Quick Error Detection

Arun Debray, mentored by David Lin and Sundaram Ananthanarayanan
CURIS 2013

Background

- Objective: detect logic errors in silicon chips by inserting error checks into programs. Quick Error Detection (QED) aims to be both fast and accurate.
- LLVM provides tools for transforming and optimizing source code during compilation, serving as a framework for QED.
- Program flow is modeled as a graph of basic blocks corresponding to branch instructions.

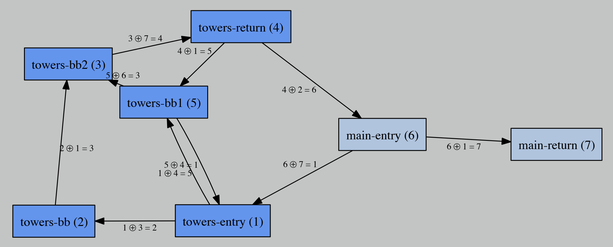

Figure 1: The control-flow graph for a Towers of Hanoi program.

- EDDI detects errors by duplicating instructions and comparing their values.

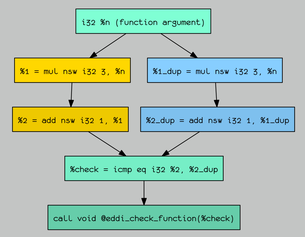

Figure 2: Duplicated instructions compared for a simple EDDI error check.

- CFCSS detects branching errors by checking the runtime predecessor of each basic block against the correct options.

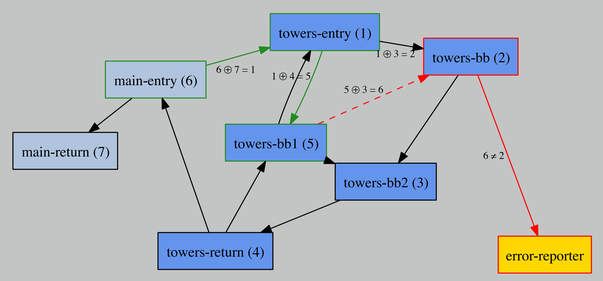

Figure 3: A depiction of a fault in the program from Figure 1. Here, CFCSS catches the incorrect branch from Block 5 to Block 2.

Global CFCSS

- By default, LLVM doesn't treat function calls as ending a basic block, but CFCSS does.
- Goal: add CFCSS checks at function calls and function returns.
- Calls through function pointers didn't interact well with global CFCSS, but many can be removed with the `instcombine` optimization pass.
- Results: small increase in coverage, but with a decrease in speed due to the additional check instructions.
  - Most faults do not manifest themselves as global-scale branches, both when tested on an FPGA and when simulated by modifying executables.
  - However, adding these checks eventually led to a simpler and faster implementation of the CFCSS algorithm, since it didn't have to work around function calls.

Optimization

- Inserting these checks for reliability clearly will cause programs to run significantly more slowly.
- Goal: how might it be possible to make QED-enhanced programs run more quickly without reducing coverage of errors?
- LLVM has a variety of compiler optimizations available, but some of them interfere with QED.
  - This interference can be avoided by running the optimizations in a separate `opt` pass before adding QED.
  - Some of the most useful `opt` passes were its collections of loop optimizations, link-time optimizations, etc.
- Improvements to details of the implementation also contributed to optimization:
  - φ nodes, used by LLVM to implement standard compiler optimizations, were repurposed to reduce the number of instructions needed to implement CFCSS.
  - Initially, the QED checks were called as separate functions, but they can be inlined, which also caused a significant speedup.
- Some of these optimizations have implications for coverage, such as reorganizing the duplicate instructions created by EDDI to occur farther from the original instructions, minimizing the likelihood of an error affecting both instructions and thus avoiding capture.
- Other optimizations reduce coverage. If speed is important, these can still be worthwhile.
- Results: significant speedup, even doing better than `-O3`. Coverage can generally be preserved by scheduling the QED pass after all other optimizations.

Optimization, continued

- Optimizations were compared by running `bzip2` compression. Each set of optimizations was tested against a non-QED executable with the same optimizations to determine whether the optimization helped QED or just the source program.

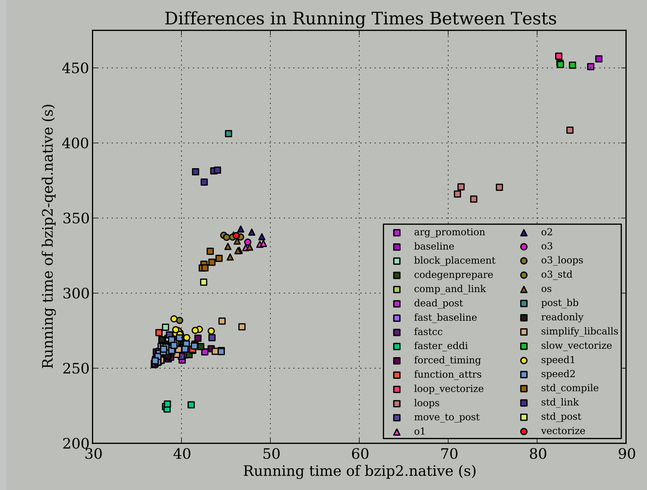

Figure 4: Timings of a `bzip2` program against different optimization collections.

Conclusion and Further Work

- Significant improvements were made in optimizing QED, though at some level it's a tradeoff between accuracy and speed.
- There's still room for improving the coverage of QED.
- QED can be made more flexible: some applications will require greater accuracy, and others more speed. This could be done, for example, by placing checks more or less densely in a program.

References

[1] Hong, T. et. al., *QED: Quick Error Detection Tests for Effective Post-Silicon Validation*, in Proc. IEEE Int. Test Conf., Nov. 2010, pp. 1-10.

[2] Lattner, Chris and Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. Proceedings of the 2004 International Symposium on Code Generation and Optimization, March 2004.

[3] Oh, Namsuk, Philip P. Shirvani, and Edward J. McCluskey. *Control Flow Checking by Software Signatures*. Center for Reliable Computing, Stanford University. April 2000.

[4] Oh, Namsuk, Philip P. Shirvani, and Edward J. McCluskey, *Error Detection by Duplicated Instructions In Super-scalar Processors*. IEEE Trans. on Reliability, Mar. 2002, pp. 63-75.

debray@cs.stanford.edu