Special Session on Debugging

Yoav Hollander[1], Alan Hu[2], João Lourenço[3] and Ronny Morad[4]

[1] Cadence, Israel
yoavh@cadence.com
[2] University of British Columbia, Canada
ajh@cs.ubc.ca
[3] New University of Lisbon, Portugal
Joao.Lourenco@di.fct.unl.pt
[4] IBM Research – Haifa, Israel
morad@il.ibm.com

1 Introduction

In software, hardware, and embedded system domains, debugging is the process of locating and correcting faults in a system. Depending on the context, the various characteristics of debugging induce different challenges and solutions. Post-silicon hardware debugging, for example, needs to address issues such as limited visibility and controllability, while debugging software entails other issues, such as the handling of distributed or non-deterministic computation. The challenges that accompany such issues are the focus of many current research efforts. Solutions for debugging range from interactive tools to highly analytic techniques. We have seen great advances in debugging technologies in recent years, but bugs continue to occur, and debugging still encompasses significant portions of the lifecycles of many systems. The session covered state-of-the-art approaches as well as promising new research directions in both the hardware and software domains.

2 Challenges in Debugging

The first talk, by Yoav Hollander of Cadence, centered around two points—(a) debugging is a hard problem that is getting harder, and (b) debug automation is a topic that is worth investing in now.

The talk focused on hardware and embedded software debugging, though debugging activity is similar in domains such as diagnostics, accident investigations, police detective work, and psychotherapy. The common goal, in all of these examples and more, is to understand why something bad happens in a complex system.

The debugging problem seems to demand a bigger slice of the verification pie now more than ever. In hardware debugging, the bug-finding techniques (constrained random testing, formal, etc.) have reached a good level so debugging has become the bottleneck. In the software domain, the verification requirements (such as coverage filling) have traditionally been lower, so most of what is called verification in this case is actually debugging.

System debugging is particularly difficult, for three main reasons. First, most systems, almost by definition, are big and heterogeneous. Their components, some of which are only lightly verified, come from many groups. Furthermore, the components come from many different disciplines, including digital, analog, kernel software, and middleware software. None of these disciplines seem to have the full picture of what is necessary for debugging. These challenges are all compounded by the parallelism within the system.

The second reason for the difficulty is that the running environments (emulation, simulation, post-silicon, etc.) each have their own set of problems—including being slow, having limited visibility, and being non-repeatable.

Finally, there is lots of incidental complexity. Your compile script breaks, you get the wrong version, you do not understand the bug well and have to re-fix, and all of this takes time.

System bugs come in several variants: system bring-up bugs, where nothing works; interaction bugs, where simple scenarios work but fail when they interact; and performance/power bugs, where some scenarios take more time and/or power than expected. Performance/power bugs are probably the hardest to detect and repair.

How can we address these challenges? One solution is automatic debugging. Much academic work has been dedicated to this field, including the work of Andreas Zeller [1] and others, some of which is described in the next two sections. Some of the work suggests finding the correlations to failing runs, simplifying failing runs, dynamic slicing, formal debugging, reverse debugging, and more. The time has come to bring such tools into the broader user domain. Cadence is already active in this area.

3 BackSpace: Formal Methods for Post-Silicon Debugging

Alan Hu, of the University of British Columbia, gave the second talk of the session. Like the preceding talk, this one also highlighted the critical importance of debugging, but also – coming from an academic perspective – emphasized that this is an exciting, new, wide-open research area. In contrast to the comprehensive overview provided by the preceding talk, this one drilled deep and narrow, focusing on a specific part of the post-silicon debug problem and presenting novel research results on using formal verification to address it.

In particular, post-silicon debugging is the task of determining what went wrong when a fabricated chip misbehaves. In contrast to ordinary, pre-silicon verification, what is being debugged is the actual silicon chip rather than a simulation model; but, in contrast to manufacturing test, the goal is to look for design errors in a newly designed chip, rather than random manufacturing defects in high-volume production. For example, imagine a new processor or SoC design. During pre-silicon verification, extensive simulation, formal verification, and possibly emulation were done to eliminate as many bugs as possible, but with the complexity of modern designs, some bugs will escape to the silicon. Also, the physical silicon is the first opportunity to validate the approximate models used for assessing electrical behavior and timing. So, it is imperative to thoroughly validate the first silicon, before ramping into high-volume production.

There are many facets to the post-silicon debugging problem, but the talk focused on a specific one: how to derive a trace of what actually happened on chip, leading up to an observed buggy behavior. Continuing the example, imagine that we discover a bug – perhaps the chip runs fine during simple bring-up tests, but crashes roughly 10% of the time after one minute of running a key software application. How do we debug this? The critical task is to uncover what is happening on the chip leading up to the crash, but unlike simulation, we cannot see the signals on-chip. Furthermore, since the silicon chip runs roughly a billion times faster than a full-chip RTL simulation, we cannot replay the failing test case ab initio on the simulator: the one minute of on-chip run time would be on the order of 60 billion seconds (almost 2000 years) of simulation time! Almost all chips have some on-chip test and debug support, e.g., scan chains and maybe trace buffers, so we can see (at least some of) the signals on-chip, but only after the chip crashes (or we otherwise stop the chip). Guessing exactly when to trigger the trace buffer or try to get a scan dump before the bug manifests, so that we can start to understand what went wrong, is a painstaking and inexact artform.

BackSpace is a revolutionary solution to this problem, providing the effect of allowing the silicon to run full-speed, yet stop at any point and go backwards, computing the execution the led to the bug, much like in a software debugger [2]. Underlying the basic BackSpace approach are some simplifying assumptions: that we are debugging a functional design error, so that the silicon corresponds to the RTL (or some other formally analyzable model); that there is a programmable breakpoint mechanism; that once the chip is stopped (breakpoint or bug), it is possible to dump out the state of the chip, e.g., via scan chains; that we can add some additional signature/history bits to the chip state; and that the test causing the bug can be run repeatedly, with the bug occurring reasonably often, e.g., at least once every few minutes. With these assumptions, the basic BackSpace algorithm proceeds as follows. From a crash state or breakpoint, we scan out the state of the chip, including the signature bits. From this scanned-out state, we use formal analysis to compute the pre-image of that state – this is the set of states that could possibly occurred in the preceding cycle, and the challenge is to determine which one actually happened. The signature bits are to keep this set small. Then, the BackSpace algorithm automatically tries each of these states as a possible breakpoint and re-runs the failing test on the silicon, repeatedly. When the correct predecessor state is chosen, the chip will (possibly after multiple tries) hit the breakpoint, thus giving us a predecessor state of the crash that is actually reached by the chip while executing the failing test. From that state, we can repeat this procedure indefinitely, computing an arbitrarily long trace of states that actually lead to the crash on-chip.

The original BackSpace paper demonstrated the theory, and subsequent work demonstrated the method on actual hardware, proving that the method works, even in the presence of non-determinism. However, the on-chip overhead was ridiculously high. The remainder of the talk was a whirlwind tour of recent, mostly as-yet-unpublished work on making BackSpace practical: reducing on-chip overhead by using partial-match breakpoint, accelerating the computation by prioritizing the pre-image states [3], handling the skids resulting pipelining the breakpoint logic, and doing BackSpace in the presence of electrical faults (thus breaking the assumption that the RTL matches the silicon). In most of these works, the key idea is the same as

the basic BackSpace computation, except that additional re-runs of the failing test are needed to compensate for the relaxed assumptions about the problem. In general, the BackSpace approach is a synergy between on-chip debug hardware, formal analysis, and the extremely high speed of the actual silicon in running test cases. Stepping back to a broader perspective, the second talk addressed the audience on multiple levels: the specifics of the BackSpace algorithms, of course, but more generally, the key idea that formal verification can aid post-silicon debug by automating complex reasoning about what is or is not possible and thereby extract maximum information from on-chip debug hardware. Another key insight is that the fast re-execution of tests on silicon means that repetition can be used to compensate for lack of observability. And most broadly, this talk was drilling deeply in a single direction, and found many promising research results, suggesting that there is a vast reservoir of interesting research to be tapped in post-silicon debug.

4 Debugging of Parallel and Distributed Programs

The final talk was given by João Lourenço, of the New University of Lisbon, and addressed software debugging. Debugging a program is a process of recognizing, identifying, locating, diagnosing and correcting deviations from a given specification. These deviations may be considered as *program errors*. Besides exhibiting sequential errors, concurrent programs do also exhibit concurrent errors. These errors are much harder to debug, making this activity orders of magnitude more complex than in sequential debugging.

One of the key issues in debugging parallel and distributed programs is the program *observation*, also orders of magnitude more complex than in sequential programs. While debugging, the following dimensions are main contributors to the higher complexity of observing concurrent programs [4].

The high number of interacting entities — leads to an exponential growth in the number of possible program states. To address this dimension, debuggers must be able to observe both local process/thread states and global consistent states. If the number of threads/processes scales up to hundreds or thousands, it may be necessary for the debugger to abstract sets of related threads/processes in groups, making these groups first-order entities in the debugging activity.

The intrinsic non-determinism — makes the program behavior dependent on local processor speeds, node workload and unpredictable communication delays. This dimension requires the debugger to provide support for detecting timing-based race conditions and to evaluate program correctness predicates (e.g., local and global assertions). Non-determinism is related to the *probe effect*, where the observation of the system interferes with the system itself and its behavior, making it possible for the debugger to both mask existing errors and trigger new ones. This requires the debugger to support reproducible program behavior, allowing repeatable coherent observation of the computations, including the erroneous ones.

The absence of a global state — as one can only make inferences on a concurrent program behavior and state based on consistent observations, and the absence of a global state makes it very hard do determine which observations are consistent and

which are not. To address this dimension, the debugger must make use of lightweight algorithms and non-intrusive techniques to identify consistent program states and observations.

A debugger needs not to fully support all the above dimensions to be useful in debugging concurrent programs (see Fig. 1). The most basic approach in distributed debugging relates to the interactive control of remote processes, achieving the observation of individual (threads/processes) and global program states. Bringing off support for reproducible program behavior, one can support repeatable observations, by way of trace and replay for deterministic re-execution of programs. Program steering can be used to analyze alternative, less probable paths, carrying out alternative observations and achieving systematic state space coverage. Some debugging infrastructures may also support observing consistent computations and detecting program properties by evaluating local and global predicates in consistent global states.

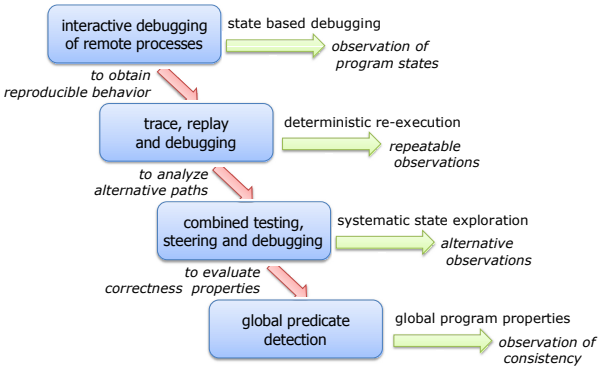

Fig. 1 Increasing functionality of concurrent debuggers.

Due to the new context of computing nodes having multi-core processors, and to the clustering of such nodes, it is vital that the support for debugging parallel and distributed programs evolve to the same maturity level as others current software development tools, such as IDEs. This evolution must take place along three main axes: debugging methodologies, e.g., state- and time-based debugging; debugging functionalities, e.g., observation and control; and abstraction levels, e.g., recognition of higher-level programming languages concepts, such as synchronization structures.

References

1. Burger, M., Lehmann K., Zeller A.: Automated debugging in eclipse. OOPSLA Companion 2005: 184-185
2. De Paula, F.M., Gort, M., Hu, A.J., Wilton, S.J.E, Yang, J.: "BackSpace: Formal Analysis for Post-Silicon Debug," Formal Methods in Computer-Aided Design (FMCAD), IEEE eXpress Publishing, 2008, pp. 35-44.
3. Kuan, J., Wilton, S.J.E, Aamodt, T.M.: Accelerating Trace Computation in Post-Silicon Debug, 11th IEEE International Symposium on Quality Electronic Design (ISQED 2010), pp. 244-249.
4. Cunha, J. C., Lourenço, J., Duarte, V.: Debugging of parallel and distributed programs. In Parallel program development for cluster computing. Advances In Computation: Theory And Practice, Vol. Volume 5. Nova Science Publishers, Inc., Commack, NY, USA 97-129 (2001)