Survey of Software Monitoring and Profiling Tools

Ben Wun [bw6@cse.wustl.edu]

Abstract

Performance monitoring and profiling tools are essential for programmers to optimize their programs and remove bottlenecks. A plethora of tools exist for a variety of systems. These tools range in scope from single program profilers to OS and hypervisor level monitors. An understanding of the availability and techniques of the different monitoring tools is essential for choosing the proper program for your task. This paper discusses various techniques used in different systems to gather this data, and provides examples of each. The benefits of these tools are compared to provide the reader with guidance for choosing the proper technique.

See Also:

Table of Contents:

1. Introduction

Amdahl's law indicates that in order to write fast and efficient programs, programmers must target the most expensive part of their code. In order to pinpoint the sections of code which should be optimized, a programmer needs detailed data on how that program behaves. Similarly, system administrators need detailed information on whole system performance in order to guide the configuration of their servers. In order to serve these needs, a number of performance monitoring tools have been developed over the years. A multitude of tools for different systems exist, but most can be classified according to a few categories. This paper will not attempt to exhaustively list all the available tools for all possible systems, but instead provides an overview of the various types of tools and techniques that exist, examines their pros and cons, and provides a detailed examination of a few examples. It is hoped that this information will help guide the reader to effectively evaluate any monitoring tool they may encounter in the future.

The characteristics of a good program monitor can differ depending on the exact scenario being measured. It may be adequate to profile a single program under study, or it may be more desirable to measure whole system performance. However, there are several goals that are shared by all monitoring systems. First and foremost, the program behavior should be modified as little as possible. If a program's behavior is radically modified by monitoring it, then the collected data may be worthless. Furthermore, if the instrumentation is an integral part of the system's design, it should not impose any overhead when inactive. For example, if hardware counters are included in a processor, they should not lie on the critical path of the circuit's logic. Flexibility is another attribute of a good system monitor. It is possible for a monitoring program to collect too little, too much, or the wrong kind of data. For example, if a program is hampered by cache misses, a call graph profiler may be of little use. On the other hand, it is possible to collect so much data that it becomes difficult if not impossible to analyze it. Good system monitoring tools allow the user to specify the amount and type of data to be collected.

This paper organizes the tools to be discussed as follows- section 2 discusses static program modification, section 3 discusses hardware support for performance profiling, section 4 discusses kernel profiling, section 5 discusses monitoring of virtual machines, and section 6 discusses monitoring of parallel systems. We summarize this information in section 7.

Back to Table of Contents

2. Static Program Modification

One approach to monitoring the runtime behavior of programs is to instrument the binaries to record desired events. Some of the simplest examples of this technique include basic block counters, such as QPT (Quick Program Profiling and Tracing System)[Larus94]. QPT is a program for MIPS and SPARC systems that inserts code to record, at runtime, how often each basic block is run. Other programs exist which can produce memory access traces or profile dynamic branching behavior.

There are several points at which a program may be instrumented. The first is at compile time. If the source code of the program and the compiler are available, the compiler can be modified to instrument the program. This has the advantage of the availability of symbolic and syntactical information that is not available to binary rewriters. However, access to the source code for both the compiler and program is not always possible. It is also not possible to instrument any libraries that may be linked to later. Using the linker to insert checks can resolve this problem.

If the only thing that is accessible is the program binary, the above methods will not work. The other option is to use an executable rewriter that instruments the binary executable directly. This has the advantage of working for any program, not just those for which the source code is available. However, binary rewriters may not have access to all the symbolic debugging information.

In this section, we will examine three examples of program profilers that use static program modification. Each instruments the program at a different stage, and table 1 summarizes their characteristics.

Table 1: Comparison of example static program monitors

Tool	Mod Time	Programmability	OS Dependent	Language Dependent	Need Source
Gprof	None	No	No	Yes	Yes
ATOM	Link	Yes	Yes	No	No
Etch	Executable	Yes	Yes	No	No

2.1 Gprof

One example of a compile time instrumenter is Gprof[Graham82]. Gprof is a call graph execution profiler that creates a call graph detailing which functions in a program call which other functions, and records the amount of time spent in each. This type of profiling is useful for figuring out which functions a programmer should spend his or her time optimizing, either because they are called very often or because they take a significant time to run.

Gprof relies on having the symbolic information available in source code in order to properly construct the call graph. Any libraries that are linked in must be compiled with support for gprof, or profiling data cannot be gathered from them. Furthermore, only certain languages are supported. When a program is compiled with gprof support, monitoring routines are inserted at strategic points in the code to produce a trace of events. As little work as possible is performed at runtime in order to avoid adversely affecting program behavior. More extensive analysis of this trace is performed off line in order to recreate the call graph. Since Gprof is implemented at the language level, it is portable across different architectures and OSes.

Gprof is very easy to use and portable, but it is limited in scope. It is only designed to produce a call graph and to tabulate the time spent in each function. Furthermore, access to the source code is required to use it. A more flexible framework that allows users to define how they want to instrument their programs is discussed next.

2.2 ATOM

ATOM [Srivastava94], which stands for Analysis Tools with OM, is a framework for building program monitoring routines, and is written for the Alpha AXP under OSF/1. Instead of limiting programmers to a predefined set of services, such as basic block counting or producing memory traces, ATOM provides a framework for the user to specify instrumentation points and actions. ATOM acts on object files, making it language independent.

When using ATOM, a programmer must write instrumentation routines that tell ATOM where to insert code, and what type of code should be inserted. Users write instrumentation routines to tell ATOM where to insert instrumentation code, and analysis procedures that define what should happen at those instrumentation points. An example instrumentation routine that records the effective address at each load follows:

```
Instrument(int iargc, char**iargv)
{
Proc* p;
Block *b;
Inst *inst;

AddCallProto("OpenRecord()");
AddCallProto("CloseRecord()");
AddCallProto("Load( VALUE)");

for(p=GetFirstProc();p!=NULL;p=GetNextProc(p)){
    for(b=GetFirstBlock(p);b!=NULL;b=GetNextBlock(b)){
       for (inst = GetFirstInst(block); inst != NULL; inst = GetNextInst(inst)){
           if(IsInstType(inst, InstTypeLoad){
              AddCallInst(inst, InstBefore, "Load", EffAddrValue);
          }
       }
    }
}
AddCallProgram(ProgramBefore, "OpenRecord");
AddCallProgram(ProgramAfter, "CloseRecord");
}
```

ATOM breaks the program up into procedures, and the procedures into basic blocks. The basic blocks are further broken down into individual instructions. The GetNextProc and GetNextBlock functions return a handle to the next procedure or basic block respectively. The GetLastInst and IsInstType routines operate on individual instructions within the current basic block. The AddCallProto primitive defines function prototypes for analysis procedures defined elsewhere. In this example Load, OpenRecord and CloseRecord are the analysis routines. The Load routine is passed the Effective address used for the load, which will be of use to the analysis routine. AddCallInst adds a call to the indicated analysis routine at the specified point. Next, we define the analysis routines:

```
File* f;
void OpenRecord()
{
    f = fopen("LoadAddrs.txt");
}

void Load(int addr)
{
    fprintf(f, "%X\n", addr);
}

void CloseRecord()
{
   fclose(f);
}
```

The open and close record routines are self explanatory. The Load routine takes as an argument the address that is being loaded and records it in the file.

ATOM imposes what its creators consider to be acceptable overhead. Using the SPEC92 benchmark suite, they found that ATOM increased execution times anywhere from 1.01x to 11.84x of the uninstrumented code.

While ATOM's use of object files as inputs frees it from dependence on any particular language or compiler, access to the intermediate object files is still needed. If a user wants to observe the behavior of a program for which only the binary executable is available, another tool, such as the one discussed in the next section, is needed.

2.3 Etch

Etch [Romer97] is another performance monitoring framework that allows users to specify how a program should be profiled. Etch has a few key differences from ATOM which make it worth discussing. First, Etch is built for x86 programs running in a Windows environment. A more interesting distinction from ATOM is that Etch works on binary executables. This means that end users with no access to source code or intermediate object files can use Etch to determine what is happening in their system. This information can then be used to directly rewrite the binary for better performance.

2.4 Summary

The three tools discussed in this section all use static instrumentation of programs to provide insight into their dynamic behavior, but each offers something different. Gprof is the most limited of the three; it is only capable of producing a call graph, and is language dependent. However, it is easier to use than the others, and is portable across CPU architectures and OSes. ATOM and Etch provide more flexible functionality- users define their own instrumentation code instead of being limited to a single function. They are also both language independent. However, unlike Gprof, they work at the binary level and are thus confined to a particular OS and architecture. If only the final binary executable is available, then Etch is the only tool of the three that will be of use.

Static instrumentation of individual binaries can be a powerful tool for observing the runtime behavior of a program. The main drawback to this approach is that it is limited in scope; the other layers of the system, such as the operating system and hardware, can have important effects on the execution of a program. Events such as scheduling, cache misses, interrupt handling and more can be the underlying cause of poor system performance, but these are not captured by the methods discussed in this section, and can only be obtained with the help of code in the OS itself, or with hardware monitoring facilities.

Back to Table of Contents

3. Hardware Counters

Modern processors often include hardware support for performance monitoring. These usually take the form of configurable counters that can be set to record certain events and to raise an interrupt when the counters overflow. This hardware support allows programmers insight into dynamic program behavior that might otherwise be unavailable. In addition to clock ticks, events such as cache misses, cache coherency events, branch mispredictions, stalled cycles and the like can be recorded. The specific events that can be monitored vary from processor to processor. While such data can be gathered by running the program in a simulator, this is often not practical. Simulators are not always available, and even when they are, executing programs on them can be orders of magnitude slower than running them on hardware. For a reasonably complex program, hardware counters are the only practical method for obtaining this type of data.

3.1 Oprofile

Access to hardware counters is usually limited to privileged instructions, and as such, programs usually access them through the operating system and supporting libraries. The inclusion of support for these counters in the OS allows the entire system, including the kernel, and not just individual user programs to be monitored. Oprofile[Oprofile] is the framework in the Linux kernel that supports hardware counters on all the processors on which that OS can run. If the Oprofile analysis tools are provided with the kernel's symbol table (usually by feeding it the System.map file for the running kernel, produced at compile time) detailed analysis of the kernel is possible.

Oprofile is the Linux kernel's framework for accessing hardware counters. Others, such as Vtune[Vtune] exist, but will not be discussed because they operate on similar principles. For more details about the counters themselves, we will now take a look at their implementation in one particular processor, the MIPS R10000.

3.2 MIPS R10000

Hardware designers must be careful when incorporating hardware counters in their chips. The additional logic must not be prohibitively expensive, and must not lie on the critical path, or would degrade performance, even when they are not in use. The performance counters for the MIPS R10000[Zagha96] use 300 lines of RTL and fewer than 5000 transistors. This was less than 0.2% of the number of transistors used in the entire processor, not counting cache. The design team felt that "the counters were well worth the silicon and design time to include them."

Now that we have discussed how hardware counters are integrated into the processor and OS, we will look at an example of their use. The authors of the MIPSR10000 paper [Zagha96] used their counters to examine the the effects of multiprocessor sharing in a system that uses a cache-snooping protocol. The authors examined a weather modeling program that did not scale well to multiple processors. The underlying cause was puzzling, because threads in this program share very little data and should run fairly independently. Using the hardware counters, the authors could determine the number of cycles the program took to execute, as well as the number of cache invalidation events and the cycles needed to service them. This data indicated that execution time was dominated by secondary cache misses, caused by a large number of invalidations. This would not have been obvious by examining the source code, because very little sharing occurs between threads in this program. It was found that false sharing, caused by allocating non shared variables to the same cache line, was the cause of the problem.

The model of programmable counters that interrupt the processor when they overflow, is the most common one, but it imposes an overhead on program execution that may have adverse effects on the system being studied. A newer generation of processors uses more advanced facilities to try to overcome this problem, as we discuss in the next section.

3.3 Cell Processor

Newer generations of chips have even more advanced hardware monitoring facilities. The Cell processor is a complex system on a chip that contains multiple, heterogeneous computing elements. In order to aid debugging, the Cell's designers have included 2200 signals distributed throughout the chip that can be monitored simultaneously [Genden06]. The monitoring hardware does not impact execution, as the collected data is sent across a separate, dedicated bus that operates at the same speed as the rest of the chip. In addition, a complex Trace Logic Analyzer (TLA) is connected to the debugging bus that performs much of the necessary monitoring functionality. This level of visibility is extremely important in a chip as complex as the Cell, and the non intrusive way it is incorporated is impressive.

3.4 Summary

Hardware monitoring facilities for processors can give programmers insight into events such as cache misses or bus activity that is unavailable from purely software methods. The use of these counters is mediated through the operating system, and most follow the model of programmable counters that invoke interrupt routines when then roll over to allow the cooperating software to perform the necessary book keeping. Newer processors like the Cell provide more advanced monitoring facilities that require less overhead and thus impact the system under study less. The choice of monitoring hardware is determined by the processor being used. Back to Table of Contents

4. Kernel Profiling

The runtime behavior of an operating system's kernel can be of interest both for understanding and improving the kernel itself, and for understanding its effects on user programs. Understanding the kernel's effect on hardware events such as cache misses or clock ticks consumed may be inadequate; a more comprehensive framework that captures the relevant state of the kernel's data structures is important, and cannot be achieved using only hardware counters. Many tools are available for kernel monitoring. Of these, the most flexible and comprehensive is the DTrace framework built into the Solaris kernel.

4.1 DTrace

DTrace [Cantrill04] is the dynamic instrumentation facility built into the Solaris kernel. DTrace provides several notable advantages. Using dynamic instrumentation, the kernel's binary does not need to be modified, and instrumentation can occur at runtime. An important aspect of DTrace is that there is zero disabled probe effect, meaning that when the system is not being monitored, DTrace imposes no additional overhead. DTrace allows for arbitrary context kernel instrumentation, including the examination of the scheduler and other delicate subsystems. Another advantage is that a unified framework is provided for profiling the kernel and user space programs. DTrace maintains the integrity of collected data- there are no windows in which data can be lost. Finally, DTrace allows the user to define probe actions rather than providing a set of inflexible, predefined probes.

Dtrace defines several thousand points in the kernel that can be instrumented. A user writes providers, which are loadable kernel modules that use the DTrace API to communicate with the kernel. The providers create probes at the instrumentation points of interest. The probes are then provided to user space consumers, which are responsible for enabling the probes and aggregating the data. When the probe is fired, the DTrace framework preempts the system and takes over. In order to improve performance and cut down on the amount of data that the system must deal with, probes can be associated with a predicate that lays out the conditions under which the probe will fire. If the conditions are not met, the probe will not be invoked. Another mechanism to address this is the use of speculative tracing. It may not be apparent that data can be discarded until after the probe has fired, and speculative tracing allows this data to be buffered and discarded at a later time if deemed appropriate.

In order to both improve performance and reduce the amount of data that needs to be processed, DTrace allows the user to specify predicates, which determine if a particular enabled probe is to be run. Probes will only run if the conditions of the predicate are met. If the predicate is met, the probe fires and executes a user defined action. Users specify this action in the D language, which is compiled into the D Intermediate Format (DIF), which is then interpreted by a virtual machine in the kernel. Several provisions, like the elimination of non-forward branches, and runtime safety checks, ensure that these user probes are safe to run in kernel mode. This is a very powerful idea, since it allows users to define how their probes should behave without having to worry about bringing down a running system.

Instrumentation of user programs is achieved through what its creators call a pid provider. As with kernel probes, there is zero disabled probe effect for user space programs, and the user program need not be restarted to be instrumented. Instead of rewriting the program to branch to instrumentation code, DTrace rewrites the target instruction to induce a kernel trap. This allows for a unified kernel and user space monitoring mechanism at the cost of runtime overhead.

DTrace aims to be a comprehensive performance evaluation and monitoring framework, but one major feature that it lacks is support for hardware counters. Furthermore, it is not portable across different OSes.

4.2 Other Kernel Profilers

DTrace is specific to the Solaris operating system. Other performance tools exist for other OSes. The Linux Trace Toolkit (LTT) [Yaghmour00] for Linux uses static instrumentation that incurs some overhead even when disabled and does not allow user specified actions. DProbes [Moore01] for OS/2 and Linux is a dynamic instrumentation system that achieves zero disabled probe overhead like DTrace and defines a language for arbitrary actions. It is designed to be portable across OSes and processor architectures. Disadvantages of DProbes include its lossy nature and the lack of support for pruning data such as predicates and speculative execution. Additionally, DProbes is not completely safe and can cause a system crash.

4.3 Summary

The choice of kernel monitoring tools is determined by the OS being studied. However, where a choice is available, a comprehensive dynamic instrumentation scheme that integrates support for hardware counters and allows for user defined probe actions is to be preferred. Thus, on a system like Linux, where some choice is available, most users should prefer DProbes to LTT, but must watch for pitfalls, such as the possibility of DProbes destabilizing the system.

The OS is not always the lowest layer in a system's software stack. Virtual machine technologies allow multiple OSes to share a single underlying hardware system. This has implications for performance monitoring that we will discuss in the next section.

Back to Table of Contents

---

5. Virtual Machines

Virtual machines add another layer of abstraction between the operating system and the underlying hardware. This can have unanticipated performance implications that need to be examined. Tools that monitor the system at the program or operating system level are not adequate to the task in such a system. They are not aware of the existence of the hypervisor or other execution domains. Furthermore, hardware counters are usually only accessible by privileged instructions; in a virtualized system, only the hypervisor, and not the OS can access these instructions. For these reasons, it is important to develop monitoring tools that are aware of the virtualized system.

5.1 XenoProf

The XenoProf [Menon05] performance monitor is a performance profiling utility for the Xen virtual machine environment. It instruments the Xen hypervisor and its guest operating systems to use the underlying hardware performance counters in a manner similar to how Oprofile works in Linux systems. Since XenoProf requires modification of the guest OS source code, it can only be used with open source guest OSes. Currently, support is only available for Linux. XenoProf profiles the execution of the hypervisor itself, but relies on the modified OS to profile its own kernel and programs. Using an OS independent interface, this allows XenoProf to operate in a system that uses different OSes in different execution domains. XenoProf can be used to profile a single domain in a multiple domain system, or to profile the entire system.

As an example of the utility of whole system monitoring in a virtual machine environment, the XenoProf creators used their creation to profile the network subsystem of a Xen system. They found that the virtualized system performed far worse than the same hardware without virtualization. Using XenoProf, they were able to trace the cause to the high cost of page remappings in the virtualized network driver. Armed with this data, they were able to greatly improve the performance of the Xen virtualized network driver.

5.2 Summary

The addition of a hypervisor to the software stack can have important performance implications for a computer system. The performance monitors built for traditional systems are inadequate to fully understanding the behavior of such a system, and new tools, such as XenoProf, have been devised to address this challenge. As with kernel monitors, the choice of virtual machine profiling tools is dictated by what is available for the particular system being studied. The most important thing to remember when profiling such a system is that the hypervisor's effect on performance must be accounted for.

All of the techniques discussed up to this point were built to monitor a single computer system. When a parallel computing cluster consisting of multiple nodes is being studied, new tools must be developed to account for the special challenges in such an environment. As we will see in the next section however, the mechanisms used in single node systems are still useful.

Back to Table of Contents

6. Parallel Systems

The monitoring and profiling of distributed parallel systems presents extra challenges not present in single node systems. Synchronizing and ordering events among multiple distributed nodes running in parallel requires special consideration, since there is no global clock. The sheer amount of data that can be produced at each node and by the inter-node communication traffic between them requires that the observer be able to specify monitoring events at the right level so that unneeded information can be pruned out.

6.1 ZM4/SIMPLE

One example of a parallel and distributed system monitoring tool is the ZM4/SIMPLE (Zahlmonitor 4/Source-related and Integrated Multiprocessor and computer Performance evaluation, modeLing and visualization Environment) system. This is an example of a hybrid hardware/software monitoring system using event driven monitoring. It is built to be scalable and is adaptable to arbitrary object systems. The nodes of the parallel systems it monitors need not be homogeneous.

ZM4[Hofmann94] is the hardware component of the ZM4/SIMPLE system. ZM4 consists of a control and evaluation computer (CEC) that controls the actions of multiple, distributed monitor agents (MA). MAs are PCs equipped with 4 dedicated probe units (DPUs). DPUs are custom printed circuit boards which link the MAs to the systems they monitor. DPUs perform tasks such as time stamping, event recording and buffering of said events. The local clock of the MA has a resolution of 100ns (which was considered adequate when the paper was published) which is synchronized to a global measure tick generator(MTG). In order for this to work, the wire length between each node and the MTG must be known. Each MA can be up to 1000m away from the CEC. Ethernet is used for communication between the MAs and CEC. In addition to the ZM4 monitor, the software is also instrumented at key points. This is necessary in order to correlate the hardware events to the software's symbols and structure.

Inferring the order of events on distributed nodes requires special consideration in a massively parallel system. A global ordering must be derived from local events. In a message passing system, there is a global ordering inherent in the send and receive operations. In a shared memory system, the use of synchronized global clocks, as provided by ZM4, is necessary to decide which accesses to shared addresses occur in what order. The timer must have resolution less than memory access times to be successful.

6.2 Summary

Other tools exist for monitoring parallel systems, and most are limited to a particular parallel machine. Intel, for example, provides a performance monitoring environment for their Paragon system [Ries93]. These systems must all grapple with the same issues the designers of ZM4/SIMPLE did, namely reconstruction of global state from local monitors, and buffering and pruning of massive amounts of information. The solution is usually similar, in the hybrid use of hardware and software monitors.

Back to Table of Contents

7. Summary

There exists a wide variety of tools available to programmers and system administrators to monitor and evaluate computer software. These range in scope from profiling a single program to monitoring of an entire system, including the operating system and hypervisor. Many techniques exist at each of these levels, with different tradeoffs and availability for different systems. The choice of the proper monitoring system is essential in performance analysis. Profiling a single program may be inadequate to pinpoint the source of problems, whereas whole system monitoring may produce too much data to be easily parsed. Different techniques introduce different sources of error. Hardware support for system monitoring can provide valuable information, but only if the tool used is aware of them. A programmer must be aware of the underlying techniques used by his monitoring tool of choice in order to choose the right tool and properly interpret the data he/she obtains from it.

Back to Table of Contents

8. References

1. [Burkhart89] H. Burkhart, R. Millen. "Performance-measurement tools in a multiprocessor environment." IEEE Transactions on Computers, Vol. 38, No. 5, May 1989. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=24274

2. [Cantrill04] Bryan M. Cantrill, Michael Shapiro, Adam Leventhal. "Dynamic Instrumentation of Production Systems." In Proceedings of the General Track: 2004 USENIX Annual Technical Conference, June 2004. http://www.usenix.org/event/usenix04/tech/general/full_papers/cantrill/cantrill.pdf

3. [Genden06] Mike Genden, Ram Raghavan, Mack Riley, John Spannaus, Thomas Chen, "Real-time Performance Monitoring and Debug Features of the First Generation Cell Processor," In Proc. of 1st Workshop on Tools and Compilers for Hardware Acceleration, September 2006

4. [Graham82] Susan L. Graham, Peter Kessler, Marshall Mckusick. "Gprof: A call graph execution profiler." ACM Sigplan Noticies, June 1982. http://portal.acm.org/citation.cfm?id=872726.806987&coll=portal&dl=ACM&CFID=11111111&CFTOKEN=2222222

5. [Hofmann94] R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle. "Distributed Performance Monitoring: Methods, Tools and Applications." IEEE Transactions on Parallel and Distributed Systems, Vol 5, No. 6, June 1994. http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/trans/td/&toc=comp/trans/td/1994/06/l6toc.xml&DOI=10.1

6. [Larus94] James R. Larus and Thomas Ball. "Rewriting executable files to measure program behavior." Software, Practice and Experience, vol 24, no. 2, pp. 197-218, Feb 1994. http://research.microsoft.com/~tball/papers/rewrite.pdf

7. [Menon05] Aravind Menon et. al. "Diagnosing Performance Overheads in the Xen Virtual Machine Environment." Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, 2005. http://portal.acm.org/citation.cfm?id=1064979.1064984

8. [Moore01] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In Proceedings of the FREENIX Track, June 2001. https://www.usenix.org/events/usenix01/freenix01/moore.html

9. [Oprofile] Oprofile – A System Profiler for Linux: http://Oprofile.sourceforge.net/news/

10. [Ries93] B. Ries, et. al. "The paragon performance monitoring environment." Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, pp. 850-859, 1993. http://portal.acm.org/citation.cfm?id=169627.169851

11. [Romer97] Ted Romer et. al. "Instrumentation and Optimization of Win32/Intel Executables Using Etch." Proceedings of the USENIX Windows NT Workshop, August 1997. http://www.usenix.org/publications/library/proceedings/usenix-nt97/full_papers/romer/romer.pdf

12. [Srivastava94] Amitabh Srivastava, Alan Eustace. "ATOM: a system for building customized program analysis tools." Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, 1994. http://portal.acm.org/citation.cfm?id=178260&coll=portal&dl=ACM&CFID=11111111&CFTOKEN=2222222&ret=1#Fulltext

13. [Vtune] Intel VTune Performance Analyzers. www.intel.com/vtune

14. [Yaghmour00] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In Proceedings of the 2000 USENIX Annual Technical Conference, 2000. http://www.usenix.org/publications/library/proceedings/usenix2000/general/full_papers/yaghmour/yaghmour_html/

15. [Zagha96] M. Zagha, B. Larson, S. Turner, M. Itzkowitz. "Performance Analysis Using the MIPS R10000 Performance Counters." 1996 ACM/IEEE Conference on Supercomputing, 1996. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1392891

Back to Table of Contents

List of Acronyms

ATOM	Analysis Tools with OM

CEC	Control and Evaluation Computer
DIF	D Intermediate Format
DPU	Dedicated Probe Unit
LTT	Linux Trace Toolkit
MA	Monitor Agent
MTG	Measure Tick Generator
OS	Operating System
QPT	Quick Program Profiling and Tracing System
TLA	Trace Logic Analyzer
ZM4/SIMPLE	Zahlmonitor 4/Source-related and Integrated Multiprocessor and computer Performance evaluation, modeLing and visualization Enviroment

Back to Table of Contents

This report is available on-line at http://www.cse.wustl.edu/~jain/cse567-06/sw_monitors2.htm
List of other reports in this series
Back to Raj Jain's home page