Delivering Binary Object Modification Tools for Program Analysis and Optimization

Linda S. Wilson
Craig A. Neth
Michael J. Rickabaugh

Digital has developed two binary object modification tools for program analysis and optimization on the Digital UNIX version 4.0 operating system for the Alpha platform. The technology originated from research performed at Digital's Western Research Laboratory. The OM object modification tool is a transformation tool that focuses on postlink optimizations. OM can apply powerful intermodule and interlanguage optimizations, even to routines in system libraries. Atom, an analysis tool with object modification, provides a flexible framework for customizing the transformation process to analyze a program. With Atom, compilation system changes are not needed to create both simple and sophisticated tools to directly diagnose or debug application-specific performance problems. The linker and loader are enhanced to support Atom. The optimizations OM performs can be driven from performance data generated with the Atom-based *pixie* tool. Applying OM and Atom to commercial applications provided performance improvements of up to 15 percent.

Historically on UNIX systems, optimization and program analysis tools have been implemented primarily in the realm of compilers and run-time libraries. Such implementations have several drawbacks, however. For example, although the optimizations performed by compilers are effective, typically, they are limited to those that can be performed within the scope of a single source file. At best, the compiler can optimize the set of files presented during one compilation run. Even the most sophisticated systems that save intermediate representations usually cannot perform optimizations of calls to routines in system libraries or other libraries for which no source or intermediate forms are available.[1]

The traditional UNIX performance analysis tools, prof and gprof, require compiler support for inserting calls to predefined run-time library routines at the entry to each procedure. The monitor routines allow more user control over prof and gprof profiling capabilities, but their usage requires modifications to the application source code. Because these capabilities are implemented as compilation options, users of the tools must recompile or, in the case of the monitor routines, actually modify their applications. For a large application, this can be an onerous requirement. Further, if the application uses libraries for which the source is unavailable, many of the analysis capabilities are lost or severely impaired.

By comparison, object modification tools can perform arbitrary transformations on the executable program. The OM object modification tool is a transformation tool that focuses on postlink optimizations. By performing transformations after the link step,OM can apply powerful intermodule and interlanguage optimizations, even to routines in system libraries.

Object transformations also have benefits in the area of program analysis. Atom, an analysis tool with object modification, provides a flexible framework for customizing the transformation process to analyze a program. With Atom, compilation system changes are not needed to develop specialized types of debugging or performance analysis tools. Application developers can create both simple and sophisticated tools to directly diagnose or debug application-specific performance problems.

The OM and Atom technologies originated from research performed at Digital's Western Research Lab (WRL) in Palo Alto, California.[2] The software was developed into products by the Digital UNIX Development Environment (DUDE) group at Digital's UNIX engineering site in Nashua, New Hampshire. Both technologies are currently shipping as supported products on Digital UNIX version 4.0 for the Alpha platform.[3]

This paper first provides technical overviews for both OM and Atom. An example Atom tool is presented to demonstrate how to use the Atom application programming interface (API) to develop a customized program analysis tool. Because OM and Atom can be used together to enhance the effectiveness of optimizations to application programs, the paper includes an overview regarding the benefits of profiling-directed optimizations.

Subsequent sections discuss the product development and technology transfer process for OM and Atom and several design decisions that were made. The paper describes the working relationship between WRL and DUDE, the utilization of the technology on Independent Software Vendor (ISV) applications, and the factors that drove the separate development strategies for the two products. The paper concludes with a discussion about areas for further investigation and plans for future enhancements.

Technology Origins

Researchers at WRL investigating postlink optimization techniques created OM in 1992.[4] Unlike compile-time optimizers, which operate on a single file, postlink optimizers can operate on the entire executable program. For instance, OM can remove procedures that were linked into the executable but were never called, thereby reducing the text space required by the program and potentially improving its paging behavior.[5]

Using the OM technology, the researchers further discovered that the same binary code modification techniques needed for optimizations could also be applied to the area of program instrumentation. In fact, the processes of instrumenting an existing program and generating a new executable could be encapsulated and a programmable interface provided to drive the instrumentation and analysis processes. Atom evolved from this work.[6,7]

In 1993, WRL researchers Amitabh Srivastava and Alan Eustace began planning with DUDE engineers to provide OM and Atom as supported products on the Digital UNIX operating system. Different product development and technology transfer strategies were used for delivering the two technologies. The section Product Development Considerations discusses the methods used and the forces that influenced the strategies.

Technical Overview of OM

OM performs transformations in three phases. It produces an intermediate representation, performs optimizations on that representation, and produces an executable image.

Intermediate Representation

In the first phase, OM reads a specially linked input file that is produced by the linker, parses the object code, and produces an intermediate representation of the instructions in the program. The flow information and the program structure are maintained in this representation.

Optimization

In the optimization phase, OM performs various transformations on the intermediate representation created in the first phase. These transformations include

- Text size reductions
- Data size reductions
- Instruction and data reorganization to improve cache behavior
- Instruction scheduling and peephole optimization
- User-directed procedure inlining

Text Size Reductions One type of text size reduction is the elimination of unused routines. Starting at the entry point of the image, OM examines the instruction stream for transfer-of-control instructions. OM follows each transfer of control until it finds all reachable routines in the image. The remaining routines are potentially unreachable and are candidates for removal. Before removing them, OM checks all candidates for any address references. (Such references will show up in the relocation entries for the symbols.) If no references exist, OM can safely remove the routine. A second type of text size reduction is the elimination of most GP register reloading sequences.[8,9]

Data Size Reductions Because it operates on the entire program, OM performs optimizations that compilers are not able to perform. One instance is with the addressability of global data. The general instruction sequence for accessing global data requires the use of a table of address constants (the .lita section) and code necessary for maintaining the current position in the table. Each entry in the address constant table is relocated by the linker. Because OM knows the location of all global data, it can potentially remove the address entry while inserting and removing code to more efficiently reference the data directly. Removing as many of the .lita entries as possible leaves more room in the data cache (D-cache) for the application's global data.

This optimization is not possible at link time, because the linker can neither insert nor remove code.

Reorganization of the Image By default, OM reorganizes an image by reordering the routines in the image as determined by a depth-first search of the routine order, starting at the main entry point. In the absence of profiling information, this ordering is usually better than the linker's ordering.

In the presence of profiling feedback, OM performs two more instruction-stream reorderings: (1) reordering of routines based on basic block counts and (2) routine ordering based on execution frequency. OM first reorganizes routines based on the basic block information collected by a previous run of the image instrumented with the Atom-based pixie tool. OM lays the basic blocks to match the program's likely flow of control. Branches are aligned to match the hardware prediction mechanism. As a result, OM packs together the most commonly executed blocks. After basic block reorganization, OM then reorders the routines in the image based on the cumulative basic block counts for each routine. The reorganized image is ordered in a way similar to the way the prof tool displays execution statistics. The reordering performed by OM is superior to that performed by cord, because cord does not reorder basic blocks. cord is a UNIX profiling-directed optimization utility that reorders procedures in an executable program to improve cache performance. The cord(1) reference page on Digital UNIX version 4.0 describes the operation of this utility in more detail.

Elapsed-time Performance The optimizations that OM performs without profiling feedback can provide elapsed-time performance improvements of up to 5 percent. The feedback-directed optimizations can often provide an additional improvement of from 5 to 10 percent in elapsed time, for a total improvement of up to 15 percent over an image not processed by OM. Several commercial database programs have realized elapsed-time performance improvements ranging from 9 to 11 percent with feedback.

Executable Image

Finally, in the third phase, OM reassembles the transformed intermediate representation into an executable image. It performs relocation processing to reflect any changes in data layout or program organization.

Technical Overview of Atom

The Atom tool kit consists of a driver, an instrumentation engine, and an analysis run-time system. The Atom engine performs transformations on an executable program, converting it to an intermediate form. The engine then annotates the intermediate form and generates a new, instrumented program.

The Atom engine is programmable. Atom accepts as input an instrumentation file and an analysis file. The instrumentation file defines the points at which the program is to be instrumented and what analysis routine is to be called at each instrumentation point. The analysis file (defined later in this section) defines the analysis routines. Atom allows instrumentation of a program at a very fine level of granularity. It supports instrumentation before and after

- Program execution
- Shared library loading
- Procedures
- Basic blocks
- Individual instructions

Supporting instrumentation at these points allows the development of a wide variety of tools, all within the Atom framework. Examples of these tools are cache simulators, memory checking tools, and performance measurement tools. The framework supports the creation of customized tools and can decrease costs by simplifying the development of single-use tools.

The instrumentation file is a C language program that contains calls to the Atom API. The instrumentation file defines any arguments to be passed to the analysis routine. Arguments can be register values, instruction fields, symbol names, addresses, etc. The instrumentation file is compiled and then linked with the Atom instrumentation engine to create a tool to perform the instrumentation on a target program.

The analysis file contains definitions of the routines that are called from the instrumentation points in the target program. The analysis routines record events or process the arguments that are passed from the instrumentation points.

By convention, the instrumentation and analysis files are named with the suffixes inst.c and anal.c, respectively. Atom is invoked as follows to create an instrumented executable:

```
% atom program tool.inst.c tool.anal.c
```

The atom command is a driver that invokes the compiler and linker to generate the instrumented program. The five steps of this process are

1. Compile the instrumentation code.
2. Link the instrumentation code with the Atom instrumentation engine to create an instrumentation tool.
3. Compile the analysis code.
4. Link the analysis code with the Atom analysis run-time system, using the UNIX ld tool with the -r option so the object may be used as input to another link.

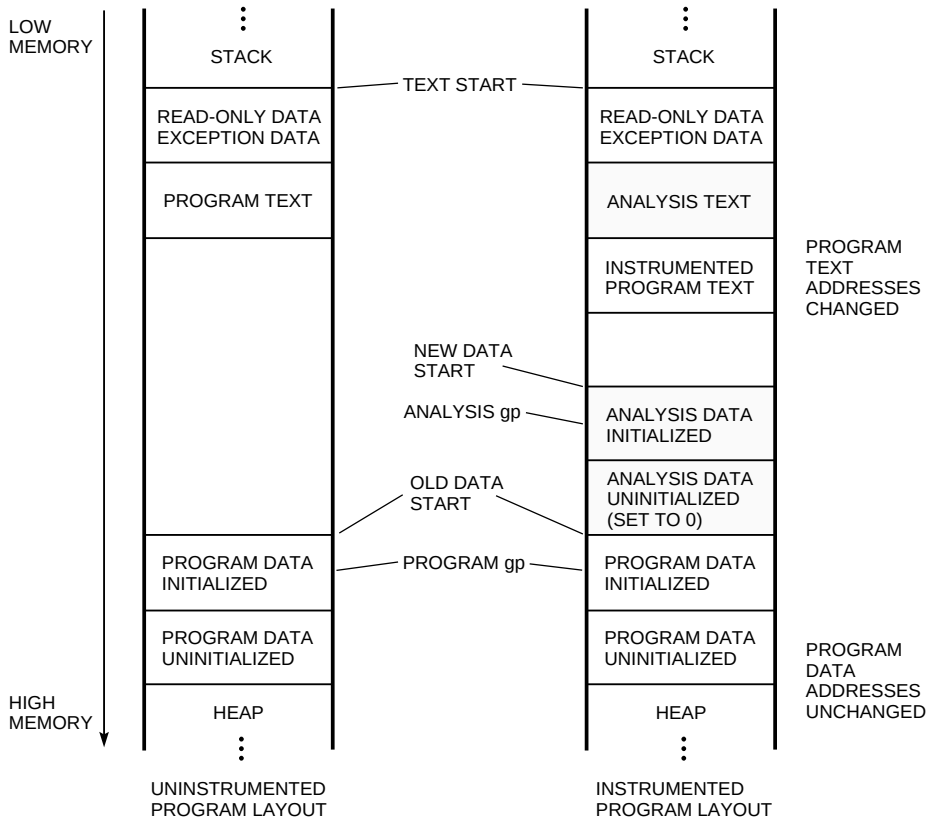
5. Execute the instrumentation tool on the target program, providing the linked analysis code as an argument.

The final step produces an instrumented program linked with the analysis code. Figure 1 shows the changes in memory layout between the original program and the instrumented program.

An Example Atom Tool for Memory Debugging

The following discussion of an example Atom tool demonstrates how to use the Atom API to develop a customized program analysis tool.

A common development problem is locating the source of a memory overwrite. Figure 2 shows a contrived example program in which the loop to initialize an array exceeds the array boundary and overwrites a



Source: A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation,* Orlando, Fla. (June 1994). This paper is also available as Digital's Western Research Laboratory (WRL) Research Report 94/2.

Figure 1
Memory Layout of Instrumented Programs

```
 1  long numbers[8] = {0};
 2  long *ptr = numbers;          /* This pointer is overwritten */
 3
 4  main()
 5  {
 6       int i;
 7
 8       for(i=0; i<25; i++)      /* by this array initialization. */
 9          numbers[i] = i;
10  }
```

Figure 2
Example Program with Memory Overwrite

pointer variable. In this case, the initialization of the *numbers* array defined in line 1 overwrites the contents of the variable *ptr* defined in line 2. This type of problem is often difficult and time-consuming to locate with traditional debugging tools.

Atom can be used to develop a simple tool to locate the source of the overwrite. The tool would instrument each store instruction in the program and pass the effective address of the store instruction and the value being stored to an analysis routine. The analysis routine would determine if the effective address is the address being traced and, if so, generate a diagnostic.

The instrumentation and analysis files for the mem_debug tool are shown in Figure 3. InstrumentInit( ) registers the analysis routines with the Atom instrumentation engine and specifies that calls to the get_args( ) and open_log( ) routines be inserted before the program begins executing. A call to the close_log( ) routine is dictated when the program terminates execution. The Atom instrumentation engine calls InstrumentInit( ) exactly once.

The Atom instrumentation engine calls the Instrument( ) routine once for each executable object in the program. The routine instruments each store instruction that is not a stack operation with a call to the analysis routine mem_store( ). Each call to the routine provides the address of the store instruction, the target address of the store instruction, the value to be stored, and the file name, procedure name, and line number.

The open_log( ) and close_log( ) analysis routines are self-explanatory. The messages could have been written to the standard output, because, in this example, they would not have interfered with the program output.

The get_args( ) routine reads the value of the *MEM_DEBUG_ARGS* environment variable to obtain the data address to be traced. The tool could have been written to accept arguments from the command line using the -toolargs switch. The instrumentation code would then pass the arguments to the analysis routine. In the case of this tool, using the environment variable to pass the arguments is beneficial because the program does not have to be reinstrumented to trace a new address.

The mem_store( ) routine is called from each store instruction site that was instrumented. If the target address of the store operation does not match the trace address, the routine simply returns. If there is a match, a diagnostic is logged that gives information about the location of the store.

To demonstrate how this tool would be used, suppose one has determined by debugging that the variable *ptr* is being overwritten. The nm command is used to determine the address of *ptr,* as follows:

```
% nm -B program | grep ptr
0x000001400000c0 G ptr
```

Instrument the program with the mem_debug tool.

```
% atom program mem_debug.inst.c
  mem_debug.anal.c
```

Set the *MEM_DEBUG_ARGS* environment variable with the address to trace.

```
% setenv MEM_DEBUG_ARGS 1400000c0
```

Run the instrumented program,

```
% program.atom
```

and view the log file.

```
% more program.mem_debug.log

Tracing address 0x1400000c0

Address 0x1400000c0 modified with\
  value 0x16:
at : 0x1200011c4  Procedure: main\
  File: program.c  Line: 9
```

Using this simple Atom tool, the location of a memory overwrite can be detected quickly. The instrumented program executes at nearly normal speed. Traditional debugging methods to detect such errors are much more time-consuming.

Other Tools

An area in which Atom capabilities have proven particularly powerful is for hardware modeling and simulation. Atom has been used as a teaching tool in university courses to train students on hardware and operating system design. Moreover, Digital hardware designers have developed sophisticated Atom tools to evaluate designs for new implementations of the Alpha chip.

The Atom tool kit contains 10 example tools that demonstrate the capabilities of this technology. The examples include a branch prediction tool, which is useful for compiler designers, a procedure tracing tool, which can be useful in following the flow of unfamiliar code, and a simple cache simulation tool.

Atom Tool Environments

Analysis of certain types of programs can require use of specially designed Atom tools. For instance, to analyze a program that uses POSIX threads, an Atom tool to handle threads must also be designed, because the analysis routines will be called from the threads in the application program. Therefore, the analysis routines need to be reentrant. They may also need to synchronize access to data that is shared among the threads or manage data for individual threads. The thread management in the analysis routines adds overhead to the execution time of the instrumented program; this overhead is not necessary for a nonthreaded program. To effectively support both threaded and nonthreaded programs, two distinct versions of the same Atom tool need to coexist. Designers developed the concept of tool environments to address the issues of providing multiple versions of an Atom tool.

```
/*
 * mem_debug_inst.c     - Instrumentation for memory debugging tool
 *
 * This tool instruments every store operation in an application and
 * reports when the application writes to a user-specified address.
 * The address should be an address in the data segment, not a
 * stack address.
 *
 * Usage: atom program mem_debug.inst.c mem_debug.anal.c
 *
 */

#include <string.h>
#include <cmplrs/atom.inst.h>

/*
 * Initializations: register analysis routines
 *                  define the analysis routines to call before and after
 *                  program execution
 *
 * get_args() - reads environment variable MEM_DEBUG_ARGS for address to trace
 * open_log() - opens the log file to record overwrites to the specified address
 * close_log() - closes the log file at program termination
 * mem_store() - checks if a store instruction writes to the specified address
 */
void InstrumentInit(int argc, char **argv)
{
    AddCallProto("get_args()");
    AddCallProto("open_log(const char *)");
    AddCallProto("close_log()");
    AddCallProto("mem_store(VALUE,REGV,long,const char *,const char *,int)");

    AddCallProgram(ProgramBefore, "get_args");
    AddCallProgram(ProgramBefore, "open_log",
                   basename((char *)GetObjName(GetFirstObj())));
    AddCallProgram(ProgramAfter, "close_log");
}

/*
 * Instrument each object.
 */
Instrument(int argc, char *argv[], Obj *obj)
{
    Proc  *proc;
    Block *block;
    Inst  *inst;
    int   base;         /* base register of memory reference */

    /*
     * Search for all of the store instructions into the data area.
     */
    for (proc = GetFirstObjProc(obj);  proc;  proc = GetNextProc(proc)) {
        for (block = GetFirstBlock(proc);  block;  block = GetNextBlock(block)){
            for (inst = GetFirstInst(block);  inst;  inst = GetNextInst(inst)) {
                /*
                 * Instrument memory references. Skip $sp based references
                 * because they reference the stack, not the data area.
                 * Memory references are instrumented with a call to the
                 * mem_store analysis routine. The arguments passed are
                 * the target address of the store instruction,
                 * the value to be stored at the target address,
                 * the PC address of the store instruction in the program,
                 * the procedure name, file name, and source line for the
                 * PC address.
                 */
```

Figure 3
Instrumentation and Analysis Code for the mem_debug Tool

```
                if (IsInstType(inst, InstTypeStore)) {
                    base = GetInstInfo(inst, InstRB);
                    if (base != REG_SP) {
                        AddCallInst(inst, InstBefore, "mem_store",
                            EffAddrValue,
                            GetInstRegEnum(inst, InstRA),
                            InstPC(inst),
                            ProcName(proc),
                            ProcFileName(proc),
                            (int)InstLineNo(inst));
                    }
                }
            }
        }
    }
}

/*
 * mem_debug.anal.c - analysis routines for memory debugging tool
 *
 * Usage: setenv MEM_DEBUG_ARGS hex_address before running
 * the program.
 * Diagnostic output is written to program.mem_debug.log
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

static FILE *log_file;                  /* Output file for diagnostics */

static caddr_t trace_addr;              /* Address to monitor */

/*
 * Create log file for diagnostics.
 */
void
open_log(const char *progname)
{
    char        name[200];

    sprintf(name, "%s.mem_debug.log", progname);
    log_file = fopen(name, "w");

    if (!log_file) {
        fprintf(stderr, "mem_debug: Can't create %s\n", name);
        fflush(stderr);
        exit(1);
    }

    fprintf(log_file, "Tracing address 0x%p\n\n", trace_addr);
    fflush(log_file);
}

/*
 * Close the log file.
 */
void
close_log(void)
{
    fclose(log_file);
}

/*
 * Get address to trace from the environment.
 */
void
get_args(void)
```

Figure 3 (continued)

```
{
    char *addr;
    if (!(addr = getenv("MEM_DEBUG_ARGS"))) {
        fprintf(stderr, "mem_debug: set MEM_DEBUG_ARGS to hex address\n");
        fflush(stderr);
        exit(1);
    }
    trace_addr = (caddr_t) strtoul(addr, 0, 16);
}

/*
 * The target address is about to be modified with the given value.
 * If this is the address being traced, report the modification.
 */
void
mem_store(
    caddr_t             target_addr,    /* Address being stored into */
    unsigned long       value,          /* Value being stored at target_addr */
    caddr_t             pc,             /* PC of this store instruction */
    const char          *proc,          /* Procedure name */
    const char          *file,          /* File name */
    unsigned            line)           /* Line number */
{
    if (target_addr == trace_addr) {
            fprintf(log_file, "Address 0x%p modified with value 0x%lx:\n",
                    target_addr, value);
            fprintf(log_file, "\tat   : 0x%p  ", pc);
            if(proc != NULL) {
               fprintf(log_file,"Procedure: %s  ", proc);
               if(file != NULL)
                 fprintf(log_file,"File: %s  Line: %d", file, line);
            }
            fprintf(log_file, "\n");
            fflush(log_file);
    }
}
```

Figure 3 (continued)

Tool environments accommodate seamless integration of specialized versions of tools into the Atom tool kit. They provide a means for extending the Atom kit. This facility allows the addition of specialized Atom tools by Digital's layered product groups or by customers, while maintaining a consistent user interface.

The versions of the Atom tools hiprof, pixie, and Third Degree that support POSIX threads are provided as a separate environment. hiprof is a performance analysis tool that collects data similar to but with more precision than gprof. pixie is a basic block profiling tool. Third Degree is a memory leak detection tool.

The following command invokes the Atom-based pixie tool for use on a nonthreaded program:

```
% atom program -tool pixie
```

The following command invokes the version of the pixie tool that supports threaded programs:

```
% atom program -tool pixie -env threads
```

Tools for other specialized environments can be provided by defining a new environment name. For example, Atom tools written to work with a language-specific run-time environment can be added to the Atom tool kit by selecting an environment name for the category of tools. Similarly, tools designed to work on the kernel could be collected into an environment.

The environment name is used in the names of the tool's instrumentation, analysis, and description files. The description file for a tool provides the names of the instrumentation and analysis files, as well as special instructions for compiling and linking the tool. For example, the pixie description file for threaded programs is named pixie.threads.desc. It identifies the threaded versions of the pixie instrumentation and analysis files. The Atom driver builds the name of the description file from the arguments to the -tool and -env switches on the command line. The contents of the description file then drive the subsequent steps of the build process.

Tool environments can be added without modification to the base Atom technology, thereby providing extensibility to the tool kit while maintaining a consistent interface.

Compact Relocations

Atom inserts code into the text of the program, thus changing the location of routines. Atom requires that relocation information be retained in the

executable image created by the linker. This allows Atom to properly perform relocations on the instrumented executable.

During the normal process of linking, the relocation entries stored in object files are eliminated once they have been resolved. Because it effectively relinks the executable, Atom must have access to the relocation information.

Consider, for example, an application that invokes a function through a statically initialized function pointer variable, as shown in the following code segment:

```
void foo(int a, int b)
{
        ...
}

void (*ptr_foo)(int, int) = foo;

void bar()
{
        ...
        (*ptr_foo)(1,2);
}
```

The address of function *foo* is stored in the memory location referenced by the *ptr_foo* variable. When Atom instruments this application, the address of *foo* will change, and Atom needs to know to update the contents of the memory location referenced by *ptr_foo.* This is possible only if there is a relocation record pointing at this memory location. Adding compact relocations to the executable file solves this problem.

Compact relocations are smaller than regular relocations for two reasons. First, the Atom system does not require all the information in the regular relocation records in order to instrument an executable. Atom changes only the layout of the text segment, so relocation records that describe the data segments are not needed. Second, the remaining relocations can often be predicted by analyzing other parts of the executable file. This property is used to store a compact form of the remaining relocation records. Since compact relocation records are represented in a different form than regular relocations, they are stored in the .comment section of the object file rather than in the normal relocation area.

Profiling-directed Optimization

OM and the Atom-based pixie tool can interoperate using profiling-directed optimization. The Atom-based pixie tool is a basic block profiler that provides execution counts for each basic block when the program is run. The execution counts are then used as input to OM for performing optimizations on the executable that are driven from actual run-time performance data.

As an example, the following steps would be performed to utilize profiling-directed optimizations with OM:

```
1. % cc -non_shared -o program *.o
2. % atom -tool pixie program
3. % program.pixie
4. % cc -non_shared -om
     -WL,-om_ireorg_feedback,program *.o
```

In step 1, a nonshared version of the program is built. In step 2, the Atom-based pixie tool instruments the program. Step 2 produces program.pixie and program.Addrs files. Step 3 results in the execution of the instrumented program to generate a program.Counts file. This file contains an execution count for each basic block in the program. The last step provides the basic block profile as input to OM. OM rearranges the text segment of the program such that the most frequently executed basic blocks and procedures are placed in proximity to each other, thus improving the instruction cache (I-cache) hit rate.

Product Development Considerations

Bringing the OM and Atom technologies from the laboratory into use on current Digital UNIX production systems required frequent communication and coordination between WRL and DUDE engineers working on opposite coasts of the U.S. The success of both projects depended upon establishing and maintaining an atmosphere of cooperation among the engineers at the two locations. Common goals and criteria for bringing the technology to product supported the teams during development and planning work.

Among the product development considerations for OM and Atom were

1. The products must address a business or customer requirement.
2. The products must meet customer expectations of features, usability, quality, and performance.
3. Engineering, quality assurance, and documentation resources must be identified to ensure that the products could be enhanced, updated to operate on new platform releases, and supported throughout their life cycles.
4. The products must be released at the appropriate times. Releasing a product too early could result in high support costs, perhaps at the expense of new development. Releasing a product too late could compromise Digital's ability to leverage the new technology most effectively.

Product Development and Technology Transfer Process for OM

As part of their research and development efforts, WRL engineers applied OM to large applications. Researchers and Digital engineers at ISV porting laboratories worked together to share information and to diagnose the performance problems of programs in

use on actual production systems, such as relational database and CAD applications. This cooperative effort helped engineers determine the types of optimizations that would benefit the broadest range of applications. In addition, the engineers were able to identify those optimizations that would be useful to specific classes of applications and make them switch-selectable through the OM interface. The performance improvements achieved on ISV applications enabled OM to meet the criteria for addressing customer needs.

Although WRL researchers also applied OM to the SPEC benchmark suite to measure performance improvements, the primary focus of the OM technology development was to provide performance improvements for applications currently in widespread use by the Digital UNIX customer base. With the focus of performance improvements on large customer applications, OM satisfied a prominent Digital business need for inclusion in the Digital UNIX development environment.

Engineers discussed the limitation that OM did not support shared libraries and the programs that used them. In this respect, the technology would not meet the expectations of all customers. Many ISV applications and other performance-sensitive programs, however, are built nonshared to improve execution times. Engineers determined that the benefits for this class of application outweighed this limitation of OM, and, therefore, the limitation did not prevent moving forward to develop the prototype into a product. Developers recognized the risks and support costs associated with shipping the prototype, yet again decided that the proven benefits to existing applications outweighed these factors.

Because of the pressing business and customer needs for this technology, DUDE and WRL engineering concurred that OM should be provided as a fully supported component in Digital UNIX version 3.0. Full product development commitments from DUDE engineering, documentation, and quality assurance could not be made for that release, however. After discussion, WRL provided technical support and extensions to OM to address this need. DUDE engineering agreed to provide consulting support to WRL researchers on object file and symbol table formats and on evaluations of text and data optimizations.

The next issue the engineers faced was how to integrate OM into the existing development environment. They evaluated three approaches.

The first approach was to make OM a separate tool directly accessible to users as /usr/bin/om. Thus, an application developer could utilize OM as a separate step during the build process. This approach offered two advantages. First, it was similar to the approach used to achieve the present internal use of OM and would require few additional modifications to the Digital UNIX development environment. The second advantage was that Atom and OM could be more easily merged into one tool since their usage would be similar. This merging would provide the potential efficiencies of a single stream of sources for the object modification technology.

A major disadvantage of this approach was that it put additional burden on the application developer. OM requires a specially linked input file produced by the linker. This intermediate input file is not a complete executable nor is it a pure OMAGIC file.[10] This approach would require customers to add and debug additional build steps to use OM on their applications. The WRL and DUDE engineers agreed that the user complexity of this approach would be a significant barrier to user acceptance of OM.

The second approach was to change the compiler driver to invoke OM for linking an executable. With this approach, a switch would be added to the compiler driver. If this switch was given, the driver would call /usr/lib/cmplrs/cc/om instead of the system linker to do the final link.

This approach had the advantage of reducing the complexity of the user interface for building an application with OM. A developer could specify one switch to the compiler driver, and the driver would automatically invoke OM. This would allow a developer to introduce feedback-directed optimizations into the program by simply relinking with the profiling information, thus making OM easier to use and less error-prone.

The disadvantage of this second approach was that the complex symbol resolution process in the linker would need to be added to OM. The process of performing symbol resolution on Digital UNIX operating systems is nontrivial. There are special rules, boundary conditions, and constraints that the linker must understand. OM was designed to modify an already resolved executable, and any problems introduced from adding linker semantics would discourage its use. Also, duplicating linker capabilities in OM would require additional overhead in maintaining both components.

The advantages and disadvantages of the second approach motivated the development of a third approach. The compiler driver could be changed to invoke OM during a postlink optimization step. As in the second approach, a switch from the developer would trigger the invocation of OM; however, OM would be run after the linker had performed symbol and library resolution.

The third approach is the one currently used. This method maintains separation between the linking and optimization phases. When directed by the -om switch, ld produces a specially linked object that will be used as input to OM. The compiler driver supplies this object as input to OM when the linking is completed.

The WRL and DUDE engineers found that this functional separation also improved the efficiency of the development efforts between WRL and DUDE. The separation permitted concurrent WRL development on OM and DUDE development on ld, with minimal interference. This approach allowed more development time to be dedicated to technical issues rather than dealing with source management and integration issues.

DUDE engineers added the OM sources into the Digital UNIX development pool and integrated updates from WRL. WRL assumed responsibility for testing OM prior to providing source updates. As previously outlined, DUDE engineers integrated support for OM into the existing development environment tools for the initial release.

Because of proven performance improvements on ISV applications, committed engineering efforts by WRL, and testing activities at both Digital sites, engineers judged the technology mature enough for release on production systems. Efficient development strategies enabled Digital to rapidly turn this leading-edge technology into a product that benefits an important segment of the Digital UNIX customer base.

WRL continued engineering support for OM through the Digital UNIX version 3.0 and 3.2 releases. Responsibility for the technology gradually shifted from WRL to DUDE over the course of these releases. Currently, DUDE fully supports and enhances OM while WRL continues to provide consultation on the technology and input for future improvements.

Product Development and Technology Transfer Process for Atom

WRL deployed early versions of the Atom tool kit at internal Digital sites, ISV porting laboratories, and universities, thus allowing developers to experiment with and evaluate the Atom API. The early availability of the tool kit promoted use of the Atom technology. User feedback and requests for features helped the engineers to more quickly and effectively develop a robust technology from the prototype.

Engineers throughout Digital recognized Atom as a unique and useful technology. Atom's API, with instrumentation and analysis capabilities down to the instruction level, increased the power and diversity of tools that could be created for software and hardware development. Hardware development teams used Atom to simulate the performance of new Alpha implementations. Software developers created and shared Atom tools for debugging and measuring program performance. The value of the Atom technology in solving application development problems provided the business justification for developing the technology into a product.

The prototype version of Atom had several limitations.

- Like OM, the prototype version of Atom worked only on nonshared applications. A production version of Atom would require support for call-shared programs and shared libraries, since, by default, programs are built as call-shared programs. A viable Atom product offering needed to support these types of programs, in addition to nonshared programs.
- Programs needed to be relinked to retain relocation information before Atom could be used. This additional build step impaired the usability of Atom.
- The Atom prototype performed poorly because it consumed a large amount of memory. Much of the data collected about an executable for optimization purposes was not needed for program analysis transformations.
- The Atom API required extensive design and development to support call-shared programs and shared libraries.

The engineers decided to allow the OM and Atom technologies to diverge so that the differing requirements for optimization and program analysis could be more effectively addressed in each component.

Because the cost of supporting a release of the Atom prototype would have been high, WRL and DUDE engineering developed a strategy for simultaneously releasing the Atom prototype while focusing engineering efforts on developing the production version. An Atom Advanced Development Kit (ADK) was released with Digital UNIX version 3.0 as the initial step of the strategy. The ADK provided customer access to the technology with limited support. Engineers viewed the lack of support for shared executable objects as an acceptable limitation for the Atom ADK but unacceptable for the final product.

In addition to allowing WRL and DUDE engineers to focus on product development, this first strategic step permitted the engineering teams more time and flexibility to incrementally add support for Atom into other production components, such as the linker and the loader. For usability purposes, minor extensions were made to the loader to allow it to automatically load instrumented shared libraries produced by Atom tools.

The second step of the strategy was to provide updated Atom kits to users as development of the software progressed. These kits included the source code for example tools, manuals, and reference pages. The update kits performed two functions; they supported users and they provided feedback to the development teams. DUDE and WRL engineers posted information internally within Digital when kits were available and developed a mailing list of Atom users. The Atom user

community grew to include universities and several external customers.

Once the Atom ADK and update strategy were established, DUDE engineering began to implement support for Atom in the linker. As mentioned earlier, Atom inserts text into a program and requires relocation information to create a correctly instrumented executable. The Atom prototype required a program to be linked to retain the relocation information, and this requirement presented a usability problem for users. Ideally, Atom would be able to instrument the executables and shared libraries produced by default by the linker.

Modifying the linker to retain all traditional relocation information by default was not acceptable since the size increase in the executable would have been prohibitive. In some cases, 40 percent of the object file consists of relocation records. Engineers did not view an increase of that magnitude as a viable solution. In addition, this solution conflicted with the goal of Digital UNIX version 3.0 to reduce object file size. As a compromise, DUDE engineering implemented compact relocation support in the linker. Compact relocations provided an acceptable solution since they required far less space than regular relocation records, typically less than 0.1 percent of the total file size.

Another side effect of using compact relocations as a solution was that it introduced a dependency between Atom and ld. All executable objects to be processed by Atom needed to have been generated with the linker that contained compact relocation support. Therefore, to support Atom, layered product libraries and third-party libraries needed to be relinked with the compact relocation support.

In Digital UNIX version 3.0, ld was modified to generate compact relocation information in executable objects. This allowed Atom to instrument the default output of ld. Engineers viewed this capability as critical to the usability and ultimate success of the Atom technology. The compact relocation support in ld was refined and extended over the course of several Digital UNIX releases as development work with Atom progressed.

Concurrently, the WRL research team expanded and began development of the Atom Third Degree and hiprof tools. WRL engineers also continued with additions and improvements to a suite of example Atom tools.

After the release of Digital UNIX version 3.0, DUDE began design and development of the production version of the core Atom technology and the API. DUDE engineers modified and extended the Atom API as tool development progressed at WRL. During peak development periods, engineers discussed design issues daily by telephone and electronic mail.

The original Atom ADK included the source code for a number of example Atom tools. Because some of these tools contained hardware implementation dependencies, they would require ongoing and long-term support to remain operational on changing implementations of the Alpha architecture. For the second shipment of the Atom ADK in Digital UNIX version 3.2, these high-maintenance tools were removed and made available through unsupported channels.

Between releases of the ADK on the Digital UNIX operating system, the engineering teams continued to deliver update kits. Engineers scheduled delivery of the update kits to coincide with key milestones in the software development process. This strategy gave them more control over the development schedule and minimized risk. The update kits provided immediate field test exposure for the evolving Atom software. The design, development, and kit process was practiced iteratively over a year to develop the original ideas into a full product. The Atom update kits were provided for Digital UNIX version 3.0 and later systems, since most users did not have access to early versions of Digital UNIX version 4.0. Providing Atom kits for use on pre–version 4.0 systems allowed the software to be exercised in the field on actual applications prior to its initial release as a supported product. Although support for earlier operating system versions added overhead and complexity to the process of providing the update kits, the engineering teams valued the abundance of user feedback that the process yielded. The benefits of user input to the software development process outweighed the overhead costs.

During Digital UNIX version 4.0 development, WRL engineers finalized the implementations of the hiprof and Third Degree tools and transferred the tool sources to DUDE for further development. The WRL developers had added support for threaded applications on pre–version 4.0 Digital UNIX systems. Because the implementation of threads changed in version 4.0, DUDE engineers needed to update the Atom tools accordingly.

DUDE engineers also developed an Atom-based pixie tool with support for threaded applications. In fact, the Atom-based pixie tool replaced the original version of pixie in Digital UNIX version 4.0. The Atom-based pixie allowed new features such as support for shared libraries and threads to be more efficiently added into the product offering. The development of an Atom-based pixie tool solved the extensibility problems that were being faced with the original version of pixie.

WRL engineers also began to use Atom for instrumenting pre–version 4.0 Digital UNIX kernels, developing special tools for collecting kernel statistics. Atom was extended by DUDE engineering as needed to support instrumentation and analysis of the kernel.

The Atom tool kit and example tools were shipped with Digital UNIX version 4.0. The pixie, hiprof, and Third Degree tools were shipped with the Software Development Environment subset of Digital UNIX version 4.0. Research regarding use of Atom for kernel instrumentation and analysis continues.

WRL continues to share ideas and consults with DUDE on the future directions for the Atom technology.

Conclusions

Developing OM into a product directly from research proved to be challenging. Problems and issues that needed to be addressed had to be handled within the schedule constraints and pressures of a committed release plan.

In contrast, the ADK method used to deliver the Atom product allowed the Atom developers to spend more time on product development issues in an environment relatively free from the pressures associated with daily schedules. The ADK mechanism, however, probably limited the exposure of Atom technology at some customer sites.

The close cooperation of engineers from both research and development was necessary to accomplish the goals of the two projects. We believe that a collaborative development paradigm was key to successfully bringing research to product.

Future Directions

This paper describes the evolution of the OM and Atom technologies through the release of the Digital UNIX version 4.0 operating system. Digital plans to investigate many new and improved capabilities, some intended for future product releases. Plans are under way to

- Provide support in OM for call-shared programs and shared libraries.
- Support the use of Atom tools on programs optimized with OM.
- Investigate providing an API to allow programmable, customized optimizations to be delivered through OM.
- Investigate reuse of instrumented shared libraries by multiple call-shared programs that have been instrumented with the same Atom tool.
- Research support for Atom tools that provide systemwide and per-process analysis of shared libraries.
- Extend Atom to improve kernel analysis.
- Simplify the use of the profiling-directed optimization facilities of Atom and OM through an improved interface.
- Extend the Atom tool kit to provide development support for thread-safe program analysis tools.

In addition to enhancements to the Atom product, original Atom-based tools are expected to become available through the development activities of students and educators at universities. Internal Digital developers will continue to develop and share tools as well.

Acknowledgments

Many people contributed to the development of the OM and Atom products. The following list gives recognition to those most actively involved. Amitabh Srivastava led the research and development work at WRL on OM and Atom and mediated many of the design discussions on the Atom design. Greg Lueck of DUDE designed and implemented the production version of Atom, compact relocations, and the Atom-based pixie tool. Alan Eustace developed Atom example tools, created the first Atom ADK, worked diligently with users, developed kernel tools, provided training and documentation on using Atom, and displayed eternal optimism. Russell Kao at WRL contributed the hiprof tool with thread support. Jeremy Dion and Louis Monier at WRL developed Third Degree and an Atom-based code coverage tool called tracker. John Williams and Chris Clark ofDUDE completed the process of turning the hiprof, pixie and Third Degree tools into products. Dick Buttlar provided documentation on every component. Last but not least, the authors wish to extend a final thanks to all the users who contributed feedback to theOM and Atom development teams.

References

1. F. Chow, M. Himelstein, E. Killian, and L. Weber, "Engineering a RISC Compiler System," *Proceedings of COMPCON,* San Francisco, Calif. (March 1986): 132–137.
2. Western Research Laboratory, located on the Web at http://www.research.digital.com/wrl.
3. R. Sites and R. Witek, *Alpha AXP Architecture Reference Manual,* 2d ed. (Newton, Mass.: Digital Press, 1995).
4. A. Srivastava and D. Wall, "A Practical System for Intermodule Code Optimization at Link-time," *Journal of Programming Languages,* vol. 1 (1993): 1–18. Also available as WRL Research Report 92/6 (December 1992).
5. A. Srivastava, "Unreachable Procedures in Object-oriented Programming," *ACM LOPLAS,* vol. 1, no. 4 (December 1992): 355–364. Also available as WRL Research Report 93/4 (August 1993).

6. A. Eustace and A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools," *Proceedings of the Winter 1995 USENIX Conference,* New Orleans, La. (January 1995). Also available as WRL Technical Note TN-44 (July 1994).

7. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation,* Orlando, Fla. (June 1994). Also available as WRL Research Report 94/2 (March 1994).

8. A. Srivastava and D. Wall, "Link-Time Optimization of Address Calculation on a 64-bit Architecture," *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation,* Orlando, Fla. (June 1994). Also available as WRL Research Report 94/1 (February 1994).

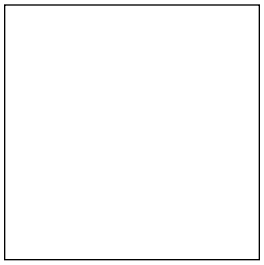
9. *Digital UNIX Calling Standard for Alpha Systems,* Order No. AA-PY8AC-TE, Digital UNIX version 4.0 or higher (Maynard, Mass.: Digital Equipment Corporation, 1996).

10. *Digital UNIX Assembly Language Programmer's Guide,* Order No. AA-PS31C-TE, Digital UNIX version 4.0 or higher (Maynard, Mass.: Digital Equipment Corporation, 1996).

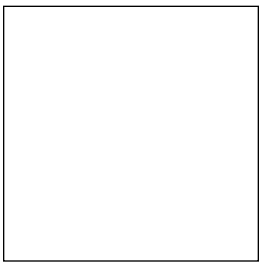
General Reference

J. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," *SIGPLAN Conference on Programming Language Design and Implementation* (June 1995).

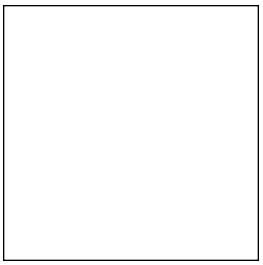
Biographies



Linda S. Wilson
As a principal software engineer in the Digital UNIX Development Environment group, Linda Wilson leads the development of program analysis tools for the Digital UNIX operating system. In prior positions, she was responsible for the delivery of other development environment components, including DEC FUSE, the dbx debugger, and run-time libraries on the ULTRIX and Digital UNIX operating systems. Linda received a B.S. in computer science from the University of Nebraska-Lincoln. Before joining Digital in 1989, Linda held software engineering positions at Masscomp in Westford, Massachusetts, and Texas Instruments in Austin, Texas.



Craig A. Neth
Craig Neth is a principal software engineer in the Digital UNIX Development Environment group, where he is the technical leader of link-time tools. In prior positions at Digital, Craig has worked on the OM object modification tool and the VAX and DEC COBOL compilers, and led the development of DEC COBOL versions 1 and 2. Craig received a B.S. in computer science from Purdue University in 1984 and an M.S. in computer science from the University of Illinois in 1986.



Michael J. Rickabaugh
Michael Rickabaugh is a principal software engineer in the Digital UNIX Development Environment group. He started his Digital career in 1986 in the SEG/CAD Engineering group as a software engineer on the DECSIM logic simulation project. In 1991, Michael transitioned to the DEC OSF/1 AXP project and was a member of the original team responsible for delivering the UNIX development environment on the DEC OSF/1 Alpha platform. He has since been a technical contributor to all aspects of the Digital UNIX link-time technology as well as the creator of the ASAXP assembler for the Windows NT operating system. Michael received a B.S. in electrical and computer engineering from Carnegie Mellon University in 1986.