Abstract Execution:
A Technique for Efficiently Tracing Programs

James R. Larus
larus@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706

February 9, 1990

Abstract

Many areas of computer performance analysis require detailed traces of events that occur during a program's execution. Collecting traces is expensive. The additional code required to record events greatly slows a program's execution. In addition, the resulting trace files can grow unmanageably large. This paper describes a technique called *Abstract Execution* that alleviates both problems.

Abstract Execution records a small set of events during the traced program's execution. These events serve as input to an abstract version of the program that generates a full trace by reexecuting selected portions of the original program. This process greatly reduces both the cost of tracing the original program and the size of the trace files. The cost of regenerating a trace is insignificant in comparison to the cost of applications that use it.

This paper also describes a system called AE that implements Abstract Execution. The paper contains measurements that demonstrate that AE can efficiently trace large programs.

Contents

1 Introduction

Many areas of computer performance analysis require detailed traces of events that occur during a program's execution. A few obvious examples include: cache memory simulation, which requires a list of memory locations accessed by a program [10]; performance analysis, which requires a count of each basic block's execution frequency [8]; and program parallelism analysis, which requires a trace of addresses and notification upon loop entry and iteration [7]. Traditionally, collecting this information is expensive. The mechanism that records events severely slows a program's execution. This slowdown not only makes large, long-running programs difficult to characterize, but it also affects the behavior and measurements of real-time and parallel programs. Equally serious is the cost of storing and manipulating huge trace files. A 10 million instructions per second (MIPS) computer produces 40–60 megabytes of trace data in a second of execution. Clever compression schemes can reduce the volume of data by an order of magnitude [9], but the size of traces is a major limitation on measuring long-running programs.

This report describes a new technique called *Abstract Execution* that greatly alleviates both problems. The technique—which has been implemented in a system called AE—records a small subset of the events produced by a program. When a full trace is needed, the system reexecutes small portions of the program—using the traced events as a guide—to generate a full trace. Abstract Execution adds only 50–80% to the traced program's execution time and compresses the trace data by 2–3 orders of magnitude.

This report has five sections. The next contains an extended example that introduces the technique. Section 3 describes Abstract Execution. The next section is a detailed description of AE. Section 5 briefly describes some applications of AE. And, the final section surveys related work. The appendix contains details of the MIPS R2000 implementation of AE and a description of the machine-specific information required by AE.

2 Example of Abstract Execution

The example in this section illustrates major aspects of Abstract Execution. The example is self-contained and can be read as a quick summary of the rest of the paper. Consider the C program in Figure 1. It sets each element in a 100×100 array to the sum of its coordinates, so when it finishes, $a[i, j] = i + j$

```
int a[100][100];

main ( )
{
  int i;

  for (i = 0; i < 100; i = i + 1)
   sub (a[i], i);
}

sub (x, i)
  int x[];
  int i;
{
  int j;

  for (j = 0; j < 100; j = j + 1)
   x[j] = i + j;
}
```

Figure 1: Example C program. It sets each element of the array `a` to the sum of its coordinates, so `a[i, j] = i + j`.

for $0 \leq i, j < 100$. We want to produce a full address trace that lists the memory address of each executed instruction and of each location read or written by an instruction.

We produce this trace by compiling the program with a modified version of the GNU C compiler. This compiler instruments the compiled code to record important events and produces a *schema file* that describes how to interpret the trace data to recreate a full address trace. To understand this process, we first need to examine the assembly code produced by the compiler (Figure 2). This code is for the MIPS R2000 processor and omits, for clarity, the procedure entry and exit code that saves and restores registers.

Figure 3 contains the schema for this program. A schema delimits each function and basic block in a program. The schema fully describes instructions that contribute to the calculation of a memory address for a load or store instruction. These instructions' values can either be recorded during execution in the trace file (`unknown_defn`) or recalculated when the schema is interpreted (`compute_defn`). Placeholders (`uneventful_inst`) represent instructions that do not contribute to the flow of control or to address cal-

```
main:
 # prologue omitted

        ori R16, R0, 0
        ori R19, R0, 99
        la  R18, sub
        la  R17, a
        j   L2
L5:
        addiu R29, R29, -16
        addu  R4, R0, R17
        addu  R5, R0, R16
        jal   R31, R18
        addiu R29, R29, 16
        addiu R17, R17, 400
        addiu R16, R16, 1
L2:
        ble   R16, R19, L5

 #epilogue omitted


sub:
 # prologue omitted

        ori R3, R0, 0
        ori R6, R0, 99
        j   L7
L10:
        addu  R2, R5, R3
        sw    R2, 0(R4)
        addiu R4, R4, 4
        addiu R3, R3, 1
L7:
        ble   R3, R6, L10

 # epilogue omitted
```

Figure 2: Compiled code for program in previous figure. Details of register saving and restoring and procedure call linkage are omitted for clarity. R0 always contains the constant 0. `ori` is a or-immediate operation, which loads a constant into a register. `la` is a load-address operation. `addiu` is an add-immediate unsigned operation and `addu` is an add unsigned operation. `jal` is jump-and-link, a procedure call. `ble` is a branch on less-than-equal operation. `sw` is a store word operation.

```
start_function main 4
... prologue code omitted ...
start_block 0
uneventful_inst
uneventful_inst
uneventful_inst                 ; la requires 2 instructions
compute_defn_2 R18 #I0 + #Ssub
uneventful_inst                 ; la requires 2 instructions
uneventful_inst
end_block_jump 0 2 %loop_entry(2 0)

start_block_target 1
uneventful_inst
uneventful_inst
uneventful_inst
call_inst sub
uneventful_inst
uneventful_inst
uneventful_inst
end_block 1 %loop_back(2 0)

start_block 2
end_block_cjump 2 3 1 %loop_exit(3 0)

start_block_target 3
end_block 3
... epilogue code omitted ...
end_function main

start_function sub 4
... prologue code omitted, but includes unknown_defn R4
start_block 0
uneventful_inst
uneventful_inst
end_block_jump 0 2 %loop_entry(2 0)

start_block_target 1
uneventful_inst
store_inst 0(#R4)
compute_defn_2 R4 #R4 + #I4
uneventful_inst
end_block 1 %loop_back(2 0)

start_block 2
end_block_cjump 2 3 1 %loop_exit(3 0)

start_block_target 3
end_block 3
... epilogue code omitted ...
end_function sub
```

Figure 3: The schema produced for the sample program. `compute_defn` operators define a value that can be computed when the schema is interpreted. `unknown_defn` operators cannot be computed later, so their result is saved in the trace file. `uneventful_inst` do not contribute to the calculation of an address. In real schemas, consecutive `uneventful_inst` are combined into one placeholder.

culations. These instructions compute and use values. The program's flow of control is dynamically recorded at conditional branches, where it cannot be predicted from the schema.

Several aspects of the schema are worth noting. First, the array reference in `main` does not produce an event since no load or store instruction in that routine uses the address. In the routine `sub`, this address appears as a parameter and is recorded as an `unknown_defn` since it is used in a store instruction. Second, certain instructions in the assembly code (e.g., load address: `la`) are actually assembler macro instructions that expand to multi-instruction sequences.

Another compiler (`aec`) translates a schema into a C program that reads the trace file produced by the instrumented program and uses it to generate a full address trace. For example, the schema for the `sub` function translates into the code in Figure 4.

Executing the sample program produces a 11,005 byte trace file (on a MIPS R2000 processor). The full address trace contains 72,737 addresses (62,325 instructions and 10,412 memory references) and consumes 363,685 bytes (5 bytes per address including a byte for the reference type). AE compressed the trace data 33 times. By subsequently compacting the trace file with the Unix `compress` utility (to 630 bytes), we can increase this figure to 577 times.

3 Abstract Execution

Abstract Execution is a general technique for tracing incidents during a program's execution. Assume that we want to notice and record occurrences of a set of events E during the execution of program P with input I (see Figure 5). Abstract Execution derives a program P' that generates the set of events E, but is simpler and faster than program P. The set of significant events SE recorded in the original program P drives program P'. SE should be smaller than E so the tracing does not disrupts the original program and so the trace file is small.

This formulation allows a continuum of interpretation programs. At one extreme, we execute an instrumented version of the original program and record events directly, so $SE = E$ and P' just reads this file. At the other extreme, P records nothing and the regeneration program is an instrumented version of the original program, so $P' = P$ and $SE = I$. In general, we want a point between the extremes so the size of SE is much less than the size of

```
/* Start of block 0 */
  L0:
  PC = in_pc+20;
  ISSUE_INST(4);
  ISSUE_INST(4);
  ISSUE_INST(4);
  LOOP_START(1);  goto L2;
/* End of block 0 */
/* Start of block 1 */
  L1:
  PC = in_pc+32;
  ISSUE_INST(4);
  ISSUE_INST(4);
  WRITE_MEM(R4+0);
  ISSUE_INST(4);
  R4 = R4+4;
  ISSUE_INST(4);
  LOOP_CONT(1);/* End of block 1 */
/* Start of block 2 */
  L2:
  PC = in_pc+48;
  ISSUE_INST(4);
  GET_BYTE(temp); /* Read from trace file */
  switch(temp)
    {
    case 1: goto L1;
    case 3:  LOOP_END(1); goto L3;
    default: cjump_err();
  }
/* End of block 2 */
/* Start of block 3 */
  L3:
  PC = in_pc+52;
/* End of block 3 */
  ISSUE_INST(4);
  R8 = R30+0;
  ISSUE_INST(4);
  READ_MEM(R8+-16);
  ISSUE_INST(4);
  READ_MEM(R8+-12);
  ISSUE_INST(4);
  R29 = R8+0;
```

Figure 4: Portion of the regeneration code for the schema in the previous figure. The schema for the sub function translates into the C code above, which generates a full address trace from the trace information recorded by the function.

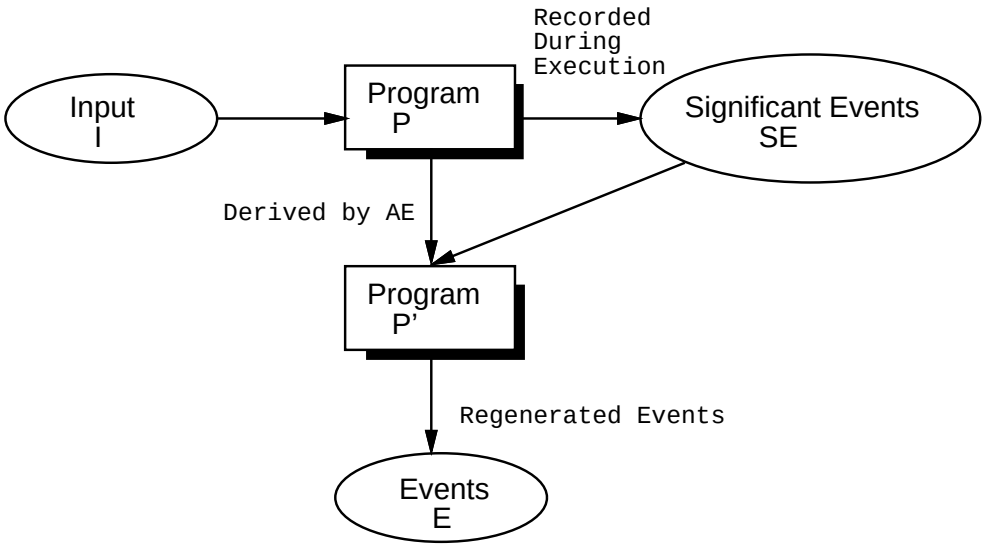

Figure 5: Process of Abstract Execution. Instead of directly instrumenting a program P to record a set of events E, Abstract Execution collect a smaller set of significant events SE and derives a program P' that uses SE to generate the desired events.

E and the cost of running P' is less than the cost of directly measuring P.

These goals conflict. The principal way to reducing the size of SE is to shift computations to program P' instead of recording values in program P. P' can perform a computation that depends only on values that are statically determinable from program P. These computations increase the cost of P', but decrease the size of SE.[1] The alternative is to save the result of a computation in P and read the value in P'. This approach may reduce the cost of P, but it increase the size of SE.

It might seem as if P' needs to recompute most of the original program. In fact, AE only recalculates instructions in a program slice for each event that must be recorded. A *program slice* with respect to an instruction I is the set of instructions in the program that directly or indirectly affect the value produced by I [6, 12]. For example, to produce an address trace, we are only interested in instructions that contribute to addresses used in load or store instructions and can ignore the other instructions that produce or

[1] However, reading the value from the trace file has an non-trivial cost, so a simple recalculation may be cheaper.

utilize the values that are loaded and stored. P' also needs to know whether an instruction in a slice executed, so AE also traces the program's flow of control. Each instruction in a slice can either be reexecuted by the regeneration program or its value can be recorded in the trace file. The choice depends on the complexity and cost of the recalculation and is discussed below.

Abstract Execution offers two advantages over other techniques of program tracing. First, since SE is smaller than E, it reduces the amount of data that must be stored. Previous systems attempted to reduce the size of files by compacting them with a data compression algorithm. As we will see, these techniques do not reduce a trace as well as AE and they can be applied to the file SE to increase AE's advantage. Second, Abstract Execution reduces the cost of tracing P. This reduction is valuable in its own right if P is long-running or if tracing affects the program's behavior. However, it may appear as if this reduction is illusionary since the cost of P plus the cost of P' can be larger than the cost of directly tracing P—ignoring system overhead required to write larger files. AE's advantage is that P' can run many times, thereby amortizing the cost of one run of P over many uses of the data. In addition, P' is smaller and simpler than the original program and uses a bounded amount of memory, so it is better suited to being linked with a program that consume the full trace.

Abstract Execution is a general technique that can record any type of event during a program's execution. In this discussion, we will assume that the events of interest are the memory addresses of the executed instructions and the referenced data, in other words, a full address trace. This is a common application of program tracing and, because of the volume of data, a demanding test of a tracing technique. We will also show that other types of events can easily be incorporated into the framework.

4 Details of AE

Figure 6 depicts the overall structure of the AE profiling system. Its first part is a modified version of `gcc`—the GNU C compiler [11]. In addition to compiling a program (`foo.c`), this version of the compiler produces a schema file (`foo.sma`) and inserts code into the executable program to record significant events. The compiled program is linked with startup code (`aecrt0.o`). When the program runs, it produces a trace file (`ae.out`) that contains the significant events. The process is similar to the Unix `prof` or `gprof` profiling

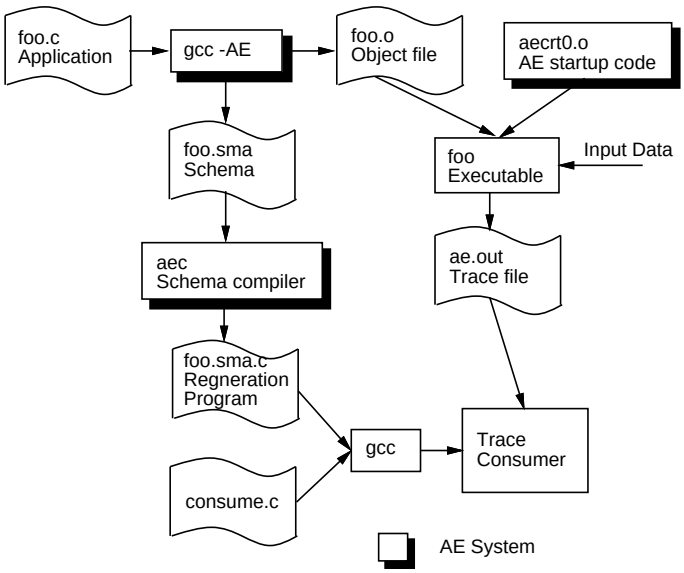

Figure 6: Overview of AE system. The -AE flag to the GNU C compiler (`gcc`) causes it to instrument the compiled program and produce a schema file (`foo.sma`). The schema file describes how to interpret the trace file (`ae.out`) produced by running the instrumented program. The schema compiler (`aec`) translates a schema into a C program (`foo.sma.c`) that reads the trace file and produces a full program trace. This program is linked with an application program (`application.c`) that uses the trace information.

systems [5].

To produce a full trace from the significant events, AE translates the program's schema into a C program that interprets the significant event trace file to generate a full trace. The schema compiler (`aec`) translates schema files into a C program. This program can be compiled and linked with a program that uses the trace data. Alternatively, the full trace can be written to a file or Unix pipe to supply input to an existing program.

The regeneration program is specific to the program compiled by `gcc`. When the original program is compiled again, `gcc` produces a new schema file, which must be translated again by `aec`. However, a regeneration program correctly interprets any `ae.out` file produced by the instrumented program, so it may be used to analyze many different executions of the instrumented program.

The rest of this section provides details about AE. The first part de-

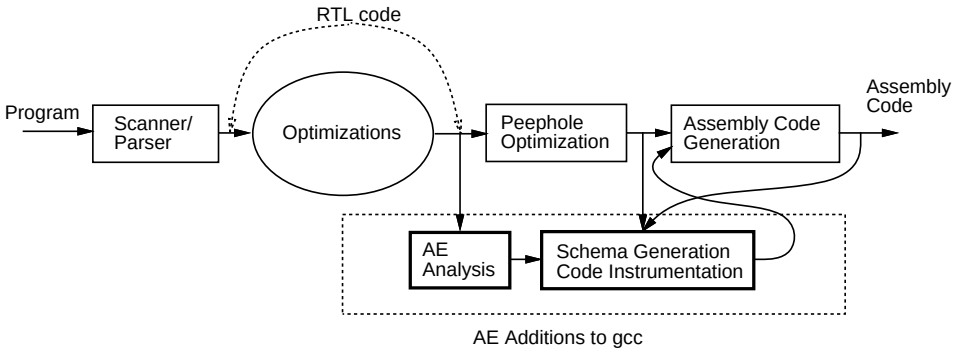

Figure 7: Organization of gcc and AE's additions to it. gcc's internal representation of a program is register-transfer code (RTL). After applying a variety of optimizations, gcc performs peephole optimization on this code and then produces assembly instructions. AE examines the RTL code immediately before peephole optimization (but, after all other optimization) and watches as it is translated to machine instructions.

scribes the changes to gcc to accommodate AE. The second part describes aec and the translation process.

4.1 Changes to gcc

The code added to gcc to implement AE was carefully designed to require minimal changes to the compiler. This requirement was desirable and necessary because gcc is an evolving program that is developed and maintained by other people. As new versions emerge, AE can be quickly "snapped" into place so it always runs in latest version of this compiler. Machine-specific information used by AE is collected into a single file in a manner similar to gcc. Appendix B describes this information in detail.

To understand where AE fits into gcc, we must briefly examine this compiler. Figure 7 illustrates the organization of gcc. This program is a well-engineered, conventional compiler that has been retargeted to a wide variety of computers. Its parser translates a program into an intermediate form called RTL, which is a register-transfer level representation of the program. This code is similar to the instruction set of a simple load/store computer that has an infinite supply of registers and a very orthogonal instruction set. All optimizations—such as strength reduction, dead code elimination, and many others—modify the RTL instructions.

In the final stage, peephole optimization tries to combine an RTL instruc-

tion with its successors. Then, code generation translates a RTL instruction into one or more instructions for the target machine. `gcc` only represents a program as RTL code. It never creates a machine language representation of the program, except in the output file.

AE interfaces to this process in three places. First, immediately before the program is put through the peephole optimizer and translated into assembly code, AE analyzes the entire RTL code sequence to find basic block boundaries, loops, and the instructions that must be instrumented. Second, after each RTL instruction is run through the peephole optimizer, but before it is translated to machine instructions, AE produces tracing code that must run before the instruction, e.g., to record the address of a load instruction. AE also examines the assembly code to count the number of machine instructions produced by `gcc` for a RTL instruction. This division of labor reflects the constraints of `gcc`. AE needs to analyze the entire body of fully-optimized code, so it must run immediately before peephole optimization, which produces non-standard RTL code. But, AE must also watch the assembly code to create a schema that accurately reflects the program.

4.1.1 Analysis

AE heavily analyzes the RTL program produced by `gcc`. Its first analysis—reaching definition dataflow analysis—has two roles. First, it identifies the basic blocks in a function. Although `gcc` also finds blocks, its optimizations that modify the flow of control typically corrupt this information. In addition, `gcc` only analyzes control flow when it optimizes a program. Therefore, basic block information is unavailable for programs compiled without the `-O` flag.

Reaching definitions analysis identifies the assignments to a register whose values reach a use of the register [2]. This analysis determines which instructions produce a value used in a memory address calculation. AE works backwards from a load or store instruction to find other instructions that contributed to the memory address. Assume that AE wants to determine the value in `R1`. Definitions reaching a use of `R1` can be divided into three categories:[2]

- *Easy instructions* of the form: `mov R1, <constant>`. These instructions move a constant, e.g., 0, 1, or the address of a global symbol,

[2]Note that all RTL instructions operate on registers since this code models a load/store architecture.

into a register.

- *Hard instructions* of the form: `add R1, R2, R3`. To calculate the value of *R1*, AE must know the contents of `R2` and `R3`, which requires a recursive application of this analysis.

- *Impossible instructions* of the form: `load R1, 4(R2)`, a function call that produces a result in `R1`, or if `R1` is an incoming argument. AE cannot predict what value will be in this register, so it instruments the program to record the value in `R1`.

Only impossible instructions require additional code to record a significant event. The first two types of instructions result in a `compute-defn`, which recalculates the value when the full trace is generated.

The regeneration program also needs to know the control flow in the original program. The minimal information necessary to recreate the flow is a record of which branch was taken in each conditional jump.[3] The program's control-flow graph identifies the instructions that are targets of conditional branches. Before each target instructions is written to the output file, AE adds code to record the instruction's basic block number in the trace file. The flow of control between all other instructions—unconditional jumps and computational instructions—is apparent from the schema file.

AE's other analysis finds loops in a program. This information identifies when each loop starts, finishes, and begins a new iteration. `gcc` also marks a rough outline of the loops in the `RTL` code. Although these markings suffice for `gcc`, they are inadequate for AE since they do not identify all edges exiting a loop or adequately distinguish the beginnings of nested loops.

AE uses a more refined analysis technique to identify control-flow edges that enter a loop, exit a loop, and begin a new iteration of a loop in a three step process. The first step identifies loop backedges and the blocks that head a loop by finding the control-flow edges whose destination dominates their source. It then uses a greedy algorithm to find all blocks within the loop. Finally, AE examines the edges from these blocks to find the exit and backedges for the loop.

[3] If AE knew the values compared in a conditional statement, it could omit the control-flow event. Although these quantities could be recorded in a manner similar to addresses, their values are in a different domain than addresses. Recomputing them might require performing most of the original program's computation.

4.1.2 Schema Generation and Instrumentation

The other part of AE examines each instruction after it has passed through peephole optimization, but before it is translated to machine code. This part produces one or more schema instructions that describe the instruction and various annotations that identify whether the instruction is the beginning or end of a program structure such as a basic block or program loop. Since an RTL instruction may translate to more than one machine instruction, AE watches the assembly code to determine the number of `uneventful_inst` produced for an RTL instruction.

AE also adds assembly code to the output file to record significant events. These sequences are written directly to the assembly output file, either immediately before or after an instruction. Currently, there are two types of significant events. The first records the number of a basic block that is the target of a conditional branch. This quantity is typically a byte, but it can be a half word in functions containing more than 256 blocks. The other type of event records the value produced by an impossible instruction. This quantity is a full word. On a MIPS R2000, the instruction sequence to record a significant event is 6 instructions for the first event in a basic block (which checks for sufficient buffer space) and 3 instructions for all subsequent events in the block. On computers without the ability to do unaligned stores (e.g., SPARC), a full word must be stored one byte at a time.

4.2 AEC

The other half of AE is a compiler (`aec`) that translates a schema into a C program that generates a full address trace from the limited information in the file of significant events. This translation is simple, so we will only outline it and describe a few interesting aspects.

`aec` takes as input the executable program and the schema files from each constituent C file. It produces a file containing a C program.[4] The resulting program has a simple and fixed structure. It begins with collection of boilerplate macro definitions, which make the translated code smaller and easier to read. It then defines a global variable for each register that is not saved on a per-function basis. For example, if register 10 is a standard register, the program declares `int R10`, so subsequent references to `R10` refer to this location.

[4]Some form of separate compilations is desirable and will be added shortly.

The rest of the file contain a sequence of function definitions derived from each function's schema. A derived function's name is formed by prepending the function's name with a distinct prefix (`__ae_`). Each function defines a local variable corresponding to each register that is saved and restored on a function-by-function basis. The body of the function is composed of:

- `ISSUE_INST (len)`. The macro `ISSUE_INST` indicates that a new instruction (whose length in bytes is `len`) is executed at the current PC.
- `READ_MEMORY (addr)`, `WRITE_MEMORY (addr)`. These macros indicate that the memory location with address `addr` is read or modified.
- Annotations. Currently AE marks the control-flow edges that enter, iterate, and exit a loop with the annotations `LOOP_ENTRY`, `LOOP_BACK`, and `LOOP_EXIT`, respectively. This set of annotations can easily be expanded.
- Function calls. Calls in which the callee is known by `gcc` directly invoke the appropriate schema routine. For example, a call on `foo` involves the translated schema routine `__ae_foo`. Calls in which the callee is unknown require a signification event during execution to record the address of the callee. The translated schema uses a table of function addresses to map the address from the trace file into the appropriate schema routine.
- Branches. Unconditional jumps translate directly into unconditional jumps in the schema program. Conditional branches must read the trace file to find which target block was reached and then jump to the appropriate point in the schema program.
- Computation. The translated schema directly executes `compute_defn` to compute values. These computations use the "registers" in the schema program. `unknown_defn`'s correspond to impossible instructions and read their value from the trace file.

The interface to AE is the routine `ae_recreate (file, func)`. `file` is the name of the trace file containing the significant events. `func` is a function that the regeneration program invokes on each event in the trace. `func` is passed two or three arguments. The first is a value indicating the type of

event. The second is the address of the event. The optional third argument is auxiliary data. `ae_recreate` returns when the trace is completely produced.

The regeneration program does not dynamically allocate memory since it does not record values. Its only memory usage is the program stack. The maximum depth (in frames) of this stack is the same as the original program.

It is not necessary to profile all pieces of a program (for example system libraries). AE produces dummy routines for all functions in the executable file for which it does not find a schema. However, it is a serious mistake to omit a schema file for a file compiled with AE profiling, since the program will produce and record significant events that confuse the recreation program.

5 Applications

AE has been used in two applications to date. The first is memory-system analysis, which feeds the sequence of memory addresses referenced by a program to a simulator that predicts the performance of a memory system. AE's main advantage in this application is the large compression of the trace file. Previously, memory address traces were compressed by computing the difference between successively-accessed addresses—which uncovers underlying regularity in access patterns—and feeding the result to a compression program such as the Unix `compress` utility [9]. This process typically reduces the volume of a full address trace by a factor of 10 and an instruction trace by a factor of 200. Section 6 demonstrates that AE can greatly exceed this compression. The additional compaction enables longer (and hence more realistic) traces to be analyzed, stored, and exchanged.

The other application is a parallelism analyzer, which examines the executions of program loops to find the possibility of parallel overlap [7]. This system, called `pp`, uses a full address trace in addition to events to indicate the beginning, end, and iteration of a loop. `pp` records the last read and modification of each memory location. It uses this information to detect loop-carried data dependences. These conflicts partially serialize a loop's execution and necessitate delays in initiating iterations. `pp` computes and reports the potential speed improvement offered by parallel execution of the program as well as data about the characteristics of loops and data dependences.

In this application, AE's main advantage is again the large compression of trace files. However, another advantage is the ease of identifying

higher-level program constructs in the trace. Some features, such as loops, can be found by analyzing an assembly language program. However, other features, such as the type of object referenced by a memory access, cannot be recovered from an assembly program. Because `gcc` can relate each RTL instruction to a source-level command, AE could distinguish conflicts over variables, arrays and pointer-linked structures.

6 Performance

Three aspects of AE's performance are important: the additional time required to trace the original program, the size of the trace file, and the time required to regenerate a full address trace. The other costs, such as additional overhead in `gcc` and the size of the schema files, are insignificant. All measurements were run on a DECstation 3100 (a 14 MIPS computer that contains a MIPS R2000 processor) with 24 megabytes of main memory and a local disk. All programs were compiled with optimization.

The test programs are:

Program	Application	Lines of Code
pdp	Connectionism simulator	4,400
compress	Unix utility	1,510
polyd	Polydominoes game	528
sgefa	Gaussian elimination	1,218

The next table shows their execution times with and without profiling. Time was measured with the `time` command. "Program time" is the execution time for the unprofiled version of a program. "Profile time" is the difference between the execution time of the profiled and unprofiled programs.

Program	Program Time (User + System)	Profile Time (User + System)	Profile/ Program Time
pdp	1.9u + 0s	1.3u + 5.7s	3.7
compress	1.3u + 0.4s	1.2u + 3.1s	2.5
polyd	3.5u + 0.0s	2.7u + 10.6s	3.8
sgefa	1.0u + 1.6s	0.6u + 1.5s	0.8

I chose the programs' input data to produce reasonable duration executions whose trace files did not exceed the capacity of my local disk. As can be

seen from the table, the overhead of profiling ranged from 0.8–3.8 times the cost of executing the program. Most of the cost is system overhead to write trace files to disk. This cost reflects the design of the Unix file system and the slow speed of the local disk. Only considering user time, the overhead of profiling is always less than the cost of running a program.

The following table shows the characteristics of the trace files. `ae.out` is the significant event trace file written by the traced program. `ae.out.Z` is the compressed version of this file.[5] The next two columns list the number of instructions and memory references in the regenerated trace.

Program	`ae.out` (bytes)	`ae.out.Z` (bytes)	# Inst.	# Refs.
pdp	10,117,340	697,790	19,777,774	6,622,419
compress	5,222,138	1,964,649	15,840,343	4,950,641
polyd	16,692,484	2,198,161	42,510,559	9,318,059
sgefa	1,590,518	137,871	12,651,297	3,464,586

The following table contains the ratio of the size of the full trace (at 5 bytes per address) to the size of the significant event file, uncompress and compressed.

Program	Size trace / Size Event	
	`ae.out`	`ae.out.Z`
pdp	9.8	141.7
compress	19.9	52.9
polyd	15.5	117.9
sgefa	50.7	584.0

The significant event file is 10–50 times smaller than the full address trace. However, when the event file is compressed by the Unix `compress` utility, this ratio increases to 50–600 times. Of course, the full address trace can be compressed also, so these figures only show the reduction with respect to the full trace. The program `pdp` has the lowest ratio for the uncompressed file because its memory reference pattern was very irregular and had to be recorded at execution time. On the other hand, `sgefa` has the largest ratio since it traversed arrays in a regular pattern that was easily reproduced by the schema program.

[5] A future modification is to directly write a compressed trace file. This change may reduce the system overhead, at the expense of greatly increasing the user time. However, it will permit longer traces to be recorded.

The next table shows the time to regenerate the address trace. The regeneration code was compiled without optimization. This test passed the generated events and addresses to a dummy routine that contained a case statement that discriminated on the type of event and then discarded the data.

Name	Regeneration Time	Addresses/Second
pdp	103.4u + 12.9s	171,533
compress	53.9u + 5.5s	350,017
polyd	190.3u + 19.0s	247,628
sgefa	40.9u + 1.3s	381,893

The test programs generated and consumed 171,000–381,000 address per second of CPU time. By profiling a test program, we find that approximately half of the time is spent in the dummy routine that consumes the addresses, rather than in the code that regenerates them. The rate at which address are generated roughly correlates with the compression factor (and inversely with the size of `ae.out`), which means that AE can calculate addresses faster than it can read them from the trace file. Even in the worst case, AE produces address much faster than an application can consume them. Cache memory simulators generally process tens of thousands of address per second. This means that the additional time to generate address is insignificant for these programs.

The cost of regeneration cannot be directly compared with the traced program's execution cost. The program trace is incomplete (its advantage) and unusable without the additional work performed by the regeneration program. Directly instrumenting a program to collect a full trace would slow its speed below that of the regeneration program since it would perform both programs' computations.

7 Related Work

Many researchers have build systems for collecting memory address traces. An early technique examined every instruction before it executed to determine its memory accesses. Because this approach trapped each instruction and did a process context switch to the analyzer, it slows program execution by two or three orders of magnitude. A faster approach modified the VAX's microcode to record the addresses that it produced. Agarwal's ATUM system still slowed programs by a factor of 20 [1]. However, it did not require

program modifications or recompilation and could trace all processes running on a computer. Neither scheme compressed the data.

Borg's system for tracing programs on the DEC Titan inspired this effort [3]. This system uses a modified linker to insert tracing code in a compiled program. This code records the execution of each basic block, the number of instructions in the block, and the address of referenced memory locations. Currently, the system saves space by not recording the interleaving of memory references among instruction addresses. Tracing slows program execution 8–12 times and produces such a large volume of data that analysis is generally conducted on-line by interrupting the program and running an analysis process. However, the system can trace all processes and the kernel running on a machine.

Independently, but slightly earlier than this work, Eggers, Keppel, Koldinger, and Levy developed a system that uses many of the same ideas as AE [4]. Their system, MPtrace, produces address traces of multithreaded parallel programs. It reduces the volume of trace data by recording control flow among superblocks (a basic block with a single entry, but multiple exits) and by recognizing special cases of load and stores (e.g., a load off the frame pointer). Their system reads and modifies the assembly code produced by the Sequent C and Fortran compilers.

From published reports, MPtrace does not have schemas, although it regenerates some address from a smaller set of data collected during execution. The special cases recognized by their system does not include all cases recognized by AE. For example, superblocks record events for unconditional jumps that AE infers from the schema file. In addition, it is unclear whether the special cases that their system recognizes includes regular array access such as the one in the example. Because MPtrace operates on an assembly language, it cannot identifying higher-level construction (see Section 5). The execution time overhead of their system is worse than AE. Programs executed 1.6–2.3 times slower without file IO and 10 times slower if the overhead to write the trace file is included. They do not report on the compression of the data file.

8 Status

AE currently runs on the MIPS R2000/R3000 and SPARC processors. AE is available by anonymous ftp from `primost.cs.wisc.edu` (128.105.8.17) in the file `~ftp/pub/ae.tar.Z`. This file is a compressed tar file containing

additions to `gcc`. AE also requires a copy of `gcc`, which is available from many places including `prep.ai.mit.edu`. AE is copyrighted and distributed under the terms of the GNU General Public License.

9 Conclusion

Abstract Execution is a powerful and general technique for tracing the execution of programs. By reducing the quantity of data collected during a program's execution, it reduces the effect of profiling on the program's behavior and the volume of data that must be stored. Missing information can easily be regenerated by reexecuting small portions of the program using the traced events as a guide. This division shifts some of the profiling costs to the program that utilizes the data, which typically is an expensive computation that is not adversely affected by a small amount of additional work.

AE is a simple, portable system for tracing events in C program. It uses Abstract Execution to greatly reduce the cost of profiling these programs in detail.

Acknowledgments

Mark Hill, David Keppel, Rick Kessler, Robert Netzer, and Ben Zorn read and commented on drafts of this paper. Their help is greatly appreciated.

References

[1] Annat Agarwal, Richard L. Sites, and Mark Horwitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 119–127, June 1986.

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

[3] Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall. Long Address Traces from RISC Machines: Generation and Analysis. Technical Report 89/14, Digital Equipment Corporation, Western Research Laboratory, September 1989.

[4] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. Technical Report 89-09-18, Department of Computer Science, University of Washington, September 1989. Accepted for SIGMETRICS 1990.

[5] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An Execution Profiler for Modular Programs. *Software Practice & Experience*, 13:671–685, 1983.

[6] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 35–46, June 1988.

[7] James R. Larus. Estimating the Potential Parallelism in Programs. Submitted to ICPP '90, January 1990.

[8] MIPS Computer Systems, Inc. *RISCompiler Languages Programmer's Guide*, December 1988.

[9] A. Dain Samples. Mache: No-Loss Trace Compaction. In *Proceedings of the International Conference on Measurment and Modeling of Computer Systems*, pages 89–97, May 1989.

[10] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[11] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, September 1989.

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

Appendix

A Details of MIPS R2000 Implementation

This section contains details of the implementation of AE on the MIPS R2000/R3000 processor. It can be read either as a more detailed description of AE or as hints about how to make AE run on other machines.

Programs compiled with the `-AE` flag to `gcc` record events in a 1 megabyte buffer that resides on the program stack (see Figure 8). This location places the buffer above most of the program's address space and so lessens the disruption caused by carving a large chunk out of the address space. It also permits a quick test for buffer overflow. The code for the first first event in a basic block checks whether the event buffer has room for all events from the block (actually 100 bytes since at this point, `gcc` does not know how many events are in the block). In AE, register 23 is no longer a general-purpose registers, but contains a pointer to the next free location in the buffer. The code for the first event is:

```
        .set noat                       # Event w/ check
        addu R1, R23, 29672
        bgtz R1, L5
        .set at
        jal ae_flush_buffer
L5:
        usw     R29, 0(R23)             # Save value in R29
        addiu   R23, R23, 4             # End Event
```

The stack on this machine grows down from just below 0x80000000, which is also a negative number, unlike the valid program addresses. The trace buffer begins 28,672 bytes below this address. This offset was chosen to allow space for the environment and arguments to `main` above the buffer and to enable the offset to fit in the signed 16-bit field in the `addu` instruction. If $R23 + 28,672 + 100$ is negative, the buffer contains fewer than 100 free locations and the routine `ae_flush_buffer` empties it. This procedure is carefully written so it preserves values in all registers, not just those normally saved by a function call. The last two instructions in the event store the contents of register `R29` into the buffer and increment the buffer pointer. The store is an unaligned access since the buffer contains bytes as well as

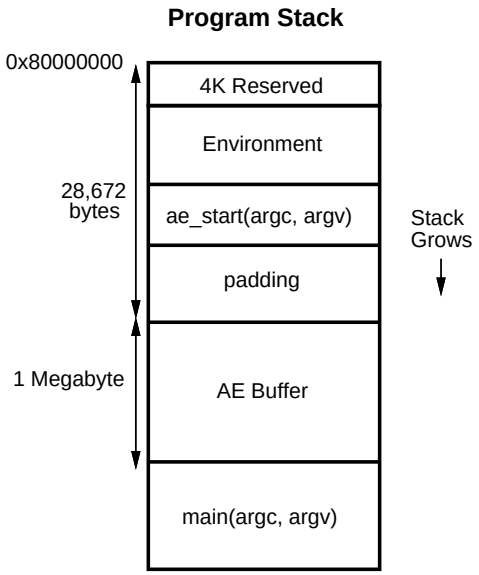

Figure 8: Organization of MIPS stack. The stack grows down from address 0x8000000. The stack frame for the startup routine (`ae_start`) begins directly below the environment. This routine allocates a 1 megabyte buffer on the stack and then invokes the user program's `main` routine.

words. Other events in the block, which do not check for free space, require only these two instructions.

The MIPS assembler hides many details of the underlying architecture by providing pseudo instructions that expand into one or more machine instructions and by performing instruction scheduling. On one hand, this approach reduces the complexity of producing code for the machine and simplifies AE. On the other hand, the assembly code produced by `gcc` differs from the executed code. AE estimates the size of the code sequence produced for pseudo instructions. However, it does not attempt to predict the instruction scheduling.

The code for `ae_start`, `ae_flush_buffer`, and `ae_terminate` is similar to the profiling code for `prof` or `gprof` and is contained in the file `aecrt0.o`.

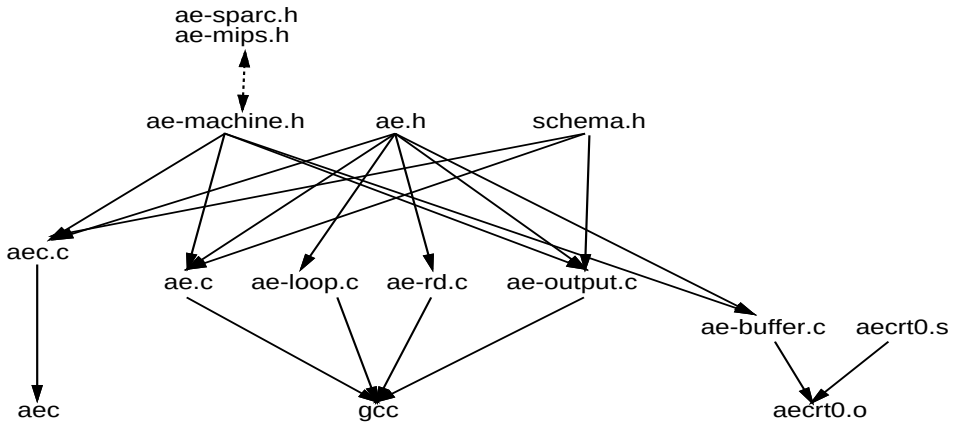

Figure 9: Organization of files in AE and AEC.

B Parameters for AE

This section describes the configuration file for AE (`ae-machine.h`). This file defines machine-specific quantities for AE and `aec` (see Figure 9. It is similar to the `gcc`'s machine description files.

- `AE_BUFFER_REG` If defined, it is a string containing the name of the register that AE uses to point to the next free byte in its buffer. If `AE_BUFFER_VAR` is defined, this quantity should be undefined.
- `AE_BUFFER_VAR` If defined, it is a string containing the name of the variable that AE uses to hold a pointer to the next free byte in its buffer. If `AE_BUFFER_REG` is defined, this quantity should be undefined.
- `MAKE_AE_BUFFER_POINTER()` An expression that produces RTL corresponding to `AE_BUFFER_REG` or `AE_BUFFER_VAR`, as appropriate.
- `AE_BUFFER_BOUND_REG` If defined, it is a string containing the name of the register that AE uses to point to the end its buffer. This quantity should be undefined if `AE_BUFFER_BOUND_VAR` is defined,
- `AE_BUFFER_BOUND_VAR` If defined, it is a string containing the name of the variable that AE uses to hold a pointer to the end of its buffer. If `AE_BUFFER_BOUND_REG` is defined, this quantity should be undefined.

- MAKE_AE_BOUND_POINTER() An expression that returns RTL form corresponding to AE_BUFFER_BOUND_REG or AE_BUFFER_BOUND_VAR, as appropriate.
- The next two values are used instead of a bound register or variable. If they are defined, AE_BUFFER_BOUND_REG and AE_BUFFER_BOUND_VAR must be undefined.
  - STACK_TOP The address of the first location of the C stack.
  - AE_BUFFER_STACK_OFFSET The offset from STACK_TOP to the end of AE's buffer.
- AE_BUFFER_SIZE The size of AE's buffer in bytes.
- AE_START_FRAME_SIZE The size of the function AE_START's stack frame.
- SP_REG A string containing the name of the stack pointer register.
- MAX_PEEP One plus the maximum number of instructions combined by the peephole optimizer.
- REGISTER_DEFINED_IN_CALL(REGNO) An expression that producing a non-zero result if REGNO is the number of a register that may be defined upon function entry.
- ASM_COMMENT_CHAR The character that precedes assembler comments.
- ASM_DIRECTIVE_CHAR The character that preceds assembler directives.
- JUMP_DELAY_SLOTS The number of instructions following an unconditional jump instruction that are executed. If the machine does not have delayed branches or the assembler hides the delay, this value is undefined.
- CJUMP_DELAY_SLOTS The number of instructions following a conditional jump instruction that are executed. If the machine does not have delayed branches or the assembler hides the delay, this value is undefined.
- CALL_DELAY_SLOTS The number of instructions following a subroutine call instruction that are executed. If the machine does not have delayed calls or the assembler hides the delay, this value is undefined.
- STD_ASM_INSN_LENGTH The length of most instructions in bytes.

- `ASM_INSN_SIZE_EXCEPTIONS` If defined, it is a list of that are longer than normal because they are pseudo instructions that expand into one or more instructions. The list is a sequence of instruction-size pairs, ordered by the instruction name.

- `BRANCH_IS_ANNULED(ASM_INSN)` An expression that produce a non-zero result if the `ASM_INSN` is a delayed branch that does not execute (annuls) the following instructions.

- `ASM_INSN_IS_CALL (ASM_INSN)` An expression that produces a non-zero results if the `ASM_INSN` is a subroutine call.

- `SCHEMA_PROLOGUE(RECORD_REG)` Invoked to produce a schema corresponding to the code generated for the function prologue. Element N is `RECORD_REG` is non-zero if register N's value should be recorded upon function entry.

- `SCHEMA_EPILOGUE(RECORD_REG)` Invoked to produce a schema corresponding to the code generated for the function epilogue.

- `GENERATE_SPACE_CHECK(COMMENT, SIZE)` Invoked to produce assembly code to check if `SIZE` bytes remain in AE's buffer.

- `GENERATE_EVENT(VALUE)` Invoked to produce assembly code to save `VALUE`.

- `GENERATE_SHORT_EVENT(VALUE, BYTES)` Invoked to produce assembly code to save `VALUE`, which fits in `BYTES` bytes.

- `GENERATE_ADDRESS_EVENT(ADDRESS, BASE, OFFSET)` Invoked to produce assembly code to save `ADDRESS`, which is composed of `BASE` and `OFFSET`.

- `AE_START_ASM` The symbol in the `#ifdef` surrounding assembly code for `ae_start`.

- `AE_FLUSH_BUFFER_ASM` The symbol in the `#ifdef` surrounding assembly code for `ae_flush_buffer`.

- `ECOFF_AOUT` Define if the machine uses MIPS's ECOFF format for `a.out` files.

- `BSD_AOUT` Define if the machine uses BSD format for `a.out` files.

- `PC_OFFSET_AFTER_CALL` Number of bytes beyond call instruction that a function returns.
- `REG_LOCAL_TO_FUNCTION(N)` An expression that produces a non-zero result if register N is local to a function.
- `INITIALIZE_REGISTERS()` Invoked to produce code to initialize registers before the generation program executes.