MARCH 1994

WRL
Research Report 94/2

ATOM: A System for Building Customized Program Analysis Tools

Amitabh Srivastava
Alan Eustace

ATOM: A System for Building Customized Program Analysis Tools

Amitabh Srivastava
Alan Eustace

March 1994

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Abstract

ATOM (Analysis Tools with OM) is a single framework for building a wide range of customized program analysis tools. It provides the common infrastructure present in all code-instrumenting tools; this is the difficult and time-consuming part. The user simply defines the tool-specific details in instrumentation and analysis routines. Building a basic block counting tool like Pixie with ATOM requires only a page of code.

ATOM, using OM link-time technology, organizes the final executable such that the application program and user's analysis routines run in the same address space. Information is directly passed from the application program to the analysis routines through simple procedure calls instead of inter-process communication or files on disk. ATOM takes care that analysis routines do not interfere with the program's execution, and precise information about the program is presented to the analysis routines at all times. ATOM uses no simulation or interpretation.

ATOM has been implemented on the Alpha AXP under OSF/1. It is efficient and has been used to build a diverse set of tools for basic block counting, profiling, dynamic memory recording, instruction and data cache simulation, pipeline simulation, evaluating branch prediction, and instruction scheduling.

This paper will appear at the SIGPLAN '94 Conference on Programming Language Design and Implementation. It replaces Technical Note TN-41, an earlier version of the same material.

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 three classes of tools for different machines and applications have been developed. The first class consists of basic block counting tools like Pixie[9], Epoxie[14] and QPT[8] that count the number of times each basic block is executed. The second class consists of address tracing tools that generate data and instruction traces. Pixie and QPT also generate address traces and communicate trace data to analysis routines through inter-process communication. Tracing and analysis on the WRL Titan[3] communicated via shared memory but required operating system modifications. MPTRACE [6] is also similar to Pixie but it collects traces for multiprocessors by instrumenting assembly code. ATUM [1] generates address traces by modifying microcode and saves a compressed trace in a file that is analyzed offline. The third class of tools consists of simulators. Tango Lite[7] supports multiprocessor simulation by instrumenting assembly language code. PROTEUS[4] also supports multiprocessor simulation but instrumentation is done by the compiler. g88[2] simulates Motorola 88000 using threaded interpreter techniques. Shade[5] attempts to address the problem of large address traces by allowing selective generation of traces but has to resort to instruction-level simulation.

These existing tools have several limitations.

First, most tools are designed to perform a single specific type of instrumentation, typically block counting or address tracing. Modifying these tools to produce more detailed or less detailed information is difficult. A tool generating insufficient information is of no use to the user.

Second, most address tracing tools 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 need 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.

Fourth, tools such as Tango Lite, which instrument assembly language code, change the application program's heap addresses. Instrumenting library routines is inconvenient as all libraries have to be available in assembly language form.

Finally, most address tracing tools provide trace data collection mechanisms. Data in form of address traces is communicated to the data analysis routines through inter-process communication, or files on disk. Both are expensive, and the large size of address traces further aggravates this problem. Using a shared buffer reduces this expense but still requires a lot of process switching and sometimes can be implemented efficiently only with changes to the operating system.

ATOM overcomes these limitations by providing the principal ingredient in building performance tools. The important features that distinguish it from previous systems are listed below.

- ATOM is a tool-building system. A diverse set of tools ranging from basic block counting to cache modeling can be easily built.
- ATOM provides the common infrastructure in all code-instrumenting tools, which is the cumbersome part. The user simply specifies the tool details.
- ATOM allows selective instrumentation. The user specifies the points in the application program to be instrumented, the procedure calls to be made, and the arguments to be passed.
- The communication of data is through procedure calls. Information is *directly* passed from the application program to the specified analysis routine with a procedure call instead of through interprocess communication, files on disk, or a shared buffer with central dispatch mechanism.
- Even though the analysis routines run in the same address space as the application, precise information about the application program is presented to analysis routines at all times.
- As ATOM works on object modules, it is independent of compiler and language systems.

In this paper, we describe the design and implementation of ATOM. We show through a real example how to build tools. Finally, we evaluate the system's performance.

2 Design of ATOM

The design of ATOM is based on the observation that although tasks like basic block counting and cache simulation appear vastly different, all can be accomplished by instrumenting a program at a few selected points. For example, basic block counting tools instrument the beginning of

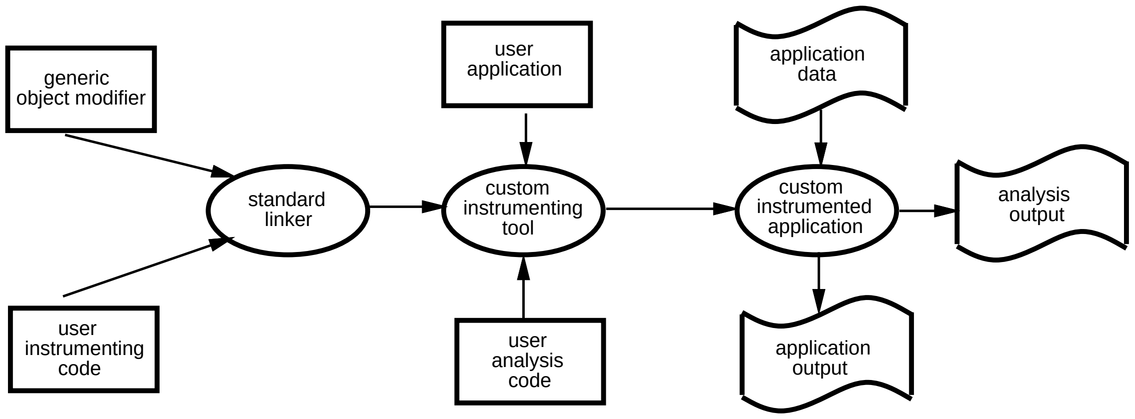

Figure 1: The ATOM Process

each basic block, data cache simulators instrument each load and store instruction, and branch prediction analyzers instrument each conditional branch instruction. Therefore, ATOM allows a procedure call to be inserted before or after any program, procedure, basic block, or instruction. 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.

Furthermore, ATOM separates the tool-specific part from the common infrastructure needed in all tools. It provides the infrastructure for object-code manipulation and a high-level view of the program in object-module form. The user defines the tool-specific part in *instrumentation* routines by indicating the points in the application program to be instrumented, the procedure calls to be made, and the arguments to be passed. The user also provides code for these procedures in the *analysis* routines. The 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.

ATOM[1] internally works in two steps, as shown in Figure 1.

In the first step, common machinery is combined with the user's instrumentation routines to

[1]Externally, the user specifies: *atom prog inst.c anal.c -o prog.atom*
to produce the instrumented program *prog.atom*.

build a custom tool. This tool will instrument an application program at points specified by the user's instrumentation routines.

In the second step, this custom tool is applied to the application program to build an instrumented application program executable. The instrumented executable is organized so that information from application program is communicated directly to procedures in the analysis routines through procedure calls. The data is passed as arguments to the handling routine in the requested form, and does not have to go through a central dispatch mechanism.

To reduce the communication to a procedure call, the application program and the analysis routines run in the same address space. ATOM partitions the symbol name space and places 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 was executing uninstrumented. Section 4 describes how the system guarantees the precise information.

ATOM, built using OM[11], is independent of any compiler and language system because it operates on object-modules. Since OM is designed to work with different architectures[2], ATOM can be applied to other architectures.

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.

The user provides three files to ATOM: the application program object module that is to be instrumented, a file containing the instrumentation routines, and a file containing the analysis routines. The instrumentation routines specify where the application program is to be instrumented and what procedure calls are to be made. The user provides code for these procedures in the analysis routines. The next two sections show how to write the instrumentation and analysis routines for our example tool.

[2]OM was initially implemented on the DECStations running under ULTRIX and was ported to Alpha AXP running under OSF/1. ULTRIX, DECStation and Alpha AXP are trademarks of Digital Equipment Corporation.

```
Instrument(int iargc, char **iargv)
{
Proc *p;
Block *b;
Inst *inst;
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");
}
```

Figure 2: Instrumentation Routines: Branch Counting Tool

Defining Instrumentation Routines

Our branch counting tool needs to examine all the conditional branches in the program. We traverse the program a procedure at a time, and examine each basic block in the procedure. If the last instruction in the basic block is a conditional branch, we instrument the instruction. The instrumentation routines are given in Figure 2.

ATOM starts the instrumentation process by invoking the `Instrument` procedure[3]. All instrumentation modules contain the `Instrument` procedure. The instrumentation process begins by defining the prototype of each procedure in the analysis routine that will be called from the application program. This enables ATOM to correctly interpret the arguments. The `AddCallProto` primitive is used to define the prototypes. In our example, prototypes of four analysis procedures `OpenFile`, `CondBranch`, `PrintBranch`, and `CloseFile` are defined. Besides the standard C data types as arguments, ATOM supports additional types such as `REGV` and `VALUE`. If the argument type is `REGV`, the actual argument is not an integer but a register number, and the run-time contents of the specified register are passed. For the `VALUE` argument type, the actual argument may be `EffAddrValue` or `BrCondValue`. `EffAddrValue` passes the memory address being referenced by load and store instructions. `BrCondValue` is used for conditional branches and passes zero if the run-time branch condition evaluates to a false and a non-zero value if the condition evaluates to true. `CondBranch` uses the argument type `VALUE`.

ATOM allows the user to traverse the whole program by modeling a program as consisting of a sequence of procedures, basic blocks and instructions. `GetFirstProc` returns the first procedure in the program, and `GetNextProc` returns the next procedure. The outermost `for` loop traverses the program a procedure at a time. In 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 specifies whether the

[3]The Instrument procedure takes argc and argv as arguments which can be optionally passed from the **atom** command line.

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 (program counter) 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[4]. 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 by the instrumentation routines.

Defining Analysis Routines

The analysis routines contain code and data for all procedures needed to analyze information that is passed from the application program. These include procedures that were specified in the instrumentation routines but may contain other procedures that these procedures may call. The analysis routines do not share the code for any procedure with the application program, including library routines.

Code for procedures `OpenFile`, `CondBranch`, `PrintBranch`, and `CloseFile` whose prototypes were defined in instrumentation routines are given in Figure 3. 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.

Collecting Program Statistics

To find the branch statistics, ATOM is given as input the fully linked application program in object-module format, the instrumentation routines, and the analysis routines. The output is 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]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.

```
#include <stdio.h>
File *file

struct BranchInfo{
    long taken;
    long notTaken;
} *bstats;

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

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);
}

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

Figure 3: Analysis Routines: Branch Counting Tool

4 Implementation of ATOM

ATOM is built using OM[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[12, 13] to the intermediate representation, and finally outputs an executable.

ATOM starts by linking the user's instrumentation routines with OM using the standard linker to produce a custom tool. This tool is given as input the application program and the analysis routines. It uses OM's infrastructure to build symbolic representations of the application program and the analysis routines. The traversal and query primitives interface with the intermediate representation of the program to provide the information requested. More details of OM's intermediate representation and how it is built are described in [11]. We extended the OM's representation so it can be conveniently annotated for procedure call insertions.

OM's code generation pass builds the instrumented executable from the intermediate representation. This pass is modified to organize the data and text sections in a specific order because ATOM has to ensure that precise information about the application is presented to the analysis routines at all times.

In this section, we first describe the extensions to the intermediate representation and the insertion of procedure calls. Next, we discuss how we minimize the number of registers that need to be saved and restored. Finally, we describe how ATOM organizes the final executable.

Inserting Procedure Calls

We extended the intermediate representation of OM to have a slot for *actions* that may be performed before or after the entity is executed. The entity may be a procedure, basic block, instruction or an edge[5]. The `AddCall` primitives annotate the intermediate representation by adding a structure to the action slot describing the call to be inserted, arguments to be passed, and indicating when the call is to be made. Currently, adding calls to edges is not implemented. The prototype of the procedure must already have been added with the `AddCallProto` primitive, and ATOM verifies that. The action slot contains a linked list of all such actions to be performed as multiple calls can be added at a point. The order in which they are added is maintained so that calls will be made in the order they were specified.

After the intermediate representation has been fully annotated, the procedure calls are inserted.

[5]An edge connects two basic blocks and represents the transfer of control between them.

This process is easy because all insertion is done on OM's intermediate representation and no address fixups are needed. ATOM, like QPT, does not steal any registers from the application program[6]. It allocates space on the stack before the call, saves registers that may be modified during the call, restores the saved registers after the call and deallocates the stack space. This enables a number of mechanisms such as signals, setjmp and vfork to work correctly without needing any special attention.

The calling conventions are followed in setting up calls to analysis routines. The first six arguments are passed in registers and the rest are passed on the stack. The number of instructions needed to set up an argument depends on the type of the argument. For example, a 16-bit integer constant can be built in 1 instruction, a 32-bit constant in two instructions, a 64-bit program counter in 3 instructions and so on. Passing contents of a register takes 1 instruction.

To make the call, a pc-relative subroutine branch instruction[7] is used if the analysis routine is within range, otherwise, the value of the procedure is loaded in a register and a `jsr` instruction is used for the procedure call. The return address register is always modified when a call is made so we always save the return address register. This register becomes a scratch register; it is used for holding the procedure's address for the `jsr` instruction.

Reducing Procedure Call Overhead

The application program may have been compiled with interprocedural optimizations and may contain routines that do not follow the calling conventions[8]. Therefore, all registers that may be modified in the call to the analysis routines need to be saved. The analysis routines, on the other hand, have to follow the calling conventions[9] as they have to allow arbitrary procedures to be linked in. The calling conventions define some registers as *callee-save* registers that are preserved across procedure calls, and others as *caller-save* registers that are not preserved across procedure calls. All the caller-save registers need to be saved before the call to the analysis routine and restored on return from the analysis routines. This is necessary to maintain the execution state of the application program. The callee-save registers would automatically be saved and restored in

[6]Pixie steals three registers away from the application program for its own use. Pixie maintains three memory locations that have the values of these three registers, and replaces the use of these registers by uses of the memory locations.

[7]Alpha[10] has a signed 21-bit pc-relative subroutine branch instruction.

[8]The application may contain hand-crafted assembly language code that often does not follow standard conventions. ATOM can handle such programs.

[9]Analysis routines are analogous to standard library routines that have to follow calling conventions so they can be linked with programs.

analysis routines if they are used by them. Two issues need to be addressed here: where to save these caller-save registers, and which caller-save registers to save.

Saving registers in the application code, where the call is being inserted, is not a good idea if there are more than a few registers to be saved, as it may cause code explosion. We create a wrapper routine for each analysis procedure. The wrapper routine saves and restores the necessary registers, and makes the call to the analysis routine. The application program now calls the wrapper routine instead of the analysis routine. Unfortunately, this creates an indirection in calls to analysis routines. However, this has the advantage that it makes no changes to the analysis code so it works well with a debugger like dbx. This is the default mechanism.

ATOM provides an additional facility in which the saves and restores of caller-save registers are added to the analysis routines. No wrapper routines are created in this case. The extra space is allocated in the analysis routine's stack frame. This requires bumping the stack frame and fixing stack references in the analysis routines as needed. This is more work but is more efficient as analysis routines are called directly. Since this modifies the analysis routines, it hampers source-level debugging. This mechanism is available as a higher optimization option.

The number of registers that need to be saved and restored is reduced by examining the analysis routines. The data flow summary information of the analysis routines determines all the registers that may be modified when the control reaches a particular analysis procedure. Only these registers need to be saved and restored. We use register renaming to minimize the number of different caller-save registers used in the analysis routines.

Moreover, if an analysis routine contains procedure calls to other analysis routines, we save only the registers directly used in this analysis routine and delay the saves of other registers to procedures that may be called. We only do this if none of the procedure calls occur in a loop. Thus we distribute the cost of saving registers; the overhead now depends on the path the program takes. This helps analysis routines that normally return if their argument is valid but otherwise raise an error. Raising an error typically involves printing an error message and touching a lot more registers. For such routines, the common case of a valid argument has low overhead as few registers are saved. This optimization is available in the current implementation.

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[11] 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 at the cost of increasing the code size. These refinements have not been added to the current system.

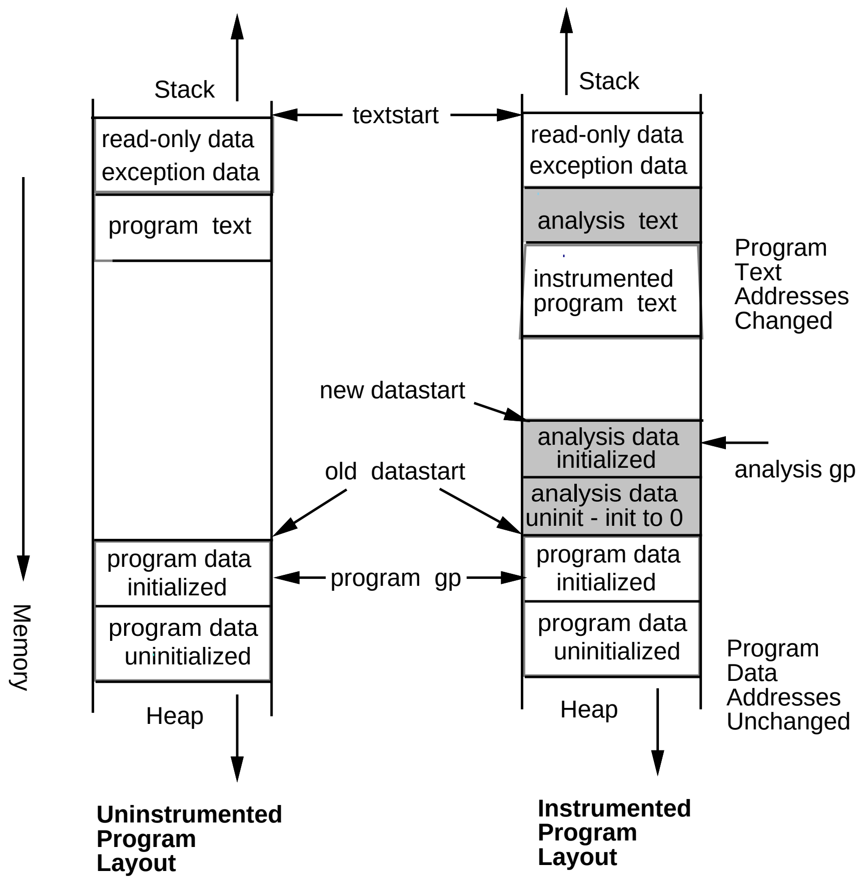

Figure 4: Memory layout

Keeping Pristine Behavior

One major 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. Analysis routines do not share any procedures or data with the application program; they contain data and code for all procedures including library routines that they may need.

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. In an executable, all initialized data must be located before all uninitialized data, so the uninitialized data of the analysis routines is converted to initialized data by initializing it with zero. The start of the stack and heap[10] are unchanged, so all stack and heap addresses are same as before. This is shown in Figure 4.

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 is not the original text address. We have not implemented in our current system the ability to return original text address in such cases.

Analysis routines may dynamically allocate data on heap. Since analysis routines and the application program do not share any procedures, there are two `sbrk`[11] routines, one in the application program and the other in the analysis routines that allocate space on the same heap. ATOM provides two options for *tools that must allocate dynamic memory*.

The first method links the variables of the two `sbrks`, so both allocate space on the same heap without stepping on each other. Each starts where the other left off. This method is useful for tools that are not concerned with the heap addresses being same as in the uninstrumented version of the program. Such tools include basic block counting, branch analysis, inline analysis and so on. This method is also sufficient for tools such as cache modeling that require precise heap addresses but do not allocate dynamic memory in analysis routines. This is the default

[10]On the Alpha AXP under OSF/1 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.

[11]sbrk routines allocate more data space for the program.

Analysis Tool	*Tool Description*	*Time to instrument SPEC92 suite*	*Average Time*
branch	prediction using 2-bit history table	110.46 secs	5.52 secs
cache	model direct mapped 8k byte cache	120.58 secs	6.03 secs
dyninst	computes dynamic instruction counts	126.31 secs	6.32 secs
gprof	call graph based profiling tool	113.24 secs	5.66 secs
inline	finds potential inlining call sites	146.50 secs	7.33 secs
io	input/output summary tool	121.60 secs	6.08 secs
malloc	histogram of dynamic memory	97.93 secs	4.90 secs
pipe	pipeline stall tool	257.48 secs	12.87 secs
prof	Instruction profiling tool	122.53 secs	6.13 secs
syscall	system call summary tool	120.53 secs	6.03 secs
unalign	unalign access tool	135.61 secs	6.78 secs

Figure 5: Time taken by ATOM to instrument 20 SPEC92 benchmark programs

behavior.

The second method is for tools that allocate dynamic memory and also require heap addresses to be same as in the uninstrumented version of the application program. To keep the application heap addresses as before, the heap is partitioned between the application and the analysis routines. The application heap starts at the same address but the analysis heap is now made to start at a higher address. The user supplies the offset by which the start of analysis heap is changed. ATOM modifies the `sbrk` in analysis routines to start at the new address; the two `sbrks` are not linked this time. The disadvantage of this method is that there is no runtime check if the application heap grows and enters into the analysis heap.

5 Performance

To find how well ATOM performs, two measurements are of interest: how long ATOM takes to instrument a program, and how the instrumented program's execution time compares to the uninstrumented program's execution time.

We used ATOM to instrument 20 SPEC92 programs with 11 tools. The tools are briefly described in Figure 5. The time taken to instrument a program is the sum of the ATOM's processing time and the time taken by the user's instrumentation routines. The time taken by a tool varies as each tool does different amounts of processing. For example, the *malloc* tool

Analysis Tool	*Instrumentation points*	Number of Arguments	*Time taken by Instrumented Program*
branch	each conditional branch	3	3.03x
cache	each memory reference	1	11.84x
dyninst	each basic block	3	2.91x
gprof	each procedure/each basic block	2	2.70x
inline	each call site	1	1.03x
io	before/after write procedure	4	1.01x
malloc	before/after malloc procedure	1	1.02x
pipe	each basic block	2	1.80x
prof	each procedure/each basic block	2	2.33x
syscall	before/after each system call	2	1.01x
unalign	each basic block	3	2.93x

Figure 6: Execution time of Instrumented SPEC92 Programs as compared to uninstrumented SPEC92 programs

simply asks for the malloc procedure and instruments it; the processing time is very small. The *pipe* tool does static CPU pipeline scheduling for each basic block at instrumentation time and takes more time to instrument an application. The time taken to instrument 20 SPEC92 programs with each tool is also shown in Figure 5.

The execution time of the instrumented program is the sum of the execution time of the uninstrumented application program, the procedure call setup, and the time spent in the analysis routines. This total time represents the time needed by the user to get the final answers. Many systems process the collected data offline and do not include those numbers as part of data collecting statistics. The time spent in analysis routines is analogous to the postprocessing time required by other systems.

We compared each instrumented program's execution time to the uninstrumented program's execution time for each tool. Figure 6 shows the ratios for the SPEC92 programs. The procedure call overhead is dependent on the code in the analysis routines, and the number and type of arguments that are passed. ATOM uses the data flow summary information along with register renaming to find the necessary registers to save. The contribution of procedure call overhead in the instrumented program execution time is also dependent on the number of times the procedure calls take place. The *inline* tool instruments only procedure call sites; the total overhead is much less than the *cache* tool, which instruments each memory reference. The amount of work the analysis routines do when the control reaches them is totally dependent on information the user is trying to compute. Although the communication overhead is small, we expect it to decrease further when we implement live register analysis and inlining.

All measurements were done on Digital Alpha AXP 3000 Model 400 with 128 Mb memory.

6 Status

ATOM is built using OM and currently runs on Alpha AXP under OSF/1. It has been used with programs compiled with Fortran, C++ and two different C compilers. The system currently works on non-shared library modules. Work is in progress for adding support for shared libraries.

ATOM has been used both in hardware and software projects. Besides the SPEC92 benchmarks, it has successfully instrumented real applications of up to 96 Megabytes. The system is being used extensively inside Digital and at a few universities[12].

Our focus until now has mainly been on functionality. Few optimizations have been added to reduce the procedure call overhead. Currently, reduction in register saves has been obtained by computing data flow summary information of analysis routines. We plan to implement live register analysis along with inlining to further improve the performance. We are just starting to instrument the operating system.

7 Conclusion

By separating object-module modification details from tool details and by presenting a high-level view of the program, ATOM has transferred the power of building tools to hardware and software designers. A tool designer concentrates only on what information is to be collected and how to process it. Tools can be built with few pages of code and they compute only what the user asks for. ATOM's fast communication between application and analysis means that there is no need to record traces as all data is immediately processed, and final results are computed in one execution of the instrumented program. Thus, one can process long-running programs. It 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.

[12]ATOM is available to external users. If you would like a copy, please contact the authors.

Acknowledgements

Great many people have helped us bring ATOM to its current form. Jim Keller, Mike Burrows, Roger Cruz, John Edmondson, Mike McCallig, Dirk Meyer, Richard Swan and Mike Uhler were our first users and they braved through a mine field of bugs and instability during the early development process. Jeremy Dion, Ramsey Haddad, Russel Kao, Greg Lueck and Louis Monier built popular tools with ATOM. Many people, too many to name, gave comments, reported bugs, and provided encouragement. Roger Cruz, Jeremy Dion, Ramsey Haddad, Russell Kao, Jeff Mogul, Louis Monier, David Wall, Linda Wilson and anonymous PLDI reviewers gave useful comments on the earlier drafts of this paper. Our thanks to all.

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] Robert Bedichek. Some Efficient Architectures Simulation Techniques. *Winter 1990 USENIX Conference*, January 1990.

[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 Research Report 89/14, Sep 1989.

[4] 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.

[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] 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.

[7] 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.

[8] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software, Practice and Experience*, vol 24, no. 2, pp 197-218, February 1994.

[9] MIPS Computer Systems, Inc. *Assembly Language Programmer's Guide*, 1986.

[10] Richard L. Sites, ed. *Alpha Architecture Reference Manual* Digital Press, 1992.

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

[12] Amitabh Srivastava and David W. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, to appear. Also available as WRL Research Report 94/1, February 1994.

[13] Amitabh Srivastava. Unreachable procedures in object-oriented programming, *ACM LOPLAS*, Vol 1, #4, pp 355-364, December 1992. Also available as WRL Research Report 93/4, August 1993.

[14] 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 Research Report 92/3, May 1992.

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Recovery in Spritely NFS.’’
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’
Amitabh Srivastava, David W. Wall.
WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’
Amitabh Srivastava, Alan Eustace.
WRL Research Report 94/2, March 1994.

WRL Technical Notes