JULY 1994

WRL
Technical Note TN-44

ATOM:
A Flexible Interface for Building High Performance Program Analysis Tools

Alan Eustace
Amitabh Srivastava

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

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`JOVE::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.pa.dec.com`
UUCP:	`decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

ATOM: A Flexible Interface for Building High Performance Program Analysis Tools

Alan Eustace
Amitabh Srivastava

July 1994

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

Abstract

Code instrumentation is a powerful mechanism for understanding program behavior. Unfortunately, code instrumentation is extremely difficult, and therefore has been mostly relegated to building special purpose tools for use on standard industry benchmark suites.

ATOM (Analysis Tools with OM) provides a very flexible and efficient code instrumentation interface that allows powerful, high performance program analysis tools to be built with very little effort. This paper illustrates this flexibility by building five complete tools that span the interests of application programmers, computer architects, and compiler writers.

The first tool reports the number of bytes read by the application. The second tool is an instruction profiler that computes the number of instructions executed in each procedure as a percentage of the total number of instructions executed. The third tool simulates the execution of the application running in a direct mapped data cache and reports hit and miss data. The fourth tool computes the total amount of memory allocated and deallocated by the application. The final tool isolates potential compiler performance bugs. Each tool is written in between 24 and 60 lines of code.

This flexibility does not come at the expense of performance. Because ATOM uses procedure calls as the interface between the application and the analysis routines, the performance of each tool is similar to or greatly exceeds the best known hand-crafted implementations.

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 determine 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[13], Epoxie[22] and QPT[11]. The second class consists of data and instruction address tracing tools. Pixie and QPT can also generate address traces. They communicate these traces to analysis routines through inter-process communication. Tracing on the WRL Titan[3] communicated with analysis routines using shared memory, but this required operating system modifications. MPTRACE [6] is 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] uses instruction level simulation to selectively generate traces. This technique offers considerable flexibility at the expense of much lower performance.

The important features that distinguish ATOM[18, 15, 16] 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 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 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.

- ATOM tool overhead is proportional to the complexity of the underlying analysis. Many interesting tools can be built that have little or no impact on application performance.

- Even though the analysis routines run in the same address space as the application, precise information about the application is presented to analysis routines at all times.

- As ATOM works on object modules, it is independent of compiler and language systems.

To illustrate the power and flexibility of this approach, this paper fully implements a variety of custom program analysis tools, including input/output, instruction profiling, cache simulation, dynamic memory allocation, procedure inlining profile driven optimizations, and evaluating the quality of compiled code. None of these tools takes more than 60 lines of code to implement. These tools form the basis of many of the tools that are distributed as part of the standard ATOM distribution.

To illustrate the performance of these tools, each was applied to the SPEC92 tool suite. The instrumented application times are compared to the uninstrumented applications using wall clock times.

2 Implementation of ATOM

ATOM is built using OM[19, 20, 21], a link-time code modification system.

Figure 1 describes this process. First, the OM generic object modification library is linked with a tool specific *instrumentation* file to produce a custom instrumenting tool. This program reads in the user application, and modifies it by adding calls to tool specific analysis procedures. ATOM completes the process by linking the instrumented application with the tool specific *analysis* file. The output of ATOM is a custom instrumented application executable that is run in exactly the same manner as the original application.

3 A Simple Example

From a user perspective, applying an ATOM tool to an application is done by executing a command such as

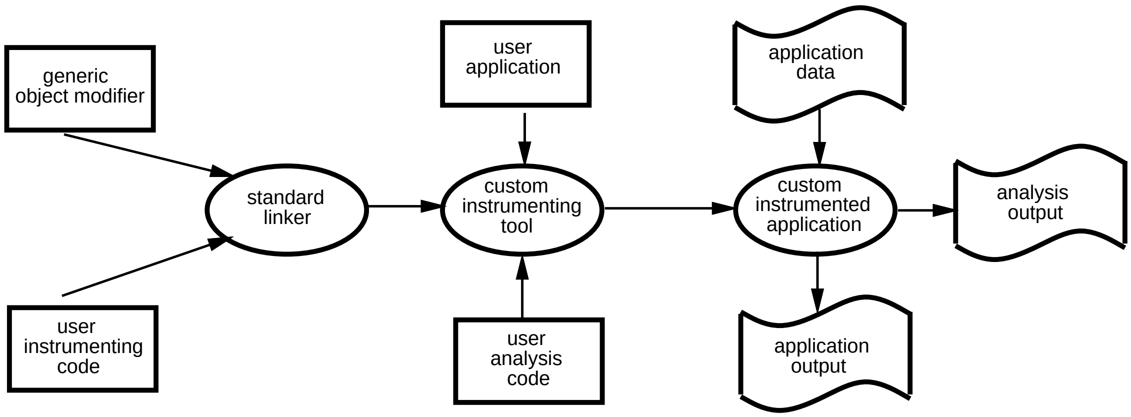

Figure 1: The ATOM Process

```
atom appl.rr read.inst.c read.anal.c -o appl.read
```

The first argument is the application program, which has been specially linked to include relocation records. The second and third arguments are the *instrumentation* and *analysis* files. In this example, we instrument the application to count and write to a file the total number of bytes read each time the instrumented application is executed.

The instrumentation file is shown of the left side of Figure 2. Line 2 includes the *instrument.h* file which defines the ATOM primitives for manipulating application programs. Line 3 defines the `Instrument` procedure, which is linked with the OM object code modification library to produce a custom instrumenting tool. All analysis procedures that are called from the application program are declared and placed by the `Instrument` procedure.

Line 5 makes use of the `AddCallProto` ATOM primitive to declare the name and arguments to the `RecordRead` analysis procedure. This procedure takes a single argument of type `REGV`. Arguments of type `REGV` are used to pass the contents of a specific processor register. Line 6 declares the `PrintResult` analysis procedure, which does not take any arguments.

Line 7 calls the `GetNamedProc` primitive to return a pointer to the `read` procedure. Line 8 checks to see if this value is `NULL`, indicating that the procedure `read` is not defined in this application. All communication between the application program and the analysis procedures is done through procedure calls.

Instrumentation File

```
1  #include <stdio.h>
2  #include <instrument.h>
3  Instrument() {
4     Proc *p;
5     AddCallProto("RecordRead(REGV)");
6     AddCallProto("PrintResults()");
7     p = GetNamedProc("read");
8     if (p != NULL) {
9        AddCallProc(p,ProcBefore,"RecordRead", REG_ARG_3);
10       AddCallProgram(ProgramAfter,"PrintResults");
11    }
12 }
```

Analysis File

```
#include <stdio.h>
long bytes = 0;
void RecordRead(long size) {
   bytes = bytes + size;
}
void PrintResults() {
   FILE *file = fopen("read.out","w");
   fprintf(file, "%ld\n", bytes);
   fclose(file);
}
```

Figure 2: Read Tool Implementation

Line 9 uses the ATOM primitive `AddCallProc` to add a call to the `read` procedure. The first argument, `p` is a pointer to the `read` procedure. The second argument, `ProcBefore`, specifies that the call is to be inserted before the `read` procedure is executed. The third argument indicates that the call is to the `RecordRead` analysis procedure. The remainder of the arguments are used to determine what values ATOM passes to `RecordRead`. In this case, the final argument passes the contents of the register `REG_ARG_3` to the analysis procedure. In the Alpha AXP calling convention, this register contains the contents of the third argument to the `read` procedure. The `RecordRead` procedure is shown on the right side of Figure 2. This procedure simply adds this size to a total.

Line 10 calls the `AddCallProgram` primitive, which adds a call to the `PrintResults` procedure after the application finishes executing. The corresponding analysis procedure opens a file, prints out the result, and closes the file. The definitions for both these procedures are shown on the right side of Figure 2.

Notice that it is important that the analysis procedure does not call the instrumented version of the `read` procedure, since reads that occur inside the analysis procedure must not increment the application totals. To guarantee this, library procedures that would normally be shared between the application and the analysis procedures are linked into the instrumented application twice. Only the version that is linked into the application is instrumented. This guarantees that calls made to `read` by the analysis procedures do not influence the statistics gathered by the read tool.

Although this tool is relatively simple, it is straightforward to extend the read tool into a general input/output tool. The first extension is to add calls to analysis procedures before and after for `open` system calls. This allows the analysis procedures to record the name of the file opened and the file descriptor returned. By instrumenting both read and write procedures and passing the first (file descriptor) and third arguments (size in bytes), the read and write totals can be accumulated for each open file. The final extension is to use the Alpha AXP cycle counter to maintain fine grain times of how long each operation takes. This allows the tool to determine the rate of read and write operations. This extended tool is called *io* and is distributed with ATOM as part of the standard tool set.

4 ATOM Primitives

ATOM tools traverse an application, find interesting places to add calls to analysis procedures, and pass arguments that correspond to data or events in the application. To provide these functions, ATOM provides three types of primitives: *navigation*, *information*, and *instrumentation*.

Navigation primitives traverse the application. The simple example presented above used the `GetNamedProc` primitive to find a specific procedure. Other navigational primitives traverse procedures, basic blocks within procedures, and instructions within basic blocks. A *basic block* is a set of sequential assembly language instructions that are not interrupted by branch or jump instructions.

Information primitives provide static information about instructions, basic blocks, procedures, or the program. For example, given an instruction, ATOM primitives can return the program counter, the opcode, the instruction class, address displacements, the source line number, a mask of the registers used or set by the instruction, etc. Given a basic block, primitives are provided to find the number of instructions in the basic block and the starting program counter of the block. Given a procedure, primitives are provided to find the file name, stack frame size, register save and restore masks, etc. General program information includes the sizes of text and data sections, along with general statistics on the number of procedures, basic blocks and procedures in the application.

Instrumentation primitives allow calls to analysis procedures to be inserted into the application before or after instructions, basic blocks, procedures. The arguments to these procedures can include any value computed by the instrumentation routine or provided by ATOM primitives. The arguments of these procedures can be constants, processor registers, effective addresses, branch condition values, arguments to application procedures, file names, line numbers, or character strings.

Although not shown in these examples, ATOM also allows command line arguments to be passed to instrumentation routines. Parameters can also be passed to analysis procedures through *setenv* variables.

5 Instruction Profiling

In this section we implement an instruction profiler based on counting the number of instructions executed in each procedure. Although it is possible to implement this tool by placing a call to an analysis procedure before every instruction in the application, ATOM's selective instrumentation can significantly reduce this overhead by instrumenting only basic blocks. For example, if a set of 10 sequential executed instructions are inside of a loop, we can keep track of the total number of instructions executed by adding 10 each time we enter the loop body.

Figure 3 defines the instrumentation and analysis files for the profile tool.

As in the previous section, lines 6 through 9 of the instrumentation file declares the interface to the `OpenFile, ProcedureCount, ProcedurePrint`, and `CloseFile` analysis procedures.

In line 10, the `AddCallProgram` primitive is used to add a call to `OpenFile` before the application begins execution. The `GetProgramInfo` ATOM primitive, when passed the `ProgramNumberProcs` argument, returns the number of procedures in the application. The corresponding analysis procedure uses this argument to allocate sufficient memory to accumulate a count for each procedure in the application.

Lines 12 through 21 navigate each procedure in the application. Within each procedure, lines 14 through 18 process each basic blocks. Line 16 calls the `AddCallBlock` ATOM primitive to add a call to the `ProcedureCount` analysis procedure. The two arguments passed are a procedure index `n`, and the number of instructions in the basic block. This value is returned by the `GetBlockInfo` primitive. The corresponding analysis procedure uses these arguments to increment the number of instructions executed by this procedure.

For each procedure in the application, line 19 adds a call to the `ProcedurePrint` analysis procedure. `ProcedurePrint` is passed the unique procedure index and the name of the procedure. This name is returned by the `ProcName` ATOM primitive. The corresponding analysis file uses these two parameters to determine if the procedure was executed, and if so, prints the procedure name, number of instructions, and percentage of instructions executed in this procedure to a file. Notice that the effect of line 19 is to add hundreds of calls to analysis

Instrumentation File

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

Analysis File

```
#include <stdio.h>
long instrTotal;
long *instrPerProc;
FILE *file;
void OpenFile(int n) {
  instrPerProc = (long *) malloc(sizeof(long) * n);
  file = fopen( “prof.out”, “w”);
  fprintf(file, “%30s %15s %10s\n”, “Procedure”,
        “Instructions”, “Percentage”);
}
void ProcedureCount(int n, int instructions) {
  instrTotal += instructions;
  instrPerProc[n] += instructions;
}
void ProcedurePrint(int n, char *name) {
  if (instrPerProc[n] > 0)
    fprintf(file, “%30s %15ld %9.3f\n”, name,
    instrPerProc[n], 100.0 * instrPerProc[n] / instrTotal);
}
void CloseFile() {
  fprintf(file, “\n%30s %15ld\n”, “Total”, instrTotal);
  fclose(file);
}
```

Figure 3: Profiling Tool Implementation

procedures to the end of the program, each with a different index and character string.

Line 22 adds a call to the `CloseFile` analysis procedure after the application completes executing.

Although this is a very simple profiling tool, many more interesting tools can be built using the same principles. Russell Kao built an ATOM based version of the popular tool *gprof*. This tool adds procedure calls at the start of each procedure to push the name of the procedure on a procedure call stack. This stack is popped by adding a similar analysis procedure call to the procedure exit. Gprof reports the percentage of time spent in a procedure and the procedures descendants. The instrumentation procedure was also expanded to use the Alpha AXP dual issue rules to compute cycles rather than instructions executed.

Many other profile based tools have also been developed. One such tool records the value of the Alpha AXP cycle counter at the start of the procedure and at the end of the procedure and computes the wall clock time spent in each procedure.

6 Cache Simulator

Processor cycle times are getting faster at a much greater rate than main memory access times. This disparity has led computer architects to place a subset of main memory into one or more levels of fast, expensive *cache memory* [9]. The effectiveness of this technique is application dependent. Applications that reference the same address multiple times or that use nearby data items benefit most from the data cache.

Although it is clear that cache memory plays an increasingly important role in application performance, measuring cache performance has been relegated to a few industrial and university research reports. Almost all of these studies have focused primarily on the performance of the SPEC92 benchmark suite.

This section presents a simple tool that simulates the execution of the application running in a 64K-byte direct mapped data cache with 32-byte blocks. The tool computes the total number of data cache references, the number of misses, and the *miss rate*. The miss rate is the number of misses divided by the number of references.

The strategy used in this tool is to instrument all load and store instructions with a call to an analysis procedure called `Reference` which is passed the effective address. This effective address is used to simulate the application running in the cache. The cache tool implementation is shown in Figure 4.

Line 5 of the instrumentation file declares the `Reference` analysis procedure. The type `VALUE` indicates that the argument does not live in a processor register, but must be computed by ATOM prior to passing the value to the analysis procedure. Lines 11 through 17 examine each instruction. Lines 13 and 14 determine if the instruction is a load or a store. If so, the `AddCallInst` ATOM primitive adds a call to instruction `i`. The `InstBefore` argument adds the call before the instruction. The name of the analysis procedure to call is `Reference`, and the argument passed is the `EffAddrValue`, which ATOM computes by adding the contents of the base register plus the sign extended displacement. Line 20 completes the tool by adding a call to the `PrintResults` procedure after the application completes execution.

The analysis procedure is shown on the right side of Figure 4. This is a simple implementation of a direct mapped cache. Lines 4 defines the `cache` data structure, which is used to hold a cache tag for each 32 byte block in the cache. Line 5 defines the `reference` and `miss` counters. Lines 7 through 9 compute the cache tag and index, and line 11 probes the cache. If the tags do not match, a miss is recorded in line 12, and the tag is updated in line 13. In either case, the number of references is incremented.

Instrumentation File

```
#include <stdio.h>
#include <instrument.h>
Instrument() {
  Proc *p; Block *b; Inst *i;
  AddCallProto("Reference(VALUE)");
  AddCallProto("PrintResults()");
  for (p = GetFirstProc(); p != NULL;
        p = GetNextProc(p)) {
    for (b = GetFirstBlock(p); b != NULL;
          b = GetNextBlock(b)) {
      for (i = GetFirstInst(b); i != NULL;
            i = GetNextInst(i)) {
        if (IsInstType(i,InstTypeLoad) ||
            IsInstType(i,InstTypeStore))
          AddCallInst(i,InstBefore,
              "Reference", EffAddrValue);
      }
    }
  }
  AddCallProgram(ProgramAfter, "PrintResults");
}
```

Analysis File

```
#include <stdio.h>
#define CACHE_SIZE 65536
#define BLOCK_SHIFT 5
long cache[CACHE_SIZE >> BLOCK_SHIFT];
long references, misses;
void Reference(long address) {
  int index =
      address & (CACHE_SIZE-1)) >> BLOCK_SHIFT;
  long tag = address >> BLOCK_SHIFT;
  if (cache[index] != tag) {
    misses++;
    cache[index] = tag;
  }
  references++;
}
void PrintResults() {
  FILE *file = fopen("cache.out", "w");
  fprintf(file, "%ld %ld %f\n",
    references, misses, 100.0 * misses / references);
  fclose(file);
}
```

Figure 4: Cache Tool Implementation

To guarantee that the results properly reflect the reference pattern of the uninstrumented program, ATOM guarantees that all data items referenced in the original program are placed in exactly the same locations when the program is instrumented. To guarantee this accuracy for instruction cache simulations, ATOM converts all references to the program counter to those of the uninstrumented program before passing the contents to the analysis procedures.

Although the number of hits and misses is useful to computer architects, this information has rarely been presented in a form that is useful to application programmers. By combining the instruction profile tool shown in the previous section with the cache modeling tool shown above, ATOM can create a hybrid tool that shows cache misses in a profile like format. This tool is called *memsys* and it is included with the standard ATOM distribution.

Instrumentation File

```
1  #include <stdio.h>
2  #include <instrument.h>
3  void Instrument() {
4     Proc *procMalloc =
5                 GetNamedProc( “malloc”);
6     Proc *procFree = GetNamedProc( “free”);
7     AddCallProto( “PrintResults()”);
8     if (procMalloc)
9        ReplaceProcedure(procMalloc, “my_malloc”);
10    if (procFree)
11       ReplaceProcedure(procFree, “my_free”);
12    AddCallProgram(ProgramAfter, “PrintResults”);
13 }
```

Analysis File

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  long totalMalloc, totalFree = 0;
4  char *my_malloc(size_t size) {
5     size_t *mptr = (long *) malloc(size+sizeof(long));
6     totalMalloc += size;
7     mptr[0] = size;
8     return ((void *) &mptr[1]);
9  }
10 my_free(void *ptr) {
11    size_t *mptr = ptr;
12    size_t size = mptr[-1];
13    totalFree += size;
14    free(&mptr[-1]);
15 }
16 void PrintResults() {
17 FILE *file = fopen( “dyn.out”, “w”);
18 fprintf(file, “%ld %ld\n”, totalMalloc, totalFree);
19 fclose(file);
20 }
```

Figure 5: Dynamic Memory Tool Implementation

7 Monitoring Dynamically Allocated Memory

Many programs make extensive use of dynamically allocated memory. Such memory is typically allocated using the `malloc` system call, and deallocated using the `free` system call. These procedures are called thousands of times by application programs, allocating, deallocating, and reallocating the same piece of memory many times. This section presents a tool that computes the total number of bytes allocated and freed over the course of the application's execution.

The implementation is shown in Figure 5.

Lines 4 through 6 of the instrumentation file are used to search for procedures with the names `malloc` and `free`. If these procedures are present in the application, these library functions are replaced in lines 6 and 7 by the procedures `my_malloc` and `my_free`. The `ReplaceProcedure` semantics require the type and arguments of the new procedure to be identical to the original procedure calls.

The analysis procedures prepend the size of allocated objects to each dynamically allocated element. Line 5 calls the standard version of `malloc`, but requests additional memory to prepend the object size. Line 6 adds this size to the total amount of allocated memory. Line 7 saves this size in the first location in the dynamically allocated memory. The pointer to the start of the requested memory is returned in line 8. Each call to `free` was replaced in the application by a call to `my_free`. In line 12, this procedure uses a negative index to access the size of the object, which it adds to the total amount deallocated by the application. Line 14 calls the standard free procedure to deallocate the memory.

The ability to replace procedures and monitor data references is fundamental to an emerging set of tools that monitor allocations, deallocations and references to memory[8]. Jeremy Dion and Louis Monier[14] recently completed an ambitious ATOM based tool called Third Degree, that finds and reports many kinds of reads of uninitialized memory, reads and writes to unallocated memory, array bound errors, and freeing the same object more than once. The technique used is to replace all calls to allocate and free library procedures with versions that keep track of the ranges of valid heap locations. Symbolic interpretation in the instrumentation procedures is used to significantly reduce the number of memory references that must be instrumented. The result is a very effective and efficient tool for testing the validity of memory operations. This tool is also included in the standard ATOM distribution.

8 Compiler Auditing

Modern compilers implement a long list of optimizations: loop unrolling, reductions in strength, software pipelining, global register allocation, instruction rearrangement. Unfortunately, these techniques are complicated and interact in non-trivial ways. The resulting code often misses simple optimizations. Tools that evaluate the quality of the compiled code and isolate potential performance problems are called *compiler auditors*[12].

This section presents a simple compiler auditing tool that adds a procedure call before each load instruction to save the contents of the destination register. Another procedure call is added after each load instruction that checks to see if the destination register was modified by the instruction. If not, the instruction loaded a value that was already in the register. These loads are termed *redundant*.

The implementation is shown in Figure 6.

This tool is similar to previous tools, with the exception of lines 17 through 24. Line 17 checks if the instruction is a load operation. If so, line 18 adds a call to the `SaveLoad` procedure before the instruction and passes the contents of the destination register, as returned by the `GetInstRegEnum` ATOM primitive. Line 20 adds a matching call to `CheckLoad` after the

Instrumentation File

```
#include <stdio.h>
#include <instrument.h>
Instrument() {
  Proc *p; Block *b; Inst *i;
  int n = 1;
  AddCallProto(“OpenFile(int)”);
  AddCallProto(“SaveLoad(REGV)”);
  AddCallProto(“CheckLoad(int,long)”);
  AddCallProto(“Print(int,long)”);
  AddCallProto(“CloseFile()”);
  for (p = GetFirstProc(); p != NULL;
      p = GetNextProc(p)) {
    for (b = GetFirstBlock(p); b != NULL;
        b = GetNextBlock(b)) {
      for (i = GetFirstInst(b); i != NULL;
          i = GetNextInst(i)) {
        if (IsInstType(i,InstTypeLoad)) {
          AddCallInst(i,InstBefore, “SaveLoad”,
            GetInstRegEnum(inst,InstRA));
          AddCallInst(i,InstAfter, “CheckLoad”,
            n, GetInstRegEnum(inst,InstRA));
          AddCallProgram(ProgramAfter, “Print”,
            n++, InstPC(i));
        }
      }
    }
  }
  AddCallProgram(ProgramBefore, “OpenFile”,n);
  AddCallProgram(ProgramAfter, “CloseFile”);
}
```

Analysis File

```
#include <stdio.h>
struct Work {
    long count;
    long wasted;
  } *work;
FILE *file;
void OpenFile(int n) {
  work = (struct Work *)
    malloc(sizeof(struct Work) * n);
  file = fopen( “work.out”, “w”);
  fprintf(file, “%11s %11s %11s\n”,
    “PC”, “Count”, “Wasted”);
}
void CloseFile() {
  fclose(file);
}
long value;
void SaveLoad(long val) {
  value = val;
}
void CheckStore(int n,long val) {
  work[n].count++;
  if (value == val) work[n].wasted++;
}
void Print(int n, long pc) {
  if (work[n].wasted != 0)
    fprintf(file, “0x%9lx %11ld %11ld\n”,
      pc, work[n].count, work[n].wasted);
}
```

Figure 6: Compiler Auditing Implementation

load instruction. The arguments are a unique index of the load instruction, and the new contents of the destination register. `CheckLoad` compares this value to the value saved by the `SaveLoad` analysis procedure and increments the appropriate counters. The output file contains a count of the number or redundant times each load is executed along with the number of times the load was redundant.

Redundant loads can be caused by redundant data, and therefore may not be indicative of potential performance bugs. This is the case in the SPEC92 benchmark *hydro2d* where an amazing 42 percent of the loads are redundant. Often compiler optimizations can detect loop invariant instructions and unnecessary spilling and restoring or registers. In one very early version of the compiler, this tool found 8 identical sequential load instructions from the same memory location to the same destination register!

Benchmark	DynMem	Read	Profile	Cache	Audit
alvinn	0.987	0.989	3.910	8.709	11.929
compress	1.007	0.980	4.143	9.367	7.524
doduc	0.990	1.008	2.915	8.300	12.053
ear	1.011	1.005	5.753	6.609	9.403
eqntott	0.995	0.997	4.177	8.117	10.841
espresso	1.050	1.006	8.919	10.664	14.490
fpppp	0.994	0.994	1.761	12.972	21.832
gcc1	1.016	1.006	5.654	8.596	10.540
hydro2d	0.987	0.991	2.403	7.422	10.512
li	1.021	1.054	6.204	11.059	12.483
mdljdp2	0.996	1.000	2.926	4.325	5.198
mdljsp2	1.020	1.027	3.183	6.113	9.367
nasa7	0.997	1.002	1.694	8.193	11.378
ora	0.931	0.931	4.221	7.883	11.652
sc	1.033	1.030	6.014	6.961	8.504
spice	1.012	1.018	3.929	7.277	9.545
su2cor	0.985	0.992	2.457	7.730	9.641
swm256	0.987	0.990	1.472	9.540	13.956
tomcatv	0.998	0.990	1.821	6.401	13.555
wave5	0.995	0.993	3.146	9.284	10.690
Average	1.000	1.004	3.613	8.269	11.230

Figure 7: Performance of Atom Tools

9 Performance

The performance of ATOM tools is a function of the number of analysis procedure calls that are executed and the amount of work done by each call. Figure 7 shows the performance of each tool over the SPEC92 benchmark suite. Each entry reflects the wall clock time of the instrumented program divided by the wall clock time of the uninstrumented program.

The Dynamic Memory and Read tools have a minimal affect on application performance, since both have relatively few instrumentation points. Contrast this with the compiler auditing tool, which adds two calls to analysis procedures for each load instruction. Also notice that there is considerable variation between benchmarks for a single tool. For example, the profile tool slows down application by as little as 1.472 for *swm256* and as much as 8.919 for *espresso*. Both instrument at basic blocks, but since the basic block size of espresso is much smaller, the instrumented application spends a larger percentage of time in the analysis procedures.

When comparing these times to other tools reported in the literature, it is important to include the time necessary to gather the data *and* to analyze the results. For example, many cache instrumentation tools studies report competitive times for gathering trace data into in-memory buffers, but do not include the times to empty the buffer, simulate the cache, and report the results.

There are many ways to substantially increase the performance of ATOM based tools. One approach is to reduce or eliminate the analysis procedure call overhead either through inlining or other compiler optimizations. Another approach is to make use of the flexibility of the instrumentation interface to reduce the frequency of analysis procedure calls. For example, the profile tool instrumentation routine can be easily rewritten to eliminate adding calls to analysis procedures for those blocks where data flow analysis determines that the count is identical to another block that has already been instrumented. Another example is instruction translation buffer simulation. Here, ATOM based tools need only instrument branches or sequential execution that crosses page boundaries. Since these are relatively infrequent, these tools are very efficient. Another example are tools that simulate branch prediction algorithms. Rather than infer branch behavior by sifting through instruction address traces, ATOM tools instrument only conditional branches.

10 Conclusions

ATOM is a unique tool for understanding program performance. The flexible interface allows a diverse set of tools to be built with minimal effort. Without the support ATOM provides, these tools would be extremely difficult to build. The performance of these tools compares favorably with hand-crafted implementations, since instrumentation is inserted only when necessary to gather statistics. Communication of data to the analysis procedures is accomplished through procedure calls, rather than relying on expensive interprocess communication. The analysis routines are always presented with information about the application program as if it was executing uninstrumented.

ATOM has been applied to many commercial applications with text sizes of up to 100MB. Hundreds of tools have been written by both industrial and university users to evaluate the performance of caches, garbage collection algorithms, branch prediction, compiler optimizations, input/output, system calls, novel CPU architectures, as well as many other aspects of system performance. Currently we are in the process of extending ATOM to be able to instrument the OSF1 kernel.

11 Acknowledgments

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 internal users, and Dirk Grunwald and Brad Calder provided our first external field test site. Jeremy Dion, Ramsey Haddad, Russell Kao, Greg Lueck, Mike McCallig, and Louis Monier contributed exciting new tools. Many, many others, provided help, support, encouragement, bug reports, flames, and endorsements! Also, Brad Chen, Ted Romer, Ramsey Haddad, Louis Monier provided helpful suggestions on the content of this paper. We thank you 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] Robert Hastings and Bob Joyce. Fast Detection of Memory Leaks and Access Errors. *Winter 1992 USENIX Conference*, January 1992.

[9] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach, pp. 408-425, Morgan Kaufmann, 1990.

[10] Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language, Prentice-Hall, 1978.

[11] 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.

[12] James R. Larus and Satish Chandra. Using Tracing and Dynamic Slicing to Tune Compilers. University of Wisconsin Computer Sciences Department Technical Report #1174. August, 1993

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

[14] Digital Equipment Corporation. *Third Degree Reference Manual*, 1993

[15] Digital Equipment Corporation. *ATOM Reference Manual*, 1993

[16] Digital Equipment Corporation. *ATOM User Manual*, 1993

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

[18] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, June, 1994.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’
David Boggs.
WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’
Joel Bartlett.
WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’
G. May Yip.
WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’
David W. Wall.
WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’
Russell Kao.
WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

‘‘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.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’
Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.
WRL Research Report 93/8, December 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.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’
Keith I. Farkas, Norman P. Jouppi.
WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’
Jeffrey C. Mogul.
WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’
Robert N. Mayo, Herve Touati.
WRL Research Report 94/5, April 1994.

WRL Technical Notes

''TCP/IP PrintServer: Print Server Protocol.''
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

''TCP/IP PrintServer: Server Architecture and Implementation.''
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

''Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.''
Joel McCormack.
WRL Technical Note TN-9, September 1989.

''Why Aren't Operating Systems Getting Faster As Fast As Hardware?''
John Ousterhout.
WRL Technical Note TN-11, October 1989.

''Mostly-Copying Garbage Collection Picks Up Generations and C++.''
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

''The Effect of Context Switches on Cache Performance.''
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

''MTOOL: A Method For Detecting Memory Bottlenecks.''
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

''Predicting Program Behavior Using Real or Estimated Profiles.''
David W. Wall.
WRL Technical Note TN-18, December 1990.

''Cache Replacement with Dynamic Exclusion''
Scott McFarling.
WRL Technical Note TN-22, November 1991.

''Boiling Binary Mixtures at Subatmospheric Pressures''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

''A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach''
John S. Fitch.
WRL Technical Note TN-24, January 1992.

''TurboChannel Versatec Adapter''
David Boggs.
WRL Technical Note TN-26, January 1992.

''A Recovery Protocol For Spritely NFS''
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

''Electrical Evaluation Of The BIPS-0 Package''
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

''Transparent Controls for Interactive Graphics''
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

''Design Tools for BIPS-0''
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

''Link-Time Optimization of Address Calculation on a 64-Bit Architecture''
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

''Combining Branch Predictors''
Scott McFarling.
WRL Technical Note TN-36, June 1993.

''Boolean Matching for Full-Custom ECL Gates''
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.