Motivation

- Developers of systems software have “rules” to check for correctness or performance. (Do X, don’t do X, do X before Y…)
- Code that does not obey these “rules” will run slow, crash the system, launch the missiles...
- Consequently, we need a systematic way of finding as many of these bugs as we can, preferably for as little cost as possible.

What Will we be Talking About?

- What’s the problem?
- What’s the solution?
- Discuss some of the interesting details
  - Assertion Checking
  - Global Rule Enforcement
  - FLASH Optimizations
- Evaluation and conclusions
- Some related work/history of the paper

What’s the Problem?

- Current solutions all have trade-offs.
- Formal Specifications-rigorous, mathematical approach
  - Finds obscure bugs, but is hard to do, expensive, and don’t always mirror the actual written code.
- Testing-systematic approach to test the actual code
  - Will detect bugs, but testing a large system could require exponential/combinatorial number of test cases. It also doesn’t isolate where the bug is, just that a bug exists.
- Manual Inspection-peer review of the code
  - Peer has knowledge of whole system and semantics, but doesn’t have the diligence of a computer.

What’s the Solution?

- Meta-level compilation (MC) combines the domain knowledge of developers with analysis capabilities of a compiler.
- Allows programmers to write short, simple, system-specific checkers that take into account unique semantics of a system.
- Checkers are then added to a compiler to check during compile-time.
What’s the Solution?

- The author’s MC system uses a high-level, state-machine language called Metal.
- Metal extensions written by programmers are linked to a compiler (xg++) that analyzes the code as it is being compiled.
  - Intra and Interprocedural analysis.

How does it work?

- The language is a high-level, state-machine language.
- Two parts of the language—pattern part and state-transition part.
  - Pattern language—finds "interesting" parts of code based on the extension the programmer writes.
  - State-transition—Based on the discovered pattern, current state, either move to a new state or raise an error.
- Tests are written and then added to the xg++ compiler. Xg++ includes a base library that includes some common, useful functions and types.

How does it work?

- Compiler generates the AST for the program that is being compiled.
- Metal extensions are compiled into a set of transitions.
- The xg++ compiler traverses the AST for the program in execution order in a depth-first manner, following the transition patterns, as they apply.

AST for an Assert Statement

- assert_exos_self_insert_t_pete(0, PG_PIPG_UIPG_W, PGROUNDOWN(va), 0, NULL) == 0;
Global Rule Checking

- Many rules apply globally across function call chains.
- Example: Rules that are expressed in terms of blocking functions, such as certain types of deadlock.
- xg++ provides mechanisms for gathering “global” data and then applying it to a xg++ extension.

Global Rule Checking—Checking for Deadlock

- “Kernel code cannot call blocking functions with interrupts disabled or while holding a spin lock. Violating this rule can lead to deadlock.”
- We need to include a rule that will handle this rule.
  - Unfortunately, when executing a rule like this, we need to know what function calls can result in a call to a blocking function.
- Solution: Use Global Rule Checking

Global Rule Checking—Checking for Deadlock

- Compiler’s 2 passes generate a call graph.
  - First pass uses a Metal extension to find those functions that potentially block, tags those functions in the resulting call graph.
  - Second pass links all files sent to xg++ into a large call graph, does a depth-first traversal to find all functions that have a path to a blocking function. Generates a listing of these functions.
- Now, we can execute a localized rule within the context of these blocking functions.

New, Improved Global Rule Checking

- Global Rule Checking was formalized in later version of Metal.
- Two Passes
  - First Pass: Each file being compiled has a temporary AST generated for it.
  - Second Pass: Reads temporary files to reconstruct the ASTs for entire program, control-flow graph is generated to trace the execution through multiple files. (Functions that aren’t called are the roots of the trees.)
- Metal extensions are then run on the AST in depth-first manner based on the control-flow graph that is generated.

FLASH Optimizations

- Not only can you detect software bugs, it should be obvious that any types of rules can be enforced using this code, including performance-enhancing rules.
- Example: FLASH Hardware/Software
  - Code for FLASH must be fast because it implements functionality usually in hardware.
  - Been aggressively optimized for many years, but MC still is able to provide hundreds of optimizations, because it’s hard to manually traverse deeply nested control paths.
FLASH Optimizations

- Buffer-free optimizations
  - Traces send calls. Detects if a buffer is needed and if the send frees the buffer.
- Redundant length assignments
  - It can be difficult down deep nesting paths to remember if a length field for a buffer has already been set.
  - Metal allows for such a scan.
- Efficient opcode setting
  - Scan to see if the message header has a known opcode already there. If so, recommend XOR'ing with the desired opcode. (Reduces assembly instructions to 1.)

Evaluation

- Anecdotal evidence throughout the paper demonstrating that MC discovers a large number of bugs.
  - Ran tests on FLASH's cache coherence code, as well as versions of Linux.
  - In both cases, the rule extensions that were run found bugs that could have potentially crashed the system.
  - In one case, there was a bug that was detected that would have required the tester to look through 300 lines of code, 20 if-statements, 4 else clauses, and 29 conditional compilations.
- The large number of bugs is magnified by the fact that the rules for finding the bugs were written in few lines of code (<100, in most cases.)

Evaluation continued

- No formal experiment done to demonstrate that their system was better than other established systems.
- For the performance evidence, there was no discussion of how much of a performance improvement there would have been if the compiler’s recommendations were actually executed.

Conclusions—The Good

- Best of all worlds (testing, formal specs, manual inspection)
- Very simple to write “rules”.
- Discovers a large number of bugs that could potentially crash the system, even with simple rules.
- Problems are identified before code is even executed.
- Flexible solution that allows for varied checks to security, stability, and even performance.

Conclusions—The Bad

- Situations occurred when there were false positives.
  - Many of which were a result of not completely fleshing out the rules, in particular for more complex scenarios.
  - Currently coming up with ways of easily writing code to eliminate these false positives. (Heuristic algorithms to determine which bugs are the most important bugs, for example.)
- No discussion of the backgrounds of people who wrote the rules.
  - How much domain knowledge did the rule authors have?
  - How much programming ability did the rule authors have?
  - More explanation of the experimental setup would have been nice.

Evaluation continued

- After paper was published, more data was gathered on bug discovery using Metal on Linux kernel.

Available at: http://metacomp.stanford.edu/linux/list.php3
Related Work/History

- Won Best Paper at OSDI 2000
- Based on previous work called Magik.
  - Much more difficult to write extensions.
- Several other papers written on topic.
- Ideas are now marketed as a company founded by Engler called Coverity.

References

- Hallem, Seth et al. A System and Language for Building System-Specific, Static Analyses, PLDI 2002

Questions?