An Overview of Software Performance Analysis Tools and Techniques: From GProf to DTrace

Justin Thiel, justin@binaryllama.net

Abstract As computer applications have grown in complexity, the ability to fine-tune the performance of an application "by-hand" has been reduced. In response to these difficulties, a wide variety of automated performance analysis techniques have been developed. This paper provides an in-depth look at the types of analysis tools that are currently avaiable, starting with those that implement simple static techiques and then moving onwards towards those that rely on advanced dynamic mechanisms to obtain application statistics. Along the way, various example tools are presented in an attempt to give the reader a better idea of how a real world implementation of each analysis mechanism functions.

Table of Contents

1. Introduction

The act of optimizing a software application to run as fast as possible on a given computing platform has never been a trivial task. In the past, developers achieved such goals by poring over hardware and software manuals, trying to locate the proper combination of assembly language instructions that would result in the level of performance that they desired. Since the computers they had available were all well documented and functioned in a wholly deterministic manner, it was relatively easy for a developer to determine the types of source-code adjustments that would work best on a given architecture.

In the past ten years, however, the software development landscape has evolved dramatically as the general public has embraced computing devices of all types and become increasingly reliant on them to accomplish everyday tasks. As the demand for more sophisticated applications has increased, designers have turned to use the use of higher-level languages and frameworks in an attempt to reduce development costs and remain competitive in the marketplace. As a result, applications have grown increasingly complex in terms of both code size and the interactions that occur within them [Cantrill04]. Therefore, while this "layered" approach to development may save time and money in the short run, it complicates the task of determining whether an observed performance issue is internal to an application or caused by the frameworks that it is built

upon.

Beyond this, computer hardware itself has been forced to change dramatically to keep up with the unrelenting demand for more computing power. The simple single-issue processors of the past have given way to super-scalar designs capable of executing multiple instructions in a single cycle while simultaneously reordering operations to maximize overall performance. As a result, the instructions passed into the processor have become merely a guideline for execution, as opposed to the written rules they were viewed as in the past. Since a developer now has no way to determine precisely how the processor will operate, the act of hand-tuning an application at the assembly level is no longer a straightforward task [Cantrill04].

As early as the 1980s, researchers were aware of these trends in computing and began developing tools to automate the task of performance analysis. These initial tools, much like the computers of the time, were simple in nature and capable of gathering only rudimentary performance statistics [Graham82]. Furthermore, due to technical limitations, these early tools focused exclusively on quantifying application-level performance, and were unable to characterize the effects of items such as library code or the operating system itself. As computers became more complex, however, advanced tools were developed to cut through the layers of abstraction caused by the use of advanced operating systems and other development frameworks to gain meaningful performance statistics for the entire software system [Cantrill04, Hundt00, Luk05]. More recently, hardware designers have begun to embed counters in the CPU that can record cache hit statistics and other meaningful information, thus allowing developers to obtain a complete performance profile for their applications [Noordergraaf02, Anderson97, Kufrin05b].

In the remainder of this paper, the various types of performance analysis tools currently available to developers will be examined in-depth with particular attention paid to real-world implementations. Static tools, which make use of counting and/or sampling methods to obtain rudimentary statistics, are treated first [Graham82, Anderson97]. This is followed by an overview of dynamic tools, which utilize binary instrumentation and probing to provide better insight into application performance [Cantrill04, Luk05]. Finally, information on hybrid tools that combine both static and dynamic analysis techniques is provided along with a set of closing remarks.

Back to Table of Contents

---

2. Static Analysis Tools

The phrase 'static analysis tool' is typically used to classify any performance evaluation mechanism that makes use of non-binary modifications to obtain application statistics. In other words, static analysis tools never modify the binary image of an application, and instead rely on techniques such as source code instrumentation or sampling to obtain their results [Anderson97]. Once recorded in either main memory or on disk, said results can be analyzed to determine any performance bottlenecks that may exist in a program.

The type of information that can be obtained from any specific static analysis tool is largely a function of the type of evaluation techniques that are employed. Typically, a static analysis tool will focus on a few areas of analysis in order to provide meaningful results that would be impossible or impractical to obtain by hand. For instance, source code instrumentation tools such as gprof can provide rudimentary timing data about the various functions defined within an application, while highly advanced sampling tools such as qprof can provide detailed statistical reports about shared library usage and kernel-level interactions [Graham82, HP06]. While such data can certainly simplify the task of tracking down performance analysis problems, other issues inherent to these types of tools prevent their use in certain situations.

By virtue of the fact that static analysis tools are incapable of modifying a running program, any statistical data that a developer wishes to collect must be specified prior to when the application is ran [Srivastava94]. Furthermore, most static analysis tools report results asynchronously, meaning that if a performance issue arises halfway through the instrumented run of an application the developer will not be notified of said issue until after the entire run has completed. As a result, any performance issues that require real-time feedback to diagnose cannot typically be detected by these types of tools.

Beyond this, the use of static analysis tools can cause a number of unintended side effects by virtue of the fact that they require either the insertion of dedicated data collection routines into a set of code or the use of external sampling routines. Such code typically causes system slowdown due to the overhead of gathering statistics, and thus can have dramatic effects upon application performance. Furthermore, this external code can potentially change the behavior of a running program by introducing performance issues that did not exist prior to analysis or falsely alleviating those that previously existed in the program [Anderson97]. Despite these issues and those listed above, however, these types of tools have become invaluable in the real world due to the useful statistics that they can gather.

In the remainder of this section, detailed descriptions of each of the major subtypes of static analysis tools are provided,

starting with the earliest compile-time instrumentation tools and proceeding onward towards modern sampling and compound tools. Along the way, specific implementations of each subtype are discussed in an effort to demonstrate the effectiveness of each static analysis technique in the real world. This is followed by a brief summary meant to assist the reader in determining the types of static analysis tools that might aid them in their own development endeavors.

2.1 Compile-time Instrumentation Tools

Compile-time instrumentation tools (CITs) were first developed during the early 1980's and, as such, are generally regarded as the oldest class of static analysis techniques. Due to their relatively old age, these tools have been largely outclassed by more advanced techniques as the field of performance analysis has evolved. Many developers, however, are familiar with the inner workings of how these tools operate, and thus are unwilling to give them up in favor of what they view to be "unproven" solutions. As a result, CITs still experience widespread usage by developers all around the world [Nikiosha96].

To begin analyzing an application, a CIT first requires access to its entire source repository. This collection of source files is then passed through a specialized compiler that is capable of instrumenting the application as its binary representation is constructed. Typically, this instrumentation consists of a set of counters or monitor function calls that are integrated into the existing function call structure [Graham82, IBM06]. When an instrumented binary is executed, the inserted statements are triggered as function calls are made within the application so that statistical data can be recorded and then analyzed at a later time. The instrumentation method used by a particular tool will have a direct effect upon the types of data that can be obtained. For instance, merely collecting the number of times a function is called via a counter is not sufficient enough to generate a call graph for the entire application [Graham82]. Figure 1 provides a visual explanation of how a typical CIT functions.

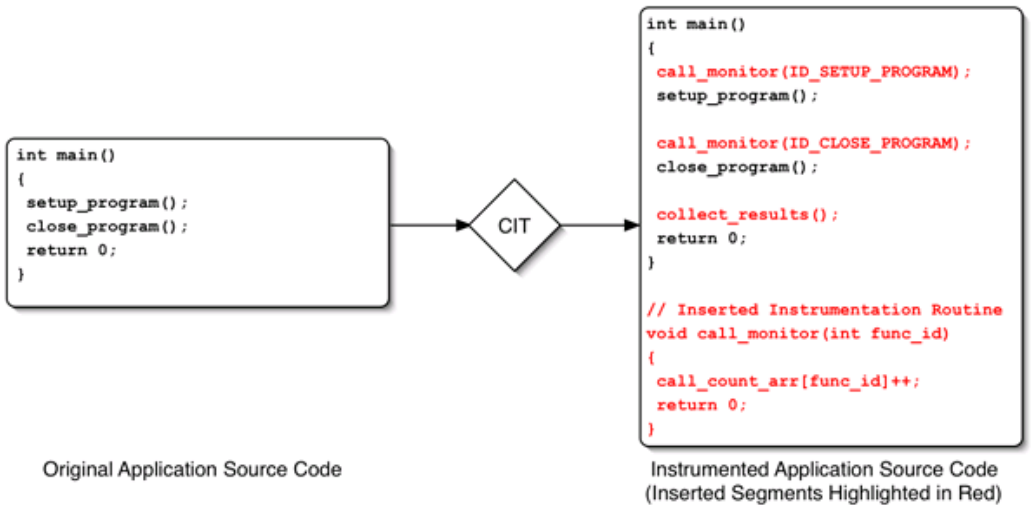


Figure 1: Operational Description of a Typical CIT

Since this instrumentation is done exclusively at the source-code level, performance data for any external libraries or external frameworks that an application utilizes can only be obtained if their source code or a pre-augmented variant is available [Graham82]. In cases where such code is not readily accessible, accurate performance statistics cannot be obtained at all levels of software interaction. As modern large-scale applications have become increasingly reliant upon closed-source dynamic libraries to implement selected portions of their functionality this inherent limitation has emerged as the fatal flaw of all CITs. As a result, these types of tools have become largely relegated for use only with simple programs where a minimum amount of external code is required.

In the increasingly rare cases where the use of CITs is appropriate, however, such tools are capable of generating an impressive array of statistics including accurate function call counts and hierarchal call graphs [Graham82, IBM06]. By analyzing the output of these tools, developers can precisely determine the flow of control through their applications and see what sets of functions are called the most. Once identified, these critical functions can be optimized, thus significantly reducing the overall execution time of the application. To better explain how this process works a description of gprof, the canonical example of a CIT, is provided below.

2.1.1 Overview of Gprof

Gprof is a cross-platform performance analysis tool that is capable of serving both as a CIT and a sampling tool (see Section 2.2). For the purposes of this description we focus on gprof's abilities as a CIT as its role as a sampling tool has been largely

outclassed by more robust alternatives in the over twenty years that have passed since its initial release [Nikiosha96]. As a sampling tool, however, it is still capable of collecting a wide array of useful function-level performance statistics from a targeted application.

To perform a compile-time analysis, gprof relies on two components: an augmenting compiler and a data analysis wrapper. The augmenting compiler is used to insert a monitoring function into a targeted application at the source level along with a set of calls to the monitor that are injected prior to each function call in the program. The resulting instrumented binary is then executed via the wrapper, causing raw performance data to be collected each time a function call is triggered. This data is then analyzed after the program terminates and presented to the end-user in an easy to parse format [Graham82].

By handling all data analysis post-execution, gprof attempts to minimize the performance overhead associated with analysis. Unfortunately, this optimization has a minimal effect at best due to the fact that every function call in the targeted application must be instrumented to generate meaningful results. Since each instrumented function call incurs a slight performance penalty, the cost of performing an analysis scales along with the number of functions in an application and can quickly become significant enough to slow execution time dramatically [Graham82].

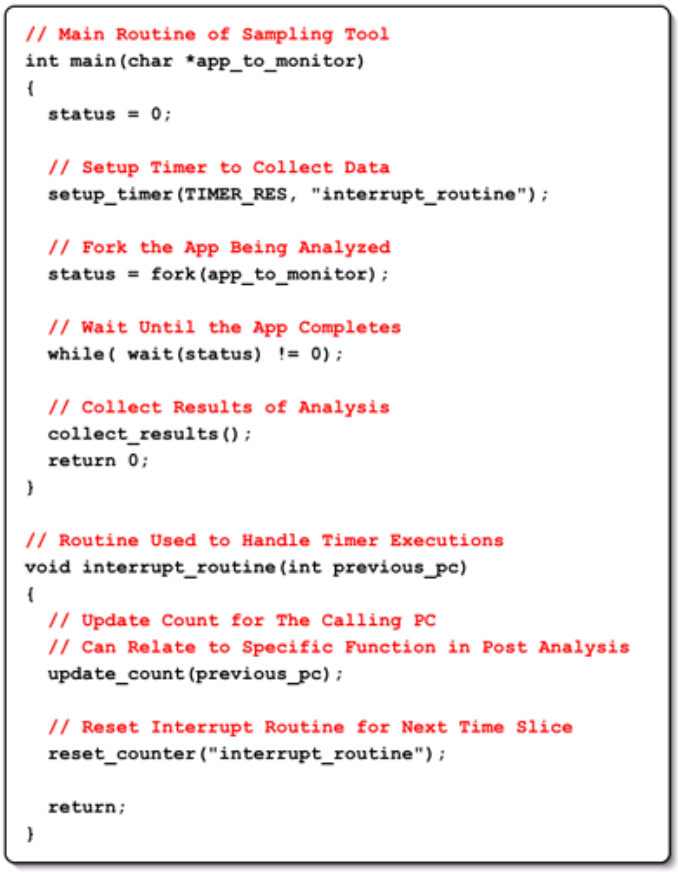
The statistical data collected by gprof is at the function level and primarily consists of call counts, call graphs, and other related information. Such results can be used to reconstruct the internal structure of an application and identify where performance bottlenecks might exist. For instance, if a single function has a relatively high call count compared to others in the application, optimizing it would likely result in a significant impact on performance. By virtue of the fact that gprof is a CIT, however, the statistical information collected from associated external library and/or kernel-level functions is minimal at best thus clouding any results [Graham82].

Despite its flaws, when gprof was first introduced in 1982 it revolutionized the performance analysis field and quickly became the tool of choice for developers around the world. Although newer tools have long since eclipsed it in terms of functionality and accuracy, the tool still maintains a large following because many developers are simply unwilling to switch to other tools. As a result, the tool is still actively maintained and remains relevant in the modern world.

2.2 Sampling Tools

Unlike compile-time tools, which require source-level modification to obtain statistics, sampling tools (STs) have no such restrictions. These tools can therefore be used to analyze a much wider of array of applications, including those that make use of libraries for which no source code is available [Kufrin05b]. Therefore, they have become fairly popular among developers and a number of such tools have been developed for almost any type of computing platform imaginable.

Sampling tools operate by wrapping themselves around the execution of an application, taking control of program flow, and then pausing execution at specific points to record the current state of the system by referencing the program counter (PC) or some other appropriate mechanism. In this manner, any wide array of statistical data can be obtained including call counts, the amount of time spent in each function, or an instruction-by-instruction execution profile [Sun01]. If debug symbols are present in the sampled binary, these results can typically be reported in an easy-to-parse format that is annotated with the proper function names [HP06]. This information can then be utilized to optimize the application in relation to any number of metrics. Figure 2 provides a visual description of how a typical ST functions.



```
// Main Routine of Sampling Tool
int main(char *app_to_monitor)
{
  status = 0;

  // Setup Timer to Collect Data
  setup_timer(TIMER_RES, "interrupt_routine");

  // Fork the App Being Analyzed
  status = fork(app_to_monitor);

  // Wait Until the App Completes
  while( wait(status) != 0);

  // Collect Results of Analysis
  collect_results();
  return 0;
}

// Routine Used to Handle Timer Executions
void interrupt_routine(int previous_pc)
{
  // Update Count for The Calling PC
  // Can Relate to Specific Function in Post Analysis
  update_count(previous_pc);

  // Reset Interrupt Routine for Next Time Slice
  reset_counter("interrupt_routine");

  return;
}
```

Source Code of a Typical Sampling Tool (ST)

Figure 2: Functional Description of a ST

Since STs rely on timer-based sampling, the results obtained from such a tool are most reliable when samples are taken fairly often so that no changes in program state are missed [Levon03]. The immediate downside to sampling often is that doing so causes the execution of an application to slow dramatically, therefore increasing the amount of time it takes to obtain performance results. This, in turn, creates a constant struggle between execution time and statistical accuracy that is often hard to balance. To alleviate this issue, partial analysis runs are often run at a variety of sampling rates so that the required accuracy level can be tuned in.

Since the results obtained from an ST are approximate at best, they should always be taken with a grain of salt. For instance, if a very small function is executed only twice in an application, the ST may only detect one or none of these calls depending upon how often samples are taken. Furthermore, if the ST relies on a software-based mechanism to trigger sampling, its activations can often be delayed due to other interactions in the system, resulting in even less accurate results. To combat this, many STs utilize platform-specific hardware timers to provide more reliable and higher resolution sampling [Zeichick03].

Even when the most accurate sampling mechanisms are employed, however, the usefulness of any ST is directly tied to how its monitoring code is implemented. If constructed as a user-level application, the monitor is only capable of seeing interactions at the application and library levels and thus cannot quantify kernel-level performance issues [HP06]. On the other hand, if the monitor is embedded in a kernel module, then it can observe interactions at all levels of the software hierarchy [Levon03]. To better illustrate these differences, overviews of qprof (a user-level ST) and Oprofile (a kernel-based ST) are provided below.

2.2.1 Overview of Qprof

Qprof is an ST that makes use of a user-level sampling mechanism to obtain performance data on Linux-based operating systems. The tool collects statistics in real-time without the need for binary-level modifications and functions as a shared library. This method of implementation allows for a higher level of integration with the operating system than is possible with a wrapper-based approach [HP06].

To instrument an application using qprof, a developer must first configure a number of environment variables that dictate how long sampling should be done and if events should be recorded at the function, line, or instruction level of granularity. A special environment variable named `LD_PRELOAD` must then be pointed at the qprof shared library to activate the tool. Once set,

the kernel will intercept any application that is executed and inject the qprof library into its memory space before it is allowed to run. Qprof is then able to take control of the binary and orchestrate the sampling of data from the application [HP06].

To allow for flexibility in analysis, the qprof shared library is constructed in a manner that allows for thread-safe operation. Furthermore, due to the fact that the `LD_PRELOAD` environment variable is applied to all processes spawned within a shell, the tool is able to seamlessly handle applications that spawn multiple processes [HP06]. As a result, complex software that relies heavily upon parallel operations can be analyzed in the same manner as a simple "Hello World" program.

Results obtained by qprof are reported in a "flat" profile format that lists how much time is spent in a particular function, line, or instruction and are typically associated with their symbolic names so long as sufficient information is present in the binary (i.e. debug symbols). Due to its architecture, qprof is only capable of collecting data at the application and library (shared, dynamic or static) levels of abstraction and cannot resolve references in the kernel [HP06]. The effect of this limitation, however, is only significant in cases where the kernel is suspected to be the limiting factor in application performance. As such, user-level tools such as qprof are applicable to a wide-range of performance analysis scenarios.

2.2.2 Overview of Oprofile

Oprofile is a kernel-based ST that can record performance statistics from applications running on the Linux operating system via a variety of monitoring mechanisms. For the purposes of this description, it is assumed that Oprofile is operating in the standard "timer" mode whereby system state samples are taken at fixed intervals in manner determined prior to run-time. In this mode, Oprofile performs quite similarly to qprof with the only major differences resulting from the fact that one resides in kernel-space while the other is a user-level application [Levon03].

The tool itself is composed of two primary components: a kernel module and a user-level daemon. The kernel module is loaded into kernel-space at system startup and creates a pseudo character driver (`/dev/oprofile`) that can be used to both configure Oprofile and retrieve results from it via the use of somewhat cryptic commands. In contrast, the daemon runs in the background within user-space and provides software developers with an easy-to-use interface through with the kernel module can be manipulated [Levon03]. By leveraging the interfaces provided by the daemon, Oprofile can be configured to monitor program performance in a relatively straightforward manner.

Internally, the kernel module tracks the samples it takes via an absolute program counter stored within the kernel itself. At pre-specified intervals, these samples are processed in bulk by sending them from the kernel module to the user space daemon. When this occurs, the absolute program counters that were recorded are referenced to the current set of running and just completed processes and are each mapped onto specific binaries on the machine. The set of recorded program counters are then modified such that they become offsets into binary images stored on the disk (i.e. relative program counters) and are passed to the user-level daemon, at which point the names of their associated functions can be determined [Levon03].

As one might imagine, this intricate data collection mechanism enables Oprofile to provide highly detailed performance profiles of applications running on a given system. For instance, kernel-level code and modules can be profiled with instruction-level accuracy and can even be resolved to the name of the function that they are associated with. Furthermore, kernel-level statistics on modules such as the process scheduler can be recorded and used to analyze system performance [Levon03].

The downside to having such power, however, is that Oprofile can be somewhat difficult to configure since it requires that a copy of the source code for the exact kernel of which it will be run be present before a compatible version of the tool can even be generated [Levon03]. Therefore, kernel-level STs such as Oprofile should really only be used in cases where it is suspected that low-level code such as the scheduler or a loadable module is the source of a performance flaw, and therefore a simple user-level ST would be incapable of discovering the problem. In all other cases, such tools are overkill for the task at hand and simpler to configure tools that can produce results in a shorter amount of time are likely available.

2.3 Hardware Counter Tools

Hardware counting tools (HCTs) take advantage of a recent trend in microprocessor design to include a number of on-chip programmable event counters [Anderson97, Kufrin05b]. These counters can be used within supporting tools to record detailed information about the internal state of the processor, thus allowing developers to analyze their applications at the lowest levels of execution. Since all statistics are gathered exclusively at the hardware level, developers need only supply an application binary in order to make effective use these types of tools.

In the past, the only types of machines that provided on-chip event timers were either research devices or high-end servers [Noordergraaf02]. Recently, however, this technology has gone mainstream due to the shrinking size of transistors and the

resulting abundance of area available on modern processor dies. As a result, now even desktop and workstation processors such as the Intel Core Duo and AMD Athlon contain the hardware necessary to implement these types of mechanisms. This, in turn, has led to increased interest in both HCTs and the types of statistics that they are capable of gathering.

Within the processor, event counters are implemented as simple circuits that increment an internal register each time an event they are programmed to watch for occurs. The events that can be monitored vary from processor to processor but typically allow the developer to count items such as the types of cache misses that have occurred or the number of floating point operations that have been executed [Kufrin05a]. The number of counters available on a particular device is strictly limited by the amount of die space devoted to them, but they can typically be reprogrammed or reset at will to change the types of events that they respond to.

HCTs exploit the ability to program these counters by providing developers with methods by which they can easily select the types of events that they wish to monitor. Once configured, the HCT acts as a wrapper around the chosen application, in much the same way as an ST, and requires that the application be paused at specified intervals to collect performance statistics [Anderson97]. Unlike STs, however, HCTs can typically provide more accurate results by virtue of the fact that the counters are automatically handled in the hardware, thus eliminating a significant portion of the guesswork. Small statistical differences between subsequent analysis runs can exist, however, due to variations in when the timer (either in hardware or software) that is used to trigger data collection is fired.

To handle cases when there are not enough hardware resources to obtain counts for all of the events that a user would care to in a single run, HCTs will often include the ability to perform multiple analysis runs using a different subset of events each time. These results of these runs can then be stitched together to form a single meta-analysis file [Kufrin05a]. In doing so, the tool can produce a detailed report of resource usage for a given application that can then be used to tune the system until the minimum possible execution time is achieved. To further illustrate the usefulness of this class of analysis mechanisms a description of an HCT known as Perfsuite is provided below.

2.3.1 Overview of Perfsuite

Perfsuite is a Linux-based HCT that implements a lightweight API through which the hardware counting mechanisms embedded within modern microprocessors can be accessed in a generalized manner. Instrumentation commands defined via this API link into a set of 3rd party kernel modules (perfmon and perfctr) that are then responsible for configuring and monitoring the counting mechanisms inside a given CPU. Currently, these modules only provide counter access services for processors that implement the x86, x86-64 and IA-64 architectures. Due to the transparent nature of the API, however, it should be relatively easy to add support for additional microprocessor types in the future [Kufrin05a, Kufrin05b].

To obtain performance statistics from an application via Perfsuite, a specialized execution wrapper known as `psrun` must be utilized. This wrapper allows the tool to configure the kernel modules and sampling mechanisms used in analysis prior to execution of the program under test. After configuration is completed, the targeted application is then allowed to run freely as samples of the system state are recorded at pre-defined intervals. When execution completes, the raw data that was collected is processed into an XML-compatible form and deposited into a file on the local disk [Kufrin05a, Kufrin05b].

Once all of the required analysis runs for a given application have been completed, the data files generated by `psrun` are typically passed into a utility known as `psprocess` in order to be converted into a human-readable form. The types of performance statistics that `psprocess` is able to report for a given application will depend a great deal upon the how `psrun` was configured prior to the start of execution. In the most general case, however, detailed information about cache hits and misses (both L1 and L2) and the types of instructions (floating point, vector, etc...) issued within in the system can always be obtained [Kufrin05a, Kufrin05b]. Said information is useful for tuning applications at a high level or determining the types of processor resources that are the performance bottlenecks for a given piece of software.

In cases where a more in-depth analysis is needed, `psprocess` is also capable of generating detailed symbolic performance profiles that are indexed by a particular hardware counter. As such, the usage of a particular hardware resource can be directly correlated to specific functions or lines of code within an application. Furthermore, by virtue of the fact that kernel-based performance monitors are utilized, the tool is capable of collecting such information at all levels of the software hierarchy for both single and multi-threaded applications [Kufrin05a, Kufrin05b]. This level of flexibility allows developers to gain an insight into hardware related-issues with a level of accuracy not possible in most other tools.

2.4 Compound Tools

Traditional STs are incapable of profiling an application at all levels of abstraction by virtue of the fact they are unable to monitor hardware events. Similarly, HCTs are unable to provide detailed software performance statistics since they operate

exclusively at the hardware level [Anderson97]. As a result, developers in the past were often forced to utilize both types of applications in they wished to accurately profile their applications at all levels of execution. In response to this, a class of analysis applications referred to as "compound" tools (CTs) were developed.

At its core, a CT simply combines sampling and hardware event monitoring tasks into a single, well-integrated tool. This means that the statistical uncertainties associated with results obtained from both traditional STs and HCTs are inherent to these types of tools as well. Due to the fact that all statistics are collected in parallel, however, the task of stitching together results from disparate tools is eliminated and results can be presented to the developer in a clear and concise manner [Intel06, Zeichick03]. This, in turn, simplifies the task of optimizing an application to run efficiently on a given set of hardware and software components. The overview of Intel's vTune provided below contains an in-depth description of the types of performance statistics that these types of tools are capable of monitoring.

2.4.1 Overview of vTune

vTune is a cross-platform compound performance analyzer produced by Intel that combines the functionality of both an HCT and ST into a single tool. In doing so, vTune is able to function as a broad-spectrum performance analyzer suitable for use in a wide array of statistics gathering tasks on both x86 and IA-64 machines. Since data obtained from the tool can be used to quantify effects at both the software and hardware level, the need for additional monitoring mechanisms can often be eliminated, thus reducing the time required to perform an analysis [Intel06].

The ST implemented by vTune operates at the user layer, supporting analysis at multiple levels of granularity (function, instruction, or source-code) while also allowing the end-user to determine the rate at which samples should be recorded. The tool is able to differentiate the effects of library functions from that of application code, but is incapable of profiling at the kernel-level due to its internal structure. As a result, software-related performance issues can only be pinpointed if their root cause does not lie in the kernel itself [Intel06].

To aid in hardware-level analysis, vTune provides an advanced HCT that is capable of monitoring a number of low-level events that most other mechanisms cannot. For example, the number of "replays" that a particular function or instruction undergoes during execution can be recorded by the tool. Stated briefly, a replay occurs as a result of special-case cache misses within the Pentium IV and requires that the set of instructions affected by the condition be re-issued to correct the mistake. With knowledge of how often this occurs, a developer can tune their application to reduce the chance of replays, thus improving performance [Intel06].

From a compatibility standpoint, vTune is clearly optimized for use with Intel's own processors. This favoritism is most clearly demonstrated by the fact that while vTune's HCT is capable of obtaining even the most obtuse hardware-level performance statistics from Intel's own processors, the functionality is wholly incompatible with non-Intel devices. When an application is being tuned to run exclusively on Intel hardware, however, vTune provides a well-integrated suite of performance analysis tools that are ideal for use in handling any static analysis tasks that a developer might encounter [Intel06].

2.5 Summary

The capabilities that a specific static performance evaluation tool can provide vary widely depending upon the types of analysis techniques that it implements. It is therefore difficult to describe how a "typical" static analysis tool operates. To remedy this, a set of four classifications based upon the basic types of static analysis techniques has been developed. Using this system, each static tool can be grouped into one these classifications and the features and shortcomings that a particular tool provides can be discussed in a generalized format. A comparison constructed in this manner is provided below in Table 1.

Subtype	Features	Shortcomings	Example Tools

Compile-time Instrumentation Tools (CITs)	1. Instruments applications at the source-code level. 2. Can gather call counts for each function in an application. 3. Can generate call graphs to show flow of control through an application. 4. Obtains data in a precise manner; does not rely on statisical methods.	1. Typically cannot gather statistics at the library or kernel level. 2. Requires that an application's entire source tree be available to instrument properly.	1. Gprof 2. Prof
Sampling Tools (STs)	1. Instruments applications via statistical sampling. 2. Can gather call counts for each function in an application. 3. Can determine how much time was spent in each portion of an application. 4. Many implementations are able to obtain statistics at both the library and kernel level.	1. All data obtained is approximate at best due to sampling methods employed. 2. Typically cannot generate call graphs. 3. Can require the presence of specialized timing hardware to obtain reliable results.	1. Qprof 2. Oprofile 3. Prospect
Hardware Counter Tools (HCTs)	1. Instruments applications via statistical sampling. 2. Can make use of hardware counters embedded in modern microprocessors to characterize applications via their hardware usage. 3. Many implementations can characterize hardware usage at both the library and kernel level.	1. All data obtained is approximate at best due to sampling methods employed. 2. Number of hardware counters available limit the types of statistics that can be obtained in a single instrumented run. 3. Requires the presence of specialized hardware that may not be available on all platforms.	1. Perfsuite 2. DCPI 3. Sunfire Link 4. Perfmon 5. Statsmod
Compound Tools (CTs)	1. Combine one or more static analysis techniques into a single tool. 2. Can provide developers with a multi-facted view of their application. 3. Provides inherent benefits of each technique they implement.	1. Run the risk of being a "jack-of-all-trades", but a master of none. 2. Can be more complex to operate than their single-use counterparts.	1. Intel vTune 2. AMD Code Analyzer

Table 1: Comparison of Static Analysis Tools [Intel06, Zeichick03, IBM06, Anderson97, Kufrin05b, Graham82, Noordergraaf02, Hough06].

As can be seen in the table, each type of static analysis tool has its own unique set of features and shortcomings. Despite this, each classification of tool shares the quality that it functions in a static manner, meaning that no modification of an applications' binary image is performed when it is analyzed [Graham82]. As a result, these tools are typically only capable of generalized analysis and not ideal for use in cases where highly detailed performance statistics need to be gathered. In the vast majority of situations, however, a static tool exists that is capable of pinpointing any type of performance issue a developers encounters, provided that it is not buried under multiple layers of abstraction.

Back to Table of Contents

---

3. Dynamic Analysis Tools

Whereas static analysis tools view the binary image of a program as a "black-box" that must never be modified, dynamic analysis tools instead rely on binary-level alterations to facilitate the gathering of statistical data from an application [Srivastava94]. Such alterations are typically inserted while the application is running so that highly accurate statistics can be gathered in real-time [Luk05]. This, in turn, enables dynamic analysis tools to provide insights into program performance that would not be possible to obtain via static examination techniques.

The types of analysis techniques that dynamic tools typically employ can be classified into one of two groups: binary instrumentation or probing. Broadly speaking, tools that make use of binary instrumentation are capable of injecting customized analysis routines into arbitrary locations within an application binary to record a wide array of performance data [Srivastava94, Luk05]. Probing tools, on the other hand, rely on support routines embedded in shared libraries and the kernel that can be activated on-the-fly to obtain detailed information about a component's internal status, thus facilitating analysis at multiple levels of abstraction [Cantrill04]. By making use of the information provided by either analysis technique, developers can profile their applications with high granularity, thus simplifying the task of performance tuning. As is the case with static analysis tools, however, such flexibility in analysis does not come without compromises.

The tradeoffs a developer must accept to make use of dynamic analysis techniques primarily result from the methods that such tools employ to obtain performance statistics. More precisely, by relying on the use of binary-level modifications, these tools effectively modify the structure of the applications that they profile. As a result, programs tend to run somewhat slower while being analyzed due to the increased overhead caused by the insertion or activation of performance monitoring routines. Beyond this, the "random" insertion of code into a binary can affect the flow of instructions through a processor pipeline, thus modifying the performance characteristics of the application [Luk05]. As a result, it is generally recommended that developers only instrument code segments that they believe to be probable sources of performance issues so that the effect upon the rest of the application is minimized.

Despite these shortcomings, efforts devoted towards the research and development of dynamic analysis tools have increased at a steady rate. This is likely a side effect of the increased use of high-level languages and software frameworks by developers in an attempt to develop applications at a rapid pace. Such mechanisms often result in the creation of convoluted software with multiple levels of functional abstraction, making it difficult to pinpoint performance issues using traditional static analysis tools. The detailed overviews of binary instrumentation and probing tools provided in the remainder of this section serve to further illustrate why dynamic analysis techniques are ideal for use in profiling modern applications.

3.1 Binary Instrumentation Tools

For the purposes of this discussion, we focus exclusively on tools that rely on real time instrumentation. As a result, tools such as ATOM that rely on off-line binary instrumentation are largely ignored, although many of the concepts provided here apply to them as well [Srivastava94]. These tools are disregarded primarily because they have largely been outclassed by their "dynamic" counterparts and are thus no longer widely used within the development community [Luk05].

Binary instrumentation tools (BITs) rely on binary modification techniques to obtain performance statistics for the applications they analyze. More specifically, these tools are able to inject analysis routines into an application while it is executing in order to record information about its internal structures. The results of this analysis can then be examined to determine where performance bottlenecks exist within a program [Luk05, Zhao06].

Prior to analyzing a given application, a BIT must first gain access to its instruction stream. Typically, this is done by linking the tool into a set of hooks provided by the host operating system that allow one application to modify the memory space of another. Once inside an application's memory space, the BIT can then inject analysis routines into the program and setup a structure (via signals or a similar method) that will allow the tool to maintain control once execution is resumed. The application is then allowed to proceed as normal from the point at which it was interrupted, processing any inserted routines that it happens to come across [Luk05, Zhao06, Kumar05]. Figure 3 provides a visual description of this process.

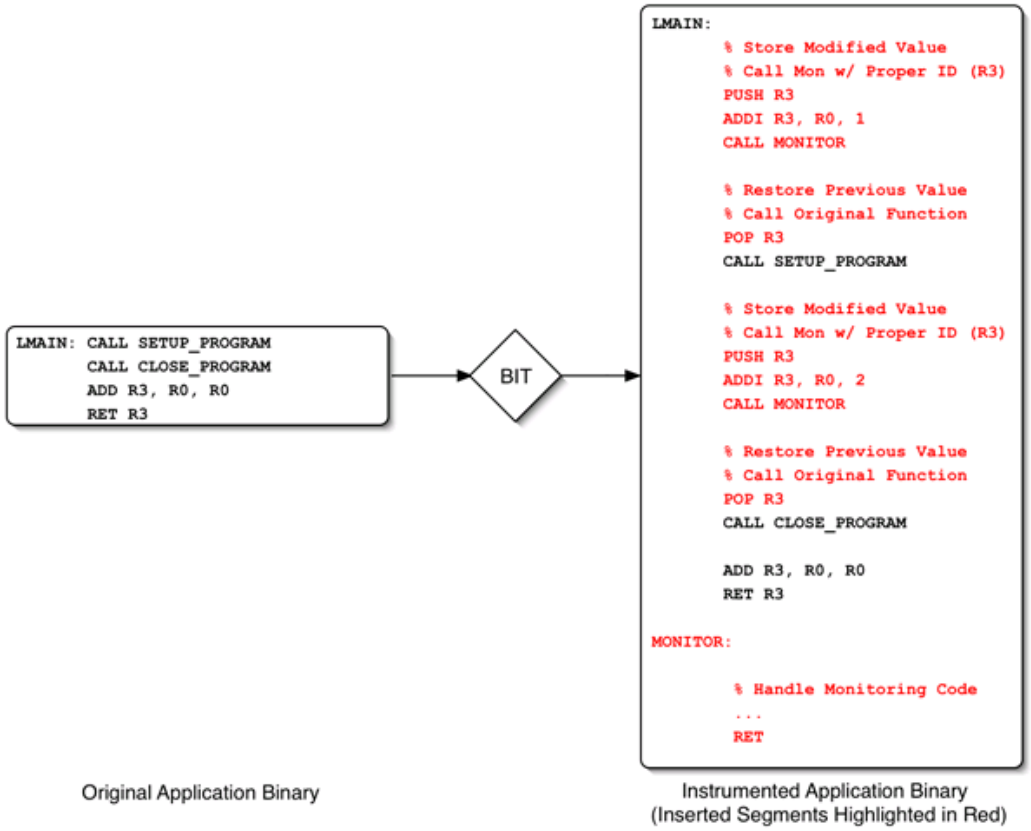


Figure 3: Operational Description of a Typical BIT

Typically, a BIT is not distributed with a large library of analysis routines and instead relies on the developer to hand-code the functionality they require. To facilitate this task, a tool-specific API is provided that dictates what types of events can and cannot be monitored within the system. By coding to the specifications of such an API, developers can then slowly build up their own collection of analysis routines from the ground up or simply enhance those that are already available [Luk05]. Since these APIs are tool-specific, however, it usually not possible for a developer to transition from one BIT to another without having to re-write their entire set of analysis routines.

In exchange for abiding by the constraints of these proprietary frameworks, BITs are able to provide developers with advanced functionality not typically seen in other types of tools. For instance, many BITs include support for selectively modifying the set of analysis routines that are associated with an application at any point during its execution. This is done via the access method setup when the BIT initially takes control of the program [Luk05]. By taking advantage of this feature, applications under test can be made to run at full-speed when statistics gathering is not required, and then instrumented "on-the-fly" when performance issues arise during execution. This, in turn, minimizes the performance impact typically associated with application profiling and potentially allows statistics to be gathered in a timelier manner.

Beyond this, BITs also provide the ability to instrument applications at multiple levels of the software hierarchy, thus increasing the breadth of performance statistics that can be obtained from a given program. Such functionality is made possible by virtue of the fact that library functions are typically loaded into the same memory space as the instrumented application at startup time, and therefore appear as just another segment of code within the binary image. Unfortunately, this ability does not extend to kernel-level code because it resides in an isolated memory space [Luk05, Kumar05, Zhao06].

Despite their advantages, the use of BITs is not quite commonplace in the world of software development. This is likely due to certain misconceptions about how these tools operate and what effects they have upon application performance. There are, however, a number of research-oriented tools available that implement a variety of binary instrumentation methods. A description of a binary tool known as Pin is provided below to give the reader a better idea of how binary instrumentation is done within a real system.

3.1.1 Overview of Pin

Pin provides a Linux-based software development framework through which portable dynamic instrumentation routines can be defined. Once created, these routines can be compiled into an x86, x86-64, or IA-64 binary format and then injected into an

application at runtime to gather performance statistics [Luk05]. In this manner, applications can be profiled on multiple platforms via a single set of instrumentation routines, thus reducing development costs and speeding up the analysis process.

To facilitate the creation of instrumentation routines, Pin implements a C-based development API. Through the use of this API, developers can define precisely where instrumented code should be injected into an application binary to collect performance statistics. Multiple granularities of instrumentation are supported, allowing code to be inserted before or after function calls, specific instructions, or the occurrence of specific blocks of code [Luk05]. This, in turn, allows developers to pinpoint performance issues within an application to their precise origins.

Instrumentation routines defined by developers can be injected into a targeted application at runtime to gather statistics. To facilitate this injection process, Pin relies on the Ptrace debug interface provided by the Linux operating system. Through this interface, the tool is able to gain control of any application that is executing on the system and inject the Pin executable into its address space. Once injected, Pin is able to take control of the application and begin inserting instrumentation routines to gather analysis data [Luk05].

In a typical program, the various jump and branch instructions present in the source code are relative, meaning, that if any additional instructions (such as instrumentation routines) are inserted into the binary the targets of these instructions will no longer resolve correctly. To address this issue, Pin performs just-in-time (JIT) compilation of all application code into an internal format and executes programs within a virtual machine (VM). By making use of a VM, the tool is able to redirect jumps and branches to their proper destinations, as well as insert instrumentation routines into the code stream without affecting other calculations within the code. Furthermore, the use of a VM allows Pin to provide a degree of portability in analysis due to the fact that the only machine-specific component of the tool is the code generator that is used to convert instructions from the VMs internal format to the native format used by a given architecture. By leveraging this functionality, Pin is currently able to provide support for the x86, x86-64, and IA-64 architectures, and is expected to add support for the ARM platform in the near future [Luk05].

The downside of the VM-based execution mechanism that Pin employs is that the overhead of instrumentation can become quite high. This is due to the fact that code is no longer directly executed on the system; instead it is interpreted, modified, re-compiled, and then converted into a native format for execution. To handle this issue, Pin provides a set of code optimization routines that operate at both the VM and native levels. By taking advantage of these features, Pin is able to reduce the instrumentation overhead to the point where only a 2 to 3x increase in execution time is incurred. As a result, developers can take advantage of the advanced functionality provided by Pin without incurring a performance penalty much higher than that caused by less flexible static analysis tools [Luk05].

3.2 Probing Tools

In the most basic terms, a probing tool is a mechanism that is able to selectively activate instrumentation routines that are embedded within software at all levels of abstraction. As such, these tools are capable of obtaining performance-related statistics from not only an application but also the various libraries and kernel routines associated with its execution. These results can then be analyzed to determine not only where issues exist within an application, but also what chain of events caused them to occur [Cantrill06].

Unlike BITs, which allow for the insertion of arbitrary code into an application, probing tools do not allow for the "random" instrumentation of code. They instead rely on a set of well-defined "probes" that allow various data values and statistics to be collected. Such probes can be defined by applications, libraries, or even kernel routines and then activated at application run-time. Upon activation, the probes are enabled in a manner similar to that used by BITs to inject analysis routines. The instrumented application is then allowed to proceed forward in execution as it typically would, triggering probes as it moves through the system [Cantrill06].

Although the probe-based approach to performance analysis may seem somewhat limited compared to the methods employed by BITs, these systems are typically far easier to use and obtain useful results from. This is because all possible probes are pre-defined and typically labeled to indicate precisely what they monitor [Cantrill04]. As a result, the developer does not have to concoct a customized analysis routine for every item they wish to monitor. This, in turn, can allow developers to quickly diagnose performance issues, thus reducing the costs of software development.

Probe-based systems are in no way flawless, however, and despite their many benefits such tools still suffer from a number of implementation-related issues. For instance, if a probe does not already exist for a particular data value or statistic embedded within an external library or the kernel then there is no way to design one that can retrieve said value [Cantrill04]. As a result, a probing tool will not necessarily be able to pinpoint the source of every performance issue within a given application. In addition to this, probing tools require extensive support at both the kernel and library levels in the form of pre-defined probe

routines in order to be of any use to developers. Beyond this, it bears mentioning that since probes are functionally equivalent to injected analysis routines, a performance hit will be incurred within an application when it is instrumented via a probe-based tool [Cantrill04].

Due to the level of support required at all levels of the software hierarchy, probing tools are still somewhat of a rarity. In fact, at the time of this writing the only widely known implementation is DTrace, a tool developed by Sun Microsystems for use with the Solaris operating system. An overview of this tool is provided below as a means of better explaining how probing mechanisms operate in the real world.

3.2.1 Overview of DTrace

DTrace is an analysis tool that operates via a set of probes embedded within both the Solaris operating system and its supporting libraries. By selectively activating these probes, various data points within the software hierarchy at the application, library, or kernel level can be recorded and correlated with one another to form a highly detailed profile of any targeted application [Cantrill06]. Developers can then use this information to pinpoint application performance issues to their precise origins.

The probes utilized by Dtrace are not defined by the tool itself, but instead supplied by a set of instrumentation providers. Providers function independently of each other and implement subsets of the probes via kernel modules that Dtrace can gain access to. Once loaded, each provider communicates the data points that it is capable of instrumenting to DTrace via a well-defined API. DTrace can then instrument the system in a dynamic fashion, activating only those probes selected by the end-user [Cantrill04, Cantrill06]. As a result, reducing the number of probes used to analyze a particular application will have the positive side effect of reducing the performance overhead caused by the tool. In other words: reducing the amount of data that is collected will enable results to be obtained in a more timely fashion.

To enable probes and define instrumentation routines via Dtrace, a C-like scripting language known as D is utilized. D provides support for typical scripting language features such as variables and control structures, while also implementing advanced features such as data aggregation and predicates. Data aggregation allows developers to combine information from multiple probes into a single data source, thus simplifying the task of identifying related probes. Predicates, on the other hand, give developers the ability to define logical expressions that are used to determine when the data from a given probe should be recorded, thus allowing useless information to be discarded before it is committed to memory. By combining these mechanisms with other facilities provided by the scripting language, developers can easily form complex instrumentation routines that collect data from probes in a logical and organized manner [Cantrill04, Cantrill06].

Once defined in the D language syntax, instrumentation routines are passed into DTrace where they are activated and set about the task of data collection. In doing so, these routines are converted into a simplified instruction set known as the "D Intermediate Format" (DIF). This simple instruction is set is coupled with a VM at the kernel level and is primarily used to ensure safety in probe firing. A safeguard such as this is required because instrumentation routines can be activated by probes attached to delicate operations in the system such as process scheduling and could therefore potentially disrupt calculations if not verified prior to execution. By using the DIF, delicate operations in the system can be safeguarded against the effects of instrumentation, allowing analysis data to be collected without irreparably affecting the state of the system [Cantrill04, Cantrill06].

Despite the advantages offered by the probing system that Dtrace implements, the tool suffers from a number of shortcomings. For instance, the tool is unable to obtain performance data that is not explicitly defined via a probe. Therefore, while DTrace does allow developers to activate over 30,000 probes there is still a chance that the data point required to analyze a specific problem is not available. Beyond this, the use of Dtrace is currently tied exclusively to the Solaris operating system, making it a somewhat useless tool for developers working on other platforms [Cantrill06]. It should be noted, however, that efforts are currently underway to port the tool to both FreeBSD and Mac OS X [Cantrill06, Chaes06]. If such efforts are successful, DTrace could potentially become the de-facto analysis tool at some point in the near future in much the same way gprof was in the past.

3.3 Summary

As is the case with static analysis tools, the capabilities that a specific dynamic performance evaluation tool can provide depend directly upon the types of analysis techniques that it implements. Broadly speaking, dynamic analysis tools can be classified into one of two subtypes: binary instrumentation tools (BITs) or probing tools. Using these classifications, a particular dynamic tool can be placed into one of these two subgroups and then described in a generalized manner. A comparison of dynamic analysis tools assembled via this approach can be seen below in Table 2.

Subtype	Features	Shortcomings	Example Tools
Binary Instrumentation Tools (BITs)	1. Instruments applications at the binary level. 2. Can analyze in real time to obtain statistics from a running application. 3. Make use of generalized APIs to customize analysis and instrumentation routines that can monitor arbitrary events in an application. 4. Can often insert instrumentation into the various libraries that an application makes use of.	1. Typically cannot gather statistics at the kernel level. 2. Analysis and instrumentation routines created with one tool are often incompatible with all others. 3. Can require a significant investment of time to obtain useful results.	1. Pin 2. Vertical Profiling 3. Adept 4. Dynamo RIO
Probing Tools	1. Instruments applications at the binary level. 2. Many implementations are able to obtain statistics at both the library and kernel level. 3. Relies on a set of pre-defined probes to determine the types of values and events that can be monitored. 4. Make use of generalized APIs that can create customized analysis routines that aggregate and examine values of various probes.	1. Additional probes beyond the set that is provided cannot be created, thus limiting the scope of the tool. 2. Require specialized support at all levels of the software hierarchy to implement. 3. Probes available in one implementation may not match those available on a different system. 4. Can require a significant investment of time to obtain useful results.	1. DTrace 2. DProbe 3. Linux Trace Toolkit

Table 2: Comparison of Dynamic Analysis Tools [Cantrill04, Cantrill06, Luk05, Kumar05, Zhao06]

As can be seen in the table, binary instrumentation and probing tools take two very different approaches to dynamic instrumentation. Whereas BITs typically allow arbitrary instrumentation at both the application and library levels, probing tools only allow for selective instrumentation, but are able to do so at all levels of the software hierarchy [Cantrill04, Kumar05]. In a sense, it can be said that the BIT approach allows for a more detailed analysis at fewer levels of abstraction, while probing tools allow for more generalized analysis across a larger set of interactions. Therefore, when selecting a binary tool for a specific performance evaluation task, the developer should keep in mind the types of relationships they wish to analyze before choosing any individual implementation.

Beyond this, it should be noted that despite their apparent differences, all binary tools share a number of common shortcomings. Chief among these being the fact that binary tools have a somewhat steep learning curve associated with their use caused by the fact that they typically rely on proprietary scripting language to define instrumentation routines [Luk05, Cantrill04]. Developers should therefore be sure to allocate adequate time towards learning the intricacies of a given tool before trusting any performance analysis results that are generated from it

Back to Table of Contents

---

4. Hybrid Analysis Tools

The developers of hybrid analysis tools take a blended approach to application profiling by combining both static and dynamic instrumentation techniques to form a unified statistics-gathering module [Hundt00, Srivastava01]. This, in turn, allows them to selectively implement only the most effective features that each class of tools has to offer. As a result, these tools are often capable of providing a level of utility unmatched by any single-purpose analysis mechanism currently available.

Due to the blended and variant nature of these types of tools, it is somewhat difficult to provide an accurate description of how

a "typical" implementation would function. It is worth nothing, however, that the run times of instrumented applications are typically higher when hybrid tools are utilized than when any other single-purpose profiling mechanism is selected. This additional overhead can be attributed to the fact that certain types of static and dynamic instrumentation mechanisms cannot execute in parallel with one another, and thus the performance issues inherent to both will effectively add together when such techniques are applied to the same application [Hundt00]. Hybrid analysis tools should therefore only be used in cases where they are capable of offering the developer useful statistical data that cannot be obtained through any other means.

As is the case with dynamic instrumentation techniques, hybrid analysis tools have not yet been widely adopted by the vast majority of software developers. This is likely due to the fact that no commercial-level implementations currently exist, and thus developers are somewhat reluctant to accept what they view as "unproven" technology. A number of interesting research-oriented implementations of such tools do exist. The remainder of this section contains an in-depth description of one such tool (HP Caliper) in an effort to show how hybrid analysis techniques can be applied in the real world.

4.1 Overview of HP Caliper

HP Caliper is a framework for HP-UX that aids in the creation of hybrid analysis tools for IA-64 microprocessors. Through the use of foundations provided by the software, developers can design customized data gathering routines that implement support for both hardware counters and binary instrumentation techniques [Hundt00]. Once completed, these routines can be combined to form a semi-independent tool that is capable of analyzing application performance issues in multiple levels of the software hierarchy.

In order to define instrumentation routines via HP Caliper a C-based API is used. By programming to the specifications of this API, hardware-counter manipulation routines can be defined alongside those that depend on dynamic instrumentation. Routines are typically attached to specific events in the system such as function calls, signal raised via processes, or the firing of a timer. In this manner, a rich set of performance data can be collected for analysis purposes [Hundt00].

To gather hardware-level statistics via the HP Caliper API, the Performance Measurement Unit (PMU) included onboard IA-64 microprocessors is leveraged. This module implements a set of filters in hardware that allow data gathering at a much finer level of granularity than is possible with the performance counters onboard traditional CPUs. These filters can operate at both the instruction and address levels allowing hardware statistics to be gathered in a bounded manner [Choi02]. In doing so, both process and function-level data statistics can be collected with a minimal amount of software-based overhead.

Once defined via the API, instrumentation routines must be injected into a running program in order to gather statistical data. To perform this task, HP Caliper makes use of the ttrace system call provided by HP-UX. After executing this system call, the targeted application pauses and passes control to a kernel level routine that then injects the HP Caliper executable into the programs address space. HP Caliper is then passed control of the application and sets about instrumenting it in the manner specified by the end-user [Hundt00].

Instrumentation in HP Caliper is done by inserting break instructions at the entry points to each function in a targeted applications address space. Once these breaks are inserted, the application is allowed to proceed as normal from the point where HP Caliper took control of execution. Each time a break is encountered, however, HP Caliper regains control of the program and inserts any instrumentation code associated with the routine to which the break is attached. In this manner, applications are instrumented in a "lazy" fashion, allowing the tool to modify only those functions that are actually executed. This, in turn, minimizes the performance impact that is incurred by using the tool [Hundt00].

While the architecture described above allows HP Caliper to perform a number of powerful analysis tasks, it suffers for a few significant shortcomings. For example, the tool cannot instrument kernel routines due to the fact that the kernel executes in a separate address space than user-level applications. It is therefore not useful in cases where kernel-level code is the source of a performance bottleneck. Furthermore, the tool is primarily targeted for use on the HP-UX operating system, with only a partial Linux port currently available [Hundt00]. As a result, it is nothing more than a novelty for the vast majority of developers. For the small subset that can take advantage of the tool, however, HP Caliper provides a rich feature set that allows developers to pinpoint application performance issues across multiple levels of system interaction.

Back to Table of Contents

Summary

As software has grown in complexity, developers have become increasingly reliant on the use of frameworks and abstraction

layers to implement new functionalities in a timely manner. In doing so, the ability to understand the precise manner in which a given segment of code operates has been reduced, leading to software projects in which no one person quite understands how the entire application functions [Cantrill04]. As a result, it has become nearly impossible to optimize applications using the manual (or "by-hand") tuning techniques of the past.

In response to this issue, a variety of automated performance analysis tools have been developed. Typically, each of these tools has attempted to address the needs of a specific subset of developers or provide support for a specialized type of analysis. Generally speaking, however, any currently available tool can be placed into one of three subgroups based upon the types of analysis mechanisms it implements. These classifications, commonly known as static, dynamic, and hybrid, are covered in-depth in Sections 2 through 5 of this paper, respectively.

Briefly stated, static tools are designed to operate strictly outside of application binaries, using source-code instrumentation or sampling techniques to obtain performance results [Graham82, HP06]. Dynamic tools, on the other hand, rely on binary-level modifications of applications and their supporting libraries to obtain performance results with a potentially higher resolution than their static contemporaries [Luk05, Cantrill04]. Finally, hybrid tools combine both static and dynamic techniques in an attempt to obtain results at a level of clarity not possible with any single-use tool [Hundt00, Srivastava01]. A high level overview of the features and shortcomings of each tool type can is provided below in Table 3.

Type	Features	Shortcomings	Example Tools
Static Analysis Tools	1. Make use of source-code modifications or sampling techniques to obtain application performance data. 2. Many implementations can obtain statistics at kernel, library, and hardware levels. 3. Typically simple to configure; require a minimal investment of time to obtain results from.	1. In cases where sampling techniques are utilized the results obtained are approximate at best. 2. Limited in scope in terms of the types of values and events that can be monitored.	1. GProf 2. QProf 3. Oprofile 4. Perfsuite 5. Intel vTune
Dynamic Analysis Tools	1. Make use of binary level instrumentation techniques to obtain performance data. 2. Some implementations can obtain statistics at both the library and kernel level. 3. Allow for the monitoring of a wide range of values and events via the use of customizable analysis and/or instrumentation routines.	1. In cases where probing is utilized, the set of probes that are available can be somewhat limiting. 2. Typically have a steep learning curve; can be difficult to obtain useful results from. 3. Analysis and/or instrumentation routines created with one tool are typically incompatible with all other implementations. 4. Often intrisically tied a specific operating system.	1. Pin 2. DTrace
Hybrid Analysis Tools	1. Combine Static and Dynamic Instumentation Techniques to Create a Flexible Tool. 2. Allow for the monitoring of a wide range of values and events via the use of customizable analysis and/or instrumentation routines. 3. Many implementations can obtain statistics at kernel, library, and hardware levels.	1. Implementations often focus on one analysis technique, with support for all others merely "tacked" on. 2. Analysis and/or instrumentation routines created with one tool are typically incompatible with all other implementations. 3. Can require a significant investment of time to obtain useful results.	1. HP Caliper 2. Vulcan

Table 3: High-Level Feature Set Listing for Each Type of Analysis Tool [Intel06, Zeichick03, IBM06, Anderson97, Kufrin05b, Graham82, Noordergraaf02, Hough06, Cantrill04, Cantrill06, Luk05, Kumar05, Zhao06]

As can be seen in the above table, each type of analysis tool implements a somewhat unique feature set and is aimed at diagnosing a different set of performance issues. It is therefore sometimes difficult to choose the ideal tool for a given analysis task. In most cases, static analysis tools are used when generalized performance statistics about a program are required [Anderson97]. Dynamic analysis tools, however, are ideal for use when attempting to pinpoint a specific performance issue in an application [Cantrill04]. Hybrid analysis tools, by virtue of their mixed construction, can technically be used to handle any type of performance evaluation task, although care should be taken to ensure that the selected tool supports static and dynamic analysis in a balanced manner.

Regardless of the task at hand, analysis tools provide developers with a level of flexibility and functionality unmatched by classical (manual) optimization methods. By taking the time to learn how to properly utilize the various types of performance analysis tools available, developers can begin to sift through the abstractions in their programs and uncover the performance issues that lie deep within their code. Once identified, these issues can often be eliminated with minimal effort, thus creating a significant boost in performance for only a small investment of time. As a result, it can reasonably be assumed that as applications continue to increase in complexity, developers will become increasingly reliant upon analysis tools to pinpoint the performance issues within them.

Back to Table of Contents

References

1. [Luk05] Luk, C., et. al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05).
   http://rogue.colorado.edu/Pin/docs/papers/pin-pldi05.pdf
   Provides an Overview of Pin, a Binary-based Dynamic Analysis Tool; Also Includes Information About a Wide Variety of Other Performance Evalutation Techniques.

2. [Cantrill04] Cantrill, B., et. al., "Dynamic Intrumentation of Production Systems," Proceedings of the USENIX 2004 Annual Technical Conference (USENIX '04),
   http://www.sun.com/bigadmin/content/dtrace/dtrace_usenix.pdf
   Provides an Overview of DTrace, a Probe-based Dynamic Analysis Tool.

3. [Graham82] Graham, S. L, et. al.,"Gprof: A Call Graph Execution Profiler," Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN '82).
   http://www.stanford.edu/class/cs295/papers/p120-graham.pdf
   Provides an Overview of gprof, a Source-level Static Analysis Tool.

4. [Hundt00] Hundt, R., "HP Caliper: A Framework for Performance Analysis Tools." IEEE Concurrency [Volume 8, Issue 4].
   Provides an Overview of HP Caliper, a Hybrid Analysis Tool.

5. [Kufrin05b] Kufrin, R., "Measuring and Improving Application Performance with PerfSuite," Linux Journal [Issue 135].
   http://www.linuxjournal.com/article/7468
   Provides an Overview of Perfsuite, a Hybrid Analysis Tool.

6. [Anderson97] Anderson, J. M., et. al., "Continuous Profiling: Where Have All the Cycles Gone?," Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97).
   ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/technical-notes/SRC-1997-016a.pdf
   Provides an Overview of DCPI, a Binary-based Dynamic Analysis Tool; Also Includes a Wide Variety of Generalized Information About Other Performance Analysis Tools.

7. [Cantrill06] Cantrill, B., "Hidden in Plain Sight" Queue [Volume 4, Issue 1].
   http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=361&page=1
   Provides an Overview of DTrace; Particular Attention Paid to Real World Use Cases of the Tool.

8. [HP06] Hewlett Packard Corporation, "Installing and Using QProf,"
   http://research.hp.com/research/linux/qprof/using_qprof.php4
   Provides an Overview of qprof, a Sample-based (User-Level) Static Analysis Tool.

9. [Levon03] Levon, J., "OProfile Internals,"
http://oprofile.sourceforge.net/doc/internals/index.html
Provides an Overview of OProfile, a Sample-based (Kernel-level) Static Analysis Tool.

10. [Kufrin05a] Kufrin, R., "Perfsuite: An Accessible, Open Source Performance Analysis Environment for Linux," Proceedings of the Sixth International Conference on Linux Clusters (LCI-05),
http://perfsuite.ncsa.uiuc.edu/publications/LCI-2005.pdf
Provides an Overview of Perfsuite, a Static Analysis Tool that Makes Use of Hardware Counting Mechanisms.

11. [Intel06] Intel Corportation, "vTune Performance Analyzer 8.0 for Linux: Getting Started Guide,"
http://cache-www.intel.com/cd/00/00/24/50/245027_245027.pdf
Provides an Overview of vTune, a Compound Static Analysis Tool.

12. [Kumar05] Kumar, N., et. al., "Low Overhead Program Monitoring and Profiling," Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE '05).
http://www.cs.pitt.edu/~childers/papers/PASTE05-Insop.pdf
Provides an Overview of Ins-Op, a Tool Designed to Optimize Instrumentation Routines Generated via Dynamic Analysis Tools.

13. [Srivastava94] Srivastava, A. and Eustace, A., "ATOM: A System For Building Customized Program Analysis Tools," Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94).
http://www.csl.cornell.edu/~sam/ece699/p528-srivastava_atom_retrospective.pdf
Provides an Overview of ATOM, an Offline Binary-Based Dynamic Analysis Tool.

14. [Zhao06] Zhao, Q., et. al., "DEP: Detailed Execution Profile," Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06).
http://www.cs.virginia.edu/~pact2006/program/pact2006/pact064-zhao1.pdf
Provides an Overview of Adept, a Binary-based Dynamic Analysis Tool.

15. [IBM06] IBM Corporation, "AIX Performance Tool Guide and Reference,"
http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.prftools/doc/prftools/prftools.pdf
Contains Information on Various Static and Dynamic Performance Analsyis Tools.

16. [Noordergraaf02] Noordergraaf, L., et. al., "SMP System Interconnect Instrumentation for Performance Analysis", Proceedings of the 2002 ACM/IEEE Conference on Supercomputing.
http://www.supercomp.org/sc2002/paperpdfs/pap.pap158.pdf
Provides an Overview of the Sunfire Link Hardware-based Statistical Sampling System.

17. [Zeichick03] Zeichick, A., "AMD Code Analyst: Getting In Touch With Your Inner Code," DevX,
http://www.devx.com/amd/Article/17305
Provides an Overview of AMD Code Analyst, a Compound Static Analysis Tool.

18. [Nikiosha96] Nikiosha, S., et. al., "Process-labeled Kernel Profiling: A New Facility to Profile System Activities," Proceedings of the USENIX 1996 Annual Technical Conference (USENIX '96),
http://www.usenix.org/publications/library/proceedings/sd96/full_papers/nishioka.txt
Provides an Overview of Various Static and Dynamic Performance Analysis Techniques.

19. [Srivastava01] Srivistava, A., et. al., "Vulcan: Binary transformation in a distributed environment," MSR-TR-2001-50,
ftp://ftp.research.microsoft.com/pub/tr/tr-2001-50.pdf
Provides an Overview of Vulcan, a Hybrid Peformance Analysis Tool.

20. [Chaes06] Chaes, Brendon, "Mac OS X Leopard Gets Sun's DTrace,"
http://www.zdnet.com.au/news/software/soa/Mac_OS_X_Leopard_gets_Sun_s_DTrace/0,130061733,139265767,00.htm
News Article Outlying Apple's Plans to Improve DTrace Support in the Next Revison of MacOS X.

21. [Choi02] Choi, Y., "Design and Experience: Using the Intel Itanium 2 Processor Performance Monitoring Unit to Implement Feedback Optimizations," Proceedings of the 2nd Workshop on EPIC Architectures and Compiler Technology (EPIC-2),
http://systems.cs.colorado.edu/EPIC2/papers/s2-3-choi.pdf
Contains Information on the Advanced Hardware-based Sampling Mechanisms Included Onboard IA-64 Processors.

22. [Hough06] Hough, R., et. al., "Cycle-Accurate Microarchitecture Performance Evaluation", IEEE Workshop on Introspective Architecture (WISA '06), http://www.arl.wustl.edu/projects/fpx/projects/liquid_arch/publications/statsmod.pdf
Provides an Overview of Statsmod, a Static Analysis Tool that Makes Use of Hardware Counting Mechanisms.

23. [Sun01] Sun Microsystems, "Analyzing Program Performance with Sun Performance Workshop", http://192.18.109.11/806-7989/806-7989.pdf
Provides an Overview of the Analysis Capabilities Provided by the Sun Performance Workshop.

Back to Table of Contents

List of Acronyms

AMD	Advanced Micro Devices
API	Application Programming Interface
ARM	Advanced RISC Machine
ATOM	Analysis Tool for Object Manipulation
BIT	Binary Instrumentation Tool
CIT	Compile-time Instrumentation Tool
CT	Compound Tool
HCT	Hardware Counter Tool
HP	Hewlett Packard Corporation
HP-UX	Hewlett Packard Unix
IA-64	Intel Architecture 64
IBM	International Business Machines Corporation
JIT	Just-In-Time
L1	Level 1
L2	Level 2
PC	Program Counter
RISC	Reduced Instruction Set Computer
ST	Sampling Tool
VM	Virtual Machine
XML	Extensible Markup Language

Back to Table of Contents

This report is available on-line at http://www.cse.wustl.edu/~jain/cse567-06/sw_monitors1.htm
List of other reports in this series
Back to Raj Jain's home page