EEL: Machine-Independent Executable Editing

James R. Larus and Eric Schnarr
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton St.
Madison, WI 53706 USA
{larus,schnarr}@cs.wisc.edu

Abstract

EEL (Executable Editing Library) is a library for building tools to analyze and modify an executable (compiled) program. The systems and languages communities have built many tools for error detection, fault isolation, architecture translation, performance measurement, simulation, and optimization using this approach of modifying executables. Currently, however, tools of this sort are difficult and time-consuming to write and are usually closely tied to a particular machine and operating system. EEL supports a machine- and system-independent editing model that enables tool builders to modify an executable without being aware of the details of the underlying architecture or operating system or being concerned with the consequences of deleting instructions or adding foreign code.

1 Introduction

A program executable holds instructions and data for a compiled program. In most situations, executables are atomic entities that are created, used (executed), and discarded. Sometimes, however, it is convenient or necessary to look inside one of these entities and observe, measure, or modify a program's behavior. *Executable editing* changes executable (compiled) code by removing existing instructions and adding *foreign code* that observes or modifies a program's execution. It is an effective technique for measuring and modifying program behavior since executables hold an entire program (including libraries)[1] and editing them does not require source code or modification to system tools such as compilers and linkers.

Executable editing is widely used for three purposes: emulation, observation, and optimization. An edited executable can emulate features that hardware does not provide. For example, the Wisconsin Wind Tunnel architecture simulator [19] drives a distributed, discrete-event simulation of a parallel computer from the logical cycle times of processors directly executing a parallel program. The underlying hardware (a SPARC processor in a Thinking Machines CM-5) does not provide a cycle counter or an efficient mechanism for interleaving computation and simulation. The Wind Tunnel system edits programs so that they update a cycle timer and return control at timer expirations. Similarly, one version of the Blizzard distributed shared-memory system [20] edits programs to insert fine-grain access tests before shared loads and stores. These tests permit data sharing at cache-block granularity, which reduces the false sharing incurred by page-granularity distributed shared-memory systems. Another emulation is software fault isolation (sandboxing) [27], which implements protection domains by modifying code to prevent it from referencing or transferring control out of its domain. In the limit, editing can replace an entire program with instructions for a different architecture. Translation is used both to migrate legacy code to new architectures (e.g., Tandem [2] and VAX [21]) and to run binaries on other systems [12].

Technology trends are increasing opportunities for editing executables. Machines, both sequential and parallel, are built almost exclusively from commodity microprocessors, which offer instructions and memory systems targeted at a

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, an NSF NYI Award CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, DOE Grant DE-FG02-93ER25176, and donations from Digital Equipment Corporation and Sun Microsystems. The U.S. Government is authorized to reproducé and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

1 This process can also be performed on components of an executable (object files). However, a patent obtained by Pure Software precludes many uses of object-file modification [17].

mass market that has no need for semantically-rich protection or memory models. Although its performance is lower than hardware's, executable editing enables research by allowing new ideas, such as sandboxing or user-level shared memory [19], to be demonstrated on existing processors and tested on real applications. Editing can also solve practical problems raised by new architectures. For example, good performance on highly-parallel superscalar or VLIW processors requires instruction scheduling tuned for a particular implementation. Rescheduling an executable (by editing) offers an attractive alternative to purchasing, distributing, managing, and updating binaries. Finally, editing offers a solution to the instruction-set compatibility issues [12] that have hindered widespread acceptance of RISC processors, despite their cost-effective performance. Binary translation provides machines with the operations necessary to run the vast amount of software for Intel processors.

Another use of executable editing is program observation. Profiling and tracing tools, such as MIPS's *pixie* [22] or *qpt* [4], edit executables to record execution frequencies or trace memory references. These tools are widely used to study program or system behavior (e.g., [6,8]) and computer architecture (e.g., [5,11,29]). More recently, a tool based on EEL, Active Memory [16], dramatically lowered the cost of cache simulation—to a 2–7x slowdown—by inserting cache-miss tests before a program's memory references rather than post-processing an address trace. In addition, software development tools, such as Pure Software's *Purify* [13], detect programming errors, such as out-of-bounds memory references or memory leaks.

Finally, executable editing has also been used for global register allocation and program optimization [24,25]. Unlike most compilers, which operate on a single file, editing can manipulate an entire program, which permits it to perform interprocedural analysis rather than stopping at procedure boundaries.

Executable editing is conceptually easy, but complex in practice because of a myriad of architectural and system-specific details [15]. This complexity reduces the attractiveness of the technique by increasing the time and effort required to produce a robust tool. Ad-hoc systems are unlikely to employ reliable, general analyses for difficult constructs, such as indirect jumps. Moreover, differences among machines and operating systems lead to tools that are closely tied to one or two platforms and which are difficult to port to other systems.

EEL (Executable Editing Library) is a new C++ library that hides much of the complexity and system-specific detail of editing executables. EEL provides abstractions that allow a tool to analyze and modify executable programs without being concerned with particular instruction sets, executable file formats, or consequences of deleting existing code and adding foreign code. EEL greatly simplifies the construction of program measurement, protection, translation, and debugging tools. EEL differs from other systems in two major ways: it can edit fully-linked executables, not just object files, and it emphasizes portability across a wide range of systems.

Both aspects require new algorithms, which this paper describes. The first part describes EEL's machine-independent abstractions and the analysis underlying them. The second part describes how EEL is parameterized to be ported easily to new systems. The third part describes some measurements and applications of EEL.

2 Related Work

As the introduction relates, many tools modify executables to perform a wide range of tasks. However, in most tools, the application and executable modification are intertwined and details of the latter have not been published.

An exception is Srivastava and Wall's OM system [25], which is a library, similar to EEL, for modifying object files. OM internally represents instructions as RTL, which can be manipulated and translated back into machine instructions. OM's RTL and EEL's instructions serve the same roles. OM, however, uses relocation information from object files to analyze a program's control structure and to relocate the edited code. EEL, by contrast, directly analyzes and modifies a program's instructions, and consequently can operate on programs without relocation information, such as fully compiled and linked programs.[1] This facility comes at a price, as EEL requires more sophisticated program analysis and occasionally falls back on run-time code when static analysis is insufficient. However, this analysis also permits EEL to provide common functionality across vastly different systems.

Larus and Ball [15] described the ad-hoc analysis used by their profiling and tracing tool *qpt* to instrument executable files. EEL extends the earlier work by providing a general library for manipulating executables that is not tied to a specific application. EEL also shows that many problems raised in the earlier paper can be handled with more powerful program analysis.

ATOM [23] is a system that provides a simple interface to OM for adding instrumentation to programs. ATOM's interface is higher-level and more concise than EEL's (or OM's), which simplifies writing tools, but provides less control over the instrumentation process. For example, ATOM does not permit existing instructions to be modified[2] and invokes foreign code through a function call. ATOM's principle advantage is that foreign code can be written entirely in a high-level language. Figure 1 contains the EEL code to implement the same branch-counting application as discussed by Srivastava and Eustace [23]. The code for the

1. In the near future, EEL will supplement and verify its analysis with relocation information, when available, and will modify this information, which will permit editing of object files.

2. Newer versions provide a limited facility for changing instructions (Wall: personal communications).

```
int main(int argc, char* argv[])
{
  executable* exec = new executable(argv[1]);
  exec->read_contents();

  routine* r;
  FOREACH_ROUTINE (r, exec->routines())
    {
    instrument(r);

    while(!exec->hidden_routines()->is_empty())
      {
      r = exec->hidden_routines()->first();
      exec->hidden_routines()->remove(r);
      instrument(r);
      exec->routines()->add(r);
      }
  }
  addr x
    = exec->edited_addr(exec->start_address());
  exec->write_edited_executable(st_cat(argv[1],
                                ".count"),
                                x);
  return (0);
}

void instrument(routine* r)
{
  static long num = 0;
  cfg* g = r->control_flow_graph();
  bb* b;
  FOREACH_BB(b, g->blocks())
    {
    if (1 < b->succ()->size())
      {
      edge* e;
      FOREACH_EDGE (e, b->succ())
        {
        e->add_code_along(incr_count(num));
        num += 1;
        }
      }
    }
  r->produce_edited_routine();
  r->delete_control_flow_graph();
}
```

FIGURE 1. Instrumentation routines for a branch counting tool (see Srivastava and Eustace [23]).

```
mach_inst incr_count_code[] =
{
#include "incr_count.bin"
};

long incr_count_offsets[] =
{
#include "incr_count.oft"
};

class incr_count_snippet
    : public tagged_code_snippet
{
public:
  incr_count_snippet()
    : tagged_code_snippet(incr_count_code,
        sizeof(incr_count_code),
        NULL,
        NULL,
        incr_count_offsets,
        sizeof(incr_count_offsets))
        {
        }
};

code_snippet*
incr_count(long counter_num)
{
  assert(0 <= counter_num);

  tagged_code_snippet* snippet
          = new incr_count_snippet();
  addr counter_addr = COUNTER_START
          + counter_num * sizeof(long);

  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);
}
```

FIGURE 2. Low-level instrumentation for branch counting on a SPARC processor.

two systems are similar. However, a much larger difference is apparent in the low-level foreign code (Figure 2).

3 EEL Abstractions

EEL provides five major abstractions (C++ class hierarchies) that allow a tool to examine and modify an executable: executable, routine, CFG, instruction, and snippet.[1] An *executable* contains code and data from either an object, library, or executable file. A tool opens an executable, examines and modifies its contents, and writes an edited version. An executable primarily contains *routines*, the second abstraction, but also contains non-executable data. A tool can examine and modify routines in any order and place them, and new routines, in the edited executable in any order. EEL represents a routine's body with two further abstractions: *control-flow graphs (CFGs)* and *instructions*. A *CFG* is a directed graph whose nodes are basic blocks (single-entry, single-exit straight-line code sequences) and whose edges represent control flow between blocks [1]. EEL provides extensive control-flow and data-flow analysis for *CFGs*. Blocks contain a sequence of *instructions*, each of which is a machine-independent description of a machine instruction. A tool edits a CFG by deleting instructions or adding *code snippets* to blocks and edges. A snippet encapsulates machine-specific foreign code and provides context-dependent register allocation. EEL modifies calls, branch, and jumps to ensure that control flows correctly in the edited program.

EEL's abstractions are similar to those found in compilers, which is not surprising given that both systems manipulate programs. Like many recent compilers—such as *gcc* [26]—EEL's internal representation is a register-transfer

1. EEL also supports interprocedural analysis and call graphs, which are not described here.

level (RTL) instruction description [10]. A crucial difference, however, is that a compiler writer can choose RTL operations with clean semantics and translate constructs to a sequence of operations, while each EEL instruction must capture the semantics of a machine instruction.

The remainder of this section presents these abstractions in more detail and describes the analysis underlying them.

3.1 Executables

EEL executable objects are an abstraction of executable files—object, library[1], or static and dynamically-linked programs—that hide differences among file formats. Most operations are inquiries that return the location or size of a named entity, such as a routine. A few operations modify a program's state by changing a memory location or replacing or adding a routine. Most editing, however, is performed on a routine's control-flow graph, as described below.

Symbol table information in executable files is typically incomplete or misleading [15], which greatly complicates accurate analysis of a program. For example, compilers "hide" routines by not producing debugger symbol table information or put data tables in the text segment with a symbol table entry indistinguishable from a routine's. In addition, symbol tables commonly record only the starting point of a routine and do not distinguish multiple entry points arising from Fortran ENTRY statements or interprocedural jumps. Relocation information, when available, can refine this information.

EEL uses a more general, but sometimes less precise, approach and refines a symbol table by analyzing a program to find data tables, hidden routines, and multiple entry points. The analysis has several stages:

1. Initially, EEL reads a program's symbol table and eliminates all duplicate, temporary, and debugging labels in the text segment. It also discards labels that are not aligned on an instruction boundary or that are the target of a branch or jump (not call!) from the preceding routine (these are probably internal labels). The remaining labels form the initial set of routines. Each routine's initial entry point is its starting address.

2. If the executable has no symbol table (i.e., is stripped), the initial set of routines contains only the program's entry point and the first address in the text segment. In this case, EEL makes an extra pass over the program's instructions to find direct subroutine calls. These instructions' targets become the initial set of routines.

3. EEL then examines instructions to find jumps out of a routine or calls on routines not in this initial set. The destination of these control transfers become entry points to the routines that contain them. This analysis is conservative. It may find invalid entries, as for example, when data is interpreted as an instruction, but it does not miss entry points, which is important to construct accurate CFGs.

4. The key part of the analysis occurs when EEL constructs a routine's control-flow graph. A reachable, but invalid, instruction in a CFG leads EEL to assume that the routine contains data. However, unreachable instructions at the end of a routine comprise another routine, which EEL records in its symbol table and analyzes. As a side effect, recognizing a new routine may add entry points to existing routines.

This analysis refines the initial symbol table and allows each routine to be identified and processed individually, with full knowledge of its interprocedural linkages. In a stripped executable, the analysis finds all routines, but cannot recreate their names, which makes many tools, such as program profilers, far less useful.

EEL maintains symbol table information for the edited program. EEL uses it to produce debugging information for the edited executable, so that standard tools, such as debuggers, work for edited programs.

3.2 Routines

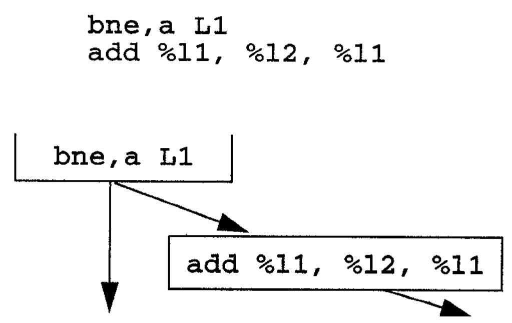
Routines are named objects in a program's text segment that contain instructions and data. EEL uses routines in two roles. First, they hold information about an entity in the text segment (its name, extent, entry points, etc.). Second, routines provide the interface to EEL's control- and data-flow analysis and editing facility, which is described below.

3.3 Control-Flow Graph

The primary program representation in EEL is a control-flow graph (CFG) of a routine. EEL represents a routine as a CFG, as opposed to a sequence of instructions, for three reasons. First, the initial application of EEL, *qpt*, required CFGs to implement efficient profiling and tracing by placing instrumentation on CFG edges [4]. Moreover, previous experience with simple tools showed that even they could reduce overhead by using control-flow information to place instrumentation intelligently. Second, EEL itself uses CFGs to adjust addresses in branch and jump instructions affected by editing.

Most important, CFGs provide an architecture-independent way of representing control flow. A key question is how to represent the semantics of instructions on a particular machine. Unlike compiler intermediate representations, which are the usual basis for CFGs, machine instructions can have internal control flow. For example, delayed branches in many RISC processors execute the subsequent instruction (in the branch *delay slot*) before transferring control. These instructions come in many variants, such as annulled branches that execute the delay slot only if the branch is taken. EEL explicitly represents instructions' internal control flow in a CFG, so that internal and external control flow are treated uniformly and instructions appear to have no control flow (i.e., are non-delayed branches). Figure 3 shows an instruction in an annulled branch's delay

1. EEL cannot yet modify libraries, although the extension is straight-forward. When complete, this feature will permit editing dynamically-linked programs by modifying their executable (already working) and dynamic libraries.



```
bne,a L1
add %l1, %l2, %l1
```

FIGURE 3. Example of CFG normalization. EEL CFGs explicitly represent control flow in instructions. In this case, the add instruction is in the delay slot of an annulled conditional branch and executes only if the branch is taken, so the add instruction appears along only one CFG edge.

slot, which is placed in its own basic block, which is linked to the appropriate outgoing edge of the branch's block. This process can repeat several times if the instruction from the delay slot is itself a delayed control-transfer. In a non-annulled branch, the delay slot instruction is duplicated along both edges.

With this explicit representation, a tool can add foreign code before or after almost any instruction without considering how the code interacts with local control flow, which means that the tool need not be aware of architectural details such as delayed branches. However, if left unreversed, duplicated delay slot instructions increase a program's size and execution time, so EEL folds instructions back into unedited delay slots.

EEL marks some CFG edges and blocks as uneditable, to simplify the process of producing executable code from an edited CFG. Most uneditable blocks and edges transfer control out of the current routine (e.g., the delay slot after a call), which would require interprocedural editing to place foreign code in another routine. Although 15–20% of edges and blocks are uneditable, it is usually easy to find an alternative location to edit (e.g., before the call).

Because EEL builds CFGs a single routine at a time, it treats subroutine calls specially. EEL uses a distinguished, zero-length basic block—after the block containing the call's delayed instruction—as a placeholder for the control transfer and possible side-effects of the subroutine's body.

In general, when control flow cannot be completely analyzed, run-time code ensures that control passes to the correct edited instruction [15]. EEL can perform several standard CFG analyses: dominators, natural loops, live registers, and slicing [1,28]. EEL uses them to improve the precision of control analysis and to reduce the need for run-time mechanisms. These analyses also provide an analytic basis for building tools.

Consider, for example, indirect jumps. Most indirect jumps occur in case statements, in which they jump through a dispatch table of addresses. EEL finds this type of table—in an architecture and compiler-independent manner—by computing a backward slice [14,28] from the jump instruction's registers. Although at the time of slicing, the CFG is incomplete, a path from the routine's entry to the jump must compute the dispatch table's address (or the jump would fail along the path). After finding the table's address, EEL builds a precise CFG for the indirect jump and subsequently modifies the table to point to edited locations. The same slice also can find the address used in the common idiom of an indirect jump to a literal value. If a slice fails to find a dispatch table or literal address, EEL marks the CFG as incomplete and inserts code to translate the jump's target address at run time.

Fortunately, EEL's slicing makes run-time translation a rare occurrence. We measured the frequency of unanalyzable indirect jumps in the SPEC92 benchmarks. On SunOS 4.1.3 using gcc version 2.6.2 and the Sun Fortran compiler, EEL found no unanalyzable indirect jumps among the 1,325 indirect jumps (and 1,027,148 instructions in 11,975 routines). On Solaris 2.4 using the SunPro compilers (version sc3.0.1), EEL found 138 unanalyzable indirect jumps among the 1,244 indirect jumps (and 1,185,018 instructions in 16,613 routines). All 138 indirect jumps resulted from optimizing a call in a return statement by popping the current stack frame and jumping to the callee. Six of these jumps occurred in compiled C code and the remainder were in the Fortran library. None of these jumps affect EEL, since EEL's CFG are intraprocedural.

3.3.1 Editing CFGs

A tool edits a routine's CFG by deleting instructions, adding new code before or after any instruction, or adding code along a control-flow graph edge. A *snippet* (Section 3.5) contains the new code. EEL accumulates edits without changing the CFG. In general, this batch style of editing works well since tools operate on the original CFG and need not see changes as they occur. Snippet call-backs (Section 3.5), which provide a final chance to modify an edit, easily handled the few exceptions.

After a tool edits a CFG, EEL produces a new version of the routine that incorporates the changes. Producing an edited routine involves laying out its blocks and snippets to minimize unnecessary jumps and adjusting displacements and addresses in control-transfer instructions—or occasionally replacing these instructions by snippets containing instructions with a longer span.

3.4 Instructions

EEL instructions are abstractions of RISC-like machine instructions. They divide instructions in functional categories and provide operations to inquire about semantics. The categories include memory references (loads and stores), control transfers (calls, returns, system calls, jumps, and branches), computations, and invalid (data). The categories are common to many machines, so a tool can analyze EEL

```
// 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);
}
```

FIGURE 4. Operation on instructions. This code computes a backward address slice for instructions that do not read memory or call routines (these are analyzed by other functions). It demonstrates how EEL instructions hide architectural detail, but still permit a tool to analyze a program. EEL's abstractions are in bold.

```
!! INCR_COUNT records a basic block or edge by
!! incrementing its counter in the count array.

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)
{
  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);
}
```

FIGURE 5. Sample code snippet (for the SPARC). Above the line is the snippet's body, which contains instructions to increment a profile counter. Labels before each line (e.g., "1*") name instructions that are customized for each counter. Below the line is *qpt* code that inserts a counter's address.

instructions in place of the underlying machine instructions. These categories cover simple RISC machines (e.g., MIPS and SPARC). Since categories are C++ classes, EEL can derive new ones that span boundaries. For example, the autoincrement load in HP's PA-RISC machines is both a memory reference and a computation. Combining classes, unfortunately, is unlikely to synthesize the semantics of CISC instructions, such as string edits. These instructions, however, are also difficult to analyze and instrument because of their dynamic behavior and internal control flow. The best representation may be a sequence of simpler instructions [25].

EEL provides many inquiries about an instruction's effect on a program's state (i.e., which registers it reads and writes, how it changes the program counter, or what its operation is). These inquiries provide enough information to analyze many aspects of a program. For example, Figure 4 contains code from *qpt* that computes a backward address slice for address tracing [14]. Because it operates on EEL instructions, this code is similar to the original algorithm and independent of an underlying machine.

To improve efficiency, EEL allocates only one instruction to represent all instances of a particular machine instruction. Typically, this optimization reduces the number of allocated EEL instructions by a factor of four.

3.5 Code Snippets

A code snippet encapsulates foreign code that is added to an executable. On one hand, EEL provides some system-independence for snippets since it allocates registers for them. On the other hand, snippets are the one point at which a tool is machine specific, since the code in a snippet is crafted for a machine. This is not a serious drawback, since the code is often short and carefully written for efficiency. A programmer writes a snippet's body in assembly language or a high-level language compiled to assembly language, in which case the snippet can be machine-independent. Figure 5 shows a sample snippet.

When a tool creates a snippet, it specifies the instructions, two sets of registers, and a call-back function (all, except the first, may be omitted). The first set contains registers used in the snippet that need to be assigned unused registers. EEL finds the live registers at the point at which the snippet is inserted and assigns dead (unused) registers to the snippet. If EEL cannot find enough dead registers, it wraps the snippet with code to spill registers to the stack. Sometimes, a snippet must use a particular register—for example, to record its value or to execute a subroutine call—and EEL should not spill or assign it. The second set specifies registers that cannot be used, even if free. This technique of register scavenging [15] is a way of utilizing unused registers in snippets.[1]

The final parameter to a snippet is a call-back procedure, which is invoked after register allocation, but before the instructions are placed in the modified program. The call-back procedure is passed the register-allocated instructions, their starting address in memory, and details of the register assignment. The call-back may modify the instructions (but cannot change their length). This mechanism has been used to adjust instruction displacements when an instruction's final location is known, record addresses for subsequent backpatching, and adjust code that records the stack pointer to discount the effects of EEL's spill code.

4 System-Dependent EEL

Beneath the machine-independent portions of EEL are system- and architecture-specific components that manipulate executable files and machine instructions. The first piece is a library to read and write Unix executable files. EEL currently uses the GNU *bfd* library [7], which is also used by the GNU assembler, linker, and debugger (*gdb*).

The second piece is an EEL-specific library to parse, decode, analyze, and modify binary instructions. Previous experience argued against implementing these routines by hand. A surprising number of bugs in *qpt* arose in machine-specific binary instruction manipulations. These bugs were of two types: improperly decoding or extracting an instruction field or omitting a particular instance from an analysis. EEL alleviates these problems by generating this low-level code from a concise, high-level machine description.

The tool *spawn* transforms a file of annotated C++ functions and a machine description into machine-specific code for analyzing and manipulating binary instructions. The code in the file defines the interface and functionality of EEL's machine-specific library (see Figure 6). The annotations identify points at which *spawn* needs to insert code, derived from a machine's description, to decode and manipulate a particular machine's instructions.

For example, consider the function in Figure 6, which creates an EEL instruction corresponding to a machine instruction. The function examines an instruction, to determine the class of the corresponding EEL instruction. The code for a particular class of instruction calls the EEL instruction constructor, passing it the machine instruction and some values extracted from instruction fields.

Spawn's annotations (in bold) specify instruction classes and attributes. *Spawn* processes the code and replaces the annotations with machine-specific code to dispatch on an instruction's type and to extract and modify instruction fields. For example, in the memory-referencing instructions, *spawn* replaces the annotation `{{WIDTH}}` by the number of bytes of accessed memory. *Spawn* is currently unaware of a system's subroutine and system call conventions [3], so these instructions require additional processing to distinguish overloaded instruction uses. For example, the code in the figure resolves the SPARC's three overload uses of a jump instruction.

```
//
// Return the EEL instruction corresponding to
// the machine instruction INST in executable
// EXEC at address PC.
//

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);
    };;
    LOAD STORE::
      return new
        memory_load_store_instruction(inst,
          {{WIDTH}});;
    LOAD::
      return new
       memory_load_instruction(inst,{{WIDTH}});;
    STORE::
      return new
       memory_store_instruction(inst,{{WIDTH}});;
    SYSTEM:: {
      if ((*inst & TRAP_COND) == TA
         && IMM(*inst) == ST_SYSCALL) {
        mach_inst *i = inst - 1;
        return new
          system_call_instruction(inst,
            {{INST i LITVAL}});
      } else return new
          system_call_instruction(inst, -1);
    };;
    VALID::
      return new computation_instruction(inst);;
    ANY::
      return new invalid_instruction(inst);;
  }}
}
```

FIGURE 6. EEL machine-specific code. This function builds an EEL instruction corresponding to a machine instruction. *Spawn* replace annotations (in bold) in the C++ code with machine-specific code that dispatches on different instruction types and extracts or modifies instruction fields.

Spawn derives its machine-specific information from a machine description, which specifies both instruction syn-

1. It is not, however, a way of freeing a register across the entire program for the foreign code. Later releases of EEL will provide a mechanism to free a register.

tax (i.e., encoding) and semantics. The syntax description is similar to the one in Ramsey's retargetable debugger [18] and NJ Machine Code Toolkit. *Spawn* extends Ramsey's work by expressing instructions' semantics with a simple register-transfer description of instruction semantics [9].

Spawn descriptions are concise and easily derived from processor architecture manuals. They first describe registers and instruction fields by specifying their width and, for registers, a type for use in semantic expressions. Each instruction is described by a pattern that identifies its binary encoding and a semantic expression that describes its operation and internal control flow.

Spawn, borrowing from Ramsey, directly supports the common convention of expressing instruction encodings as a matrix of instruction names. *Spawn*'s concise instruction encoding encourages complete specification of an instruction set, which allows spawn-generated code to reliably detect invalid instructions and enables EEL's control-flow analysis to distinguish data from instructions by detecting where control passes to an invalid instruction.

Figure 7 shows a portion of *spawn*'s SPARC description. To make a description concise, similar instructions are grouped together and described by a common semantic function (which can be parameterized for small differences among instructions). For example, this figure contains the semantics of many SPARC control-transfer instructions. The function `branch` describes all conditional branches. It consists of an expression parameterized by a condition code register (i.e., integer or floating point) and a branch test. When these arguments are bound, the expression describes a particular branch instruction's semantics. The description also contains minimal timing information: the semicolon in the function's body indicates that the first statement executes before the second statement (which overlaps the next instruction's execution). Immediately below its description, this function is applied to the integer condition codes (PSR) and a vector of test conditions, yielding a vector of fully instantiated semantic functions. The semantic statements (`sem`) binds each function to the corresponding instruction (whose encoding was defined previously).

Spawn extracts much information about a machine's instructions and registers from a machine description. It determines a classification for each instruction (jump, call, store, invalid, etc.). It finds registers that each instruction reads and writes and literal values in instruction fields. It finds the number of registers in each register set and their width and type. It even generates C++ code to replicate the computation in most instructions, such as computing the target address of a jump or load and the result of an add.

Machine descriptions of this type are far more concise than hand-written code to manipulate instructions. For example, the SPARC description is 145 non-comment, non-blank lines and the mostly machine-independent annotated C++ file is 504 lines. The handwritten equivalent is 2,268 lines (*spawn* produces a file 6,178 lines long). For compari-

```
// 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

// General purpose register set
//
register integer{32} R[35]
alias integer{32} PSR is R[32]
alias integer{32} FSR is R[33]

register integer{32} pc

// 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]


// 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]

sem [fbu fbg fbug fbl fbul fblg fbne fbe
     fbue fbge fbuge fble fbule fbo]
 is branch FSR @ ['u 'g 'ug 'l 'ul 'lg 'ne 'be
     'ue 'ge 'uge 'ble 'ule 'o]

sem call is
  t:=pc+(#(integer{32})disp30<<2),  R[15]:=pc;
    pc:=t
sem jmpl is t:=addr,  R[rd]:=pc; pc:=t
```

FIGURE 7. Portion of *spawn*'s SPARC description. The first part defines resources such as registers and instruction fields. The middle part defines the encoding (syntax) of some control-transfer instructions. The final part defines these instructions' semantics. In this description, the keyword "val" introduces a (function) binding. The statements started by "sem" define the semantics of instructions in their first argument by the corresponding semantic function in their second argument. A comma separates operations that can execute in parallel. A semicolon separates sequential operations.

son, a spawn description of the MIPS R2000 architecture is 128 lines and the Digital Alpha architecture is 138 lines.

5 Experience

EEL currently runs on workstations with SPARC processors, under SunOS and Solaris (an older version also ran on MIPS under Ultrix). We rewrote *qpt* to use this library. In the process, *qpt* dropped from 14,500 non-comment, non-blank lines of C code to 6,276 lines of C++ code, of which 975 lines are system-dependent manipulations of snippets (which contain 116 lines of assembly code). More importantly, the new *qpt* is far easier to understand and modify and contains several machine-specific optimizations that were too cumbersome to implement in the old system.

Tool Version	Program Size (bytes) (text & data)	Run Time (sec.) (user + system)
qpt	246,784	4.4
qpt -O2	164,864	3.5
qpt2	950,240	19.0
qpt2 -O2	810,976	8.4
qpt2 -ND	868,320	18.0
qpt2 -O2 -ND	720,864	7.7

TABLE 1. Comparison of *qpt* and *qpt2*. *qpt* is the original program profiler. *qpt2* is a new profiler, based on EEL, that uses the same algorithms. Both tools instrumented a small program, *spim,* that consists of 320,536 bytes of text and data. Programs were compiled with gcc (g++) version 2.6.3 and run on a SPARCstation 20/61, with a 60Mhz SuperSPARC processor and local disk. Times are best of three runs. Base versions were compiled without optimization. The -O2 versions were compiled at that optimization level. The -ND versions were compiled without assert statements (*qpt* contains no asserts).

EEL consists of 13,960 non-comment, non-blank lines of C++ code, of which 1,501 lines are system-dependent manipulations of snippets (which contain 114 lines of assembly code), 410 lines are executable-format specific, and 2,268 are (handwritten) architecture-specific code. Unfortunately, this reduction in a tool's program length and complexity comes at the cost of increased tool size and execution time. Table 1 shows the size and running time of the old (*qpt*) and new (*qpt2*) versions, compiled several ways. Without optimization, the new version runs 4.3 times slower. Optimization, however, narrows the gap to 2.4 times slower. It is worth noting that *qpt2*'s performance is still acceptable. These measurements used the hand-written machine specific code, even though the spawn-generated code ran at the same speed.

The time gap is attributable to inefficiencies introduced by C++ and the style in which EEL is written. Turning on optimization narrows the gap because at -O2, g++ produces much better code and inlines member functions, which greatly reduces the overhead of EEL's abstract datatypes. Better compilers should further reduce the C++ penalty.

EEL, because of its object-oriented programming style and its explicit program representations, allocates many more objects (317,494 vs. 84,655) than the old code, which itself increases execution time. In particular, EEL's CFGs are larger (26,912 vs. 15,441 blocks[1]), which disproportionately increases execution time because many CFG algorithms are non-linear.

To date, we have used EEL to build four other tools. Alvin Lebeck and David Wood built Active Memory [16], which is a platform for efficiently simulating memory systems. It inserts a quick test before load and store instructions to check the state of the accessed location. Different states invoke handlers to perform tasks such as cache simulation. Active Memory exploits EEL's ability to insert foreign code efficiently and to add many routines (another program, in fact) to an executable. Steven Reinhardt built a direct-execution architectural simulator called Elsie. Elsie replaces loads, stores, and system calls in a program with simulator calls (using EEL) and then loads the edited executable into the simulator. Sashikanth Chandrasekaran is rewriting the Wisconsin Wind Tunnel architectural simulator [19] using EEL. We also used EEL to re-implement Blizzard-S's fine-grain access control [20]. The old version of Blizzard-S used code from *qpt* to insert access-control tests. The new version greatly improves performance with several optimizations that would have been difficult to implement in the old system. For example, one optimization exploits EEL's live register analysis to insert a faster test sequence when condition codes are not live. In spite of these optimizations, the new version consists of roughly 1,300 lines of code (exclusive of EEL), while the old version contains approximately 2,800 lines (exclusive of *qpt*-specific code).

6 Conclusions

EEL is a library that aids programmers in writing portable tools to edit executable programs. Tools to modify executables have proven their value in many areas. However, these tools are difficult and expensive to develop and usually are specific to an architecture and operating-system. EEL addresses these problems by providing a mostly architecture- and entirely system-independent set of operations to read, analyze, and modify code in an executable file. EEL itself is highly portable because of its extensive program analysis and because its machine-specific portion is derived from a concise machine description. EEL does not solve all problems in executable editing (self-modifying code and unrestricted indirect jumps and calls are open problems) but

1 The two programs use slightly different definitions of a basic block (EEL's blocks end at calls). However, a more important difference is EEL's 12,774 delay slot blocks, 920 CFG entry/exit blocks, and 1,942 call surrogate blocks, none of which exist in the old code

it simplifies the analysis and manipulations of most programs.

For an EEL programmer's manual and information on the status of EEL, check:
`http://www.cs.wisc.edu/~larus/eel.html`

Acknowledgments

Norman Ramsey and Mary Fernandez kindly provided copies of their machine descriptions and information on their toolkit. Amitabh Srivastava, Tom Ball, Trishul Chilimbi, Mark Hill, Alvin Lebeck, Anne Rogers, Brad Richards, David Wall, and David Wood provided helpful comments on drafts of this paper. Sashikanth Chandrasekaran, Babak Falsafi, Alvin Lebeck, and Steve Reinhardt and other members of the Wisconsin Wind Tunnel project bravely used a new tool and contributed many helpful suggestions.

References

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[2] Kristy Andrews and Duane Sand. Migrating a CISC Computer Family onto RISC via Object Code Translation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 213–222, October 1992

[3] Mark W. Bailey and Jack W Davidson A Formal Model and Specification Language for Procedure Calling Conventions In *Conference Record of POPL '95· 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–310, January 1995

[4] Thomas Ball and James R. Larus Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, 16(4).1319–1360, July 1994.

[5] Anita Borg, R. E. Kessler, and David W. Wall. Generation and Analysis of Very Long Address Traces. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 270–281, May 1990.

[6] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying Behavioral Differences Between C and C++ Programs. *Journal of Programming Languages*, 1995 To appear.

[7] Steve Chamberlain *libbfd: The Binary File Descriptor Library*. Cygnus Support, bfd version 3 0 edition, April 1991.

[8] J. Bradley Chen and Brian N Bershad The Impact of Operating System Structure on Memory System Performance In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles (SOSP)*, pages 120–133, 1993.

[9] Jack W. Davidson and Christopher W. Fraser. Code Selection through Object Code Optimization *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984

[10] Jack W. Davidson and Christopher W. Fraser Register Allocation and Exhaustive Peephole Optimization. *Software Practice & Experience*, 14(9).857–865, September 1994

[11] Amer Diwan, David Tarditi, and Eliot Moss. Memory Subsystem Performance of Programs Using Copying Garbage Collection. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1994.

[12] Tom R. Halfhill Emulation· RISC's Secret Weapon. *Byte*, pages 119–130, April 1994

[13] Reed Hastings and Bob Joyce. Purify· Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter Usenix Conference*, pages 1–12, January 1992

[14] James R Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practice & Experience*, 20(12):1241–1258, December 1990.

[15] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software Practice & Experience*, 24(2):197–218, February 1994.

[16] Alvin R Lebeck and David A Wood. Active Memory· A New Abstraction for Memory-System Simulation In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1995. To appear

[17] Pure Software. United States Patent 5,193,180, March 1993.

[18] Norman Ramsey and David Hanson. A Retargetable Debugger. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, pages 22–31, June 1992.

[19] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[20] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K Reinhardt, James R. Larus, and David A Wood. Fine-grain Access Control for Distributed Shared Memory In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[21] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G Robinson. Binary Translation. *Communications of the ACM*, 36(2):69–81, February 1993

[22] Michael D. Smith. Tracing with pixie. Memo from Center for Integrated Systems, Stanford Univ , April 1991.

[23] Amitabh Srivastava and Alan Eustace. ATOM A System for Building Customized Program Analysis Tools In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994

[24] Amitabh Srivastava and David Wall Link-Time Optimization of Address Calculation on a 64-bit Architecture. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 49–60, June 1994.

[25] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[26] Richard M. Stallman *Using and Porting GNU CC*. Free Software Foundation, October 1993. For GCC Version 2.5

[27] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles (SOSP)*, pages 203–216, December 1993.

[28] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[29] Cheryl A. Wiecek. A Case Study of VAX-11 Instruction Set Usage for Compiler Execution. In *Proceedings of Symposium on Architectural Support for Programming Languages and Operations Systems*, pages 177–184, April 1982.