Performance Measurement Research: A High-Level Overview

Draft – Version 0.9*

Tim Brecht
brecht@cs.uwaterloo.ca

May, 2001

Abstract

The importance of good tools to evaluate and understand the performance of various aspects of a computing environment is far too often underestimated. The thought of going into a hospital because you are feeling ill and having a surgeon move you into an operating room to replace your, heart, liver, or kidney without first conducting extensive and conclusive tests seems absurd.

And yet an analogous situation takes place on a regular basis in the computer and communications industry. When applications, systems or services aren't performing up to expectations, it is far too often the case that sufficient diagnostics are not performed or that they lack the sophistication required to accurately pinpoint the source of the problem. Often applications and subsystems are modified or replaced, or extra memory, disks, bandwidth and even machines are added to the environment – all in the hope these changes will solve the problem (and often they do). Unfortunately, the approach used to make the decision regarding what action to take too frequently lacks solid evidence that points directly to the bottleneck. Additionally it is seldom clear from the information provided what action is appropriate. Being able to quickly, easily and accurately locate and alleviate performance problems in complicated systems is becoming increasingly critical and increasingly difficult.

In this document we provide a high-level overview of some of the research being conducted related to this problem.

*A final version reflecting feedback from the talk and discussions will be submitted.

Contents

1 Introduction **4**

2 Possible Categorizations **4**
2.1 An Alternate Categorization 6

3 Direct Observation Using Instrumentation **6**
3.1 Source Code Insertion 7
3.1.1 Pros: 7
3.1.2 Cons: 8
3.1.3 Scope of Applicability 8
3.1.4 Possible Avenues for Investigation 8
3.2 Compiler-based Code Insertion 8
3.2.1 Pros: 9
3.2.2 Cons: 9
3.2.3 Scope of Applicability 10
3.2.4 Possible Avenues for Investigation 10
3.3 Linker-based Code Insertion 11
3.3.1 Pros: 12
3.3.2 Cons: 12
3.3.3 Scope of Applicability 13
3.3.4 Possible Avenues for Investigation 13
3.4 Rewriting Binaries to Insert Code 13
3.4.1 Pros: 16
3.4.2 Cons: 16
3.4.3 Scope of Applicability 16
3.4.4 Possible Avenues for Investigation 16

4 Direct Observation Without Instrumentation **17**
4.1 Instruction Set Simulation 17
4.1.1 Pros: 18
4.1.2 Cons: 18
4.1.3 Scope of Applicability 18
4.1.4 Possible Avenues for Investigation 18
4.2 Hardware Profiling 19
4.2.1 Hardware Performance Counters and Instrumentation 19
4.2.2 Using Hardware Performance Counters Without Instrumentation 20
4.2.3 Pros: 20
4.2.4 Cons: 20
4.2.5 Scope of Applicability 20
4.2.6 Possible Avenues for Investigation 21
4.3 Execution Tracing 21
4.3.1 Pros: 21
4.3.2 Cons: 21

4.3.3 Scope of Applicability 21
4.3.4 Possible Avenues for Investigation 21

5 Indirect Observation **22**
5.1 Benchmarks and Workload Generators 22
5.1.1 Parallel Application Benchmarks 22
5.1.2 World-Wide-Web Workload Generators 22
5.1.3 Pros 23
5.1.4 Cons 23
5.1.5 Possible Avenues for Investigation 23
5.2 Observation Tools 24
5.2.1 Pros 24
5.2.2 Cons 24
5.2.3 Possible Avenues for Investigation 25

6 Summary **25**
6.1 Some Potential Opportunities 25
6.1.1 Measuring and Improving Performance in Complex Environments 25
6.1.2 New and Improved Tools that Leverage Existing Technology 26
6.1.3 Binary Rewriting to Reduce Lock Contention 26
6.1.4 A Note on Tuning Notes – Autotuning 26

7 Future Work **27**

1 Introduction

While implementing, testing, and debugging applications, systems and services[1] will continue to be a major concern in most software development projects, a growing concern is their performance. This is especially true in environments where significant monetary gains can be obtained by offering a competitive advantage or through cost savings due to a reduction in the required resources. As a result, significant interest and research in performance has been focussed on areas with large monetary incentives (e.g., computer architecture design, databases and e-commerce servers) or areas with high performance demands (e.g., parallel computing and embedded systems).

In order to demonstrably improve performance one must be able to: identify the environment and workload in which performance is an issue, determine appropriate performance metrics, reliably measure performance using a representative environment and workload, modify the system under study, re-measure performance, and to draw statistically significant conclusions regarding the differences in performance.

In this document, we provide a high-level overview of some of the research being conducted related to measuring the performance of applications, systems and services. We loosely divide the current work into categories and then provide a slightly more detailed example of one or two pieces of work in each category. The goal at this time is not to provide a detailed survey of work in this area, but instead to provide a high-level road map in order to further explore areas of mutual interest and to identify possible avenues for more detailed exploration.

Since the ultimate goal of this project is to investigate and develop techniques and tools to more quickly and easily pin-point bottlenecks and to help to determine methods for alleviating those bottlenecks, we focus on work related to measurement rather than modelling or simulation.

We assume that the reader is reasonably familiar with issues related to performance evaluation (for background information see [33] [24]). As a result we forego the usual discussion related to the pros and cons of analytic models, simulation and empirical measurements. We also try to avoid replicating some of the more common information that is available elsewhere. For example, we do not discuss information regarding standard UNIX tools, techniques and tools that are described in books like [17] [14] [27], or web pages related to the subject [57] [42] [22].[2]

2 Possible Categorizations

One difficulty in summarizing a significant body of work is trying to make sense of and provide insights into the relationships and similarities between elements of that work. An approach that is often helpful in this regard is to attempt to categorize the work and to place each individual contribution into an appropriate category.

While we attempt one such categorization here, our goal is not to devise an all encompassing categorization but to provide a simple framework within which to begin describing some of the research related to performance measurement[3].

[1] We will frequently just use the term systems to refer to applications, systems and services.

[2] I'm in the process of building a web page that will provide links to various companies, research projects and other sources of information related to the topic. With permission, I would like to make this information publicly available – of course I would acknowledge SUN's support for its creation.

[3] Feedback regarding this and other possible categorizations would be appreciated.

Our current categorization is derived from the tools and techniques used to measure and analyze performance. At the first level we divide techniques and tools according to whether or not they *directly* or *indirectly* observe the performance of the target application, system or service (sometimes referred to simply as the *system*) while operating under normal conditions.

An example of direct observation might be to run and measure the execution time of an application in order to understand how it executes on a particular machine. On the other hand, indirect observation might involve running a set of benchmark applications (but not the actual application of interest) on the particular machine and then attempting to draw conclusions about the behaviour of the application of interest on that machine based on the observations obtained from the benchmark. Another example of indirect observation might involve the use of a program like `netperf` [25] to observe the bandwidth of network transfers between two hosts in an attempt to determine if the network may be a bottleneck.

While direct observation is typically preferable, it may not always be possible or appropriate.

We further sub-divide direct observation into categories that require or do not require instrumenting the system being observed. Here instrumentation of the system involves modifying the execution of the system by modifying its underlying source code, libraries, run-time system, or executables.

Before delving into each of these categories in detail we briefly summarize them below and include some examples of the types of tools and techniques that would fall under each category.

Direct Observation

- Instrumenting Techniques

  Instrumenting a system requires that a person or program either understands the system or is somehow able to discern its behaviour. For this reason, this approach can be viewed as a white-box approach because some understanding of what the system is doing is used in order to instrument its behaviour. Examples of techniques that fall under this category are:

  - Manual insertion of timing and or statistics gathering code.
  - Compiler-based automatic profiling (e.g., gprof).
  - Using shims, wrappers or shared library interposition.
  - Rewriting the executable.

- Non-instrumenting Techniques

  If instrumentation is not being used, the system can be treated as a black-box. In this case the original system is not modified and external aids are used to measure behaviour. Examples of some of the tools and techniques that would fall under this category are:

  - Tracing execution using a simulator.
  - Using hardware profiling (e.g., performance counters and registers).
  - Tracing execution (e.g., using `strace`, `truss` or something similar).

Indirect Observation

- Benchmarks to examine the performance of the system.
  - Splash-2 parallel application benchmarks.
  - `httperf` workload generator for measuring web server performance
- System measurement and observation tools (e.g., `netperf`, `ping`, `vmstat`, `iostat` and `netstat`).

2.1 An Alternate Categorization

Another possible approach to categorization would be to consider the scope with which the tool or technique can be applied. For example, the division might reflect whether the tool or technique is designed to measure the performance of:

- The underlying hardware.
- The Operating System.
- Libraries.
- An application. This may be further divided into categories like:
  - Single-threaded user applications.
  - Multi-threaded user applications.
  - Parallel and/or distributed applications.
  - Specialized server applications (e.g., DNS, databases, httpd, etc.)
- Aspects of a network.
- A combination of some of the above.
- A service or combination of services (e.g., multi-tiered Internet service).

Throughout this document we'll attempt to place each tool and technique into one of the main categories outlined above (direct observation with instrumentation, direct observation without instrumentation, or indirect observation) and also try to outline the scope of applicability.

3 Direct Observation Using Instrumentation

The most popular technique for observing the performance of an application, system or service is to instrument that system, run that system, directly observe and measure its behaviour and to analyze its performance. Arguably the most active area of recent research appears to be in the area of instrumention of systems for direct observation. In addition to natural improvements made to

existing techniques, recent developments in rewriting program executables and binary translations have lead to some interesting breakthroughs in this area.

Figure 1 shows possible points at which instrumentation code can be inserted into an application (denoted by the gray areas). We next briefly describe and discuss the techniques used to insert instrumentation code at each of these points.

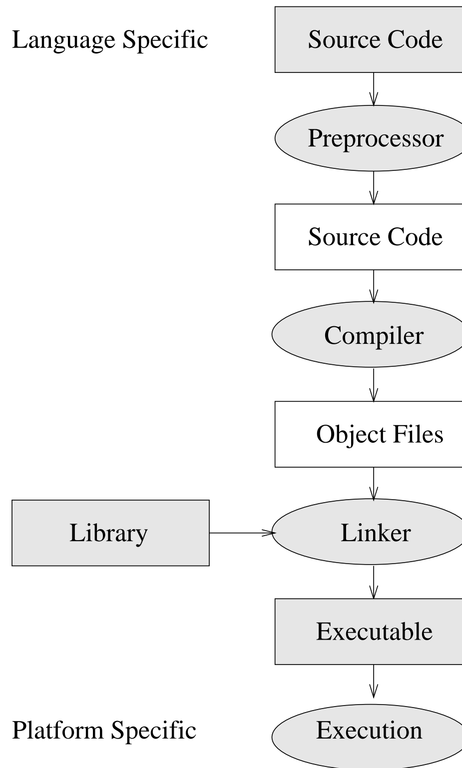

Figure 1: Points of possible instrumentation (diagram from [53])

3.1 Source Code Insertion

This approach involves modifying the source code of a system to insert code to track information of interest. Most of the work in this area [29] [52] [13] involves developing macros and libraries that the programmer can call. These macros and libraries implement common portions of code that would often be used when instrumenting a program to measure its performance. Examples of such support include starting and ending timings, print timing information, and tracking and printing statistics or visualizing collected data.

3.1.1 Pros:

- As portable as the instrumented source code.

- Can obtain detailed information on targeted information of interest.
- Can be useful where compiler-based profiling is weak (e.g., threaded/parallel applications and for including things like overheads due to C++ templates and virtual functions).
- Quite a good scope of applicability.

3.1.2 Cons:

- Requires source code.
- Time consuming.
- Error prone.
- Requires fairly detailed understanding of the system.
- May be less effective in distributed and/or client-server applications.

3.1.3 Scope of Applicability

Because what is being measured is directly controlled by the programmer, this approach has perhaps the widest scope of applicability. If used with hardware performance counters it can be used to monitor hardware performance. It can also be used to measure the performance of an application, operating system and network, and possibly combinations of the above. With the support of an external method for synchronizing clocks (e.g., NTP or GPS receivers), it can even be used to measure several aspects of client/server or distributed systems.

3.1.4 Possible Avenues for Investigation

- It might be possible to develop techniques to automate the process of code insertion for certain types of measurements. For example, it would be trivial to instrument specified functions by inserting timing code at function entry and exit points. More interesting however, might be to develop a system or language that permits one to specify code fragments or patterns (e.g., using regular expression) that are to instrumented.
- It also seems that better support could be provided to assist programmers when they are adding measurement code.

3.2 Compiler-based Code Insertion

The notion of inserting extra instructions into the instruction stream in order to help to measure and understand the performance of applications has been around for quite some time. A tool like *gprof* uses a combination of compiler inserted instructions and interrupts to collect data regarding the execution of an application. The main idea is to periodically interrupt the executing application and, while in the interrupt handler, read the program counter and increment an appropriate data

structure. If the interrupts are relatively frequent and the application executes for sufficiently long, the result is a fairly accurate picture of where the application is spending most of its time executing.

The compiler is used to insert code to store and later gather information about procedure calls. This code is used to produce a call graph and to count the number of times each function is called. Similarly, basic-block counting is implemented by inserting instructions to increment an element of a zero-initialized static array, prior to entering each new block (i.e., for each `if` statement). Each element of the array is used to count the number of times each basic block is executed. After execution the data is written and later correlated with the program's source code.

The power of this approach lies in the ability to directly pinpoint where the application is spending most of its time executing by correlating the collected execution data with the source code. Therefore, if one wants to improve the application's performance one knows where focus attention – on those parts of the program that account for the greatest portion of the execution time.

A modification to this approach was required for parallel or threaded applications. Quartz [4] also interrupts the execution of the application and samples the program counter, but during the interrupt it also checks to see how many of the other CPUs are busy. This information is used to compute the normalized processor time:

$$\sum_{i=1}^{P} \frac{Time_with_i_cpus_busy}{i}. \qquad (1)$$

When reporting normalized processor time, each term is summed separately and the time spent in each state is also reported (e.g., busy, spinning, blocked, ready).

Figure 2 shows an example of the type of output that might be produced using a tool like Quartz (the example is a slightly modified version of the one used in the original paper [4]). For each of the possible numbers of processors that could be used it shows a breakdown of the normalized processor time spent on each of the main routines. Here one can see that a significant portion of the time is spent executing the "cone" routine sequentially. As a result, it provides guidance that indicates that if one could parallelize this routine, the execution time of the application could be significantly reduced and better speedups could be obtained.

The Quartz paper also describes further improvements made to the example program and other experiences using their tool to improve parallel programs.

3.2.1 Pros:

- Easy to use (little or no barrier to entry).
- Language independent.
- No understanding of system required.
- Focuses attention in the right place through strong ties with the source code.

3.2.2 Cons:

- Requires source code and recompiling.

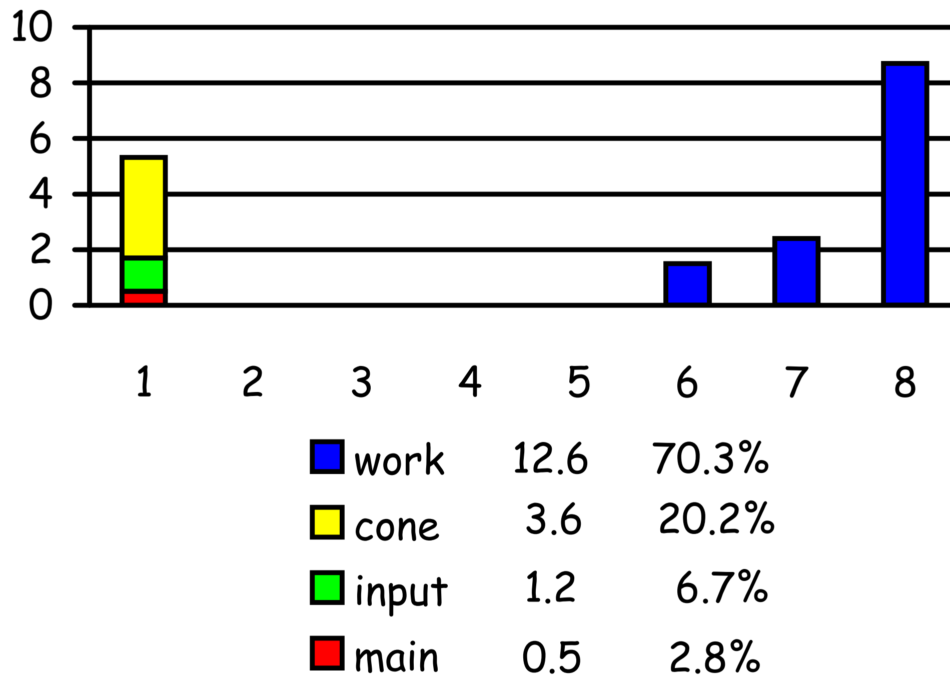

Figure 2: Normalized processor time (diagram modified from [4])

- Mostly helps to figure out where to work on performance and not much help in how to improve performance.
- Fairly limited scope of applicability (compiled application).

3.2.3 Scope of Applicability

The scope of applicability is fairly limited here. It is mainly helpful in trying to determine where a program is spending much of its time executing and not necessarily why. It is generally not very helpful in understanding the performance of anything except an application.

3.2.4 Possible Avenues for Investigation

- Support for complex systems and services. In applications that consist of multiple executing programs (possibly executing on different hosts), it might be possible to profile each of the programs and to somehow combine and correlate the collected data. See the work done by Digital/Compaq on Continuous Profiling [3] for an example of a similar approach with a larger scope of applicability.

3.3 Linker-based Code Insertion

One technique that has been deployed for some time but until recently hasn't been discussed in detail is the use of a method for shared library interposition such as a shim or wrapper.[4] Recent work has described this technique in detail and demonstrated how it can be deployed [41] [28]. While the authors of this work make no claim that the idea is original (as they've heard numerous anecdotes of this or similar techniques being used by others), this is the first known work describing the approach in detail and demonstrating how it can be used to measure a real system.

A `shim` is a thin piece of material placed between two objects. In software, it is a small function that essentially replaces the real function and is used to collect performance data regarding calls to the real function. The shim is responsible for implementing the same interface, passing all parameters through to and calling the real function, collecting any performance data of interest, and returning results to the calling (if there are any).

This technique can be applied whenever the interface to the function being instrumented is well known. Figure 3 shows that the shim can be placed between the caller and the callee because the interface is known.

Figure 3: Using shims to insert instrumentation code (diagram from [41])

In an environment that supports shared libraries, each shim implements the interface to the real function using that function's name. In this way the shim will be called instead of the real function. The key is being able to call the real function from within the shim.

This is accomplished, for example, by compiling the shims into a dynamically linked library named `libshimdo.so` and then renaming that library to the name used by the original library (e.g., `libdo.so`) but storing the shim library in a different directory (e.g., /usr/lib/shims/libdo.so). Next, the search order for dynamically linked libraries is changed so that the shim library is found and used prior to the original library (e.g., `/usr/lib/libdo.so`). This order can be changed on many systems by changing the environment variable `LD_LIBRARY_PATH`. Then the program being instrumented is relinked using an option that instructs the linker/loader to call an initialization function, `init_shims()` when the dynamic libraries are loaded. In some environments this is done by passing the option `-init init_shims` to the linker. When the program is run and the dynamic libraries are loaded it calls the function `init_shims`, which loads the actual library and

[4] This approach is reportedly used in the SUN Thread Event Analyzer [62].

renames the functions to the names used by the calling shim function. Examples of a function's prototype, shim, and initialization code are shown in Figure 4.

```
int do(int param);          // prototype

int do(int param)           // instrument the real function
{
    int tmp;
    do_call_count++;
    start_time(do);
    tmp = real_do(param);
    stop_and_record_time(do, do_call_count);
    return(tmp);
}

void init_shims()          // load and link the real function
{
    dynlib = dlopen("/usr/lib/libdo.so");
    real_do = dlsym(dynlib, "do");
}
```

Figure 4: Example prototype, shim and initialization code

A similar approach is commonly known and used at the source code level, but the the key advantage of using a shim is that it can be used without the source code for the program being instrumented and it can be easily inserted and removed.

3.3.1 Pros:

- Doesn't require source code.
- Can collect data of interest.
- Language independent.
- Does not require a detailed understanding of the system.

3.3.2 Cons:

- Limited opportunities for instrumentation (can really only instrument at interfaces).
- Potentially time consuming and somewhat error prone (need to create the shims).

- Fairly limited scope of applicability (information captured mainly applies to the application, libraries and/or the operating system).

3.3.3 Scope of Applicability

If used with hardware performance counters this technique could be used to monitor hardware performance. It can also be used to measure the performance of an application, libraries and/or aspects an operating system and possibly combinations of the above. The papers describing this technique [41] [28] use it to measure the performance of a distributed computing environment (DCE) and are quite successful at identifying where time is being spent in a distributed application.

3.3.4 Possible Avenues for Investigation

- The use of shims seems to offer considerable benefits while being relatively easy to use. The difficulty lies mainly in creating the shims and in determining what to measure. There might be considerable value in producing a set of shims that could be widely used. A user could then link with the versions of the libraries containing the shims and control options to specify which shim functions would collect data and what data would be collected.

- It might also be interesting to examine techniques for automatically generating a set of shims. That is, given a set of function prototypes might one be able to automatically generate a set of useful shims.

3.4 Rewriting Binaries to Insert Code

Recently much research [19] [7] [54] [31] [32] [5] [46] has been conducted into techniques and tools for rewriting program executables after they have been compiled and linked so that they contain additional code in order to instrument some aspect of the application or system being studied. Interestingly, some of this research has involved modifying executables at run-time [45] [20] [37] [59] [12] in order to dynamically insert and delete instrumentation code. Previous work had dynamically modified code for debugging purposes [26] [10].

Typically each of the studies described above also includes a description of how they've applied their tool or technique in order to measure and improve the performance of an existing system.

Furthermore, the ability to instrument executables has also been used to develop other tools, some of which are not strictly designed for measuring performance. For example a tool has been developed for: detecting data races in multi-threaded programs [49], instrumenting threaded applications [63], and implementing fine-grained distributed shared memory systems [64] [51] [50].

One of the earlier known tools that utilized binary rewriting techniques is *pixie* [15] [38]. Pixie relies on the compiler to instrument an executable to collect performance related data during execution. After execution it uses that data to rewrite the binary to improve performance based on the previous execution or executions of that program.[5]

[5] Several papers classify Pixie as a binary rewriting tool but Pixie may actually require one to recompile the program and the compiler may utilize the collected performance data to generate more efficient code.

The collected data is used to generate more efficient code. For example, the program may be made more efficient by having better information regarding branch predictions and the number of times loops execute (for loop unrolling).

The procedure used to rewrite binary files in order to instrument program behaviour (including performance) is as follows:

1. Determine the location where instrumentation code is to be inserted.
2. Patch the executable to include the new instrumentation code but ensure that all existing instructions are executed properly.

For example, a typical tool might instrument all loads and stores in order to obtain memory reference traces, conduct cache studies, or implement a fine-grained distributed shared memory system. Figure 5 shows an example of how an instruction (e.g., a load or store) might be replaced with code that calls or *trampolines* to instrumentation code and transfers execution back to the original instructions.

```
instr x                 instr x
instr x+1               call <code patch>
instr x+2               instr x+2

<code patch>
// inserted measurement code
instr x+1  // or equivalent
// branch back to next instruction
```

Figure 5: Example of rewriting executable code

The modification of code can change the addresses of existing instructions. Therefore, these addresses may need to be modified in the new executable being produced. The approach typically used roughly three phases (based on the description in [5]):

1. Analysis
   - Break program into functions and basic-blocks.
   - Analyze control transfers (jumps, branches, calls).
   - Generate a flow control graph for the program.
2. Instrumentation
   - Insert, change, remove code.

3. Regeneration

   - Compute address translations.
   - Generate a translation table if required.
   - Regenerate the basic-blocks, patching control transfers at that time.
   - Update symbol table and relocation information if required.

Fairly early in the process of developing binary rewriting techniques people realized the sizable amount of effort that would be involved in developing a new tool that employs binary rewriting techniques. As a result, much initial work concentrated on tools for developing binary rewriting tools. A typical example of one such tool is EEL [32]. EEL provides a set of APIs and a C++ library to ease the process of binary rewriting. EEL handles loading the executable and producing a control-flow graph; it also provides access to and iterators for applying instrumentation to commonly instrumented portions of a program. For example, one can iterate across every routine, basic block and edge in the control-flow graph and delete instructions or add code using *snippets* which encapsulate machine-specific instructions and provide context dependent register allocation. Additionally, EEL handles the modification of calls, branches and jumps to ensure that the control flow is correct in the edited program.

Provided one is able to easily describe and locate points in the executable at which code is to be inserted or modified, the technique of rewriting executables can be utilized to implement very powerful tools. Although it is not strictly designed as a tool for performance measurement, Eraser [49] is a good example of one such tool. Eraser is designed to detect and report on data races in multithreaded programs. This is done by rewriting binaries to instrument every shared-memory reference and by verifying that locks are used consistently. Unprotected accesses to shared data are reported as well as inconsistent use of locks. The general idea behind how this works can be seen by examining their first lockset algorithm which is used to track the set of locks used by each thread [49]. The algorithm is subsequently refined later in the paper but this version provides a good understanding.

Let $locks_held(t)$ be the set of locks held by thread t
For each shared variable v, initialize the candidate locks $C(v)$ to the set of all locks.
On each access to v by thread t,
    set $C(v) = C(v) \cap locks_held(t)$
    if $C(v) = \{\}$, then issue a warning

One of the reasons for describing this approach in detail is that it might be possible to use a similar approach in order to measure lock performance and possibly even to provide suggestions regarding lock splitting. This is discussed in slightly more detail in the section on possible avenues for investigation (Section 3.4.4).[6]

A more flexible approach to performance measurements can be achieved by modifying the program binary dynamically during its execution [20]. One of the key advantages of dynamic instrumentation is that it permits the program to execute un-instrumented until it reaches the point

[6] One would have to see if this is a better approach than using dynamic instrumentation for threaded programs as described in [63].

of interest. This not only avoids the overheads due to instrumentation prior to the point of interest but also can significantly reduce the amount of data collected (because instrumenting code can be added and deleted when needed). This approach has been used to build a system for measuring the performance of parallel applications and systems called Paradyn [37]. In Paradyn, instrumentation is controlled by the Performance Consultant module, which is used to automatically direct the placement of instrumentation code. Since it has information regarding performance bottlenecks and program structure it can associate bottlenecks with specific parts of a program. Additionally, overhead due to instrumentation can be limited by setting a user controlled threshold and performance data can be visualized using a supplied visualization library.

This work on dynamic instrumentation has continued by extending the technique so that it can also be used with threaded applications [63] and operating systems [59]. Additionally, work has been done to make the code available and to publish an API for runtime code patching [12].

3.4.1 Pros:

- Doesn't require source code.
- Language independent.
- In some cases direct correlation between performance data and source code (e.g., [37]).
- Fairly good opportunities for instrumentation (can really instrument at many places/levels).

3.4.2 Cons:

- It would seem that a greater number of useful tools would have been generated.
- Fairly large barrier to entry unless there exists a tool that measures what you want.
- Potentially time consuming (if learning and using one tool to generate a new tool).
- Fairly limited scope of applicability (information captured mainly applies to the application, libraries and/or the operating system).

3.4.3 Scope of Applicability

If used with hardware performance counters this technique could be used to monitor hardware performance as has been done in [2]. It can also be used to measure the performance of an application, libraries and/or aspects an operating system, and possibly combinations of the above.

3.4.4 Possible Avenues for Investigation

- Utilize the technology to create useful tools. Might it be possible to create something like Eraser but use it for measuring overhead due to locks and lock contention and for providing insights into how to split locks. For example, might one be able to monitor the shared data that is protected by the same lock but not accessed concurrently and determine that the single lock might be split.

- More and better demonstrations of utility of the tools and techniques. Typically papers show a relatively trivial application of the tools or technique to improve the execution of a fairly simple program.
- Look at methods for improving the scope of applicability. For example, could one instrument several programs so that applications consisting of multiple executing programs and/or processes (possibly communicating) could be measured. This might be plausible given the work that has been done with Shasta [50].

4 Direct Observation Without Instrumentation

This category of tools and techniques involves executing the system of interest without making any modifications. In this case, performance is observed using methods that are external to the executing system. Common techniques include instruction set simulation, hardware profiling, and execution tracing.

4.1 Instruction Set Simulation

Simulation has been deployed as a tool for performance analysis for a long time. One of the problems is that it typically involves trading off speed of simulation for accuracy in what is being simulated. Advances in instruction set simulation [8] [16] [36] [30] [39] [35], combined with increasing processor speeds have made it feasible to provide highly accurate instruction set simulators. Using such simulators one is able to take unmodified binaries that have been compiled and linked for the target architecture and execute them on a simulator.

Some of the important work in this area has been the development of techniques for greatly improving the speed of the simulation while maintaining accuracy. Shade [16] drastically improved simulation speeds by cross-compiling instructions for the target machine into instructions for the host machine that is running the simulator. By caching host-code and by integrating trace information into the host-code they are able to amortize the cost of cross-compilation and enable high-speed simulation and data collection.

An alternate approach taken by SimGen [30] and subsequent work on SimICS [39] uses an intermediate representation for instructions which is designed to be easy to simulate in software. SimGen takes a definition of the target instruction set and generates a highly-efficient intermediate representation, instruction decoder, disassembler, encoder, and required service routines. The belief is that this approach is able to generate simulators that execute more quickly than those that can be generated manually. One way that speed is improved is by generating versions of the simulator that collect information about the execution of the simulator and then feeding that information about the simulator's behaviour back into the generator to produce a faster version. These papers typically include some measurement of the *slowdown* of an application executing on the simulator. This is how much slower the program executes on the simulator when compared with the execution on the actual target machine. In the case of the benchmarks used in [39] slowdowns are typically in the range of 30-50 times slower.

Originally simulators were restricted to executing only individual programs. However, recent advances [47] [35] have enabled the simulation of hardware devices and now enable the full scale

simulation of a complete operating system. A standard operating system is booted and executed inside the simulator. This provides users with new insights concerning both the execution of the operating system and the behaviour of complex systems under more realistic and mixed workloads.

4.1.1 Pros:

- Run executables on simulator.
- Language independent.
- Does not require a detailed understanding of the system in order to use it (but might to interpret the results).
- Quite good scope of applicability.
- Simulator can be as detailed as desired. Can collect any or all data of interest.

4.1.2 Cons:

- Slowdowns are too large to make this approach practical for some applications.
- Performance information provided is typically geared more towards hardware designers and compiler writers because these have usually been the people who have developed and used such simulators. Information that is typical concerns cache misses, instruction stalls, and information regarding branch predictions.

4.1.3 Scope of Applicability

Because the simulator can be modified, one can collect any or all information that is of interest. This enables data to be collected regarding the underlying hardware, the application, libraries, and the operating system.

4.1.4 Possible Avenues for Investigation

- While simulators can provide significant insights it is not clear that they has been used very widely. It would be interesting to study and rectify this situation. For example, it is quite likely that the information provided by simulators is at a level that is too low for the average user.
- One might be able to simulate more complex environments by connecting several machine simulators (possibly even executing on different hosts). This might be used, for example, to more effectively study client/server and or distributed systems.
- It might also be possible to combine the use of machine simulators with a network simulator, like *ns* [60] [6] to provide a more complete environment and a very detailed picture of the performance of fairly complex systems.

4.2 Hardware Profiling

As modern processor design has become more complex and as workloads executing on systems built using these processor have increased in diversity, the set of tradeoffs made when designing processors has become overwhelming. One technique being used by designers in order to better understand both the efficacy of their designs and the workloads being executed on them has been to include methods for obtaining information related to performance directly from the processor.

This is done by including one or more control and counting registers directly on the processor. By manipulating the control register(s), the programmer is able to control what information is to be collected in the performance counting registers. While executing, the processor updates counters as guided by the control register(s). An interrupt can be generated when a counter register is about to overflow in order for the value to be recorded in software. As a result, an advantage of this technique is that it can potentially offer very accurate information with extremely low overhead. A few examples of the types of data that can be collected are: instruction counts, instruction stalls, and cache reads, writes and misses.

Hardware performance counters can be used in one of two ways. First a program can be instrumented to make direct calls to obtain the data from the counters (this would fit under the classification of instrumented programs). This would be classified as a direct observation technique using instrumentation. The other way they can be used is to utilize the performance counters at a different level, thus not requiring modifications to the program being measured. This would be classified as a direct observation approach without instrumentation and has been used in Digital/Compaq's Continuous Profiling Infrastructure (DCPI) [3].

4.2.1 Hardware Performance Counters and Instrumentation

Although out of place with respect to the other techniques in this category we now briefly discuss issues related to instrumenting programs to use hardware performance counters. This is done in order to keep the discussion of the two different uses of hardware performance counters together.

One of the problems facing people who want to use these performance counters is that each processor provides different counters and different methods for controlling their use. The problem is exacerbated by the fact that each operating system provides a different mechanism for accessing these counters. In order to provide a clean and portable interface, some work has begun to provide a standard library for accessing performance counters on common platforms. PCL (Performance Counter Library) [9] provides such support by designing and implementing a library that has been ported to several combinations of processors and operating systems.

A competing project, PAPI (Performance API) [11] is attempting to define a standard that can be used to program and access performance counters on different platforms. They have defined two interfaces. The first is a high-level interface for acquiring simple measurements and the second is a fully programmable, thread safe, low-level interface for more sophisticated users. PAPI provides some important features that are not supported by PCL. For example PCL does not provide methods for handling counter overflow or for software multiplexing.

As hardware designers are not yet ready to commit much real estate to what is often perceived as frivolous functionality such as performance counting registers, the number of counters is typically severely limited and as a result the data one is able to gather at one time is severely limited.

As a result DCPI [3] and PAPI [11] support the multiplexing of hardware performance counters in order to provide a more complete picture of a program's execution.

It is interesting to note that some work [2] has combined the use of hardware performance counters with program instrumentation techniques.

4.2.2 Using Hardware Performance Counters Without Instrumentation

The Digital/Compaq Continuous Profiling Infrastructure (DCPI) [3] utilizes hardware performance counters to implement a sampling-based profiling system designed to run continuously on production systems with relatively low overhead (1-3% slowdown for most workloads). The system consists of two parts: a data collection system that relies on periodic interrupts generated by performance counters available on Alpha processors to sample program counters (periodically recording them to disk) and tools for analyzing the stored data. Collected data is used to produce typical information of interest, such as the time spent per image, procedure, source line and instruction. In addition, a more complex analysis tool is able to determine the average number of cycles spent executing each instruction.

The power of this technique of continuous profiling is that because of its low overhead it can be used on production systems to examine commercial workloads. Much of the paper describes the techniques used in order to support the high sampling rate.

4.2.3 Pros:

- One of the advantages of performance counters is that they can often be used to obtain information at a much higher resolution than other techniques. As an example, one can obtain a count of the actual number of instructions used to execute a function.
- The technique of continuous profiling doesn't require source code and it is language independent.
- Continuous profiling has a very low barrier to entry. Essentially all programs are being measured so it's a matter of using the analysis tools to determine information regarding performance.

4.2.4 Cons:

- As is the case for simulators, performance information provided is typically geared more towards hardware designers and compiler writers because these have usually been the people who have developed and used hardware performance counters. Information that is typical concerns cache misses and instruction stalls.
- Correlating the information obtained with the source code is typically not as easy as one would like.

4.2.5 Scope of Applicability

This technique can also be used to measure the performance of an the underlying processor, an application, libraries and/or aspects of an operating system and possibly combinations of the above.

4.2.6 Possible Avenues for Investigation

- While most microprocessors contain hardware profile counters it's not clear that they are being utilized, except perhaps by the most performance hungry users of parallel computers. One would think that better tools could be developed that would make it easier to use performance counters and that would provide a more clear link between the performance data and where and how the system would be modified to improve performance.
- One might be able to use information from separate systems running continuous profiling to obtain better and more detailed information about the performance of more complex systems. For example, client/server and/or distributed systems.

4.3 Execution Tracing

There exist some tools that permit one to trace an existing program without modification to that program. An example is those programs designed to trace system calls, which include: `strace` [34], `truss` [58], and `tusc` [21].

We're not aware of research related to such tools but wanted to include them because they can be very powerful for understanding and debugging systems that aren't working. We're also not sure how often they are used to measure performance but perhaps they could be extended or improved to make them more effective or more widely applicable.

4.3.1 Pros:

- Doesn't require source code and is language independent.
- Very easy to use.

4.3.2 Cons:

- Only provide information related to system calls and signals.
- Fairly limited scope of applicability.

4.3.3 Scope of Applicability

The existing tools report on all system calls and signals and with the proper options will report on the time that elapses between each system call.

4.3.4 Possible Avenues for Investigation

- Might it be possible to modify `strace` to include the time spent in each system call and to provide a summary report at the end of the execution.
- Might it be possible to produce a similar tool that can be used for a broader scope than system call. For example, might one be able to leverage shim technology to build a tool that instead of tracing and reporting on system calls reports on library calls. Can this be extended to include all function calls?

5 Indirect Observation

There is a subtle but important difference between our category of "Direct Observation Without Instrumentation" and "Indirect Observation". The key difference is whether the actual system that one would like to improve is executed while measurements are being made. If the system of interest is running while data is being collected, we view this as direct observation. If it is not running while data is being collected or only a portion of the system is running, we consider this to be indirect observation.

The technique of indirect observation is often used when the system of interest can't be executed in order to measure performance or the metric of interest can't be obtained during the system's execution.

As an example, consider a web server that is not providing the level of desired throughput. One might suspect that the 1 Gbps network connection is a bottleneck and observes that during execution a goodput of 560 Mbps is obtained. There may not be much that can be done in the context of the running web server to test this hypothesis. An indirect approach might be to run an external program (for example, `netperf` [25] using appropriate sizes for the average request and response and to observe the goodput obtained over the suspected bottleneck link. If the observed goodput with `netperf` is near 560 Mbps thus may indicate that the network is a potential bottleneck. However, if the goodput obtained using `netperf` is significantly higher than 560 Mbps that would indicate that the network bandwidth is likely not the bottleneck.

5.1 Benchmarks and Workload Generators

There are a large number and variety of methods for performing indirect measurements. Two of the main approaches are to use benchmarks or workload generators of some sort (which may include artificial or trace driven workloads) or to use some form of special purpose observation tool. We now very briefly give examples of some work being conducted in each category.

5.1.1 Parallel Application Benchmarks

An example of some benchmarks that have been produced and studied in some depth are the SPLASH parallel application benchmarks [61]. This set of benchmark applications has been used for a variety of purposes including: microprocessor design, multiprocessor interconnect design, compiler design and implementation, and evaluating and comparing cache coherence schemes in multiprocessor and distributed virtual shared memory systems. These applications have been used in multiprocessor environments in much the same way that the SPEC benchmarks have been used in uniprocessor environments.

5.1.2 World-Wide-Web Workload Generators

A significant topic of interest is the performance of web and proxy servers. As a result, several techniques have been used to measure their performance. One technique is to use data obtained from traces gathered from real web sites and then to use the traces to drive queries from a workload generator.

One of the problems with these approaches is that it hasn't been clear how to scale the workload for larger systems nor how to generate a workload that is significantly more demanding than the server is capable of handling. One approach to offering sustained load and overload for web servers is provided in the `httperf` [40] tool. The key observation made in this work is that once servers become saturated, they are unable to accept and process new connections from clients and as a result workload generators are significantly slowed because of delays and timeouts caused by TCP. In order to maintain a high rate of requests, clients that are generating load must also timeout and terminate unresponsive connections.

5.1.3 Pros

- If benchmarks are representative and repeatable they can be used to gain very powerful insights.
- One can develop benchmarks that are based on predictions of future workloads.

5.1.4 Cons

- The benchmarks must be representative of real workload.
- System designers and builders can spend too much time developing systems that run benchmarks really well.
- Running benchmarks and analyzing the data can consume considerable time and effort. For example, benchmarking a highly-scalable multi-tiered e-service architecture could require considerable hardware, software and human resources just to get to the point where the system could be tested.

5.1.5 Possible Avenues for Investigation

It would seem that this is an area where there is much potential for future work. Some possibilities include:

- How does one design workloads that scale in order to effectively drive large scale systems and supposedly scalable systems?
- It would be interesting to investigate techniques for monitoring and reporting web server response times as seen by clients. One would likely start by looking at products that have appeared on the market recently that reportedly provide this service.
- A difficult problem for e-service providers is figuring out how to measure their system's performance in such a way that bottlenecks are identified. It would seem that there is much work that could be done in this area.

5.2 Observation Tools

Tools designed for observing the performance of a system could be used to conduct direct or indirect observations. However, because some of these tools are unable to provide only information regarding a specific program or subsystem we discuss them here, under the category of techniques and tools for indirect observation.

Two examples of recent work conducted in this area involve tools for observing network performance.

The first example is `clink` [18] which was developed as a result of conducting experiments and observing problems with the accuracy of the bandwidth and latencies reported by `pathchar` [23]. By using better and adaptive sampling techniques `clink` is able to provide more accurate results, often in a shorter time.

A second example is `Sting` [48] which has been developed to measure network packet loss rates. Some of the advantages of `Sting` are:

- It is able to measure packet loss rates in both forward and backward directions. `ping` and `traceroute` (see [55] [56] for more information) can't handle loss asymmetry and are susceptible to ICMP filtering.
- It does so without requiring measurement infrastructure such as software daemons running on both the sender and receiver of interest as required by existing approaches [43] [1] [44].

In order to measure packet loss one needs to know the number of packets sent and the number of packets received. A one-way loss rate is calculated as $1 - \frac{packets_received}{packets_sent}$. At the source you can know the number of packets sent but not the number received and at the destination you know the number of packets received but not the number sent. The key question is then how do you obtain such measurements without using measurement infrastructure.

The big contribution made in this work is to recognize that one can use TCP's error control mechanisms to derive the unknown variable. Sting cleverly uses these mechanisms and knowledge of TCP's behaviour in order to first send some data (a phase they call data-seeding) and then to discover which packets were lost (a phase they call hole-filling). Hole-filling is accomplished by first sending a packet with a sequence number that is one larger than the last packet sent in the data-seeding phase. If the destination responds with an acknowledgement, it either indicates where the hole is or that there is no hole (because TCP uses inclusive ACKS and it will ACK the last last packet sent). For each hole discovered the source retransmits the missing packet and the procedure is repeated until all holes have been discovered and filled.

5.2.1 Pros

- These tools are typically very easy to use.
- They can be used directly while the system of interest is executing or indirectly.

5.2.2 Cons

- Some of these tools provide only general system wide information so it can often be difficult to determine the performance of an individual entity of interest.

- Some tools are designed to very specifically measure or observe only one aspect of system behaviour.

5.2.3 Possible Avenues for Investigation

- Of course one imagines there is always a need for new and better tools. One approach here would be to determine what aspects of performance aren't covered well by existing tools. Another approach might be to extend existing tools (e.g., to perhaps direct more focused attention to particular aspects of the system).

  For example, we are not aware of a tool or options to a tool that can be used to show the network bandwidth being consumed by an individual process. It might prove useful to develop a tool that operates like top but that can be used to show the bandwidth being consumed by each process (and on which interface), as well as the bandwidth being consumed on each interface, and the total amount of bandwidth being consumed in the entire system. Because this may require kernel support an interesting question would be, could this be done efficiently.

- We expect that more work could be done towards effectively using information obtained from combinations of tools to provide better insights into system performance and to help pinpoint bottlenecks. A starting point here might be to determine the main weaknesses of SE [17].

6 Summary

Because applications, systems and services are becoming increasingly tied with a corporation's bottom line, there is a growing concern with their performance. In many instances good performance is essential – or more to the point, poor performance can be costly. As a result, much time and effort is expended measuring, improving and continually monitoring the performance of key computer and communication components. We believe that the amount of time and effort spent on these endeavours can be significantly reduced by using tools and techniques that clearly pinpoint problem areas. With the growing complexity of computer and communication systems it is also becoming more important for these tools to not only help determine where problems lie but to provide help in determining how to alleviate these problems.

6.1 Some Potential Opportunities

Our goal is to identify key areas of need and to investigate tools and techniques for to help to quickly, easily and accurately diagnose and solve performance problems. To this end we now briefly discuss a few of the more interesting or important areas where we believe existing research could be improved or extended or where new opportunities might exist.

6.1.1 Measuring and Improving Performance in Complex Environments

Many applications and services are being created by combining several communicating programs, often executing on different systems. We believe that further work is required to better serve the

needs of this type of environment. This may require new approaches and new tools or perhaps a combination of existing methods can be deployed simultaneously. Interesting questions here are how does one: determine the right combination of tools, avoid excessive overhead, and can collected data be correlated in such a way that it can identify problem areas and provide insights into alleviating the problems.

6.1.2 New and Improved Tools that Leverage Existing Technology

Some of the recently developed technologies offer considerable opportunities if utilized effectively. Since some of these techniques have only recently been developed and because tools developed during research are quite often designed mainly as a proof of concept for the underlying methods, some existing tools could be significantly improved. Some tools appear to be fairly difficult to deploy, produce data that will be difficult for an average person to interpret, do not supply enough information with regards to pinpointing the source of the problem, are not very helpful when trying modify the system to alleviate problems, or all of the above. Potential work in this are would be to more effectively utilize existing technology to more quickly, easily and directly locate problems and improve performance.

6.1.3 Binary Rewriting to Reduce Lock Contention

This work would examine the use of binary rewriting technology in order to track the time and application spends holding various locks, track shared data accessed while those locks are held and investigate techniques for determining where locks might be split and/or moved in order to enhance performance. This would operate in a fashion that is similar to Eraser but with a significantly different goal. Information could also be gathered during this time that might be useful in projecting bounds on parallel application performance when executing in larger multiprocessor environments.

Although it is not clear whether it would be possible, ultimately a tool such as this would rewrite the binaries to automatically split locks and/or move locks to reduce lock contention.

6.1.4 A Note on Tuning Notes – Autotuning

Even before the advent of the World-Wide-Web, but especially since its development companies and organizations have produced a number of short documents (sometimes white papers) explaining how to tune an application or operating system in order to improve performance. Good examples of this type of document are the vast number of web pages devoted to tuning your particular combination of operating system and web server in order to obtain peak performance. There are often several different versions available that have been produced by the organization who developed the operating system, the organization that produced the web server, and individuals who are trying to help others by providing them with insights they've gained while trying to improve performance in their environment.

One of the biggest complaints I've heard and experienced myself with these documents is that they fail to explain which of the tens or hundreds of "tweaks" apply under which circumstances, which are the most important, and the impact they are likely to have on performance.

Useful work in this area could involve conducting a systematic study of the impact of such proposed tweaks and to document their impact on performance in order to provide more useful information to those who are faced with the prospect of tuning system performance.

However, a more interesting, challenging and potentially rewarding approach would be to investigate methods for automatically measuring and tuning systems. The idea would be to run a representative workload (or even the actual workload), gather performance data, automatically analyze the data, make those tweaks determined to most likely be helpful, and to repeat the process. Ideally performance would converge so that peak-performance would be obtained after a few iterations.

This would first require first ensuring that mechanisms exist for easily changing important performance parameters (i.e., such an approach would be much less helpful if each test run requires recompiling the kernel and/or rebooting the operating system).

7 Future Work

Clearly many performance measurement and analysis tools are being developed by a number of companies, many of which specialize in performance tools. Before reinventing an existing tool, further work is required to survey existing products. This will be more difficult than survey existing research since in some cases the only way to understand what a tool is really capable of is to purshase and use that tool. An additional difficult with respect to developing new techniques is that companies do not like to disclose the methods used to obtain information used by their tools. Hopefully the increase in the number of available demo versions of tools will ameliorate the problem with testing expensive software.

References

[1] G. Almes. Metrics and infrastructure for IP performance. http://www.advanced.org/-surveyor/presentations.html, 1997.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, 1997.

[3] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[4] T.E. Anderson and E.D. Lazowska. Quartz: A tool for tuning parallel program performance. In *The ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, May 1995.

[5] R. Arpaci and M. Fahndrich. revEELing Solaris. IRAM project Spring 96, Computer Science Division, EECS University of California at Berkeley, May, 1996.

[6] Sandeep Bajaj, Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, Padma Haldar, Mark Handley, Ahmed Helmy, John Heidemann, Polly Huang, Satish Kumar, Steven McCanne,

Reza Rejaie, Puneet Sharma, Kannan Varadhan, Ya Xu, Haobo Yu, and Daniel Zappala. Improving simulation for network research. Technical Report 99-702b, University of Southern California, March 1999. revised September 1999, to appear in IEEE Computer.

[7] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *The 19th Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[8] R.C. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the Winter 1990 USENIX Conference*, pages 53–63, January 1990.

[9] R. Berrendorf and H. Zieler. PCL – the performance counter library: A common interface to access performance counters on microprocessors. Version 1.3, http://www.fz-juelich.de/-zam/PCL.

[10] J.S. Brown. The application of code instrumentation technology in the Los Alamos debugger. Technical report. Los Alamos National Laboratory, October, 1992.

[11] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[12] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[13] Peter A. Buhr and Robert Denda. uprofiler: Profiling user-level threads in a shared-memory programming environment. In *The Second International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, pages 159–166, December 1998.

[14] H. Frank Cervone. *Solaris 7 Performance Administration Tools*. McGraw Hill, New York, 2000.

[15] F. Chow, A.M. Himelstein, E. Killian, and L. Weber. Engineering a RISC compiler system. In *IEEE COMPCON*, March 1986.

[16] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[17] A. Cockcroft and R. Pettit. *Sun Performance And Tuning: Java and the Internet*. Sun Microsystems Press (Prentice-Hall), Mountain View, CA, 1998.

[18] Allen B. Downey. Using pathchar to estimate internet link characteristics. In *Measurement and Modeling of Computer Systems*, pages 222–223, 1999.

[19] A.J. Goldberg and J. Hennessy. Performance debugging shared memory multiprocessor programs with MTOOL. In *Supercomputing '91*, pages 189–200, November 1991.

[20] J.K. Hollingsworth, B.P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference*, pages 841–850, May 1994.

[21] HP-UX Manual Pages. *tusc man page*.

[22] IEEE Computer Society. ParaScope: A List of Parallel Computing Sites. http://www.-computer.org/parascope/.

[23] V. Jacobson. Pathchar: A tool to infer characteristics of internet paths. http://www.caida.org/-tools/utilities/others//pathchar.

[24] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling,*. Wiley-Interscience, New York, NY, USA, 1991.

[25] Rick Jones. Public Netperf Homepage. http://www.netperf.org/netperf/NetperfPage.html.

[26] P.B. Kessler. Fast breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*, pages 78–84, June 1990.

[27] Patrick Killelea. *Web Performance Tuning Speeding Up the Web*. O'Reilly and Associates. Inc., October 1998.

[28] D.P. Konkin, G.M. Oster, and R.B. Bunt. Exploiting software interfaces for performance measurement. In *Workshop on Software Performance*, pages 208–218, Santa Fe, NM, 1998.

[29] F. Lange, R. Kroger, and M. Gergeleit. JEWEL: Design and implementation of a distributed measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):657–671, 1992.

[30] F. Larsson, P. Magnusson, and B. Werner. SimGen: Development of efficient instruction set simulators. Technical report. SIC-R97/03, Swedish Institute of Computer Science, November, 1997.

[31] James R. Larus and Thomas Ball. Rewriting executable files to measure program behaviour. *Software, Practice and Experience*, 24(2):197–218, February 1994.

[32] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. *ACM SIGPLAN Notices*, 30(6):291–300, 1995.

[33] E. D. Lazowska, J. L. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice-Hall, 1984.

[34] Linux Manual Pages. *strace man page*.

[35] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrom, and B. Werner. SimICS/sun4m: A virtual workstation. In *Usenix 1998 Annual Technical Conference*, pages 119–130, 1998.

[36] P. Magnusson and B. Werner. Efficient memory simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium*, pages 62–73, 1995.

[37] Barton P. Miller, Jon Cargille, R. Bruce Irvin, Tia Newhall, Mark D. Callaghan, Jeffrey K. Hollingsworth, Karen L. Karavanic, and Krishna Kunchithapadam. The Paradyn parallel performance measurement tools. *IEEE Computer*, pages 37–46, November 1995.

[38] MIPS Computer Systems. *UMIPS-V Reference Manual (pixie and pixstats)*, 1990.

[39] P. Magnusson J. Montelius. Performance debugging and tuning using an instruction set simulator. Technical report. SIC-T–97/02-SE, Swedish Institute of Computer Science, June, 1997.

[40] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59–67, June 1998.

[41] Greg Oster and Rick Bunt. Using Shim technology to monitor DCE runtime performance. In *IBM Center for Advanced Studies Conference (CASCON)*, 1997.

[42] Parallel Tools Consortium. The Parallel Tools Consortium. http://www.ptools.org/.

[43] V. Paxson. End-to-end routing behaviour in the internet. In *Proceedings of SIGCOMM 1996*, pages 25–38, August 1996.

[44] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. RFC-2230, May, 1998.

[45] B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richard, and W. Smith. The paragon performance monitoring environment. In *Supercomputing '93*, pages 850–859, November 1993.

[46] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–8, August 1997.

[47] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[48] Stefan Savage. Sting: A TCP-based network measurement tool. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[49] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[50] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *The 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[51] I. Schoinas, B. Falsafi, A.R. Lebek, S.K. Reinhardt, J.R. Larus, and D.A. Wood. Fine-grain access control for distributed shared memory. In *The 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[52] S. Shende. Portable profiling and tracing for parallel scientific applications using C++. In *ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pages 134–145, 1998.

[53] Sameer Shende. Profiling and tracing in Linux. In *Proceedings of the Extreme Linux Workshop*, June 1999.

[54] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 29(6):196–205, 1994.

[55] W.R. Stevens. Addison Wesley, 1994.

[56] W.R. Stevens. Prentice Hall, 2nd edition, 1998.

[57] Sun Microsystems Inc. Sun Performance Information. http://www.sun.com/sun-on-net/-performance.html.

[58] SUNOS Manual Pages. *truss man page*.

[59] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.

[60] UCB/LBNL/VINT. Network simulator - ns (version 2). http://www-mash.CS.Berkeley.-EDU/ns/.

[61] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *The 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[62] P.T. Wu and P Narayan. Multithreaded performance analysis with sun workshop thread event analyzer (Revision 03). Technical report. Sun Microsystems Inc., Authoring and Development Tools, SunSoft, April, 1998.

[63] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *Principles Practice of Parallel Programming*, pages 49–59, 1999.

[64] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *The First Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.