CrossWalk: A Tool for Performance Profiling Across the User-Kernel Boundary

Alexander V. Mirgorodskiy and Barton P. Miller
{mirg,bart}@cs.wisc.edu
Computer Sciences Department
University of Wisconsin
Madison, Wisconsin, 53706, USA

Performance profiling of applications is often a challenging task. One problem in the analysis is that applications often spend significant amount of their time in the operating system. As a result, conventional user-level profilers can, at best, trace the performance bottleneck to a particular system call. This information is often insufficient for the programmer to deal with the performance problem—system calls can be quite complex, making it hard to pinpoint the cause of the problem.

To address this deficiency, we have designed a tool called CrossWalk that is able to do performance analysis across the kernel boundary. CrossWalk starts profiling at the user level, profiles the main function, then its callees and walks further down the application call graph, refining the performance problem to a particular function. If it determines that this function is a system call, it walks into the kernel code and starts traversing the kernel call graph until it locates the ultimate bottleneck.

The key technologies in CrossWalk are dynamic application instrumentation and dynamic kernel instrumentation. For the former, we use an existing library called Dyninst API. For the latter, we have designed a new framework called Kerninst API with an interface modeled after Dyninst. When combined, the two libraries provide a unified and powerful interface for building cross-boundary tools.

We demonstrate the usefulness of the cross-boundary approach by analyzing the Squid proxy server with CrossWalk. By drilling down into the kernel, we were able to identify the ultimate cause of Squid's performance problems and remove them by modifying the application's source code.

1. Introduction

Many applications make heavy use of functions provided by the operating system. Naturally, the performance of such applications depends on how they make use of these functions and how efficiently these functions are implemented by the operating system. For example, I/O is key to the efficiency of many high-performance applications. Network performance is often the constraining factor for such tools as Web and proxy servers, and efficient use of synchronization primitives is crucial for multithreaded applications.

Finding performance problems in OS-bound applications has always been a challenging task. A user-level profiler might locate a region of application's code where most of the system time is spent, but it might be unable to explain why this is happening or how to fix the problem. For example, if an application spent 90% of its time in the `open` system call, the profiler might be able to detect it and report to the programmer. However, `open` is a complex system call; it can create new files, truncate existing ones or even do network I/O in distributed file systems. Without knowing the exact cause of the problem the programmer may not know how to fix it.

If there were no opaque boundary between the user and kernel spaces, we could have continued profiling further down into the kernel. By refining the `open` bottleneck, the profiler might find that the performance problem is in file truncation that happens as an option in `open`. With this information, the programmer may be able to modify the application to work around the bottleneck. For example, it may be possible to avoid truncating files on `open` where not necessary. Alternatively, the programmer could tune certain kernel parameters to make the problem go away. The tuning might be done through a standard operating system interface, obviating the need for kernel recompilation or even reboot.

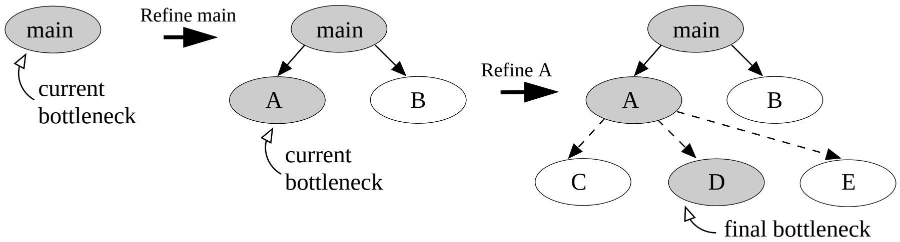


Figure 1. Walking the call graph

The goal of this work is to demonstrate that such cross-boundary profiling is feasible and indeed effective in finding performance problems. Our approach is based on the idea of using dynamic instrumentation to traverse the call graph in the search for bottlenecks. This technique has already been studied for call graphs of user applications [3]. We generalize it to seamlessly walk down the joined call graph of the application and the kernel.

The key technologies here are dynamic application instrumentation and dynamic kernel instrumentation. For the former, we use an existing library called Dyninst API. For the latter, we have designed a new framework called Kerninst API with an interface modeled after Dyninst. The synergy between the two libraries provides a unified and powerful interface for developing cross-boundary tools.

By bringing all the pieces together, we have designed a tool called CrossWalk, which can locate bottlenecks in an application, then seamlessly cross the kernel boundary and find the corresponding bottlenecks in the kernel code. With CrossWalk, a performance bottleneck can be refined from the `main` application entry point through the application's code and then all the way down into the kernel space to a kernel function that is the ultimate problem. While CrossWalk is still a simple prototype, it has already proved valuable in finding performance problems in our study of the Squid proxy server. Using CrossWalk's results, we were able to boost Squid's performance on a standard workload by more than a factor of 2.5.

2. Related Work

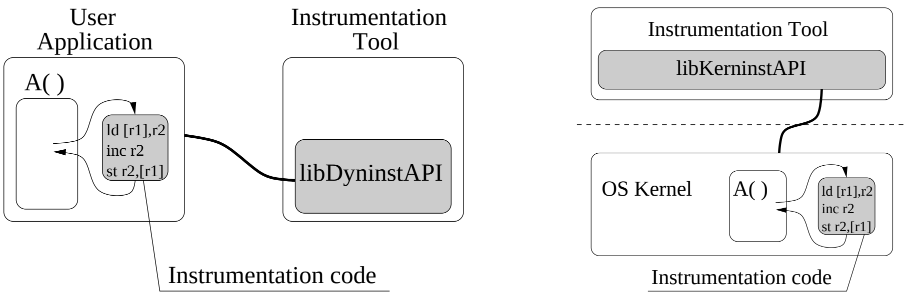
There exist several tools that already accomplish certain parts of our goal and can be combined together. For example, one could profile an application with Gprof [5] and then profile the kernel with Kgmon [7] or Kernprof [6]. Such analysis might find a set of bottlenecks in the user space and another set of kernel-space bottlenecks. However, understanding how the two sets are related to each other is not an easy task. In fact, it is even possible that all the found kernel bottlenecks are induced by other processes and have no connection to the process under study. Therefore, using two disjoint tools is often insufficient for understanding the cause of the problem.

While there are no complete tools for cross-boundary analysis, there exist several different technologies for implementing such tools. One can use sampling [5], tracing and profiling through source-code modification [8,9,12], or dynamic instrumentation [3,14]. In CrossWalk, we use dynamic instrumentation since it works on production systems and provides cycle-accurate timing results with little overhead.

After narrowing the choice down to dynamic instrumentation, again, one can choose from several application instrumentation frameworks [2,4,13,10]. However, the number of options for kernel instrumentation is smaller [10,11]. Of the two, only DProbes [10] appears to be currently maintained. While DProbes has sufficient functionality for our purposes, its instrumentation is trap-based. From our experience, the overhead of traps is prohibitively high to be used for performance monitoring.

3. Basic Technology

An effective way to search for performance bottlenecks is to walk down the application call graph using dynamic instrumentation. Paradyn uses this technique to look for bottlenecks in user code [3]. The following two sections describe call-graph walking and dynamic instrumentation in greater detail.



(a) A Dyninst-based system

(b) A Kerninst-based system

Figure 2. Frameworks for dynamic application and kernel instrumentation

3.1. Call-Graph Walking

Assume that we are searching for a performance bottleneck, that is, a function which takes up more than a certain amount of wall-clock time (e.g., 25% of total running time of the application). The call-graph approach mimicks what an experienced performance analyst would do when searching for a bottleneck. The corresponding step-by-step diagram is shown in Figure 1.

First, the method examines all functions called directly from the `main` application entry point. If the inclusive time spent in such a function is above the threshold, the search marks it as a bottleneck. Next, it tries to refine this bottleneck by examining immediate callees of that function. If one of them is a bottleneck, the search will descend in that function and continue doing so until it reaches a function that has no callees above the threshold (or no callees at all). That function is the final user-space bottleneck for which we were searching.

If the identified function is a system call, CrossWalk's analysis does not stop there. The tool seamlessly walks into the kernel, starts at the system call entry point, descends into its callees if necessary and keeps doing so until the final in-kernel bottleneck is identified. The result of the analysis is a chain of functions connecting `main` and the located bottleneck.

We collect performance data through dynamic instrumentation. While the application is running, the tool injects code at a function's entry point to start a timer and code at the exit points to stop the timer. The code is removed when it is no longer needed. This approach does not require source code modification or even recompilation. We discuss it in more detail below.

3.2. Application Instrumentation: Dyninst API

CrossWalk uses Dyninst API for application instrumentation. Dyninst is a C++ library that allows you to write your own instrumentation tools. The typical structure of such a tool is shown in Figure 2(a). The library lets you attach to a running application and perform the following operations:

Instrument: the tool can generate instrumentation code and ask the library to inject it at a specified point in an application. With this facitily, CrossWalk puts its timing code into functions.

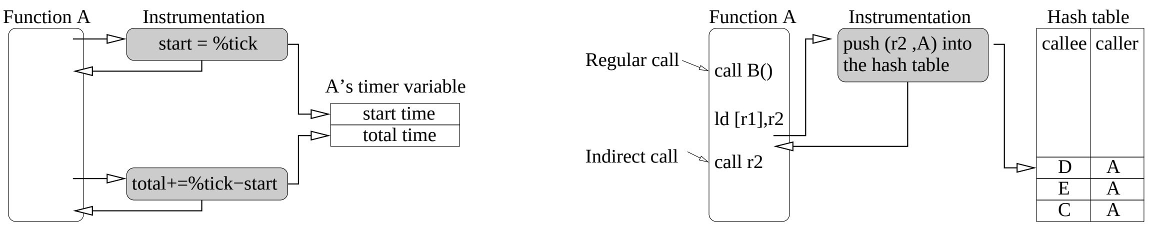
Browse code resources: the tool can locate application's modules, functions, basic blocks, and retrieve control flow information. For example, CrossWalk uses this functionality to locate `main` and identify all functions called from a given function to walk down the call graph.

Inspect/modify application's data: the tool can read/write an application's variables while the application is running. With this facility, CrossWalk peeks at values of the timers that its instrumentation code created.

3.3. Kernel Instrumentation: Kerninst API

Dyninst API allows us to instrument user applications, but to drill into the kernel one needs to use another mechanism. We already had some experience with kernel instrumentation after designing a kernel performance monitor called Kperfmon [14]. With a reasonable amount of effort, we factored out all instrumentation-related functionality into a library, which we called Kerninst API.

Kerninst API is closely modeled after Dyninst API. It provides the same functionality, has the same interface and the same syntax where possible to let the programmer use a single and consistent model



(a) Timing a function

(b) Identifying callees of a given function

Figure 3. Major building blocks of CrossWalk

for both application and kernel instrumentation. Finally, to enable tools like CrossWalk, we designed Kerninst API to coexist with Dyninst in the same application if necessary.

A typical Kerninst-based system is shown in Figure 2(b). The system has a few important properties. First, even though the instrumentation tool now injects code into the kernel, the tool itself is a user-space application. Second, Kerninst works on a standard production operating system. There is no need to recompile the kernel or even reboot the machine. The tool simply attaches to the kernel and starts instrumenting it. Third, Kerninst requires the user to have root privileges in the system. Otherwise, allowing a user to insert and execute code in the kernel would be a security hole.

4. Implementation

Two major issues in implementing CrossWalk is how to time a function, and how to find a function's callees to walk down the call graph. The similarity of Dyninst and Kerninst APIs let us implement these features in a unified way for both the kernel and user spaces.

4.1. Timing in CrossWalk

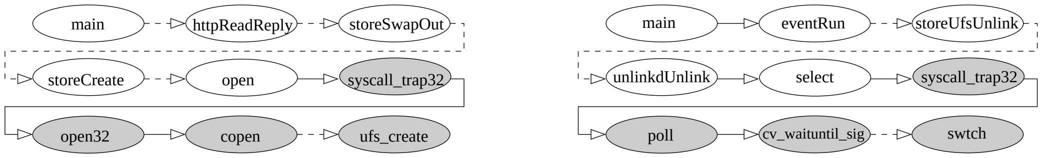
Figure 3(a) shows the sketch of instrumentation that we insert into a function of interest. To provide cycle-accurate results with low overhead, the code uses processor hardware counters for timer readings. It samples the cycle counter at the entry point of the function, samples it again at the exit point and computes the delta. As the code executes, the per-function timer variable accumulates the total number of wall-clock cycles spent in the function. Note that this approach can be easily extended to handle other types of performance metrics, such as CPU time or number of cache misses.

CrossWalk uses similar primitives for timing application and kernel code. The only difference is that in the kernel it has to constrain timing to the process under study. If multiple threads execute within a kernel function, the timer should only reflect the time spent there by our process's thread. CrossWalk achieves this goal by wrapping the kernel start/stop primitives with a check for the proper process identifier of the currently running thread. The check is fast, taking only five instructions.

4.2. Identifying a function's callees

When a function proves to be the bottleneck, CrossWalk will try to refine it by looking at the function's callees. Both Dyninst and Kerninst API provide a method for identifying callees through static analysis. The method locates all call instructions in the function and extracts the destination addresses from them. While static analysis works in most cases, it fails to analyze indirect calls whose destinations are computed at run time. Such constructs are common in practice: calls through function pointers, callbacks, and C++ virtual functions all get translated to the indirect call instruction.

CrossWalk solves this problem in the same way as Paradyn [3]; it instruments the indirect call sites to identify callees at run time. As Figure 3(b) shows, when the call-site instrumentation is invoked, a callee address has already been computed, so the instrumentation simply adds it to a hash table. As a result, the hash table will soon contain all callees invoked from the watched sites. This approach has worked well. Without support for following indirect calls, CrossWalk would not be able to advance past the system call entry point, where system call handlers are dispatched through an indirect call.



(a) Bottlenecks in the original version

(b) Bottlenecks in the tuned version

Figure 4. Squid's bottlenecks. Links are dashed where functions were omitted for brevity.

5. Experimental Results: Squid

A popular technique for optimizing Web transfers is proxy caching. It places an intermediary, a proxy server, between Web servers and their clients and caches server responses to the clients. While the technique takes extra load off Web servers and the network infrastructure, its effectiveness heavily depends on performance of the proxy server. A study from four years ago [14] found and fixed some performance problems in a popular open-source proxy called Squid [15]. Here, we use CrossWalk to see if the current version of the application can be enhanced further.

We ran Squid-2.5.STABLE3 on a Sun Ultra-10 workstation with UltraSparc-IIi 440MHz processor, 256 MB of RAM, 9GB IDE hard drive, 100Mbit network card, and the 64-bit version of Solaris 8. To model typical Web activities, we subjected it to a standard workload known as the Wisconsin Proxy Benchmark [1]. The workload issues concurrent requests from 60 clients to 10 different Web servers through Squid. The benchmark's measure of performance is how much wall-clock time it takes to serve 12000 requests (200 requests per client).

CrossWalk attached to Squid and found the chain of functions shown in Figure 4(a) to be above the 25% wall-clock time threshold (white nodes belong to the application and gray to the kernel). Functions `httpReadReply` and `storeSwapOut` (receive an object from a Web server and store it in the on-disk cache) appear as bottlenecks. However, it was not the network `read`, nor the file `write` that was the problem; surprisingly, the bottleneck was in the `open` system call. By skimming through the source code for Squid's `storeCreate` we realized that these calls to `open` correspond to opening individual cache files for creating new objects in them. At that point, a user-level profiler would have to stop, but CrossWalk was able to drill further into the kernel and traced the `open` bottleneck to the `ufs_create` routine. Apparently, Squid spends most time creating new files in the on-disk cache.

A possible way to remove the bottleneck that CrossWalk found is to pre-create all files in the Squid's cache. Now, when a new object comes from a Web server, Squid can open an existing zero sized file and save the object in it, avoiding the need to create a new file. An obstacle to this approach is that Squid removes old files from the cache periodically to keep the cache size under control. As a result, it will eventually fall back to the original mode of creating files. This obstacle is minor however, since one could modify Squid's source to never remove old files, but simply truncate them to zero size.

As we discovered, Squid already provides support for this truncation strategy, though it is not enabled by default. When we turned it on by recompiling Squid with appropriate parameters, the running time of the Wisconsin Proxy Benchmark dropped by more than a factor of 2: from 625 seconds for the original scheme (removing files) to 275 seconds for the modified one (truncating files).

To optimize Squid further, we ran it under CrossWalk again and discovered problems shown in Figure 4(b). The new user-space bottleneck is the `unlinkdUnlink` function calling `select`. Refinement into the kernel suggests that Squid simply blocks on a condition variable (`cv_waituntil_sig`) and gets switched-out while doing so (`swtch`). As this chain indicates little kernel activity, the problem must be in the application. We examined the source for `unlinkdUnlink` and found that it is waiting until Squid's helper daemon, `unlinkd`, consumes some file-truncate requests from a filled-up queue.

If the requests come in bursts, one might give `unlinkd` a chance to catch up simply by making the queue longer. When we extended it from 20 to 200 entries and recompiled Squid, the running time of the benchmark dropped by additional 11% from 275 to 245 seconds. Again, CrossWalk correctly identified the cause of this performance problem and provided enough information for us to fix it.

To estimate the overhead of profiling, we ran the benchmark multiple times with and without Cross-

Walk and determined that CrossWalk increased the total running time by approximately 15%. To isolate individual contributions of application and kernel instrumentation, we then modified CrossWalk to perform application-only or kernel-only call-graph walks. We found that the application instrumentation accounted for approximately 60% of the total slowdown, the kernel one for 40%. Overall, the overhead was reasonably low and did not prevent CrossWalk from finding correct bottlenecks.

6. Future work

We have identified two directions for further improvement of CrossWalk. The first is to support analysis of multithreaded applications, which are becoming increasingly common. Paradyn can already search for user-space bottlenecks in such applications [16]. Providing matching support for the kernel-space analysis should be relatively straightforward.

The second direction is to account for asynchronous kernel activities. Often, applications spend a lot of time in the kernel, but perform no system calls. A typical example is an out-of-core application, which does not fit into the main memory and thus suffers from excessive paging activities. In such application, CrossWalk would narrow the problem down to a user-level function that takes up most time, but it would not be able to refine it further down into the kernel, as the system call entry point will not be the bottleneck. One possible way to account for such activities is to walk the kernel call graph from all entry points simultaneously, but the feasibility of this approach is yet to be studied.

REFERENCES

[1] J. Almeida and P. Cao, "Wisconsin Proxy Benchmark", Technical Report 1373, Computer Sciences Department, University of Wisconsin-Madison, April 1998.

[2] B. Buck and J.K. Hollingsworth, "An API for runtime code patching", *Journal of High Performance Computing Applications* **14**, 4, pp. 317–329, Winter 2000.

[3] H.W. Cain, B.P. Miller, and B.J.N. Wylie, "A Callgraph-Based Search Strategy for Automated Performance Diagnosis", *Euro-Par 2000*, Munich, Germany, August 2000.

[4] L. DeRose, J.K. Hollingsworth, and T. Hoover, "The dynamic probe class library—an infrastructure for developing instrumentation for performance tools", *International Parallel and Distributed Processing Symposium*, April 2001.

[5] S. Graham, P. Kessler, and M. McKusick, "gprof: A Call Graph Execution Profiler", *SIGPLAN '82 Symposium on Compiler Construction*, Boston, June 1982, pp. 120–126.

[6] Kernprof, http://oss.sgi.com/projects/kernprof/

[7] M. McKusick, "Using gprof to Tune the 4.2BSD Kernel", *European UNIX Users Group Meeting*, April 1984.

[8] B.P. Miller, M. Clark, J.K. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Parallel and Distributed Systems* **1**, 2, pp. 206–217, April 1990.

[9] B. Mohr, D. Brown, and A. Malony, "TAU: A portable parallel program analysis environment for pC++", *CONPAR 94 - VAPP VI*, Linz, Austria, September 1994.

[10] R.J. Moore, "Dynamic Probes and Generalised Kernel Hooks Interface for Linux", *4th Annual Linux Showcase & Conference*, Atlanta, October 2000.

[11] D.J. Pearce, P.H.J. Kelly, T. Field, U. Harder,"GILK: A Dynamic Instrumentation Tool for the Linux Kernel", *12th International Conference on Computer Performance Evaluation*, April 2002.

[12] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment", *Scalable Parallel Libraries Conference*, Los Alamitos, CA, October 1993, pp. 104–113.

[13] M. Ronsse and K. De Bosschere, "JiTI: A Robust Just in Time Instrumentation Technique", *Workshop on Binary Translation*, Philadelphia, October 2000.

[14] A. Tamches and B.P. Miller, "Using Dynamic Kernel Instrumentation for Kernel and Application Tuning", *International Journal of High Performance Computing Applications* **13**, 3, Fall 1999.

[15] D. Wessels, "Squid Internet Object Cache", http://squid.nlanr.net/Squid/, August 1998.

[16] Z. Xu, B.P. Miller, and O. Naim, "Dynamic Instrumentation of Threaded Applications", *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, May 1999.