Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions

Dawson Engler
Benjamin Chelf
Andy Chou
Seth Hallem
Stanford University

Matthew Thornton

November 9, 2005

1

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.

2

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

3

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.

4

What’s the Problem?

- None of the current methods seem to give us what we’re looking for.
- Can the compiler check the code?
  - It would be nice to put the code in the compiler and have it check all of the “rules.”
  - Unfortunately, those “rules” are based on semantics of the system that the compiler doesn’t understand. (Lock and Unlock are valid to the compiler, but how and when they should be used isn’t.)
- Need some technique that merges the domain knowledge of the developer with the analysis of a compiler.

5

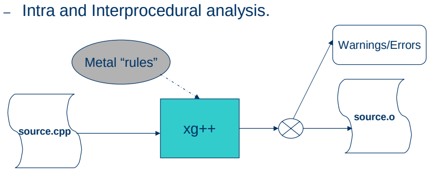
What’s the Solution?

- 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.

6

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.



7

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.

8

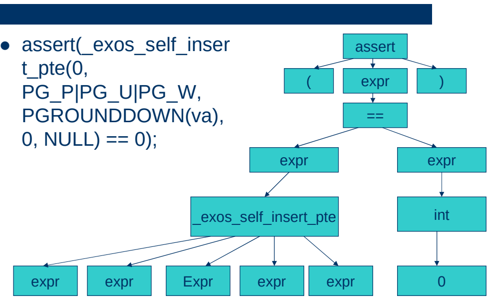
How does it work?

- 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.

9

AST for an Assert Statement

- assert(_exos_self_insert_pte(0, PG_P|PG_U|PG_W, PGROUNDDOWN(va), 0, NULL) == 0);



10

Assertion Checking

- Assertions should check a value, a state, etc.
- Assertions should not change any values or the state of the program. (ie, you should be able to take them out of the program without any change.)
- Example that would crash if it was removed:

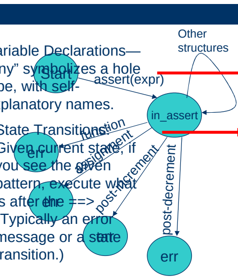
  assert(_exos_self_insert_pte(0, PG_P|PG_U|PG_W, PGROUNDDOWN(va), 0, NULL) == 0);

11

Assertion Checking

Variable Declarations—"any" symbolizes a hole type, with self-explanatory names.

State Transitions: Given current state, if you see the given pattern, execute what is after the ==> (Typically an error message or a state transition.)




```
sm Assert flow_insensitive {

decl { any } expr, x, y, z;
decl { any_call } any_fcall;
decl { any_args } args;

start: { assert(expr); } ==>
{ mgk_expr_recurse(expr, in_assert); } ;
in_assert:
{ any_fcall(args) } ==>{ err("function
    call"); }
| { x = y } ==> { err("assignment"); }
| { z++ } ==> { err("post-increment"); }
| { z-- } ==> { err("post-decrement"); } ;
}
```

12

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.

13

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

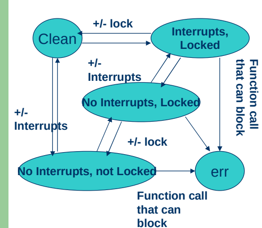
14

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.

15

Global Rule Checking—Checking for Deadlock



- With the list of blocking functions available, a second extension is run through the program code.
- Rules include detecting when spin locks are enabled/disabled or when interrupts are enabled/disabled.
- When in the state where locks are enabled or interrupts are disabled, a blocking function cannot be called because it can cause deadlock in the Linux implementation.

16

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.

17

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.

18

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.)

19

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.)

20

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.

21

Evaluation continued

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

	2.4.1	2.4.0	2.3.0	2.2.0	2.1.0	1.3.0	1.2.0	1.0	Total
BLOCK	206	206	28	27	5				472
FLOAT	10	10	2						22
FREE	17	17	5	5	1				45
INTERNAL_NULL	69	69	19	17	4	1	1	1	181
INTR	27	27	6	6	2				68
LOCK	26	26	2	2					56
NULL	124	122	17	18	2			1	284
PARAM	7	10	1	1					19
RANGE	54	47	3	3					107
REALLOC	10	9							19
SIZE-CHECK	3	3							6
VAR	33	30	8	8	3	1	1		84
Total	586	576	91	87	17	2	2	2	

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

22

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.

23

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.

24

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.

25

Related Work/History

- Application-specific information in compilers—Eraser.
- Formal verification, strong type checkers.
- Extensible compilers
  - Ctool—Traverse the AST, look for domain-specific issues.
  - Meta-object protocols—Extensions written into compiler.
  - Aspect-Oriented Programming—Weave checks into existing code.

26

References

- Engler, D. et al. *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI 2000.
- Engler, D. *Incorporating Application Semantics and Control into Compilation*. IEEE Transactions on Software Engineering, May/June, 1999. Vol 25, Number 3, 387-400.
- Hallem, Seth et al. *A System and Language for Building System-Specific, Static Analyses,* PLDI 2002

27

Questions?

?

28