A Survey of Performance Analysis Tools

Patrick Wood, pwood@wustl.edu

Abstract

In this paper, we present a survey of various tools that can be used to aid in performance analysis of computer software programs. We present and discuss different techniques in emerging performance analysis tools as well as provide examples of each type of method. For example, we will cover simple techniques such as timer routines with manual instrumentation to statistical profiling tools and more advanced dynamic instrumentation techniques, in which code can be monitored and changed on the fly. Additionally, a brief comparison of the tools will be presented with some tradeoff considerations.

Keywords:

Performance Analysis Tools, Profiler, Instrumentation, gprof, Pixie, CLR, ATOM, PIN, DynInst

Table of Contents

1. Introduction
2. Brief History
3. Performance Analysis Objectives
4. Timers
5. Profiling Tools
   5.1 gprof
   5.2 SGI Pixie
   5.3 CLR Profiler
   5.4 Profiling Tools Summary
6. Instrumentation Methods
   6.1 ATOM
   6.2 PIN
   6.3 DynInst
   6.4 Instrumentation Methods Summary
7. Conclusion
8. Acronyms
9. References

1. Introduction

Computer system and application performance is an essential and fundamental key in modern development. With increased complexity and requirements, applications are expected to perform greater computation in

less time. Because of this increase in complexity, developers must turn to tools to aid in determining application or system bottlenecks. When using a performance tool, the workload under study is a real world application. Because of the real world workload under study, program tradeoffs in algorithm selection, data structure usage, and more can be evaluated through several runs of a tool. The results can be compared relatively to each run on the targeted system.

Using tools, developers can identify sections of code that, if optimized, would yield the best overall speed-up. Such sections of code are referred to as hot spots and bottlenecks. These will be discussed later. The benefits of using the tool can be to decrease execution time, to increase efficiency of resource utilization, or a combination of the two. For example, an embedded application might need to be studied to decrease memory usage and increase speed. There are three main categories or types of tools: simple timers, profilers, and instrumentation methods. The benefits and drawbacks of these types of tools, as well as specific examples of each, will be presented.

Back To Table Of Contents

---

2. History

Development in the area of performance analysis tools got its start in the early 1980s. In 1982, the first paper was written on the subject of profilers entitled "Gprof: a Call Graph Execution Profiler". This paper led to the development of UNIX's gprof, a call graph execution profiler [Graham82]. A profiler is a program or set of statements that collect data on a program in order to track the performance. Later, in 1994, the area of instrumentation was initiated in a paper written at Digital Equipment Corporation (DEC). The paper detailed the operation of a tool called ATOM. This concept of this tool was to convert a program into its own profiler. At compile time, instructions would be inserted into the program that would output analysis data during execution of the program. Both the gprof and ATOM papers have played a significant role in tools development. In 2004, these have been honored as the top 20 most influential papers by the Programming Language Design and Implementation (PLDI) [Wiki06]. More recent advances have allowed analysis tools more dynamic interaction with the application under study; however, the bases for these tools are rooted in these previous works.

Back To Table Of Contents

---

3. Performance Analysis Objectives

In order to discuss some of the available performance analysis tools, one must first understand the objectives that might be involved in an analysis. First, the system parameters are typically fixed when using a tool. That is, a developer may wish to isolate the function that takes the most time to execute. The application will have a targeted system type, and performance of the given application should be consistent on the system. Secondly, the development team needs to identify the constraint under investigation. For an embedded application, the overriding constraint may be memory consumption and utilization.

For a majority of applications, execution time is the most important factor. For execution time, the best

measure is the wall clock time rather than the CPU time because for most applications, the user's time is the most important. By using a tool, a wealth of data surrounding normally complex program behavior can be presented to the development team. For example, time spent within each procedure, number of calls to a given procedure, memory usage with time, and other similar data regarding the application runtime can be gathered. This data can be used to direct future changes to reduce bottlenecks and hot spots, with a drastically reduced amount of time and effort by the developers.

Most performance analysis tools are best utilized in an iterative approach. First, one would use the tool to gain a broad picture of the application performance under a standard and representative input or workload. Using the results from the tool, the developers can eliminate one bottleneck and then run the tool again. This iterative refinement method will allow the team to eliminate parts of the program that dominate the execution time until satisfactory results are obtained. Another stopping point for the iterations may be time or budget concerns. Furthermore, if the software requirements are SMART (Specific Measurable Acceptable Realizable and Thorough), the development will have a concrete performance goal which can be measured using an analysis tool. By using such a process, the development team is able to impart the most impact on performance up front.

Back To Table Of Contents

---

4. Timers

Perhaps the simplest tool that can be used to analyze the performance of a system is a timer. Timers are universally available on modern systems. The accuracy and resolution of timers is system and language specific. Most languages provide simple, high level calls which provide a coarse measurement. For example, the function gettimeofday is part of the standard C library. It will return the time in seconds and microseconds since January 1, 1970 [LLNL06]. When a programmer adds code to an application to perform timing measurements, the code has been instrumented. Alternatively, the application developers may just be concerned with the total runtime of the application. In this case, the time may be kept by the operating system and reported as a statistic after the program execution. While this call is portable, the resolution is dependent on the system hardware. For most PCs, the resolution is in the microseconds. There are two main categories of time measurements: wallclock or CPU time.

Wallclock time is defined as the total time it takes to execute a program or process from start to finish. Wallclock time is a good measure because it reflects the amount of time a user will spend while executing an application. It includes all overhead including system calls, possible context switches, etc. Therefore, the overall system load will have a definite impact on the wallclock time. It is important that conditions such as system load be presented when the wallclock time is used.

An alternative to wallclock time is CPU time. CPU time represents the time a given application is being actively processed on the system. This can be broken down into system and user times. System times represent time spent during the execution of operating system calls. User time is the total time spent in executing your program. The benefits of breaking down time in such a fashion is beneficial to show a breakdown of time spent in code that you as the developer have control over. While such times may be of interest to a developer attempting to isolate bottlenecks, these times will most likely be of little interest to an end user, who really only cares about how much of their time will be required to run the program end to end.

Back To Table Of Contents

5. Profiling Tools

Statistical profilers are able to report back data that summarizes the code run within an application. It may report back function call count, call traces, time spent within procedures, memory consumption, etc. These tools essentially summarize complex interactions within a program that might take a developer weeks to analyze. Profilers can help isolate down to data types or algorithms that need further development or refinement.

Back To Table Of Contents

5.1 gprof Profile Tool

One of first profilers developed was a tool called gprof, which is a call graph execution profiler. At runtime, the tool gathers three key pieces of data about the executing application. It monitors call counts, call execution time, and call sequences. After executing, the three pieces of information gathered during the application run are combined into a dynamic graph. The edges of the graph contain execution times. The result is a representation of calls with execution times between different calls. This allows a quick visualization to the developer in order to identify regions of code that may be most in need of optimization [Graham82].

Because of the nature of gprof, one must consider the statistical inaccuracies in the data collection methods. Of the data collection described above, the procedure execution count is derived by counting; therefore, it is not prone to inaccuracies during the actual collection process. Similarly, the execution calls are not susceptible to such error. However, the runtimes are subject to inaccuracies due to sampling error. A problem can occur for functions that run for only a small period of time. For this reason, the output of the gprof includes the sampling period, which indicates how often samples are taken. It is generally accepted that the execution time is accurate if it is considerably larger than the sampling period.

Here in Figure 1 is sample output of a flat profile generated by a gprof run.

```
Flat profile:

 Each sample counts as 0.01 seconds.

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name

 33.34      0.02     0.02     7208     0.00     0.00  open

 16.67      0.03     0.01      244     0.04     0.12  offtime

 16.67      0.04     0.01        8     1.25     1.25  memccpy

 16.67      0.05     0.01        7     1.43     1.43  write

 16.67      0.06     0.01                             mcount

  0.00      0.06     0.00      236     0.00     0.00  tzset
```

0.00	0.06	0.00	192	0.00	0.00	tolower
0.00	0.06	0.00	47	0.00	0.00	strlen
0.00	0.06	0.00	45	0.00	0.00	strchr
0.00	0.06	0.00	1	0.00	50.00	main
0.00	0.06	0.00	1	0.00	0.00	memcpy
0.00	0.06	0.00	1	0.00	10.11	print
0.00	0.06	0.00	1	0.00	0.00	profil
0.00	0.06	0.00	1	0.00	50.00	report

Figure 1 - gprof Profile Data [Fenlason93]

As can be seen in the data above, the open procedure accounts for both the most number of calls and the most percentage of time spent for this application. Also, notice that samples are taken every 0.01 seconds. The total execution for the program given by the cumulative seconds field is 0.06 seconds, so only 6 samples were taken. Two samples occurred in the open function and only 1 sample was taken from the mcount function through the report function. Therefore, limited information can be gained on these functions based on this limited execution time.

Back To Table Of Contents

5.2 SGI Pixie

Another common statistical profiler is SGI's Pixie utility. It is a tool that is used to measure execution frequency for procedures within an application. The Pixie program takes in a program and generates an equivalent program which contains additional code to perform the counting. When the application is executed, it generates an output file with the relevant data. This data can be combined with other tools, such as prof, to analyze the results and produce a more readable version of the data.

Unlike some other profiler tools, the Pixie tool offers unique advantages which are slightly different from other such tools. First, it can be used to determine if all code paths have been executed. This is beneficial to relay information to the development team to indicate either dead code or it can identify weaknesses in the test cases that are being run against the application in question. Additionally, Pixie can be used to reorder a program [LLNL06]. Since user program text is demand-loaded, it is important to map commonly used routines together. This would prevent a scenario where highly used procedures are scattered throughout the memory space with seldom used routines. Thus, the overhead of reading routines into memory can be reduced. Figure-2 below shows how the Pixie tool could move a low use procedure from Block 1 to Block 2, which would then allow the three high use procedures to remain in memory potentially longer.

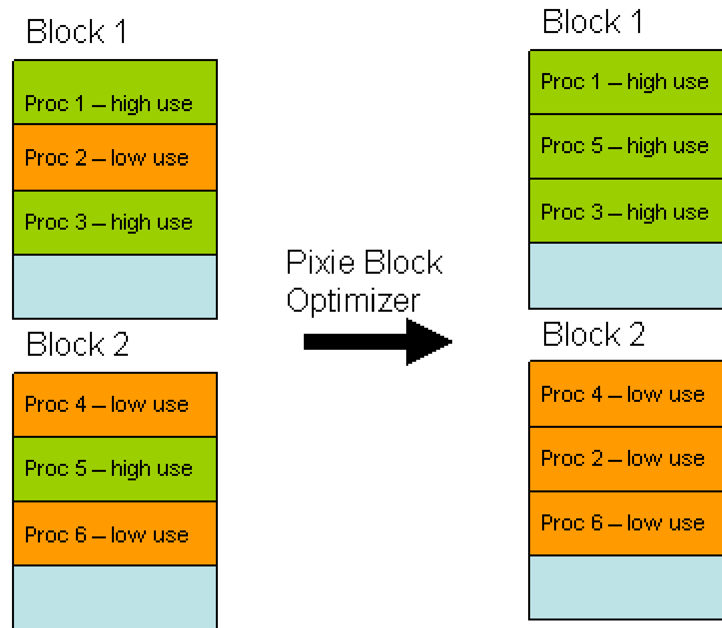

Figure 2: Example Pixie Block Reorganization

Back To Table Of Contents

5.3 CLR Profiler

The Common Language Runtime (CLR) Profiler is used to generate profile information for application written for the .NET framework. A unique obstacle with the CLR profiler is the fact that applications written under .NET are made to be portable between different types of systems. With advances in portability, applications written under a Windows platform can be run on Linux computers. Thus, one must consider implications of different platforms while performing the iterative refinement and optimization process already discussed. Additionally, there is no special compilation process in order to profile an application. A third party application runs which monitors the application being profiled and the .NET framework.

The CLR profiler can provide a large insight into the inner operation of an application, in particular with respect to understanding of how a particular application interacts with the managed heap. It can provide information regarding: who allocated a managed heap, which objects survive on the managed heap, who is holding the objects and what the garbage collector does over the lifetime of the application. Because of the level of detail achieved using this profiler, applications are drastically slowed during execution by as much as 10 to 100 times. For larger applications, this can pose a problem. As a partial remedy to this, one can begin a profile at any point during the application run to focus specifically on an area of interest. Additionally, custom profiling can be generated through the API provided with the .NET CLR profiler. Using Just-In-Time (JIT) compiling techniques, code can be dynamically instrumented and profiled during a live application run [Salchner04].

The CLR provides a graphical view into the results of the application run. Memory related data can be reported as: histogram of allocated types, histogram of relocated types, objects by address, histogram by age and an allocation graph [Salchner04]. A time line view displays the garbage collector over the lifetime of the

application. Using this view, one can see when the garbage collector runs and how the objects are promoted through the generations. Similar to other profilers, call graphs can be used to procedure call sequences as well as call frequency. Using this data, one can determine which procedures require the highest percentage of time, which may make it the most promising area for optimization.

For a specific look at the CLR profiler, one can look at the example "Call Tree" view below in Figure 3.

Call Tree View

View Options

4208 | 4172 | 4192

Function name	Calls (incl)	Calls	Bytes (incl)	Bytes	Objects (incl)	Objects	New function	Ass...
NATIVE FUNCTION (UNKNOWN ARGUM...	1102560	20	2468703		43044	44	4454	
System.OutOfMemoryException (64 ...			64	64				
System.StackOverflowException (64 ...			64	64				
System.ExecutionEngineException (6...			64	64				
System.Object [] (2064 bytes)			2064	2064				
System.Object [] (4096 bytes)			4096	4096				
System.SharedStatics (32 bytes)			32	32				
System.AppDomain (88 bytes)			88	88				
System.String (26 bytes)			26	26				
void System.AppDomain::SetupDom...	24	1	2835		47	4	14	
System.AppDomainSetup (20 byt...			20	20				
System.String [] (72 bytes)			72	72				
System.String (32 bytes)			32	32				
System.String (32 bytes)			32	32				
void System.AppDomain::Set...	23	9	2680		39	26	13	
void System.AppDomain::ResetBindi...								
System.Drawing.Size (16 bytes)			16	16				
System.Drawing.Rectangle (24 bytes)			24	24				
System.Drawing.Point (16 bytes)			16	16				
static void System.Security.Permissi...	3	1	28		1	1	3	
System.Security.PermissionSet (28 b...			28	28				
void System.Security.PermissionSet:...	2	2						
System.Byte [] (658 bytes)			658	658				
bool System.Security.PermissionSet:...	5659	11	14954		231	12	127	
static void System.Security.Permissi...								
static void System.Security.CodeAcc...	1	1					1	
System.Security.PermissionSet (28 b...			28	28				
void System.Security.PermissionSet:...	2	2						
System.Byte [] (642 bytes)			642	642				
bool System.Security.PermissionSet:...	5485	9	3166		81	6	4	
static void System.Security.CodeAcc...	1	1						
System.Collections.Specialized.BitVe...			12	12				
System.Collections.Specialized.BitVe...			12	12				

Name
System.AppDomain::Setup...

Figure-3: Sample CLR Profiler Call Tree View [Salchner04]

The "Name" field provides the method called or the type of allocated object. The "Calls (incl)" field shows the number of sub-procedures that are called from within the method. The "Calls" field shows only the number of sub-procedures that are called directly from the current procedure. Figure 4 illustrates the difference. Here, function A would have a Calls (incl) count of 3, but would only have a call count of 1, since it only directly calls functionB.

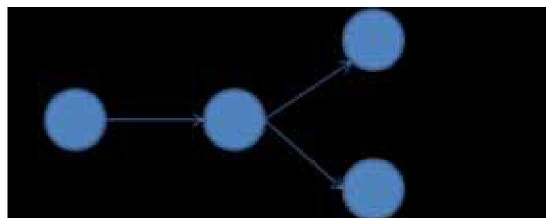

Figure-4 : Function Tree

Additionally, the view shows the number of bytes allocated within the function call (both inclusive and

non-inclusive). It gives a count of the objects created by the method, which is inclusive of sub-procedure calls. Additionally, it shows the number of functions that were JIT'ed in order for them to be executed in the near future. As explained earlier, the JIT compiler generates machine instructions and stores them on the JIT Code Heap for execution.

Back To Table Of Contents

5.4 Profiling Tools Summary

The following Table 1 summarizes the key uses and benefits for the tools presented in this section.

Table 1 : Profiling Tools Summary

Tool	Summary
gprof	• Statistical profiler • Provides call counts, execution time, and call sequences
SGI Pixie	• Statistical profiler that measures function execution frequency • Determines code execution path coverage • Can re-order a program block
CLR Profiler	• Statistical profiler • Provides detailed insight into application memory usage • Dynamic instrumentation through JIT

Back To Table Of Contents

---

6. Instrumentation Methods

Instrumentation is a method used by some performance analysis tools in which the source code or program is modified with additional commands in order to output analysis data during the execution of the application under investigation. It is the logical extension to the profiling and it goes hand in hand with timer usage.

There are five main types of techniques used to instrument code: manual, compiler assisted, binary translations, runtime instrumentation, runtime injection. Manual instrumentation is where a developer will personally add instruction to explicitly perform performance analysis. For example, a developer may

explicitly add a timer within the code over specific regions in order to determine runtime of a procedure. Compiler assisted techniques are where performance analysis code is inserted into the executable at the time of compilation. An example of a compiler assisted instrumentation method is gprof, which was discussed earlier. With binary translations methods, the tools will take a pre-compiled application and modify it with additional check points for performance analysis. Runtime instrumentation is a technique in which the program under investigation is supervised and controlled by an external tool which is monitoring and measuring the performance. The final technique is runtime injection, in which the code is modified at runtime to jump to a helper function [Wiki06].

Back To Table Of Contents

6.1 ATOM

ATOM is a collection of tools developed in 1994 at Digital Equipment Corporation that can be applied to a variety of applications for specific investigations. This tool set relies on compiler assisted instrumentation techniques. The code must be compiled in linked with a selected tool from the set. Tool objectives within the ATOM package include: instruction profiling, system call summary, I/O summary, memory leak detection, and more [DEC94]. The process used by ATOM is shown in Figure-5.

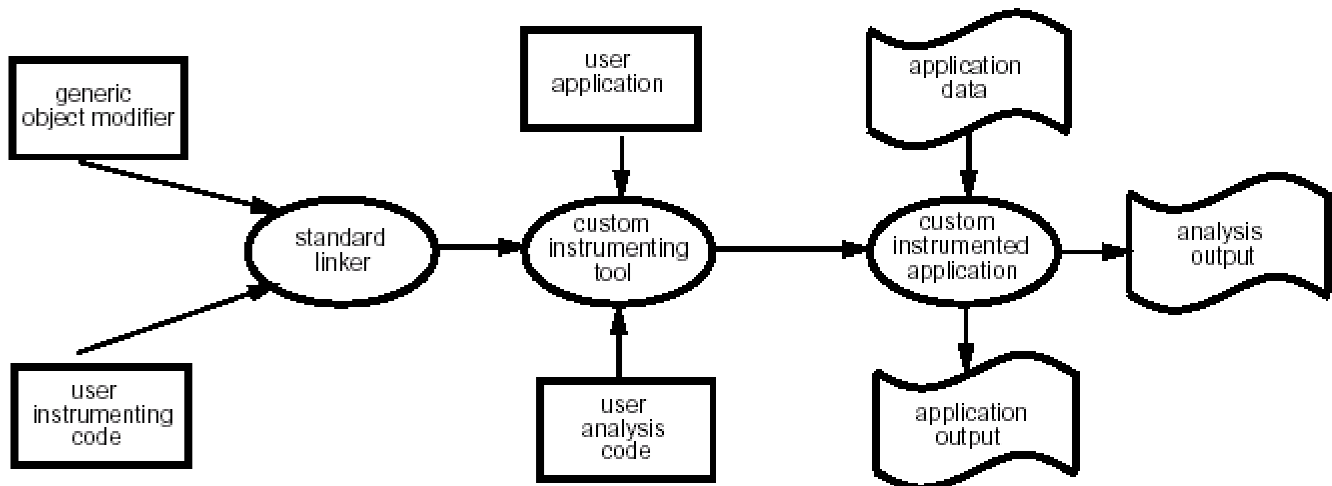

Figure-5 : ATOM Process [Eustace94]

Procedure tracing is performed by a tool called ptrace. It outputs the procedure name as each procedure executes. It is important to note that only calls within the original program are instrumented. Similarly, procedures calls that output names are not output. This organization of only necessary and relevant information is important to keep the investigation focused. While this is a simple tool, and if used alone, may not yield and great breakthroughs, it still can identify complex program sequences and procedure call

frequency.

Prof is the profiling tool used with the ATOM package. Much like gprof and the other profiler discussed earlier, prof is a tool used for identifying sections of code that are in need of optimization. The tool works by counting instructions within each basic block. It can then report the number of instructions executed within a procedure, the total number of instruction executed and the percentage of instructions executed per procedure. The instrumentation works by adding a procedure call to the analysis routine that passes the procedure number and the number of instructions within the basic block. After the application runs to the end, the analysis routine and report the summary to the investigator. Here is sample output from the tool:

Procedure	Instructions	Percentage
__start	148	0.000
main	107	0.000
compress	58040713	66.487
output	28907385	33.114
rindex	130	0.000
...	...	...
Total	87296502	

Figure 6 - Sample ATOM profile output [Eustace94]

ATOM provides a mechanism to add instruction and data address tracing to code. The compiler will insert code which will save the first address executed in each basic block along with the effective address of all load and store addresses. This data can then be used to replay the execution using a tool like an instruction and data cache simulator.

Accuracy was an important objective for the ATOM tools. One problem that could change the behavior or characteristic of an executing application is if memory is allocated dynamically by the instrumentation code. For example, if an analysis routine calls printf, which in turns makes a library call to the malloc procedure, the next memory location dynamically allocated in the program under study would be different. In order to avoid the malloc, a setbuffer is inserted after the fopen and before the first read/write operation. Another approach is to pass data in an array if the data size is known statically. Yet another approach would be for the tools to dynamically allocate necessary memory prior to the start of the application. Thus, the references within the application should remain consistent even though they will shift the dynamic memory addresses by some constant. Additionally, a problem may exist if file descriptors given by operating system are important. While this would only be a problem in a poorly written application relaying on particular values, the designers of ATOM wanted to make sure the problem can't exist with use of the tool. In order to alleviate the issue, the analysis program opens a file, duplicates the new file descriptor, then closes the original file. Thus, the original file designator is available for the program under study. Finally, a very minor difference may exist in the setup routines that setup the argc, argv, and environmental variable for the application. Any minor differences in a basic block count may exist even if you make the smallest of changes to a program. These small changes should not have much effect in the overall execution of the program [DEC94].

6.2 PIN

Released in 2004, PIN is an example of a tool that utilizes binary instrumentation techniques for Linux applications. The tool is designed to be easy to use, portable, transparent, and robust. The PIN user model is similar to ATOM, which makes it easy for developers to transition between the tools. Additionally, this tool uses JIT in order to insert and optimize code. The ability to fully automate register reallocation, inlining, liveness analysis, and instruction scheduling set this tool apart from other instrumentation tools. Additionally, it is noted that this tool may be a good introductory tool in a classroom environment in order to give students insight into program behavior [Chi-Keung05].

PIN provides an easy to use interface, much like modern debuggers. In order to instrument, profile, or perform other analysis, the developer simply attaches to the already running application. The developer and then perform the desired data collection. When the necessary data has been collected, the PIN tool can detach from the application and the application can continue running as normal. The benefit to this is that the analysis can be performed at specific points in the application run. For example, some applications may have a large startup overhead, which is unimportant to the application analysis. The application could be started, and then the profile data can be gathered during later execution.

Here in Figure 7 is sample output from the PIN tool documentation. One can see the procedure name is output, along with the image library, the address, the call count and the instruction count per call.

```
$ pin -t proccount -- /bin/grep proccount.C Makefile

proccount_SOURCES = proccount.C

$ head proccount.out

Procedure               Image       Address      Calls  Instructions

_fini                   libc.so.6   0x40144d00   1      21

__deregister_frame_info libc.so.6   0x40143f60   2      70

__register_frame_info   libc.so.6   0x40143df0   2      62

fde_merge               libc.so.6   0x40143870   0      8

__init_misc             libc.so.6   0x40115824   1      85

__getclktck             libc.so.6   0x401157f4   0      2
```

munmap	libc.so.6	0x40112ca0	1	9
mmap	libc.so.6	0x40112bb0	1	23
getpagesize	libc.so.6	0x4010f934	2	26
$				

Figure 7 - Example PIN call/instruction count output [Cohn05]

Back To Table Of Contents

6.3 DynInst

DynInst is a performance analysis tool that uses the runtime injection instrumentation method. Such a tool offers several benefits, which include, flexible debugging, performance monitoring, and application steering. Application steering is a process in which adjustments can be made at runtime in order to exercise different program paths or resource usage.

In order for DynInst to work, the developer must write a mutator application, which will act as the entry point for the developer in order to impose chance in the mutatee application (i.e. the application under study). The mutator application will connect up with the mutatee at runtime. Using this method, the possible mutations are restricted by the mutator application. That is, all mutations that will occur at runtime must be developed and designed by the developer prior to initiating the process with DynInst [Williams04]. Additionally, while dynamic instrumentation is possible, it must be implemented by the developer in the mutator application. Therefore, the level of interaction achievable with DynInst is determined by the work done by the developer prior to the start of the process.

In order to perform the instrumentation, DynInst inserts a call to a base function, which is called the base trampoline. The base trampoline function makes a call to what is called the mini trampoline. Within the mini trampoline, the registers are saved off, the arguments for the instrumented call are setup, the instrumented code is executed, then the registers are restored and the control is returned to the base trampoline function. The base trampoline performs post processing before returning to the executing application. The architecture allows for multiple base trampoline and mini trampoline calls to be inserted at the same point in the program [Williams04]. See Figure-8 below to see the high level operation of the DynInst process.

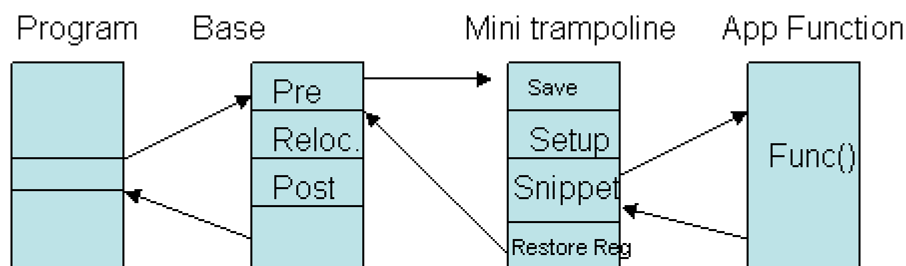

Figure-8: Example DynInst Process

In order to use DynInst as a performance analysis tool, the user is most likely need to go through several iterations of the mutator function. The first iteration would collect high level data on the running application. This would identify sections of code that are the biggest bottleneck. The mutator could then be refined to collect more fine grained data on the areas of code that have been identified as potential bottlenecks. As the process continues, the effects of running different algorithms could be examined by use of the mutator function. The draw back is that this approach could be more time consuming than other available tools; however, it would yield very good feedback to the developers the further along in the process one gets.

Back To Table Of Contents

6.4 Instrumentation Methods Summary

The following Table 2 summarizes the key uses and benefits for the tools presented in this section.

Table 2 : Instrumentation Tools Summary

Tool	**Summary**
ATOM	• Rich collection of performance analysis tools • Utilizes compiler assisted instrumentation technique • Tools to perform: instruction profiling, system call summary, I/O summary, memory leak detection, etc
PIN	• Utilizes binary instrumentation techniques • Designed to be portable, transparent and robust • Good as an educational tool
DynInst	• Utilizes runtime injection instrumentation technique • Mutator application performs dynamic instrumentation

Back To Table Of Contents

---

7. Conclusion

In this paper, we have presented a summary of several different performance analysis tools that are available today. Table 3 below shows a quick summary of the tools discussed in this paper. We have shown the evolution of tools, from manual instrumentation techniques using simple timers to advanced runtime injection techniques such as those employed in DynInst. While using statistical profilers, a development team is able to rapidly examine the application bottlenecks, however, one must perform offline analysis and optimization then repeat the cycle in order to ensure the desired effects are achieved. Modern techniques are more and more relying on JIT processes, in which the developer can inject code changes on the fly. This provides immediate feedback to the developer; however, there are varying degrees in which this process can be dynamic. For example, DynInst requires the mutator to be defined prior to the start of the debug process; whereas, the CLR profiler could be used to insert and modify entire sections of code. While each tool currently offers its own set of benefits, each also has some drawbacks. Since these tools are constantly evolving, it appears likely that future tools will be able to provide a rich set of runtime code changes with live profiler statistics as the application executes.

Table 3 : Tools Summary

Tool	**Summary**
Timer	• Simple, universal • Provides runtime feedback • Typically uses manual instrumentation technique
gprof	• Statistical profiler • Provides call counts, execution time, and call sequences
SGI Pixie	• Statistical profiler that measures function execution frequency • Determines code execution path coverage • Can re-order a program block
CLR Profiler	• Statistical profiler • Provides detailed insight into application memory usage • Dynamic instrumentation through JIT
ATOM	• Rich collection of performance analysis tools

	• Utilizes compiler assisted instrumentation technique • Tools to perform: instruction profiling, system call summary, I/O summary, memory leak detection, etc
PIN	• Utilizes binary instrumentation techniques • Designed to be portable, transparent and robust • Good as an educational tool
DynInst	• Utilizes runtime injection instrumentation technique • Mutator application performs dynamic instrumentation

Back To Table Of Contents

---

8. Acronyms

API - Application Programming Interface

ATOM - Analysis Tools and Object Modification

CLR - Common Language Runtime

JIT - Just In Time

PLDI - Programming Language Design and Implementation

SMART - Specific Measurable Acceptable Realizable Thorough

Back To Table Of Contents

---

9. References

[Alexander00] Alexander, W.P. Berry, R.F. Levine, F.E. Urquhart, R.J. "A unifying approach to performance analysis in the Java environment." IBM Systems Journal, vol. 39, Nov 1, 2000.

[Blevins91] Blevins, David. Bartholomew, Christopher. Graf, John. "Performance analysis of personal computer workstations". Hewlett-Packard Journal, Oct 1991.

[Chi-Keung05] Chi-Keung Luk, Robert Cohn, et al. "Pin: Building Customized Program Analysis Tools

with Dynamic Instrumentation". ACM, June 2005.

[Cohn05] Cohn, Robert and Muth, Robert. "PIN 2.0 User Guide". http://rogue.colorado.edu/pin/docs/1541/. [The online user's guide for the PIN tool].

[DEC94] Digital Equipment Corporation. "ATOM: User Manual." DEC, March 1994.

[Eustace94] Eustace, Alan and Srivastava, Amitabh. "ATOM, A Flexible Interface for Building High Performance Program Analysis Tool". DEC, 1994.

[Fenlason93] Fenlason, Jay and Stallman, Richard. "GNU gprof". http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html [This web page provides information and documentation on the gprof GNU profiler tool.]

[Graham82] Graham, S. Kessler, P. McKusick, M. "Gprof: a Call Graph Execution Profiler". University of California, 1982.

[Guitart02] Guitart, Jordi. Torres, Jordi. Ayguade, Eduard. Bull, J. "Performance Analysis Tools For Parallel Java Applications on Shared-memory Systems". CEPBA, 2002.

[Hansen95] Hansen, Olav. Krammer, Johann. "A Scalable Performance Analysis Tool for PowerPC Based MPP Systems". IEEE, 1995.

[Lindahl05] Lindahl, Michael. "Using Hardware Trace for Performance Analysis". Dr. Dobb's Journal, October 2005.

[LLNL06] "Performance Analysis Tools". http://www.llnl.gov/computing/tutorials/performance_tools/ [This web page contains information on performance consideration, strategies and tools including timers, profilers and analysis toolkits.]

[Marathe03] Marathe, Jaydeep; Mueller, Frank; Mohan, Tushar; de Supinski, Bronis R.; McKee, Sally and Yoo, Andy. "METRIC: Tracking Down Inefficiencies in the Memory Hierarchy via Binary Rewriting". Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2003.

[Martorell05] Martorell, X. Smeds, N. Walkup, R. Brunheroto, J.R., et al. "Blue Gene/L performance tools". IBM Journal of Research and Development, Mar-May 2005.

[Metz03] Metz, Edu. Lencevicius, Raimondas. "A Performance Analysis Tool for Nokia Mobile Phone Software". Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), September 2003.

[Meyers00] Meyers, Nathan. "PerfAnal: A Performance Analysis Tool". Sun Developer Network, March 2000.

[Moore01] Moore, Shirley. Cronk, David. London, Kevin. Dongarra, Jack. "Review of Performance Analysis Tools for MPI Parallel Programs". UTK, 2001.

[Neuendorf01] Neuendorf, Dave. Neuendorf, Bob. Wiener, Richard. "OptimizeIt versus JProbe". Application Development Trends, June 2001.

[Salchner04] Salchner, Klaus. "CLR Profiler". http://www.c-sharpcorner.com/Code/2004/Aug/CLRProfiler.asp [This web page provides information

and resources regarding the Common Language Runtime (CLR) profiler used for .NET applications.]

[Wiki06] "Performance Analysis". http://en.wikipedia.org/wiki/Performance_analysis [This web page provides a basic overview of performance analysis, historical information, and information on many different tools.].

[Williams04] Williams, Chadd C. and Hollingsworth, Jeffrey K. "Interactive Binary Instrumentation". Second International Workshop on Remote Analysis and Measurement of Software Systems, May 2004.

[Worley05] Worley, P. Candy, J. Carrington, L. Huck, K. Kaiser, T. Mahinthakumar, G. Malony , A. Moore, S. Reed, D. Roth, P. Shan, H. Shende , S. Snavely, A. Sreepathi, S. Wolf, F. Zhang, Y. "Performance analysis of GYRO: a tool evaluation". IOP Publishing Ltd., 2005.

Back To Table Of Contents

This report is available on-line at http://www.cse.wustl.edu/~jain/cse567-06/perf_tools.htm
List of other reports in this series
Back to Raj Jain's home page