# EEL: Machine-Independent Executable Editing

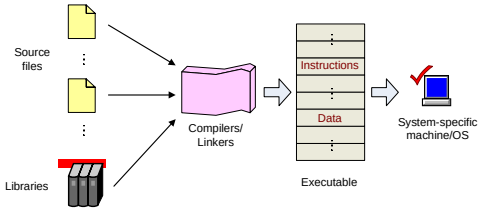
James R. Larus and Eric Schnarr

Presented by C. Shen
CMSC714, Fall 2005

## Outline

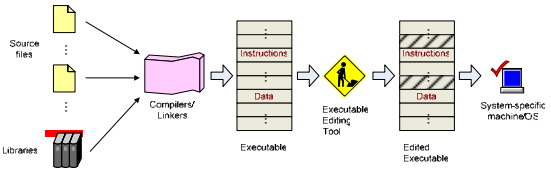
- Introduction
- EEL Abstractions
  - Executables
  - Routines
  - CFG: Control-Flow Graph
  - Instructions
  - Code Snippets
- System-Dependent EEL
- EEL Status
- Conclusions

## Introduction
*Executables*




- Program executables are atomic entities.
- How to observe, measure, or modify a program's behavior?

## Introduction
*Executable Editing*




- Remove existing instructions and add foreign code in executable.
- Executable editing is widely used for emulation, observation, and optimization.

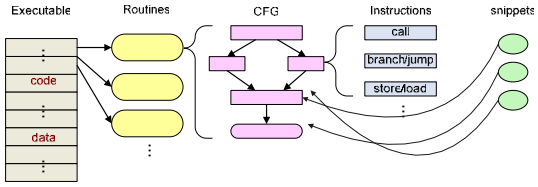
## Introduction
*EEL*

- EEL: Executable Editing Library
- A C++ library for building tools to analyze and modify an executable program
- EEL can edit fully-linked executables.
- EEL emphasizes portability across systems.
- Mostly machine-independent interface
  - machine-independent abstractions
- Applications: *qpt* (A Quick Program Profiling and Tracing System)

## Outline

- Introduction
- EEL Abstractions
  - Executables
  - Routines
  - CFG: Control-Flow Graph
  - Instructions
  - Code Snippets
- System-Dependent EEL
- EEL Status
- Conclusions

## EEL Abstractions



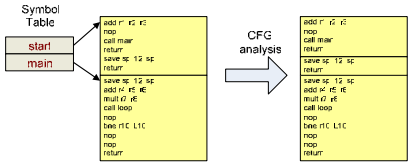

- Five major machine-independent abstractions that allow a tool to examine and modify an executable.
- Internal representation: register-transfer level (RTL) instruction description

## EEL Abstractions
*Executables*

- EEL executable objects are an abstraction of executable files
  - Object, library, or static and dynamically-linked programs
- EEL refines symbol information
  - Data tables, hidden routines, and multiple entry points
- EEL maintains symbol table information for the edited program
  - Debugging information for the edited executable

## EEL Abstractions
*Routines*

- Routines are named objects in a program's text segment that contain instruction and data.
  - Hold information about an entity in the text segments
  - Provide interfaces to EEL's control-/data- flow analysis
- Control-flow analysis may split routine

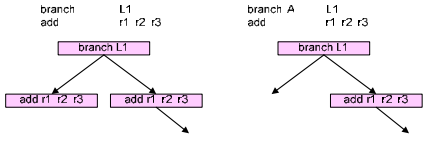



## EEL Abstractions
*CFG*

- CFG: Control-Flow Graph
  - Directed graph
  - Nodes: basic blocks
  - Edges: control flow between blocks
- The primary program representation in EEL
  - Represent a routine as a CFG
- Why CFG?
  - Implement profiling and tracing on CFG edges
  - Adjust addresses in branch and jump instructions
  - Provide an architecture-independent way of representing control flow

## EEL Abstractions
*CFG (cont'd)*

- Architecture-independent control-flow representation
  - Basis for program analysis
  - EEL uses internally
  - Normalization




## EEL Abstractions
*CFG (cont'd)*

- Tools edit CFG
  - Delete instruction
  - Add new code before/after instruction or along edge
  - Accumulate edits without changing the CFG (batch style editing)
- After editing CFG
  - Produce a new version of the routine
    - Incorporate the changes
    - Involve laying out blocks and snippets
    - Update control-transfers instructions (calls, branches, jumps)

## EEL Abstractions
*Instructions*

- RISC-like machine instructions
  - Memory references (loads and stores)
  - Control transfers (calls, returns, system calls, jumps, and branches)
  - Computations
  - Invalid (data)
- C++ classes
  - Combine for more complex instructions
  - E.g. autoincrement load = a memory reference + a computation

## EEL Abstractions
*Instructions (cont'd)*

```
// Compute a backward address slice with
respect
// to register R, from PC.

bool instruction::backward_slice(bb* b,
                                 addr pc,
                                 int_reg r)
{
  if (is_easy() || is_hard())
    // Already in earlier slice
    return (true);
  else if (writes()->is_member(r))
    // Modifies register R
    {
      if (!fp_reads()->is_empty())
        // Do not trace floating point ops
        mark_as_impossible(b, pc);
      else if (reads()->is_empty())
        // Easy instruction reads nothing
        mark_as_easy(b, pc);
      else
        {
          // Hard instruction reads registers.
          mark_as_hard(b, pc);
          int_reg read_reg;
          // Continue slicing them
          FOREACH_REG (read_reg, reads())
          {
            b->backward_slice(pc, read_reg);
          }
        }
      return (true);
    }
  return (false);
}
```

- Inquiries about an instruction's effect on a program's state
- Inquiries independent of an underlying machine
  - Code is similar to the original algorithm

## EEL Abstractions
*Snippets*

```
1* sethi %hi(0x1), %g6! upper bits of &counter
2* ld [%lo(0x1) + %g6], %g7! load counter
   add %g7, 1, %g7! increment
3* st %g7, [%lo(0x1) + %g6]! store counter
```

```
code_snippet*
routine::incr_counter_code(long counter_num)
{
  assert(0 <= counter_num);

  tagged_code_snippet* snippet
    = new incr_count_snippet();
  addr counter_addr = PROFILE_COUNTER_START
        + counter_num * sizeof(counter);

  SET_SETHI_HI(*snippet->find_inst(1),
               counter_addr);
  SET_SETHI_LOW(*snippet->find_inst(2),
                counter_addr);
  SET_SETHI_LOW(*snippet->find_inst(3),
                counter_addr);

  return (snippet);
}
```

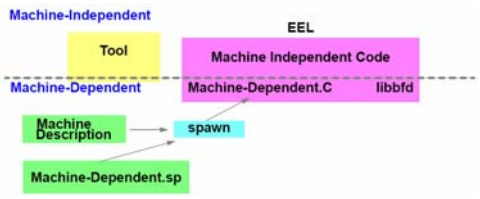
- Foreign code added to an executable
  - EEL allocates registers from unused (dead) or freed registers at insertion point.
  - Code in snippet is not machine-independent.

## Outline

- Introduction
- EEL Abstractions
  - Executables
  - Routines
  - CFG: Control-Flow Graph
  - Instructions
  - Code Snippets
- System-Dependent EEL
- EEL Status
- Conclusions

## System-Dependent EEL

- Instrumentation code is machine-specific




## System-Dependent EEL
*spawn*

- A EEL tool for binary instruction analysis and manipulation
- Customize annotated C++ code from high-level machine description

```
instruction*
mach_inst_make_instruction(executable* exec,
                           mach_inst* inst,
                           addr pc)
{
 {{INST inst AT pc CATEGORY
  CALL DIRECT:: return new call_instruction(inst);;
  JUMP DIRECT:: return new jump_instruction(inst);;
  BRANCH DIRECT:: return new branch_instruction(inst);;
  JUMP:: {
   if (mach_inst_do_op(inst, OP_ICALL))
    return new indirect_call_instruction(inst);
   if (mach_inst_do_op(inst, OP_RET))
    return new return_instruction(inst);
   if !({{IS LITERAL}} && {{READ 1}} == 0)
    return new jump_instruction(inst);
   return new indirect_jump_instruction(inst);
```

## System-Dependent EEL
*spawn (cont'd)*

- Example: portion of spawn's SPARC description

```
// Instruction field definitions:
//
instruction{32} fields
 op 30:31, op2 22:24, op3 19:24, opc 5:13,
 rd 25:29, rs1 14:18, rs2 0:4, iflag 13:13,
 simm13 0:12, imm22 0:21, disp22 0:21,
 disp30 0:29, cond 25:28, aflag 29:29,
 asi 5:12

// Control-transfer instruction syntax:
//
pat
[ bn    be    ble   bl    bleu  bcs   bneg  bvs
  ba    bne   bg    bge   bgu   bcc   bpos  bvc
  fbn   fbne  fblg  fbul  fbl   fbug  fbg   fbu
  fba   fbe   fbue  fbge  fbuge fble  fbule fbo
  cbn   cb123 cb12  cb13  cb1   cb23  cb2   cb3
  cba   cb0   cb03  cb02  cb023 cb01  cb013 cb012]
is op0 && op2=[0b010 0b110 0b111]
     && cond=[0..15]
```

## System-Dependent EEL
*spawn (cont'd)*

- Added description of instruction semantics

```
// General purpose register set
//
register integer{32} R[35]
alias integer{32} PSR is R[32]

// Control-transfer instruction semantics:
//
val disp is (integer{32})disp30
val branch is
     \r.\op.(t:=pc+disp; op r ? pc:=t : aflag=1 ? annul)

sem [bne be bg ble bge bl bgu bleu bcc bcs bpos bneg bvc bvs]
is branch PSR
     @ ['ne 'e 'g 'le 'ge 'l 'gu'leu 'cc 'cs 'pos 'neg 'vc 'vs]
```

## Outline

- Introduction
- EEL Abstractions
  - Executables
  - Routines
  - CFG: Control-Flow Graph
  - Instructions
  - Code Snippets
- System-Dependent EEL
- EEL Status
- Conclusions

## EEL Status

- EEL runs on SPARC processors under SunOS & Solaris
- Spawn not yet distributed
- QPT2: a EEL-based profiler
- Other applications:
  - Active Memory (a memory system simulation platform)
  - Elsie (a direct-execution architectural simulator)
  - Wisconsin Wind Tunnel architectural simulator
  - Blizzard-S's fine-grain access control

## Conclusions

- Tools to modify executables have proven their value in many areas
  - Monitor program behavior and performance
  - Architectural experiments
- EEL is a highly portable library for editing executable programs
  - Provides mostly architecture- and system-independent set of operations
  - Provides machine-independent CFG and program analysis
  - Simplify the analysis and manipulation of most programs