Moving Atom to Windows NT for Alpha

Eric B. Betts David P. Hunter Sharon L. Smith
Software Partner Engineering
Digital Equipment Corporation[1]

January 30,1999

ABSTRACT

For the last several years, the ATOM technology has provided a flexible tool for instrumenting and analyzing programs on DIGITAL Unix platforms. It has been used to design useful performance and debugging tools, such as basic block counters, cache simulators, and hierarchical profilers, that have been applied to a wide variety of applications. In this paper we present NT-Atom, a new implementation of the Atom technology for Alpha-based Windows NT systems. NT-Atom, which is built on top of the image modification capabilities of SPIKE, is a complete tool development framework for performance tuning and debugging. We discuss the challenges in migrating Atom to Windows NT and maintaining its compatibility with the Unix version of Atom. We also describe how the NT-ATOM graphical user interface supports several new features that facilitate ease-of-use and aid in the development, maintenance, and deployment of custom analysis tools.

INTRODUCTION

As processor implementations become increasingly complex, program behavior becomes more sensitive to a machine's memory hierarchy and other architectural features. A good arsenal of software tools can help illuminate bottlenecks in a program and find potential problems, thereby facilitating performance debugging and tuning. Unfortunately the development of powerful performance tools does not always keep up with changes in hardware.

In order to address this issue, researchers at DIGITAL's Western Research Laboratory in Palo Alto developed the Atom application programming framework [Atom1, Atom2]. A major advantage of this technology is that it allows programmers to develop their own performance tools. The Atom framework does this by allowing the flexible instrumentation and analysis of programs. Atom is based on object modification technology, in which transformations are applied to the program as a postlink step. These transformations allow the rewriting of binary images without requiring source code. The Atom technology has been used to design useful performance and debugging tools, such as basic block counters, cache simulators, and hierarchical profilers, that have been applied to a wide variety of applications. DIGITAL has effectively deployed Atom as a product on DIGITAL Unix platforms [Atom3]**.**

DIGITAL's Software Partner Engineering (SPE) group is chartered with assisting software vendors in migrating their applications to DIGITAL Unix, OpenVMS and Microsoft's Windows NT operating systems. The Atom framework had been an

[1] Since the time of writing DIGITAL has been acquired by Compaq Computer Corporation, and Sharon Smith and David Hunter have joined Compaq's Advanced and Emerging Technologies group.

invaluable tool for isolating application anomalies and for performance profiling under DIGITAL Unix. Unfortunately, a similar tool development framework was not available for Windows NT. In response, the SPE group in Palo Alto in collaboration with the Architecture and Compiler Advanced Development group in Hudson, migrated the Atom technology to the Windows NT operating system for Alpha. The result of this effort is NT-Atom.

In bringing the Atom framework to Windows NT, the primary goal was to provide a framework for a tool development environment that would improve support for building performance tools. Experiences with the DIGITAL Unix implementation and feedback from our software partners indicated that a simple port of the application programming interface alone might not be sufficient to get most engineers using NT-Atom. While users are very supportive and enthusiastic about the technology, it appeared that only the most sophisticated users ventured beyond using the example Atom toolset that ships with DIGITAL Unix. The goal was thus to build an environment that would allow users to develop, deploy, and maintain custom Atom tools.

We had several objectives for creating the NT-Atom tool development environment. The first was to preserve the existing Unix application programming interface as much as possible. This would allow users from the Unix environment to port their existing Atom tools to Windows NT with minimal changes. A second objective was to port a set of the example tools that existed on DIGITAL Unix. A third objective was to maintain the simplicity of Atom and expand it in ways that would promote its ease-of-use. A final objective was to leverage existing work within DIGITAL. DIGITAL's Architecture and Compiler Advanced Development group has developed SPIKE, a postlink optimizer running under Windows NT [SPIKE]. NT-Atom utilizes the image modification capabilities of SPIKE. In this way NT-Atom is able to handle NT images, without our having to migrate the Atom mechanisms that are specific to Unix images.

In the remainder of the paper we describe the implementation of NT-Atom for Alpha-based Windows NT systems. The paper is organized as follows. The next sections give background on Atom and related tools and describe the implementation and design of the core NT-Atom engine. We then describe the NT-Atom tool development environment and illustrate its functionality using a hierarchical profiling tool. The final section summarizes the contributions of this work and describes our plans for future work.

Background

The Atom technology, developed by DIGITAL's Western Research Laboratory in Palo Alto, is a framework that allows for the instrumentation of executable images.
Instrumentation is accomplished by inserting instrumentation points into an executable image at procedure, basic block, or instruction boundaries. At these specified points, new procedure invocations are introduced into a user program. Instrumented programs are executed in the same manner as the original program, with the new procedures being invoked at the instrumented points in the user program. These new procedures, or analysis routines, provide the capability for debugging and other types of analysis of the

program. When executed, the instrumented application produces the desired type of analysis output as it runs.

There are examples of other efforts, mostly within the research community, that have produced framework tools that operate in a manner similar to Atom. The most general of these is Etch, a tool for instrumenting x86 binaries [Etch]. Etch was developed at the University of Washington as a research vehicle for understanding program behavior and for facilitating postlink optimization, as is done in SPIKE [SPIKE] and OM [OM]. Etch was influenced by the Atom project and has a very similar instrumentation API.

Another research project similar to the Atom work is EEL, the Executable Editing Library, that was developed at the University of Wisconsin [EEL]. EEL has primarily been used to modify executables for the purpose of testing new architectural designs. Like Atom, EEL creates an intermediate representation of the executable image that can be easily modified to add instructions to the text segment. In addition, EEL provides a compact description file for different RISC architectures that facilitates translation of the intermediate representation into the appropriate instruction set.

In addition to framework tools that can be programmed to modify an executable in a specific way, various commercial products have been developed that use instrumentation to perform a specific function. Among these are hierarchical profilers to instrument and analyze executables and their components, such as Hiprof by Tracepoint Technology [Hiprof], and Visual Quantify by Pure Atria Corporation, and tools for detecting memory leaks and access violations, such as Purify [Purify] also by Pure Atria and BoundsChecker[NuMega] by NuMega.

Design and implementation of NT-Atom.

In the NT-Atom framework, instrumentation and analysis are accomplished with *tools* that specify how an executable image is to be instrumented and subsequently analyzed. An NT-Atom tool consists of two parts: an instrumentation dynamically linked library (DLL) and an analysis DLL. The instrumentation DLL contains routines that indicate where to place instrumentation points in an executable image, and the analysis DLL contains analysis routines to be called at the various instrumentation points.

The major steps in the NT-Atom instrumentation process are as follows:

1. The user specifies an application to be instrumented and a tool to accomplish the instrumentation:

    A. An application consists of an executable image (EXE) and possibly several DLLs. The user specifies the program EXE and some subset of the program's DLLs for instrumentation.

    B. The user specifies a tool consisting of an instrumentation DLL and an analysis DLL to NT-Atom. Alternatively, the user can specify an instrumentation program and an analysis program that NT-Atom uses to create the instrumentation and analysis DLLs.

2. NT-Atom loads the instrumentation DLL into memory. The instrumentation DLL contains a routine that is called by NT-Atom that specifies how to instrument the EXE and, if applicable, any specified DLLs. The result of this step is an instrumented program.

3. When the instrumented program is executed, the analysis DLL specified in step 1B above is loaded into the program's address space so that calls to the routines in the analysis DLL can be made at the instrumented points in the program. This step produces any analysis output specified by the tool.

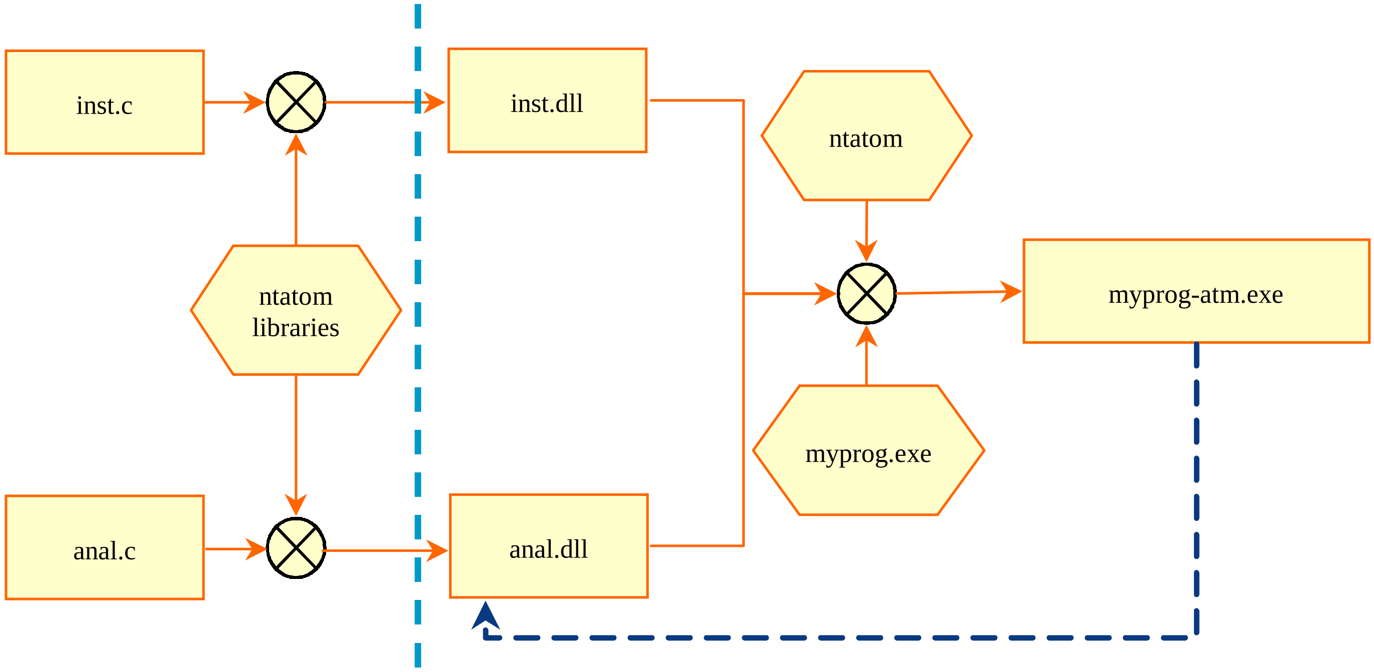


Figure 1: **NT-Atom process overview.**

All of these steps are illustrated in Figure 1. As described above in Step 1, a user can specify an existing tool for instrumentation, such as the example tools provided, or write a custom user defined tool. In the latter case, source files (written in C) for the custom defined tool are compiled into dynamically linked libraries (DLLs), either by the developer or NT-Atom. Step 1B is depicted on the left side of the light blue dotted line in Figure 1. Step 2 is illustrated in the middle of Figure 1 for a hypothetical program, myprog.exe. The result of the instrumentation process is the instrumented executable, myprog-atm.exe. Figure 1 shows a dotted dark blue line between myprog-atm.exe and the analysis DLL, anal.dll. This represents Step 3 of the instrumentation process in which the instrumented program is executed, the analysis DLL loaded into the same address space as the program, and calls into the analysis DLL are made at the instrumented points in the program.

The remainder of this section is concerned with how the instrumentation capability of NT-Atom (Step 2) is implemented. To perform instrumentation on the user program, NT-Atom first builds an intermediate representation of the program's executable image. The program is then navigated as specified by the instrumentation DLL and instructions inserted at the appropriate places to provide calls to the analysis routines. Building the program's intermediate representation and the navigation of the program are accomplished through calls to the SPIKE library. SPIKE is an optimizer for Alpha/NT executables and works directly with manipulating the executable image [SPIKE]. The developers of SPIKE have provided us with a special packaging of the SPIKE internal library calls for the NT-Atom project. In the next sections we describe the NT-Atom application programming interface and how it uses this special SPIKE library.

NT-Atom application programming interface and implementation.

The NT-Atom application programming interface consists of data structures that represent the various constructs in the program, and routines that allow the manipulation of these data structures. The user level data structures in NT-Atom are shown in Table 1. There are objects (or executable images and dynamically linked libraries in NT), procedures, basic blocks, and instructions. The NT-Atom application programming interface (API) allows for the instrumentation of a whole program, that is, an executable image and the dynamically linked library images on which the executable image depends.

Data Structure	Description
Obj	Executable Image or DLL data structure.
Proc	Procedure data structure.
Block	Basic block data structure (all blocks of instructions with a single exit and entry).
Inst	Instruction data structure.

Table 1: Data structures used in the NT-Atom API.

The NT-Atom API set consists of several groups of functions. There are functions for navigating a program, functions for querying information about a program and its constructs, functions for adding instrumentation analysis calls to a program, and functions for building the intermediate representation of an executable image and writing out an instrumented program as a new image. Table 2 lists the function groups and some representative routines from each group.

Function Type	Function Name	Description
Navigation Routines	GetFirstObj(void)	Returns the first object in the program.
	GetNextObj(Obj *)	Returns the next object in the program.
	GetFirstObjProc(Obj *)	Returns the first proc in the object.
	GetNextProc(Proc *)	Returns the next proc in the object.
	GetFirstBlock(Proc *)	Returns the first block in the procedure.
	GetNextBlock(Block *)	Returns the next block in the procedure.
	GetFirstInst(Block *)	Returns the first instruction in the block.
	GetNextInst(Inst *)	Returns the next instruction in the block.
Query Routines	GetObjInfo(Obj *,ObjInfoType)	Get specified information about the given object.
	GetProcInfo(Proc *, ProcInfoType)	Get specified information about the given procedure.
	GetBlockInfo(Block *, BlockInfoType)	Get specified information about the given block.
	GetInstInfo(Inst *, InstInfoType)	Get specified information about the given instruction.
	IsInstType(Inst *, ITypeType)	Determine if instruction is of specified type.
Instrumentation Routines	AddCallProto(const char *)	Add a prototype for the routine named by the const char * string.
	AddCallProgram(PlaceType, const char *, ...)	Add an analysis call before or after a program with the given arguments.
	AddCallProc(Proc *, PlaceType, const char *, ...)	Add an analysis call before or after a procedure with the given arguments.
	AddCallInst(Inst *, PlaceType, const char *, ...)	Add an analysis call before or after an instruction with the given arguments.
Image Building and Writing Routines	BuildObj (Obj *)	Builds up the intermediate representation of the object.
	WriteObj(Obj *)	Writes out the instrumented version of object.

Table 2: Some representative routines in NT-Atom.

To implement the API functions, we started with a snapshot of the Unix Atom software. We reused several of the Unix Atom internal data structures and much of the code that manages the instrumentation at a high level. At lower levels of abstraction, for example in the navigation of the program constructs, we used functions from the SPIKE library and SPIKE data structures in order to accomplish the task. The SPIKE library provides data structures that describe the executable image as a collection of routines. Each routine is a collection of basic blocks and each basic block contains several instructions. NT-Atom uses these data structures for the representation of the text segment of an image. All of these structures are visible to the user. SPIKE also provides several data structures that are used internally in the NT-Atom instrumentation engine. Examples of these are data structures representing relocations, imported symbols and exported symbols in a program.

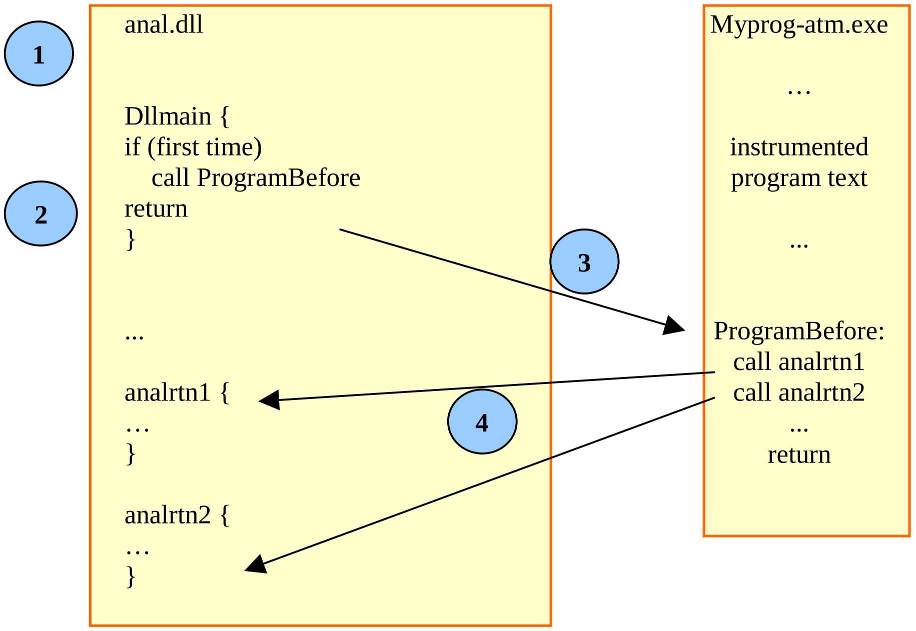
The NT-Atom navigation, query, and image building functions have the most straightforward implementation using SPIKE library functions. In many cases the NT-Atom navigation and query functions map into one or two SPIKE library functions. The NT-Atom API provides additional error checking in order to be consistent with its exposed data structures, but the details of manipulating data structures such as instructions, are left to the SPIKE library. An exception is the navigation of object (or image) level data structures. NT-Atom maintains and navigates a linked list of these data structures. SPIKE operates on a single image at a time, so NT-Atom manages the bookkeeping associated with the images.

The NT-Atom image building and writing functions are implemented with several calls to the SPIKE library. SPIKE provides all of the functions for laying out and writing an image, so NT-Atom deals only with high level abstractions. In this way, NT-Atom is cleanly separated from image level manipulations. This is advantageous since future changes to the executable image format are completely transparent to NT-Atom.

The NT-Atom functions for adding analysis calls to an executable image require the introduction of new instructions and possibly new argument data into the image. NT-Atom calls routines in the SPIKE library to insert new instructions into an image. In addition, SPIKE provides primitives to add new data to an image, so that array or character string argument data can be passed to an analysis function. In order to pass arguments by reference to an analysis routine in NT-Atom, each added piece of argument data is associated with a relocation entry when the image is written. The placement of the relocation in the image is managed by SPIKE. Any instruction referencing the new argument data references the argument's relocation. In this way, the exact addresses for argument data do not have to be known when the instructions to pass argument data are added to the instrumented executable.

NT-Atom has several functions that allow the introduction of calls to analysis routines into an executable image. Most analysis routines that occur in the text are added with AddCallProcedure, AddCallBlock, and AddCallInst, that each add calls to analysis routines that occur before or after the given procedure, basic block, or instruction executes. Calls before or after a program or DLL is loaded are achieved with AddCallProgram or AddCallObj. AddCallProgram adds a call to an analysis routine to occur before a program starts or after it completes, and AddCallObj adds a call to an analysis routine to occur before any code in a DLL is executed or after all code in the DLL completes.

AddCallProgram analysis calls that occur before a program is executed are implemented in the following manner. Recall that the analysis routines are compiled and linked into a separate DLL, which we refer to as the analysis DLL, that is loaded into the program's address space when the instrumented program is run. We link the analysis DLL with a library that provides a special startup routine for the analysis DLL. This startup routine, which is called DllMain, calls back to a routine added to the instrumented executable that contains all of the code for invoking the analysis procedures that must occur before a program is executed. In Windows NT, the user specified DllMain startup routine is executed before any user code at the time the DLL is loaded. The analysis DLL is loaded when the program is loaded and executes its DllMain routine before the program exe's startup routine or any other functions in the main program are executed. The steps involved are illustrated in Figure 2. The implementation of AddCallProgram calls that must occur after a program finishes and the implementation of AddCallObj routines are accomplished in a similar fashion.



1. Myprog-atm.exe is loaded.
2. DllMain is called
3.ProgramBefore routine in instrumented image is called.
4. Analysis routines in anal.dll are called.

Figure 2: Implementation of AddCallProgram for ProgramBefore analysis routines.

Variations from Unix Atom

The NT-Atom application programming interface (API) is very similar to that used in Unix. Maintaining the same API was one of our goals in migrating Atom to Windows NT and we accomplished this by keeping the same data structures and functionality as in Unix. The majority of differences between NT-Atom and Unix Atom arise in the underlying implementation of the data structures and API functions.

One major difference is that NT-Atom uses SPIKE data structures for the internal program representation, whereas the Unix implementation of Atom has explicit data structures for procedures, basic blocks, and instructions. Furthermore, the management of these data structures is accomplished transparently in NT-Atom through calls to the SPIKE library. In contrast, the manipulation of data structures for the Unix implementation is done within the Unix Atom instrumentation engine. The exception is that both Unix Atom and NT-Atom have the same high-level representation for objects and navigate them in the same manner. In principal, it would be possible to place the navigation of object level data structures in NT-Atom at the SPIKE level, although the SPIKE library does not currently provide this functionality. Functions for manipulating objects, such as laying out and writing an image, are all done at the SPIKE level in NT-Atom. In Unix Atom, all of the code layout and adjustment of addresses is done within the core instrumentation engine.

Another major difference between the NT-Atom and Unix Atom implementations appears in the functions that add analysis calls to a program. As indicated above, adding analysis calls to an executable image requires the introduction of new instructions and possibly new argument data into the image. NT-Atom uses SPIKE library routines to add new instructions and data to the image, and to manage relocations that are used in passing analysis data arguments by reference. In Unix Atom, the insertion of new instructions and data into the image is done explicitly by the instrumentation engine. Unix Atom also explicitly manages the calculation of addresses for passing arguments by reference. It does this by determining the offset between argument data and a referencing instruction during the layout and adjustment of addresses, just prior to writing out the executable image.

A further difference for functions that add analysis calls to a program is in the implementation of AddCallProgram and AddCallObj. In the implementation of AddCallProgram, for example, we do not assume that debugging symbols are available for an image in NT-Atom. Thus we cannot instrument the canonical startup and exit routines (for example, __start and _exit in C) in a program's executable image, as is done in Unix Atom. The NT-Atom implementation for AddCallProgram, described in the previous section, is considerably more complex than its Unix counterpart. The advantage is that the NT-Atom implementation does not limit instrumentation of images to only those possessing debugging symbols.

In addition to differences in the implementation of the Atom API, we have added two new functions to the NT-Atom API that did not exist in Unix. These are BuildProcFlowGraph, and ReleaseProcFlowGraph, which build and destroy the control flow graph for a given procedure. Their addition was necessary because the NT-Atom implementation builds flow graphs for procedures one at a time, whereas Unix Atom builds the flow graphs of all of the procedures at the beginning of instrumentation. The NT-Atom implementation is done in this way in order to minimize the amount of memory required to instrument an executable. The disadvantage of this approach is that we only have access to the flow graph of one procedure at a time, and this complicates the implementation of certain NT-Atom API routines, such as ResolveTargetProc.

BuildProcFlowGraph, and ReleaseProcFlowGraph are used implicitly in the implementation of the navigation routines GetFirstProc and GetNextProc in order to allow navigation through the basic blocks and instructions of a procedure, so the user does not need to call them explicitly when navigating and instrumenting procedures in a sequential manner. When procedures are accessed arbitrarily, however, as with the use of the ResolveNamedProc function, these new functions must be used to explicitly build and destroy a flow graph before and after the procedure is instrumented via AddCallInst, AddCallBlock or AddCallProc. This change is the only extension of the Atom API.

A final difference between the two implementations is the way in which the analysis routines are handled. In NT-Atom, all of the analysis routines are compiled into a separate DLL, that is loaded into the instrumented program's address space at the time that the instrumented program is run (see Step 3 in Figure 1.) In contrast, in Unix Atom the binary representation of the analysis code is inserted directly into the same image as the instrumented program.

User Interface Implementation Issues

NT-Atom has both a command line interface and a graphical user interface (GUI). The command line interface is similar in appearance to that used in Unix Atom. The command line arguments are parsed by functions in a separate DLL that we call the common library, or COMLIB. The interaction between COMLIB, the command line interface, the GUI and the rest of the NT-Atom software is shown in Figure 3. COMLIB determines, for example, if a tool has been specified or if instrumentation and analysis routines have been specified and need to be built into DLLs. In addition to parsing the command line arguments, the COMLIB has functions to build the instrumentation and analysis DLLs if necessary, perform various types of error checking, and prepare for instrumentation. The NT-Atom GUI takes advantage of this same functionality by calling a function in the COMLIB that it passes arguments to, in the same format as accepted by the command line interface. Thus both the command line interface and the GUI use the same set of pre-instrumentation routines to prepare for the NT-Atom instrumentation engine.

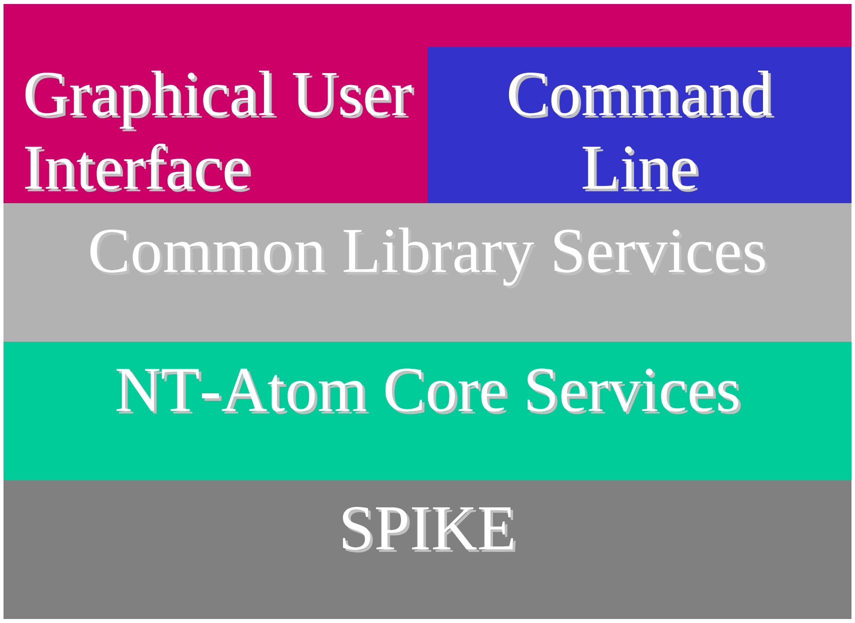


Figure 3: Components of the NT-Atom software system.

In addition to the functions listed above, COMLIB has functions that provide a list of all of the DLLs upon which a program may depend. It does this by examining the DLLs listed in the executable image of a program, as well as the DLLs used in each DLL that the program calls. Both the command line interface and the GUI use this function to determine all of the DLLs that are available to be instrumented. The user selects a subset of these DLLs to instrument. In particular, operating system DLLs cannot be instrumented at this time. All other DLLs are eligible for instrumentation and, when instrumented, are renamed so as not to conflict with other applications that may use the same DLL.

The next section describes the NT-Atom tool development environment and graphical user interface.

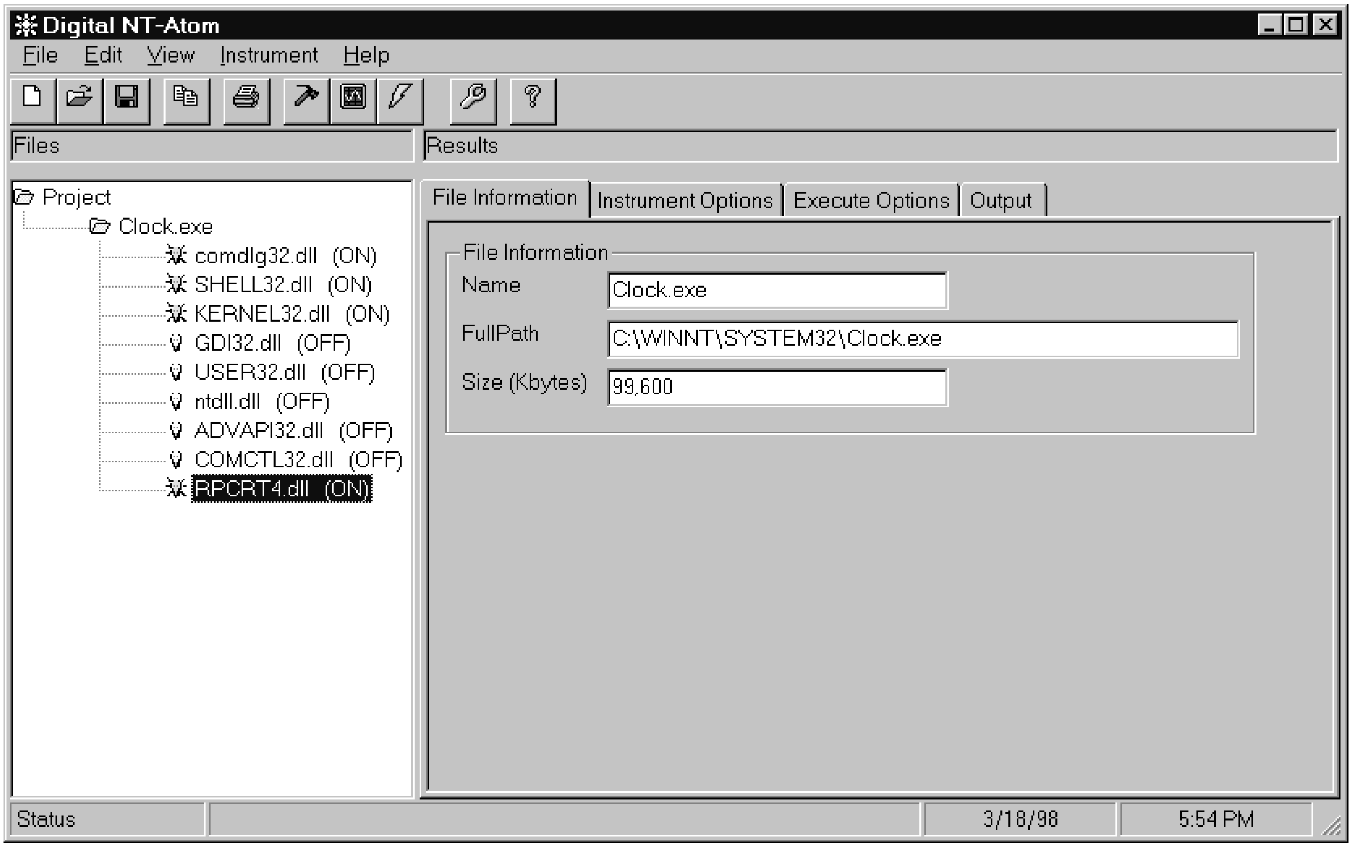
NT-Atom Tool Development Environment

A main objective in the development of NT-Atom was to improve its overall ease-of-use. To accomplish this, we created the NT-Atom Tool Development Environment (TDE). The TDE provides a streamlined graphical user interface that highlights the instrumentation and analysis process. The TDE visually presents NT-Atom's features and functionality in a Microsoft Windows style point and click environment. The TDE takes advantage of Windows user interface features such as tree views and tabbed dialogs for organized data presentation and effortless manipulation of user and analysis output data. The TDE has full on-line help support in the familiar Windows help format. The TDE also has printing capabilities as well as the ability to cut and paste analysis output data directly to other Windows programs.

Instrumenting a program and gathering analysis information with NT-Atom is accomplished in four basic phases:

1. Select a program and optionally select some subset of the program's DLLs.
2. Select an instrumentation tool.
3. Instrument the program.
4. Run the program.

The TDE assists the user by enforcing these basic four phases. For example, the TDE's main display is shown in Figure 4. In this Figure, the "Instrument Options", "Execute Options" and "Output" tabs are disabled until a program to instrument and an instrumentation tool have been selected. As the user completes each phase of the instrumentation process, the TDE's main display automatically updates and changes to the next display panel. Each of these panels displays data relevant to the current phase of instrumentation.



Project
- Clock.exe
  - comdlg32.dll (ON)
  - SHELL32.dll (ON)
  - KERNEL32.dll (ON)
  - GDI32.dll (OFF)
  - USER32.dll (OFF)
  - ntdll.dll (OFF)
  - ADVAPI32.dll (OFF)
  - COMCTL32.dll (OFF)
  - RPCRT4.dll (ON)

File Information

Name: Clock.exe

FullPath: C:\WINNT\SYSTEM32\Clock.exe

Size (Kbytes): 99,600

3/18/98 5:54 PM

Figure 4: The graphical user interface for NT-Atom's Tool Development Environment.

When using the Tool Development Environment the user first selects a program for instrumentation. Once a program is selected, the TDE automatically displays all of the program's dependent DLLs in a flat list directly beneath the program name. At this point the user has the option to selectively click and choose which program dependent DLLs to instrument together with the main program. When a program dependent DLL is selected, the light bulb icon preceding the DLL lights and text following the DLL name indicate that it has been selected. Figure 4 shows a list of dependent DLLs for the program *clock.exe*. Its dependent DLLs, *comdlg32.dll*, *shell32.dll*, *kernel32.dll* and *rpcrt4.dll*, are selected for instrumentation.

For instrumentation, the user can select an example tool or specify a custom tool, as well as add new tools to the environment, as described below. Once a tool is selected, the user can then initiate the instrumentation of the program and its DLLs. After the program is instrumented, the analysis phase is accomplished by executing the program. When the program's execution is complete, analysis data are displayed in the TDE's output window. The user can repeat the instrumentation phase for other tools or other combinations of programs.   The ability to quickly select programs, their DLLs and instrumentation tools allows for the efficient analysis of programs and greatly contributes to NT-Atom's ease-of-use.

Tool and Package Management

In the Unix version of Atom, there is no consistent way to group tools together or easily distribute custom developed tools. These two functions have a direct impact on the ease-of-use of Atom.   We have addressed these two issues with the concept of *Tool Packages.*

A Tool Package is a hierarchical grouping of custom NT-Atom tools for program analysis, performance tuning or program debugging.   Similar to the directory structure in UNIX or Windows NT, the package concept provides a convenient way to group together related tools in a logical manner. In addition to the grouping mechanism provided by directories, packages also allow the user to save information specific to a tool that is used by the TDE whenever an application is instrumented and executed. This feature promotes the ability to easily experiment with and evaluate different tools.

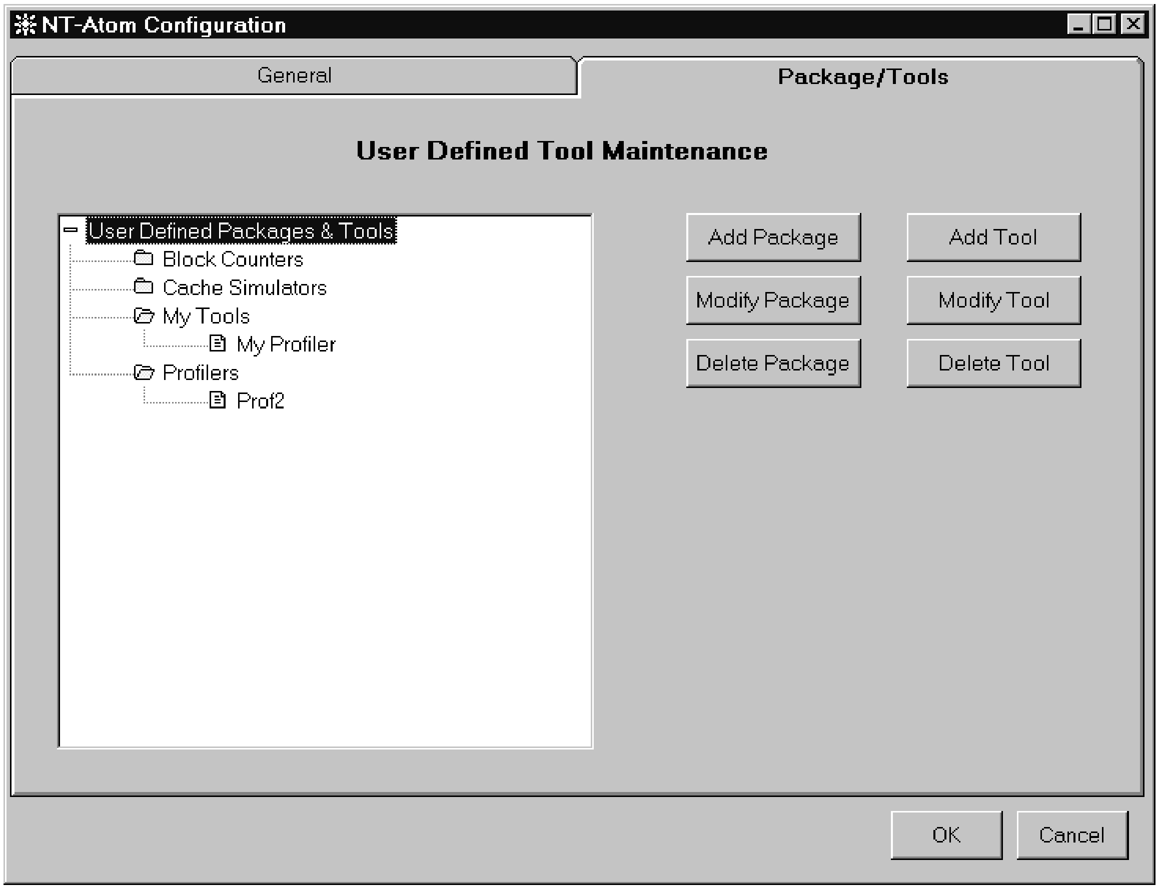


Figure 5: Package and Tool maintenance in the NT-Atom TDE.

The TDE provides a Tool Maintenance facility to manage the addition, modification, and removal of packages and tools through an integrated management framework, as shown in Figure 5. This provides for package and tool grouping selections as chosen by the NT-Atom user. The names of packages and tools are of the user's choice and there are no limitations on how tools can be assigned to packages other than a tool must belong to at least one package. The only package that cannot be directly managed is the group of example

The Tool Development Environment maintains package and tool information through integration with the Windows NT registry. The use of the registry over a file guarantees that information for the TDE cannot be inadvertently deleted. In this manner, package and tool information is dynamically preserved and readily available for the current and future analysis sessions.

Finally, the easy deployment of groups of tools was a goal in the design of the TDE and we have achieved this using the package concept. Packages can be shared and modified among groups of users. This facilitates the distribution of packages that pertain to a particular problem area, such as database applications.

Project State Management

The Tool Development Environment also assists the user in the analysis process by embracing the need to preserve specific user run-time state information. This state information is preserved in an NT-Atom Project. An NT-Atom Project is a snapshot of the current analysis session that preserves the program selected for instrumentation, the selected tool and any command line arguments for tool creation or the execution of the instrumented program. Once preserved, the user can easily continue a previous analysis session or test with different analysis tools and settings. The information for an NT-Atom Project is physically stored in files with the .NAP extension. These files are directly created and maintained by the Tool Development Environment. At any phase of analysis, an NT-Atom project file can be created or a previously stored project retrieved.

NT-Atom projects give the user the flexibility and freedom to explore and preserve different analysis scenarios. This is an important feature that contributes to NT-Atom's ease-of-use by simplifying the analysis process when one or more types of analysis are to be performed on one or more programs.

NT-Atom Example Tools

One of our major objectives in moving Atom to Windows NT was to enable the easy migration of Atom tools developed for DIGITAL UNIX to Windows NT for Alpha. We wanted to ensure that anyone who had built an Atom tool would be able to quickly and transparently port it to NT-Atom. By maintaining the complete API set, we achieved this goal. To validate this, we ported a set of the example Atom tools available and solicited other Atom users to port their tools to NT-Atom. Lastly, we created a few new tools using NT-Atom and ported them back to UNIX Atom. For an overview of how to create tools for Atom refer to the DIGITAL UNIX documentation set [UnixCh9] and the NT-Atom help documentation.

Conversion of Existing Tools

Table 3 lists the set of tools that are distributed with UNIX Atom and indicates which ones are currently available on NT-Atom. The majority of the tools listed were ported in a matter of days or hours. Platform specific code consumed the most amount of time in the port. This includes operating system specific code —thread identifiers, memory heaps, process models, etc. … — and variations in data types. To give an example of the steps that were required in tool migration and how we addressed them, we describe the steps involved in porting the example tool that required the most amount of effort, *Hiprof*, a hierarchical profiler. In addition, we demonstrate how Hiprof might be used to identify performance problems on Windows NT for Alpha.

Tool	Description	Available on NT-Atom
Hiprof	Produces a flat profile of an application that shows the execution time spent in a given procedure, and a hierarchical profile that shows the execution time spent in a given procedure and all its descendents.	Yes. Additional output format in HTML.
Pixie	Produces a profile of an application - by procedure, source line, or instruction by partitioning it into basic blocks and counting the number of times each basic block is executed	Yes.
Branch	Instruments all conditional branches to determine how many are predicted correctly.	Yes.
Cache	Determines cache miss ratio for an 8KB direct-mapped cache.	Yes, updated to handle a 16KB direct-mapped cache also.
Dtb	Determines the number of data translation buffer misses.	Yes.
Dyninst	Provides fundamental dynamic counts of instructions, loads, stores, blocks, and procedures.	Yes.
Inline	Identifies potential candidates for inlining.	No.
Iprof	Prints the number of times each procedure is called and the number of instructions executed by each procedure.	Yes.
Malloc	Records each call to malloc and prints out a summary of the application's allocated memory.	Yes.
Prof	Prints out the dynamic instructions executed by procedure.	Yes.
Trace	Generates an address trace, logs the effective address of each load and store operation.	Yes.

Table 3: Atom Tools.

```
#include <windows.h>
int one(void);
int two(void);
int three(void);
int b(int);

void main(void) {
   int i,j,k;
   i = one();
   for (i=0;i<10; i++) {
      j = two();
   }
}
int one()
{
   int i,j,k=1,l;
   for (i=0; i<15; i++) {
      k = k*(i+1);
      l = b(k);
   }
   return(k);
}
```

```
int two()
{
   int i,j,k=1,l;
   for (i=0; i<105; i++) {
      k = k*(i+1);
      l = three();
   }
   return(k);
}
int three()
{
   int i,j,k=1,l;

   for (i=0; i<1545; i++) {
      k = k*(i+1);
   }
   return(k);
}
int b(int in)
{
   int i,j,k,l;
   for (j=1; j<1; j++) {
      in = in * j;
   }
   return(in);
}
```

Figure 6. An Example Program

Hiprof

Hiprof is a hierarchical program profiler tool that was originally developed for Atom by Russell Kao and is distributed with Atom on the DIGITAL Unix platform [UnixCh8]. The tool generates both flat and hierarchical profiles. The flat profile, similar to the *prof* [UnixCh8] tool, displays the execution tick count of each procedure. This tick count can either be based on instruction counts or time. The hierarchical profile records and displays the same information, but collates the information into a parent-child relationship. This is similar to the UNIX *gprof* profiler, however gprof uses statistical tick counts whereas Hiprof uses actual tick counts. This additional information provides the user with more insight into the actual operation of a program. To illustrate the difference between the two let us look at the program in Figure 6. This program's call tree would look like Figure 7 with each edge in the tree illustrating the number of times the call was made, as well as the number of ticks, or events, that are associated with each edge. If we use the example NT-Atom tool *prof*, we will see a flat profile as illustrated in Figure 8. The same call tree profiled using the hierarchical method, would produce a profile that presented the data for each procedure that included

1. The time spent in its children on its behalf,

2. The time spent within its own procedure excluding its children,

3. The number of times it called each of its children,

4. The total amount of ticks spent in each child procedure, and

5. The total ticks spent in each parent procedure and its children.

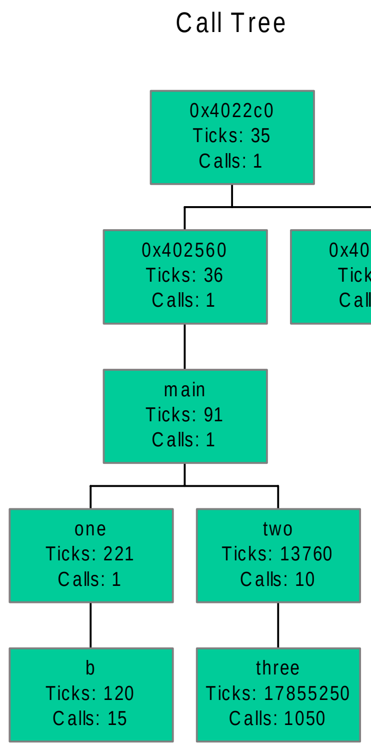


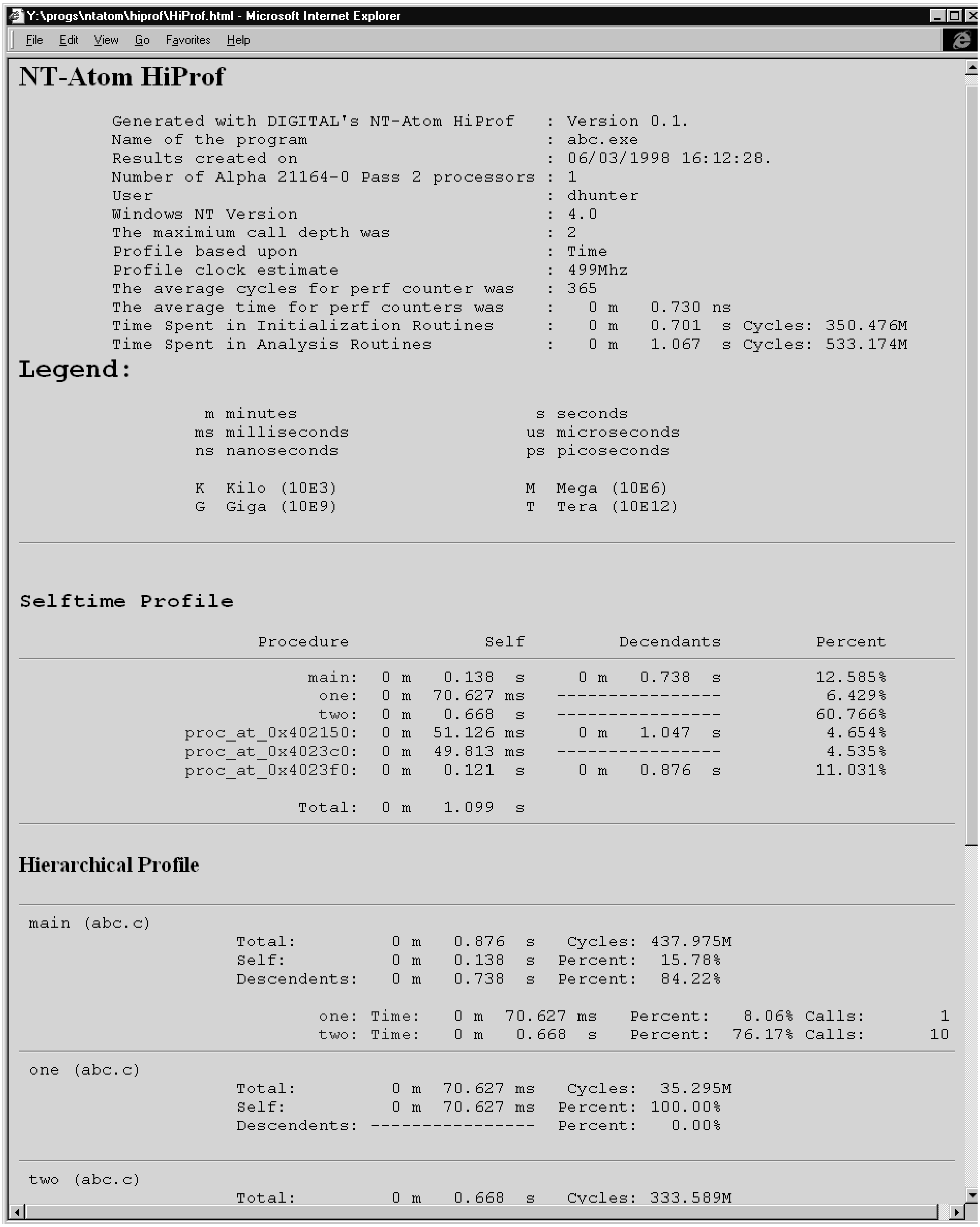
Figure 7: Call graph for example program.

```
          Procedure    Instructions Percentage
------------------------------------------------
               main              91      0.001
                one             221      0.001
                two           13760      0.077
              three        17855250     99.920
                  b             120      0.001
proc_at_0x4022c0                 65      0.000
proc_at_0x402530                  1      0.000
proc_at_0x402560                  6      0.000
              Total        17869514
------------------------------------------------
```

Figure 8: Output from prof tool

Figure 9 illustrates what the same call-tree would look like using a hierarchical profiler.

Hiprof works by selectively instrumenting procedures to record procedure entry and exit points. It does this either at the procedure level, for time based ticks or at the basic block level, for instruction based tick counting. In order to maintain the correct call invocation tree, Hiprof implements a simple stack machine to keep track of the nested procedure calls and to handle incidents of abnormal procedure exits, such as setjmp/longjmp pairings. As each element on the stack is manipulated, information is recorded such as the time or number of instructions executed, the caller of the procedure and the number of


Y:\progs\ntatom\hiprof\HiProf.html - Microsoft Internet Explorer
File
Edit
View
Go
Favorites
Help

NT-Atom HiProf

```
         Generated with DIGITAL's NT-Atom HiProf   : Version 0.1.
         Name of the program                       : abc.exe
         Results created on                        : 06/03/1998 16:12:28.
         Number of Alpha 21164-0 Pass 2 processors : 1
         User                                      : dhunter
         Windows NT Version                        : 4.0
         The maximium call depth was               : 2
         Profile based upon                        : Time
         Profile clock estimate                    : 499Mhz
         The average cycles for perf counter was   : 365
         The average time for perf counters was    :   0 m   0.730 ns
         Time Spent in Initialization Routines     :   0 m   0.701  s Cycles: 350.476M
         Time Spent in Analysis Routines           :   0 m   1.067  s Cycles: 533.174M
```

Legend:

```
                  m minutes                        s seconds
                 ms milliseconds                  us microseconds
                 ns nanoseconds                   ps picoseconds

                 K  Kilo (10E3)                   M  Mega (10E6)
                 G  Giga (10E9)                   T  Tera (10E12)
```

Selftime Profile

```
                       Procedure              Self          Decendants          Percent

                            main:  0 m   0.138  s     0 m   0.738  s          12.585%
                             one:  0 m  70.627 ms   ----------------           6.429%
                             two:  0 m   0.668  s   ----------------          60.766%
               proc_at_0x402150:  0 m  51.126 ms     0 m   1.047  s           4.654%
               proc_at_0x4023c0:  0 m  49.813 ms   ----------------           4.535%
               proc_at_0x4023f0:  0 m   0.121  s     0 m   0.876  s          11.031%

                          Total:  0 m   1.099  s
```

Hierarchical Profile

```
 main (abc.c)
                    Total:         0 m   0.876  s   Cycles: 437.975M
                    Self:          0 m   0.138  s  Percent:  15.78%
                    Descendents:   0 m   0.738  s  Percent:  84.22%

                            one: Time:   0 m  70.627 ms   Percent:   8.06% Calls:       1
                            two: Time:   0 m   0.668  s   Percent:  76.17% Calls:      10

 one (abc.c)
                    Total:         0 m  70.627 ms   Cycles:  35.295M
                    Self:          0 m  70.627 ms  Percent: 100.00%
                    Descendents: ----------------  Percent:   0.00%

 two (abc.c)
                    Total:         0 m   0.668  s   Cycles: 333.589M
```

Figure 9: Example Hiprof HTML Output

times the call is made. This information is later used to reconstruct the flat and hierarchical profile of the program's execution. The core Hiprof functionality is contained in approximately 8000 lines of code spread out among ten separate instrumentation and analysis modules.

Instrumentation Routines

Only a few instrumentation procedures were modified in the initial port of Hiprof. The majority of the code was easily transported, because of compliance with ANSI C and because of the fact that the NT-Atom API set is essentially identical to that used in Unix. The majority of these changes dealt with subtle differences in the compilers available on each platform — DIGITAL's *cc* for UNIX and Microsoft's *Visual C/C++* for Windows NT. For example, Hiprof uses a long variable to hold the tick counts. Under DIGITAL UNIX, a long is 64 bits wide, whereas on Windows NT, a long is only 32 bits wide. To correct this, we modified the variable and others like it to Windows NT Alpha's 64 bit type, *__int64*.

Another area that required modification was in the handling of threads. The UNIX version of Atom had a mechanism to support threads that was tied tightly to DIGITAL UNIX's thread implementation. When we looked at duplicating this on Windows NT, it became apparent that this method would not effectively work under Windows NT. To handle threads in NT-Atom's version of Hiprof, we have implemented a mechanism that stores the unique thread identifier with each tick entry and records it into a memory buffered log. We rely upon our mechanism for maintaining the correct call-stack to keep the profile synchronized. At reconstruction time, we use this log to generate the appropriate tick counts for each parent-child relationship created by the call-tree hierarchy.

Analysis Routines

Porting the analysis routines required attention to the same general porting issues that are observed when moving an application from UNIX to Windows NT [UnixNT]. In addition to the standard textual output of the UNIX Hiprof, we took advantage of hypertext markup language (HTML) to produce an easy to navigate display mechanism. This output format can be viewed in a web browser and easily navigated by clicking on the page links. In designing the ability to output HTML format, we also generate and document an additional output format that describes the call-tree and resulting collected data. We envision that other utilities can use this data to perform such actions as generating graphical views of the data or comparing different runs of the same application. Figure 9 illustrates the HTML style output.

```
#include <windows.h>

#include <winnt.h>
#include <math.h>

void DotProd();
void FloatInit();
void BuggyFloatInit();

typedef struct _point{
double x, y, z;
} point;

point a[10000], b[10000];
double c[10000];

void main( int argc, char *argv[])
{
BuggyFloatInit(a, b, c);
FloatInit (a, b, c);
}

void FloatInit(point a[], point b[], double c[])
{
int i;
for (i = 0; i < 10000; i++)
{
a[i].x  = (double) i;
a[i].y  = (double) i;
a[i].z = 0.0;
b[i].x  = -(double) i;
b[i].y  = -(double) i;
b[i].z = 0.0;
}
DotProd(a, b, c);
}

void BuggyFloatInit(point a[], point b[], double c[])
{
int i;
double x = 1.6e308;
double y = 1.0e308;
for (i = 0; i < 10000; i++)
{
a[i].x  = (double) i;
a[i].y  = (double) i;
a[i].z  = x;
b[i].x  = -(double) i;
b[i].y  = -(double) i;
b[i].z  = y;
}
DotProd(a, b, c);
}

void DotProd(point a[], point b[], double c[])
{
int i;
for (i = 0; i < 10000; i++)
{
c[i] = a[i].x * b[i].x + a[i].y * b[i].y + a[i].z * b[i].z;
}
}
```

Figure 10: Example program to illustrate the Hiprof performance tool.

A performance example using Hiprof

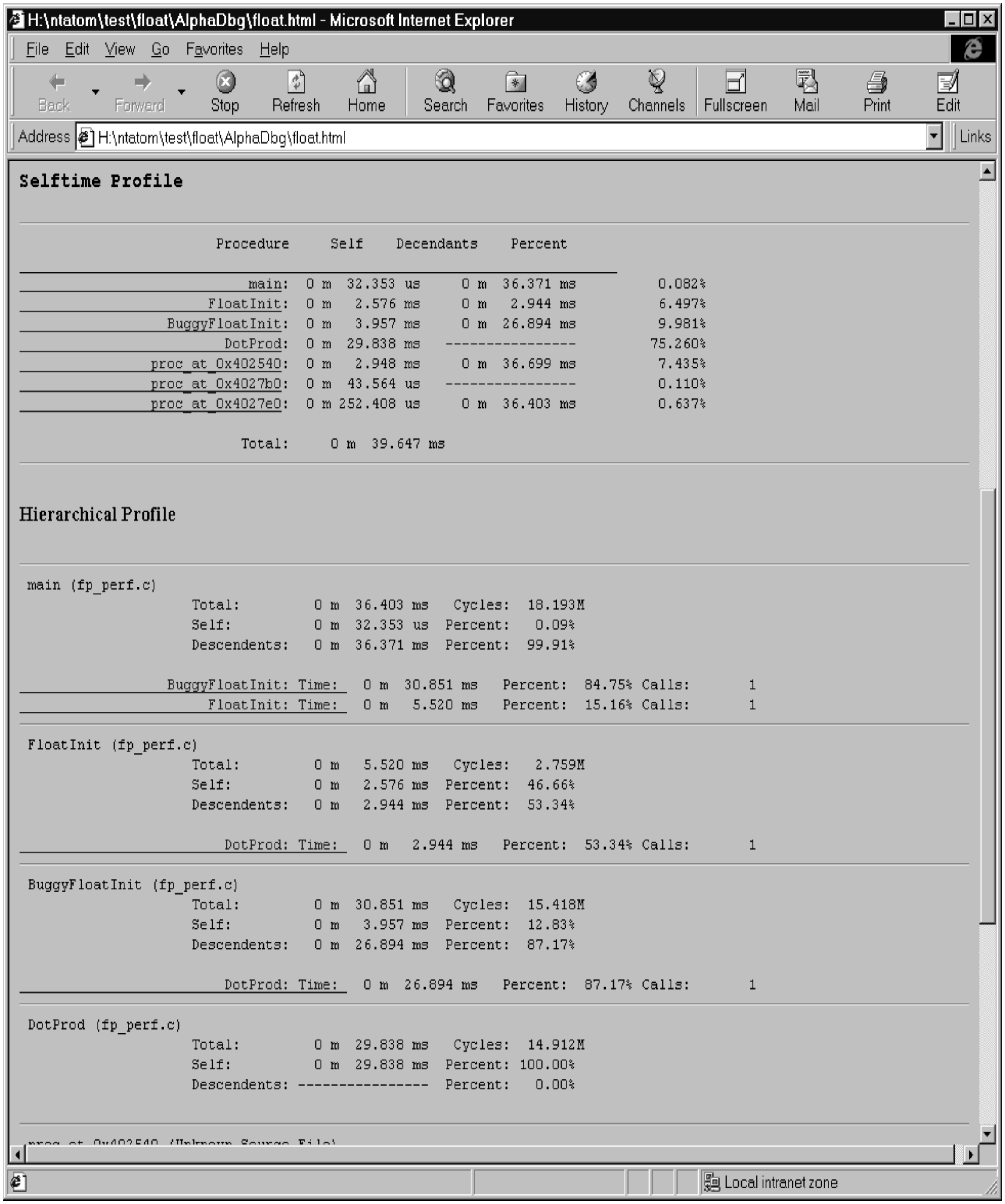
The following example illustrates how Hiprof might be used to identify performance problems that are peculiar to Windows NT and Alpha. The example program shown in Figure 10 contains computations that are often found in applications that perform two and three-dimensional numerical computations. In the example, functions `FloatInit` and `BuggyFloatInit` are called from the main program, and each of these functions calls the function `DotProd`. `DotProd` computes the dot product of two arrays of three-dimensional points. The dot product computation is an important operation in graphical and CAD/CAM applications.

In the example shown, both functions `FloatInit` and `BuggyFloatInit` are identical, with the one exception that function `FloatInit` initializes the z component of the point data structure to 0 for arrays a and b. Function `BuggyFloatInit` stores a very tiny floating point number into the z component of its arrays a and b. Calls to the `DotProd` function from functions `FloatInit` and `BuggyFloatInit` are the same, so that the dot product computation is performed twice on identically sized arrays.

The program was compiled under Visual C/C++ using the /QAieee switch, that forces the compiler to use floating point instructions that adhere to the IEEE floating point standard. In the Windows NT implementation, this means that if a floating point exception arises, the hardware will trap to the NT kernel, and the floating point operation will be completed in software without generating a user level exception. In the example program, the floating point computation on line number 52 of the function `DotProd` would normally result in a user level floating point underflow exception on the Alpha if the `DotProd` function is called from `BuggyFloatInit`. Using IEEE arithmetic, the result of the product is not affected by underflow and can be used in subsequent floating point operations.

This program was instrumented using NT-Atom’s Hiprof. The output from an instrumented run of the program is shown in Figure 11. In this Figure, the surprising result is that the times measured for the functions `FloatInit` and `BuggyFloatInit` are considerably different, despite the fact that they are doing almost the same computation. Further investigation shows that the large difference in time is not due to the different way in which functions `FloatInit` and `BuggyFloatInit` initialize their arrays, but primarily due to the `DotProd` function itself. Since the `DotProd` is operating on the same sized arrays, one would expect the time spent in `FloatInit`’s invocation of `DotProd` to be equal to the time spent in `BuggyFloatInit`’s invocation of `DotProd`. This is not the case because the z component of the arrays used in `BuggyFloatInit` causes the `DotProd` computation on line 52 to underflow and be completed in software on Windows NT for Alpha. This incurs a significant performance penalty, as illustrated by the Hiprof output of our instrumented example program.

The example program illustrates a common problem with numerical codes that are ported from Intel based NT systems to Alpha NT systems, and one that we have observed on a CAD/CAM application in our performance work. Because the Intel architecture implements the IEEE floating point software standard in hardware, bugs, such as underflow due to the product of two very small numbers, are often overlooked. In order


H:\ntatom\test\float\AlphaDbg\float.html - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Channels
Fullscreen
Mail
Print
Edit
Address
H:\ntatom\test\float\AlphaDbg\float.html
Links
Local intranet zone

Selftime Profile

```
                        Procedure     Self     Decendants     Percent

                            main:  0 m  32.353 us     0 m  36.371 ms         0.082%
                       FloatInit:  0 m   2.576 ms     0 m   2.944 ms         6.497%
                  BuggyFloatInit:  0 m   3.957 ms     0 m  26.894 ms         9.981%
                         DotProd:  0 m  29.838 ms   ----------------        75.260%
                proc_at_0x402540:  0 m   2.948 ms     0 m  36.699 ms         7.435%
                proc_at_0x4027b0:  0 m  43.564 us   ----------------         0.110%
                proc_at_0x4027e0:  0 m 252.408 us     0 m  36.403 ms         0.637%

                           Total:     0 m  39.647 ms
```

Hierarchical Profile

```
 main (fp_perf.c)
                    Total:         0 m  36.403 ms   Cycles:  18.193M
                    Self:          0 m  32.353 us  Percent:   0.09%
                    Descendents:   0 m  36.371 ms  Percent:  99.91%

                  BuggyFloatInit: Time:   0 m  30.851 ms   Percent:  84.75% Calls:       1
                       FloatInit: Time:   0 m   5.520 ms   Percent:  15.16% Calls:       1

 FloatInit (fp_perf.c)
                    Total:         0 m   5.520 ms   Cycles:   2.759M
                    Self:          0 m   2.576 ms  Percent:  46.66%
                    Descendents:   0 m   2.944 ms  Percent:  53.34%

                         DotProd: Time:   0 m   2.944 ms   Percent:  53.34% Calls:       1

 BuggyFloatInit (fp_perf.c)
                    Total:         0 m  30.851 ms   Cycles:  15.418M
                    Self:          0 m   3.957 ms  Percent:  12.83%
                    Descendents:   0 m  26.894 ms  Percent:  87.17%

                         DotProd: Time:   0 m  26.894 ms   Percent:  87.17% Calls:       1

 DotProd (fp_perf.c)
                    Total:         0 m  29.838 ms   Cycles:  14.912M
                    Self:          0 m  29.838 ms  Percent: 100.00%
                    Descendents: ----------------  Percent:   0.00%


 proc_at_0x402540 (Unknown Source File)
```

Figure 11: Hiprof output from floating point performance example

to get such code to run on an Alpha, it is necessary to compile with the IEEE software enabled and have the operating system step in when floating point exceptions occur. Unfortunately this leads to a degradation in performance on the Alpha.

The Hiprof tool is instrumental in illuminating such performance problems. A flat profile would break down the times for each procedure, and it would not have been apparent that there was a performance problem in the `BuggyFloatInit`'s invocation of the `DotProd` function. The hierarchical breakdown of times becomes even more important when there are thousands of procedures in a program, as opposed to just two or three.

Other Tools

When we released our first beta, we contacted several members of DIGITAL's research community who had developed several sophisticated Atom tools. These tools were being used to study various compiler and processor issues. We asked them to port their tools over to NT-Atom. To our delight, the researchers were able, through the efforts of a summer intern, to quickly port one of the simulator-input tools from Unix to Windows NT for Alpha. In doing so they uncovered no Atom API incompatibilities. Additional researchers have begun to move their Atom tool suites over to NT-Atom. Currently several internal and three university research groups are using NT-Atom to look at issues such as processor design, cache hierarchy mechanisms, and cache prefetching algorithms. This expansion of research into the Windows NT realm is a benefit to DIGITAL as Windows NT increases in use.

Future Directions

When we started out to port Atom to Windows NT for Alpha, one of the lessons we observed from the UNIX version was that it took a more sophisticated user to benefit from the Atom framework. There was a larger potential audience for NT-Atom, namely, those who were either unwilling or unable to develop sophisticated program analysis tools. To reach these users, we have begun to think of ways to make NT-Atom more user friendly while maintaining its framework concept. Several ideas we are looking at include the graphical display of collected data, automated tool development and integration with other performance tools.

As we examined the unique challenges of keeping NT-Atom as a tool development framework, it became apparent that a means for uniformly displaying the data collected would be very useful. Other tools such as Pure Atria's Purify and NuMega's tool suites present their data in this manner [Purify, NuMega]. Since we have decided to keep the paradigm of a tool building framework, we have begun to work on adding a third phase to the NT-Atom structure. We have named this new phase the *Data Display* phase. It will provide a concise way in which the user will describe their data and how they would like to graphically display it. This new phase will take advantage of existing applications, such as Excel, for graphical data display. In addition, we foresee the creation of new display functionality as appropriate.

We also plan to investigate the area of ease-of-use. We are looking at providing the header files and definitions to more easily allow users to develop both instrumentation as well as analysis files using Microsoft's Visual Basic programming language. In addition, we believe that research in tool development automation would improve NT-Atom's ease of use. For example, in creating an instrumentation tool, we can use a wizard to generate the code that navigates an image.

A final area that we would like to explore is integrating the NT-Atom framework with other performance tools such as SPIKE for optimization and DCPI for sample-based profiling [DCPI]. The goal of this work would be to have a common interface for all of the tools, so that instrumentation, analysis, profiling, and optimization could be easily accomplished without having to use several separate programs.

Conclusions

In this paper we presented the implementation of the Atom technology under the Windows NT operating system. Because of DIGITAL's increased presence in the NT market, NT-Atom was developed in order to address a growing need for performance tools. NT-Atom's core instrumentation engine, built on top of the SPIKE system for optimization of Alpha NT executables, provides the complete functionality of the Atom application programming interface.

The NT-Atom core instrumentation engine has two main innovations. First, the design of NT-Atom using SPIKE allows a clean separation between the details of the executable image and the high level Atom API implementation. This will make it easy to adapt to future changes to the executable image format. Second, the implementation of program level analysis calls in NT-Atom can be accomplished on images that do not contain symbols.

In addition to contributions made in the NT-Atom core implementation, we have substantially improved the user interface for NT-Atom. This paper introduced the NT-Atom tool development environment that provides a graphical user interface for the instrumentation process. In addition, we introduced the concept of tool packages, that allow the development and easy deployment of groups of tools for various user interests. Finally, the concept of NT-Atom projects helps users utilize the tool environment in an efficient way.

In making NT-Atom available, we now have the ability to develop sets of tools to conduct performance investigations and debug programs that can be used on either DIGITAL Unix or Windows NT for Alpha. We hope that NT-Atom will encourage new collaboration between the two user communities and will benefit DIGITAL's developers, partners and customers.

Acknowledgments

NT-Atom did not become a reality on its own. Many people were instrumental in helping us to bring it to fruition. We would like to thank Robert Cohn, David Goodwin and

Geoff Lowney for helping us to integrate with the SPIKE environment and source, and for providing us with the code for the COMLIB DLL image finder. We want to thank Paul Hebert for helping with the design of the GUI, as well as our summer interns, Jack Tihon and Lisa Eklund, for working on the porting of the UNIX tools and a first version of the GUI. We wish to thank Jamey Hicks and summer intern Edwin Foo for being the first fearless users of NT-Atom. Finally, we would like to thank Sas Durvasula and Vehbi Tasar for their encouragement and support of this project.

References

[Atom1] 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).

[Atom2] A. Eustace and A. Srivastava, “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).

[Atom3] L. S. Wilson, C. A. Neth, and M. J. Rickabaugh, “Delivering Binary Object Modification Tools for Program Analysis and Optimization”, *Digital Technical Journal,* Vol. 8 No. 1 (1996).

[SPIKE] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, “Spike: An Optimizer for Alpha/NT Executables”, *Proceeding of the USENIX Windows NT Workshop,* August 11-13, 1997.

[EEL] J. Larus and E. Schnarr, “EEL: Machine-Independent Executable Editing,” *SIGPLAN Conference on Programming Language Design and Implementation*. (June 1995), La Jolla, CA, pp. 291-300.

[NuMega] *SmartDebugging for Component-based, Multi-language Software Environments*: *A White Paper from NuMega Technologies, Inc.*, (www.numega.com).

[Purify] *NTSL Final Report for Rational Software: Performance Testing of Rational Software's software product Purify*, (http://www.rational.com/support/techpapers/pnt-ntsl.pdf) , 9 October 1997

[UnixNT] *UNIX to Windows NT Application Migration Guide,*Digital Equipment Corporation, Maynard, MA., May 1996.

[UnixCh9] DIGITAL UNIX *Programmers Guide*, Chapter 9 Using and Developing Atom Tools, Digital Equipment Corporation, Maynard, MA, March 1996

[UnixCh8] DIGITAL UNIX *Programmers Guide,* Chapter 8 Profiling Programs to Improve Performance, Digital Equipment Corporation, Maynard, MA, March 1996.

[Etch] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and Brian Bershad, "Instrumentation and Optimization of Win32/Intel Executables Using Etch", *Proceeding of the USENIX Windows NT Workshop,* August 11-13, 1997.

[Om] A. Srivastava and D. Wall, "A Practical System for Intermodule Code Optimization at Link-time", *Journal of Programming Languages,* vol 1 (1993): 1-18.

[Hiprof] S. Sipe, "C++ Code Profilers", *PC Magazine Online*, October 21, 1997.

[DCPI] J. M. Anderson, L. Berc, S. Ghemawat, M. Henzinger, S. A. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, , and W. E. Weihl. "Continuous profiling: Where have all the cycles gone?." *Proceedings of the 16th Symposium on Operating Systems Principles, pages 1-14. ACM SIGOPS, October 1997.*