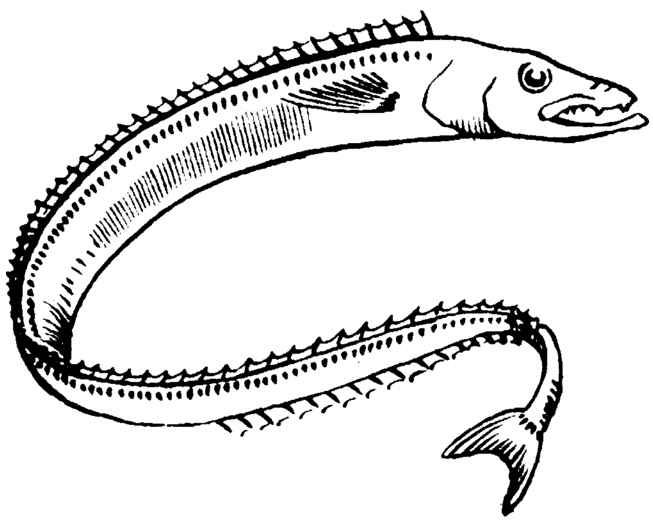
EEL: Machine-Independent Executable Editing*

James R. Larus

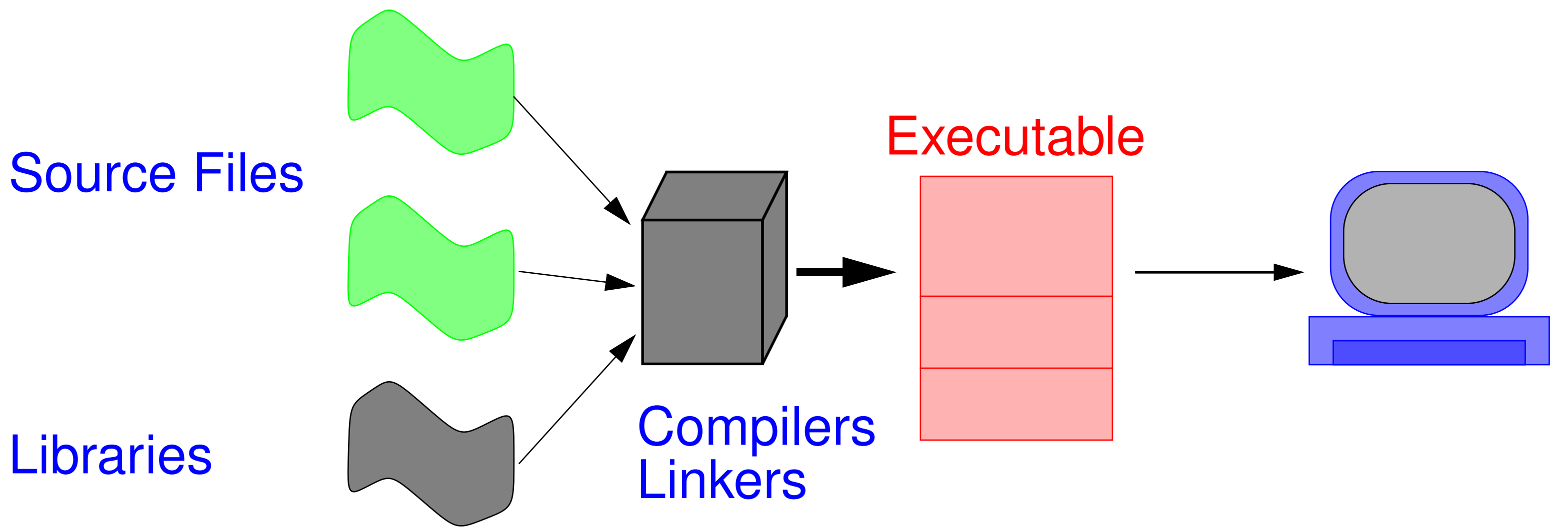
Wisconsin Wind Tunnel Project
University of Wisconsin
larus@cs.wisc.edu



*. Supported by ARPA (B550, Wright Labs #F33615-94-1-1525), NSF (MIP-9225097 & CCR-9101035), NSF PYI/NYI (MIP-8957278, CCR-9157366, & CCR-9357779), DOE (DE-FG02-93ER25176), UW Grad School, WARF, Bell Labs, DEC, Sun, TMC & Xerox

Status Quo

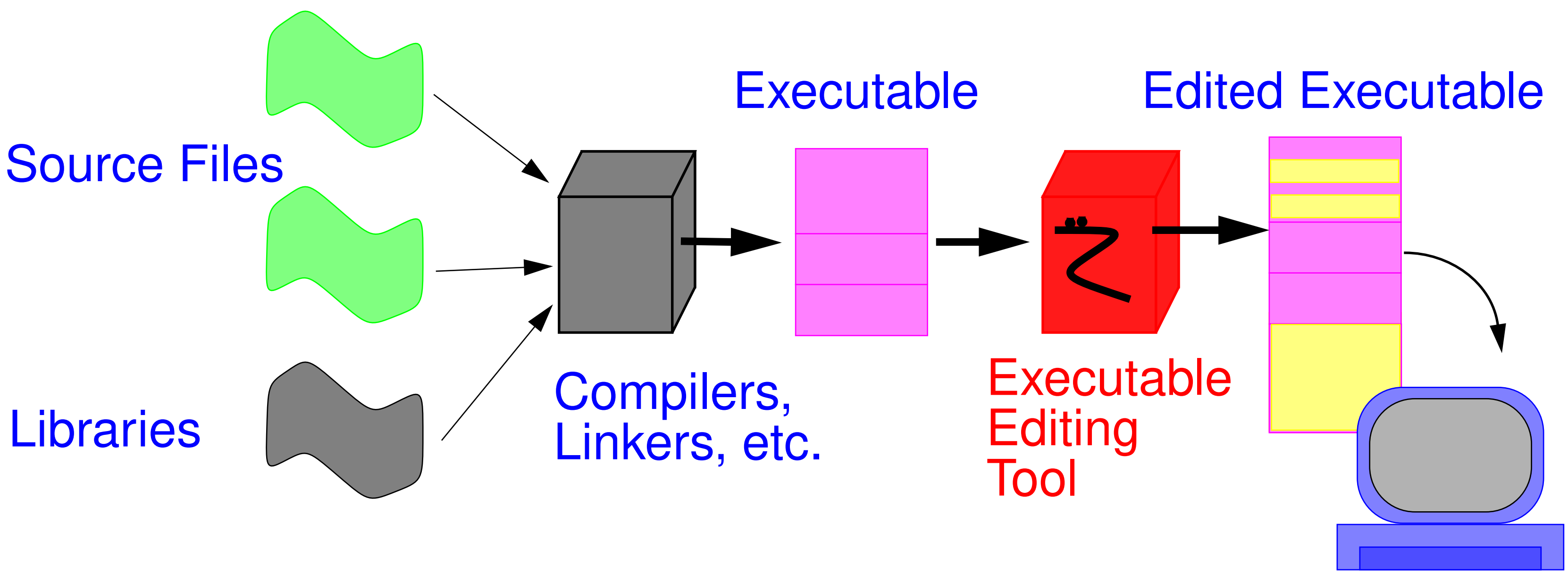
- Program executables are atomic entities



- What happens when program runs?
  - How to measure or monitor execution?
- How to modify program behavior?

Executable Editing

- Add foreign code or modify existing code in executable



+System tools unchanged

+Expose complete process (including libraries)

+Does not require source code

+Convenient to use

Use 1: Emulation

- Emulate features not provided by hardware
  - Commodity processors have no exotic features
  - Demonstrate ideas on stock hardware
- Prototype future systems
  - Wisconsin Wind Tunnel (WWT)
- Software fault isolation (sandboxing)
- Migrate code from legacy architectures
  - Binary translation (x86 $\rightarrow$ your favorite machine)
- Reschedule instructions for different implementations

Use 2: Observation

- Measure program performance
  - MIPS's *pixie* and Ball & Larus's *qpt* profiling/tracing systems
  - Lebeck & Wood's *Fast Cache* system
  - *PP* path profiler
- Detect programming errors
  - Pure Software's *purify*

Use 3: Optimization

- Program optimization
  - Wall's global register optimization
  - 64 bit address calculations
- Entire program analysis and optimization

Why Doesn't Everyone Do It?

- Complexity
  - Every instruction set and OS differs
  - Existing architectures, compilers, and executables hamper process
- Large amount of effort to build robust tool
  - Result: 1 tool on 1 platform
- Static program analysis alleviates problems

EEL

- C++ library for editing executables
    - Edits object files, archives (libraries), and linked executables
    - (Mostly) machine-independent interface
    - Solid analytic base for robust tools
    - EEL and tools portable across systems
- Goal: Machine-independent tools

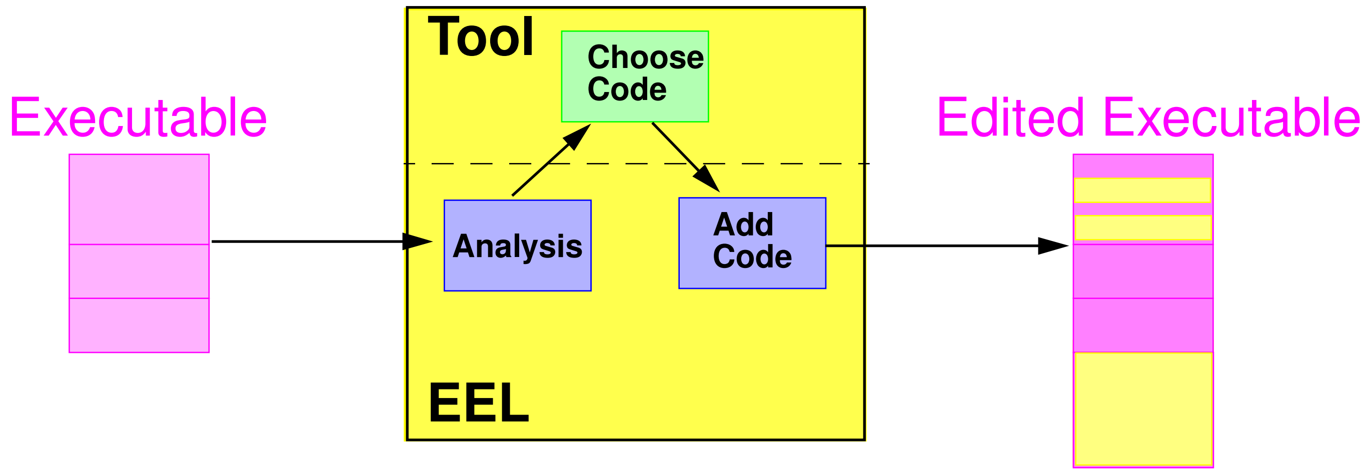
Related Work

- Srivastava and Wall's OM and ATOM systems
  - OM is "higher-level" interface to ATOM
  - Insert code into DEC Alpha object files
    - EEL operates on fully linked executables
    - EEL is designed to be portable across systems

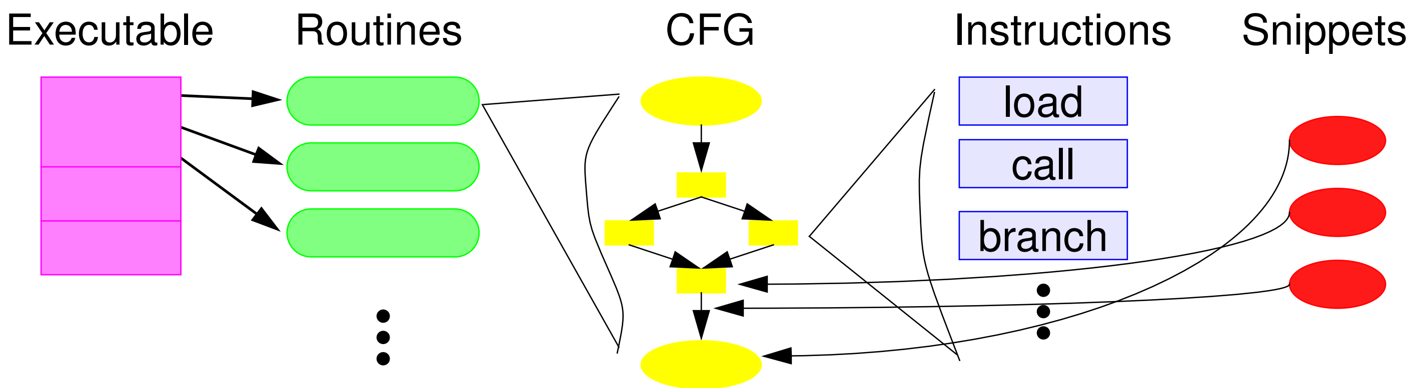
How to Edit an Executable

- Four step process:
  - Analyze program's control flow
  - Select instrumentation
  - Insert instrumentation
  - Produce modified executable

Application



EEL Abstractions



- Executable
    - Code and data from executable (object, library)
- Routine
    - Examine and modify in any order
- CFG
    - Edited by tool
- Instruction
    - One-for-one machine-independent description
- Snippet
    - Instrumentation (foreign) code

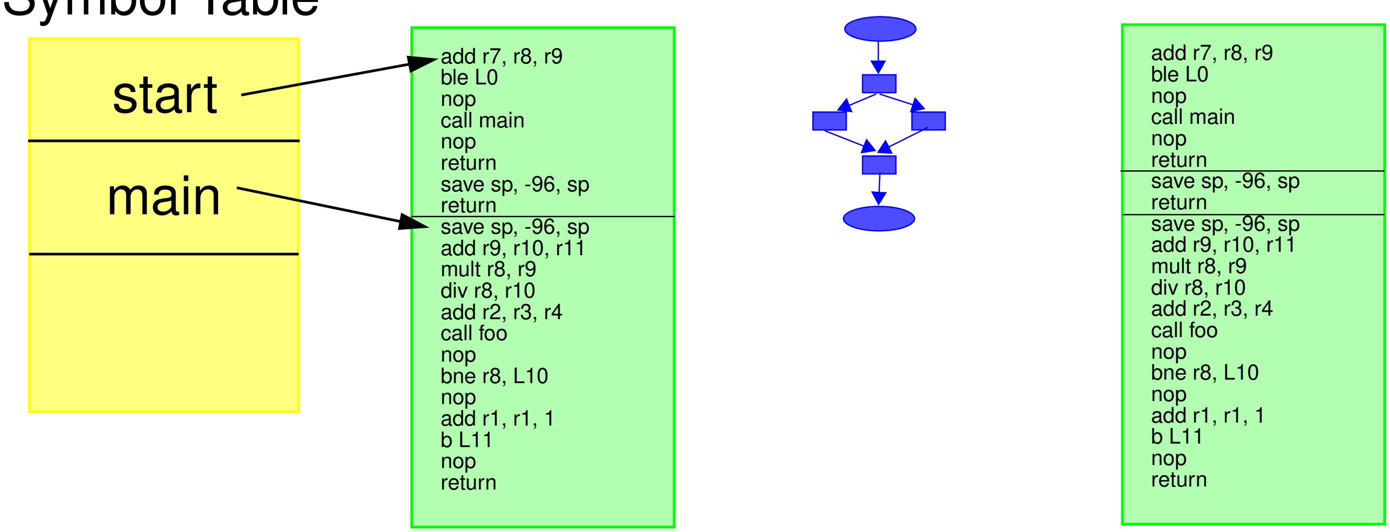
Executable

- File containing machine (binary) code
- EEL refines (incomplete) symbolic information
  - Unnamed procedures
  - Multiple entry points
  - Interprocedural jumps
- Can free global/local registers for application's use

Identify Routines

- Read symbol table. Discard duplicate and internal labels
- Partitions into initial set of routines
- Examine routines to find calls and interprocedural jumps
  - Record multiple entry points
- Control-flow analysis may split routine

Symbol Table

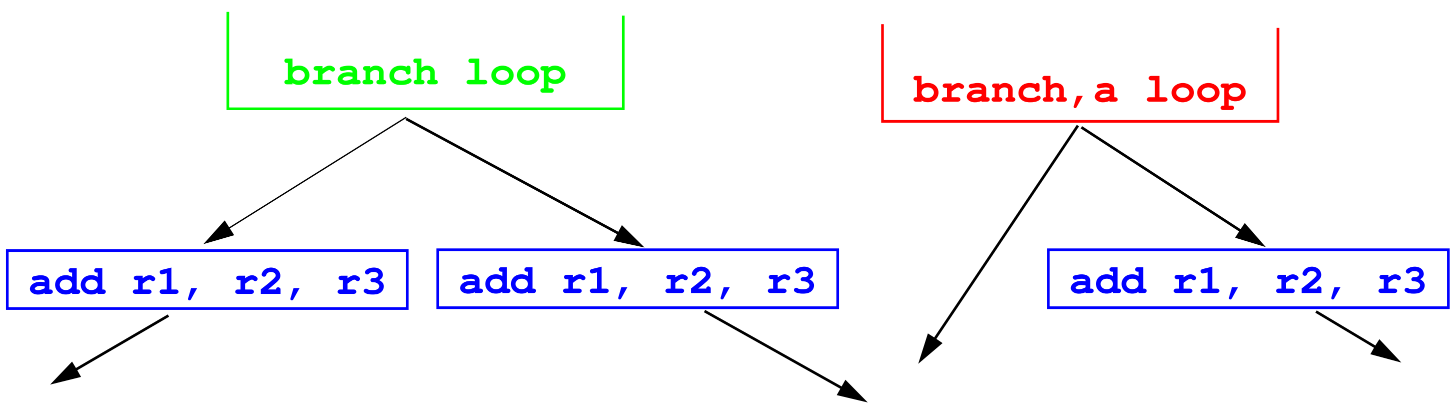


Control-Flow Graph (CFG)

- Architecture-independent control-flow representation
  - Basis for program analysis (and instrumentation)
  - EEL uses internally
- Normalized control flow
  - Instructions in block execute sequentially
  - No delayed transfers

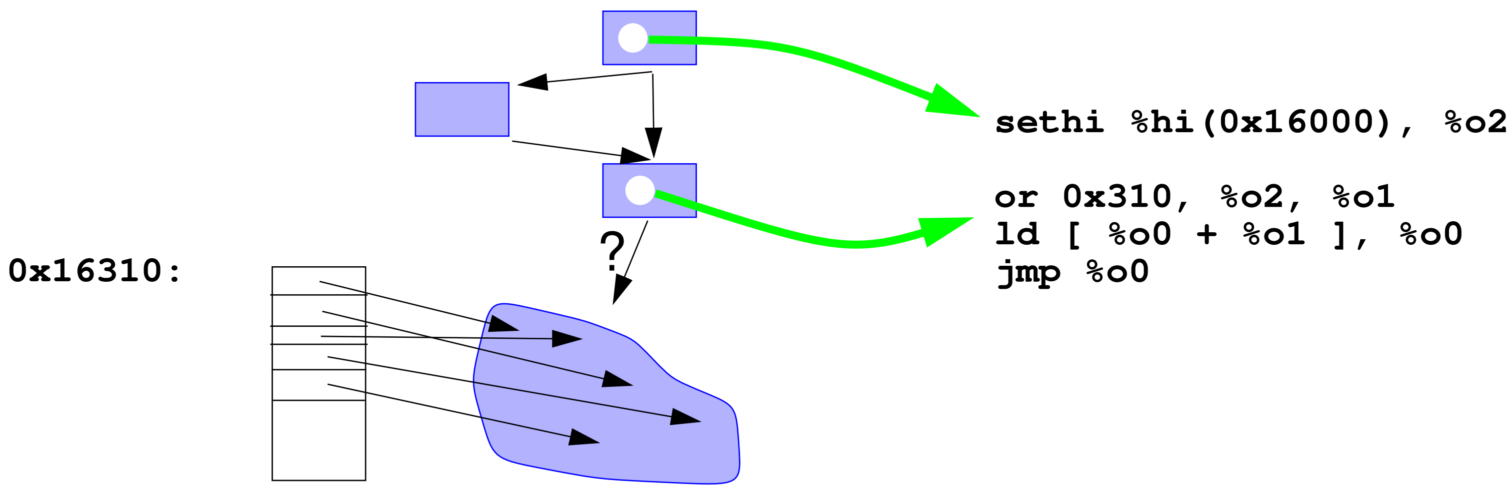
```
branch loop
add r1, r2, r3
```

```
branch,a loop
add r1, r2, r3
```



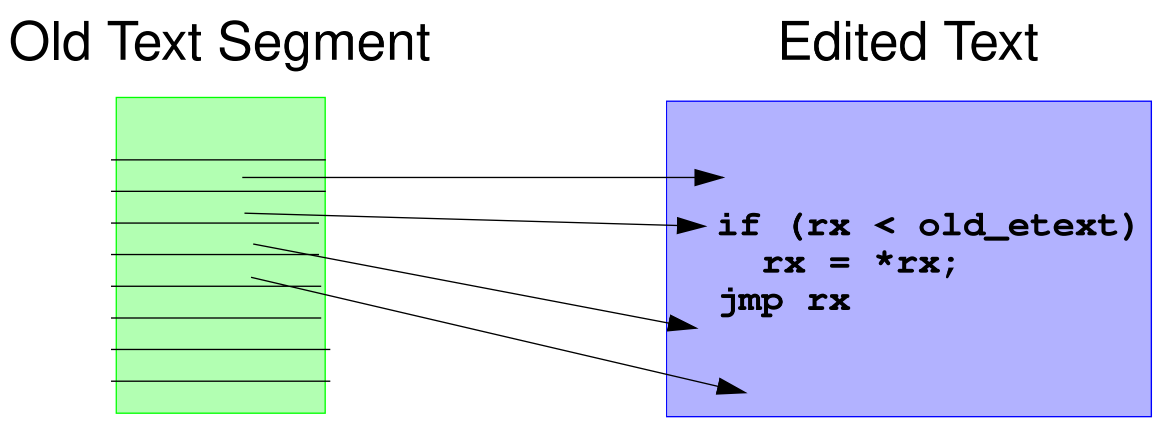
Control-Flow Analysis

- EEL provides standard analyses:
  - dominators, natural loops, live registers, and register slicing
- EEL uses analysis to refine control flow
- e.g., indirect jumps
  - High-level languages use indirect jumps in stylized manner
  - Find switch statement's dispatch table (address & size)
  - Backward slice of partial CFG finds table address



Unknown Control Transfers

- Some indirect jumps are unanalyzable
  - Dynamically computed address
  - Control-flow graph incomplete
- Edited program must execute correctly
- Map from old → new addresses
  - At run-time, convert jump address

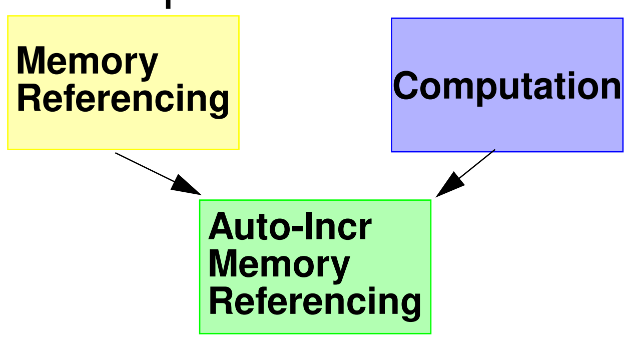


Editing CFG

- Tools edit CFG
  - Remove instruction
  - Add foreign code before/after instruction or along edge
  - “Batch” editing: CFG does not change
- Foreign code changes instructions’ positions
  - EEL updates control-transfers (calls, branches, and jumps)
- Data references unchanged
  - Foreign code may affect registers
  - Live-register analysis finds unused registers
- Update symbol table

Instructions

- RISC-like instructions
  - Memory references
  - Control transfers
  - Computation
  - Invalid (data)
- One-to-one representation of program’s instructions
  - Inquiries independent of particular architecture
- C++ classes
  - Combine for more complex instructions



  - VAX/x86 CISC instructions do not fit

Code Snippets

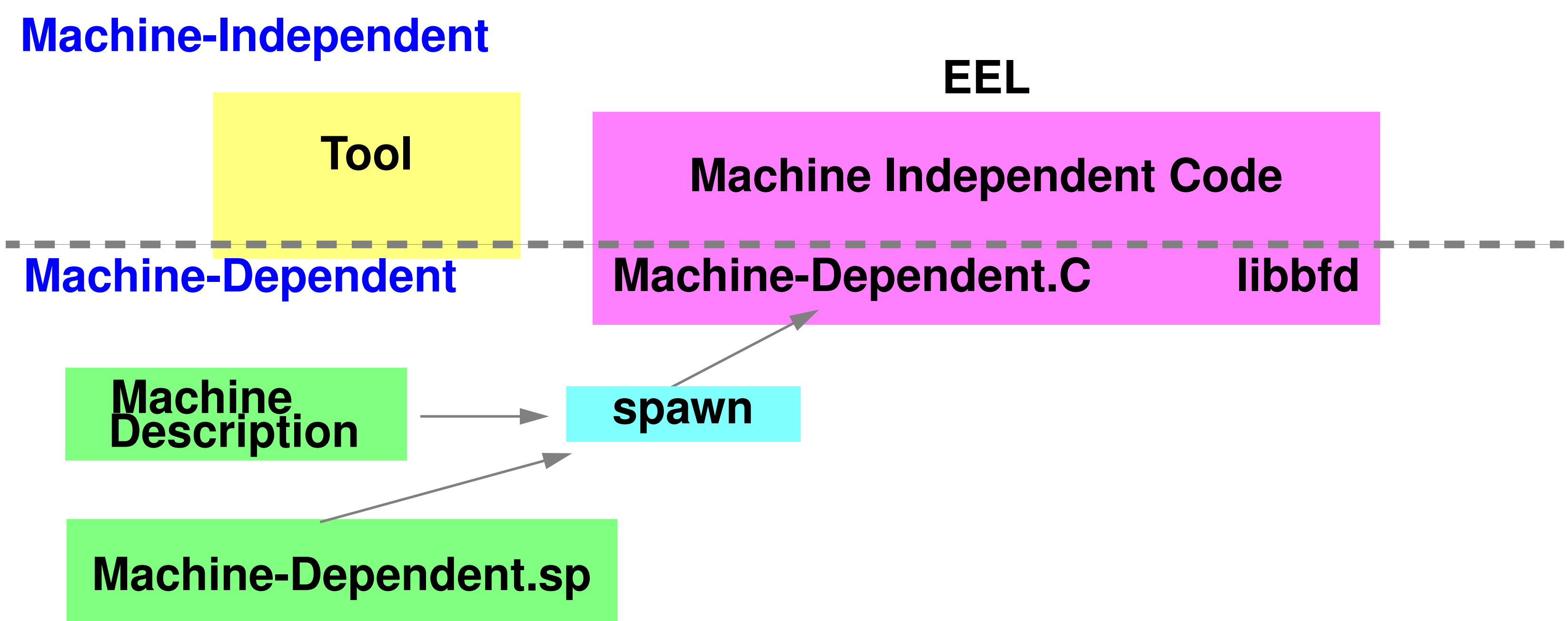
- Foreign code inserted into program
  - EEL allocates registers from dead or freed registers at insertion point
  - Code in snippet is not machine-independent!

```
1*      sethi 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){
   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);
}
```

EEL Portability

- Tools mostly machine-independent
  - Instrumentation code is machine-specific
- EEL analysis and modification is machine- and OS-independent



Spawn

- Binary instruction analysis and manipulation is error-prone
- EEL tool *spawn*
  - 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);
```

Instruction Syntax

- Instruction description derived from Ramsey' NJ Machine Toolkit

```
// 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]
```

Instruction Semantics

- Added description of instruction semantics

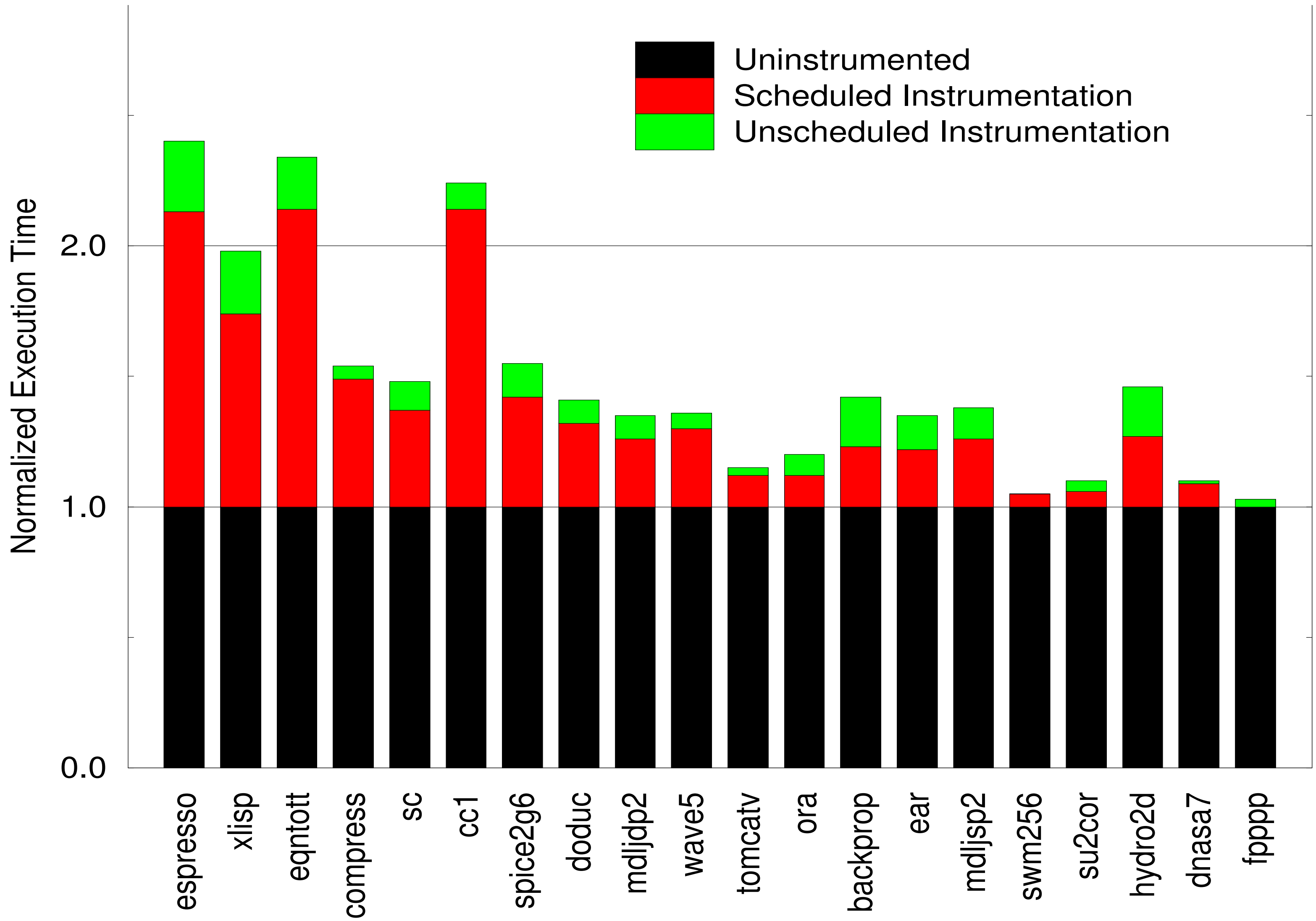
```
// General purpose register set
//
register integer{32} R[35]
alias integer{32} PSR is R[32]

// 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]
```

- SPARC description 145 lines, MIPS 128 lines, Alpha 138 lines
  - + 827 line (machine-independent) description = ~2,500 lines

Status

- EEL version 2.1 distributed
  - SPARC under SunOS & Solaris
  - Heavily used at Wisconsin & elsewhere
  - UC Berkeley has instrumented Solaris kernel
- QPT ported to EEL
  - 17,000 $\rightarrow$ 6,000 lines
  - EEL is 13,600 lines
- Port to IBM RS6000 working
- Spawn not yet distributed
  - SPARC description functions equivalently

Experimental Features: Code Scheduling



Conclusion

- Modifying executables is basis for building “insight” tools
  - Expose program behavior and performance
  - Architectural experiments
- Details are messy and machine-specific
  - Tool development is time-consuming and uneconomical
- EEL is a library for editing executables
  - Hides most machine and OS details
  - Provides machine-independent CFG and program analysis
  - “Right” abstractions for manipulating programs
- EEL permits construction of system-independent insight tools