ATOM

User Manual

March, 1994
Digital Equipment Corporation
Maynard, Massachusetts

1. Introduction

Program Analysis tools are extremely important for computer architects and software engineers. Computer architects use them to test and measure new architectural designs, while software engineers use them to identify critical pieces of code in programs or to examine how well a branch prediction or instruction scheduling algorithm is performing. Although each of these tools appears quite different, each can be implemented simply and efficiently through code instrumentation.

ATOM provides a flexible code instrumentation interface that is capable of building a wide variety of interesting tools.

2. Motivation

Commercial applications are frequently large poorly understood programs. Performance depends on many factors: program organization, algorithms, compilers, operating systems, libraries, microprocessors details, memory system organization, etc. The key to improving performance is to understand the dynamic behavior of these programs. Unfortunately, gathering statistics about program behavior is often difficult, if not impossible.

For example, even determining the order procedures are called during the execution of simple programs is extremely difficult. Program call trees can be complicated, and programs often make calls to library routines, which call other libraries. Conceptually, this information can be found by modifying each procedure in the user program to print the procedure name. When the program is recompiled and executed, a trace of procedure names will be printed. If the user program calls the math library, the math library would also need to be modified. Unfortunately, hand instrumenting breaks down for the standard IO library, since instrumenting the printing routines with print statements is not possible.

Often important information cannot be gathered by hand instrumented source code. An instruction cache simulator needs the address of every instruction executed, and this information is not available at the source level. Hand instrumentation of the assembly language instructions is very difficult, and the result would change the instruction addresses, and therefore disturb the cache model.

ATOM provides the power and flexibility of hand instrumented code without these drawbacks. Special purpose tools are easily constructed that can be applied to any application program. These tools can be built simply and efficiently using the ATOM procedural interface.

3. Application Program

ATOM takes as input a specially linked application program. To illustrate the process, consider a simple application program called helloworld.c.

```
#include <stdio.h>
main()
{
   printf("Hello World\n");
}
```

To prepare the application for applying ATOM based tools, the application is linked using the **-Wl,-r -non_shared** flags to **cc** or **f77**.

> cc -Wl,-r -non_shared helloworld.c -o helloworld.rr

The **-Wl,-r** argument passes the **-r** switch to the linker to indicate that relocation information should be included in the output file. The **-non_shared** switch chooses non-shared rather than shared libraries. The output of this compile step is placed by convention in a relocatable object file with the **.rr** extension.

ATOM provides a long and growing list of standard analysis and performance tools. A partial list is shown below.

Tool	**Description**
iprof	instruction profiling tool
liprof	instruction profiling tool at basic block level
pipe	pipeline stall tool
lpipe	pipeline stall at basic block level
syscall	system call summary tool
memsys	memory system bandwidth tool
lmemsys	low level memory system bandwidth tool
io	input/output summary tool
io2	new input/output summary tool
unalign	byte and short data profile
gprof	gprof profiling tool
3rd	memory checker and leak finder (a-la-Purify)
pixie	superset of the pixie
heapcheck	leak detection tool

The following command can be used to apply the *syscall* tool to the helloworld.rr application.

> atom helloworld.rr -tool syscall -o helloworld.syscall

By convention, the output file name is formed by concatenating the original executable name with the tool name. Each time *helloworld.syscall* is executed, a *syscall.out* file is created.

```
> helloworld.syscall
Hello World
> more syscall.out
System Call             Calls        Time(ms)
write                   1            4.4
close                   3            0.0
ioctl                   1            0.2

Total                                4.6
```

The original uninstrumented executable can be created directly from the relocatable object file using the standard linker.

> ld helloworld.rr -o helloworld

4. Introductory Tools

Fundamentally, there will always be more problems than tools. ATOM allows provides a rich set of tool building primitives that can be applied to a very wide range of problems. This section describes three basic tools: procedure tracing, instruction profiling, and instruction and data address tracing.

An ATOM based tool requires the tool builder to specify two files, the *instrumentation* file and the *analysis* file. The *instrumentation* file defines where procedure calls are to be added to the application program, and the *analysis* file defines those procedures and the data structures that are necessary to record the and print the results. Both files are written in **C**.

4.1. Procedure Tracing

The *ptrace* tool prints the names of each procedure called in the order the procedures are executed. This process requires instrumenting the application program to add a procedure call at the start of every procedure in the application program to print the name of the procedure each time the procedure is executed.

By convention the instrumentation for the *ptrace* tool is placed in the file *ptrace.inst.c*.

```
 1  #include <stdio.h>
 2  #include <instrument.h>
 3
 4  Instrument()
 5  {
 6    Proc *proc;
 7    AddCallProto("ProcTrace(char *)");
 8
 9    for (proc = GetFirstProc(); proc != NULL; proc = GetNextProc(proc))
10      {
11        AddCallProc(proc,ProcBefore,"ProcTrace",ProcName(proc));
12      }
13  }
```

ATOM links the instrumentation routine with the ATOM library to create an executable that reads the application program, calls the **Instrument** procedure to modify the application program, and produces the instrumented executable.

Statement 2 includes the definitions for the ATOM procedural interface and data structures. Statement 4 defines the required **Instrument** procedure, which defines the interface to each analysis procedure and places calls to those procedures at the correct positions.

Statement 7 calls **AddCallProto** to defines the *ProcTrace* analysis procedure. **ProcTrace** is defined to take a single argument of type pointer to character.

Statement 9 uses **GetFirstProc** and **GetNextProc** to step through each procedure in the application program. Since this is a fully linked program, these procedures include both user defined procedures, and all the procedures defined in any libraries. For each procedure, statement 11 calls **AddCallProc**, which adds call to the procedure pointed to by *proc*. The *ProcBefore* argument indicates that the procedure call is to be added before the procedure starts execution. The name of the procedure to call "ProcTrace". The final argument passed is the string returned by **ProcName**, which when given a pointer to a procedure, returns the procedure name.

This tool is not complete without the definition of the **ProcTrace** procedure, defined in the file *ptrace.anal.c*.

```
1  #include <stdio.h>
2
3  void ProcTrace(char *name)
4  {
5    fprintf(stderr,"%s\n",name);
6  }
```

ProcTrace prints to standard error the character string passed as an argument. Notice that in statement 3, the type returned by the procedure is *void*. Analysis procedures are forbidden from returning a value to the application program.

The *ptrace* tool is applied to the helloworld application as shown below.

> atom helloworld.rr ptrace.inst.c ptrace.anal.c -o helloworld.ptrace

The instrumented application program is placed in the file helloworld.ptrace. The result of running this tool may be somewhat surprising.

```
> helloworld.ptrace
__start
main
printf
_doprnt
__getmbcurmax
strchr
strlen
memcpy
fwrite
_wrtchk
_findbuf
__geterrno
__isatty
__ioctl
__seterrno
memcpy
memchr
_xflsbuf
__write
hello world
exit
__ldr_atexit
__ldr_context_atexit
_cleanup
fclose
fflush
__close
fclose
fflush
__close
fclose
__close
```

This tool is deceptively simple, yet there are many fundamental concepts at work. Only the procedure calls in the original application program are instrumented. The call to **ProcTrace** did not occur in the original application, and therefore it does not appear in a trace of the application procedures executed. Likewise, the procedure calls that output the names of each procedure are also not included. If the application and the analysis program both call *printf*, ATOM links in two copies, and only instruments the copy that occurs in the application program. ATOM also correctly deciphers procedures that have multiple entry points, placing a call to the *ProcTrace* routine at each of the potential entry points.

4.2. Profile Tool

The *prof* tool is useful for finding critical sections of code. This tool instruments an application such that each time the application is run, a file called *prof.out* is created that contains the number of instructions that are executed. Here is the output of applying the *prof* tool to *helloword.rr* and executing the *helloworld.prof* program.

> more prof.out

Procedure	Instructions	Percentage
__start	157	11.418
main	14	1.018
exit	22	1.600
printf	38	2.764
memcpy	50	3.636
__ldr_atexit	16	1.164
_cleanup	169	12.291
fclose	176	12.800
fflush	91	6.618
_xflsbuf	68	4.945
_wrtchk	35	2.545
_findbuf	59	4.291
_exit	2	0.145
_doprnt	132	9.600
__ldr_context_atexit	11	0.800
__geterrno	5	0.364
__seterrno	7	0.509
__close	12	0.873
__write	4	0.291
__isatty	17	1.236
strlen	26	1.891
__getmbcurmax	7	0.509
strchr	93	6.764
fwrite	118	8.582
__ioctl	4	0.291
memchr	42	3.055
Total		1375

This tool requires that the number of instructions in the program be counted. The most efficient place to compute instruction counts is inside each basic block. Each time a basic block is executed, a fixed number of instructions are executed. This information can be communicated to the analysis procedures by adding a procedure call to an analysis routine inside of each basic block that passes a the procedure number and the number of instructions in the basic block as arguments.

The *prof.inst.c* instrumentation procedure is shown below.

```
 1  #include <stdio.h>
 2  #include <instrument.h>
 3
 4  Instrument()
 5  {
 6    Proc *proc;
 7    Block *block;
 8    Inst *inst;
 9    int number = 0;
10
11     AddCallProto("OpenFile(int)");
12     AddCallProto("ProcedureCount(int,int)");
13     AddCallProto("ProcedurePrint(int,char *)");
14     AddCallProto("CloseFile()");
15
16     AddCallProgram(ProgramBefore,"OpenFile",GetProgramInfo(ProgramNumberProcs));
17     for (proc = GetFirstProc(); proc != NULL; proc = GetNextProc(proc))
18       {
19         for (block = GetFirstBlock(proc); block != NULL; block = GetNextBlock(block))
20           {
21             AddCallBlock(block,BlockBefore,"ProcedureCount",
22                          number,GetBlockInfo(block,BlockNumberInsts));
23           }
24         AddCallProgram(ProgramAfter,"ProcedurePrint",number,ProcName(proc));
25         number++;
26       }
27     AddCallProgram(ProgramAfter,"CloseFile");
28  }
```

Statements 11 through 14 define the interface to the analysis procedures. In statement 16, the application is instrumented with a call to the *OpenFile* procedure before the first instruction in the application program is executed. The argument is defined in statement 11 to be an integer, and the value of this argument is provided by a call to the **GetProgramInfo** procedure. The information requested is the *ProgramNumberProcs*, which returns the number of procedures in the application program. This value is passed to the *OpenFile* procedure.

In statement 17, each procedure in the application program is traversed. For each procedure, statement 19 traverses each of the basic block. Statement 21 adds a call to the *ProcedureCount* analysis procedure. The *BlockBefore* argument places the call to *ProcedureCount* before the block is executed. *ProcedureCount* is defined in statement 12 to take two integer arguments. This first argument is the procedure number, and the second is the value returned by the *GetBlockInfo* procedure. When passed an argument of *BlockNumberInsts*, this function returns the number of instructions in the basic block.

For every procedure in the program, statement 24 adds a call to *ProcedurePrint*. These calls are added consecutively at the end of the program. The first argument is the procedure index and the second argument is the procedure name.

Statement 27 adds a call to the *CloseFile* procedure after the application program finishes execution and prints the result file.

The analysis procedures are defined in the *prof.anal.c* file.

```
 1  #include <stdio.h>
 2  #include <assert.h>
 3
 4  long instrTotal;
 5  long *instrPerProc;
 6
 7  FILE *file;
 8  void OpenFile(int number)
 9  {
10    instrPerProc = (long *) malloc(sizeof(long) * number);
11    assert(instrPerProc != NULL);
12    bzero(instrPerProc, sizeof(long) * number);
13    file = fopen("prof.out","w");
14    assert(file != NULL);
15    fprintf(file,"%30s %15s %10s\n","Procedure",
16            "Instructions","Percentage");
17  }
18
19  void ProcedureCount(int number, int instructions)
20  {
21    instrTotal += instructions;
22    instrPerProc[number] += instructions;
23  }
24
25  void ProcedurePrint(int number, char *name)
26  {
27    if (instrPerProc[number] > 0)
28      {
29        fprintf(file,"%30s %15ld %9.3f\n",
30                name, instrPerProc[number],
31                ((float) instrPerProc[number] / instrTotal)*100.0);
32      }
33  }
34
35  void CloseFile()
36  {
37    fprintf(file,"\n%30s %15ld\n", "Total", instrTotal);
38    fclose(file);
39  }
```

Statement 8 defines the **OpenFile** analysis procedure, which takes a single argument that defines the number of procedures in the application program. Statements 10 allocates an array of longs that holds the number of instructions executed indexed by the procedure number. Statement 11 assures that the *malloc* call succeeded, and statement 12 initializes all counters to start at zero. Statement 15 prints the column headings.

Statement 19 defines the **ProcedureCount** analysis procedure which takes the procedure number and the number of instructions in the basic block as arguments and increments the total number of instructions executed and the number of instructions executed in each procedure.

Statement 24 defines the procedure that prints the results, and statement 26 filters unexecuted procedures from the output file.

Statement 33 defines the **CloseFile** procedure, which prints the total for the application and closes the output file.

4.3. Instruction and Data Address Tracing Tool

Instruction and data address tracing has been used for many years as a technique for capturing and analyzing program behavior. The tool saves the first address executed in each basic block and the effective address of all load and store addresses. The output is used to replay the execution of the program for postprocessing tool like instruction and data cache simulators.

```
 1  #include <stdio.h>
 2  #include <instrument.h>
 3
 4  Instrument()
 5  {
 6    Proc *proc;
 7    Block *block;
 8    Inst *inst;
 9
10    AddCallProto("OpenFile()");
11    AddCallProto("InstReference(REGV)");
12    AddCallProto("DataReference(VALUE)");
13    AddCallProto("CloseFile()");
14
15    AddCallProgram(ProgramBefore,"OpenFile");
16
17    for (proc = GetFirstProc(); proc != NULL; proc = GetNextProc(proc))
18      {
19        for (block = GetFirstBlock(proc); block != NULL; block = GetNextBlock(block))
20          {
21            AddCallBlock(block,BlockBefore,"InstReference",REG_PC);
22            for (inst = GetFirstInst(block); inst != NULL; inst = GetNextInst(inst))
23              {
24                if (IsInstType(inst,InstTypeLoad) || IsInstType(inst,InstTypeStore))
25                  {
26                    AddCallInst(inst,InstBefore,"DataReference",EffAddrValue);
27                  }
28              }
29          }
30      }
31    AddCallProgram(ProgramAfter,"CloseFile");
32  }
```

Statement 11 defines the **InstReference** analysis procedure to take one argument of type *REGV*. This type is used to pass the contents of processor registers. Statement 23 adds the **InstReference** procedure before each basic block. The argument passed is the program counter of the first instruction in the basic block. This is the program counter had the program not been instrumented.

Statement 12 defines the **DataReference** analysis procedure, which takes a single argument of type *VALUE*. This type is used to pass values that ATOM must explicitly compute. In Statement 30, a call to **DataReference** is added before load or store instructions and passed the effective address, which ATOM computes by adding the base address from the register file and the displacement from the instruction.

```
 1  #include <stdio.h>
 2  #include <assert.h>
 3  #define BUFSIZE 65536
 4  char buf[BUFSIZE];
 5
 6  FILE *file;
 7  void OpenFile()
 8  {
 9    file = fopen("trace.out","w");
10    assert(file != NULL);
11    setbuffer(file,buf, BUFSIZE);
12  }
13
14  void InstReference(long pc)
15  {
16    pc |= 1L << 63;
17    fwrite(&pc,sizeof(pc),1,file);
18  }
19
20  void DataReference(long address)
21  {
22    fwrite(&address,sizeof(address),1,file);
23  }
24
25  void CloseFile()
26  {
27    fclose(file);
28  }
```

The analysis procedures are very simple. **InstReference** takes the program counter as an argument, sets the high order bit, and writes the modified address to a file. Statement 16 is used to OR the high order bit with a one. Notice that **1L** is used to create the 64 bit constant. **DataReference** writes the effective address directly to the file.

One important detail in this implementation is the call to *setbuffer* in line 11. This is a standard library function that associates the file descriptor *file* with the statically defined buffer defined in statement 4. If this statement is left out, the first call to *fprintf* calls *malloc* to create a output buffer. If the application also calls malloc, the application addresses will be offset by the buffer size. To maintain the exact data addresses, the *setbuffer* call provides statically allocated storage to hold the trace buffer.

As machines speeds increase, the ability to store trace files to disk has almost completely disappeared. The Alvinn SPEC92 benchmark executes 961,082,150 loads, 260,196,942 stores, and 73,687,356 basic blocks, for a total of 2,603,010,614 Alpha instructions. Storing the program counter for each basic block entered, and the effective address of all the loads and stores would take in excess of 10GB, and slow down the application by a factor of well over 100. More effecient cache simulations simulate the cache directly in the analysis procedures.

5. Instrumentation Interface

ATOM provides a rich interface for instrumenting programs. This interface provides support for passing arguments, navigating the executable, asking detailed questions about the application program, and instrumenting the program at any point. There are also a large number of different application values that can be passed to analysis programs, including the integer and floating point registers, effective addresses, branch conditions, instruction fields, cycle counter, and procedure arguments and return values.

5.1. ATOM Command Line

There are many optional arguments to the ATOM command line. Some of the more important ones are shown below.

Option	**Description**
-O	Apply standard linker optimizations
-A1	Applies ATOM optimizations
-dbx	Runs instrumentation under DBX
-g	Add symbolic information for DBX
-heap	Partition the heap
-toolargs	Pass arguments to the Instrument routine
-tool	Run standard tools

A few optimizations are applied by the Alpha AXP OSF1 Linker. To run ATOM on optimized applications, the **-O** switch should also be passed to ATOM. ATOM complete the link time optimizations before instrumenting the code, so that the instructions exactly match the output of the standard linker.

The **-A1** switch is a new ATOM optimization flag. This flag uses sophisticated register allocation techniques to lower the instrumentation overhead. In some tools, this increases tool performance by a factor of 2.

The **-dbx** switch is used to debug instrumentation routines by running the ATOM instrumentation step under the symbolic debugger.

The **-g** compiles and links the instrumented application and retains enough symbol table information to run the application under **DBX**. This process only allows stops to be placed in the user defined analysis procedures. The application is assumed to run correctly.

The **-heap** switch is used to partition the heap. The first section is reserved for the application program. The second is reserved for the analysis procedure. The **-heap** switch requires an argument that specifies the maximum size of the heap in the application program. The analysis heap is defined to start at this point.

The **-atomtools** argument allows atom command line arguments to be passed to directly to the instrumentation routine. The argument list looks identical to the way arguments are passed to the standard **C** programs using the *argc* and *argv* arguments to the *main* program.

```
#include <stdio.h>
#include <instrument.h>

Instrument(int argc, char **argv)
{
  int i;
  fprintf(stderr,"argc:    %d\n",argc);
  for (i = 0; i < argc; i++)
    {
      fprintf(stderr,"argv[%d]: %s\n",argv[i]);
    }
}
```

If this tool was part of the instrumentation for the helloworld application, the **atom** command would print out the following argument list.

```
> atom helloworld.rr tool.inst.c tool.anal.c -toolargs=8192,4 -o helloworld.tool
argc:    2
argv[0]: helloworld
argv[1]: 8192
argv[2]: 4
```

5.2. Application Navigation

This section defines the procedural interface for navigating around an application program. These procedures are used in the *Instrument* routine to find "interesting" places to add procedure calls.

```
Proc *GetFirstProc();
Proc *GetLastProc();
Proc *GetNextProc(Proc *);
Proc *GetPrevProc(Proc *);
Proc *GetBlockProc(Block *);

Block *GetFirstBlock(Proc *);
Block *GetLastBlock(Proc *);
Block *GetNextBlock(Block *);
Block *GetPrevBlock(Block *);
Block *GetInstBlock(Inst *);

Inst *GetFirstInst(Block *);
Inst *GetLastInst(Block *);
Inst *GetNextInst(Inst *);
Inst *GetPrevInst(Inst *);
```

GetFirstProc and **GetLastProc** return pointers to the first and last procedures in the application program. Given a procedure, the **GetNextProc** and **GetPrevProc** procedures return the next and previous procedures. Both return NULL if there are no more procedures. Similar procedures are defined for blocks and instructions.

To move from an instruction back to the enclosing basic block, ATOM provides the **GetBlockProc** procedure. Similarly, the **GetInstBlock** moves from a basic block back to the enclosing procedure.

5.3. Program Operations

ATOM provides static information at the program, procedure, basic block, edge, and instruction level. This section describes the ATOM procedural interface to general program information.

long **GetProgramInfo**(PInfoType type);
unsigned int ***GetProgramInstArray**();
int **GetProgramInstCount**();
char ***GetProgramName**();

GetProgramInfo returns information about the program based on the *type* argument. the type argument.

Type	Description
ProgramNumberInsts	number of instructions in the application.
ProgramNumberBlocks	number of blocks in the application.
ProgramNumberProcs	number of instructions in the application.
TextStartAddress	starting address of the text segment
TextSize	size of the text segment in bytes
InitDataStartAddress	start of the initialized data segment.
InitDataSize	size of the initialized data segment.
UninitDataStartAddress	start of the uninitialized data segment.
UninitDataSize	size of the uninitialized data segment.

GetProgramInstArray returns a pointer to an array of unsigned integers that contains all the instructions in the application program. The following instrumentation example loads the first instruction in the text segment into *firstInstruction*.

unsigned int *textSegment = **GetProgramInstArray**();
unsigned int firstInstruction = textSegment[0];

GetProgramInstCount returns the number of instructions in the text segment array. This number is always greater than or equal to the *ProgramNumberInsts*, since compilers often pad the text segment to force alignments. These instructions cannot be reached during program execution and they are not seen by the standard **GetFirstInst** and **GetNextInst** instrumentation procedures.

5.4. Procedure Operations

ATOM provides a variety of procedures to access information about application procedures and procedure stack frames.

long **GetProcInfo**(Proc *, ProcInfoType type);
char ***ProcName**(Proc *);
char ***ProcFileName**(Proc *);
long **ProcPC**(Proc *);

GetProcInfo returns information about the procedure. Valid types are shown below.

Type	Description
ProcNumberBlocks	number of blocks in the procedure
ProcNumberInsts	number of instructions in the procedure

Detailed information on the stack frame can be obtained using types such as *FrameSize*, *IRegMask*, *IRegOffset*, *FRegMask*, and *FRegOffset*. These values of this types are defined in the DEC OSF1 Assembly Language Programmer's Guide.

ProcName returns the the procedure name. **ProcFileName** takes the same argument, but returns the source file name. This information can be combined with the **InstLineNo** instruction operation procedure described below to locate the file name and line number that of any instructions in the application program. This mechanism is very useful for reporting results that are easy to track down in the source code. **ProcPC** returns the program counter of the first instruction in the procedure.

5.5. Basic Block Operations

ATOM provides a limited number of basic block operations.

```
long GetBlockInfo(Block *, BlockInfoType type);
long BlockPC(Block *);
```

GetBlockInfo returns information about the basic block. Currently the only type supported is *BlockNumberInsts*, which returns the number of instructions in the basic block. **BlockPC** takes a pointer to a basic block and returns the program counter of the first instruction in the basic block.

5.6. Edge Operations: NOT YET IMPLEMENTED

An *edge* is a path into or out of a basic block. *Predecessor* edges are the paths that lead into a basic block. *Successor* edges are the paths out of a basic block.

```
Edge *GetFirstSuccEdge(Block *);
Edge *GetNextSuccEdge(Edge *);
Edge *GetFirstPredEdge(Block *);
Edge *GetNextPredEdge(Edge *);
Block *GetEdgeTo(Edge *);
Block *GetEdgeFrom(Edge *);
```

The **GetFirstPredEdge** returns a pointer to the first incoming edge to the block. The **GetNextPredEdge** returns a pointer to the next incoming edge. Outgoing edges can found using the **GetFirstSuccEdge** and the **GetNextSuccEdge**. A edge always connects two basic blocks. **GetEdgeFrom** returns the block that is executed first, and **GetEdgeTo** returns the block that is executed second.

5.7. Instruction Operations

Instructions are the fundamental unit of computation in an application program and much of the ATOM interface is devoted to parsing instructions and providing the useful syntactic and semantic information.

```
IClassType GetInstClass(Inst *);
int GetInstInfo(Inst *, IInfoType);
int IsInstType(Inst *, enum IType);
RegvType GetInstRegEnum(Inst*, IInfoType);
long InstPC(Inst *);
int GetInstBinary(long pc);
char *GetInstProcCalled(Inst *);
void GetInstRegUsage(Inst *, InstRegUsageVec *);
```

GetInstClass takes an instruction as input and returns the instruction class. Each Alpha instruction is mapped to exactly one instruction class. These classes are fully defined in the 20164 Hardware Reference Manual and can be used directly to determine the dual issue and instruction scheduling rules.

Class	**Description**
ClassLoad	Integer load
ClassFload	Floating point load
ClassStore	Integer store data
ClassFstore	Floating point store data
ClassIbranch	Integer branch
ClassFbranch	Floating point branch
ClassSubroutine	Integer subroutine call
ClassIarithmetic	Integer arithmetic
ClassImultiplyl	Integer longword multiply
ClassImultiplyq	Integer quadword multiply
ClassIlogical	Logical functions
ClassIshift	Shift functions
ClassIcondmove	Conditional move
ClassIcompare	Integer compare
ClassFpop	Other floating point operations
ClassFdivs	Floating point single precision divide
ClassFdivd	Floating point double precision divide
ClassNull	call pal, hw_x etc

IsInstType evaluates to true if the instruction belongs in the instruction subset defined by *type*. These types combine instruction classes into the following commonly used subsets.

Type	**Description**
InstTypeLoad	Load instruction
InstTypeStore	Store instruction
InstTypeCondBr	Conditional branch
InstTypeUncondBr	Unconditional branch

The **GetInstInfo** parses the 32 bit instruction and returns either the entire 32 bit instruction or the appropriate subfield.

Type	**Description**
InstBinary	32 bit binary instruction.
InstOpcode	opcode
InstMemDisp	memory displacement(sign extended to 32 bits)
InstBrDisp	branch displacement(sign extended to 32 bits)
InstRA	register field A
InstRB	register field B
InstRC	register field C

No error checking is done. **GetInstinfo** with a type of *InstMemDisp* returns the lower 16 bits of instruction sign extended to 32 bits, even if the instruction does not reference memory.

Notice that if **GetInstInfo***(inst,InstRA)* returns a 5, the register could either refer to the integer or floating point register files, depending on the opcode of the instruction. *GetInstRegEnum* takes an argument of either *InstRA*, *InstRB*, or *InstRC* and returns the unique enumerated register type. For example, the procedure returns *REG_5* if the instruction is an integer instruction and *FREG_5* if the instruction is a floating point instruction. This distinction is extremely important in later sections, when enumerated types are used to pass the contents of processor registers. If the register requested is not defined for this instruction, *REG_NOTUSED* is returned.

InstPC returns the program counter of the instruction. For instructions in the class *ClassSubroutine*, **GetInstProcCalled** returns the name of the procedure called, or *NULL* for indirect procedure calls.

GetInstBinary returns the 32 bit instruction. This instruction can be used in much the same way as the array returned by **GetProgramInstArray** described above.

GetInstRegUsage is used for data flow analysis. This procedure sets a bit mask with one bit for each possible source register and one bit for each possible destination register. Consider the following example.

```
InstRegUsageVec usageVec;
Inst *inst = GetLastInst(GetFirstBlock(GetFirstProc()));
GetInstRegUsage(inst,&usageVec);
```

This small code fragment sets *inst* to point to the last instruction in the first basic block in the first procedure. Assume the first instruction was an "ADDQ r0,r2,r7" instruction. This instruction adds of the contents of register 0 to the contents of register 2 and places the result in register 7. The value returned in the usageVec.ureg_bitvec[0] is *0x5*, since register 0 and register 2 are both used. The value of usageVec.dreg_bitvec[0] is set to *0x40*, indicating that register 7 is set by the add instruction. Each bit mask contains two words, with one bit position for each of the 32 integer registers, 32 floating point registers, the program counter and the cycle counter.

5.8. Add Call Prototypes

AddCallProto defines the procedural interface to the analysis routines.

```
void AddCallProto(char *define);
```

The format of *define* is similar to a a **C** function definition. The name of the analysis procedure is followed a parenthesized list of arguments. There are three basic types of arguments: constants, register values, and computed values.

Constant types include *char*, *int*, *long*, *char **, *char []*, *int[]*, *long[]*. Passing arrays in ATOM is a simple way to communicate static information to analysis procedures. Although this information could be passed as procedure arguments, arrays are more efficient for large data structures. This example passes an array of program counters, one for each procedure in the application.

```
#include <stdio.h>
#include <instrument.h>

Instrument()
{
  char prototype[100];
  Proc *proc;
  int i = 0;
  array = (long *) malloc(sizeof(long)*GetProgramInfo(ProgramNumberProcs);
  for (proc= GetFirstProc(); proc != NULL; proc = GetNextProc(proc))
    {
      array[i++] = ProcPC(proc);
    }
  sprintf(prototype,"PrintProcPC(int,long[%d])",i);
  AddCallProto(prototype);
  AddCallProgram(ProgramAfter,"PrintProcPC",i,array);
}
```

The array is accessed by the analysis procedure as an ordinary array.

```
#include <stdio.h>

void PrintProcPC(int number, long *array)
{
   long *firstPC = array[0];
   ...
}
```

The *VALUE* type is used to pass values that ATOM must compute before passing the value to the analysis procedures. These values include the effective address of load and store instructions, and conditional branch conditions.

Type	Argument	Description
VALUE	EffAddrValue	Effective address
VALUE	BrCondValue	Branch condition value
REGV	REG_*n*	Integer Register *n*
REGV	FREG_*n*	Floating Point Register *n*
REGV	REG_RA	Compiler Temporary
REGV	REG_GP	Global pointer
REGV	REG_SP	Stack Pointer
REGV	REG_ZERO	Integer Register 31
REGV	REG_CC	Processor cycle counter
REGV	REG_PC	Program Counter at the instrumentation point
REGV	REG_ARG_*n*	Procedure arguments 1 through 4
REGV	RET_RES_1	Function return value

The *REGV* type is used to pass the contents of processor registers. These include integer and floating point registers. Symobolic names are provided for common predefined registers. For example, Alpha keeps the global pointer in integer register 29. Passing the contents of REG_29 is identical to passing REG_GP. The calling standard places the first four arguments to procedures in registers 16 through 19. These values are available by the symbolic names of *REG_ARG_1*, *REG_ARG2*, *REG_ARG3*, and *REG_ARG_4*. The example below passes the first argument to the *open* system call to the the *PrintFileName* analysis procedure.

```
#include <stdio.h>
#include <instrument.h>

Instrument()
{
  Proc *proc = GetNamedProc("open");
  AddCallProto("PrintFileName(REGV)");
  if (proc != NULL)
    {
      AddCallProc(proc,ProcBefore,"PrintFileName",REG_ARG_1);
    }
}
```

Alpha provides fine grain cycle counter at user level through the processor cycle counter that can be used to time short application application events. Unfortunately, this counter is only 32 bits, and therefore wraps every 15 seconds. Below is a short example that accumulates time for the procedure defined in the *atomtools* argument.

```
#include <stdio.h>
#include <instrument.h>

Instrument(int argc, char **argv)
{
   Proc *proc = GetNamedProc(argv[1]);
   if (proc != NULL)
     {
       AddCallProto("Start(REGV)");
       AddCallProto("Stop(REGV)");
       AddCallProc(proc,ProcBefore,"Start",REG_CC);
       AddCallProc(proc,ProcAfter,"Stop",REG_CC);
     }
}
```

The analysis routine is complicated by the format of the cycle counter. The low order 32 bits contain a running cycle count. The high order 32 bits of the counter are an offset that when added to the low-order 32 bits gives the cycle count for this process. It is modifed by PAL code at context switches to contain the correct, per process offset.

```
long total;
long process;
int ccStart;
int ccStartProcess;

void Start(unsigned long cc)
{
  ccStart = (cc << 32) >> 32;
  ccStartProcess = ((cc << 32) + cc) >> 32;
}
void Stop(unsigned long cc)
{
  int ccEnd = (cc << 32) >> 32;
  int ccEndProcess = ((cc << 32) + cc) >> 32;
  total += (unsigned) (ccEnd - ccStart);
  process += (unsigned) (ccEndProcess - ccStartProcess);
}
```

5.9. Adding Procedure Calls

The fundamental instrumentation mechanism is the ability to add arbitrary procedure calls. These calls can be added before and after procedure, basic blocks, edges, instructions, or before and after the program executes.

```
void AddCallProgram(enum Place, char *pname, ... );
void AddCallProc(Proc *, enum Place, char *pname,  ... );
void AddCallBlock(Block *, enum Place, char *pname, ... );
void AddCallEdge(Edge *, char *proc, ...);
void AddCallInst(Inst *, enum Place, char *pname,  ... );
void ReplaceProcedure(Proc *, char *pname);
```

If multiple calls are added to the same point in the program, the calls are ordered by program scope. For example, the instrumentation routine below adds many calls before and after the first instruction in the application program.

```
#include <stdio.h>
#include <instrument.h>

Instrument()
{
  Proc *proc = GetFirstProc();
  Block *block = GetFirstBlock(proc);
  Inst *inst = GetFirstInst(block);

  AddCallProto("Instruction(int)");
  AddCallProto("Block(int)");
  AddCallProto("Procedure(int)");
  AddCallProto("Program(int)");

  AddCallInst(inst,InstBefore,"Inst",1);
  AddCallProc(proc,ProcBefore,"Proc",1);
  AddCallProc(proc,ProcBefore,"Proc",2);
  AddCallInst(inst,InstBefore,"Inst",2);
  AddCallInst(inst,InstAfter,"Inst",3);
  AddCallBlock(block,BlockBefore,"Block",1);
  AddCallProgram(ProgramBefore,"Program",1);
}
```

The analysis procedures will be executed in the order: *Program(1)*, *Proc(1)*, *Proc(2)*, *Block(1)*, *Inst(1)*, *Inst(2)*, the application instruction, and finally *Inst(3)*. By creating a scope hierarchy, ATOM guarantees that the *OpenFile* procedure added at *ProgramBefore* is executed before procedures added in lower scopes. Procedures added at *ProgramAfter* execute after all application instructions have executed. A side affect of this placement is that these analysis procedures cannot write to standard output or standard error, since the application program has already closed these file descriptors.

ATOM preserves the semantics of *AddCall* in situations where the program or procedure has multiple entry or exits. For example, if a procedure has multiple entry points and the instrumentation attempts calls *AddCallProc* at *ProcBefore*, ATOM must instrument each entry points identical procedure call to analysis routines.

5.10. Replacing Procedures

Certain tools must replace procedure call rather than simply monitor the arguments or return values.

```
void ReplaceProcedure(Proc *, char *pname);
```

For example, the Third Degree tool keeps track of all memory that is allocated and free by replacing all calls to dynamic memory allocation routines with special purpose procedures. Here are the statements in the Third Degree instrumentation file that replace the standard *malloc* procedure with a private version called *3rd_malloc*.

```
#include <stdio.h>
#include <instrument.h>

Instrument()
{
   Proc *proc = GetNamedProc("malloc");
   if (proc != NULL)
     {
        ReplaceProc(proc,"3rd_malloc");
     }
}
```

3rd_malloc is defined in the analysis file with the same arguments and return value as the original *malloc*.

6. Analysis Procedures

Once the application program is instrumented, the analysis procedures are used to compute tool specific functions. Analysis procedures are called from the instrumented application programs to perform tool-specific functions.

The analysis procedures can call any system call or library routine. ATOM links in identical but physically distinct copies of each of these routines so that the instrumentation is applied only to the application program. For example, instrumenting the *fprintf* routine in the application has no affect on the *fprintf* used in the analysis routine.

The most important call that is supported by ATOM is the *getenv* system call. This is the primary method of passing parameters to analysis procedures. In the example below, the *OpenFile* analysis procedure appends the process identifier to the file name if the ATOMUNIQUE setenv variable is set.

```
FILE *file;
void OpenFile();
{
  char name[100];
  if (getenv("ATOMUNIQUE") != NULL)
      sprintf(name,"tool.out.%d",getpid());
  else
      strcpy(name,"tool.out");
  file = open(name,"w");
}
```

This option can be used if the user does not want the output files from multiple runs to overwrite the same file. Even in a single run, the output file can be overwritten if the application executes a *fork*, and then an *exec* of the same program. With the ATOMUNIQUE environment variable set, two different output files are created.

If a process executes a *fork*, but does not call *exec* the process is cloned, and the child inherits an exact copy of the parent state. In many cases, this is exactly the behavior that an ATOM tool expects. For example, the *trace* tool opens a file at *ProgramBefore* and the file remains open after the fork. Since both the parent and the child share the same file descriptor, the trace contains both parent and child addresses.

For some tools, the behavior is more interesting. In the case of the instruction profiling tool. The file is opened at *ProgramBefore* and therefore the output file is shared between the parent and the child processes. If the results are printed at *ProgramAfter* the output file contains the parent data, followed by the child data (assuming the parent finished first). For tools that count events, the data structures that hold the counts should be returned to zero in the child process after the fork, since they occured in the parent, not the child. ATOM can support correct *fork* handling, if the *fork* procedure call return value is passed to an analysis routine. Since *fork* returns 0 to a child process, the analysis routine can zero out the childs count data structures prior to execution.

7. Accuracy

ATOM goes to elaborate lengths to present analysis routines with the exact state of the application program without any instrumentation. If the analysis routines substantially change the behavior of the application programs, then the data collected by the analysis procedures would be of very little value. In almost all cases, ATOM can guarantee that the values presented to the analysis routines are exactly the same values that occurred in the uninstrumented application program.

There are a few cases when the analysis program could have minor affects on the behavior of the application program. The most obvious example is if the application program and the analysis program dynamically allocate memory. Even though each calls an independent copy of *sbrk*, the same operating system call is invoked, and the call returns the next block of memory. If the application and analysis routines alternately call malloc, the addresses that are returned to the application will be different than had the application run independently. In many applications, this does not matter. For example, if the tool counts the number of times each procedure is executed, an exact correspondence of dynamic memory locations is not important. If the tool is a data cache simulator, then this is very important.

There are several approaches to avoid this problem. The first is to never dynamically allocate memory in the analysis procedure. This is complicated by the fact that any call to printf in the analysis program makes a library call to the *malloc* procedure, which could slightly change the heap addresses that would otherwise be returned. To avoid calling *malloc* when using the *fopen* system call, a call to *setbuffer* must be inserted after the *fopen* and before the first read or write operation.

Another approach is to pass data structures from the instrumentation to the analysis routines using *AddCallProto* with array arguments. This approach can be used in any tool where the size of the data structure is know statically.

A third approach is to dynamically allocate all the memory at the start of the program. This will shift all the dynamic memory addresses by some constant, but the overall cache behavior is not likely to change.

The final approach is to use the *-heap* option to ATOM, which splits the dynamically allocated space into two sections, one for the application program, and one for the analysis routines. This behavior, although highly desirable, increases considerably the size of the running process, and could cause problems for systems with small amounts of swap space, or for applications that need large amounts of dynamically allocated memory.

Another case where analysis routines have a small affect on application program behavior is file descriptors. These small integers are returned by the operating system in the order that files are opened. If a poorly written application program depends on getting particular values of the file descriptors on each call to open, calling open in the analysis program could change the behavior of the application program. The problem can be avoided if the analysis routine opens the file, then calls the *dup2* system call to duplicate the new file descriptor, the calls close on the original file. Then the next time open is called the operating system will return the correct file descriptor.

Another case where the instrumented application and the original application may have very slightly different behavior is in the setup procedures in the routine __start. This procedure sets up the arguments (argc, argv) and environment variables for the application. Some of the loops in this procedure execute a number of times based on the size of the application program name and the number of environment variables that are set. Thus, you might expect to find very minor differences in the basic block counts if you change the program name from foobar to foobar.prof, or if you have changed any environment variable from one run to the next.

8. Further Examples

In addition to the standard set of performance tools, the distribution contains many simple examples of ATOM based tools. The /usr/lib/atom/tools directory contains the instrumentation and analysis procedures for these simple tools:

- **ptrace**: instruments an application to print out the name of each procedure entered.
- **cache**: instruments an application to simulate execution in a direct mapped 8K byte data cache.
- **malloc**: instruments an application to compile a histogram of calls to malloc and a total count of allocated memory.
- **trace**: instruments an application to trace the starting address of each basic block, and the effective address of each data reference.
- **dyninst**: instruments an application to compute dynamic instruction, load, store, basic block, and procedure counts.
- **pixi2**: instruments an application to count the number of times each basic block is executed
- **prof**: instruments an application to count the number of instructions in each procedure and prints a profile of where the execution time is spent.
- **branch**: instruments an application to compute the branch prediction rate for each conditional branch in the program and an overall branch prediction rate using a 256 entry two bit history table.
- **dtb**: instruments an application to determine the number of dtb misses given 8K byte pages and a fully associative translation buffer.
- **classes**: compute the number of instructions in each of the EV4 instruction classes. This file is computed during instrumentation leaving the application unchanged.
- **inline**: instruments an application to determine potential candidates for procedure inlining.

Appendix I
Appendix: instrument.h

The complete definition of the instrumentation interface is defined in the instrument.h file. Here is the current version of instrument.h.

```
/*
 * @DEC_COPYRIGHT@
 */
/*
 * HISTORY
 * $Log: instrument.h,v $
 * Revision 1.4  1994/05/18  22:18:24  amitabh
 * Added GetProcCalled primitive
 *
 * Revision 1.3  1994/05/17  03:56:09  amitabh
 * Entry Point extension for third degree
 *
 * Revision 1.2  1994/05/13  17:54:47  amitabh
 * Bug fixes
 *
 * $EndLog$
 */
/***************************************************************************/
/*                                                                         */
/*                                                                         */
/*                        THE ATOM SYSTEM                                  */
/*                                                                         */
/*          sources adapted  from the  THE OM LINK-TIME SYSTEM             */
/*                                                                         */
/*                                                                         */
/*                       Copyright (c) 1994-95 by                          */
/*               Digital Equipment Corporation, Maynard, MA                */
/*                       All rights reserved.                              */
/*                                                                         */
/*                                                                         */
/*-------------------------------------------------------------------------*/
/*  Change History:                                                        */
/*                                                                         */
/*  Author:  Amitabh Srivastava                       July 1993            */
/*           Digital Equipment Corporation                                 */
/*           Western Research Laboratory                                   */
/*           Palo Alto, California                                         */
/*                                                                         */
/*                                                                         */
/***************************************************************************/
#ifndef INSTRUMENT_H
#define INSTRUMENT_H

typedef struct procedure Proc;
typedef struct basicblock Block;
typedef struct rtl_inst Inst;
typedef struct edge Edge;

/* Inst Class */

typedef enum IClass {
  ClassLoad,                   /* Integer load */
  ClassFload,                  /* Floating point load*/
  ClassStore,                  /* Integer store data*/
  ClassFstore,                 /* Floating point store data*/
  ClassIbranch,                /* Integer branch*/
  ClassFbranch,                /* Floating point branch*/
  ClassSubroutine,             /* Integer subroutine call*/
  ClassIarithmetic,            /* Integer arithmetic*/
  ClassImultiplyl,             /* Integer longword multiply*/
  ClassImultiplyq,             /* Integer quadword multiply*/
  ClassIlogical,               /* Logical functions*/
```

```
  ClassIshift,                  /* Shift functions*/
  ClassIcondmove,               /* Conditional move*/
  ClassIcompare,                /* Integer compare*/
  ClassFpop,                    /* Other floating point operations*/
  ClassFdivs,                   /* Floating point single precision divide*/
  ClassFdivd,                   /* Floating point double precision divide*/
  ClassNull                     /* call pal, hw_x etc*/
  } IClassType;

/* InstType */
/* this is obsolete  -- please use Inst Class, it is much better */

typedef enum IType {
  InstTypeLoad,
  InstTypeStore,
  InstTypeJump,
  InstTypeFP,
  InstTypeInt,
  InstTypeDiv,
  InstTypeMul,
  InstTypeAdd,
  InstTypeSub,
  InstTypeCondBr,
  InstTypeUncondBr
  } ITypeType;

/* Instruction Information */

typedef enum IInfo{
  InstMemDisp,
  InstBrDisp,
  InstRA,
  InstRB,
  InstRC,
  InstOpcode,
  InstBinary,
  InstAddrTaken,
  InstEntryPoint
} IInfoType;

/* Block Information */

typedef enum BlockInfo{
   BlockNumberInsts
} BlockInfoType;

/* Procedure Information */
#define PROCEDURE_FRAMESIZE(x) (x->framesize)
#define PROCEDURE_RETURN_REG(x) (x->return_reg)
#define PROCEDURE_FRAME_REG(x) (x->frame_reg)
#define PROCEDURE_ISAVED_REGS(x) (x->isaved)
#define PROCEDURE_FSAVED_REGS(x) (x->fsaved)
#define PROCEDURE_ISAVED_OFFSET(x) (x->ioffset)
#define PROCEDURE_FSAVED_OFFSET(x) (x->foffset)


typedef enum ProcInfo{
   FrameSize,
   IRegMask,
   IRegOffset,
   FRegMask,
   FRegOffset,
   gpPrologue,
   gpUsed,
   LocalOffset,
   FrameReg,
   PcReg,
   ProcNumberBlocks,
   ProcNumberInsts
```

```
} ProcInfoType;

/* Program Information */

typedef enum PInfo{
   TextStartAddress,
   TextSize,
   InitDataStartAddress,
   InitDataSize,
   UninitDataStartAddress,
   UninitDataSize,
   ProgramNumberBlocks,
   ProgramNumberProcs,
   ProgramNumberInsts
} PInfoType;


/* Registers */

/* use type REGV in prototypes to pass these as arguments */

typedef enum Regs{
  REG_NOTUSED = -1,
  REG_0,  REG_1,  REG_2,  REG_3,  REG_4,  REG_5,  REG_6,  REG_7,
  REG_8,  REG_9,  REG_10, REG_11, REG_12, REG_13, REG_14, REG_15,
  REG_16, REG_17, REG_18, REG_19, REG_20, REG_21, REG_22, REG_23,
  REG_24, REG_25, REG_26, REG_27, REG_28, REG_29, REG_30, REG_31,
  FREG_0, FREG_1, FREG_2, FREG_3, FREG_4, FREG_5, FREG_6, FREG_7,
  FREG_8, FREG_9, FREG_10,FREG_11,FREG_12,FREG_13,FREG_14,FREG_15,
  FREG_16,FREG_17,FREG_18,FREG_19,FREG_20,FREG_21,FREG_22,FREG_23,
  FREG_24,FREG_25,FREG_26,FREG_27,FREG_28,FREG_29,FREG_30,FREG_31,
  REG_PC, REG_CC
  } RegvType;

#define REG_MAX REG_CC

#define REG_RA    REG_26
#define REG_GP    REG_29
#define REG_SP    REG_30
#define REG_ZERO  REG_31
#define REG_ARG_1 REG_16
#define REG_ARG_2 REG_17
#define REG_ARG_3 REG_18
#define REG_ARG_4 REG_19
#define RET_RES_1 REG_0

/* Values */

/* use type VALUE in prototypes to pass these as arguments */

typedef enum Value{
   BrCondValue,       /*  valid only for conditional branches */
   EffAddrValue       /*  effective address of load and store instructions */
} ValueType;
/*  Potential placements of calls to user routines */

typedef enum Place {
  ProgramBefore,
  ProgramAfter,

  ProcBefore,
  ProcAfter,

  BlockBefore,
  BlockAfter,

  InstBefore,
  InstAfter
  } PlaceType;
```

```
Proc *GetFirstProc(void);
Proc *GetLastProc(void);
Proc *GetNextProc(Proc *);
Proc *GetPrevProc(Proc *);


Block *GetFirstBlock(Proc *);
Block *GetLastBlock(Proc *);
Block *GetNextBlock(Block *);
Block *GetPrevBlock(Block *);

Inst *GetFirstInst(Block *);
Inst *GetLastInst(Block *);
Inst *GetNextInst(Inst *);
Inst *GetPrevInst(Inst *);

/* Program */

char *GetProgramName(void);
char *GetOutName(void);
char *GetAnalName(void);
long GetProgramInfo(PInfoType type);
unsigned int *GetProgramInstArray(void);
int GetProgramInstCount(void);

/* Procedure */
Proc *GetNamedProc(char *);
char *ProcName(Proc *);
char *ProcFileName(Proc *);
long  ProcPC(Proc *);
long GetProcInfo(Proc* p, ProcInfoType type);

/* Block */
Proc *GetBlockProc(Block *);
int   IsBranchTarget(Block *);
int   IsFallThrough(Block *);
Inst *GetInstBranchTarget(Inst *);
long  BlockPC(Block *);
long GetBlockInfo(Block* b, BlockInfoType type);

/* Edge  */
Edge *GetFirstSuccEdge(Block *);
Edge *GetNextSuccEdge(Edge *);
Edge *GetFirstPredEdge(Block *);
Edge *GetNextPredEdge(Edge *);

/* Inst */
Block *GetInstBlock(Inst *);
int    IsInstType(Inst *, ITypeType);
int    GetInstInfo(Inst *, IInfoType);
long   InstPC(Inst *);
long   InstLineNo(Inst *);
char *GetInstProcCalled(Inst *);
Proc *GetProcCalled(Inst *);

IClassType GetInstClass(Inst *i);

int GetInstBinary(long pc);

RegvType GetInstRegEnum(Inst*, IInfoType); /* InstRA InstRB and InstRC */

/* Adding Procedure Calls */

void AddCallProto(char *);
void AddCallProgram(PlaceType, char *pname, ... );
void AddCallProc(Proc *, PlaceType, char *pname,  ... );
void AddCallBlock(Block *, PlaceType, char *pname, ... );
void AddCallInst(Inst *, PlaceType, char *pname,  ... );
void AddCallEdge(Edge *, char *proc, ...);
```

```
/*  Reg usage masks */

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))

void GetInstRegUsage(Inst *, InstRegUsageVec *);

/* Experimental Features */

void ReplaceProcedure(Proc *, char *);
#endif  /* INSTRUMENT_H */
```

Table of Contents