementary to some formal methods in a number of ways. First, we record concurrency scenarios that can be analyzed by some formal methods. Second, our approach can produce any scenario identified as interesting by a formal method. This capability, when combined with algorithms for checking all concurrency scenarios (e.g., the ExitBlock [3] algorithm or model checking [13]) can provide a provably complete testing framework.

2 Related Work

Much of the related work we have found involves approaches that concentrate on source level transformations [7, 11, 15]. One such approach produces two versions of the program, one recording a scenario and the other reproducing it. The approach depends upon having a specific transformation method for each synchronization mechanism used by the program. In contrast, our approach uses a similar recording transformation but includes the mechanisms necessary for reproducing any scenario in the programming environment. The advantage of our approach is that it supports reproducibility for all synchronization mechanisms, both current and future, without modification.

Albertsson et al have produced a machine simulator for Linux, and simulation-based debuggers for soft real-time applications [2, 1]. Executions of the simulated machine are deterministic, since deterministic timing models are used. This means that executions are reproducible, but it also means that a given concurrent program will execute in only one way. In contrast, our techniques allow exploration of the space of all possible concurrent execution scenarios of a given program.

The Rivet Java Virtual Machine contains a replay mechanism that has some similarities with the one we use [3]. Their approach records the instruction stream in detail, and then executes it as recorded. In our approach, we only record events noting the entry to basic blocks and other events influencing the execution scenario, including context switches and signals. Then, we use this set of recorded events to guide a subsequent execution through the same scenario. Our technique also supports the construction of interleaving scenarios not exhibited by any actual execution of the program, in order to experiment with program behavior.

Our work is compatible with several testing techniques explored in the literature, such as deterministic testing, prefix-based testing [14], and reachability testing [9]. The capabilities our work provides enables these techniques to be implemented in simpler ways than is currently possible.

There is a large body of more broadly related work in the literature. Summaries of existing research are given for debugging of parallel systems in [8], and for debuggers based on execution replay in [5].

3 Implementation

In order to support both model checking and execution reproduction of concurrent programs, we need several capabilities. First, we need a way of specifying *which* execution scenario of the program is of interest.

Second, we need a tool that will guide the program through the execution scenario required. For reproducibility purposes, we also need a way of recording the execution scenario taken by a running program. In the following sections, we describe our construction of each of these capabilities in turn, and then describe a thread library, BThreads, that takes advantage of these capabilities.

3.1 Execution Specification

Fully specifying an execution scenario can be done by listing the exact sequence of CPU instructions executed, along with markers indicating the points at which signals are delivered and at which context switches take place, as well as all data transferred between the kernel and the program due to system calls. However, while sufficient, such a complete specification is not necessary. Furthermore, such complexity would make recording or constructing execution specifications an excessively onerous task.

Instead, we take advantage of the fact that, in the absence of context switches and signals, most user code executes deterministically. In particular, when the CPU enters a basic block¹, we assume that it executes deterministically until it exits the basic block. Hence, instead of a stream of CPU instructions, we can specify an execution with a stream of basic block labels. This approach considerably reduces the size and complexity of a specification. Signal and context switch events are inserted as appropriate labels into the stream. Each is labeled with the CPU's program counter at the time of the event which places it within the current basic block. In addition, signals are labeled with the signal number, and context switches are labeled with the old and new thread identifiers. It is possible to further compress the label stream in a number of ways, which we will implement as part of our future work.

3.2 Program playback

The problem of playing back a specified execution scenario can be solved in various ways. For example, one could construct a whole machine simulator, and play a specified execution instruction by instruction (or basic block by basic block). However, a simpler approach is possible.

Debuggers already have the ability to “single step” through a program execution, and to stop an execution at arbitrary places in the code via breakpoints. What current debuggers are missing is the ability to specify arbitrary actions to be taken when a breakpoint is reached. A previous project, SmartGDB, rectified this lack by integrating the Tcl/Tk scripting and GUI support framework with the command language of GDB, the GNU debugger, producing a hybrid scripting language within which it was possible to attach procedures containing GDB commands to breakpoints [6]. SmartGDB also contained an interface to a simple API in a threading package that would support reproducing execution scenarios by permitting the controller to *force a context switch among threads at a breakpoint*. SmartGDB was targeted at several user-level thread packages and the Linux kernel threads package of the time.

Since then, the GDB development team has produced Insight, an extension to GDB that is similar to SmartGDB in supporting a command language including a superset of the GDB and Tcl commands, and providing access to Tk widgets from the debugger to support GUIs. It also provides limited support for executing debugger code at a breakpoint, but it does *not* support attaching an arbitrary procedure written in the hybrid Tcl/GDB command language, which proved such a powerful feature of SmartGDB. We recently added this feature to Insight, calling the new version Clever Insight (CI).

With the ability to attach an arbitrary procedure to a breakpoint, it was comparatively simple to demonstrate the ability to reproduce a recorded CPU-bound concurrency scenario. CI reads the recorded event

¹A *basic block* is defined in the compiler literature as a consecutive series of CPU instructions with no branches in or out, other than the first and last instruction.

history and sets a breakpoint where the next context switch or signal event occurs. Then, as the computation executes, CI checks the history being generated to see if it matches the proper context. When it does, CI forces the specified context switch or delivers the specified signal [10].

3.3 Program instrumentation

Debugging of concurrent software is difficult, especially when a bug is caused by incorrect use of synchronization mechanisms. A particular bug may manifest rarely, since it only occurs when context switches take place within a very small range of addresses. Finding such bugs would be much easier if a running program could produce one of the execution specifications described above, thereby enabling the use of Clever Insight to produce the same execution under the control of a debugger.

Let us set aside the necessity of recording context switches and signals for the moment, and concentrate on producing the stream of basic block labels needed for an execution specification. The GNU compiler, GCC, already has a similar capability, used for profiling program performance. It creates counters for every basic block, and increments them on every execution of the basic block. We modified this feature to emit an *event* at the entry of each block, using the unique identifiers already available for the profiling counters. The resulting stream can be directed to a file, and subsequently fed into Clever Insight for reproduction.

It is also necessary to instrument context switches and signal deliveries. However, the mechanism by which these events will be instrumented will necessarily vary from thread library to thread library. In the next section, we describe the approach we took for a user-level POSIX-compliant thread library, BThreads. Part of our future work will be to take a similar approach for the native kernel-level POSIX threads on Linux platforms.

3.4 BThreads

As an illustration and test of our ideas, we constructed BThreads [12], a user-level replacement for the POSIX threads (pthreads) library on Linux. This library depends fundamentally on an innovative use and generalization of the Reactor software pattern [4]. A Reactor based computation is structured as a set of handlers that the Reactor executes in response to the occurrence of specific events. The Reactor thus serves to organize execution of a set of handlers for external events including I/O operations, signal delivery, and timer expiration. However, it is also simple to treat all major components of an application as handlers, and to transfer control from one component to another through the Reactor. The fundamental insight underlying our innovative generalization of the Reactor pattern is that it can be used to abstract *all* of the actions of a computation affecting reproducibility. In effect, the Reactor becomes a virtual machine supporting reproducible concurrency.

We have made it possible to switch context between threads whenever control passes through the Reactor object, and wrote wrappers for potentially blocking system calls that operate on file descriptors. These wrappers register the file descriptor of interest with the controlling Reactor object, then call the BThreads scheduling function, possibly resulting in a context switch. Finally, when the `select()` system call at the heart of the Reactor object chooses a file descriptor to handle, the associated handler causes a context switch to the thread that registered that file descriptor. The result is a preemptive user-level thread library, called BThreads.

The BThreads library was constructed so that, in conjunction with the GCC extension discussed in Section 3.3, it would be suitable for recording and replaying concurrent executions. We provided a control method in BThreads enabling Clever Insight to force a context switch to a specific thread and force a signal delivery to a particular thread at any breakpoint. The BThreads library itself contains the instrumentation

needed to push context switch and signal labels onto the appropriate event stream as they occur. Initial testing with some small existing pthreads-based programs show that we can reliably reproduce concurrent program behavior. We are in the process of scaling our techniques up to larger applications.

3.5 Evaluation

Our initial evaluations have been performed with small focused tests. We have written several tests with deliberate race conditions. Successive executions of these tests show that the races are resolved in different ways in various executions. With the tools described above we are able to capture an execution scenario and guide it to replay in exactly the same way, thus demonstrating the reproducibility of arbitrary scenarios. While we are able to make small modifications to such specifications by hand, in order to produce new tests to perform, we are currently lacking a way to check that a given specification is a possible execution of the program in question. We are currently beginning to explore solutions to this problem.

4 Conclusions and Future Work

We have produced a set of tools for producing specified interleavings of multithreaded software. We have not produced a formal process whereby provably correct concurrent programs can be produced, but we have placed concurrency completely under the programmer's control, thereby eliminating the stochastic nature of current testing methods. We made it possible to conduct scientifically valid experiments, because the execution scenarios are reproducible.

Future work will extend our tools in a number of ways. First, we currently deal only with compute-bound threads and OS signals. We will extend our work to relevant system calls, and file and network I/O. In effect, we will reproduce the environment of a thread, as well as the synchronous and asynchronous actions of that thread. Second, we will extend our work into the Linux kernel. Having support for specified executions embedded in the programming environment is a vital step forward, in our view. Third, it is possible to compress the label stream significantly. As our work scales up to larger and larger applications, we expect the overhead of producing the label stream to become more onerous, so such compression is important. We have a theoretical result, to be provided in a future work, which shows that a very small amount of information suffices to reproduce a given interleaving scenario. Finally, we are working on a tool that will assist in the interactive creation of an interleaving scenario to be executed. This tool is built on top of Clever Insight, so it includes the full power of the underlying debugger. It will, in effect, execute the program as the interleaving scenario is specified, thereby enabling the operator to determine that a given step is incorrect or inappropriate.

References

- [1] ALBERTSSON, L., AND MAGNUSSON, P. S. Simulation-based temporal debugging of Linux. In *Proc. of the 2nd Real-Time Linux Workshop* (Lake Buena Vista, FL, 2000).
- [2] ALBERTSSON, L., AND MAGNUSSON, P. S. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Proc. of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (San Francisco, CA, 29 Aug.–1 Sept. 2000), pp. 191–8.

- [3] BRUENING, D. L. Systematic testing of multithreaded Java programs. Master's thesis, EECS Department, MIT, 21 May 1999.
- [4] COPLIEN, J. O., AND SCHMIDT, D. C., Eds. *Pattern Languages of Program Design*. Addison-Wesley, 1995, ch. 29. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching.
- [5] DIONNE, C., FEELEY, M., AND DESBIENS, J. A taxonomy of distributed debuggers based on execution replay. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (Sunnyvale, CA, 9–11 Aug. 1996).
- [6] HALBHAVI, S. Thread debugger implementation and integration with the SmartGDB debugging paradigm. Master's thesis, ITTC, EECS Department, University of Kansas, 1995.
- [7] HANSEN, P. B. Reproducible testing of monitors. *Software—Practice and Experience* 8, 6 (Aug. 1978), 721–9.
- [8] HUSELIUS, J. Debugging parallel systems: A state of the art report. Tech. Rep. 63, Department of Computer Science and Engineering, Mälardalen University, Sept. 2002.
- [9] HWANG, G.-H., TAI, K.-C., AND HUANG, T.-L. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering* 5, 4 (1995), 493–510.
- [10] MALLAVADI, S. A thread debugger for testing and reproducing concurrency scenarios. Master's thesis, ITTC, EECS Department, University of Kansas, Jan. 2003.
- [11] OLSSON, R. A. Reproducible execution of SR programs. *Concurrency: Practice & Experience* 11, 9 (Aug. 1999), 479–507.
- [12] PENUMARTHY, S. Design and implementation of a user-level thread library for testing and reproducing concurrency scenarios. Master's thesis, ITTC, EECS Department, University of Kansas, Dec. 2002.
- [13] STOLLER, S. D. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer* 4, 1 (Oct. 2002), 71–91.
- [14] TAI, K. C., AND CARVER, R. H. Deterministic execution testing and debugging of concurrent programs. In *Proc. 1989 Pacific Northwest Software Quality Conference* (1989), pp. 170–82.
- [15] TAI, K. C., CARVER, R. H., AND OBAID, E. E. Debugging concurrent Ada programs by deterministic execution. *IEEE Trans. Softw. Eng. SE-17*, 1 (Jan. 1991), 45–63.