ADDITIONAL INFORMATION ON IMPROVING QUICK ERROR DETECTION

ARUN DEBRAY
AUGUST 28, 2013

Contents

Abstract. This document is a set of notes to go along with a poster summarizing a summer of work on the QED project. The poster is located at `http://www.stanford.edu/~adebray/CURIS/adebray_qed_poster.pdf`, and these annotations (the document you are reading) can be found at `http://www.stanford.edu/~adebray/CURIS/annotations.pdf`. The goal is to provide more comprehensive information to explain methodology, motivations, background, etc. in ways that may not fit on or be ideal for the poster. As such, they are more complete than the poster, but may not make as much sense without it.

1. Background

Post-Silicon Validation and an Introduction to Quick Error Detection. The objective of Quick Error Detection (QED) is to test reliability in the context of possibly faulting hardware. It was developed as a way to more accurately report logic errors in silicon chip testing, known as post-silicon validation, but can be used for other, similar purposes as well. Specifically, QED sets as its goal to report errors extremely soon after they happen; rather than running a test program or test suite and comparing the output to what should have been calculated, for example, QED modifies a program so that error checking is incorporated into the code itself.. This makes error reporting much faster, which allows for more efficient testing in general. These methods can be implemented entirely with software, making them useful on a variety of different testing platforms, unlike more specialized solutions. For a more detailed overview of QED, see [1].

LLVM. Here is an example of the LLVM intermediate language:

```
; This program does exactly what you would expect it to.

@hello = private constant [14 x i8] c"Hello, world!\00"

declare i32 @puts(i8* nocapture)

define i32 @main() {
entry:
    call i32 @puts(i8* getelementptr inbounds ([14 x i8]* @hello, i64 0, i64 0))
    ret i32 0
}
```

The LLVM intermediate representation (henceforth LLVM IR) is an assembly language, but one with several high-level constructs, such as strong, static typing and the ability to declare attributes of functions or variables corresponding to optimization (e.g. declaring a function inline or cold). Additionally, there is an API for manipulating this IR, which makes it particularly effective for implementing optimization passes for a compiler. A complete description of the language is given in [3].

The example given above illustrates a few aspects of the IR already: integer types are denoted `i`n to get n bits (so that the string is really just an array of `char`s, and so on), and the use of C library functions such as `puts`.

LLVM is designed to be a set of modular components, and compiling a program with Quick Error Detection uses the following tools:

- A C or C++ program is translated into the LLVM IR. This can be done with the `clang`[1] and `llvm-gcc` compilers, both of which accept an `-emit-llvm` option when creating assembly code. It is perfectly possible to compile other languages to LLVM assembly, as some compilers for languages such as Haskell use LLVM backends, but we didn't do this.

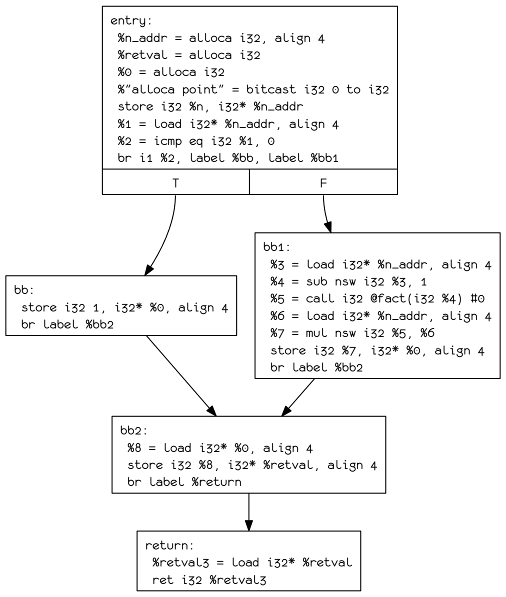
[1]See `http://clang.llvm.org`.

- The assembly code is optimized. An LLVM utility called `opt` does the majority of the optimization along compilation, and the list and order of optimization passes to run on the program can be specified on its command line. Quick Error Detection was implemented as an `opt` pass, which allowed it to be easily inserted into arbitrary programs as part of this compilation process.
- The modified LLVM assembly is converted into native assembly by the LLVM IR compiler, `llc`, and then can be made into machine code using any assembler, such as `gcc`.

This framework allows for many different program analyses to be implemented as passes through `opt`; the control-flow diagrams seen here and on the poster were generated with such a pass, and a different pass was used to inject faults into programs for testing CFCSS.

Some Definitions.

- A *basic block* is a sequence of instructions that is always executed in order, with no jumps or breaks. Thus, it consists of a bunch of "local" instructions followed by a branch, jump, goto, etc. Function calls are usually considered to terminate a basic block, but the LLVM IR doesn't follow this convention.
- The *control-flow graph* of a program is a directed graph in which the nodes are basic blocks and an edge $v_1 \to v_2$ indicates that it is possible to branch directly from the end of v_1 to the beginning of v_2. Thus, paths on the graph correspond to ways that the program can execute at runtime. An example is given in Figure 1.



```
entry:
 %n_addr = alloca i32, align 4
 %retval = alloca i32
 %0 = alloca i32
 %"alloca point" = bitcast i32 0 to i32
 store i32 %n, i32* %n_addr
 %1 = load i32* %n_addr, align 4
 %2 = icmp eq i32 %1, 0
 br i1 %2, label %bb, label %bb1
```

```
bb:
 store i32 1, i32* %0, align 4
 br label %bb2
```

```
bb1:
 %3 = load i32* %n_addr, align 4
 %4 = sub nsw i32 %3, 1
 %5 = call i32 @fact(i32 %4) #0
 %6 = load i32* %n_addr, align 4
 %7 = mul nsw i32 %5, %6
 store i32 %7, i32* %0, align 4
 br label %bb2
```

```
bb2:
 %8 = load i32* %0, align 4
 store i32 %8, i32* %retval, align 4
 br label %return
```

```
return:
 %retval3 = load i32* %retval
 ret i32 %retval3
```

Figure 1. A simple control-flow graph for a function, written in LLVM IR.

- A *flip-flop* is a type of hardware error in which a bit is incorrectly flipped at some logic gate. These errors can be modeled on an FPGA, or can be simulated by modifying the code section of an executable.
- The symbol $\oplus$ is used to represent bitwise `xor`.

EDDI. EDDI, or error detection by duplicated instructions, is a form of error detection presented in [5] in which two distinct flows of instructions are executed in or nearly in parallel, and the results are intermittently tested for equality. Though the key idea is simple, implementing it requires some thought about how to handle pointers, function calls, global variables, and so on. Additionally, significant speedup can be gained by scheduling instructions to take advantage of processor scheduling.

CFCSS. CFCSS (control-flow checking by software signatures), presented in [4], assigns each basic block a unique ID number, and assigns a global variable to keep track of the ID of the block executing at a given moment. At the beginning of each basic block, it checks whether the global ID is that of one of its valid predecessors; if not, then an illegal branch was taken and the program signals an error. If the branch was correct, the global ID is updated and execution continues. More formally, the exact algorithm is as follows:

(1) Every basic block v in the program is assigned a unique integer ID $I(v)$.[2]

[2] In practice, not all basic blocks are reachable in all programs, and those that are unreachable aren't assigned an ID. These include the blocks of dead functions, return blocks of functions that call `exit()`, etc.

(2) A global variable G is created to track the ID of the basic block currently being executed at any point during runtime, and a runtime signature D is created to handle fan-in nodes (see below).
(3) Let $P(v)$ be the set of predecessors to v; that is, the set of basic blocks that call or branch to v. Then, a signature difference d is created to allow for efficient checking. How it is created depends on the size of $P(v)$:
  (a) If $P(v)$ is empty, no checks need to be inserted.
  (b) If $P(v) = \{p\}$, so that v has the sole predecessor p, then set $D = d = I(p) \oplus I(v)$.
  (c) Otherwise, v is a fan-in node, in that it has multiple predecessors $p_1, \dots, p_n$. In this case, pick an arbitrary node p_1 and set $d = I(p_1) \oplus I(v)$. Then, in node p_i, for $1 \leq i \leq n$, insert an instruction updating $D = I(p_1) \oplus I(p_i)$.
(4) For each predecessor $p \in P(v)$, update G as $G = G_{\text{old}} \oplus D$ before the branch.
(5) Add an instruction $G = G_{\text{old}} \oplus D \oplus d$ to the start of v.
(6) Now, in an error-free program, $G = I(v)$, so raise an error if this is untrue.

This setup ensures that when all of the `xors` cancel out, the signature check detects most illegal branches. In denser control-flow graphs, where multiple nodes have the nearly but not quite identical sets of predecessors, an illegal branch can go undetected, but this setup catches most such faults.

In practice, φ instructions can be used for fan-in nodes. These instructions initialize a variable differently depending on which basic block was the predecessor at runtime. φ instructions are standard tools in compiler optimization, and are supported in the LLVM IR. Thus, the correct values of G and D can be computed in the above algorithm by adding a φ instruction to update them correctly.

2. Global CFCSS

Overview. LLVM doesn't treat function calls as basic block boundaries. This is reasonable for managing some aspects of runtime, such as activation records, and it makes the most sense given the structure of its assembly (with function calls abstracted away from just branches), but for implementing CFCSS, it is not the most natural approach. CFCSS treats a program as a graph given by branch instructions, designed for implementation on a lower level where functions don't really exist. Thus, within the LLVM framework, it was necessary to add global checks separately. Checks were inserted across function calls and function returns, and though the code used to write the pass had special cases, the checks are no different in the assembly code. The resulting CFCSS checks are slightly different than what one would imagine if function calls ended basic blocks; here, when a function returns, it returns to the basic block that called it. This is not believed to have a significant effect on coverage, because the probability of a fault that caused a branch that was silent to one of these methods but not the other is small.

One interesting stumbling block was the presence of indirect function calls, which are calls made through a function pointer. Since the value of the pointer can change at runtime, the LLVM API cannot provide any information about the called function while QED is being inserted. However, this information is required by global CFCSS, since it tries to insert function call checks that require knowing both the caller and the callee. Later, I realized that a preexisting LLVM optimization pass would eliminate many such indirect calls, which was sufficient to solve our issues (see below). This pass is invoked by calling `opt -instcombine` before the QED pass, and in a separate invocation of `opt`; calling with both passes on the same run doesn't guarantee that they will be run in order. If for some reason this isn't viable, or it leaves some indirect calls intact, then it would be necessary to adjust CFCSS such that every basic block containing a function pointer is a possible predecessor of *every* function, which is less ideal but not incorrect. However, we didn't do this, because `-instcombine` eliminated this from our testing programs.

Function pointers are rare enough in programming that this didn't seem like it would be a significant issue at first, but for some reason the LLVM tools automatically converted some direct function calls into indirect calls. I could find no pattern indicating which function calls would be converted, though it was consistent across compilations and common enough to interfere with useful testing programs. However, while trying to optimize Quick Error Detection, I discovered that `-instcombine` eliminated these inserted indirect calls, which was sufficient for our purposes.

Results. Global CFCSS led to a minor increase in coverage, at a cost of performance. This makes sense; most faults won't manifest as function calls or global branches. However, those that did caused the minor coverage increase. The performance cost was due to the additional checks, especially at the start and end of commonly called functions.

However, these global checks meant that the CFCSS algorithm was implemented completely and correctly. Though it required special-casing in the code for the LLVM pass, it ended up producing more unified assembly code. This allowed optimizations to be applied more simply, since they didn't have to work around the special-casing. For example, the φ-node optimization described below is most effective when all branches, including function calls, are tracked by CFCSS.

3. Optimization

Overview. Adding all of these checks is nice for reliability, and greatly increases fault detection, but at the cost of running time. This is understood as an inevitable aspect of reliability testing, but since it was possible to make QED-enhanced programs run faster, we might as well, as it will simplify large-scale testing. A number of techniques were used to make these programs run faster:

- One minor source of slowdown was that QED programs kept track of a CFTSS array, which tracks the last n basic blocks to have been executed, where n is set by the user, with a default of 10. Then, it dumps this information when a QED check fails. Though invaluable to debugging, this was less necessary for coverage testing, where it was more of a concern that something failed than understanding when it failed. Thus, it was possible to turn this option off by a command-line flag, and when $n = 1$ there is a bit less data to carry around, allowing other simplifications.
- The bulk of the optimizations came from passes specified by `opt`. These passes cover a variety of analyses and program transformations, and are included by default in the LLVM toolchain. These are discussed in greater detail below. The trick to this sort of optimization is to ensure that coverage isn't compromised, which can be guaranteed by running `opt` multiple times to force passes to be run in a certain order.
- The use of φ nodes was helpful in simplifying global CFCSS checks. Before they were introduced, each basic block had to check a large number of possible previous IDs, introducing some overhead. For a basic block v, if $|P(v)| = n$, then this created n comparison instructions of the runtime ID against the ID of each predecessor, along with an additional $n - 1$ instructions to coalesce these checks into a single boolean value. However, φ nodes allow this to be simplified, since the check that needs to be made can be chosen by the path the program takes.
- Finally, the specific implementation of error reporting is worth mentioning. This isn't explicitly touched on in the papers introducing various types of QED, and in our implementation the checks were in a separate function, as follows:

```
define i32 @fft_compute(i32 %argc, i8** %argv) {
entry:
  %0 = load i32* @LAST_GLOBAL_CFTSS_ID
  %1 = icmp eq i32 %0, 239
  call void @cfcss_check_function(i1 %1)
; ... and then the rest of the basic block

define void @cfcss_check_function(i1) {
entry:
  br i1 %0, label %return_block, label %exit_block

return_block:       ; preds = %exit_block, %entry
  ret void

exit_block:         ; preds = %entry
  %1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([15 x i8]*
      @cfcss_error_msg, i32 0, i32 0)) #0
  call void @error_detected()
  br label %return_block
}
```

  This was useful for debugging, but slowed down performance considerably. However, the LLVM IR has a function attribute `inlinehint`, which marks functions as ideal for inlining.[3] Then, one must run the `-inline` optimization pass for this to actually do anything, but it causes a significant speedup in the program, as there are fewer function calls. It is possible to explicitly inline these checks while generating them (i.e. modifying the QED LLVM pass to not generate the check functions), but calling the inlining pass is easier and just as efficient.

Testing. The tests were conducted in the following manner:

(1) The QED executable was made with the command
`opt $(PRE_PASSES) -load Hello.dylib $(OPT) -hello -QED-Mode=15 $(POST_PASSES)`. Here, `PRE_PASSES` was a Makefile variable that contained the list of passes to be executed before QED, and `POST_PASSES` was a variable for the passes conducted after QED. Finally, `OPT` held the optimization level for the test. For each trial, the passes and optimization levels are given. Note that `opt` does not always run its passes in order; thus, I eventually switched to executing the pre-QED passes, the QED pass, and the post-QED pass over three separate calls to `opt`. The trials described below used this convention.
(2) At the same time, a non-QED executable was made with the command `opt $(PRE_PASSES) $(POST_PASSES)`. This executable was used to compare the times given by the QED and non-QED incarnations of `bzip2`.
(3) The programs were tested on the input of `input.graphic`, a 6.7 MB TIFF picture that can be seen in Figure 2. There are wildly differing numbers of tests, because I wanted extra data about some of the programs. To make the tests most fair, I used my computer lightly while they were run, and in particular never ran two tests simultaneously.

[3]There is also an `alwaysinline` attribute, with the corresponding `-always-inline` optimization pass, but these were effectively the same as the `inlinehint` attribute for testing QED.

Figure 2. The input file used for benchmarking different optimization passes.

The passes specified below are referenced at `http://llvm.org/docs/Passes.html`. Rather than list all of the data here, I have posted it to `http://www.stanford.edu/~adebray/CURIS/optimization_data.pdf`. However, a few trials merit mention:

- The first trial, `baseline`, served as the control; no passes were run on it except for QED. This run took over 7 minutes to conclude.
- When the optimization level was set to `-O3`, there was a considerable speedup to about 5.5 minutes.
- The `speed2` trial was run at `-O0` and with several collections of passes: `-std-compile-opts`, `-std-link-opts`, and several loop optimizations, followed by the QED pass. These optimizations sped up the test to take about 4 minutes 20 seconds, and changing the optimization level actually made it run more slowly. Various modifications to this trial didn't make much of a difference, and the recommended way to run these programs includes all of these passes.
- Other methods were eventually able to push the time down to about 3 minutes, though these were changes to QED mentioned above, rather than optimizations applied by `opt`.

These optimizations must be applied with care in that if run after QED, they can interfere in its coverage; for example, EDDI creates duplicated instructions that don't do anything unless faults are introduced; thus, an optimization pass that eliminates redundant instructions would reduce coverage if introduced after QED. `opt` does not force an order from the passes given to it on the command line, so in order to make sure QED is run last, it is necessary to call `opt` multiple times.

In addition to optimization-focused testing, different tests had to be introduced to determine the coverage of QED. There are many ways of doing this:

- An FPGA (field-programmable gate array) is the most direct way to simulate logic errors on a silicon chip, since it acts as an effective chip simulator and can be modified to insert faults. This is used for large-scale testing, though I didn't do this.
- One way to do software-based testing is to inject errors using `gdb`. Its control over the runtime allows one to stop the program at a random location and modify memory, simulating a flip-flop error. Additionally, a Python interface makes scripting relatively clean.
- CFCSS was tested with branch fault injection, the method used in [4]. This entails adding, removing, or changing branches in the assembly code, and leaving the rest of the code unchanged. Thanks to the already existing LLVM framework, I was able to implement this as an LLVM pass that introduced one error into the control-flow graph, and feed it into the normal compilation process. I found this very useful for calculating CFCSS coverage.

- In [5], EDDI is tested by modifying the code section of an executable. Specifically, a bit is randomly chosen and flipped. This was easy to implement as a standalone script, and can even be done in a relatively platform-independent way by reading the file into an array, modifying a random bit, and then writing the modified array back to the file. To determine whether the code section was in fact modified, one can disassemble the executable and check if the code section looks any different. This was very useful for both EDDI and CFCSS, as such modifications could trigger either error-handling mechanism.

  Most programs produced in this way fall into one of two categories. Some simply don't run, and the OS rejects them. However, a significant majority simply show no error whatsoever; their output is completely unchanged from the correct output. This makes testing more difficult, because both of these are not used in the final testing data, but take up much of the running time of the trials. It is possible that future work could improve this, but then it might lose platform-independence, or be less useful since a similar percentage of flip-flop errors on the FPGA also had no noticeable effects.

Results. The graph of results is given on the poster, and here it can be explained in more detail. Each test was run several times, as described above, and the running time of the program without QED checks was plotted against the running time of the program with QED checks. Trials near the bottom of the chart are favored, because they correspond to the best speedups. Many such speedups will also increase the speed of the non-QED executable, so the best trials ended up near the bottom left of the graph.

4. Conclusion and Further Work

- I was pleasantly surprised to discover how possible it was to improve the speed and coverage of QED; I had initially seen a nearly complete implementation and wondered how I would be able to approach it.
- LLVM is a very powerful tool for transforming programs, and has lots of possible uses outside of its initial conception as a tool for writing optimizing compilers. I expect that if I want to do program analysis, profiling, or optimization for some other project, I will probably work with LLVM.
- Despite the gains in coverage and optimization mentioned above, there is still more to do. For example, many errors cause the program to suddenly terminate, and some of these cannot be caught by existing methods. Is there a way to detect whether the program exited correctly, or to catch the mistakes that led to an unexpected exit?
- Lost coverage could also be due to faults affecting the error reporting; CFCSS and EDDI currently report failure via `printf`, and faults may affect the runtime in a way that prevents this from working correctly. Other solutions might improve coverage without actually affecting the algorithm; for example, one could protect system calls or library functions from the faults (which is a bit harder on the FPGA, but is reasonably possible with software testing). One might also be able to isolate error reporting on a different thread or process that is known to be free of faults. Finally, it may be possible to raise and catch signals, which could be less fragile.
- The LLVM IR can be used as an intermediate representation for many languages, while we only dealt with C and sometimes C++. Extending QED to other languages would give more freedom to error testing and possibly provide more meaningful results if an environment is known to mostly run a given programming language.
- Similarly, QED doesn't work on all kinds of C programs. Most of this is due to EDDI not copying library calls that are hard to duplicate effectively, such as getting the system time (which can change slightly between instructions that EDDI duplicates). Figuring out what exactly should be duplicated would also provide more meaningful data on a wider variety of programs.
- EDDI and CFCSS aren't the only ways to implement QED, just the ones written for LLVM. Adding others would be a useful way forward.

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] *LLVM Language Reference Manual*, `http://llvm.org/docs/LangRef.html`. August 26, 2013.

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

[5] 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.