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

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

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

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
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

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

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)

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

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

bool instruction::backward_alias(int b, int pc, int_reg r)
{
  if (is_easy()) return true;
  // Already in earlier alias
  else if (writes() || is_number(r))
    // Modify register R
    
    if ((fp_reads() || is_empty()))
      // Do not trace floating point ops
      return true;
    else if (reads() || is_empty())
      // Easy instruction reads nothing
      return true;
    else
      
      // Hard instruction reads registers.
      int reg read_reg;
      // Continue aliasing these
      FOR EACH UNO (read_reg, reads())
        b = backward_alias(pc, read_reg);
      } return (true);
  return (false);
EEL Abstractions

Snippets

1. \texttt{maddi 0x1, $gp6; upper bits of counter}
2. \texttt{ld ([lo $(ax)] + $gp6), $gp7! load counter}
   \texttt{add $gp7, 1, $gp7! increment}
3. \texttt{st $gp7, ([lo $(ax)] + $gp6)! store counter}

- 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.

```
code_snippet
routine: incr_counter_code(long counter_num) {
    assert (0 == counter_num);
    tagged_code_snippet* snippet
        = new incr_count_snippet();
    addr counter_addr = PROFILING_COUNTER_START
        + counter_num * sizeof(counter);
    SHT_SETNI_HI(*snippet->find_inst(1),
        counter_addr);
    SHT_SETNI_LOW(*snippet->find_inst(2),
        counter_addr);
    SHT_SETNI_LOW(*snippet->find_inst(3),
        counter_addr);

    return (snippet);
}
```
System-Dependent EEL

- Instrumentation code is machine-specific

```
isntruction*  
  mach_inst_make_instruction(executable* exec,  
  mach_inst* instr,  
  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(Instr, OP_ICALL))  
  return new indirect_call_instruction(inst);  
  if (mach_inst do_op(Instr, OP_RET))  
  return new return_instruction(inst);  
  if (isa Literal) & & (READ 1) -- 0  
  return new indirect_jump_instruction(inst);  
  return new indirect_jump_instruction(inst);  ```
Example: portion of spawn’s SPARC description

```
// Instruction field definitions:
//
// instruction[32] fields
// op 30:31, cp2 22:24, cp3 19:24, cpc 5:13,
// rd 25:29, rs1 14:18, rs2 6:4, iflag 13:13,
// imm13 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
// [ kn be ble bl bleu bcc bneg bvs
// ha hno bge bgu bcc bpos bvs
// fbn fbone fbnp fbul flbl flaug fbg fbu
// fba fbe fbeu fbe gflag fbde fbule fble fbo
// cbn cb123 cb12 cb12 cb1 cb2 cb3
// cbde cb0 cb02 cb03 cb01 cb013 cb012]
// is op0 &\& op2=[0b010 0b110 0b111]
// &\& cond=[0..15]
```

Added description of instruction semantics

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

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

sem [bne be bg ble bge b1 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