MC: Meta-level Compilation

Extending the Process of Code Compilation with Application-Specific Information – for the layman developer (code monkey)

Gaurav S. Kc
8th October, 2003

Outline

- Dawson Engler
- Overview of the Compilation Process
- Meta-level Compilation
  - early days with MAGIK
  - current incarnation: MC
  - good for detecting bugs:
    - NULL pointer misuse
    - memory leak (failure to deallocate memory)
    - memory corruption (illegal use of deallocated memory)
    - security holes (buffer overflow, formatstring vulnerabilities)
- Conclusions

Dawson Engler

- The man behind MAGIK and MC
- PhD from MIT '98
- Stanford Faculty (Metalevel Compilation Group)
  - http://metacomp.stanford.edu
    "The goal of the Meta-level Compilation (MC) project is to allow system implementors to easily build simple domain- and application-specific compiler extensions to check, optimize, and transform code."
  - Publications on MC at OSDI, PLDI, SOSP, Oakland Symposium, ACM CCS
- Coverity.com: commercialised MC

Compilers

- S/W lifecycle phases
  - Requirements engineering
  - Design, and implementation
  - Repeat, and maintain
- Compilation phases
  - Pre-process (**`cpp`**) : macro processing
  - Compiler proper (**`cc1`**)
    - front end synthesis: source → IR, symbol table, control-flow, data-flow
    - middle end optimisation: IR →IR
    - back end generation: IR → optimised machine assembly
  - Assembler (**`as`**): assembler macro processing, translate ASCII instructions into binary machine code
  - Linkage editor (**`ld`**): combine several object modules (and library files) to produce static or dynamically-linked executables

Meta-level Compilation

- Static information generated by the front-end synthesis phase is lost after compilation
- Application-specific compiler extensions & optimisations can benefit from this information
  - Compiler developer cannot anticipate all possible domain-specific extensions
  - Application writer doesn't want to learn compiler internals
- Need: Simpler mechanism for coding application-specific extensions for integration into compiler

MC Paper:

Incorporating Application Semantics and Control into Compilation

Dawson R. Engler, First Conference on Domain Specific Languages, 1997

- Programmers can be active users of compilers
- Incorporate domain-specific extensions into the compilation process
- Facilitate previously impossible "application-level" optimisations and semantic-checking (dereference **`NULL`**)
- Leave application source code unmodified
  - Source-level (IR) modifications for portable user extensions
  - Full compiler optimisations on modified IR
- Leave compiler source code unmodified
  - Extensions will be exhibit "built-in" behaviour in compiler

magik: An ANSI-C api to LCC IR

- Dynamically linked into modified LCC compiler
- User extensions:
  - Code: invoked at every function definition
  - Data: invoked at every `struct` definition
- Examples:
  - Automatically replace a poly-typed function (*output*) with *printf* and appropriate format-string

```
output("i = ", i, ", j = ", j)
printf("%s%d%s%d", "i = ", i, ", j = ", j)
```

  - Mandatory checking of return codes for system calls

```
read(fd, buffer, size)
if (0 > read(fd, buffer, size))
  error("failed system call <read>\n")
```

magik: illustration

Replace poly-typed function with *printf* equivalent

```
foreach function-call ( "output" )
  foreach function-argument ( = arg )
    switch argument-type ( arg ) {

      case Integer:
       strcat ( typestring, "%d" ); break;

      case Pointer:
        if rawPointerType ( arg ) == CHAR
          strcat ( typestring, "%s" );
        else strcat ( typestring, "0x%p" );
        break;
    }

  replace-call ( function-call, "printf" )
  insert-argument (function-call, typestring )
```

MC Paper:

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

Operating Systems Design and Implementation, 2000

- System rules for Operating System Kernel
  - Kernel sanitises user-space data before accessing it (**do X before Y**)
  - A lock must have a corresponding unlock on every code path (**when X, do Y**)
- Peer reviews for manual inspection of source code: not rigorous, human error.
- Automated enforcing of system rules
  - Testing: time-consuming, not exhaustive since complexity/size scale with system size. Impractical to test all device drivers for Linux
  - Formal Verification: model checkers, theorem provers/checkers to validate consistency of abstract specification of system. Hard to accurately represent system in specifications: over-simplification, omission of features, unless generating code from specs
- Compiler-based static analysis tools are useful
  - No scalability problem. Works directly on source code
  - System rules have straightforward mapping to source code
  - Rules are enforced as new phases in the compilation

metal: A high-level, state machine language

- *yacc*-like specification for SM: matched patterns in source code causes transitions between different states
- Linkable object code compiled from *metal* specifications using *mcc*.
- Dynamically linked into compiler, `xg++` (based on GNU `g++`, working on `gcc` version)
- SM is applied down all possible control-flow paths for each function

MC / *metal*: illustration

Ensure corresponding *sti* (re-enable interrupts) for every *cli*

```
sm check_interrupts {

  // Patterns
  pattern enable  = { sti(); };
  pattern disable = { cli(); };

  // States and transitions / actions
  is_enabled : disable ==> is_disabled
             |  enable ==> { error( "double enable" ); }
             ;

  is_disabled:  enable ==> is_enabled
             | disable ==> { error( "double disable" ); }
             | $end-of-path$ ==> { error( "exit w/ intr" ) }
             ;
}
```

Other MC / *metal* checks

- Make the kernel check user-space pointers before de-referencing (applicable to library interfaces)
- For states {*unknown, null, not_null, freed*}, find when pointers are used:
    - before being checked
    - on **`NULL`** paths
    - after being **`free`**'d
- Find double-free errors
- Find error paths (returning a negative value) that don't free allocated memory
- Cannot handle multi-threaded applications

Conclusions

- Meta-level compilation
- New phases for user-extensible compiler
- Domain-specific checks for
  - locating application bugs
  - enforcing system rules
  - …
- Compiler experience ~~required~~