ATOM

Reference Manual

December 1993

Digital Equipment Corporation
Maynard, Massachusetts

Contents

List of Figures

1 Introduction

Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical pieces of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing or to provide input for profile-driven optimizations.

Over the past decade many tools for different machines and applications have been developed. We briefly describe these tools by grouping them into three classes. The first class consists of basic block counting tools like Pixie[8], Epoxie[12] and QPT[7] that count the number of times each basic block is executed. The second class of tools are address tracing tools that generate data and instruction traces. Pixie and QPT can generate traces and can communicate to analysis routines through inter-process communication. Mahler was used for generating address traces on Titans[3]. This system used a more efficient method for communicating information than Pixie but required operating system modifications. MPTRACE [4] is also similar to Pixie but it collects traces for multiprocessors by instrumenting assembly code. ATUM [1] generates address traces by modifying microcode. ATUM performs trace compression and saves the trace in a file that is analyzed offline. The third class of tools are based on simulation. Tango Lite[6] is a multiprocessor simulator that instruments assembly language code. PROTEUS[2] is also a multiprocessor simulator but instrumentation is done by the compiler. Shade[5] is an efficient instruction-set simulator that can generate selective instruction traces.

These existing tools have several drawbacks.

First, most tools are designed to compute fixed data. It is not easy for a user to modify such tools to compute more or less data. A tool computing insufficient information becomes useless for the user.

Second, most address tracing tools, except Shade, compute detailed address information. However, too much computed information renders the tool inefficient for the user. For example, a user interested in branch behavior has to sift through the entire instruction trace, even though only conditional branches needed to be examined. The instruction and address traces are extremely large even for small programs and typically run into gigabytes.

Third, tools based on instruction-level simulation add large overheads to the processing time. Several techniques have been used to make the simulation faster, such as in the Shade system, but simulation nevertheless makes the programs run many times slower.

Finally, there are two common methods of communicating information from application program to user's analysis program: files on disk and some form of inter-process communication. Both are expensive. The large size of address traces further aggravates this problem.

In this paper we describe ATOM, a flexible and efficient tool that solves these problems and presents a simple way to build customized program analysis tools. ATOM provides a single framework under which a diverse set of tools ranging from basic block counting to cache and instruction-level simulators can be built. ATOM provides the machinery to instrument the program and allows the tool designer to specify the instrumentation and the analysis details. The method of communication between the application program and the analysis routines is a simple procedure call. ATOM provides precise information and uses no simulation. We first describe the design of ATOM and show how we overcome the drawbacks of current systems. We show through a real example how such tools can be built. Finally, we describe the implementation of ATOM and give performance numbers.

2 Design of ATOM

Program analysis tools are needed for problems ranging from basic block counting to instruction and data cache simulation. As there are a many interesting problems to be solved, building specific tools from scratch is clearly not a viable solution. Generating information like address traces is inefficient if not all the information is needed. The design of ATOM is based on the observation that although problems such as basic block counting and cache simulation appear vastly different, all such tools can be built by instrumenting a program at a few selected points. ATOM separates the common part in all problems from the problem specific part by providing machinery for instrumentation and object-code manipulation, and allowing the tool designer to specify what points in the program are to be instrumented. The designer also provides code for analysis routines. The instrumentation mechanism provided by ATOM needs to be dealt with only once and is shared by all tools built with ATOM. The designer can concentrate on the analysis related to the problem. Also, tools built with this approach compute only what the tool designer asked for.

A program is viewed as a linear collection of procedures, procedures as a collection of basic blocks, and basic blocks as a collection of instructions. Most tools want to instrument a program at a procedure, basic block or an instruction boundary. Basic block counting tools instrument the beginning of each basic block, data cache simulators instrument each load and store instruction, branch prediction analyzers instrument each conditional branch instruction, and instruction translation buffer instrument instructions that cross instruction page boundaries.

Therefore, ATOM allows a procedure call to an analysis routine to be inserted before or after a program, a procedure, a basic block, or an instruction.

ATOM uses a simple procedure call as the mechanism of transferring information directly from the application program to the analysis routines. ATOM avoids using the file system or any form of inter-process communication as they are expensive. To reduce the communication to a simple procedure call, the application program and the analysis routines have to run in the same address space. ATOM partitions the name space and lays the application and analysis routines in the executable such that they do not interfere with each other's execution. More importantly, the analysis routine is always presented with the information (data and text addresses) about the application program as if it were executing uninstrumented. Section 4 describes how ATOM computes the precise information.

ATOM is independent of any compiler and language as it operates on object modules that make up the complete program. It uses the OM[9] link-time technology to conveniently modify an executable.

3 Building Customized Tools: An Example

In this section we show how to build a simple tool that counts how many times each conditional branch in the program is taken and how many times it is not taken. The final results are written to a file. To build a tool with ATOM, the designer of the tool has to provide the instrumentation and the analysis routines. Through the instrumentation routines, the designer specifies where the application program needs to be instrumented and what procedure calls to the analysis routines are to be made. The designer provides code for procedures in the analysis routines. The instrumentation and analysis routines are written in C, and are shown in Figure 1.

Defining Instrumentation Routines

The instrumentation routines contain an `Instrument` procedure which describes where the application program needs to be instrumented and which analysis routines are to be called. The `AddCallProto` primitive defines the prototypes of procedures in the analysis routines that will be called from the application program. In our example, prototypes of four analysis procedures `OpenFile`, `CondBranch`, `PrintBranch`, and `CloseFile` are defined. Besides the standard C data types as arguments, ATOM uses additional types such as `REGV` and `VALUE`. If the argument type is `REGV` the actual argument is a register number and the contents of

Instrumentation Routines

```
#include <instrument.h>

Instrument(){
    Proc *p;  Block *b;  Inst *i;
    int nbranch = 0;

   AddCallProto("OpenFile(int)");
   AddCallProto("CondBranch(int,VALUE)");
   AddCallProto("PrintBranch(int,long)");
   AddCallProto("CloseFile()");

   for (p = GetFirstProc(); p != NULL; p = GetNextProc(p)){
      for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b)){
         inst= GetLastInst(b);
         if  (IsInstType(inst,  InstTypeCondBr)){
           AddCallInst(inst, InstBefore, "CondBranch", nbranch, BrCondValue);
           AddCallProgram(ProgramAfter, "PrintBranch", nbranch, InstPC(inst));
           nbranch++;
         }
      }
    }

    AddCallProgram(ProgramBefore, "OpenFile", nbranch);
    AddCallProgram(ProgramAfter, "CloseFile");
}
```

Analysis Routines

```
#include <stdio.h>
File *file;
struct BrInfo
  long taken;
  long notTaken;
} *bstats;
```

```
void OpenFile(int n){
  bstats = (struct BrInfo *)
       malloc (n * sizeof(struct BrInfo));
  file = fopen("btaken.out", "w");
  fprintf(file, "PC \t Taken \t Not Taken\n");
}
```

```
void CloseFile(){
   fclose(file);
}
```

```
void CondBranch(int n, long taken){
  if (taken)
    bstats[n].taken++;
  else
    bstats[n].notTaken++;
}
```

```
void PrintBranch(int n, long pc){
  fprintf(file, "0x%lx \t %d \t %d\n",
     pc, bstats[n].taken, bstats[n].notTaken);
}
```

Figure 1: Building Tool for Branch Prediction Analysis

the specified register are passed. For the `VALUE` argument type, the actual arguments may be `BrCondValue`. `BrCondValue` is used for conditional branches and passes zero if branch condition will evaluate to a false and non-zero value if the condition will evaluate to true. `CondBranch` uses the argument type `VALUE`.

ATOM allows the designer to traverse the whole program using a high level programming model of a program consisting of procedures, basic blocks and instructions. `GetFirstProc` returns the first procedure, and `GetNextProc` returns the next procedure. The outermost `for` loop traverses the program a procedure at a time. Inside each procedure `GetFirstBlock` returns the first basic block and `GetNextBlock` returns the next basic block. Using these primitives the inner loop traverses all the basic blocks of a procedure.

In this example, we are interested only in conditional branch instructions. We find the last instruction in the basic block using the `GetLastInst` primitive and check if it is a conditional branch using the `IsInstType` primitive. All other instructions are ignored. With the `AddCallInst` primitive, a call to the analysis procedure `CondBranch` is added at the conditional branch instruction. The `InstBefore` argument specifies that the call is to be made before the instruction is executed. The two arguments to be passed to `CondBranch` are the linear number of the branch and its condition value. The condition value determines whether the branch will be taken.

The `AddCallProgram` is used to insert calls before (ProgramBefore) the application program starts executing and after (ProgramAfter) the application program finishes executing. These calls are generally used to initialize analysis routine data and print results at the end respectively. A call to `OpenFile` before the application program starts executing creates the branch statistics array and opens the output file. We insert calls for each branch to print its pc and its accumulated count at the end. Note that these calls are made only once for each conditional branch after the application program has finished executing. Another method would be to store the PC of each branch in an array and pass the array at the end to be printed along with the counts. ATOM allows passing of arrays as arguments. Finally, the `CloseFile` procedure is executed which closes the output file. If more than one procedure is to be called at a point, the calls are made in the order in which they were added.

Defining Analysis Routines

The analysis routines contain code for all procedures needed to do the analysis. The `OpenFile` uses its argument containing the number of branches to allocate the branch statistics array. It also opens a file to print results. The `CondBranch` routine increments the branch taken or branch not taken counters for the specified branch by examining the condition value argument. `PrintBranch` prints the PC of the branch, the number of times the branch is taken and number of times it is not taken. `CloseFile` closes the output file.

Instrumented Program

To find the branch statistics, ATOM is given three files as input: the fully linked application program in object-module format, the instrumentation routines, and the analysis routines. ATOM builds the customized tool which takes as input the application program and outputs the instrumented program executable. When this instrumented program is executed the branch statistics are produced as a side effect of the normal program execution.

4 Implementation of ATOM

ATOM is built using OM[9, 10, 11], a link-time code modification system. OM takes as input a collection of object files and libraries that make up a complete program, builds a symbolic intermediate representation, applies instrumentation and optimizations to the intermediate representation, and finally outputs an executable. ATOM currently produces non-shared executables; work is in progress for writing shared executables.

ATOM works in two steps. This is illustrated in Figure 2.

In the first step, the tool designer's instrumenting routines are linked with OM to produce a customized tool. The instrumenting routines specify the points in the application program that are to be instrumented and the analysis routines to be called.

In the second step, ATOM uses OM to convert an object module to an intermediate representation. The customized tool builds a symbolic intermediate representation of the program and the analysis routines. The designer's instrumenting routines for the customized tool annotate the application program's representation to specify the procedure calls to be made and the arguments to be passed. The application program code is modified as specified by the designer, and the executable file containing the instrumented application program and analysis routines is produced. ATOM takes care that precise information about the application program is presented to the analysis routines.

To reduce the communication between the application program and analysis routines to a simple procedure call, the application program and analysis routines run in a single address

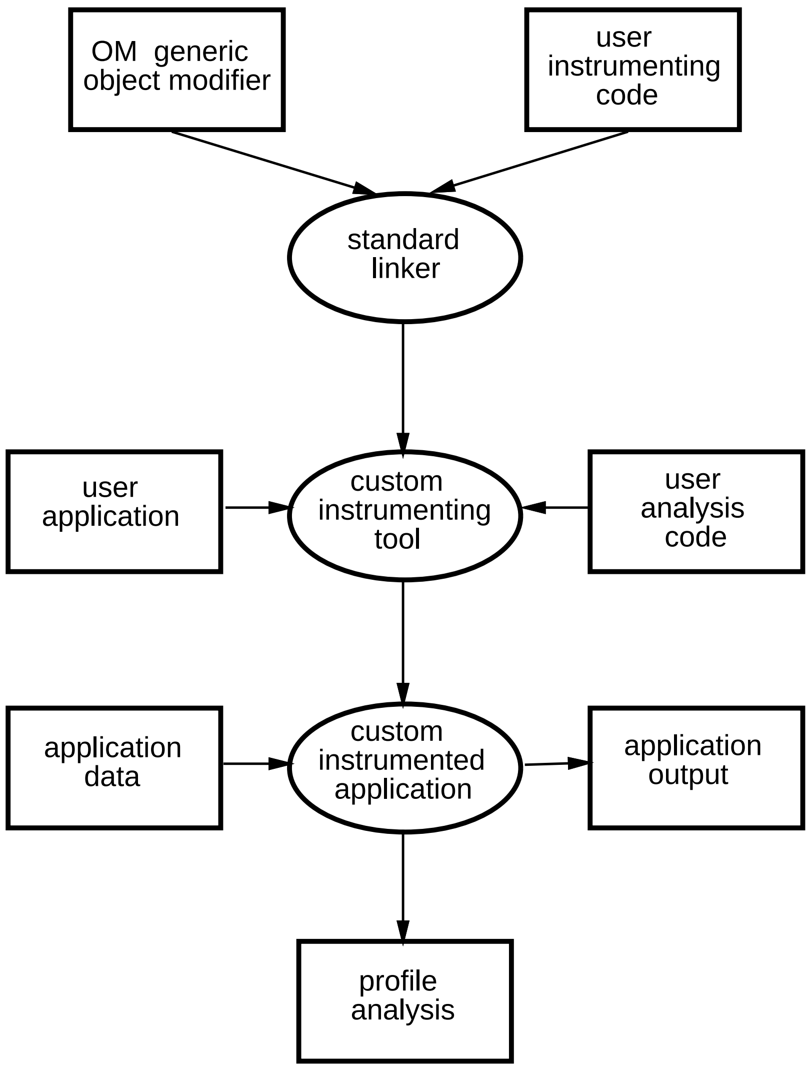

Figure 2: The ATOM Process

space. Analysis routines do not share any procedures or data with the application program; if both the application program and the analysis routines use the same library procedure, like `printf`, there are two copies of `printf` in the final executable, one in the application program and the other in the analysis routines.

Keeping Pristine Behavior

The main goal of ATOM is to avoid perturbing the addresses in the application program. Therefore, the analysis routines are put in the space between the application program's text and data segments.

The data sections of the application program are not moved, so the data addresses in the application program are unchanged. The initialized and uninitialized data of analysis routines is put in the space between the application program's text and data segments. As in an executable all initialized data is before all uninitialized data, the uninitialized data of the analysis routines is converted to initialized data by initializing it with zero. The start of the stack and heap[1] are unchanged, so all stack and heap addresses are same as before.

The text addresses of the application program have changed because of the addition of instrumented code. However, we statically know the map from the new to original addresses. If an analysis routine asks for the PC of an instruction in the application program, the original PC is simply supplied. This works well for most of the tools.

However, if the address of a procedure in the application program is taken, its address may exist in a register. If the analysis routine asks for the contents of such a register, the value supplied may not be the original address. To handle this problem, we relocate the program with original addresses, so all registers contain original addresses. Direct procedure calls would work as before because we can determine the destination address statically. For indirect call where the destination depends on the contents of a register, the mapping is done dynamically. This mechanism is only necessary if the tool requires register contents to be passed to analysis routines. Pixie uses this mechanism for translating calls.

Reducing Procedure Call Overhead

The communication between the application program and the analysis routines is through procedure calls. The tool designer specifies the procedures in analysis routines to be called at certain points in the application program. All the temporary registers need to be saved before the call

[1] On the Alpha AXP under OSF stack begins at start of text segment and grows towards low memory, and heap starts at end of uninitialized data and grows towards high memory.

Program Analysis Tool	*Tool Description*	*Time taken to instrument entire SPEC92 suite*
Data Translation Buffer	model 32 entry fully associative dtb	146.41 secs
Data Cache	model direct mapped 8k byte cache	133.50 secs
Dynamic Instruction	computes dynamic instruction counts	134.52 secs
Branch Prediction Analysis	computes branch predictions rates	119.08 secs
Inline Analysis	finds potential inlining candidates	150.83 secs

Figure 3: Time taken by ATOM to instrument 20 SPEC92 benchmark programs

to the analysis routine and restored on return from the analysis routines. This is necessary to preserve the execution state of the application program.

The number of registers that need to be saved and restored can be reduced in the following two ways. By computing summary information of the analysis routines, we can determine all the registers that may be used when the control reaches a particular analysis procedure. Only these registers need to be saved and restored.

The number of registers that need to be saved may be further reduced by computing live registers in the application program. OM can do interprocedural live variable analysis[9] and compute all registers live at a point. Only the live registers need to be saved and restored to preserve the state of the program execution. Optimizations such as inlining further reduce the overhead of procedure calls.

5 Performance

To find how well ATOM performs, two measurements are interesting: One is how long it takes ATOM to produce an instrumented program; the other is how long it takes for the instrumented program to execute. All measurements are done on Digital Alpha AXP 3000 Model 400 with 128 Mb memory.

To find how long it takes ATOM to build instrumented programs, we ran ATOM over the SPEC92 benchmark suite consisting of 20 programs to build 5 tools: data translation buffer modeling, data cache modeling, dynamic instruction count analysis, branch prediction analysis, and inline analysis. The time taken for each tool to build the instrumented programs for the 20 program SPEC92 suite is shown in Figure 3. The average time spent to instrument each benchmark is about 8 seconds.

Program Analysis Tool	*Instrumented Program Execution Time*
Data Translation Buffer	19.55 times
Data Cache	14.12 times
Dynamic Instruction	5.80 times
Branch Prediction Analysis	5.10 times
Inline Analysis	0.24 times

All ratios with respect to base application execution time

Figure 4: Analysis of Instrumented Program Execution Time

The execution time of the instrumented program is the sum of the execution time of the uninstrumented application program, the procedure call overhead, and the time spent in the analysis routines. This total time represents the time needed for the user to get the final answers. Since ATOM does communication through procedure calls rather than through file I/O or any form of interprocess communication, the communication time is limited to the procedure call overhead. The time spent in analysis routines is equivalent to the postprocessing time required by other systems. Many systems do the processing of collected data offline and do not include those numbers as part of data collecting statistics. The uninstrumented program's execution time is used as the base to compute the instrumented program's execution time. Figure 4 shows the execution time of the instrumented program for each tool. The procedure call overhead is not constant because ATOM uses the summary information to find the necessary registers to save. Moreover, since Inline Analysis only instruments procedure call sites, the total overhead is much less than Data Translation Buffer tool that instruments each memory reference and simulates a fully associative translation buffer. We expect the procedure overhead to decrease further when we implement live register analysis and inlining.

6 Conclusion

ATOM provides a simple and powerful mechanism through which a wide variety of tools can be built under a single framework. ATOM separates the object-module modification details from the transformations to be performed. A tool designer concentrates only on what information has to be collected and how to process it, while ATOM provides the machinery for cumbersome instrumentation details. Tools can be built with few pages of code that compute only what the designer desired. The information is communicated from the application program to analysis routines through simple fast procedure calls; thus, the communication overhead is small. ATOM has already been used to build a wide variety of tools to solve hardware and software problems. We hope ATOM will continue to be an effective platform for studies in software and architectural design.

References

[1] Anant Agarwal, Richard Sites, and Mark Horwitz *ATUM: A New Technique for Capturing Address Traces Using Microcode*, Proceedings of the 13th International Symposium on Computer Architecture, June 1986.

[2] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl, *PROTEUS: A High-Performance Parallel-Architecture Simulator*, MIT/LCS/TR-516, MIT, 1991.

[3] Anita Borg, R.E. Kessler, Georgia Lazana, and David Wall, *Long Address Traces from RISC Machines: Generation and Analysis*, Proceedings of the 17th Annual Symposium on Computer Architecture, May 1990, also available as WRL RR 89/14, Sep 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*, SIGMETRICS Conference on Measurement and Modeling of Computer Systems, vol 8, no 1, May 1990.

[5] Robert F. Cmelik and David Keppel, *Shade: A Fast Instruction-Set Simulator for Execution Profiling*, Technical Report UWCSE 93-06-06, University of Washington.

[6] Stephen R. Goldschmidt and John L. Hennessy, *The Accuracy of Trace-Driven Simulations of Multiprocessors*, CSL-TR-92-546, Computer Systems Laboratory, Stanford University, September 1992.

[7] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. University of Wisconsin Computer Sciences Tech. Report 1083, March 1992.

[8] MIPS Computer Systems, Inc. *Language Programmers's Guide*, 1986.

[9] Amitabh Srivastava and David W. Wall, *A Practical System for Intermodule Code Optimization at Link-Time*, Journal of Programming Language, vol 1, March 1993, pp 1-18. Also available as WRL RR 92/6, December 1992.

[10] Amitabh Srivastava and David W. Wall, *Link-Time Optimization of Address Calculation on a 64-bit Architecture*, WRL TN-35, June 1993, submitted to PLDI '94.

[11] Amitabh Srivastava. *Unreachable procedures in object-oriented programming*, to appear in ACM LOPLAS Vol 1, #4. Also available as WRL RR 93/4, August 1993.

[12] David W. Wall. Systems for late code modification. In Robert Giegerich and Susan L. Graham, eds, Code Generation - Concepts, Tools, Techniques, pp. 275-293, Springer-Verlag, 1992. Also available as WRL RR 92/3, May 1992.

APPENDIX A: Getting Started

The application program, instrumentation file and analysis file are needed to get started. As a standard convention we suffix the instrumentation file with **.inst.c** and analysis file with **.anal.c**. Let us use the branch prediction tool illustrated in Figure 1 as our example. The instrumentation file is contained in branch.inst.c and analysis files in branch.anal.c. Let appl.c be the the application program that we wish to instrument. The following two steps are needed to produce the instrumented the executable.

- **Preparing the Whole Program for input to ATOM**

  To produce a fully-linked program module with relocation records, use the -Wl,-r flag to cc

  cc -Wl,-r appl.c -o appl.rr

  The output linked application program is *appl.rr*. As a convention, use the suffix .rr because the linker keeps the relocation entries in the program module.

  If the application is produced using a *makefile* the -Wl,-r can be added to the final step with -o being modified to appl.rr

- **Producing the Instrumented Program Executable**

  To produce the instrumented program, atom is given the fully-linked program module, the analysis file, and the instrumentation file.

  atom appl.rr branch.inst.c branch.anal.c -o appl.branch

  ATOM takes the three files as input and produces the instrumented program *appl.branch*. When *appl.branch* is executed the desired branch prediction information is produced.

Instrumentation and Analysis Programs in multiple files?

The instrumentation and analysis files given as input to ATOM may be C files or .o modules. Compile each of the instrumentation and analysis files. Separate instrumentation files may be linked together using ld with the -r option. If instrumentation program is in files file1.inst.c, file2.inst.c and file3.inst.c, compile them separately using cc and link them as follows:

ld -r file1.inst.o file2.inst.o file3.inst.o -o file.inst.o

This mechanism may also be used to link instrumentation and analysis files with special libraries.

APPENDIX B: Using DBX with ATOM

Debugging Analysis Routines

To debug the analysis routines, pass the -g flag to ATOM. ATOM puts debugging information in the instrumented executable. Note only the analysis routines may be debugged; that is, one can put stops in analysis procedures and examine its variables but not the application procedure and variables.

atom appl.rr branch.inst.c branch.anal.c -o appl.branch -g

Debugging Instrumentation Routines

To debug the instrumentation routines pass the -dbx switch to atom:

atom appl.rr branch.inst.c branch.anal.c -o appl.branch -dbx

The ATOM driver executes the initial steps to produce a tool, and the runs dbx over it. It puts a stop at the Instrument procedure, and executes atom till the control reaches there. After dbx finishes ATOM continues its processing.

APPENDIX C: Using malloc in Analysis routines

ATOM, by default, tries to link the sbrk's of the application and analysis program so they share the same heap space. However, in this approach each starts allocating from where the previous left off, the application's heaps addresses may not be accurate. Some tools don't care about this.

To obtain pure heap addresses for the application program, the heap may be split between the application and the analysis routines. An *offset* can be specified with *-heap* option to atom. The heap of the analysis program starts at application-heap-address + offset. The user has to take care that application program's heap usage will not cross the offset specified. ATOM does no runtime checking for this.

APPENDIX D: Semantics of the ATOM User Interface

The prototypes and enumerations are defined in the file instrument.h.

1. ## Basic Program Structures

   Proc, Bloc, and Inst are three program structure types for procedures, basic blocks, and instructions.

   Proc Procedure structure type.

   Block Basic block Structure type.

   Inst Instruction Structure type.

2. ## Navigation primitives

   Proc *GetFirstProc();

   Returns a pointer to the first procedure in the program.

   Proc *GetLastProc();

   Returns a pointer to the last procedure in the program.

   Proc *GetNextProc(Proc * p);

   Returns a pointer to the procedure next to *p*. If *p* is the last procedure, a NULL pointer is returned.

   Arguments:

   p is a pointer to a procedure.

   Proc *GetPrevProc(Proc *p);

   Returns a pointer to the procedure previous to *p*. If *p* is the first procedure, a NULL pointer is returned.

   Arguments:

   p is a pointer to a procedure.

   Block *GetFirstBlock(Proc *p);

   Returns a pointer to the first basic block of procedure *p*.

   Arguments:

   p is a pointer to a procedure.

Block *GetLastBlock(Proc *p);

Returns a pointer to the last basic block of procedure *p*.

Arguments:

p is a pointer to a procedure.

Block *GetNextBlock(Block *b);

Returns a pointer to the basic block next to *b*. If *b* is the last block, a NULL pointer is returned.

Arguments:

b is a pointer to a basic block.

Block *GetPrevBlock(Block *b);

Returns a pointer to the basic block previous to *b*. If *b* is the last block, a NULL pointer is returned.

Arguments:

b is a pointer to a basic block.

Inst *GetFirstInst(Block *b);

Returns a pointer to the first instruction in basic block *b*.

Arguments:

b is a pointer to a basic block.

Inst *GetLastInst(Block *b);

Returns a pointer to the last instruction in basic block *b*.

Arguments:

b is a pointer to a basic block.

Inst *GetNextInst(Inst *i);

Returns a pointer to the instruction next to *i*. If *i* is the last instruction, a NULL pointer is returned.

Arguments:

i is a pointer to an instruction.

Inst *GetPrevInst(Inst *i);

Returns a pointer to the instruction next to *i*. If *i* is the first instruction, a NULL pointer is returned.

Arguments:

i is a pointer to an instruction.

3. Query primitives

char *GetProgramName();

Returns a string containing the name of the application program. The program name is derived from the application program name given to the atom command. If the application program was /dir1/dir2/appl.rr, the program name returned is "appl".

long GetProgramInfo(PInfoType info);

Returns a long containing the requested information of the program.

Arguments:

info is of type *PInfoType*. The valid values are:

TextStartAddress Program's text segment start address.

TextSize Program's text segment size.

InitDataStartAddress Program's initialized data segment start address.

InitDataSize Program's initialized data segment size.

UninitDataStartAddress Program's uninitialized data segment start address.

UninitDataSize Program's uninitialized data segment size.

long GetProcInfo(Proc *p, ProcInfoType info);

Returns a long containing the requested information of the procedure.

Arguments:

p is a pointer to a procedure.

info is of type *PInfoType*. The valid values are:

FrameSize Frame size of the procedure *p*.

IRegMask Saved integer register mask.

IRegOffset Saved integer register offset.

FRegMask Saved floating point register mask.

gpPrologue Byte size of the gp prologue.

gpUsed True if the procedure uses gp.

LocalOffset Offset of local variables from the virtual frame pointer.

FrameReg Frame pointer register.

PcReg Return program counter register.

PROC *GetNamedProc(char *pname);

Returns a procedure with name *pname*.

char *ProcName(Proc *p);

Returns a string containing the name of the procedure *p*. If the procedure is not found NULL is returned.

Arguments:

p is a pointer to a procedure.

char *ProcFileName(Proc *p);

Returns a string containing the file name of the procedure *p*.

Arguments:

p is a pointer to a procedure.

long ProcPC(Proc *p);

Returns a long program counter of the start of the procedure *p*. The PC of the procedure is the PC of the first instruction of the procedure.

Arguments:

p is a pointer to a procedure.

Proc *GetBlockProc(Block *b);

Returns a pointer to the procedure containing the basic block *b*.

Arguments:

b is a pointer to a basic block.

Inst *GetInstBranchTarget(Inst *i);

Returns a pointer to an instruction that is a target of the branch. An error is raised if the instruction *i* is not a branch.

Arguments:

i is a pointer to an instruction.

char *GetInstProcCalled(Inst *i);

Returns a string containing the name of the procedure being called. If the name is not known statically, a NULL pointer is returned. An error is raised if the instruction *i* is not a JSR or a BSR instruction.

Arguments:

i is a pointer to an instruction.

long BlockPC(Block *b);

Returns a long program counter of the start of basic block *b*. The PC of the basic block is the PC of the first instruction of the basic block.

Arguments:

b is a pointer to a basic block.

Block *GetInstBlock(Inst *i);

Returns a pointer to the basic block containing the instruction *i*.

Arguments:

i is a pointer to an instruction.

long InstPC(Inst *i);

Returns a long program counter of the instruction *i*.

Arguments:

i is a pointer to an instruction.

int GetInstBinary(long pc);

Returns the binary instruction at the specified *pc*.

Arguments:

pc is the program-counter of the desired instruction.

long InstLineNo(Inst *i);

Returns the source line number of the specifed instruction *i*.

Arguments:

i is a pointer to an instruction.

int GetInstInfo(Inst *i, IInfoType info);

Returns an int containing the information of the alpha instruction requested.

Arguments:

i is a pointer to an instruction.

info is of type *IInfo* field. The valid values are:

InstMemDisp Returns the sign extended 16-bit memory displacement field.

InstBrDisp Returns the sign extended 21-bit branch displacement field.

InstRA Returns the ra field of the instruction.

InstRB Returns the rb field of the instruction.

InstRC Returns the rc field of the instruction.

InstOpcode Returns the opcode field of the instruction.

InstBinary Returns the 32-bit binary coding of the instruction.

RegvType GetInstRegEnum(Inst* i, IInfoType info);

Instead of returning the 5-bit register encoded in the instruction, this primitive returns the register from the RegvType.

Arguments:

i is a pointer to an instruction.

info is of type IInfo. The valid values are InstRA, InstRB, and InstRC.

int IsInstType(Inst *i, ITypeType class);

Returns 1 if instruction is of type *class*, otherwise returns 0.

Arguments:

i is a pointer to an instruction.

class is of type *IType*. The valid values are:

InstTypeLoad Instruction is any integer or floating load.

InstTypeStore Instruction is any integer or floating store.

InstTypeJump Instruction is jump (JMP, JSR, RET, JSR_COROUTINE).

InstTypeDiv Instruction is a divide instruction.

InstTypeMul Instruction is a multiply.

InstTypeAdd Instruction is integer or floating point add or scaled add.

InstTypeSub Instruction is integer or floating point add or scaled sub.

InstTypeCondBr Instruction is integer or floating conditional branch.

InstTypeUncondBr Instruction is integer or floating unconditional branch.

IClassType GetInstClass(Inst *i);

Returns the class of the instruction *i* from the set IClassType:

The instructions classes are mutually exclusive.

ClassLoad Instruction is an integer load.

ClassFload Instruction is a floating point load.

ClassStore Instruction is an integer store.

ClassFstore Instruction is a floating point store.

ClassIbranch Instruction is an integer branch.

ClassFbranch Instruction is an floating point branch.

ClassSubroutine Instruction is an subroutine call.

ClassIarithmetic Instruction is an integer arithmetic operation except multiply and divide.

ClassImultiplyl Instruction is an integer long multiply.

ClassImultiplyq Instruction is an integer quad multiply.

ClassIlogical Instruction is an integer logical operations.

ClassIshift Instruction is an integer shift operation.

ClassIcondmove Instruction is an integer conditional move.

ClassIcompare Instruction is an integer compare instruction.

ClassFpop Instruction is an floating point operate instruction.

ClassFdivs Instruction is an floating point single precision divide.

ClassFdivd Instruction is an Floating point double precision divide.

ClassNull All other instructions such as call-pal, mb etc.

Arguments:

i is a pointer to an instruction.

4. Register Usage Bit Vectors

The registers used and updated in an instruction are marked in a bit vector structure. The structure InstRegUsageVec contains the bit vectors.

```
typedef struct inst_reg_usage{
    unsigned long ureg_bitvec[2];
    unsigned long dreg_bitvec[2];
} InstRegUsageVec;
```

```
#define UseRegBitVec(x) (((x)->ureg_bitvec))
#define DestRegBitVec(x) (((x)->dreg_bitvec))
```

The bitvectors use the enumerated type RegvType for each register number. If the instruction updates the PC such as the branch and jump instructions, PC is marked as a destination.

void GetInstRegUsage(Inst* i, InstRegUsageVec* bitvec);

Fills the InstRegUseVec *bitvec* with used and updated register information.

Arguments:

i is a pointer to an instruction.

bitvec is a pointer an structure of type InstRegUsageVec.

5. Instrumentation Points

The Instrumentation points are members of the PlaceType.

ProgramBefore The procedure call is made before the program starts executing. If the program forks child processes, currently the procedure call is made only once for parent process.

ProgramAfter The procedure call is made after the program finishes executing. If the program forks child processes, currently the procedure call is made once for each terminating process.

ProcBefore The procedure call is made before the procedure starts executing. If the procedure has multiple entry points, the procedure is executed for each of them.

ProcAfter The procedure call is made after the procedure finishes executing. If the procedure has multiple exit points, the procedure is executed for each of them. Some procedures do not return because exit was called, or a longjump was executed. If a procedure doesn't return the procedure call is not made.

BlockBefore The procedure call is made before the basic block starts executing.

BlockAfter The procedure call is made after the block finishes executing. If the block ends with an unconditional branch or jump, the procedure call is executed after the block finishes executing. However, if it ends with a jsr, and the jsr does not return because exit was called, or a longjump was executed, procedure call is not made.

InstBefore The procedure call is made before the instruction starts executing.

InstAfter The procedure call is made after the instruction finishes executing. If the instruction is an unconditional branch or jump, the procedure call is executed after the instruction finishes executing. However, if the instruction is a jsr and the procedure does not return because exit was called, or a longjump was executed, procedure call is not made.

6. Adding Procedure Calls

(a) Argument Types

char Argument is char.

int Argument is int.

long Argument is long.

char* Argument is a null terminated string.

char[n] Argument is an array of n 8-bit charcters where n is a integer.

int[n] Argument is an array of n 32-bit integers where n is a integer.

long[n] Argument is an array of n 64-bit integers where n is a integer.

REGV Argument is 64-bit contents of register specified.

VALUE Argument is 64-bit value.

(b) Registers

REG_NOTUSED, REG_0 ... REG_31, FREG_0 ... FREG_31, REG_PC, REG_CC Registers are members of the RegvType. REG_PC is a pseudo register containing the program counter, and REG_CC is the register containing the cycle counter. Use parameter type REGV in the prototypes to obtain the contents of the registers. This type was previously name reg. Some predefined registers are REG_GP, REG_SP, REG_RA, and REG_ZERO.

(c) VALUE

BrCondValue Returns 0 if branch will evaluate to false, and a 64-bit non-zero if branch condition will evaluate to true. This is valid only on conditional branch instructions and before the instruction is executed (InstBefore). An error is raised if instruction is not an conditional branch instruction.

EffAddrValue This is valid only for load and store instructions and before the instruction is executed (InstBefore). It returns the 64-bit effective address, the sum of the address contained in the base register and the signed 16-bit displacement.

(d) Adding Prototypes

void AddCallProto(char *s);

Prototypes of all procedure call that will be added have to be specified. The prototype is specifies as a C prototype contained in a string. The types of the arguments must be one of the valid argument types.

Arguments:

s is a null terminated string containing the procedure prototype.

(e) Adding Procedure Calls

The following primitives are used to add procedure calls. If multiple procedure calls are added to the same point, the order in which they were added is maintained for their invocations.

void AddCallProgram(PlaceType place, char *pname, ...);

A procedure call to procedure *pname* with the specified arguments is added.

Arguments:

place is of type Place. The valid values are ProgramBefore and ProgramAfter.

pname is a null terminated string containing the name of the procedure to be called.

... arguments to passed to the procedure specified by *pname*.

void AddCallProc(Proc* p, PlaceType place, char *pname, ...);

A procedure call to procedure *pname* with the specified arguments is added at *place* in the procedure p.

Arguments:

p is a pointer to procedure to which the call will be added.

place is of type Place. The valid values are ProcBefore and ProcAfter.

pname is a null terminated string containing the name of the procedure to be called.

... arguments to passed to the procedure specified by *pname*.

void AddCallBlock(Block *b, PlaceType place, char *pname, ...);

A procedure call to procedure *pname* with the specified arguments is added at *place* in the basic block *b*.

Arguments:

b is a pointer to basic block to which the call will be added.

place is of type Place. The valid values are BlockBefore and BlockAfter.

pname is a null terminated string containing the name of the procedure to be called.

... arguments to passed to the procedure specified by *pname*.

void AddCallInst(Inst *i, PlaceType place, char *pname, ...);

The valid values for place are InstBefore and InstAfter. A procedure call to procedure *pname* with the specified arguments is added at *place* in the instruction i.

Arguments:

i a pointer to instruction to which the call will be added.

place is of type Place. The valid values are InstBefore and InstAfter.

pname is a null terminated string containing the name of the procedure to be called.

... arguments to passed to the procedure specified by *pname*.

(f) Dynamic Memory Allocation

By default, Atom tries to link the sbrk's of the application and analysis program so they share the same heap space. However, in this approach the application's heap addresses may not be accurate. Some tools don't care about this. To obtain pure addresses, the heap may be split between the application and the analysis routines. Specify an *offset* with the -heap option to *atom*. The heap of the analysis program starts at application-heap-address + offset.

APPENDIX E: The ATOM man page

ATOM

atom - a system for building customized analysis tools

atom [application program] [instrumentation file] [analysis file] options ...

DESCRIPTION

The **atom** command:

Takes an application program, instrumentation file, and analysis file as input, and produces a non_shared instrumented application program executable. The name of the output executable may be specified with -o option, otherwise progname.atom is used.

If only the application program is specified, and no analysis and instrumentation files are specified, atom produces a non_shared uninstrumented executable.

The application program must be a fully liked non_shared object module. It may be produced by giving cc the Wl,-r and non_shared option. See ATOM's user manual on how to build application program modules.

The instrumentation and analysis file may be .c or .o file. If analysis routines are in more than one file, the .o of each file may be linked together into one file using ld with a -r option. The instrumentation file modules may also be combined in this manner.

OPTIONS

-o filename

filename is the name of the output executable.

-P toolname

toolname is the name of the tool produced in the intermediate step. It is not deleted.

-T textaddress

textaddress is the start of the text address for the application program.

-32addr

the textaddress starts at 0x20000000 and data address starts at 0x40000000.

-heap offset

the start of the heap of the analysis routines is bumped by offset. This is used to divide the heap between the application and analysis routines.

-D dataaddress

dataaddress is the start of the data address for the application program.

-tool toolname

toolname is the name of a general-purpose tool that is provided with the ATOM kit. Do ‘man atomtools ’ to find the current set of tools that are supported.

-g

produce the instrumented program with debugging information. This enables debugging of analysis routines

-dbx

allows debugging of instrumentation routines. ATOM puts the control in dbx with a stop at Instrument routine. See the Reference Manual Appendix B on how to use dbx with ATOM.

-v

display each step of atom.

-version

display the version number of ATOM.

DIAGNOSTICS

The diagnostic messages produced by **atom** are printed on the standard error file.

APPENDIX F: Known Deficiencies

- ATOM currently works only on non-shared modules with relocation records. These modules have to be linked with -Wl,-r and -non_shared switch.
- Optimization phases are not yet implemented. ATOM currently performs some simple analysis.
- Dynamic translation of addresses is not currently done. This would affect indirect procedure calls, because the call register would contain the instrumented program counter.