Analyzing Root Causes of Latency Distributions

A Dissertation Presented

by

Avishay Traeger

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2008

Stony Brook University

The Graduate School

Avishay Traeger

We, the dissertation committee for the above candidate for

the degree of Doctor of Philosophy

hereby recommend acceptance of this dissertation.

Dr. Erez Zadok, Dissertation Advisor
Associate Professor, Computer Science Department

Dr. Klaus Mueller, Chairperson of Defense
Associate Professor, Computer Science Department

Dr. Robert Johnson
Assistant Professor, Computer Science Department

Julian Satran
Distinguished Engineer, IBM Haifa Research Lab

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Analyzing Root Causes of Latency Distributions
by

Avishay Traeger

Doctor of Philosophy
in
Computer Science

Stony Brook University

2008

OSprof is a versatile, portable, and efficient profiling methodology based on the analysis of latency distributions. Although OSprof offers several unique benefits and has been used to uncover several interesting performance problems, the latency distributions that it provides must be analyzed manually. These latency distributions are presented as histograms and contain distinct groups of data, called peaks, that characterize the overall behavior of the running code. Our thesis is that by automating the analysis process, we make it easier to take advantage of OSprof's unique features.

We have developed the Dynamic Analysis of Root Causes system (DARC), which finds root cause paths in a running program's call-graph using runtime latency analysis. A root cause path is a call-path that starts at a given function and includes the largest latency contributors to a given peak. These paths are the main causes for the high-level behavior that is represented as a peak in an OSprof histogram. DARC uses dynamic binary instrumentation to analyze running code. DARC performs PID and call-path filtering to reduce overheads and perturbations, and can handle recursive and indirect calls. DARC can analyze preemptive behavior and asynchronous call-paths, and can also resume its analysis from a previous state, which is useful when analyzing short-running programs or specific phases of a program's execution.

In this dissertation we present the design and implementation of DARC. Our implementation is able to find user-space and kernel-space root cause paths, as well as paths that originate in user-space and terminate in kernel-space. We also investigate the possibility of using OSprof and DARC in virtual machine environments. We show DARC's usefulness by analyzing behaviors that were observed in several interesting scenarios. We compared the analysis of these behaviors when using DARC to the manual analysis required by the original OSprof methodology, and found that DARC provides more concrete evidence about root causes while requiring less time, expertise, and intuition. In our performance evaluation, we show that DARC has negligible elapsed time overheads for normal use cases.

To my family:

אבא, אמא, נדב, יואב, ושירה

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank my advisor Erez Zadok for all of his support and guidance throughout the course of my Ph.D. career. There is a long list of things that I need to thank Erez for. I thank him for always being available for discussions, for taking the time to review any text I prepared for publication (including this dissertation), and for taking me to prestigious conferences and introducing me to colleagues. I thank him for allowing me to work independently and for helping me to develop the skills that allowed me to do so. I thank him for the internship referrals, for creating a great atmosphere in his lab, and for his dedication. I thank him for being a great advisor.

I thank Scott Stoller for serving on my dissertation proposal committee, Rob Johnson for serving on my defense committee, and Klaus Mueller and Julian Satran for serving on both. I am especially grateful to Julian for traveling a long distance to Stony Brook.

Thanks to the anonymous SIGMETRICS 2008 reviewers and my shepherd, Arif Merchant, for their valuable comments on the DARC publication.

I thank my mentors, managers, and colleagues during my two internships at the IBM research labs in Tel Aviv, especially Julian Satran, Dalit Naor, and Kalman Meth. I came up with the idea for this work and an initial design during my second internship there, and I thank them for allowing me to take the time to work on it.

I worked with many great people at the FSL over the years. First I would like to thank Ivan Deras Tabora and Vasily Tarasov for all of their help with the DARC project. They contributed in substantial ways to this work. I would like to thank Charles "Chip" Wright for answering the constant flow of questions during the start of my studies, for acting as my L1 cache, for showing me his tree-climbing technique, and for being a good friend. I thank Nikolai Joukov for all the guidance that he gave me during the OSprof project, for his input on the design of DARC and his help with Chapters 9 and 10. I also thank him for all of the good times that we had and for showing me how Russians celebrate birthdays. I thank Sean Callanan and Gopalan Sivathanu, who joined the FSL in the same semester as I did, for their company and help over the years. Sean also proofread almost every paper that I wrote (accept this won). Thanks to Jeff Sipek for answering my kernel-related questions and for the many trips through the stargate. Thanks to DJ for making the lab fun and for pushing me to do one more rep when I didn't think I had it in me. I appreciate Devaki Kulkarni's advice on benchmarking inside of VMware Workstation, and the VMware benchmark team for reviewing Chapter 10. I would like to also like to thank my other lab-mates that collaborated with me on other projects, provided me with good feedback on my work, or just provided me with some laughs: Akshat Aranya, Puja Gupta, Rakesh Iyer, Aditya Kashyap, Adam Martin, Kiran-Kumar Muniswamy-Reddy, Harry Papaxenopoulos, Dave Quigley, Abhishek Rai, Rick Spillane, Kumar Thangavelu, and Tim Wong.

I have several special people in my life who helped and supported me. I had many great and fun experiences with them throughout my Ph.D. career, and am very thankful for that. Thanks, alphabetically, to Andrew, Avideh, Dave, Eli, Jon, Raam, Tal, Tom, and Vered.

Last, but certainly not least, I would like to thank my family. My parents, although physically far away, still always managed to be there for me. I could not have done it without their support.

My cross-oceanic siblings, Yoav and Shira, were always close via frequent instant message conversations and phone calls, and Karen always brightened my day when she could get online (that is, when Ron and Lia weren't making trouble). My brother Nadav, his wife Zahava, along with Noam and Eyal were my immediate family in New York. Having them here was invaluable, and their support was always appreciated.

Further, my grandparents, Bubby, Saba, and Savta, were a great support as well, even though all they understood about my work is that it was "something with computers." Most importantly, though, they always made sure that I was eating. Zaidy also inspired me to succeed, and I am sure that he is proud. I also must give a special thanks to Helen, Steve, and their family, for taking me in as one of their own during my college years, and for not changing the locks when I went off to grad school. I also thank Miriam, Stu, and their family for always inviting me for holidays. I always appreciated my aunt, uncle, and cousins for stepping in as my immediate family when most of my immediate family was in Israel.

The department's systems and secretarial staff provided valuable administrative support. This work was partially made possible thanks to an NSF HECURA award CCF-0621463, an NSF CSR-AES award CNS-0509230, and an IBM Ph.D. fellowship.

Chapter 1

Introduction

An important goal of performance analysis is finding the root causes for some high-level behavior that a user observes. OSprof [27–30] presents these high-level behaviors to the user by collecting latency distributions for functions in histograms. These histogram profiles contain groups of operations, called peaks. Figure 2.1 shows example OSprof profiles for single and multiple processes calling the `fork` operation. We discuss this profile further in Chapter 2. For now, note that there are two distinct peaks in the multi-process profile (white bars): the first spans bins 15–19, and the second spans bins 20–25. These types of peaks are characteristic of OSprof profiles, and are indicative of some high-level behavior. In this case, the left peak characterizes the latency of the actual fork operation, and the right peak shows a lock contention.

These histogram profiles are presented to the user, and with OSprof, the user then manually analyzes the profiles using a variety of techniques. One technique is to compare peaks from two different profiles to reach some conclusion. To analyze the multi-process `fork` workload shown in the white bars of Figure 2.1, a user would need to have the expertise and insight to compare the profile to a single-process workload's profile. Because the right-most peak does not appear in the single-process profile, the user can guess that a lock contention caused the peak.

Despite the manual analysis required to analyze profiles, OSprof is a versatile, portable, and efficient profiling methodology. It includes features that are lacking in other profilers, such as the ability to collect time-lapse profiles, small profile sizes, and low overheads (in terms of time, memory usage, and code size). Based on user experiences, it is clear that interesting behavior

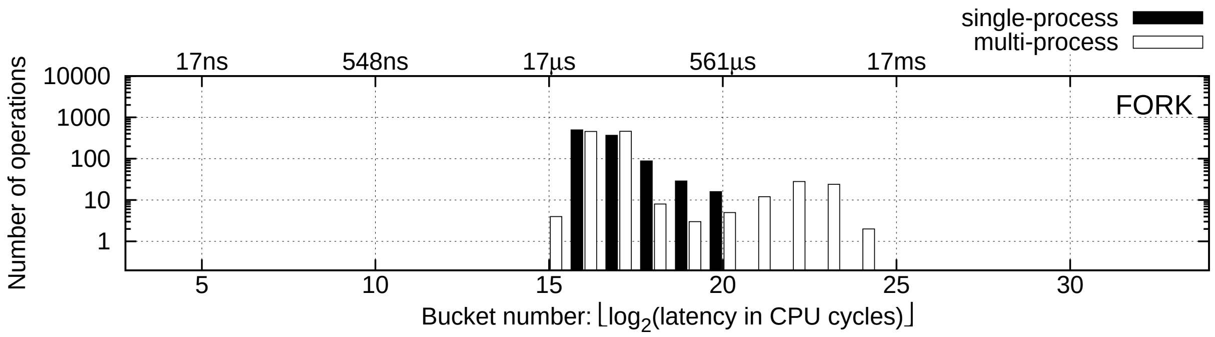


Figure 1.1: Profiles of FreeBSD 6.0 `fork` operations with single-process (black bars) and multi-process (white bars) workloads.

can be observed from these high-level profiles. We believe that DARC can help users take full advantage of OSprof.

We designed the Dynamic Analysis of Root Causes (DARC) system to remedy the problem of manual profile analysis [74]. DARC dynamically instruments running code to find the functions that are the main latency contributors to a given peak in a given profile. We call these functions *root causes*. DARC's output is the call-paths that begin with the function being analyzed, and consist of root cause functions. This provides the user with the exact sequence of functions that were responsible for the peak of interest.

DARC can narrow down root causes to basic blocks and can analyze recursive code as well as code containing indirect functions. If the root cause of a peak is a preemptive event, DARC can determine the type of event (a disk interrupt, for example). DARC can also analyze asynchronous paths in the context of the main process. Although DARC generally does not require much time to perform its analysis, DARC may not be able to fully analyze programs with short runtimes, and longer running programs with short phases that are of interest. To solve these issues, DARC can resume its analysis from a previous point. The program can be run again, and the analysis continues from the previous point. An OSprof profile from the previous run can optionally be automatically compared to a profile from the current run to ensure that the runtime environment has not changed significantly. To minimize false positives and reduce overheads, DARC performs both process ID (PID) and call-path filtering. PID filtering ensures that only calls made by a specific process or thread group are analyzed. Call-path filtering ensures that DARC analyzes only calls which originate from the function of interest and proceed through root cause functions.

We implemented DARC and present several use cases that show the advantages of automatic root cause analysis. Not only is DARC's analysis faster than manual analysis, but it also provides more definitive explanations than those obtained from manual analysis while requiring less expertise and intuition from the user.

Our current DARC implementation can report root cause paths that reside in user-space, in the kernel, and those that originate in user-space and terminate in the kernel. This allows users to analyze behaviors that are observed in user-space but whose root cause lies in the kernel. In addition to running in user-space and in the kernel, we also discuss the possibility of using OSprof and DARC in virtual machine environments.

We measured DARC's overheads and show that they are acceptable for normal usage. Although DARC can make fast memory-bound operations run up to 50% slower, the analysis can be completed quickly, resulting in a negligible effect on overall elapsed time. Further, our instrumentation adds no noticeable overhead on slower I/O-bound operations.

To aid in reproducing our results [75], we have made DARC's source code available, as well as a detailed description of our experimental testbeds and our benchmark results at
`http://www.fsl.cs.sunysb.edu/docs/darc/`.

1.1 Dissertation Organization

The remainder of the dissertation is organized as follows. We describe OSprof in Chapter 2. We detail our design in Chapter 3 and our implementation in Chapter 4. We discuss DARC's limitations in Chapter 5. Chapter 6 describes the test machine that we used for all experiments. Chapter 7 shows examples of how DARC finds root causes. We evaluate the performance of DARC in Chapter 8.

Chapter 9 discusses various methods for comparing OSprof profiles. We describe how OSprof and DARC behave in virtual machine environments in Chapter 10. We discuss related work in Chapter 11. We conclude and discuss future work in Chapter 12.

Chapter 2

OSprof

OSprof [27–30] is a powerful profiling methodology. Latencies for a specified function are measured using the CPU cycle counter (TSC on x86) and presented in histogram form. OSprof measures latency using CPU cycles because it is a highly precise and efficient metric available at runtime. Figure 2.1 shows an actual profile of the FreeBSD 6.0 `fork` operation. The `fork` operation was called concurrently by one process (black bars) and by four processes (white bars) on a dual-CPU SMP system. The operation name is shown in the top right corner of the profile. The lower x-axis shows the bin (or bucket) numbers, which are calculated as the logarithm of the latency in CPU cycles. The y-axis shows the number of operations whose latency falls into a given bin. Note that both axes are logarithmic. For reference, the labels above the profile give the bins' average latency in seconds. In Figure 2.1, the two peaks in the multi-process histogram correspond to two paths of the fork operation: (1) the left peak corresponds to a path without lock contention, and (2) the right peak corresponds to a path with a lock contention. The methods used to reach this conclusion are described later in this chapter.

The relative simplicity of the profiling code makes OSprof highly portable. It has been used to find and diagnose interesting problems on Linux, FreeBSD, and Windows XP, and has been used to profile from user-space and at several kernel instrumentation points. OSprof can be used for gray-box OS profiling. For example, binary instrumentation was used to instrument Windows XP system calls. The latency distributions of these system calls included information about the Windows

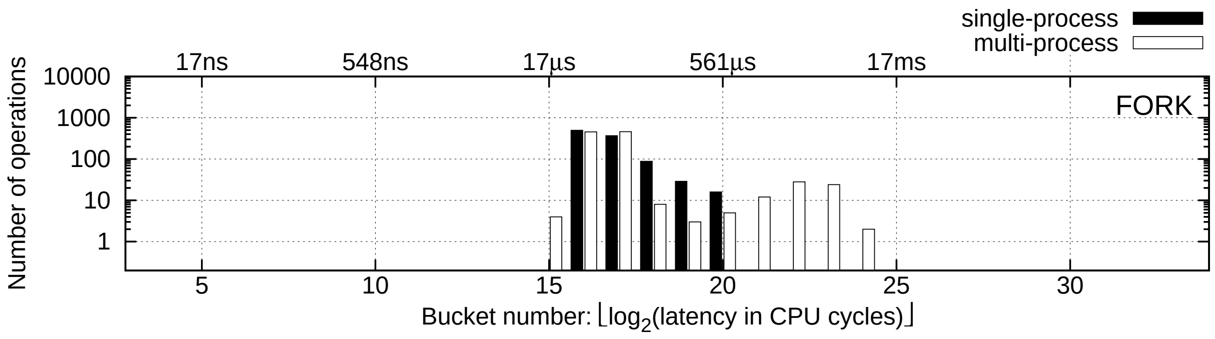


Figure 2.1: Profiles of FreeBSD 6.0 `fork` operations with single-process (black bars) and multi-process (white bars) workloads. This figure is a duplicate of Figure 1.1, and is recreated here for convenience.

kernel. OSprof is also versatile: it can profile CPU time, I/O, locks, semaphores, interrupts, the scheduler, and networking protocols.

OSprof has negligible performance overheads. Its small profiles and code size minimize the effects on caches. Additionally, having small profiles enables OSprof to collect time-lapse profiles, where a separate profile is used for each time segment. This allows the user to see how latency distributions change over time. The performance overhead for profiling an operation is approximately 40 cycles per call. This is much faster than most measured functions, especially since OSprof is generally used to profile high-level functions.

The drawback of OSprof is the manual investigation required to find the root cause of a particular behavior, which is seen as a peak in a profile. The investigation typically requires some deep understanding of the code, as well as taking the time to profile more sections of code. Let us consider the profile shown in Figure 2.1 in more detail. In the single-process case, only the left-most peak is present. Therefore, it is reasonable to assume that there is some contention between processes inside of the fork function. In addition to the differential profile analysis technique used here, other techniques have also been used, such as using prior knowledge of latencies, layered profiling, correlating latencies to variables, and profile sampling [28]. We show some examples of these techniques in Chapter 7, where we compare the analysis methods of OSprof with DARC.

Chapter 3

Design

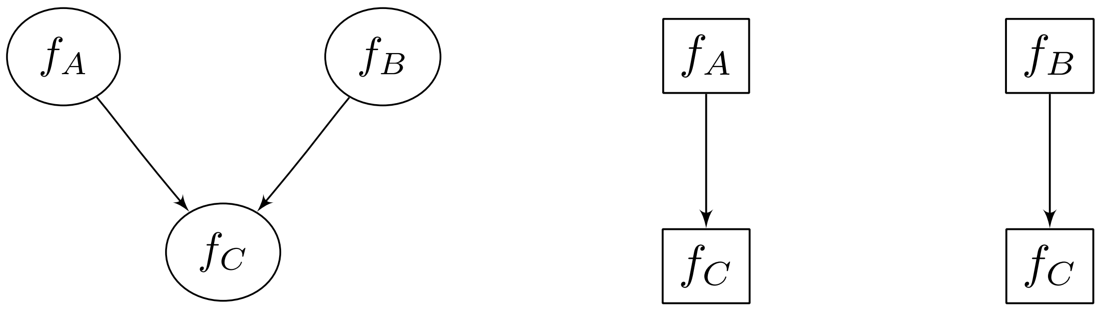
We define a root cause function to be a function that is a major latency contributor to a given peak in an OSprof profile. The key to searching for root causes lies in the fact that latencies are additive: the latency of a function's execution is roughly equal to the latency of executing the function itself, plus the latency to execute its callees. This concept can be extended recursively to the entire call-graph, providing us with an effective method for finding the largest latency contributors. DARC searches the call-graph one level at a time, identifying the main latency contributors at each step, and further searching the sub-trees of those functions.

When starting DARC, the user specifies the process ID (PID) of the target program, the function to begin analyzing (we refer to this as f_0), and the maximum search depth. We call a path from f_0 to a root cause a *root cause path*. DARC's goal is to find root cause paths and present them to the user.

3.1 The Function Tree

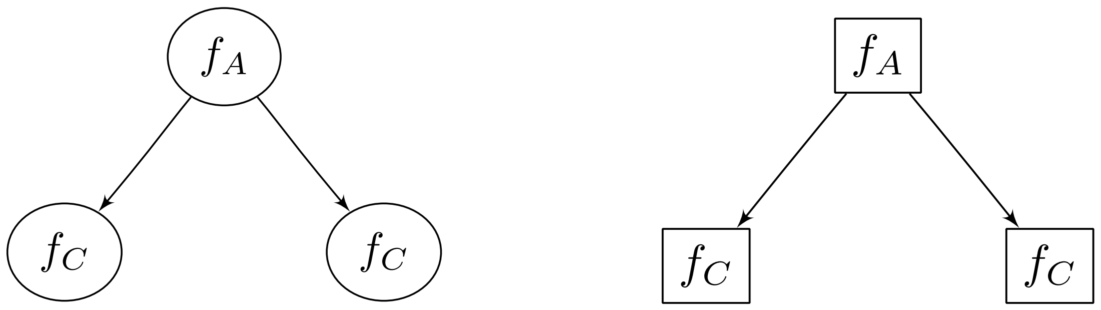
Over time, DARC creates an in-memory tree that represents the function calls along root cause paths. We call this the Function Tree, or *ftree*, and it is composed of *fnode*s. Initially, there is a single fnode in the tree, representing calls to f_0. The depth of the ftree increases during DARC's analysis, until either the specified maximum depth is reached, or DARC has finished its analysis. The PID is used to ensure that only function calls that are invoked on behalf of the given process or thread group are analyzed.

It is important to note that fnodes do not represent functions, but rather function calls in the context of call-paths. For example, we can see in Figure 3.1(a) that both f_A and f_B call f_C. In this case, there would be two nodes for f_C, as shown in Figure 3.1(b). This concept also holds for situations where one function calls a different function twice. In this case, there will be one node for each call site, as shown in Figure 3.2. The ftree is a proper tree, as it contains only sequences of function calls, and so it does not contain loops or nodes with more than one parent. The ftree grows as DARC finds more functions that belong to root cause paths.



(a) call-graph (b) ftree

Figure 3.1: An example of a call-graph (left) with a possible corresponding ftree (right), where a function appears in the ftree twice.



(a) call-graph (b) ftree

Figure 3.2: An example of a call-graph (left) with a possible corresponding ftree (right), where a function calls a different function twice.

3.2 Initial f_0 Instrumentation

DARC begins by instrumenting f_0 with OSprof code, as shown in Figure 3.3. In our notation, the callees of f_0 are $f_{0,0}$ to $f_{0,n}$. The ellipses represent any code present in the original program that is not relevant to our discussion. The `GET_CYCLES` function reads the current value of the register which contains the current number of clock ticks on the CPU (e.g., `RDTSC` on x86). These notations are also used for Figures 3.5, 3.7, and 3.11.

The instrumentation accumulates profile data, which is displayed to the user upon request. DARC examines changes between bins in the histogram to identify peaks, and displays the peak numbers to the user along with the histogram. The peak analysis takes the logarithmic values of the y-axis into account, mimicking the way a human might identify peaks. The output that the user sees is presented in Figure 3.4. Here we can see the address of f_0, the total number of times f_0 was called, the total latency of the profiled calls to f_0, the profile in array form, and a visual representation of the profile. The peak numbers are shown under the bin numbers.

At this point, the user may communicate the desired peak to DARC. DARC then translates the peak into a range of bins. If the desired peak is known ahead of time, the user may specify the peak number and the number of times f_0 should be called before translating the peak into a bin range. Once f_0 is called that number of times, the profile is displayed so that the user may see the profile that the results will be based on.

```
f0 {
    time1 = GET_CYCLES();
    ...
    f0,0();
    ...
    f0,i();
    ...
    f0,n();
    ...
    time2 = GET_CYCLES();
    latency = time2 - time1;
    record_latency_in_histogram(latency);
}
```

Figure 3.3: The instrumentation DARC adds to f_0 when DARC is started. $f_{0,0}$, $f_{0,i}$, and $f_{0,n}$ are functions that f_0 calls.

3.3 Main f_0 Instrumentation

Once a peak is chosen, the original instrumentation is replaced by the instrumentation shown in Figure 3.5. When f_0 is executed, DARC measures the latencies of f_0 and its callees. The maximum latency for each function is stored in the appropriate fnode. The maximum is used because a function may be called more than once in the case of loops (this is explained further later in this section). Because the latencies of the callees are measured from within f_0, the latency stored in the fnode of $f_{0,i}$ is guaranteed to be the latency of $f_{0,i}$ when called by f_0. The latencies are processed only if the latency of f_0 lies in the range of the peak being analyzed. Otherwise, they are discarded. Note that in Figure 3.5, the *start* and *latency* variables in the fnode are thread-local to support multi-threaded workloads.

The goal of the `process_latencies` function (see Figure 3.5) is to find the largest latency contributors among f_0 and its callees. The `process_latencies` function first approximates the latency of f_0 itself:

$$latency_{f_0} - (\sum_{i=0}^{n} latency_{f_{0,i}}) \tag{3.1}$$

It then finds the maximum latency among f_0 and its callees. The largest latency contributors are those whose latency has the same logarithm as the maximum latency. In other words, f_0 is chosen if $log_2(latency_{f_0}) = log_2(latency_{max})$, and an $f_{0,i}$ is chosen if $log_2(latency_{f_{0,i}}) = log_2(latency_{max})$.

To improve accuracy, DARC does not make root cause decisions based on a single call of f_0. Instead, it increments a counter, `maxcount`, in the fnodes of the largest latency contributors. Root cause decisions are made after a user-defined amount of times where the latency of f_0 has been in the range of the peak being analyzed. The main latency contributors are those whose value of `maxcount` are within a user-defined percentage (defined by the `maxcount_percentage` parameter) of the largest `maxcount` value. These functions are root cause functions, and their fnodes are marked as such. DARC always clears the latencies that were recorded before f_0 returns.

```
f0 address: 3222666524
total operations: 12426
total cycles: 33484691285
profile: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 43 11981 266 106 5 0 3
22 0 0 0 0 0 0 0 0

10000   |                  .
1000    |                  .
100     |                  ...
10      |                 ....   .
1       |                 ..... ..
------------------------------------------
bin:      01234567890123456789012345678901
peaks:                    11111 22
```

Figure 3.4: The OSprof profile and peaks that DARC presents to the user.

Case	**Is the condition true?**	**Should f_c be a root cause function?**	**Is f_c the only function that should be a root cause function?**
A	Y	Y	Y
B	Y	Y	N
C	Y	N	-
D	N	Y	Y
E	N	Y	N
F	N	N	-

Table 3.1: Possible scenarios for a function contained within a conditional block.

3.4 Loops and Conditional Blocks

We can now consider the case of a function f_l being called from a loop. In this case, DARC should designate f_l as being the root cause if it has a high latency, regardless of being in the loop. If the latency is due to f_l being called from a loop, then DARC should designate the calling function as the root cause. To accomplish this, DARC uses the maximum latency of f_l, so that it is as if f_l was called once. The latencies for f_l in other iterations of the loop are then automatically attributed to the calling function, as per Equation 3.1 that approximates the latency of f_0.

The case of conditional blocks, such as `if`, `else`, and `case` requires no special handling. We refer to the function that is contained within the conditional as f_c. We have three questions whose answers will affect how DARC handles conditional blocks. First, is the condition true? f_c will be called only if the condition is true. Second, should f_c be a root cause function? In other words, is it a main latency contributor to the specified peak? Third, is f_c the only function that should be a root cause function? In other words, is there another function that should also be considered as a main latency contributor?

We present the possible scenarios in Table 3.1. The cases are handled as follows:

```
f0 {
    root->start = GET_CYCLES();
    ...
    f0,0();
    ...
    c = root->child[i];
    c->start = GET_CYCLES();
    f0,i();
    c->latency = GET_CYCLES() - c->start;
    if (c->latency > c->maxlatency) {
        c->maxlatency = c->latency;
    }
    ...
    f0,n();
    ...
    root->latency = GET_CYCLES() - root->start;
    if (is_in_peak_range(root->latency)) {
        process_latencies();
        num_calls++;
    }
    if (num_calls == decision_calls == 0) {
        choose_root_causes();
        num_calls = 0;
    }
    reset_latencies();
}
```

Figure 3.5: The instrumentation DARC adds to f_0 after a peak is chosen. The instrumentation for $f_{0,0}$ and $f_{0,n}$ is similar to that of $f_{0,i}$, and was elided to conserve space.

A Here, f_c's `maxcount` value will be incremented, as expected.

B The `maxcount` value for f_c will be incremented, along with potentially any other function that should be chosen as a root cause function during this round of analysis, as expected.

C The latency of f_c will be measured, but because f_c should not be a root cause function, its `maxcount` value will not be incremented.

D As f_c is the only function that should be chosen to be a root cause function during the current round of analysis, the latency of f_0 should not be within the range of the desired peak if f_c is not called.

E Although f_c should be chosen as a root cause function, its latency was not measured during this iteration because the condition was false. Because f_c is not the only function that should be chosen as a root cause function, it is possible that the `maxcount` value of some other functions were incremented, and therefore f_c may ultimately not be selected to be a root cause function. The user must adjust the `maxcount_percentage` parameter if f_c is to be chosen—lower values will allow functions with lower `maxcount` values to be identified as root cause functions. However, the root cause functions that *were* chosen will have had a bigger impact, and so the omission of f_c is acceptable.

F The latency of f_c will not be measured, and consequently it will not be a candidate to have its `maxcount` value incremented. This is acceptable, as f_c should not be a root cause function.

3.5 Left Shift

Before describing how descendants of f_0 that have been marked as root cause functions are instrumented, the concept of *left shift* must be introduced. As DARC descends deeper into the code, the peak being analyzed shifts to the left. To understand why this occurs, assume the peak is in bin N of f_0's profile. Further, the main latency contributor for this peak is $f_{0,i}$. The peak, as seen in the profile of f_0, includes the latency for f_0 itself, as well as the latencies for the other functions that f_0 calls. However, the peak in $f_{0,i}$ does not contain these additional latencies, and so the peak may shift to the left in the profile of $f_{0,i}$.

We can see the effects of left shift in Figure 3.6. Here we repeatedly called the `stat` system call on a single file and captured OSprof profiles of the root cause functions, beginning with the top-level stat function. The appropriate Linux kernel functions were manually instrumented to collect these profiles. The profiles for the call-path are presented in descending order, with the top-level function,`vfs_stat`, on top. The machine that this was run on is described in Chapter 6. We can see in the figure that the vast majority of operations fall into bin 11 in the `vfs_stat` profile, and this progressively shifts to bin 10 as we move to lower-level functions.

3.6 Lower Function Instrumentation

To avoid the effects of left shift, DARC does not calculate the location of the peak in $f_{0,i}$. Instead, DARC keeps the decision logic in f_0. Root cause functions other than f_0 are instrumented as shown in Figure 3.7. Assume f_0 calls functions $f_{0,0}$ to $f_{0,n}$, and $f_{0,i}$ is chosen as a root cause. Further, $f_{0,i}$ calls $f_{0,i,0}$ to $f_{0,i,m}$. In $f_{0,i}$, latencies for each $f_{0,i,j}$ are calculated, but not processed. DARC does not add instrumentation to measure the latency of $f_{0,i}$ because it is measured in f_0. DARC creates fnodes for each $f_{0,i,j}$, with $f_{0,i}$ as the parent.

Each new root cause function ($f_{0,i}$ in our example) is added to a list of nodes that f_0 processes before returning. Before returning, if f_0's latency is within the peak, DARC traverses this list to process latencies, and possibly chooses the next round of root cause functions. Placing the latency information on a queue to be processed off-line by a separate thread may minimize the impact of the decision code on the latency of f_0. We evaluate this design decision in Chapter 8. DARC removes instrumentation that is no longer needed using a second lower-priority queue for the instrumentation removal requests. This is because removing instrumentation is a performance optimization and can be a slow operation, and so the delays on the analysis should be minimized.

When DARC determines that a function (and not any of the functions that it calls) is responsible for the specified peak, DARC stops exploring along that call-path. DARC also stops exploring

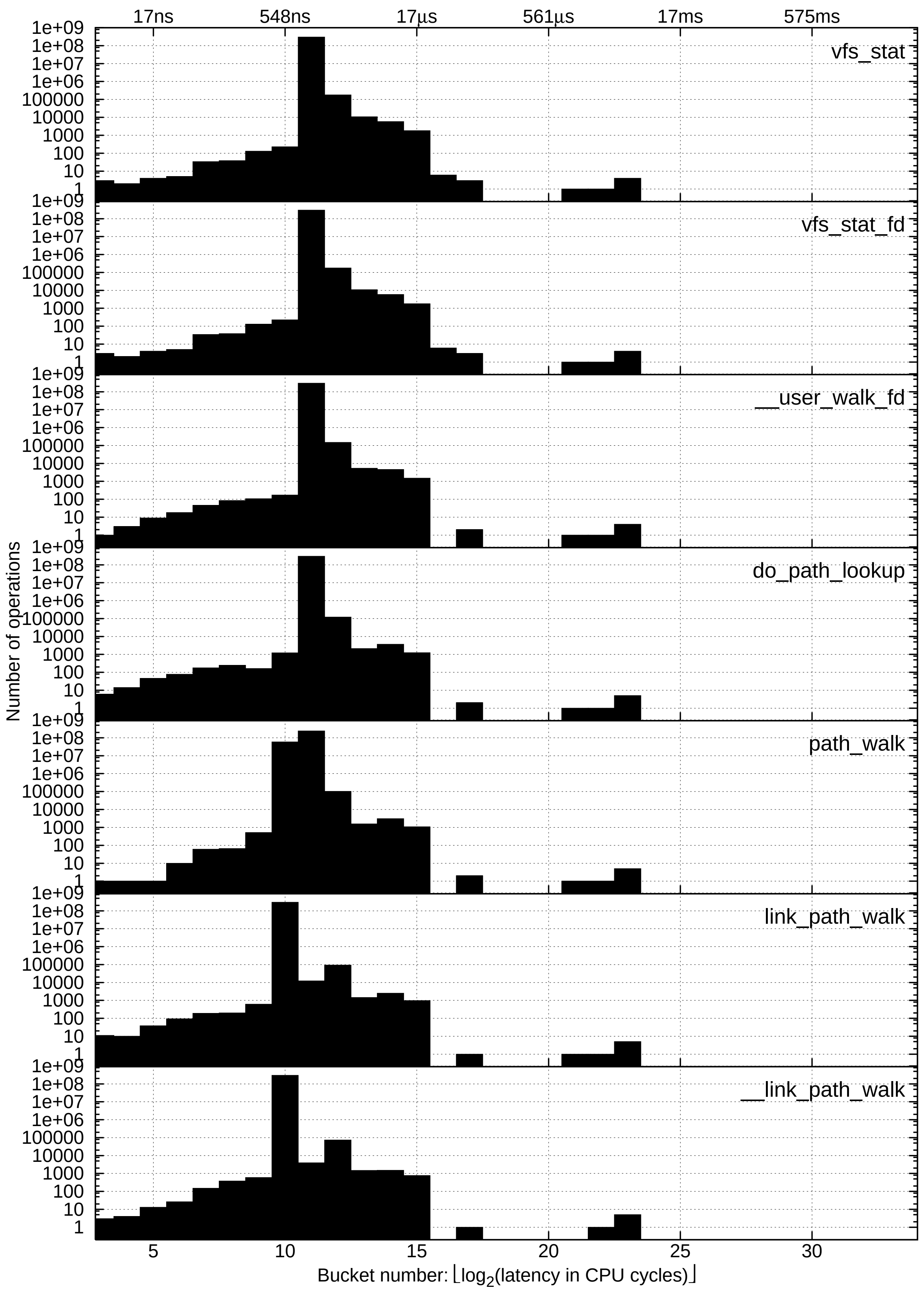


Figure 3.6: OSprof profiles for the root cause path functions under the stat workload. Each function whose profile is shown here calls the function whose profile is shown below it.

```
f0,i {
    ...
    c = parent->child[j];
    c->start = GET_CYCLES();
    f0,i,j();
    c->latency = GET_CYCLES() - c->start;
    if (c->latency > c->maxlatency) {
        c->maxlatency = c->latency;
    }
    ...
}
```

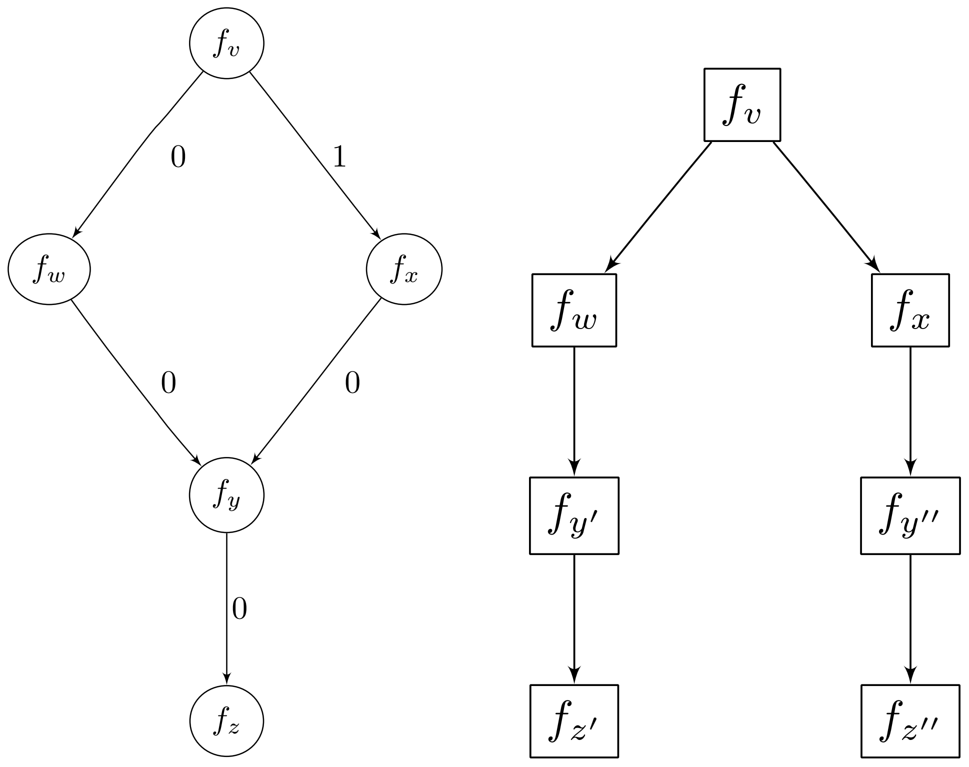
Figure 3.7: The instrumentation DARC adds to $f_{0,i}$. Function $f_{0,i,j}$ is a function that $f_{0,i}$ calls.

a call-path if it reaches a function that does not call any functions, or after the root cause path has grown to the user-specified length. When all call-paths have completed, DARC removes all remaining instrumentation and the program continues to run as normal. DARC's status may be queried by the user at any time. This status includes the latency histogram for f_0, the analysis status ("in progress," "maximum depth reached," or "root cause found"), the ftree, and the `maxcount` values for each fnode.

3.7 Tracking Function Nodes

Before instrumenting a function, DARC must check if the function has already been instrumented to avoid duplicating instrumentation. An example of how this could occur is shown in Figure 3.8. Here the latency of f_y is measured from f_w and f_x. DARC then determines that f_y is a root cause of both paths. Because f_y was chosen as a root cause twice, it would be instrumented twice to measure the latency of f_z. We avoid this by using a hash table to track which functions have been instrumented. The first time DARC tries to instrument f_y, it searches the hash table using the address of f_y as the key. It is not found, and so f_y is instrumented, and an entry is inserted into the hash table. Before DARC tries to instrument f_y a second time, it searches the hash table, finds the entry for f_y, and therefore does not instrument it a second time. A hash table is used because instrumented functions cannot be tracked by marking fnodes, because there may be multiple fnodes for a single function.

When more than one fnode exists for each instrumentation point, the fnode cannot be tied to the instrumentation. For example, there are two fnodes for f_y, so f_y's instrumentation cannot always use the same fnode. To solve this, DARC decouples the fnode tracking from the instrumentation by using a global (thread-local) fnode pointer, `current_fnode`, which points to the current fnode. This pointer is always set to f_0 at the start of f_0. Each instrumented function sets the `current_fnode` pointer by moving it to a specific child of the fnode that `current_fnode` is pointing to. It does so using the fnode identifiers (see labels on the call-graph edges in Figure 3.8(a)). These fnode identifiers are simply an enumeration of the callees of the parent function. In addition, each fnode contains a thread-local `saved_fnode` pointer, where the value of the



(a) call-graph (b) ftree

Figure 3.8: An example of a call-graph (left) with a possible corresponding ftree (right) that requires a hash table to avoid duplicate instrumentation. Labels on the edges of the call-graph are fnode identifiers, which are enumerations of each fnode's children. All nodes belong to root cause paths.

global pointer is saved so that it can be restored after the function call. In Figure 3.8, f_y's instrumentation will save `current_fnode`, and then change it to point to the first child of the current fnode. This will cause `current_fnode` to point to the correct f_y fnode regardless of whether it was called via f_w or f_x.

3.8 Filtering

DARC performs two types of filtering to ensure that only relevant latencies are measured and analyzed. First, process ID (PID) filtering ensures that only function calls that are called in the context of the target process or thread group are analyzed. This is important for functions that reside in shared libraries or the operating system. Second, it performs *call-path filtering*. It is possible for functions that are not part of a root cause path to call a function that DARC has instrumented. In this case, latency measurements should not be taken, because they may reduce the accuracy of the analysis. For example, lower-level functions are generally called from several call-paths, as the functions tend to be more generic. Performing this filtering can increase the accuracy of DARC's analysis by reducing noise in the captured latencies. Call-path filtering also ensures that no function that is called from outside of the root cause paths will modify the `current_fnode` pointer.

Figure 3.9 demonstrates the need for call-path filtering. Here we used the `grep` utility to search

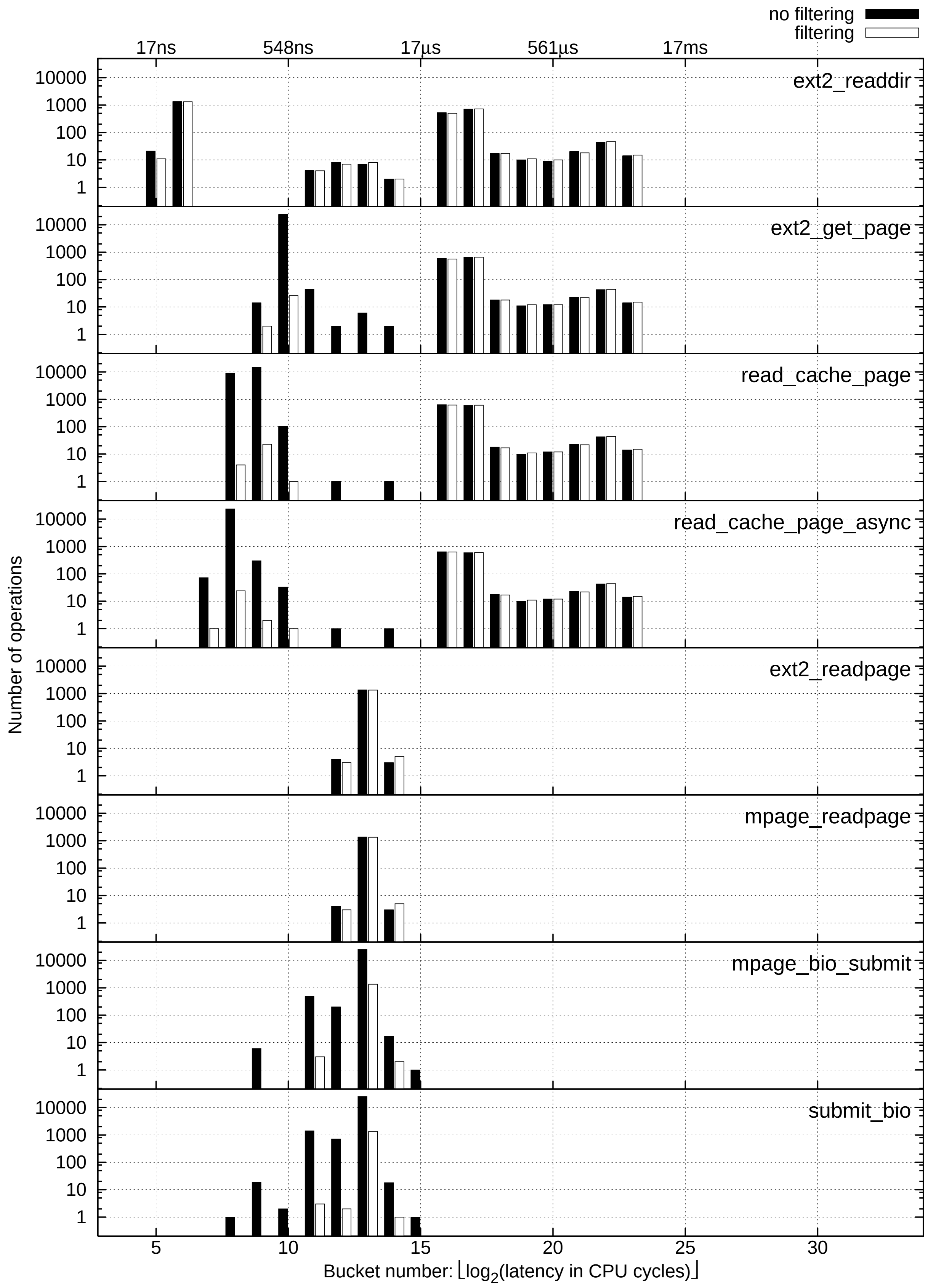


Figure 3.9: OSprof profiles for the root cause path functions under the grep workload, with filtering (white bars) and without filtering (black bars).

the Linux kernel sources recursively for a non-existent string. This workload is further described in Chapters 6 and 7. We manually instrumented the appropriate Linux kernel functions to collect these profiles. To add filtering, we added an extra parameter to the functions. The added parameter was off by default, and was turned on only along the root cause path. Profiling was enabled only if the added parameter was set. In Figure 3.9 we can see the root cause path that begins with the `ext2_readdir` function, which reads a directory listing. We can see that bin 16–23 are nearly identical in the first four profiles (`ext2_readdir`, `ext2_get_page`, `read_cache_page`, `read_cache_page_async`) for both the filtered and unfiltered cases. However, the left-most bin look quite different when filtering is enabled. Further, the `ext2_readpage` and `mpage_readpage` functions look similar for the filtered and unfiltered cases. This is because these functions are called mostly for reading metadata; data is read using the corresponding `readpages` functions rather than the `readpage` functions, which perform readahead. The final two functions, `mpage_bio_submit` and `submit_bio`, show large differences between the filtered and unfiltered versions. This is because these functions are low-level, generic block I/O functions, and are used to perform both data and metadata I/O.

DARC uses an efficient and portable call-path–filtering technique. Each fnode contains a thread-local flag to specify that it was called along a root cause path. The flag in f_0's fnode is always set. Before a root cause function calls another root cause function, it sets the flag of its callee's fnode if its own flag is set. The latency measurements and analysis are only executed when the flag of the current fnode is set.

Others have used a relatively expensive stack walk to perform call-path filtering [6]. Although it has been shown that a full stack walk is not necessary [22], a stack walk is highly architecture and compiler-dependent. Our method is more portable, and we can prove its correctness. We first show inductively that if a function is called as part of a root cause path, then the flag in its corresponding fnode that indicates this fact is set.

Proof: Base case: f_0 is always the start of a root cause path. The flag for f_0 is always set by definition.

Inductive step: Assume that the flag is set for the first i functions of a root cause path. The flag for function $(i + 1)$ will be set by i because the flag for function i is set, and $(i + 1)$ is a root cause function.

We now show that if a function is not called as part of a root cause path, then its flag will not be set.

Proof: Consider a call-path P, which is composed of three segments, $P = XYZ$. The functions in X and Z are root cause functions, while those in Y are not ($\overline{X} \geq 0$, $\overline{Y} \geq 1$, $\overline{Z} \geq 1$). We show that the flags for functions in Y and Z are not set. We have already shown that functions in X will have their flags set. Before the last function in X, X_{last}, calls the first function in Y, Y_{first}, it checks if Y_{first} is a root cause function. Because it is not, the flag in Y_{first} is not set. Alternatively, if $\overline{X} = 0$, the flag Y_{first} will not be set because no root cause function has set it. Because the flag in Y_{first} is not set, any subsequent function in Y will not have its flag set. Because Y_{last} does not have its flag set, it will not set the flag of Z_{first}, and so no function in Z will have its flag set.

3.9 Profiling Basic Blocks

If the instrumentation method used to implement DARC has knowledge about basic blocks, DARC can instrument these as well. This is useful in two cases. First, when DARC reaches the end of a root cause path, DARC can then proceed to narrow down the root cause to a basic block in that function. DARC acts on basic blocks in the same way as it does on functions: it creates an fnode for each basic block, and sub-blocks are treated as callees of those blocks. When displaying the ftree, DARC reports the type of basic block instead of a function name.

The second case where basic block instrumentation is useful is if a function calls a large number of other functions. Instrumenting all of the functions at once may add too much overhead. The user may specify a threshold for the maximum number of functions to be instrumented at once. If this threshold is about to be exceeded, DARC instruments only those function calls that are not called from a basic block nested within the function, and also instruments any basic block containing a function call. There is no need to instrument basic blocks that do not call functions because their latencies will be automatically attributed to the function itself (recall that the latency of the calling function is estimating by subtracting the latencies of its callees from its latency). After DARC narrows down the root cause to a basic block, it may further instrument that block to continue its analysis.

DARC can be set to always instrument basic blocks before function calls. This reduces the overhead incurred at any given point in time. The trade-off is that because there are more steps to finding a root cause, the period of time in which overheads are incurred is prolonged. In addition, DARC consumes more memory because the ftree contains basic blocks as well as functions.

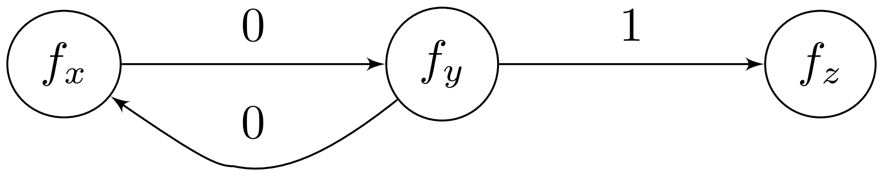
3.10 Resuming DARC

DARC can use its output as input in a future run, allowing it to continue a root cause search without repeating analysis. After parsing the previous output, DARC rebuilds the ftree (including the `maxcount` values), and inserts the appropriate call-path filtering and latency instrumentation. The ability to resume analysis is important in two cases. First, a user may search for root cause paths up to a specified length and later need more information. Second, a program may not run for enough time to fully analyze it, or the user may be analyzing a specific phase of a program's execution. In this case, the program may signal DARC on when to begin and end the analysis.

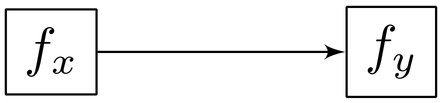
If desired, a new OSprof profile for f_0 can be collected before DARC resumes analysis, and this profile can be compared to the previous profile to ensure that the latency distribution has not changed. A change in the distribution may be caused by factors such as changes in the execution environment or different input to the process. DARC compares the profiles using the Earth Mover's Distance (EMD), which is an algorithm commonly used in data visualization as a goodness-of-fit test [63]. Further discussion on profile comparison methods can be found in Chapter 9.

3.11 Recursion

To handle recursion, the ftree needs to have one fnode for every instance of a function call, as described in Section 3.7. Additionally, DARC needs to know when the code execution goes past a leaf in the ftree and then re-enters it by way of recursion. For example, in Figure 3.10, DARC



(a) call-graph



(b) ftree

Figure 3.10: A recursive call-graph (left) with a corresponding ftree (right), where only f_x has been identified as a root cause at this point. The numbers on the edges of the call-graph are the fnode identifiers.

must know that after f_y calls a function, it is no longer in the ftree. This is because f_y may call f_x, which would incorrectly set `current_fnode` to f_x. To solve this, DARC has a thread-local flag that tracks when the execution leaves the ftree. In this example, the instrumentation would look like the code in Figure 3.11. DARC uses the `recursion_count` variable to ensure that the same function execution that set `in_tree` to `false` also sets it to `true`. This is needed to prevent the second execution of f_x from setting `in_tree` to `true`, whereas the first execution of f_x set it to `false`.

3.12 Analyzing Preemptive Behavior

Preemptive behavior refers to any case where the primary thread that is being investigated is stopped and another piece of code is set to run. This can be when the main process is preempted for another process to run or when an interrupt occurs. Preemptive behavior may concern us when the latency of a secondary code path is incorporated into the latencies that DARC measures, although in general these latencies may be ignored [28]. The original OSprof methodology used system-specific knowledge about the quantum length to determine when the process was preempted, and intuitive guesses to determine when interrupts caused peaks.

DARC measures preemptive behavior only if the added latency will be incorporated into the current latency measurements. This happens if the code being preempted in the primary thread is in the subtree of an fnode that is currently being investigated. These fnodes contain extra variables to store preemption and interrupt latencies. In cases where multiple preemptive events of the same type occur, DARC stores the sum of their latencies (recall that DARC resets latency information after each execution of f_0).

If the name or address of the appropriate scheduler function is available, DARC can instrument it to check if the target process was preempted, and for how long. DARC stores the total amount of time spent while preempted in the appropriate fnode, and uses this data when searching for root causes. If preemption was the main factor in the peak, DARC reports "preemption" as the cause.

```
fx {
    ...
    if (in_tree) {
        current_fnode = fnode(fx);
        // latency measurement code for fy
        fnode(fx)->recursion_count++;
        in_tree = false;
    }
    fy();
    fnode(fx)->recursion_count--;
    if (fnode(fx)->recursion_count == 0) {
        in_tree = true;
        // latency measurement code for fy
        current_fnode = saved_global;
    }
    ...
}
```

Figure 3.11: The instrumentation DARC adds to f_x around the call to f_y in the example shown in Figure 3.10 to handle recursion.

For interrupts, if the name or address of the main interrupt routine is known, DARC instruments it to record the latencies in an array contained in the proper fnode that is indexed by the interrupt number. Latencies are only recorded if the target process was executing in the subtree of a function being analyzed. In addition, DARC keeps a small auxiliary array to handle the case where an interrupt occurs while processing an interrupt. If an interrupt is determined to be a root cause, DARC reports the interrupt number and handler routine name.

3.13 Analyzing Asynchronous Paths

An asynchronous path refers to a secondary thread that acts upon some shared object. Examples of this are a thread that routinely examines a system's data cache and writes modified segments to disk, or a thread that takes I/O requests from a queue and services them. Asynchronous paths are not uncommon, and it may be desirable to analyze the behavior of these paths. Work done by asynchronous threads will generally not appear in a latency histogram, unless the target process waits for a request to be completed (forcing synchronous behavior). An example of such behavior can be seen in the Linux kernel, where a request to read data is placed on a queue, and a separate thread processes the request. Because the data must be returned to the application, the main thread waits for the request's completion.

To analyze asynchronous paths, the user may choose a function on the asynchronous path to be f_0. This requires no extra information, other than the name or address of f_0. However, if it is desirable to analyze an asynchronous path in the context of a main path, DARC requires extra information. For cases with a request queue, DARC needs to know the address of the request structure. DARC adds call-path filtering along the main path up to the point where the request structure is available to DARC. At this point, DARC adds the request structure's address to a hash table if the PID and call-path filtering checks all pass. When the secondary thread uses the request object, DARC checks the hash table for the object's address. If it is there, DARC knows that the target process enqueued the object along the call-path that is of interest to the user.

In the case where the asynchronous thread is scanning all objects (with no request queue), the object can be added to the hash table when appropriate. This technique requires the same extra knowledge as the situation with a request queue: the call-path to filter and the name of the request object. In the case where this information is not available, DARC proceeds without PID or call-path filtering.

Chapter 4

Implementation

In implementing DARC, we aimed to make it efficient while being easy and feasible to use in a variety of environments. The user interface is the part of the implementation that the user directly sees. We describe the user interface that we implemented in Section 4.1.

With regard to instrumentation, we investigated two options: compile-time source code modification and dynamic binary instrumentation (DBI). We selected the latter for several reasons, including the ability to instrument without recompiling and restarting the application, smaller total code size, reduced overheads for inactive instrumentation, and handling of indirect calls. We discuss these reasons further in Section 4.2.

Our initial version of DARC only supported analysis of Linux kernel functions [74]. This decision was dictated by two facts. First, the Linux kernel provides kprobes, a robust mechanism for binary instrumentation of its code [17, 43]. Second, the ability to dynamically add modules to the kernel allows us to easily have the instrumented code and the instrumentation in the same address space. This implementation is detailed in Section 4.2.1. However, the kprobes mechanism is too rich for our specific needs, and therefore too expensive in terms of instrumentation overheads. This realization brought us to reduce the overheads by implementing several optimizations that we detail in Section 4.2.2.

DARC's unique features can be helpful not only in analyzing operating system kernels, but user-space applications as well. Our DARC implementation allows users to find root cause paths that reside in user-space programs, in the Linux kernel, or those that cross the user-kernel boundary. We extended DARC so that the user may specify the start function, f_0, to be any function in a user application or the kernel. If f_0 is a user-space function, the root cause search may potentially continue into the kernel via the system call interface. This would help users to determine if the root cause of a behavior that is observed in user-space is actually caused by the kernel. Figure 4.1 shows an example of an ftree that crosses the user-kernel boundary. We discuss this further in Section 4.2.4.

Two major problems arise when extending DARC to handle user-space applications. First, it is usually difficult to find a place in the application's address space to store DARC's instrumentation safely while the program is running. This means that we may need to store the instrumentation in the address space of some other process or the kernel. In the case of another process, a context switch is unavoidable every time DARC's instrumentation is executed, which can cause excessive overheads. Therefore, we chose to store the instrumentation in kernel space. This also allowed us to analyze root cause paths that cross the user-kernel boundary transparently: no data synchronization

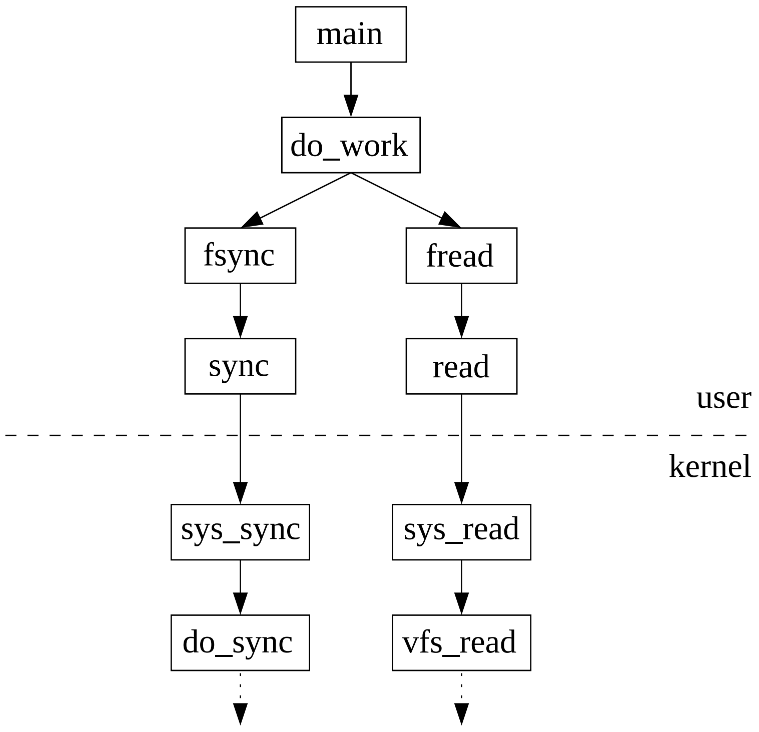


Figure 4.1: An example of an ftree that crosses the user-kernel boundary. Only root cause fnodes are shown.

is required between the user-space and kernel portions of the function tree. The second problem in implementing DARC for user-space applications is the lack of reliable in-kernel mechanisms for binary instrumentation of user-space applications. Several developers have attempted to create solutions that are similar to kprobes, but none were accepted by the Linux kernel community [33, 34, 56, 57]. We selected one of these solutions [57] and simplified it with the hope that it will increase reliability. The reduced framework is powerful enough for DARC's purposes, but does not include any additional functionality.

The extended DARC implementation with support for user applications is presented in Section 4.2.3.

4.1 Interaction

The first step to start using DARC is to insert the kernel module into the running kernel via the `insmod` command. When loaded, the module creates the `/proc/darc` file which provides the basic interface between the user and DARC. Reading from this file provides information about the progress of analysis, while performing `ioctl` operations on it controls DARC. To facilitate the control process, DARC includes a user-level program, called `darcctl`, that interacts with the kernel component using `ioctl`s.

After the module is loaded, the user invokes `darcctl` with parameters that describe the analysis to be performed. Because the list of parameters can be rather long, DARC includes a script that takes a configuration file as input, translates some parameters, loads the module and calls `darcctl` with the appropriate arguments. A simple configuration file can be seen in Figure 4.2.

If a program name is given for the `program_name` parameter, the script translates it to a PID before passing it to the DARC kernel module. This option can only be used if the name maps to a single PID—otherwise, a PID must be passed to DARC. Similarly, the `start_function` parameter is translated to a code address. If the function is located in a user application, then the `binary` parameter should be specified. In this case, the script extracts symbol information from the binary and passes it to the module. Section 4.2.3 explains why symbol information is required by DARC. The script also passes the address of the system call table to the module if f_0 is located in a user application. DARC uses this address to determine which system call was called if the user-kernel boundary was crossed.

Our DARC implementation is designed to be used in two modes: manual and automatic. In the first case, after DARC is started, the user manually selects the peak of interest and when DARC should select the next round of root cause functions. Alternatively, the user may specify the `decision_time`, `start_ops`, and `start_peak` parameters in the configuration file at the very beginning; if so, DARC performs all these actions automatically. Manual mode is preferred when a user is first starting to analyze a latency distribution, while automatic is better when working with familiar distributions and for reproducing the results at a later time.

The script and configuration file make DARC easier to use, because the user does not need to look up the PID and code address manually. In addition, the user may save configuration files for future use or reference. Internally, DARC does not use any function names, so its output contains only function addresses. DARC includes a user-level script to process the output, translating addresses to function names and interrupt numbers to interrupt names.

4.2 Instrumentation

DARC operates by using dynamic binary instrumentation (DBI) to find the root causes of a peak. An alternative to DBI would be a compile-time method, where all of the instrumentation is added to the source code [12, 31]. A compile-time method has the benefits of lower overheads to activate instrumentation and the ability to report the names of inline functions. However, there are five main drawbacks to compile-time methods. First, the build system would need to be changed to add the code. In large projects, this may be a daunting task. Second, the application would need to be stopped to run the version with the new instrumentation. This is a problem for critical or long-running applications. Third, because all instrumentation must be inserted ahead of time, there can be a large increase in code size, and all code paths would incur overheads even when skipping over the instrumentation. Fourth, all source code needs to be available. Although application code is usually available to its developers, libraries and kernel code may not be. Finally, indirect calls can only be resolved at runtime, and so not all of the functions that require instrumentation can be known at compile time.

4.2.1 Kernel-Level Instrumentation

The current DARC prototype is implemented as a kernel module for the Linux 2.6.23 kernel. Excluding the optimizations discussed in Section 4.2.2, it uses kprobes for DBI [17]. Simply put, a *kprobe* replaces a given instruction with the following sequence: a call to an optional *pre-handler*

```
# Name or PID of program to analyze.
program_name = grep

# Name of function to start analyzing.
start_function = vfs_readdir

# Number of function calls (within the latency range) between
# decisions.  Raising this value increases the number of function
# calls that decisions are based on, possibly improving accuracy,
# but also increasing the analysis time.
decision_time = 50

# DARC stops its analysis at this depth if not already finished.
max_depth = 10

# Percent closeness to maximum maxcounts values, used to choose which
# functions will be chosen as the next level of root causes.
# Acceptable values range from 0 to 100.  Higher values will produce
# more conservative results, but may omit some less frequently called
# functions.  Lower values will allow for more functions to
# potentially be chosen as root causes, but may also increase the
# amount of false positives.  We recommend starting with a value of at
# least 90, which should be suitable for most situations, and lowering
# it if more root cause options are necessary.
maxcount_percentage = 97

# Smallest allowable latency for a root cause function (specified by
# an OSprof bucket number).  This can be used so that DARC does not
# continue its analysis once it chooses a root cause function whose
# latency is too low to matter.  For example, when analyzing an I/O
# function, buckets representing faster CPU-bound operations may be
# ignored.
min_bucket = 6

# Number of elements to collect in the OSprof histogram before
# choosing a peak (optional).  The peak that will be chosen is
# specified by the ’start_peak’ parameter.  This value should be high
# enough to ensure that the histogram peaks are stable.  These
# parameters are generally used only for automating the analysis
# process while benchmarking.
start_ops = 100

# Peak to choose - see comments for the ’start_ops’ parameter
# for more information (optional).
start_peak = 1
```

Figure 4.2: A simple DARC configuration file.

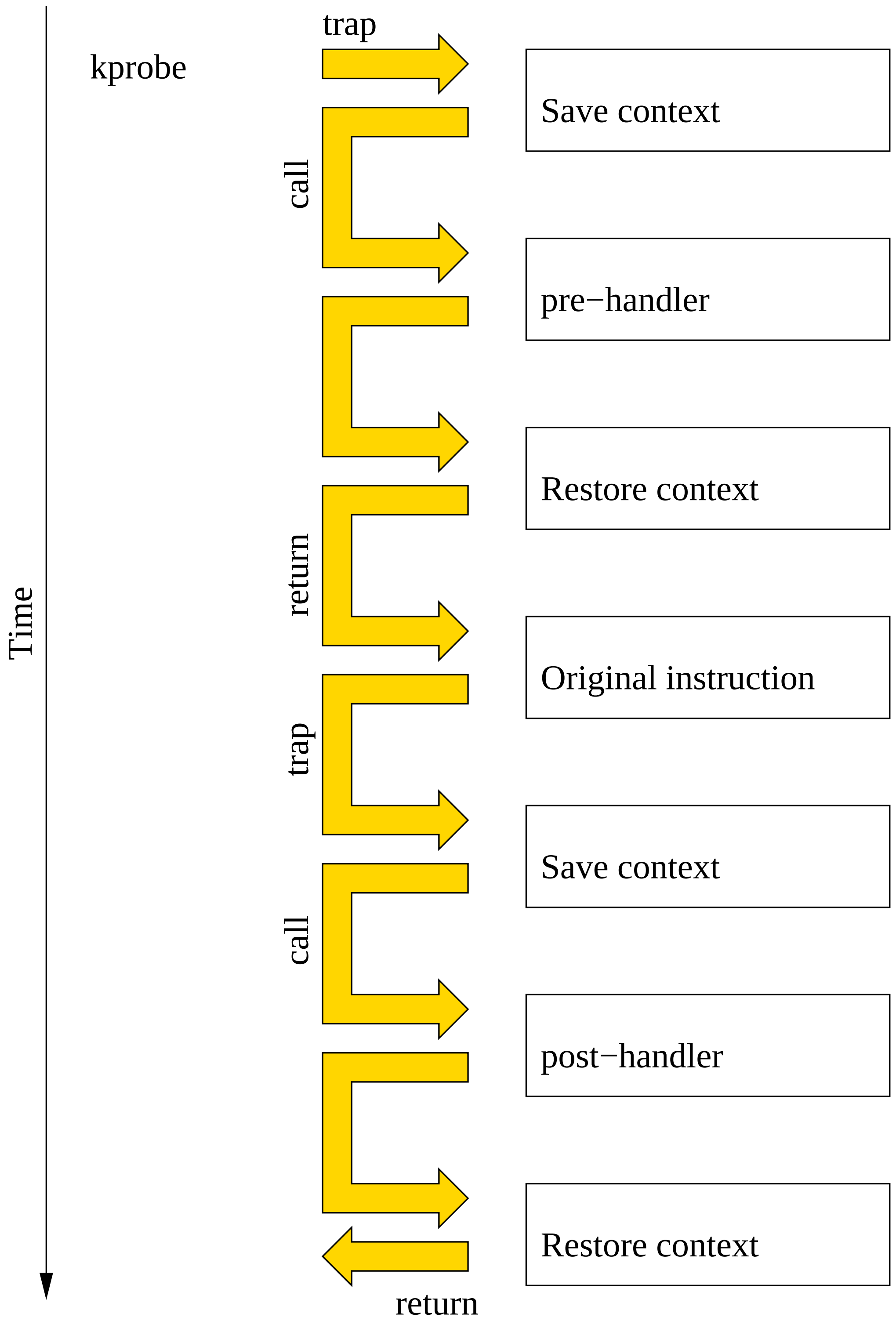


Figure 4.3: The main steps required to execute a kprobe that utilizes both pre-handlers and post-handlers.

function, the instruction itself, and a call to an optional *post-handler* function. The main steps in a kprobe's execution are shown in Figure 4.3. In more detail, when DARC registers a kprobe, the probed instruction is saved, and the instruction is replaced with an interrupt. A trap occurs when this interrupt instruction is executed, the kernel saves the context, and it then calls the kprobes interrupt handler. The handler first disables preemption and saves the flags from the exception context. In the saved context, it disables interrupts, gets ready to single-step or execute the replaced instruction, and calls the pre-handler if it is defined. When returning from the trap, the kernel restores the context and the original instruction is executed. At this point, another trap occurs, and the kernel once again saves the context and calls the interrupt handler. Now the handler calls the post-handler if it is defined, restores the saved flags, fixes program-counter–relative results, sets the saved program counter, and restores preemption. The handler then returns, and the kernel restores the context and continues executing the code from the instruction after the replaced instruction.

Most DARC instrumentation is inserted using ordinary kprobes. However, the instrumentation that is executed before f_0 returns (see end of Figure 3.3) is inserted using a *kretprobe*, or a "return kprobe." This type of kprobe is executed before a function returns from any point. Additionally, to handle function pointers (i.e., indirect calls), DARC adds additional code to the kprobe at the call site that checks the appropriate register for the target address.

We used kprobes for two reasons. First, it is part of the mainline Linux kernel. Code inside the mainline kernel tends to be stable and well-maintained, and is available in any recent kernel version. Kprobes are currently available for the i386, x86_64, ppc64, ia64, and sparc64 architectures and it can be expected that other architectures that Linux supports will be supported by kprobes in the future. The second reason is that kprobes provide a minimalistic interface common to most DBI techniques—they place a given section of code at a given code address. This shows that DARC can be implemented using any DBI mechanism, and can be ported to other operating systems and architectures.

We considered two alternative DBI frameworks for implementing DARC. Kerninst [72, 73], which is available for Sun UltraSparc I/II/III, x86, and IBM PowerPC, was not suitable for us because of its cumbersome API and instability. Kerninst requires that code that is to be inserted be constructed using their API. For example, a simple instruction such as incrementing a variable A, would look like:

```
kapi_arith_expr incr_A(kapi_plus, A, 1);
kapi_arith_expr assign_A(kapi_assign, A, incr_A);
```

In addition to the API problems, Kerninst failed to instrument some kernel functions that we tested, resulting in kernel crashes.

The second DBI framework that we considered was PinOS [10]. This is an operating system version of the user-space DBI framework called Pin [40]. Unfortunately, the code for this project was not made available, and therefore we could not test its suitability.

DARC requires a disassembler in order to identify appropriate instrumentation points, such as calls, returns, and interrupts. We used a disassembler that we modified from the Hacker Disassembler Engine v0.8 [59]. We chose this disassembler because it is very small and lightweight (298 lines of assembly in our version). We converted the NASM syntax to GNU assembler syntax using intel2gas [51], and converted the opcode table from NASM to C. We also added a C function that returns the callsites of a given function, which took 84 lines of code. This function also handles tail-call optimizations, where if a function x calls a function y immediately before returning, the

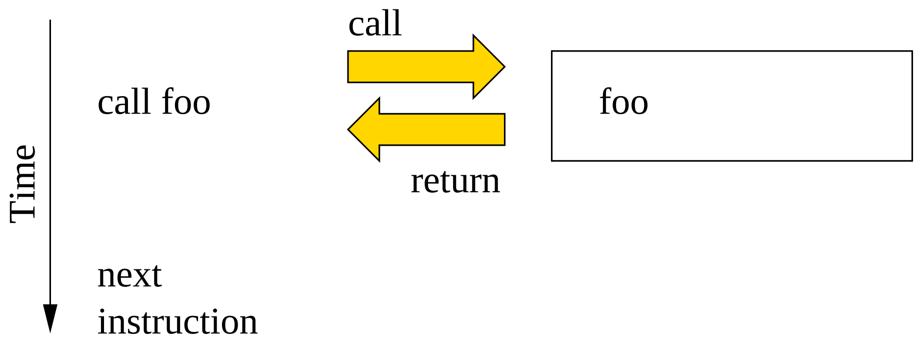


Figure 4.4: An uninstrumented call site. Function `foo` is called, and another instruction is executed after returning from the call.

call and return are be replaced by a jump to y, and y then returns to x's return address. This provides the minimum functionality necessary for DARC except for the identification of basic blocks, which our current prototype does not yet support.

An earlier DARC prototype inserted two kprobes per call site: one on the call instruction itself and one on the subsequent instruction. Figure 4.4 illustrates an uninstrumented call site, while Figure 4.5 depicts the same call site patched by DARC. We will refer to this instrumentation as `2-kprobes`. This implementation option may cause problems if the code path jumps to the address of the second kprobe without executing the first one. We had to include an extra check to ensure that the code in the second kprobe was executed only if the first kprobe was called.

Another problem with the `2-kprobes` implementation is that if the function body contains two consecutive call instructions, then two kprobes will be placed on the address of the second call: the second kprobe for the first call site and the first kprobe for the second call site. In this scenario, the order in which the kprobes are placed is important. One more obstacle appears in the presence of tail calls: the instruction after such a call resides beyond the function body. Section 4.2.2 illustrates how our optimizations solve these problems above in addition to improving DARC's performance.

One more interesting aspect of DARC's instrumentation is related to synchronization. DARC instruments the code of the running process without stopping it. This fact creates the possibility that some of DARC's instrumentation can be inserted asynchronously and consequently executed (or not executed) unexpectedly. We assume in our design that the time spent on code modifications by DARC is much shorter then the duration of the workload being analyzed. We then ensure that any DARC instrumentation will not crash in the case of an unexpected invocation. A possible error that could occur in this case is that incorrect time measurements are associated with some function nodes around the time of instrumentation. However, we can safely disregard these situations, as the amount of time needed to modify the code is small.

4.2.2 Optimizations

Each kprobe hit causes an interrupt, which is very costly on all computer architectures. Consequently, our earlier prototype, which used two kprobes per function invocation, was quite inefficient, especially if the ftree was wide. Our optimizations aim to reduce the number of kprobes used by DARC. In addition to improving performance, these optimizations provide other benefits that are discussed below. The performance impact of these optimizations is described in Chapter 8.

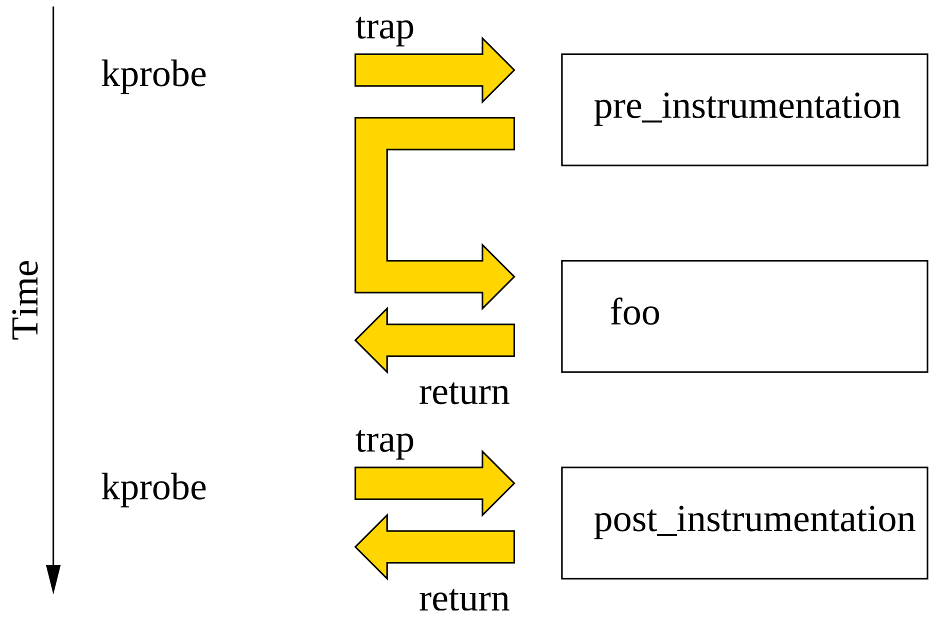


Figure 4.5: The standard way that DARC instruments function calls with kprobes (the `2-kprobes` implementation). Here the call to `foo` was replaced with a trap for one kprobe, and the next instruction was replaced by a trap for the second kprobe.

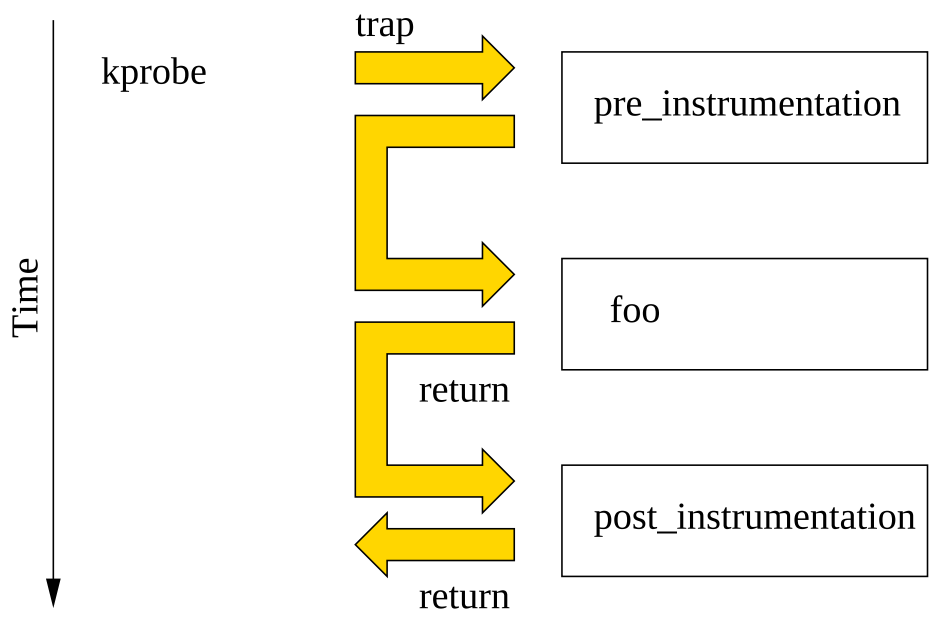


Figure 4.6: A kernel-level DARC return probe optimization that reduces the number of kprobes by half (the `rettramp` implementation). The pre-instrumentation code modifies the return address of `foo` on the stack so that it points to the post-instrumentation. The next instruction that was shown in Figure 4.4 is not affected and is not shown.

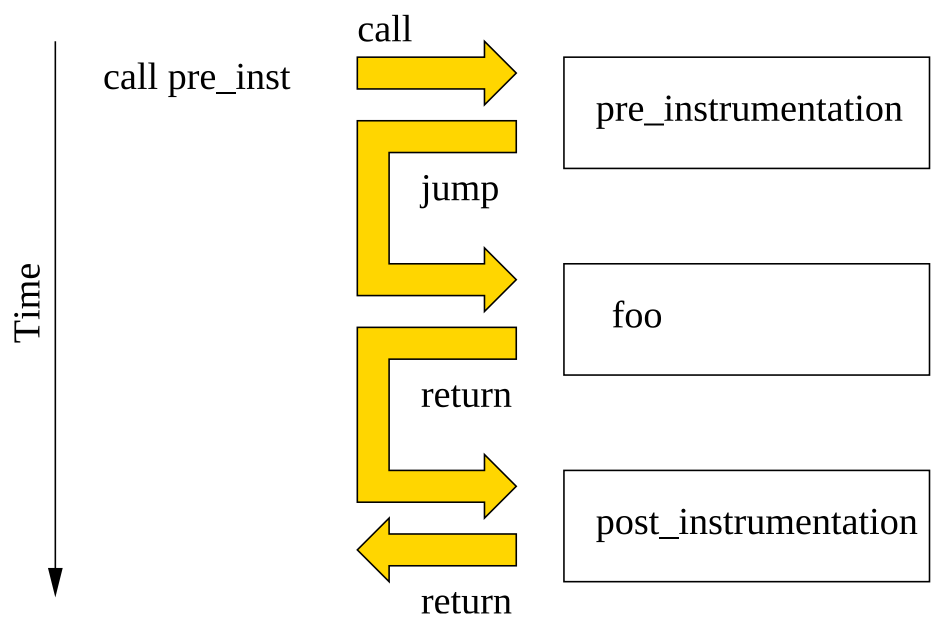


Figure 4.7: Using light-weight call trampolines instead of kprobes to instrument call sites (the `calltramp` optimization). The original call destination (`foo`) is replaced so that it points to `pre_instrumentation`. The next instruction that was shown in Figure 4.4 is not affected and is not shown.

The first optimization we have developed is the removal of kprobes that intercept exits from a function (the second kprobe placed on a call site). To accomplish this, the kprobe installed on the call instruction saves the return address of the function f that is being called. The kprobe then modifies f's return address on the stack to point to DARC's function that essentially contains the code of the second kprobe handler. This function, after measuring the latency of the callee, jumps to the saved return address, as illustrated in Figure 4.6. We refer to this as the `rettramp` optimization, for return trampoline.

The `rettramp` optimization has five benefits. First, it guarantees that the code contained in the second kprobe is executed only when the entrance instrumentation was also executed. Second, we no longer need to specially handle the installation precedence of the kprobes. Third, DARC now uses only one kprobe per function call, which improves performance. Fourth, tail calls are automatically handled by this instrumentation. Finally, the `recursion_count` variable that was introduced in Section 3.11 is no longer needed.

Another part of the `rettramp` optimization is in the instrumentation of f_0. This function has a kprobe at its start which cannot be removed using the above techniques. However, we can remove the return kprobe for f_0 by using the same approach: saving the original return address and replacing it by the address of post-instrumentation function.

The `rettramp` DARC implementation installs kprobes only on call instructions (except for the kprobes in f_0). This observation allowed us to perform an additional optimization. Rather than placing a kprobe on the call instruction, DARC saves the original address of the callee (which is encoded in the instruction itself) and replaces it with the address of the first DARC handler. When this patched instruction is executed, the DARC handler is called: it performs its required job and jumps to the address of the original callee. This is illustrated in Figure 4.7, and we refer to this optimization as `calltramp`.

With the `calltramp` optimization, DARC does not use kprobes for function calls. The notable exception here is indirect calls. The size of the indirect call instruction is only two bytes on the x86 architecture with one byte for the address, so there is no possibility to overwrite this address by the handler's address, and DARC is forced to use a kprobe in this case. Fortunately, systems almost always use direct calls. For example, this optimization can instrument approximately 95.7% of calls in the Linux kernel itself, and 97.5% of calls found in Linux kernel modules. Additionally, this limitation does not exist for architectures with fixed-width instructions.

Chapter 8 evaluates the effects of our optimizations on DARC's performance.

4.2.3 User-Space Instrumentation

Two related questions arise when implementing DARC for analyzing user-space applications: whether to perform the instrumentation in user-space or kernel-space, and which binary instrumentation framework will work best. We developed the *suprobes* (simple user probes) framework, which is a modified version of IBM's uprobes (user probes) framework [34]. There are three main differences between the existing uprobes implementation and suprobes. First, uprobes requires the user to specify the application using a file system identifier and inode number. To make the interface more similar to our existing kernel implementation of DARC, we changed this to use the PID of the running process instead. Second, we added the ability to assign priorities, which is useful if more than one suprobe is placed on a given instruction. Third, the code was updated to run on our newer kernel.

Suprobes resemble kprobes, but allow for the insertion of probes into user-space applications. In addition to a code address and the addresses of handlers required by kprobes, the PID of the process must be provided to suprobes. Internally, suprobes work similarly to kprobes: they replace the first byte of the instruction at the specified address with a trap instruction. When the trap fires, a kernel handler is called. The suprobes framework is implemented as a kernel module with a small patch for the kernel itself (35 lines), which exports some required kernel functions to the module.

Using our suprobes framework, we implemented DARC using almost the same code as for the kernel implementation: we only had to replace kprobes by suprobes where necessary. Our current implementation does not utilize any of the optimizations that we used in the kernel portion because the return and call trampolines need instrumentation addresses in user-space to jump to. To provide such addresses safely, DARC would need to examine the process's mapped page ranges and insert the instrumentation functions in a range of unused addresses. Further, the data structures would reside in user-space, and the kernel portion of DARC would map them into its own memory to avoid data copies across the user-kernel boundary. This means that the overhead for using DARC to analyze user-space applications is higher than the overhead for using DARC in the kernel. Chapter 8 provides detailed performance evaluations and comparisons.

In order to detect call instructions, DARC uses a disassembler. To search for call instructions, we need to know the length of the function's body so that we know when to conclude our search. In the kernel, this information is provided by the in-kernel symbol table. For user-space applications, we extract the symbol information before DARC starts its analysis and load it into the kernel using the `darcctl` tool.

To instrument the return from f_0, the kernel portion of DARC uses a special type of kprobe called a *return kprobe*. However, the suprobes framework does not support this type of probe, as it

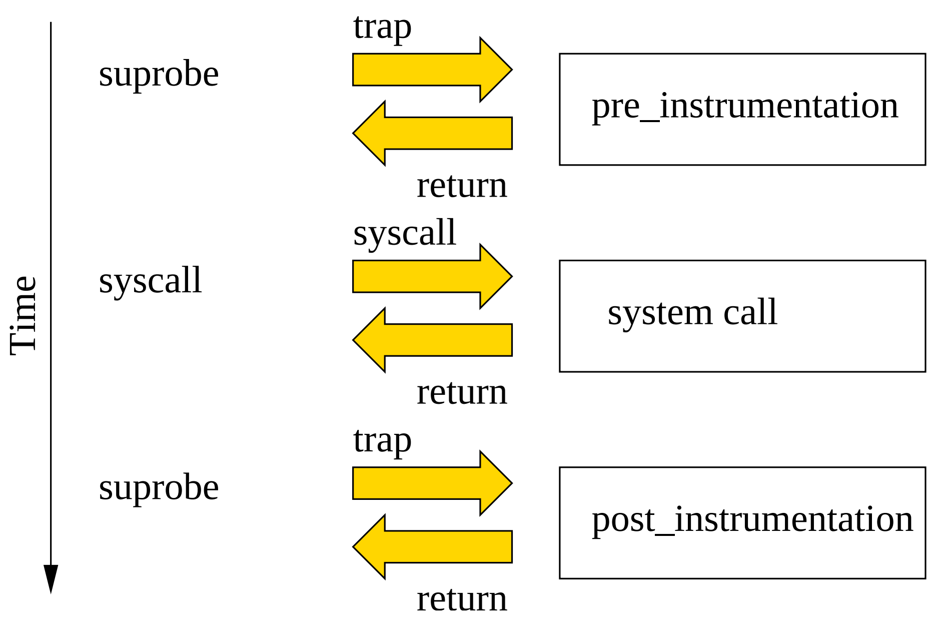


Figure 4.8: Using suprobes to instrument system calls. The probes are placed on the instruction before and after the system call being instrumented.

would require the instrumentation code to reside in user-space. Because of this, we scan the body of f_0 to find all return instructions and place suprobes on them. Note that if we have a call instruction immediately followed by a return instruction in f_0's body, then two suprobes will be placed on the return instruction. Moreover, the second suprobe for the call should be called before the suprobe for f_0's return. We used the priority feature of suprobes to handle this case, which allows us to properly arrange the order of suprobes invocations.

4.2.4 Crossing the User-Kernel Boundary

An interesting feature of our DARC implementation is the ability to cross the user-kernel boundary [49]. To accomplish this, we place suprobes on the two instructions surrounding interrupt, syscall, and sysenter instructions. This is illustrated in Figure 4.8. We cannot place suprobes directly on these instructions because according to the suprobes design, these instructions will be executed in a single-stepped atomic context. This will break the blocking system calls, such as `read` and `write`. When entering the kernel, we record the system call number and translate it to the system call handler using the system call table. If the system call is selected as a root cause function, its handler is instrumented as a kernel function.

Chapter 5

Limitations

DARC has three main limitations. First, DARC assumes that the latency distribution of f_0 is fairly static. If the behavior of the code being analyzed changes during analysis, DARC may not be able to conclude the analysis. However, we expect this to be rare. Second, inline functions and macros cannot be analyzed separately because this DARC implementation uses binary instrumentation. Third, if the source of the code being analyzed is not available, the binary should include symbols so that the output can be translated from function addresses to names. The function names should also be descriptive enough for the user to guess what the function does. Otherwise, the user must disassemble the binary to understand the root cause.

Chapter 6

Experimental Setup

We now describe the experimental setup used for our use cases (Chapter 7) and our performance evaluation (Chapter 8). The test machine was a Dell PowerEdge SC 1425 with a 2.8GHz Intel Xeon processor, 2MB L2 cache, and 2GB of RAM, and a 800MHz front side bus. The machines were equipped with two 73GB, 10,000 RPM, Seagate Cheetah ST373207LW Ultra320 SCSI disks. We used one disk as the system disk, and the additional disk for the test data.

The operating system was Fedora Core 6, with patches as of October 08, 2007. The system was running a vanilla 2.6.23 kernel that with the suprobes patch applied. The file system was ext2, unless otherwise specified. Some relevant program versions, obtained by passing the `--version` flag on the command line, along with the Fedora Core package and version are GNU grep 2.5.1 (grep 2.5.1-54.1.2.fc6) and GNU stat 5.97 (coreutils 5.97-12.5.fc6).

To aid in reproducing these experiments, the DARC and workload source code, a list of installed package versions, the kernel configuration, and benchmark results are available at `http://www.fsl.cs.sunysb.edu/docs/darc/`.

Chapter 7

Use Cases

In this section we describe some interesting examples that illustrate DARC's ability to analyze root causes. To highlight the benefits of using DARC, we show three usage examples that were first published in the OSprof paper [28]. We compare the use of DARC to the manual analysis described in the OSprof paper. We show that DARC does not require as much expertise from the user, is faster, and gives more definitive results.

The use cases that we present highlight three of DARC's interesting aspects: analyzing interrupts, investigating asynchronous paths, and finding intermittent lock contentions. We recreated all of the test cases on our test machine. The first example used a workload that reads zero bytes of data in a loop. The remaining two examples used a `grep` workload, where the `grep` utility searched recursively through the Linux 2.6.23 kernel source tree for a nonexistent string, thereby reading all of the files. The kernel was compiled using the *make defconfig* and *make* commands. We will specify the values of the `start_ops` and `decision_time` parameters (described in Chapter 4) for each use case.

7.1 Analyzing Interrupts

Figure 7.1 shows a profile of the read operation issued by two processes that were repeatedly reading zero bytes of data from a file. This profile contains three peaks, and we order them from left to right: first (bins 7–9), second (10–13), and third (14–18). The first peak is clearly the usual case when the read operation returns immediately because the request was for zero bytes of data.

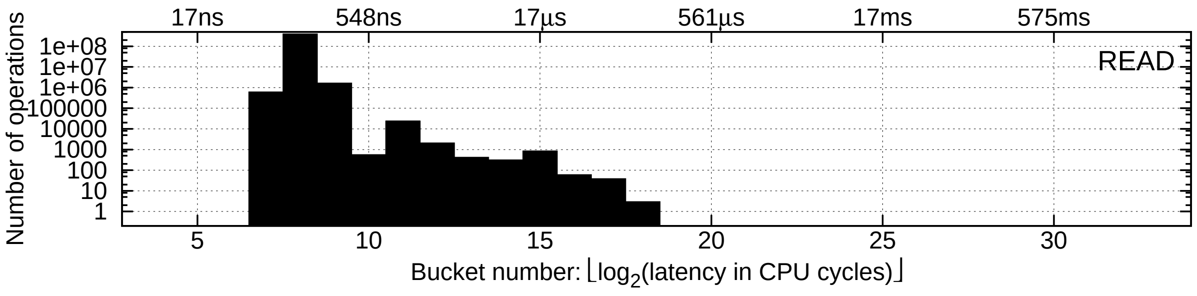


Figure 7.1: A profile of the read operation that reads zero bytes of data.

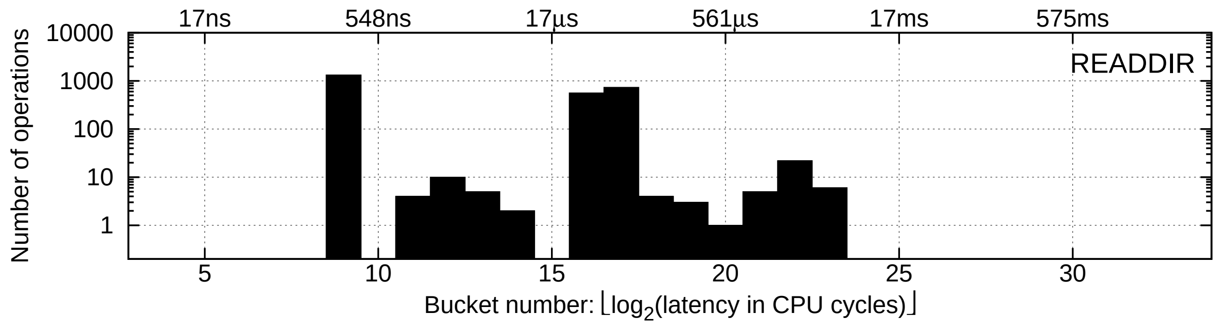


Figure 7.2: A profile of the ext2 readdir operations captured for a single run of grep -r on a Linux 2.6.23 kernel source tree.

For this peak, DARC reports a root cause path showing the read path up to the point where the size of the read is checked to be zero. At this point, the functions return because there is no work to be done. The root cause path as shown by DARC is:

```
vfs_read →
    do_sync_read →
        generic_file_aio_read
```

Note that the second and third functions here are indirect calls, but DARC displays name of the target function. This output tells us that the read operation is responsible for the peak.

In the OSprof paper, the authors hypothesized that the second peak was caused by the timer interrupt. They based this on the total runtime of the workload, the number of elements in the peak, and the timer interrupt frequency. Recall from Section 3.12 that DARC instruments the main interrupt handling routine (`do_IRQ` in our case). This instrumentation checks if the target process was executing a function that DARC is currently analyzing. If so, it records the latencies for each interrupt type and attributes these measurements to the executing function.

In our case, we set f_0 to `vfs_read`, as before. DARC reported “interrupt 0” as the root cause, which our post-processing script translated as “timer interrupt.” DARC arrived at this conclusion because after comparing the latencies of `vfs_read`, its callees, and the latencies for each interrupt number, interrupt 0 always had the highest latency. Although it was possible to determine the cause of the peak without DARC, doing so would have required deep insight and thorough analysis. Even so, the cause of the peak could not be confirmed with manual analysis. DARC confirmed the cause, while requiring much less expertise from the user.

DARC discovered that the third peak was caused by interrupt 14 (the disk interrupt). This was not reported in the OSprof paper, as it is very difficult to analyze manually. DARC analyzed this third peak in the same way as it did with the second, and reported the root cause clearly and easily.

For analyzing this profile, we set `start_ops` to 100 and `decision_time` to 20. From our experience, we found these values to be generally sufficient.

7.2 Analyzing Asynchronous Paths

Running the `grep` workload on ext2 resulted in the profile shown in Figure 7.2. There are four peaks in the profile of the `readdir` operation, ordered from left to right: first (bin 9), second (11–

14), third (16–17), and fourth (18–23). In the OSprof paper, prior knowledge was used as a clue to the cause of the first peak. The OSprof authors noted that the latency is similar to the latency of reading zero bytes (see the first peak in Figure 7.1). This implies that the operation completes almost immediately. The OSprof authors guessed that the cause of the peak is reading past the end of the directory. They confirmed this by modifying the OSprof code to correlate the latency of the first peak with the condition that the `readdir` request is for a position past the end of the directory. We ran DARC to analyze this first peak, and the resulting root cause path consisted of a single function: `ext2_readdir`. This is because the function immediately checks for reading past the end of the directory and returns.

The causes of the remaining peaks were analyzed in the OSprof paper by examining the profile for the function that reads data from the disk. The OSprof authors noted that the number of disk read operations corresponded to the number of operations in the third and fourth peaks. This indicates that the operations in the second peak are probably requests that are satisfied from the operating system's cache, and the operations in the third and fourth peaks are satisfied directly from disk. Further, based on the shape of the third peak, they guessed that the operations in that peak were satisfied from the disk's cache. Based on the knowledge of the disk's latency specifications, they further guessed that the operations in the fourth peak were affected by disk-head seeks and rotational delay.

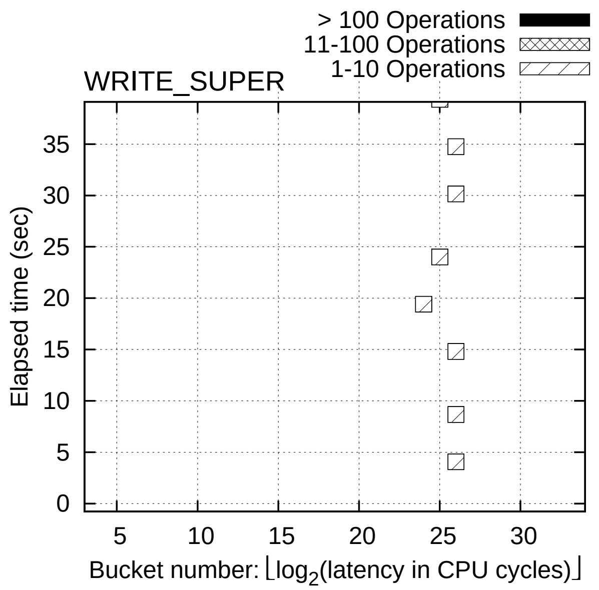
When DARC analyzed the second peak, it displayed the following root cause path, which clearly indicates that the root cause is reading cached data:

```
ext2_readdir →
   ext2_get_page →
      read_cache_page →
         read_cache_page_async →
            __read_cache_page
```

For the third peak, DARC produced the following root cause path:

```
ext2_readdir →
   ext2_get_page →
      read_cache_page →
         read_cache_page_async →
            ext2_readpage →
               mpage_readpage →
                  mpage_bio_submit →
                     submit_bio
```

Notice that first portion of the root cause path is the same as for the second peak. However, after calling the `__read_cache_page` function to read data from the cache, it calls `ext2_readpage`, which reads data from disk. Eventually the request is placed on the I/O queue, with the `submit_bio` function (`bio` is short for block I/O). For all of the use cases, DARC was tracking asynchronous disk requests, as described in Section 3.13. If a request reaches the `submit_bio` function and is not filtered by PID or call-path filtering, DARC records the address for the current block I/O structure in a hash table. After DARC analyzed this peak for the first time, we restarted DARC, giving it the previous call-path output as input. We set f_0 to the function that dequeues the requests was the queue. This allowed us to analyze the asynchronous portion of the root cause



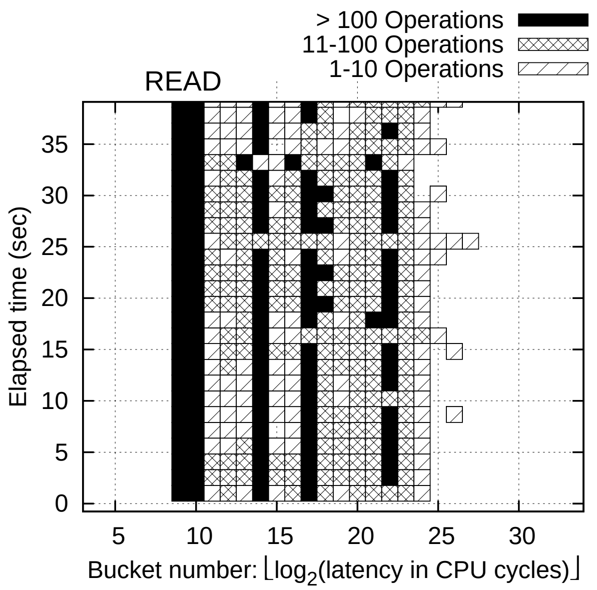


Figure 7.3: Reiserfs 3.6 file-system profiles sampled at 1.5 second intervals.

path while retaining the PID and call-path filtering from the main path. DARC then produced the following output:

```
__make_request →
    __elv_add_request →
        elv_insert →
            cfq_insert_request →
                blk_start_queueing →
                    scsi_request_fn
```

In this root cause path, `__make_request` removes the request from the queue. The `cfq_insert_request` function is a function pointer that is specific to the I/O scheduler that the kernel is configured to use (CFQ in this case). Finally, `scsi_request_fn` is a SCSI-specific function that delivers the request to the low-level driver. The root cause path for the fourth peak was identical to that of the third, indicating that disk reads are responsible for both peaks, as reported by OSprof. Unfortunately, because requests from both peaks are satisfied by the disk, the factor that differentiates the two peaks is hardware, and therefore software techniques cannot directly find the cause. In this case, one must use manual analysis to infer the causes.

7.3 Analyzing Intermittent Behavior

On Reiserfs, the `grep` workload resulted in the profiles shown in Figure 7.3. These are time-lapse profiles. Because OSprof profiles are small, OSprof can store latency measurements in different histograms over time to show how the distributions change. The x-axis represents the bin number, as before. The y-axis is the elapsed time of the benchmark in seconds, and the height of each bin is represented using different patterns. The profile on the left is for the `write_super` operation, which writes the file system's superblock structure to disk. This structure contains information pertaining to the entire file system, and is written to disk by a buffer flushing daemon every five seconds by default. The profile on the right is for the `read` operation.

The bins on the right side of the `read` profile correspond to the bins in the `write_super` profile. This indicates that there is some resource contention between the two operations, but the OSprof analysis methods cannot confirm this, nor can they give more information about the resource in question. The OSprof authors attributed this behavior to a known lock contention. A user that was not informed about this lock contention would struggle to analyze this behavior manually. First, the user may be confused as to why the file system is writing metadata during a read-only workload. The user must know that reading a file changes the time it was last accessed, or *atime*. Further, the atime updates are written by the buffer flushing daemon, which wakes periodically. This would lead the user to collect a time-lapse profile for this case. In the end, only source code investigation would provide an answer.

Using DARC, we analyzed both profiles shown in Figure 7.3. We first ran DARC on the read path. We set `start_ops` to 5,000 so that enough delayed read operations would be executed, and we set `decision_time` to 5, because the read operations do not get delayed very often. The root cause path that DARC displayed was:

```
vfs_read →
   do_sync_read →
      generic_file_aio_read →
         do_generic_mapping_read →
            touch_atime →
               __mark_inode_dirty →
                  reiserfs_dirty_inode →
                     lock_kernel
```

We can see from this that the read operation (`vfs_read`) caused the atime to be updated (`touch_atime`). This caused the `lock_kernel` function to be called. This function takes the global kernel lock, also known as the big kernel lock (BKL). To understand why this happens, we can look at the siblings of the `lock_kernel` fnode (not shown above because they are not root causes): `journal_begin`, `journal_end`, and `unlock_kernel`. This tells us clearly that Reiserfs takes the BKL when it writes the atime information to the journal.

For the `write_super` operation, we turned off PID filtering because the superblock is not written on behalf of a process. We set both `start_ops` and `decision_time` to 5, because the `write_super` operation does not get called frequently. DARC produced the following root cause path:

```
reiserfs_write_super →
   reiserfs_sync_fs →
      lock_kernel
```

Again, the siblings of the `lock_kernel` fnode are journal-related functions. We now know that the lock contention is due to Reiserfs taking the BKL when writing atime and superblock information to the journal.

Chapter 8

Performance Evaluation

We first measured the performance of the pure instrumentation. The results are described in Section 8.1. For measuring the performance of DARC, we had two main requirements for choosing a workload. First, it should run for a long enough time to obtain stable results—we ensured that all tests ran for at least ten minutes [75]. Second, the workload should call one function repeatedly. This function will be the one DARC analyzes, so the DARC instrumentation will be executed as often as possible. We measured DARC's performance using three workloads. The first workload measures the overhead of DARC on an artificial source code tree. We varied the fanout and height of the tree to see how these parameters affect DARC's overheads. We call this the *synthetic* workload, and it is described in Section 8.2. The second, described in Section 8.3, repeatedly executes the `stat` system call on a single file, which returns cached information about the file. This shows DARC's overhead when investigating a relatively low-latency, memory-bound operation. This is a worst-case scenario, as DARC is operating on a single fast function that is called at a very high rate. The third workload reads a 1GB file in 1MB chunks 50 times using direct I/O (this causes data to be read from disk, rather than the cache). This shows the overheads when investigating a higher-latency, I/O-bound operation. This workload is further discussed in Section 8.4.

Recall the two parameters that affect DARC's overhead that we described in Chapter 7: `start_ops` and `decision_time`. For the `stat` and `read` benchmarks, we chose these values such that the analysis does not finish by the time the benchmark concludes, forcing DARC to run for the entire duration of the benchmark. We present the values of these parameters for each benchmark, which are orders of magnitude higher than the values used in the use cases presented in Chapter 7.

We used the Autopilot v.2.0 [81] benchmarking suite to automate the benchmarking procedure. We configured Autopilot to run all tests at least ten times, and computed the 95% confidence intervals for the mean elapsed, system, and user times using the Student-t distribution. In each case, the half-width of the interval was less than 5% of the mean. We report the mean of each set of runs. To minimize the influence of consecutive runs on each other, all tests were run with cold caches. We cleared the caches by re-mounting the file systems between runs. In addition, the page, inode, and dentry caches were cleaned between runs on all machines using the Linux kernel's `drop_caches` mechanism. This clears the in-memory file data, per-file structures, and per-directory structures, respectively. We called the `sync` function first to write out dirty objects, as dirty objects are not free-able.

8.1 Instrumentation Overheads

Instrumentation Method	**Overhead (μs)**
`2-kprobes` (post-handlers)	2.75
`2-kprobes` (pre-handlers)	1.89
`rettramp`	1.54
`calltramp`	0.76
`suprobes`	4.59

Table 8.1: Overheads for various instrumentation methods.

DARC makes do with minimalistic instrumentation: the instrumentation need only insert code at a specific code address. We used several instrumentation methods in DARC's implementation, as discussed in Section 4.2.2.

To measure the overhead of each instrumentation method, we created a benchmark where the parent function calls a child function 2×10^8 times. We measured the time required to execute the benchmark without any instrumentation. We then measured the time required to execute the same program, but added instrumentation to the call of the child function. We used the results to compute the mean overhead for a single instrumented function invocation. Table 8.1 summarizes the results for the various instrumentation methods: `2-kprobes`, `rettramp` (one kprobe and a return trampoline), `calltramp` (a call trampoline and a return trampoline), and `suprobes` (2 suprobes).

We use the first four techniques shown in the table only in the kernel. We implemented the `2-kprobes` technique using kprobes' post-handlers and pre-handlers. Using pre-handlers is faster, because kprobes that use post-handlers require two exceptions and four context switches, as described in Section 4.2.1. Depending on the replaced instruction, kprobes that utilize only a pre-handler are implemented using only one exception and two context switches. This is because there is no need to execute a post-handler, so after executing the replaced instruction the kernel can resume executing from the instruction following the kprobe. For the remainder of this chapter, `2-kprobes` refers to the pre-handler technique. The `rettramp` technique is faster still, but it must use a post-handler in the kprobes that it inserts, and so the benefits of the optimization should be compared to the post-handler version of `2-kprobes`. Finally, the `calltramp` optimization is the fastest, as expected.

For user-space instrumentation, we see that the `suprobes` technique is more expensive than any kernel technique. This is because a suprobe requires two context switches into the kernel: one to execute the handler, and another to single-step the replaced instruction. This means that each replaced instruction incurs four context switches. We expect that porting the kernel optimizations to DARC's user-space instrumentation will yield significant improvements.

8.2 Synthetic Workload

DARC's overhead depends not only on the instrumentation method used, but also on the structure of the analyzed source tree. A larger fanout means that more call sites need to be instrumented for a particular level. A deeper tree means that there will be more levels, and more analysis must be

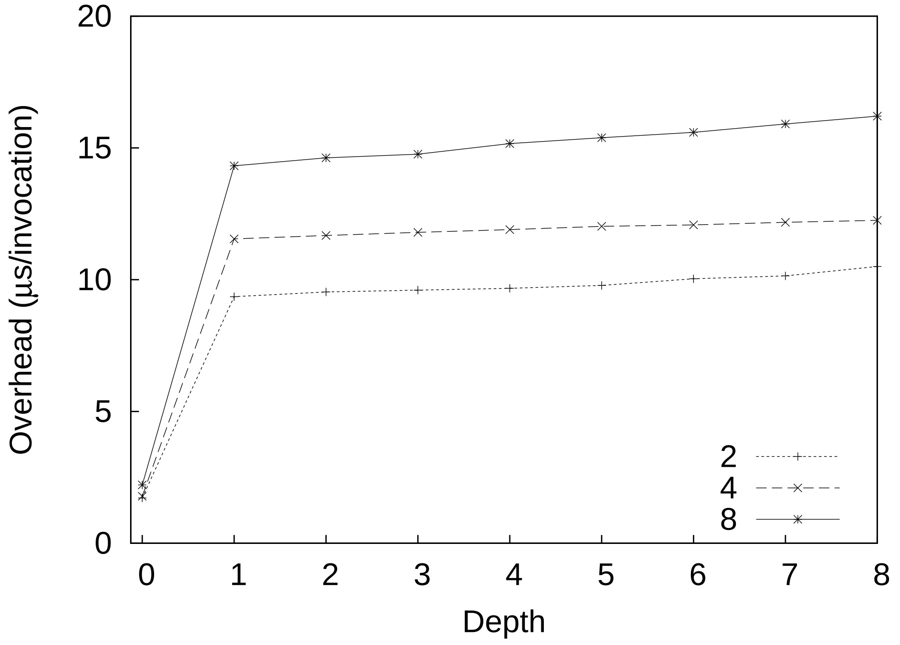


Figure 8.1: Results for the synthetic workload with the in-kernel trees for fanouts of 2, 4, and 8. The analysis here is performed asynchronously.

performed. The synthetic workload presented here illustrates how DARC's overhead is affected by the structure of the tree.

We ran this benchmark with three configurations. In the first, the tree resides entirely in a kernel module. The kernel code is executed by the benchmark using the `ioctl` system call. In the second configuration, the source tree resides entirely in a user-space program. In the third configuration, the initial half of the tree resides in a user-space program, and the second half resides in the kernel. To execute the kernel portion, the user-space code executes a system call that we created: the corresponding handler is the root of the kernel portion of the tree. The depth of each tree is 8 levels, and we used trees that had a fanout of 2, 4, and 8. Only one function in the leaf of each tree has a latency that is higher than the others, providing a unique root cause path to the bottom of the tree. The benchmark invokes the f_0 function enough times so that the elapsed time for the uninstrumented case is more than 10 minutes.

For this benchmark we had DARC descend into the code differently than for the `stat` or random-read benchmarks. Here, rather than have DARC descend through the levels of the tree in equally-spaced intervals throughout the benchmark, we forced DARC to descend to the required level at the start of the benchmark. DARC analyzed at that level for the entire duration of the benchmark. This allowed us to measure the overheads for each phase of DARC's analysis separately, based on the depth in the source tree. We also measured the overhead for the phase where only the OSprof instrumentation (see Figure 3.3) is inserted. We refer to this as depth 0. In depth 1, the overhead for the instrumentation shown in Figure 3.5 is measured. Subsequent depths have more functions instrumented, as shown in Figure 3.7.

Figure 8.1 presents the results for the kernel trees. We tested only the `calltramp` implementation, as it clearly performs the best. For depth 0, the overheads were approximately the same for

		Depth								
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Fanout	**2**	12	15	15	15	15	15	15	15	15
	4	12	15	15	15	15	15	15	15	15
	8	13	15	15	15	15	15	15	15	15

Table 8.2: The minimum OSprof bucket numbers whose elements do not shift on average due to DARC's overheads in the kernel. The analysis here is performed asynchronously.

		Depth								
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Fanout	**2**	12	14	14	14	14	14	14	14	14
	4	12	14	14	14	14	14	14	14	14
	8	13	15	15	15	15	15	15	15	15

Table 8.3: The minimum OSprof bucket numbers whose elements do not shift on average due to DARC's overheads in the kernel. The analysis here is performed synchronously.

all the trees. This makes sense, as the fanout should not affect the OSprof phase. When DARC begins to measure latencies of callees in depth 1, fanout starts to affect the overhead, with the addition of f trampolines (where f is the fanout). For each level between depths 1 and 8, we remove $f - 1$ trampolines from the previous level (we uninstrument all call sites except for the root cause function), and add f trampolines to the current level, so in total we add 1 trampoline for each level. Notice that there is a large increase in overhead between depths 0 and 1, and then the overhead increases gradually when DARC descends to lower levels between depths 1 and 8. The large initial increase happens because at this point DARC begins to gather and process the latencies. This tells us that the latency analysis code affects overheads much more than the instrumentation.

Table 8.2 shows the minimum OSprof bucket numbers whose elements do not shift due to DARC's overheads while running in the kernel. We arrive at these numbers by calculating the average latency for each OSprof bucket i:

$$\frac{3}{2}2^i cycles \times \frac{10^6 \mu s}{2.8 \times 10^9 cycles} \tag{8.1}$$

Bucket i has a range from 2^i to 2^{i+1} cycles, so we take the midpoint of this range. We then convert this value from cycles to μs for a 2.8GHz CPU (the speed of the CPU used for the benchmarks). We see here that operations up to bucket 15 will be shifted by at least one bucket due to DARC's overheads. As we can see from Figure 7.2, this means that latencies corresponding to CPU operations will generally be shifted. Although we do experience shifts in lower buckets, we showed in Section 7.1 and 7.2 that we were able to use DARC to analyze fast CPU operations between buckets 7 and 9. Still, lowering DARC's overheads would improve the user's experience.

We theorized that performing the analysis asynchronously was actually hurting performance. This is because each operation required allocating a data structure, copying the latency information into it, locking the request queue, and then placing the request on the queue. We modified the code to perform the analysis synchronously. The results are shown in Figure 8.2. The increase in

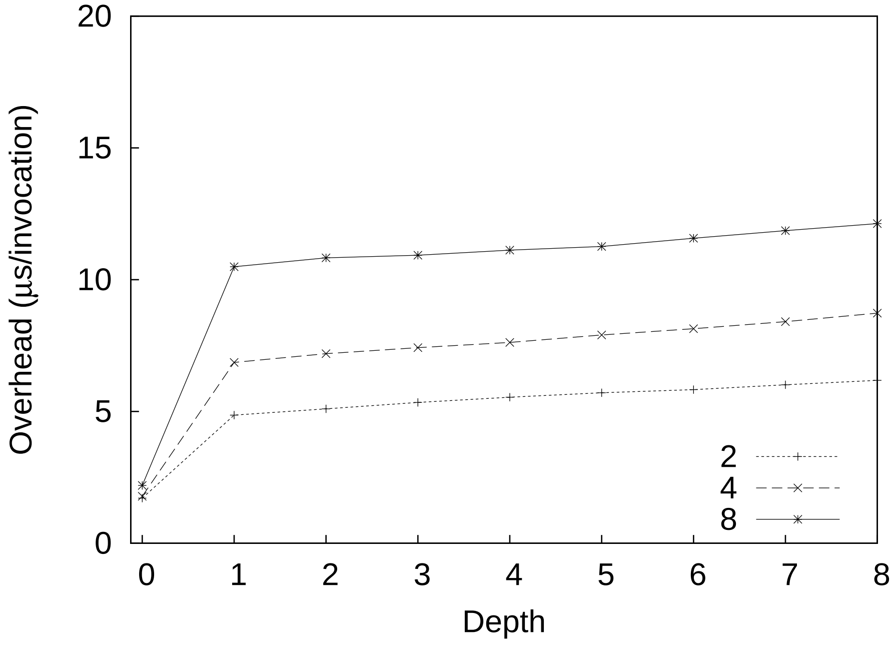


Figure 8.2: Results for the synthetic workload with the in-kernel trees for fanouts of 2, 4, and 8. The analysis here is performed synchronously.

latencies between levels 0 and 1 has now been significantly reduced. As expected, the remainder of the results improve by the same constant, as the amount of analysis stays constant regardless of depth. The overall improvement was between 25.2% and 48.1%, with an average of 34.8%. We can see in Table 8.3 that the bucket values were lowered by 1 for fanouts of 2 and 4. We use only the synchronous implementation for the remainder of the synthetic tests.

We expect that using function replacement [52] for instrumenting code will yield further performance improvements. This technique makes a copy of a given function and places the instrumentation directly in this copy. The first instruction of the original function is then replaced by a jump to the new function. Function replacement can be used to remove the relatively expensive kprobe that is currently needed to instrument f_0. It may also prove faster than our `calltramp` optimization for other instrumented functions. Another optimization would be to use a *djprobe* rather than a kprobe for instrumenting f_0. A djprobe replaces a given instruction with a jump instruction rather than an interrupt. This significantly reduces the overheads of the probe [24], but djprobes have several problems, and are still under development [55]. The problems mostly arise from the fact that jump instructions on the x86 architecture may be longer than other instructions because they require a destination parameter (between two and five bytes), while interrupt instructions require only one byte.

The results for DARC operating on the user-space source tree are shown in Figure 8.3. Here we see the same large increase between depths 0 and 1 as we did with the kernel tree. However, in this case, descending to lower levels is more expensive than in the kernel. The reason is the high overhead of the suprobes framework, as discussed in Section 8.1. These results indicate that although we were able to analyze this short-running function, it would be worthwhile to port DARC's kernel optimizations to user space to reduce overheads.

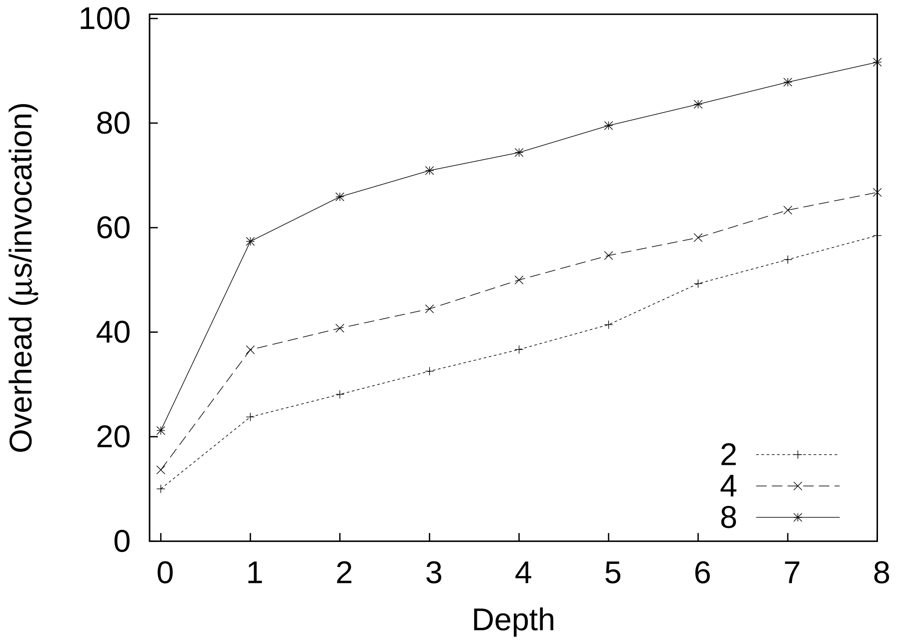


Figure 8.3: Results for the synthetic workload with the user-space trees for fanouts of 2, 4, and 8. The analysis here is performed synchronously.

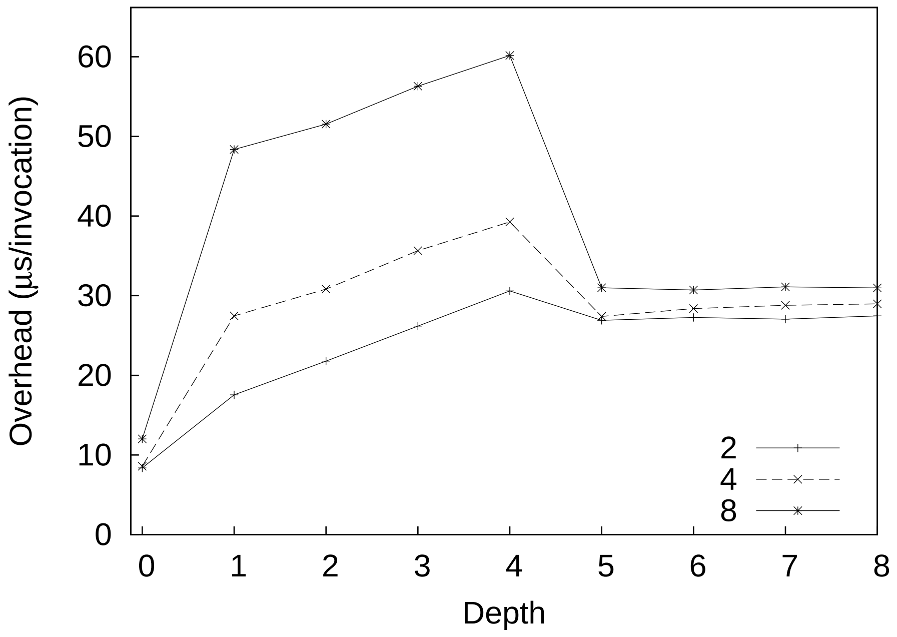


Figure 8.4: Results for the synthetic workload with trees of fanout 2, 4, and 8 that cross the user-kernel boundary. The analysis here is performed synchronously.

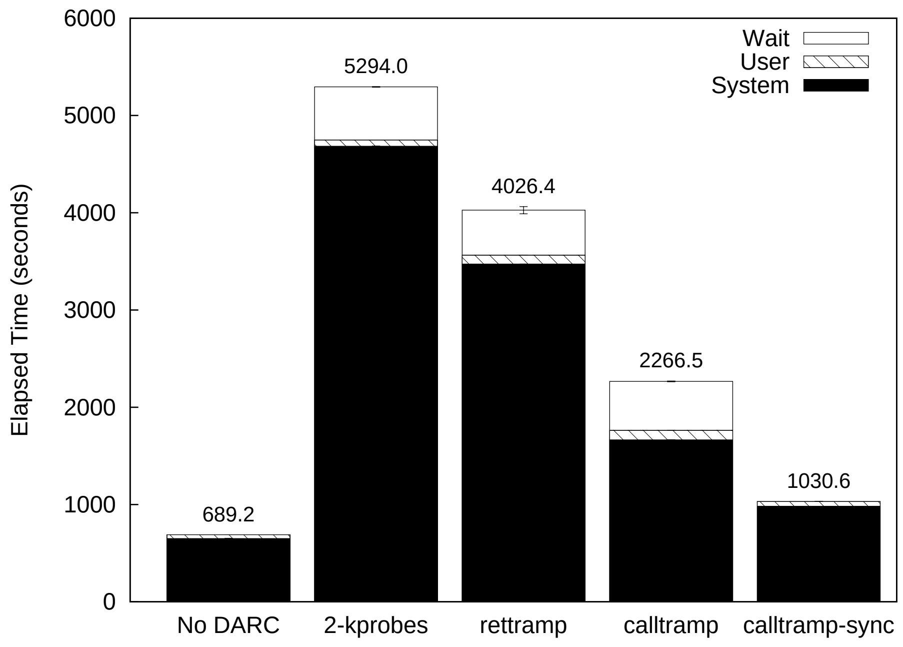


Figure 8.5: Results for the `stat` benchmark with various optimizations. Note that error bars are always drawn, but may be too small to see.

Finally, we measured the overhead of DARC operating on the benchmark that crosses the user-kernel boundary. The results are presented in Figure 8.4. Until depth 4, the graph resembles that of the user-space graph shown in Figure 8.3. This is logical, as this is the user-space portion of the tree. The code at depth 5 is located in the kernel, and here the overhead decreases. This is because when DARC moves from depth 4 to depth 5, it removes all of the suprobes from the previous level except for those that correspond to the root cause functions. For example, in the tree of fanout 4, it means that 6 suprobes are removed (3 call sites, each with 2 suprobes). In the next level, which is in the kernel, 8 call/return trampolines are added, but these add very little overhead. In addition, between depths 5 and 8, the variation in performance between different fanout levels is minimal. This is because the same number of suprobes are in place in all cases, because they are only present along the root cause path, which is the same in all cases. Now the performance difference between fanouts is due to in-kernel call trampolines, which impose significantly lower overheads as compared to suprobes.

8.3 Stat Workload

We ran the `stat` workload with 300 million operations, resulting in an elapsed time of approximately 689 seconds without DARC. We then used DARC to analyze the `stat` call, and saw that there was one peak, with a single root cause path that is five levels deep (shown later). The number of fnodes in each level of the ftree, from top to bottom, were 3, 3, 11, 1, and 6. Because we wanted DARC to reach the maximum depth without finishing its analysis, we set `start_ops` to 1,000 operations and `decision_time` to 60,000,000 operations.

The results are summarized in Figure 8.5. At first we ran all of our configurations with DARC's

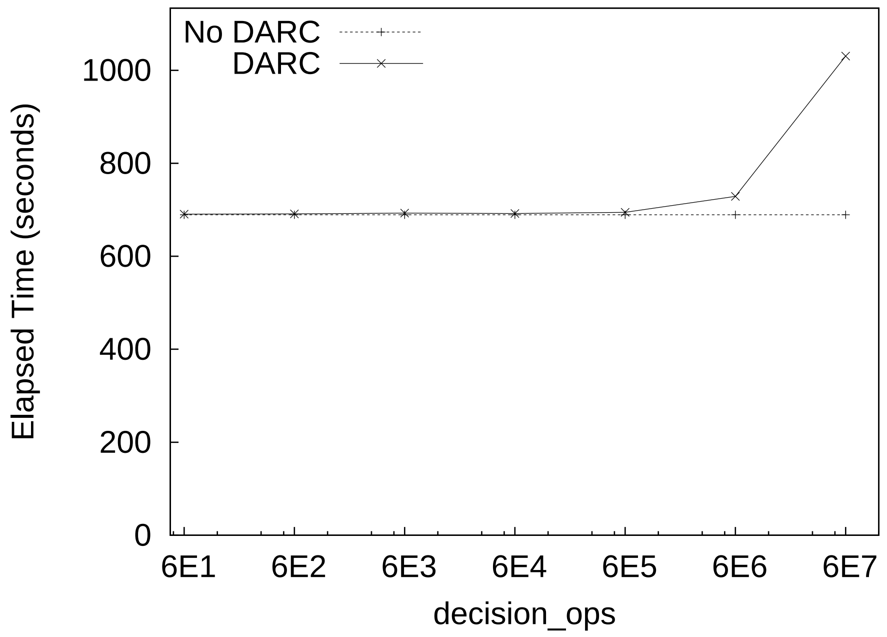


Figure 8.6: The overheads for running DARC with the stat benchmark using different values for the `decision_time` variable. Results without running DARC are shown as a baseline. Note that the x-axis is logarithmic.

analysis being performed asynchronously. The runtime with the `2-kprobes` version of DARC took approximately 5,294 seconds, or about 7.7 times longer than running without DARC. The `rettramp` version reduced the overhead to 5.8 times, and finally the `calltramp` version reduced it to 3.3 times. We refer to the portion of time that is not counted as user or system time as *wait time* (seen as the white portion of the bars). This is the time that the process was not using the CPU. In this case, we hypothesized that the wait time overhead was mostly due to the asynchronous analysis, as the a spinlock was protecting the request queue. This was not affected by the instrumentation optimizations. The system time overhead, however, did improve.

We then ran the version with the `calltramp` optimization with synchronous analysis. We call this configuration `calltramp-sync`. We can see that the wait time overhead is no longer present, and that the system time improved as well. The elapsed time overhead dropped to 49.5%. It is still somewhat high because the `stat` operation returns very quickly—the average latency of a `stat` operation is 2.3 microseconds, and the DARC `calltramp-sync` implementation adds approximately 1.1 microseconds per operation.

It is important to note that these figures depict a worst-case scenario. Under realistic conditions, such as in the use cases presented in Chapter 7, `decision_time` was on the order of tens of operations, whereas here it was on the order of tens of *millions* of operations. DARC is designed to analyze longer-running applications, and the time spent performing the analysis is negligible compared to the application's total run time.

To show how the `decision_time` variable affects overheads, we ran the benchmark with different values for `decision_time`, and kept `start_ops` at 1,000 operations. The results for the `calltramp-sync` implementation are shown in Figure 8.6. The results for DARC with values

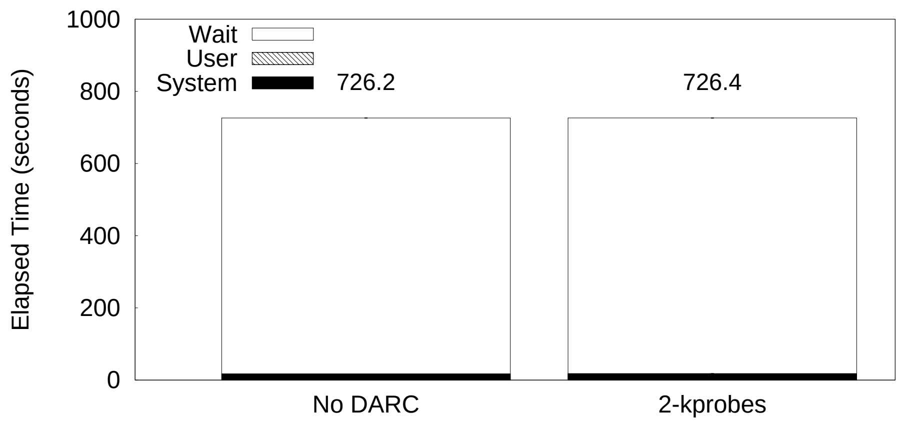


Figure 8.7: Results for the random-read benchmark. Note that error bars are always drawn, but may be too small to see.

of up to 60,000 and the results when not running DARC at all were statistically indistinguishable. In addition, we calculated that DARC analyzed latencies for one second or less for these values. When we set `decision_time` to 6×10^5, 6×10^6, and 6×10^7, the overheads were 0.8%, 5.7%, and 49.5%, respectively (note that the last data point here is the same one depicted in Figure 8.5). For these same values, DARC performed its analysis for approximately 1.5%, 14.1%, and 100% of the total runtime. This shows how the overheads increase as DARC's analysis was prolonged.

The percentage of time that DARC was running may be approximated as:

$$\frac{\frac{elapsed\_time\_full}{total\_operations} \times total\_darc\_operations}{elapsed\_time\_part} \times 100 \tag{8.2}$$

where $elapsed\_time\_full$ is the elapsed time when DARC's analysis is running for the entire duration of the benchmark, $elapsed\_time\_part$ is the elapsed time when `decision_time` is lowered, and $total\_darc\_operations$ is calculated as:

$$(decision\_time \times ftree\_depth) + start\_ops \tag{8.3}$$

DARC reported the same root cause path for all cases:

```
vfs_stat_fd →
    __user_walk_fd →
        do_path_lookup →
            path_walk →
                link_path_walk
```

This shows that DARC has negligible overheads for real-world configurations, and that the analysis can complete in less than one second while remaining sound.

8.4 Random-Read Workload

For the random-read benchmark, we set the start function to the top-level read function in the kernel. There was only a single peak in the profile, and DARC informed us that there was one root cause path consisting of eight functions. Because the benchmark executes 51,200 operations, we set `start_ops` to 1,000 operations, and `decision_time` to 6,500. This allowed DARC to reach the final root cause function without completing the analysis. The root cause path for this benchmark was:

```
vfs_read →
    do_sync_read →
        generic_file_aio_read →
            generic_file_direct_IO →
                ext2_direct_IO →
                    __blockdev_direct_IO →
                        io_schedule
```

The ftree for this benchmark was rather large, with the levels of the ftree having 1, 7, 3, 4, 5, 1, 39, 3, and 28 fnodes, from top to bottom. The results are shown in Figure 8.7. Running the benchmark with and without DARC produced statistically indistinguishable runtimes, regardless of the instrumentation method used, with the elapsed times for all configurations averaging approximately 726 seconds. Therefore, we only present the results for the worst-case `2-kprobes` implementation. There is no distinguishable elapsed time overhead because the overheads are small compared to the time required to read data from the disk. The average read operation latency was approximately 14 milliseconds, and we saw from the `stat` benchmark that DARC adds only a few microseconds to each operation.

Chapter 9

OSprof Profile Comparison Methods

It is often useful to compare two OSprof profiles. DARC performs this comparison when it is restarted to ensure that the runtime environment had not changed (see Section 3.10). Comparing profiles is also essential to the OSprof methodology. A user may collect profiles for dozens of operations at once. It is often useful to compare this set of profiles to another set that were collected under different conditions to analyze some behavior of the system. For example, a profile of one version of a file system or one type of workload may be compared with a profile of a different file system or the same file system under a different workload. This technique is called *differential analysis*. Generally only a few profiles will differ between the two sets, and automatically filtering out the ones that are similar allows the user to focus on important changes.

There are several methods of comparing histograms where only bins with the same index are matched. Some examples are the chi-squared test, the Minkowski form distance [71], histogram intersection, and the Kullback-Leibler/Jeffrey divergence [37]. The drawback of these algorithms is that their results do not take factors such as distance into account because they report the differences between individual bins rather than looking at the overall picture. For example, consider a histogram with items only in bucket 1. In a latency profile, shifting the contents of that bucket to the right by ten buckets would be much different than shifting by one (especially since the scale is logarithmic). These algorithms, however, would view both cases as simply removing some items from bucket 1, and adding some items to another bucket, so they would report the same difference for both. We implemented the chi-square test as a representative of this class of algorithms because it is “the accepted test for differences between binned distributions” [61].

Cross-bin comparison methods compare each bin in one histogram to every bin in the other histogram. These methods include the quadratic-form, match, and Kolmogorov-Smirnov distances [14]. Ideally, the algorithm we choose would compare bins of one histogram with only the relevant bins in the other. These algorithms do not make such a distinction, and the extra comparisons result in high numbers of false positives. We did not test the Kolmogorov-Smirnov distance because it applies only to continuous distributions.

The Earth Mover’s Distance (EMD) algorithm is a goodness-of-fit test commonly used in data visualization [63]. The idea is to view one histogram as a mass of earth, and the other as holes in the ground; the histograms are normalized so that we have exactly enough earth to fill the holes. The EMD value is the least amount of work needed to fill the holes with earth, where a unit of work is moving one unit by one bin. This algorithm does not suffer from the problems associated with the bin-by-bin and the cross-bin comparison methods, and is specifically designed for visualization.

The EMD algorithm can also compare time-lapse profiles, as it was originally intended for inputs that have those dimensions. As we show in Section 9.2, EMD indeed outperformed the other algorithms.

9.1 Implementation

We have implemented several scripts that allow us to compare pairs of individual profiles. Automatic profile-independence tests are useful to select a small subset of operations for manual analysis from a large set of all operations. Also, such tests are useful to verify similarity of two profiles, as DARC does when it resumes analysis from a previous state. Let us call the number of operations in the b^{th} bucket of one profile n_b, and the number of operations in the same bucket of the same operation in another profile m_b. Our goodness-of-fit tests return percent difference D between two profiles:

TOTOPS The degree of difference between the profiles is equal to the normalized difference of the total number of operations:

$$D = \frac{|\sum n_i - \sum m_i|}{\sum n_i} \times 100$$

TOTLAT The degree of difference between the profiles is equal to the normalized difference of the total latency of a given operation:

$$D = \frac{|\sum \frac{3}{2}2^{n_i} - \sum \frac{3}{2}2^{m_i}|}{\sum \frac{3}{2}2^{n_i}} \times 100 = \frac{|\sum 2^{n_i} - \sum 2^{m_i}|}{\sum 2^{n_i}} \times 100$$

CHISQUARE We have implemented the chi-square test as a representative of the class of algorithms that performs bin-by-bin comparisons. It is defined for two histograms as follows:

$$\chi^2 = \sum \frac{(n_i - m_i)^2}{n_i + m_i}$$

The χ value can be mapped to the probability value P between 0 and 1, where a small value indicates a significant difference between the distributions. To match the semantics and scale of the previous two tests, we present $D = (1 - P) \times 100$. We utilized the standard **Statistics::Distributions** Perl library [36] in the implementation.

EARTHMOVER We implemented the calculation of the EMD value as a greedy algorithm. After normalizing the two histograms, one is arbitrarily chosen to represent the mounds of earth, say n, and the other represents the holes in the ground, say m. The algorithm moves from left to right, and keeps track of how much earth is being carried, which is calculated as the amount currently being carried, plus $n_i - m_i$. Note that this value can be negative, which handles the case where the holes appear before the mounds of earth. The absolute value of the carrying value is then added to the EMD value.

We also created several profile comparison methods that combine simple techniques that we use when manually comparing profiles. We distinguish the peaks on the profiles using derivatives. This is the same technique that DARC uses to display the peaks to the user, as described in Chapter 3. If the number of peaks differs between the profiles, or their locations are not similar, the profiles are considered to be different (score of 100). As we will see in Section 9.2, these preparation steps alone significantly decrease the number of incorrectly classified profiles. If, after these preparation steps, the profile analysis is still not over, then we can perform further comparisons based on the previously described algorithms (`TOTOPS`, `TOTLAT`, `CHISQUARE`, and `EARTHMOVER`). We have implemented the following two methods that first examine the differences between peaks, and then compare the profiles using another method if the peaks are similar:

GROUPOPS If the peaks in the profiles are similar, the score is the normalized difference of operations for individual peaks.

GROUPLAT This method is same as `GROUPOPS`, except that we calculate latency differences for individual peaks.

9.2 Evaluation

To evaluate our profile-analysis automation methods we compared the results of the automatic profile comparison with manual profile comparison. In particular, we analyzed 150 profiles of individual operations. We manually classified these profiles into “different” and “same” categories. A false positive (or a type I error) is an error when two profiles are reported different whereas they are same according to the manual analysis. A false negative (or a type II error) is an error when two profiles are reported same whereas they are different according to the manual analysis. Our tests return the profile’s difference value. A difference threshold is the value that delimits decisions of our binary classification based on the test’s return values.

Figures 9.1–9.6 show the dependencies of the number of false positives and false negatives on the normalized difference of the two profiles calculated by the six of the profile comparison methods that we have implemented. As we can see, the EMD algorithm has a threshold region with the smallest error rates of both types, though all algorithms have a point where both error rates were below 5%. However, both our custom-made methods `GROUPOPS` and `GROUPLAT` have a wide range of difference thresholds where both errors are below 5%. This means that these methods can produce reliable and stable results for a wide range of profiles.

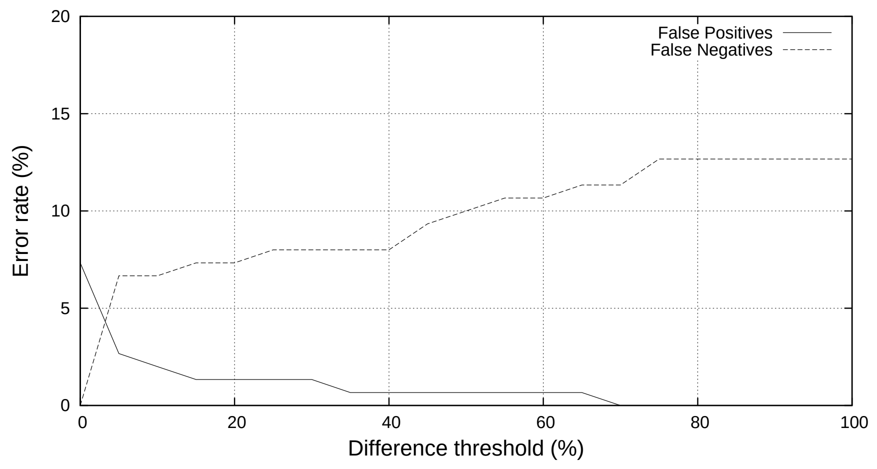


Figure 9.1: `TOTOPS` test results compared with manual profile analysis.

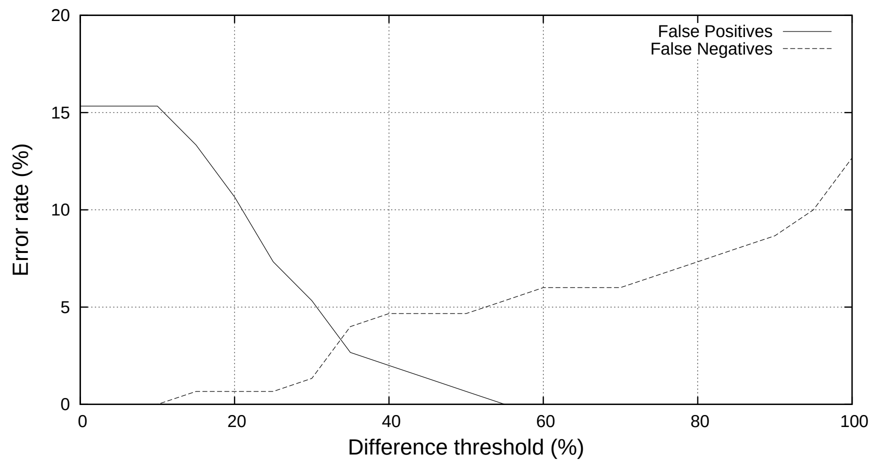


Figure 9.2: `TOTLAT` test results compared with manual profile analysis.

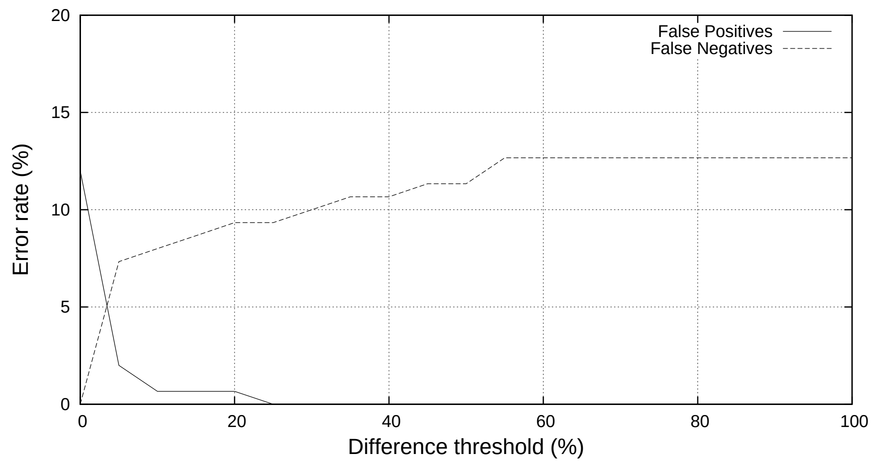


Figure 9.3: `CHISQUARE` test results compared with manual profile analysis.

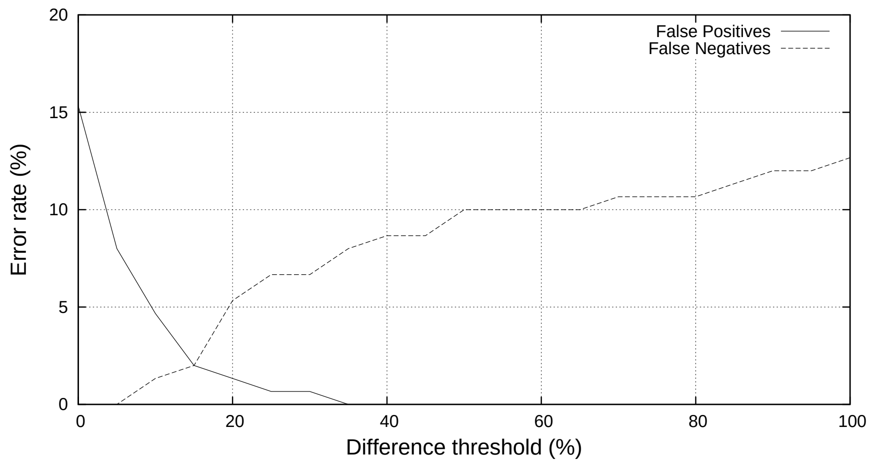


Figure 9.4: EARTHMOVER test results compared with manual profile analysis.

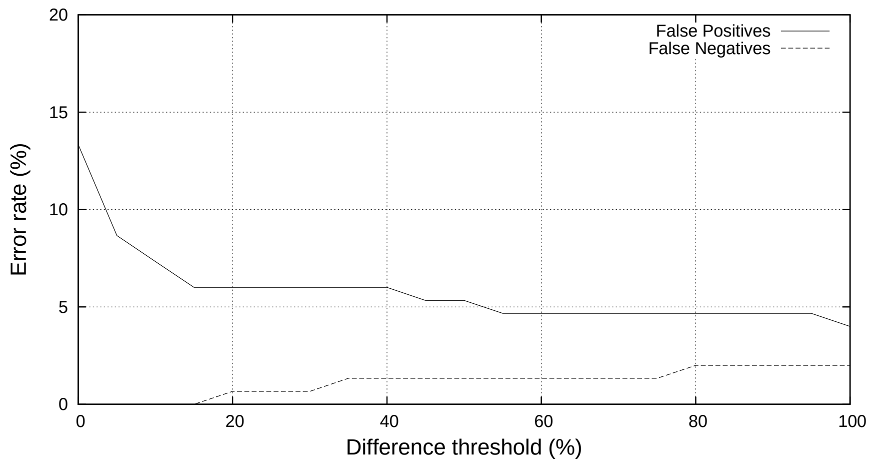


Figure 9.5: GROUPOPS test results compared with manual profile analysis.

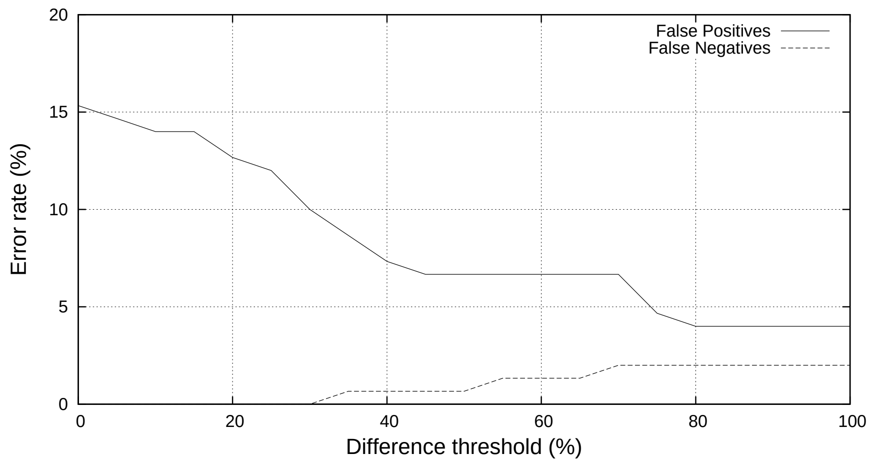


Figure 9.6: GROUPLAT test results compared with manual profile analysis.

Chapter 10

OSprof and DARC in Virtual Machine Environments

Virtual machine technology is becoming more pervasive, mainly being used for server consolidation, and so exploring how OSprof and DARC behave when used in virtual machine environments is important. The suitability of OSprof for use within virtual machines depends mainly on the accuracy of the clock cycle register in virtual machines. DARC's suitability additionally depends on acceptable overheads.

Conceptually, operating system profiling inside of virtual machines is not different from ordinary profiling. However, it is important to understand that the host (underlying) operating system and the virtual machine itself affect the guest operating system's behavior [45]. Therefore, the benchmarking and profiling results collected in virtual machines do not necessarily represent the behavior of the guest operating system running on bare hardware. Other virtual machines running on the same system exacerbate the problem even more.

Nevertheless, there are two situations when profiling in virtual machines is necessary:

1. It is not always possible or safe to benchmark or profile on a real machine directly.

2. Developers of virtual machines developers or systems intended to run in virtual environments naturally benchmark and profile systems running in virtual environments.

These situations have contradicting requirements. In the first case, it is necessary to minimize the influence of virtualization on the guest operating system. In the second case, it is necessary to profile the interactions between the virtual machines as well as their interactions with the host operating system. We will focus on the first case, and describe situations in which the behavior seen in virtual machine environments differs from those seen when running directly on the operating system.

An interesting research direction would be to use VMware's VProbes [79] to extend DARC. These probes can instrument a running virtual machine to collect latency information, allowing DARC to cross the guest-host barrier.

We next describe the setup for the experiments that we ran with VMware Workstation and Xen in Section 10.1. In Section 10.2, we introduce several virtual machine technologies. Then, in Section 10.3, we measure the accuracy of the clock cycle register; and in Section 10.4 we measure the overheads of running DARC in a virtual machine.

10.1 Experimental Setup

The experimental setup for this chapter differs from the one used in Chapters 7 and 8 because we needed more machines to run the benchmarks. We used two identical test machines, one configured with VMware Workstation 6.0.1 [32] (hereafter referred to as WS601) and one with Xen 3.1.0 [5]. The test machines were Dell PowerEdge 1800s with 2.8GHz Intel Xeon processors, 2MB L2 cache, and 1GB of RAM, and a 800MHz front side bus. The machines were equipped with six 250GB, 7,200 RPM Maxtor 7L250S0 SCSI disks. We used one disk as the system disk, one disk for the virtual machine to run on, and the additional disk for the test data.

The operating system was Fedora Core 6, with patches as of October 08, 2007. The system was running a vanilla 2.6.18 kernel and the file system used was ext2. We chose 2.6.18 because that is the default Xen kernel, and we were wary of using the 2.6.23 kernel that we used in the rest of the dissertation because Xen modifies its guest operating systems. We made the kernel configurations for the VMware Workstation and Xen machines as similar as possible, but the two virtual machine technologies require some different configuration options. The differences were mainly in drivers and processor features that do not impact performance.

To aid in reproducing these experiments, the list of installed package versions, workload source code, and kernel configurations are available at
`http://www.fsl.cs.sunysb.edu/docs/darc/`.

10.2 Virtual Machine Technologies

Several types of virtual machine technology exist for x86 systems today. What they all have in common is a virtualization layer called a hypervisor that resides somewhere between the guest operating system and the host's hardware. The hypervisor virtualizes components such as CPUs, memory, and I/O devices. We focus on CPU virtualization here, as this has the largest effect on our results.

One distinction between hypervisors is the location of the hypervisor in the hardware-software stack. With a *hosted* hypervisor, the virtualization layer is run as an application inside the host's operating system. A hosted hypervisor is employed by VMware in their Workstation, Player, and ACE products [78], as well as by Microsoft Virtual Server [46], Parallels Desktop, and Parallels Workstation [58]. On the other hand, a native, or "bare-metal" hypervisor is located directly on the hardware, with no host operating system in between. Examples of these include Xen [82] and VMware's ESX Server. Guest operating systems running on bare-metal hypervisors are generally faster than those running on hosted hypervisors, because there are fewer layers between the hardware and the guest operating system.

A second feature that distinguishes hypervisors is how they handle privileged instructions that are generated by a guest operating system. VMware products historically employed binary translation to translate any privileged code into unprivileged code or code that jumps into the hypervisor to emulate the instruction. This allows guest operating systems to run unmodified. Xen, on the other hand, historically used *paravirtualization*. With paravirtualization, guest operating systems are modified to use calls to the hypervisor (hypercalls) in place of privileged instructions. Recently, CPU manufacturers have extended their instruction sets to aid virtualization [1, 26]. Xen can now utilize these instructions to run unmodified guest operating systems. VMware's ESX server uses these new instructions in some cases as well.

Linux kernels after version 2.6.20 include `kvm`, the Kernel-based Virtual Machine, which utilizes the virtualization provided by these CPUs [35]. However, we did not benchmark `kvm` because we had to use kernel version 2.6.18 because of Xen. In addition, `kvm` was still marked as "experimental" even in the newer 2.6.23 kernel that we used in the remainder of the dissertation.

We cannot make any fair comparisons between VMware Workstation and Xen when presenting our results because they employ different virtualization technologies (hosted hypervisor with binary translation vs. bare-metal hypervisor with a modified guest operating system). Instead, our goal is to analyze how OSprof and DARC perform on each one.

10.3 Clock Counter Accuracy

OSprof and DARC both rely on the CPU's clock counter read instruction (RDTSC on x86), so we conducted an experiment to determine how accurate this instruction is on the virtual machines. With Xen, we always used the standard instruction. With WS601, however, we tried two alternatives. First, we used the guest operating system's (apparent) time, which is the same CPU clock counter read instruction that we normally use. This would allow timing information to reflect the amount of time the virtual machine actually received because the virtual TSC register should increment only when the guest is allowed to run. Unfortunately, this does not provide I/O isolation and depends on the quality of the clock counter virtualization. Second, we used the host operating system's (wall clock) time, by specifying

monitor_control.virtual_rdtsc = false

in WS601's configuration file [76, 77]. This allows us to include the CPU time spent not executing the guest.

Figure 10.1 shows user-mode profiles of an idle-loop workload generated by one process and captured with four different configurations. This workload consists of a loop that gets executed 100,000,000 times. The loop only reads the clock cycle counter twice and plots the difference in an OSprof profile. We would expect most of the profile to be on the very left, because no work is being done between the clock counter readings. The top-most profile of Figure 10.1, labeled "Host" shows this workload as it runs on the host. Most of the events fell into the first few buckets, as expected. We also have some events in buckets 9–19, which are due to clock and disk interrupts, as described in Section 7.1.

The second profile shown in Figure 10.1, labeled "Xen" is the same workload running on Xen. This profile is very similar to that of running on the host, which tells us that using the clock counter is not a problem for Xen. The third profile, labeled "WS601_Apparent" is for the workload running on WS601 using the apparent, or virtualized, clock counter. We can see that there is a large difference here due to WS601's virtualization. This difference is not present in the fourth profile, labeled "WS601_Real," which shows WS601 using the host operating system's clock counter. We therefore recommend turning off the `virtual_rdtsc` option when using OSprof and DARC in VMware Workstation. In the remainder of this chapter, we turn off clock counter virtualization.

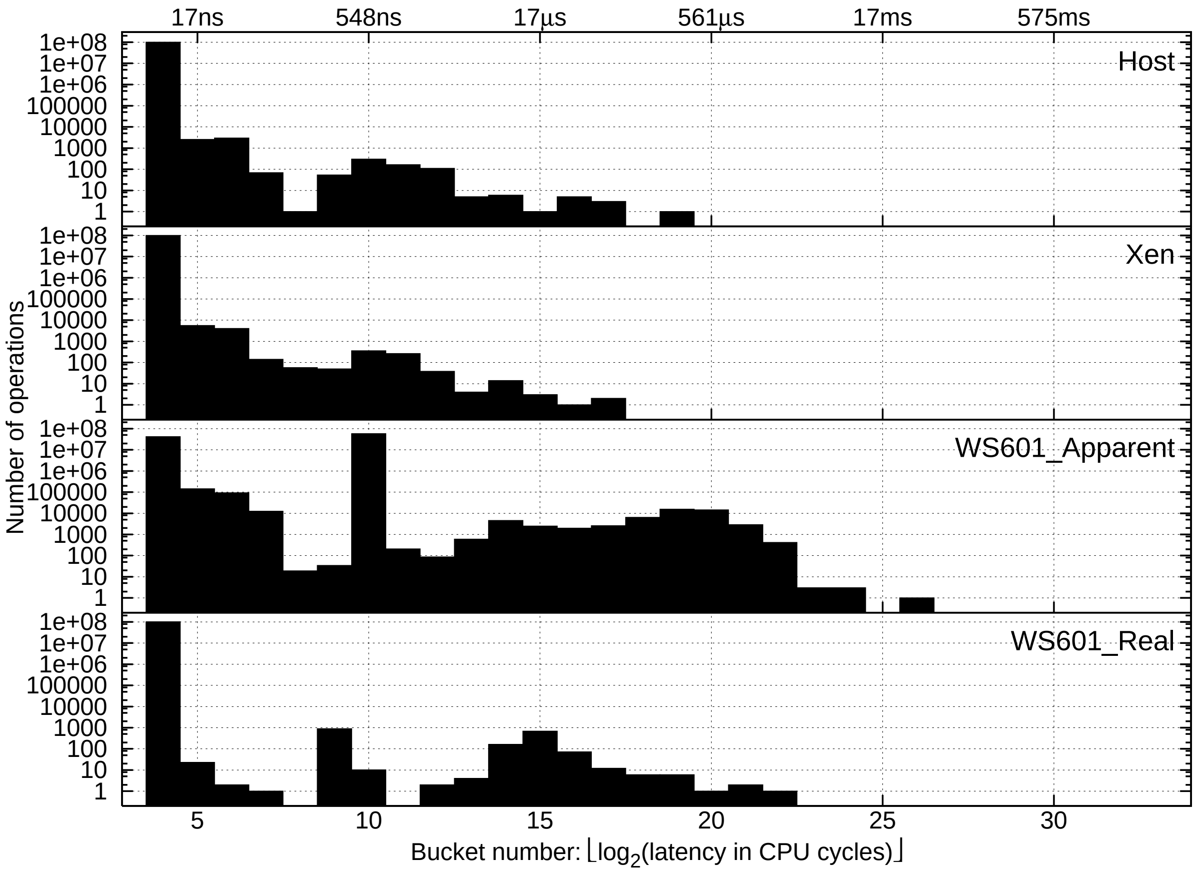


Figure 10.1: Idle-loop profiles captured on the host, on Xen, on VMware Workstation 6.0.1 captured using the guest operating system' (apparent) time, and on VMware Workstation 6.0.1 captured using host operating system's real time.

10.4 DARC Overheads

To determine how DARC behaves in virtual environments, we decided to run the synthetic workload described in Section 8.2 on both WS601 and Xen. We chose this benchmark because it provides us with data describing how the fanout and depth of the tree affect performance. Additionally, we used the tree that crosses the user-kernel boundary, because this shows us how DARC behaves both in user-space and in the kernel. For all tests, we used the `calltramp` implementation of DARC that performs synchronous analysis. Our main concerns were the overheads imposed by virtualization and the interrupt-handling capabilities of WS601 and Xen (each call site that is instrumented with a suprobe requires four interrupts). To allow for fair comparisons between running on virtual machines and directly on hardware, we present the results for this benchmark with no virtualization on the same machines used in this chapter in Figure 10.2.

Figure 10.3 shows the results for the workload running on WS601. Note that the shape of the graph is very similar to the one depicting this workload running without virtualization (Figure 8.4):

Levels 0–1: There is a large initial increase due to the suprobes and DARC analysis code.

Levels 1–4: There is a more gradual slope that reflects one additional instrumented call site per

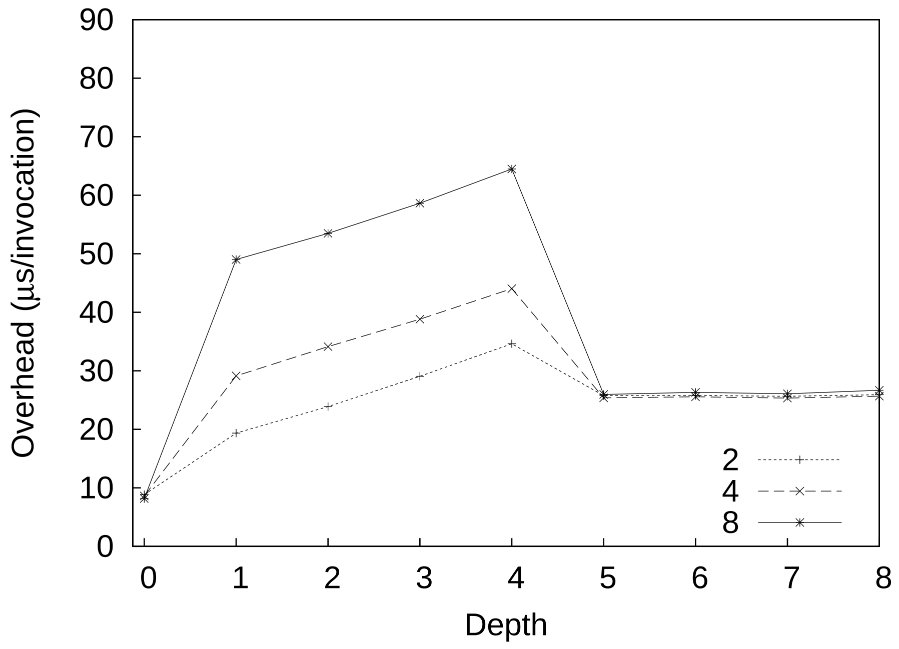


Figure 10.2: Results for the synthetic workload with trees of fanouts 2, 4, and 8 that cross the user-kernel boundary with no virtualization.

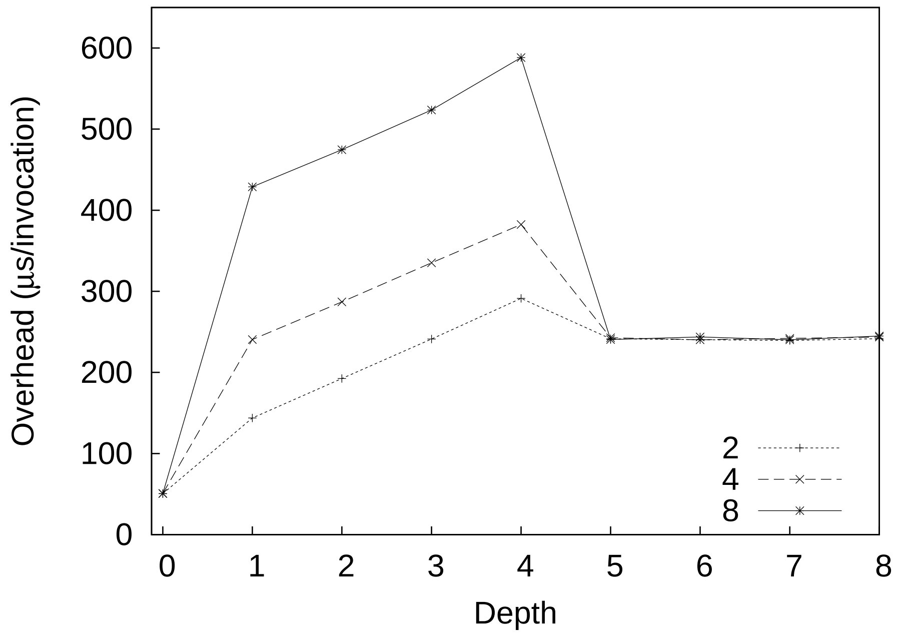


Figure 10.3: Results for the synthetic workload with trees of fanouts 2, 4, and 8 that cross the user-kernel boundary in VMware Workstation 6.0.1.

		Depth								
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Fanout	**2**	16	18	18	18	18	18	18	18	18
	4	16	18	18	18	18	18	18	18	18
	8	16	18	18	18	18	18	18	18	18

Table 10.1: The minimum OSprof bucket numbers whose elements do not shift on average due to DARC's overheads when running the synthetic workload that crosses the user-kernel boundary in VMware Workstation 6.0.1. The analysis here is performed synchronously.

		Depth								
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Fanout	**2**	15	16	16	17	17	17	17	17	17
	4	15	17	17	17	17	17	17	17	17
	8	15	17	17	18	18	17	17	17	17

Table 10.2: The minimum OSprof bucket numbers whose elements do not shift on average due to DARC's overheads when running the synthetic workload that crosses the user-kernel boundary in Xen. The analysis here is performed synchronously.

level.

Levels 5–8: The graph is fairly flat during this interval, and the results are not affected much by the fanout. This is because the only suprobes in place at this point are along the root cause path in user-space, and so we have the same number of suprobes regardless of fanout. In the kernel portion, DARC uses call trampolines, which are significantly faster than suprobes, and so the incline of the graph is hard to see.

The fact that the shape of the graph is not affected by virtualization indicates to us that DARC's behavior is not affected significantly when running in WS601. However, we note that the overheads are rather high—we can see that the benchmark runs several times slower on WS601. Table 10.1 shows the minimum OSprof bucket numbers whose elements do not shift due to DARC's overheads when running in WS601. We can see that the bucket numbers are now in the range of I/O operations, meaning that analyzing faster CPU-bound operations can be difficult. In fact, we had to increase the latency of the operation being executed in the benchmark so that DARC could reliably locate the peak. However, most of this overhead is due to suprobes, and so we expect that implementing some of the optimizations discussed in Section 8.2 will remedy this problem.

We present a similar evaluation for DARC running on Xen in Figure 10.4 and Table 10.2. Once again, we see a similar shape for the graph, but the overheads are now close to what we saw for DARC running without virtualization. This is because Xen's hypervisor is close to the hardware, while WS601's hypervisor is a process running on the host operating system.

We can conclude that DARC's general operation is not inhibited by virtualization. In cases where user-space code is being analyzed, the higher overhead of suprobes combined with the overheads of VMware Workstation 6.0.1 make it difficult to analyze some low-latency behaviors. Porting DARC's kernel optimizations to user space will mitigate these problems.

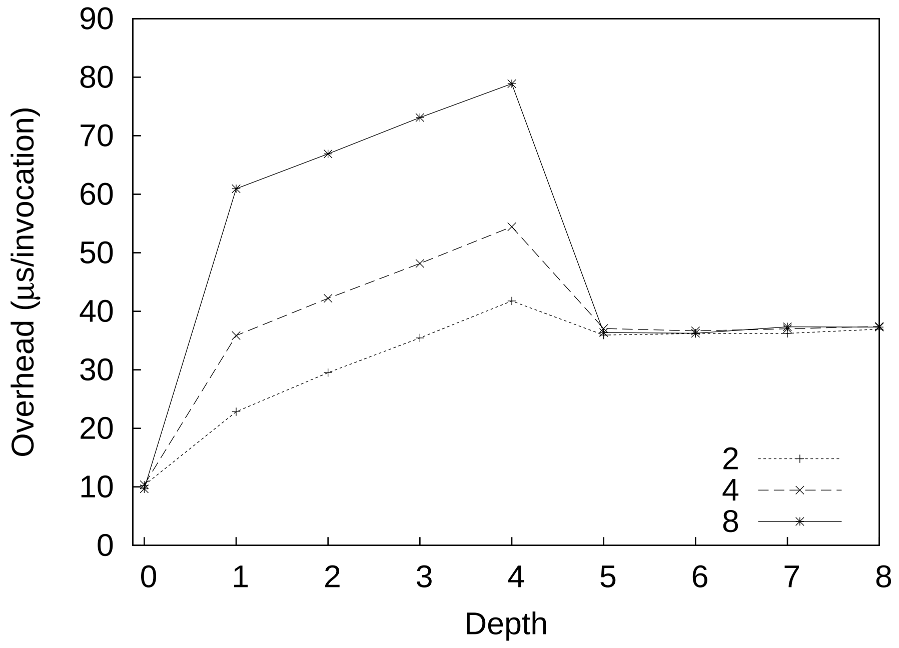


Figure 10.4: Results for the synthetic workload with trees of fanouts 2, 4, and 8 that cross the user-kernel boundary in Xen.

Chapter 11

Related Work

Previous work in this area has focused on dynamic binary instrumentation (DBI) frameworks, using call-paths as a unit for metric collection, and using DBI to investigate bottlenecks. We discuss each of these topics in turn.

11.1 Dynamic Binary Instrumentation

One method for performing binary instrumentation is static binary instrumentation, where the binary is instrumented before it is executed [18, 21, 39, 62, 70]. Tools also exist that modify binaries to optimize performance [3, 7, 15]. However, we focus on dynamic binary instrumentation, as that is the method that our DARC implementation uses.

Many dynamic binary instrumentation frameworks exist today that modify user-space applications. We discuss them in an approximate chronological order. Shade [16], is an important earlier work that influenced other DBI frameworks, but is now obsolete. On Windows, the Detours library can be used to insert new instrumentation into arbitrary Win32 functions during program execution [25]. It implements this by rewriting the target function images. Vulcan [69] is another DBI framework for Windows, but is only used internally at Microsoft. Strata [64, 65], DELI [20], DynamoRIO [8], and DIOTA [41, 42] can perform both dynamic binary instrumentation and optimization. The Valgrind framework [52–54, 66], available for Linux on x86 architectures, is widely-used, and has been utilized to create several checkers and profilers. We did not consider using these frameworks for DARC because they do not have counter-parts in the kernel, which would necessitate separate user-space and kernel implementations of DARC.

The `ptrace` interface allows for limited user-space application modification. Although it does not actually modify the binary, the process-tracing facility allows a *monitor* to intercept and modify system calls and signals [23]. However, the overheads for using `ptrace` are rather restrictive because of an increased number of context switches for system calls. This was addressed by expanding the interface [68, 80].

Our user-space instrumentation framework, suprobes, is a modified version of uprobes [56, 57]. We describe both in Section 4.2.3.

For kernel instrumentation, users may use kprobes on Linux (described in Chapter 4). Systemtap [60] allows users to write instrumentation scripts that get translated into kernel modules that use kprobes (and soon, uprobes as well). This makes kprobes and uprobes easier to use, although it also restricts what can be done with them.

Some frameworks exist that allow users to instrument both user-space and kernel-space binaries. For example, Sun's Dtrace [13, 47] allows users to insert instrumentation at pre-defined hooks or user-specified locations in both user programs and in the Solaris kernel. Dyninst [9] allows users to instrument user applications on a variety of architectures, and Kerninst [72, 73], which was developed by the same laboratory, allows users to instrument a running kernel. These two frameworks can be used together to instrument an entire running system [49]. Finally, another pair of frameworks also allow for whole-system instrumentation. Pin [40] can be used to instrument user programs, and PinOS [10] can instrument kernels. The latter works by having the kernel run in a virtual machine.

11.2 Call-Path Profiling

Others have explored using call-paths as a main abstraction for performance profiling. These projects have also utilized dynamic binary instrumentation for their profiling. PP [4] implements an algorithm for path profiling, which counts the execution frequencies of all intra-procedural acyclic paths by encoding the paths as integers. This work was extended to use hardware metrics rather than relying on execution frequencies and to use a *calling context tree* (CCT) to store metrics [2]. The CCT is similar to our ftree, but has a bounded size because it does not contain multiple entries for loops, and it does not differentiate between a function calling another function multiple times. Our ftree can afford to be contain these extra nodes because it contains few paths, rather than an entire call-tree, and is also bounded by the user-specified maximum depth. PP was also extended to handle inter-procedural paths [38, 44].

The TAU parallel performance system [67] has various profiling, tracing, and visualization functionality to analyze the behavior of parallel programs. One mode of operation which is similar to DARC is call-path profiling. Here, TAU generates its own call stack by instrumenting all function calls and returns. This method handles both indirect and recursive calls. This stack is used to provide profiling data that is specific to the current call-path. An extension to TAU, called KTAU [50], was created to supplement TAU with latency information from the kernel. However, KTAU only collects simple latency measurements from pre-defined locations in the kernel and uses source instrumentation, so a meaningful quantitative comparison could not be made here either.

CATCH associates hardware metrics with call-path information for MPI and OpenMP applications [19]. It builds a static call-graph for the target application before it is executed. CATCH uses a method similar to that of DARC to keep track of the current node that is executing, but uses loops in its tree for recursive programs. It is also possible for users to select subtrees of the call-graph to profile rather than tracking the entire execution of the program. CATCH cannot cope with applications that use a function name for more than one function, and cannot profile applications that use indirect calls.

A major difference between these projects and DARC is that DARC performs call-path *filtering*, rather than call-path *profiling*. This means that DARC instruments only the portion of the code that is currently being investigated, rather than the entire code-base. Additionally, it generally runs for a shorter period of time. However, DARC also collects less information than profiling tools.

Another project, iPath, provides call-path profiling by instrumenting only the functions that are of interest to the user [6]. Whereas DARC searches for the causes of behavior seen in higher-level functions, iPath analyzes lower-level functions and distinguishes latencies based on the call-paths used to reach them. It does so by walking the stack to determine the current call-path, sampling the desired performance metric, and then updating the profile for that call-path with the performance data. This provides two main benefits. First, overheads are incurred only for functions that the user is profiling. Second, iPath walks the stack, so it does not need any special handling for indirect or recursive calls. The main problem with performing stack walks is that they are architecture and compiler-dependent. There are some compiler optimizations that iPath cannot cope with, and it would be difficult to port iPath to use other compilers and optimizations. In contrast, DARC's method relies more on simple and portable dynamic binary instrumentation techniques.

Combining call-path profiling with sampling, `csprof` samples the running program's stack periodically, and attributes a metric to the call-path [22]. It also introduces a technique to limit the depth of the stack walk when part of that stack has been seen before. However, although the stack walk is more efficient than that of iPath, `csprof` is also tied to the code of a specific compiler and its optimizations.

11.3 Dynamic Bottleneck Investigation

Kperfmon [73] is a tool that uses the Kerninst [72] dynamic instrumentation framework. For a given function or basic block, Kperfmon can collect a metric, such as elapsed time, processor time, or instruction cache misses. A user may search for a root cause by examining the results and running Kperfmon again to measure a new section of code.

CrossWalk [49] combines user-level [9] and kernel-level [72] dynamic instrumentation to find CPU bottlenecks. Starting at a program's *main* function, CrossWalk performs a breadth-first search on the call-graph for functions whose latency is greater than a pre-defined value. If the search enters the kernel via a system call, the search will continue in the kernel. It is not clear if CrossWalk can handle multiple paths in the call-graph. CrossWalk does not handle multi-threaded programs, asynchronous kernel activities, or recursion. It does not perform call-path or PID filtering.

Paradyn [48] uses the Dyninst dynamic instrumentation framework [9] to find bottlenecks in parallel programs. It does this using a pre-defined decision tree of bottlenecks that may exist in programs. Paradyn inserts code to run experiments at runtime to determine the type of bottleneck and where it is (synchronization object, CPU, code, etc.). Instances when continuously measured values exceed a fixed threshold are defined as bottlenecks. Paradyn can narrow down bottlenecks to user-defined phases in the program's execution. Paradyn's original code-search strategy was replaced by an approach based on call-graphs [11].

One difference between the two code search strategies in Paradyn is that the original used *exclusive* latencies and the new strategy used *inclusive* latencies, because they are faster and simpler to calculate. To calculate the exclusive latency of a function, Paradyn stopped and started the timer so that the latencies of the function's callees would not be included. DARC can use exclusive latencies because the latencies of the callees are already being calculated, so it does not add much overhead. Paradyn's search strategy was also changed. Originally, Paradyn first attempted to isolate a bottleneck to particular modules, and then to particular functions in those modules. They did this by choosing random modules and functions to instrument [11]. This was replaced by a method that began at the start of the executable, and continued to search deeper in the call-graph as long as the bottleneck was still apparent.

DARC is both more flexible and more accurate than these solutions. It has the ability to search for the causes of any peak in an OSprof profile, rather than checking only for pre-defined bottle-necks. Additionally, these methods would not be suitable for finding the causes of intermittent behavior. DARC also introduces several new features, such as call-path filtering, distinguishing recursive calls, resuming searches, and investigating asynchronous events.

Chapter 12

Conclusions

We designed DARC, a new performance-analysis method that allows a user to easily find the causes of a given high-level behavior that is seen in an OSprof profile. DARC allows the user to analyze a large class of programs, including those containing recursive and indirect calls. Short-lived programs and programs with distinct phases can be analyzed easily using DARC's resume feature. Access to more source code information allows the user to use DARC to analyze preemptive behavior and asynchronous paths. DARC minimizes false positives through the use of PID and call-path filtering.

Our DARC implementation can be used to analyze source trees that reside in the kernel, in user-space, and those that originate in user-space and cross the user-kernel boundary. The overheads when analyzing high-latency operations, such as disk reads, were statistically insignificant. For faster operations, such as retrieving in-memory file information, the runtime with DARC can increase by up to 50%. However, these overheads are imposed only for the time that DARC is analyzing the code. DARC is designed for analyzing long-running applications, and the period of time that this overhead is incurred is negligible compared to the overall run time. In the benchmark exercising a fast operation, described in Section 8.3, we showed that DARC required less than one second to perform its analysis. This was also seen in all of the use cases that we presented. In addition, DARC's overheads did not affect the analysis in any case.

We have shown how DARC can be used to analyze behaviors that were previously more difficult to explain. These cases include preemptive behavior, asynchronous paths, and intermittent behavior. Whereas OSprof was generally helpful for users to guess about the causes of these behaviors, DARC provided more direct evidence, while requiring less time, expertise, and intuition.

12.1 Future Work

In this section we describe four possible future research directions for our root cause analysis method.

1. Our DARC implementation currently does not include basic block instrumentation. The ability to identify basic blocks would allow DARC to narrow down root causes to basic blocks and minimize any perturbations caused by the instrumentation. Adding this ability would add to DARC's usefulness, but the biggest challenge would be implementing this functionality in DARC's disassembler.

2. Overheads can be reduced by further optimizing the instrumentation. For example, as described in Section 8.2, the kprobe at the start of f_0 can be removed by making a copy of the function that includes the instrumentation, and jumping to this new function from the original. Additionally, the kernel instrumentation optimizations can be made to work for the user-space instrumentation as well.

3. Our current DARC implementation can cross the user-kernel boundary. Future DARC implementations may benefit from the ability to cross the guest-host boundary in virtual machine environments (described in Chapter 10) and the ability to cross the client-server boundary in client-server environments.

4. Users may not always be aware of abnormal performance behavior in running systems, and, by the time they notice it, it may be too late to analyze. To remedy this situation, DARC can be made to run automatically. OSprof profiles can be collected and automatically analyzed over time for abnormal behavior using an algorithm such as the Earth Mover's Distance (see Chapter 9). If some abnormal behavior is detected, DARC can be made to run automatically to provide an administrator with information about the detected behavior.

Bibliography

[1] Advanced Micro Devices, Inc. Industry leading virtualization platform efficiency, 2008. `www.amd.com/virtualization`.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, June 1997. ACM Press.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000)*, pages 1–12, Vancouver, Canada, June 2000. ACM.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, Paris, France, December 1996. IEEE.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, Bolton Landing, NY, October 2003. ACM SIGOPS.

[6] A. R. Bernat and B. P. Miller. Incremental call-path profiling. *Concurrency and Computation: Practice and Experience*, 19(11):1533–1547, 2007.

[7] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Austin, TX, December 2001. ACM.

[8] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*, pages 265–276, San Fransisco, CA, March 2003.

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

[10] P. P. Bungale and C. Luk. PinOS: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the Third International Conference on Virtual Execution Environments (VEE '07)*, pages 137–147, San Diego, CA, June 2007. USENIX Association.

[11] H. W. Cain, B. P. Miller, and B. J. N. Wylie. A callgraph-based search strategy for automated performance diagnosis. In *Proceedings of the 6th International Euro-Par Conference*, pages 108–122, Munich, Germany, August-September 2000. Springer.

[12] S. Callanan, D. J. Dean, and E. Zadok. Extending GCC with modular GIMPLE optimizations. In *Proceedings of the 2007 GCC Developers' Summit*, Ottawa, Canada, July 2007.

[13] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 15–28, 2004.

[14] Chakravarti, Laha, and Roy. *Handbook of Methods of Applied Statistics, Volume I*. John Wiley and Sons, 1967.

[15] W. Chen, S. Lerner, R. Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the 3rd ACMWorkshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Monterey, CA, December 2000. ACM.

[16] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1994)*, pages 128–137, Nashville, TN, May 1994. ACM.

[17] W. Cohen. Gaining insight into the Linux kernel with kprobes. *RedHat Magazine*, March 2005.

[18] O. Cole. Aprobe: A non-intrusive framework for software instrumentation. `www.ocsystems.com/pdf/AprobeTechnologyOverview.pdf`, 2004.

[19] L. DeRose and F. Wolf. CATCH - a call-graph based automatic tool for capture of hardware performance metrics for MPI and OpenMP applications. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 167–176, Paderborn, Germany, August 2002. Springer-Verlag.

[20] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: A new runtime control point. In *Proceedings of the 35th Annual Symposium on Microarchitecture (MICRO35)*, pages 257–270, Istanbul, Turkey, November 2002.

[21] Manel Fernández and Roger Espasa. Dixie: A retargetable binary instrumentation tool. In *Proceedings of theWorkshop on Binary Translation*, Newport Beach, CA, October 1999. IEEE.

[22] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 81–90, Cambridge, MA, June 2005. ACM Press.

[23] M. Haardt and M. Coleman. *ptrace(2)*. Linux Programmer's Manual, Section 2, November 1999.

[24] M. Hiramatsu. Overhead evaluation about kprobes and djprobe (direct jump probe). `http://lkst.sourceforge.net/docs/probes-eval-report.pdf`, July 2005.

[25] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[26] Intel Corporation. Intel virtualization technology (Intel VT), 2008. `www.intel.com/technology/virtualization/`.

[27] N. Joukov, R. Iyer, A. Traeger, C. P. Wright, and E. Zadok. Versatile, portable, and efficient OS profiling via latency analysis. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, October 2005. ACM Press. Poster presentation, `http://doi.acm.org/10.1145/1095810.1118607`.

[28] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating system profiling via latency analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[29] N. Joukov, A. Traeger, C. P. Wright, and E. Zadok. Benchmarking file system benchmarks. Technical Report FSL-05-04, Computer Science Department, Stony Brook University, December 2005. `www.fsl.cs.sunysb.edu/docs/fsbench/fsbench.pdf`.

[30] N. Joukov, C. P. Wright, and E. Zadok. FSprof: An in-kernel file system operations profiler. Technical Report FSL-04-06, Computer Science Department, Stony Brook University, November 2004. `www.fsl.cs.sunysb.edu/docs/aggregate_stats-tr/aggregate_stats.pdf`.

[31] N. Joukov and E. Zadok. Adding secure deletion to your favorite file system. In *Proceedings of the third international IEEE Security In Storage Workshop (SISW 2005)*, pages 63–70, San Francisco, CA, December 2005. IEEE Computer Society.

[32] A. S. Kale. VMware: Virtual machines software. `www.vmware.com`, 2001.

[33] J. Keniston. updated uprobes patches, May 2007. `http://sourceware.org/ml/systemtap/2007-q2/msg00328.html`.

[34] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In *Proceedings of the 2007 Linux Symposium*, pages 215–224, Ottawa, Canada, June 2007. Linux Symposium.

[35] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)*, volume 1, pages 225–230, Ottawa, Canada, June 2007.

[36] M. Kospach. *Statistics::Distributions - Perl module for calculating critical values and upper probabilities of common statistical distributions*, December 2003.

[37] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, March 1951.

[38] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 259–269, Atlanta, GA, May 1999. ACM Press.

[39] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)*, pages 291–300, La Jolla, CA, June 1995. ACM.

[40] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 190–200, Chicago, IL, June 2005. ACM Press.

[41] J. Maebe and K De Bosschere. Instrumenting self-modifying code. In *Proceedings of the Fifth International Workshop on Automated and Algorithmic Debugging (AADEBUG2003)*, Ghent, Belgium, September 2003.

[42] J. Maebe, M. Ronsse, and K De Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, September 2002.

[43] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu. Probing the guts of kprobes. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 109–124, Ottawa, Canada, July 2006.

[44] D. Melski and T. Reps. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 47–62, Amsterdam, The Netherlands, March 1999. Springer.

[45] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the First International Conference on Virtual Execution Environments (VEE '05)*, pages 13–23, Chicago IL, June 2005. USENIX Association.

[46] Microsoft Corporation. *Windows Server Virtualization - An Overview*, 2006. `www.microsoft.com/windowsserversystem/virtualserver/techinfo/virtualization.mspx`.

[47] Sun Microsystems. Solaris dynamic tracing guide. `docs.sun.com/app/docs/doc/817-6223`, January 2005.

[48] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[49] A. V. Mirgorodskiy and B. P. Miller. CrossWalk: A tool for performance profiling across the user-kernel boundary. In *Proceedings of PARCO 2003*, pages 745–752, Dresden, Germany, September 2003. Elsevier.

[50] A. Nataraj, A. Malony, S. Shende, and A. Morris. Kernel-level measurement for integrated parallel performance views: the KTAU project. In *Proceedings of the 2006 IEEE Conference on Cluster Computing*, Barcelona, Spain, September 2006. IEEE.

[51] C. Nentwich and M. Tiihonen. Intel2gas. `http://www.niksula.hut.fi/~mtiihone/intel2gas/`, 2000.

[52] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, United Kingdom, November 2004.

[53] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 89(2), 2003.

[54] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, CA, USA, June 2007. ACM.

[55] S. Oshima. Re: djprobes status, September 2006. `http://www.ecos.sourceware.org/ml/systemtap/2006-q3/msg00517.html`.

[56] P. S. Panchamukhi. [0/3] Kprobes: User space probes support, March 2006. `http://lkml.org/lkml/2006/3/20/2`.

[57] P. S. Panchamukhi. [RFC] [PATCH 0/6] Kprobes: User-space probes support for i386, May 2006. `http://lkml.org/lkml/2006/5/9/25`.

[58] Parallels, Inc. Parallels - virtualization and automation software for desktops, servers, hosting, SaaS. `www.parallels.com`, 2008.

[59] V. Patkov. Hacker disassembler engine. `http://vx.netlux.org/vx.php?id=eh04`, 2007.

[60] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Linux Symposium*, pages 49–64, Ottawa, Canada, July 2005. Linux Symposium.

[61] W. H. Press, S. A. Teukolskey, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2002.

[62] 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 1–7, Seattle, WA, August 1997. USENIX Association.

[63] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *Proceedings of the Sixth International Conference on Computer Visin*, pages 59–66, Bombay, India, January 1998.

[64] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. In *Proceedings of the Workshop on Binary Translation (WBT '01)*, Barcelona, Spain, September 2001.

[65] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*, pages 36–47, San Fransisco, CA, March 2003.

[66] J. Seward, N. Nethercote, and J. Fitzhardinge. Valgrind. `http://valgrind.kde.org`, August 2004.

[67] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[68] R. P. Spillane, C. P. Wright, G. Sivathanu, and E. Zadok. Rapid file system development using ptrace. In *Proceedings of the Workshop on Experimental Computer Science (EXPCS 2007), in conjunction with ACM FCRC*, page Article No. 22, San Diego, CA, June 2007.

[69] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[70] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*, pages 196–205, Orlando, FL, June 1994. ACM.

[71] M. J. Swain and D. H. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

[72] A. Tamches. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. PhD thesis, University of Wisconsin-Madison, 2001.

[73] A. Tamches and B. P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications*, 13(3):263–276, Fall 1999.

[74] A. Traeger, I. Deras, and E. Zadok. DARC: Dynamic analysis of root causes of latency distributions. In *Proceedings of the 2008 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2008)*, pages 277–288, Annapolis, MD, June 2008. ACM.

[75] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80, May 2008.

[76] VMware, Inc. *Timekeeping in VMware Virtual Machines*, 2005. `www.vmware.com/pdf/vmware_timekeeping.pdf`.

[77] VMware, Inc. *Performance Tuning and Benchmarking Guidelines for VMware Workstation 6*, 2007. `www.vmware.com/pdf/WS6_Performance_Tuning_and_Benchmarking.pdf`.

[78] VMware, Inc. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*, 2007. `www.vmware.com/files/pdf/VMware_paravirtualization.pdf`.

[79] VMware, Inc. *VProbes Reference Guide*, 2007. `http://www.vmware.com/products/beta/ws/vprobes_reference.pdf`.

[80] C. P. Wright. *Extending ACID Semantics to the File System via ptrace*. PhD thesis, Computer Science Department, Stony Brook University, May 2006. Technical Report FSL-06-04.

[81] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 175–187, Anaheim, CA, April 2005. USENIX Association.

[82] XenSource, Inc. *Xen Architecture Overview*, 2008. `http://wiki.xensource.com/xenwiki/XenArchitecture?action=AttachFile\&do=get\&target=Xen+Architecture_Q1+2008.pdf`.