Using Dynamic Kernel Instrumentation for Kernel and Application Tuning[1]

Ariel Tamches and Barton P. Miller
Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
{tamches,bart}@cs.wisc.edu

Abstract

We have designed a new technology, fine-grained dynamic instrumentation of commodity operating system kernels, which can insert runtime-generated code at almost any machine code instruction of an unmodified operating system kernel. This technology is ideally suited for kernel performance profiling, debugging, code coverage, runtime optimization, and extensibility. We have written a tool called KernInst that implements dynamic instrumentation on a stock production Solaris 2.5.1 kernel running on an UltraSparc CPU. We have written a kernel performance profiler on top of KernInst. Measuring kernel performance has a two-way benefit; it can suggest optimizations to both the kernel and to applications that spend much of their time in kernel code. In this paper, we present our experiences using KernInst to identify kernel bottlenecks when running a web proxy server. By profiling kernel routines, we were able to understand performance bottlenecks inherent in the proxy's disk cache organization. We used this understanding to make two changes—one to the kernel and one to the application—that cumulatively reduce the percentage of elapsed time that the proxy spends opening disk cache files for writing from 40% to 7%.

1 Introduction

Operating system kernels are complex entities whose internals are often difficult to understand, much less measure and optimize. We have designed *fine-grained dynamic kernel instrumentation*, a low-level technology that allows arbitrary code to be *spliced* (inserted) at almost any kernel machine code location during runtime. Dynamic kernel instrumentation allows runtime measurements, optimizations, and extensibility to be performed on unmodified commodity kernels.

Dynamic kernel instrumentation provides, in a single infrastructure, the means for monitoring functionality (such as debugging and profiling) alongside mechanisms for extensibility and adaptability. When used in this manner, a kernel becomes an evolving entity, able to measure and adapt itself to accommodate real-world runtime usage patterns. Evolving operating systems can form the foundation of such dynamic environments, since their code can adapt to demands. Performance profiling through dynamic kernel instrumentation is a natural fit for meta-computing (Grid) environments, which place dynamic demands on an operating system, making runtime performance gathering essential [7].

A previous paper describes the low-level technology of dynamic kernel instrumentation and its implementation on unmodified Solaris 2.5.1 kernel [18]. This paper presents a case study using KernInst to optimize a web proxy server, by making changes to both the kernel and the proxy. We have found that understanding kernel performance has a two-way benefit; it provides information useful for tuning *both* the kernel and user processes. Thus, kernel profiling is useful to both kernel and application developers.

The remainder of this paper is organized as follows. Section 2 summarizes KernInst; Section 3 gives an overview of the benchmark we used to drive the kernel; Section 4 is our study of kernel performance bottlenecks when running the benchmark, our solutions, and ideas for future optimization; Section 5 discusses related work; and Section 6 concludes.

2 KernInst

We have designed and implemented a tool called KernInst that performs fine-grained dynamic instrumentation of a stock, unmodified Solaris 2.5.1 kernel running on an UltraSparc CPU. KernInst allows desired runtime-generated code to be inserted, removed, or changed almost anywhere in the kernel's code space

1. This work is supported in part by Department of Energy grant DE-FG02-93ER25176, NSF grant CDA-9623632, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

(at machine code instruction granularity), entirely at runtime. KernInst instruments dynamically; unlike a kernel binary rewriter, no reboot is required for instrumentation to take effect.

We have built a kernel performance profiling tool using KernInst that allows a user to insert performance-gathering primitives into in the kernel at runtime. Performance instrumentation is determined by choosing a *metric* and a *resource*. A metric determines the type of measurement; a resource determines what function is measured. In this paper, we use two metrics: the first measures the number of procedure calls made to a specified function ("calls made to"), and the second measures the number of kernel threads executing within a specified function ("concurrency"). Measuring a function's concurrency is similar to measuring its latency, and has more meaning in a multi-threaded environment. Total latency (with units of seconds) only measures the total time spent in a function; if *k* threads simultaneously execute within a function, the overlapping time is only counted once. Total concurrency (with units of thread-seconds) will include the overlapping time interval *k* times. In a plot of concurrency over time (such as the bottom curve in Figure 1), the *x*-axis denotes time and the *y*-axis denotes the number of threads executing within the specified function at that moment (hence the metric name "concurrency").

Because instrumentation is deferred until the user requests it, kernel performance profiling using KernInst is a completely interactive and dynamic activity. Metric/resource selections can be made, appropriate instrumentation inserted, and measurements gathered, at any time. Because finding bottlenecks is an interactive activity involving successive refinement of the set of routines to be measured, it is essential for a tool to allow the user to decide what measurements should be made at runtime. KernInst is ideally suited to this paradigm.

3 Performance Study Parameters

As a case study in kernel performance profiling, we used KernInst to measure the performance of the Solaris 2.5.1 kernel running the Squid web proxy server. In this section, we outline two major components of the benchmark: Squid and the Wisconsin Proxy Benchmark.

3.1 Squid Web Proxy Server

Web proxy servers are an effective means for reducing the load on web servers. Web clients attach to a local proxy (instead of the actual web server), which caches some of the web server's contents. The proxy retrieves files as needed from the web server when it cannot satisfy a client's HTTP request from its cache.

We studied the performance of the Solaris 2.5.1 kernel while running version 1.1.22 of Squid, a freely available web proxy server [12]. Squid provides two levels of cache: in-memory and on-disk. Incoming requests are first searched in Squid's memory cache. If it misses there, Squid tries its disk cache. If Squid misses there, it fetches the file from the server. Squid's disk cache was installed to a local disk running the default Unix file system (UFS).

A heavily loaded proxy server can expect hundreds (or thousands) of simultaneous TCP connections, which it must multiplex alongside any local disk activity of its own. Squid does not create a thread or process to handle each request. Instead, a single thread of control multiplexes among all active TCP connections and pending file operations using non-blocking I/O operations.

3.2 Wisconsin Proxy Benchmark

We used version 1.0 of the Wisconsin Proxy Benchmark [1] to drive Squid. Thirty synthetic client processes connect to a Squid proxy. Squid in turn connects to a synthetic server process. Three machines (client, Squid, server) are used.

Each client process connects to Squid and makes HTTP GET requests with no thinking time in between. Requests are sent in two stages. In the first stage, 100 requests are made for files. The same file is never requested twice, so there is no locality. The purpose of this stage is to populate the cache and to stress its cache replacement algorithm. In the second stage, the client sends 100 requests, but this time with temporal locality pattern designed to lead to a proxy hit ratio of 50%.

The server process listens on a particular port number for HTTP requests. When one arrives, it parses the URL to determine the appropriate file. If it has not yet been (synthetically) created, the server creates a file. The file's size will be uniformly distributed from 3K to 40K 99% of the time; the other 1% of the time, a 1 MB file size is used.

4 Performance Study

We used KernInst to measure the performance of the Solaris 2.5.1 kernel running the above benchmark. The search for an application (Squid) bottleneck led into the kernel, requiring a kernel profiling tool. This section discusses two bottlenecks that we found, optimizations (one to the kernel and one to Squid) that address them, and ideas for further optimization.

We found that kernel performance profiling has a two-way benefit. First, kernel performance profiling allows us to understand kernel bottlenecks, leading to kernel optimizations. Second, because proxy servers spend much time in the kernel performing I/O, it is necessary to measure in-kernel performance to understand its bottlenecks and optimize the application.

4.1 The First-Order Bottleneck: File Opens

Because the Wisconsin Proxy Benchmark has a working set size larger than Squid's in-memory cache, we hypothesized that Squid might be disk bound[2]. We first examined the Squid code that was causing the disk thrashing. We hypothesized the bottleneck could be disk reads (files that missed in Squid's in-memory cache but hit when reading from disk) or disk writes (files that missed in Squid's on-disk cache also, requiring the file be brought in from the true server). We ran the Quantify profiling tool [14] on Squid to determine the source of this first-order bottleneck, and found it to be neither.

Squid's bottleneck is in the routine storeSwapOutStart(), which is called to demote a file from Squid's in-memory cache to its on-disk cache. Interestingly, the bottleneck occurred not in writing the disk cache file, but in the call to open()! Squid was spending 76% of its non-idle (i.e., excluding time spent waiting in a select() statement) run time simply *opening* on-disk cache files for writing. To explain this result, we investigated Squid's cache organization.

Squid maintains one file per HTTP object being cached on disk. Thus, Squid's on-disk cache is not a single (huge) fixed-size file, but a collection of all files being cached, with differing file sizes. This simplifies Squid's code, but as we will see, is the cause of severe bottlenecks.

Squid organizes its files into a hierarchy, to keep the number of files in any one directory manageable. Squid uses a three level directory structure, fanned out based on fixed program constants. A hash function maps a file table entry number to a full path name. If the file was used for a since-ejected cached object, the old file will still be present. Squid truncates an existing file to zero size by passing the O_TRUNC flag to open. By reusing files, Squid avoids the need to delete files; this avoids expensive meta-data I/O required when deleting a UNIX file (updating the parent directory file and freeing the file's inode and disk blocks) [4]. However, since Squid is spending so much time in open, this strategy was clearly ineffective.

Any bottleneck in the open system call is serious in a program that multiplexes between many file descriptors, because there is no interface for a non-blocking open system call. This contrasts with read and write, which can return EWOULDBLOCK if they would block the process, allowing work to proceed on other, ready file descriptors. Without non-blocking opens, the entire Squid process (which can include dozens of open file descriptors) is blocked whenever open on any file blocks.

Further performance study required examining *why* open was slow. This in turn requires understanding the operations that take place in an open system call. User-level performance profilers see systems calls as a black box, and can offer little guidance in understanding kernel performance. So we used KernInst to continue the performance study where user-level tools leave off: the user-kernel boundary.

4.2 Understanding Kernel Performance of open()

Given the dynamic nature of KernInst, we can *interactively* find kernel bottlenecks in much the same way that bottlenecks are found in user programs. We start with a function that is performing slowly, and measure the latency of that routine and its callees. If measurement determines that a callee is a bottleneck, then the process is repeated for the callee. We used KernInst to calculate the number of cycles spent in a routine by instrumenting its entry and exit points with code that starts and stops a cycle counter, respectively. Since KernInst allows instrumentation of an unmodified, running, commodity kernel, we were able to perform this iterative refinement on our stock Solaris 2.5.1 UltraSparc workstations while Squid was running.

2. An interesting symptom that tends to support this claim was the constant noise of disk seeks coming from the hard drive of the computer running Squid.

We first measured the kernel routine copen(), which implements both file creation and file opening[3]; the results are shown in Figure 1. Although copen is only called 20-25 times per second, 40% of Squid's

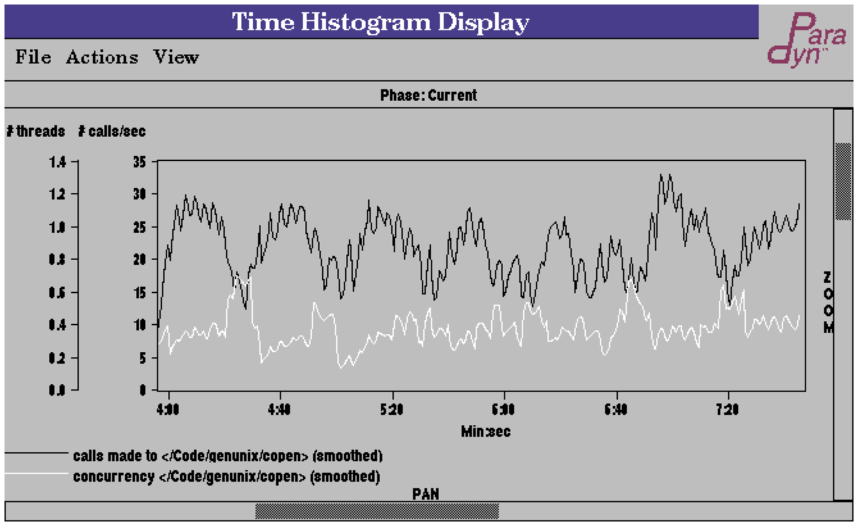


Figure 1: copen
Although called only 20-25 times/sec, copen() is a clear bottleneck. On average, 0.4 kernel threads are executing in this routine at any given time; this translates to 40% of Squid's elapsed time, since it is a single-threaded program.

elapsed time is spent here[4]. We next examined where copen was spending its time. copen calls falloc(), to allocate an available entry in the process's file descriptor table, and then vn_open(), to perform the open. Times for these routines are shown in Figure 2. We were surprised to find that falloc was taking negligible

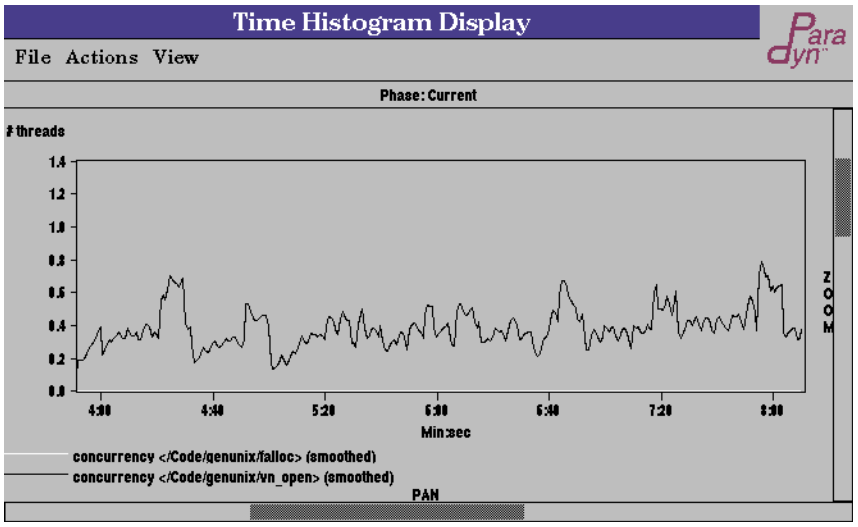


Figure 2: copen()'s major callees: falloc and vn_open
Negligible time is spent in falloc

time, because file table allocation has been reported to be a bottleneck under heavy disk load [5].

Since most of copen's time was spent in vn_open, we examined it next. It is this routine where file creation and opening diverge, calling vn_create if the O_CREAT flag was passed to open. vn_open is called about

3. open() with the O_CREAT flag essentially performs a creat().
4. The percentage of elapsed time differs from what Quantify reports for open() (76%) because Quantify's percentage is of *non-idle* time (elapsed wall time *excluding* time spent waiting idle in a select() call).

20-25 times per second; of these, about 8 go to vn_create when creating an on-disk cache file. The remaining calls to vn_open are non-creating, indicating hits in the on-disk cache. However, an examination of the latencies in Figure 3 show that the time spent in vn_open is almost entirely consumed by the time spent in vn_create. Thus, the 8 file creations per second account for Squid's file opening bottleneck.

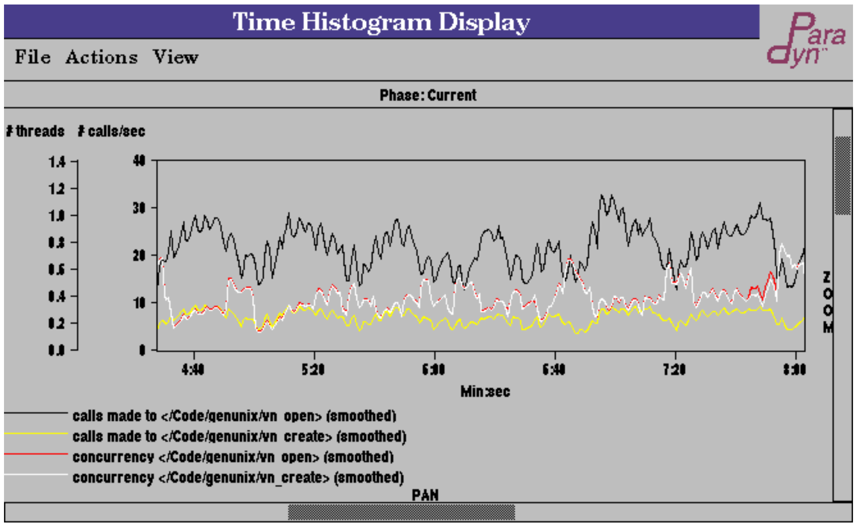


Figure 3: vn_open spends most of its time in vn_create
The concurrency curves for vn_open and vn_create overlap (averaging about 0.4 threads in them at any given time)

Since file creation is a bottleneck, we next examined the vn_create routine. vn_create calls lookuppn() (better known as namei) to translate the file path name to a vnode. It then passes the vnode to the file system-specific creation routine; since we ran Squid on the local disk's default Unix File System (UFS), vn_create invokes ufs_create(). The results are shown in Figure 4, and reveal that file creation has two distinct bottlenecks: path name lookup (lookuppn) and UFS file creation (ufs_create). We will return to the bottleneck in

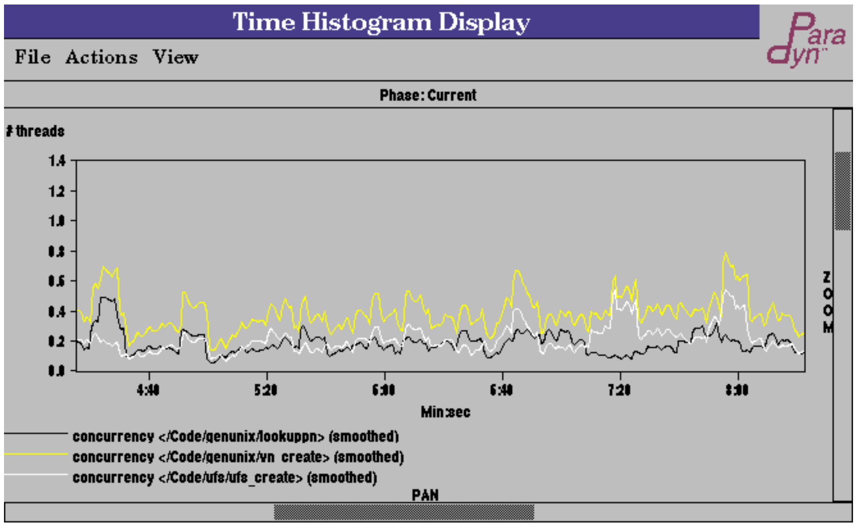


Figure 4: vn_create and its two main callees: lookuppn() and ufs_create()
Both lookuppn and ufs_create are (distinct) bottlenecks (each about 20% of elapsed time)

lookuppn later; for now we turn our attention to ufs_create.

We were surprised to find that ufs_create was a bottleneck, because file creation on UFS performs only local meta-data operations. As shown in Figure 4, Squid is spending about 20% of its time there. We traced the time spent in ufs_create to ufs_itrunc, which is invoked by ufs_create when the O_TRUNC flag is passed to the open system call. The two routines have nearly identical performance numbers, as shown in Figure 5.

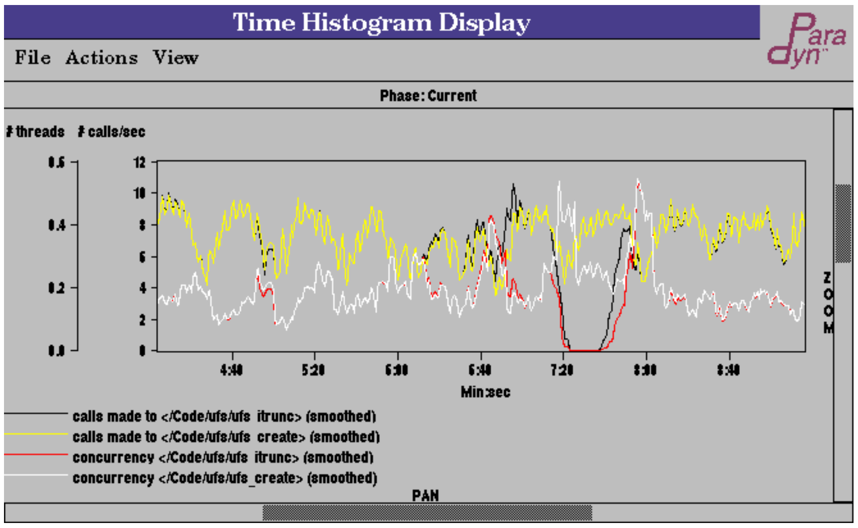


Figure 5: ufs_create time is mostly spent in ufs_itrunc
UFS file creation time is dominated by inode truncation; the latency curves for ufs_create *and* ufs_itrunc *almost completely overlap*

Thus, about 20% of Squid’s elapsed time is spent truncating existing files to zero size when opening them. To determine why ufs_itrunc was so slow, we next looked at its callees; the results are shown in Figure 6.

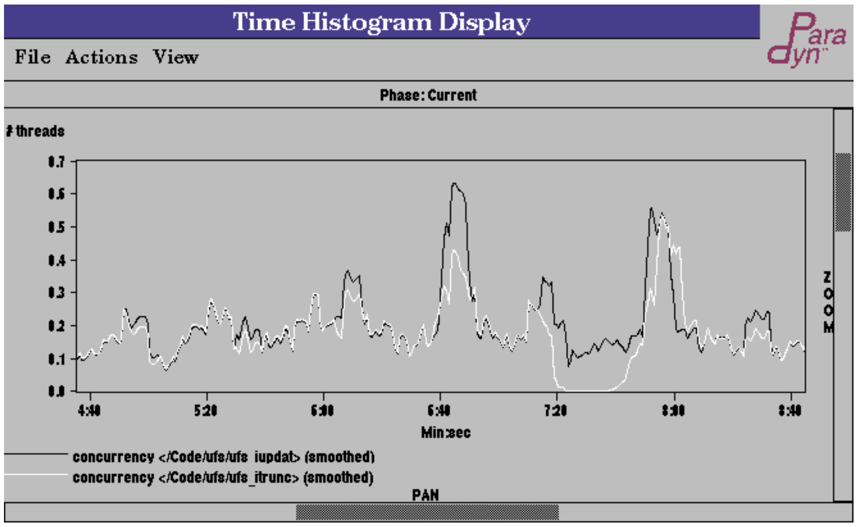
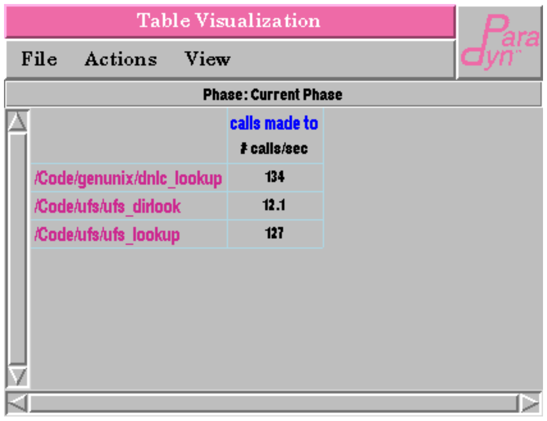


Figure 6: Most of ufs_itrunc’s time is spent in ufs_iupdat
File truncation is slow because UFS meta-data updates are made synchronous by ufs_itrunc

Most of ufs_itrunc’s time is spent in ufs_iupdat, which synchronously writes to disk any pending updates to the file’s inode. That is, truncation is slow because inode changes are made synchronously. This is in keeping with Unix file semantics that dictate synchronous updates to meta-data, to ensure file system integrity in case of a crash. *Squid’s strategy of overwriting existing cache files to avoid the expensive meta-data updates required in file deletion is backfiring.* We present an optimization that addresses this bottleneck in Section 4.4.

Recall from Figure 4 that lookuppn is a bottleneck. To some extent, this is not surprising, since obtaining a vnode from a path name can require, for each path name component, reading a directory file to obtain an inode disk location, and reading the inode. Solaris tries to optimize path name lookup through the directory name lookup cache, or DNLC [5]. The DNLC is a hash table, indexed by path name component, containing a pointer to an entry in another cache, the inode cache. A hit in the DNLC allows the operating system to bypass both reading the directory file (ufs_dirlook()) and reading the inode (ufs_iget()). As shown in Figure 7, the DNLC hit ratio is about 90%. Unfortunately, the miss penalty (execution of ufs_dirlook) is sufficiently



	calls made to
	# calls/sec
/Code/genunix/dnlc_lookup	134
/Code/ufs/ufs_dirlook	12.1
/Code/ufs/ufs_lookup	127

Figure 7: *The DNLC hit ratio is about 90%*
The miss routine, ufs_dirlook, is only invoked once per 10 calls to ufs_lookup

high to account for the ufs_lookup bottleneck, as shown in Figure 8.

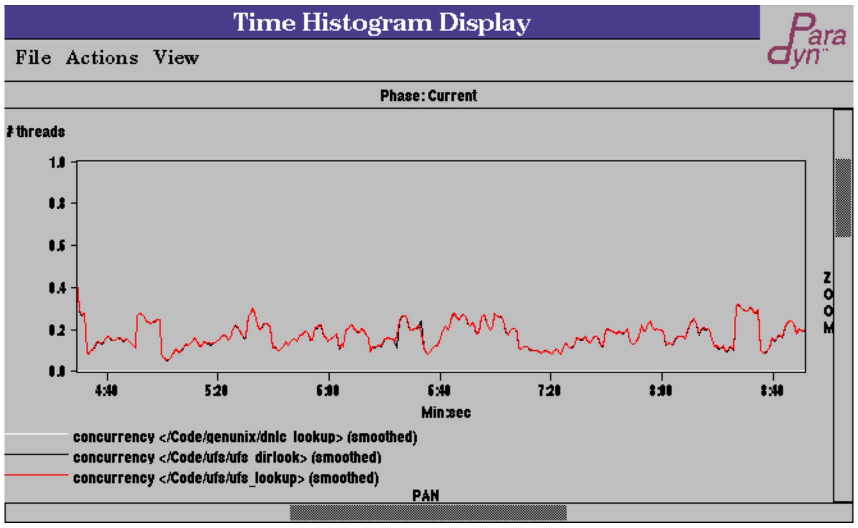


Figure 8: ufs_lookup spends most of its time in ufs_dirlook
Despite a low miss rate, the DNLC miss penalty (ufs_dirlook) is high enough to account for the entire ufs_lookup bottleneck (the ufs_dirlook and ufs_lookup curves almost completely overlap). The dnlc_lookup curve is essentially zero because checking for a DNLC hit or miss is always quick.

4.3 Addressing the Pathname Lookup Bottleneck

The DNLC, by default, contains 2181 entries on our machine. Since there are over 6,000 Squid cache files in our benchmark plus hundreds of subdirectories in the three-level cache hierarchy, the DNLC was ineffective because Squid's preponderance of small files overwhelmed it. Since Solaris 2.5.1 sets the DNLC size based on the kernel variable maxusers, we were able to increase the DNLC size to 17,498 by increasing maxusers to 2048 in /etc/system (the maximum allowed [5]) and rebooting.

The effects of increasing the DNLC size can be seen in Figure 9. Once the benchmark has run long enough to warm up Squid's disk cache (about one minute), the DNLC miss rate, once 10%, drops to 1%. Correspondingly, the total time spent in the miss routine (ufs_dirlook) drops to a negligible percentage of Squid's running time. Best of all, ufs_lookup (and by implication lookuppn) is no longer a bottleneck.

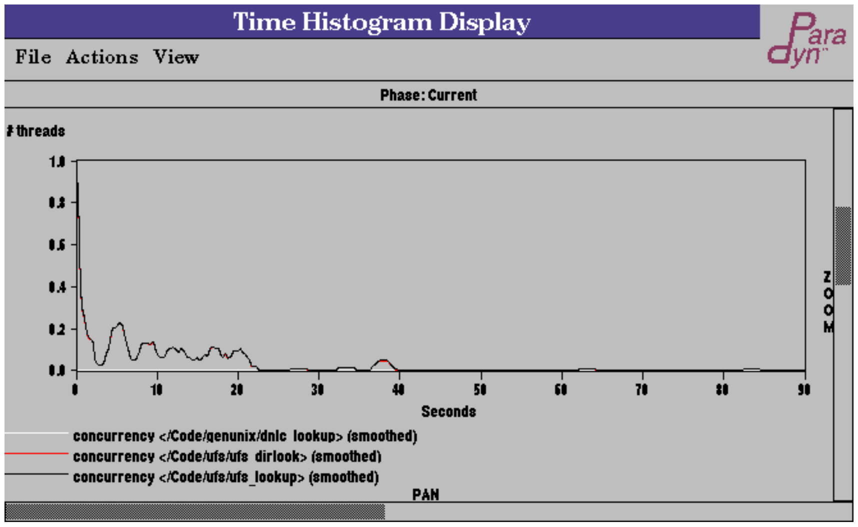


Figure 9: The effect of increasing DNLC size on ufs_lookup latency

For the first twenty seconds of the benchmark, there are enough DNLC misses to account for 10% of Squid's run time. As file names are reused more often, however, the DNLC hits become more frequent, and the ufs_dirlook bottleneck is gone (compare to Figure 8).

4.4 Addressing the File Truncation Bottleneck

We now address the ufs_create bottleneck: synchronous inode updates when truncating a file to zero size. We modified Squid to reduce this bottleneck.

Squid has a bottleneck in ufs_create because it truncates existing cache files to zero size when overwriting them. This involves updating the file's inode, which is done synchronously in keeping with UFS semantics. We note that truncation is redundant because data blocks will be added (and again, synchronously) as the new version of the file is written. If the file's new size is greater than or equal to the original size, then truncating the file is superfluous, because every data block that is deleted will be re-created. If the file's new size is less than the original size, a variant of the optimization presents itself: only truncate the extra blocks at the end of the file, representing the change in file size.

Only three minor changes to Squid, totalling 15 lines of source code, were needed to implement these changes. First, when opening a disk cache file, the O_TRUNC flag has been removed. The new contents of the file are then written (we removed Squid code that always seeked to the end-of-file before writing). When done writing, an fcntl() call with the parameter F_FREESP was used to truncate the file size to the present location of the file pointer. Thus, if the new file size is smaller than the previous file size, the inodes representing the now-unused end of the file are deleted. If the new file size is greater than or equal to the original file size, the fcntl will have no effect.

After optimization, we re-ran KernInst to examine file creation performance. As shown in Figure 10, performance has improved; less than 20% of Squid's time is spent creating cache files, compared with 40% earlier.

File truncation latency in the optimized version of Squid is shown in Figure 11. Calls to ufs_itrunc are no longer made by the open system call, because Squid no longer passes the O_TRUNC flag when opening cache files for writing. Instead, before the file is closed, Squid invokes an fcntl with the F_FREESP flag to truncate the portion of the file that is not needed. This results in kernel code ufs_freesp() being executed. With the smarter truncation policy, time that Squid spends updating meta-data has reduced from about 20% to about 6%.

4.5 Combined Effects of Both Optimizations

In Section 4.3, we saw that increasing the DNLC size reduced path name lookup time from about 20% of Squid's run time to about 1%; in Section 4.4, we saw that avoiding unnecessary file truncation in Squid reduced UFS file creation time from about 20% of Squid's run time to a negligible value. The combined effects of the two optimizations are shown in Figure 12. File creation, which once took about 40% of

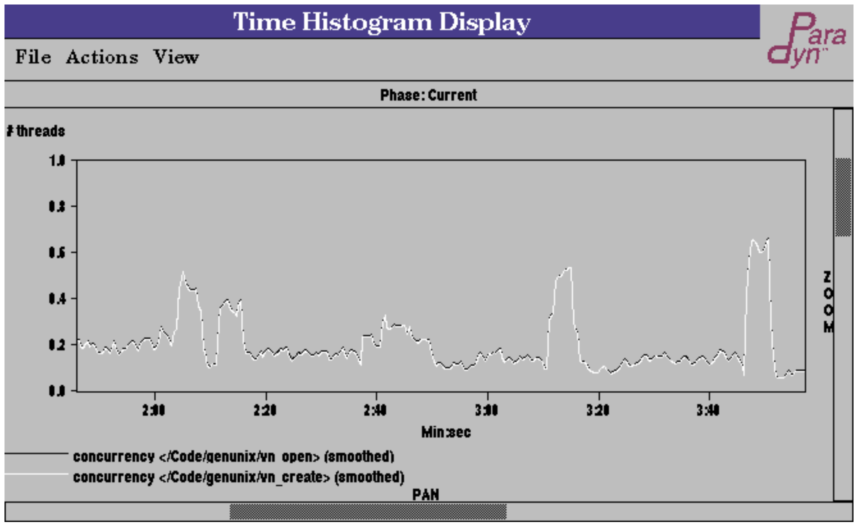


Figure 10: File creation performance when the truncation bottleneck is addressed
File creation once took 40% of Squid's run time; the inode truncation optimization reduces it to 20%.

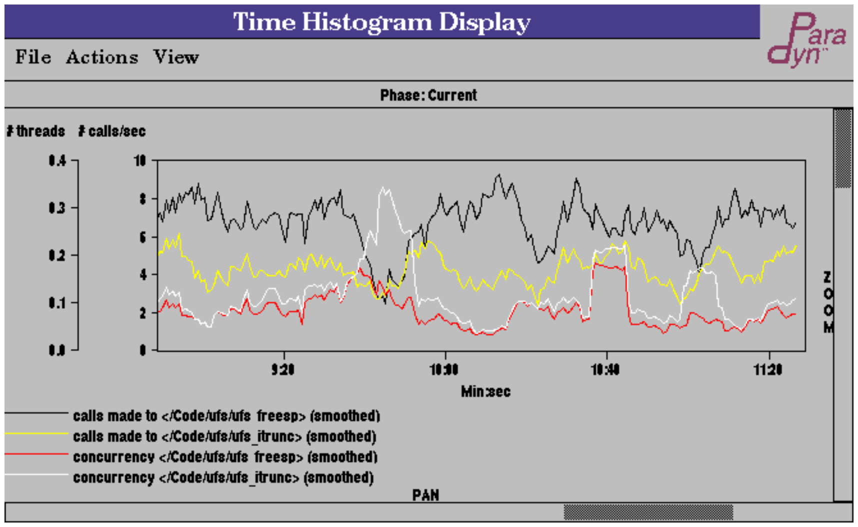


Figure 11: ufs_itrunc in optimized Squid
Because we no longer use the O_TRUNC flag in Squid when opening cache files, truncation is now mostly performed when explicitly requested via fcntl (ufs_freesp()).

Squid's run time, now takes less than 1%. To this must be added the time spent explicitly truncating inodes via fcntl (ufs_freesp), which (from Figure 11) is about 6%. Thus, what once took 40% of Squid's elapsed run time now takes about 7%.

We re-ran Quantify on Squid, and found that only 15% of Squid's elapsed time (excluding idle time in select) was now being spent in storeSwapOutStart, opening cache files for writing; this contrasts with 76% before optimization. The first-order bottleneck in Squid is now in write(), which takes 44% of Squid's non-idle elapsed time.

4.6 Further Ideas for Squid Optimization

Both of the bottlenecks identified by KernInst involve meta-data updates when opening a local disk file for writing. Although we have presented optimizations that significantly reduced these bottlenecks, 7% of Squid's elapsed time is still spent waiting for the open system call to complete. We believe that a fundamental redesign of Squid's disk cache would further improve performance. Instead of one on-disk cache

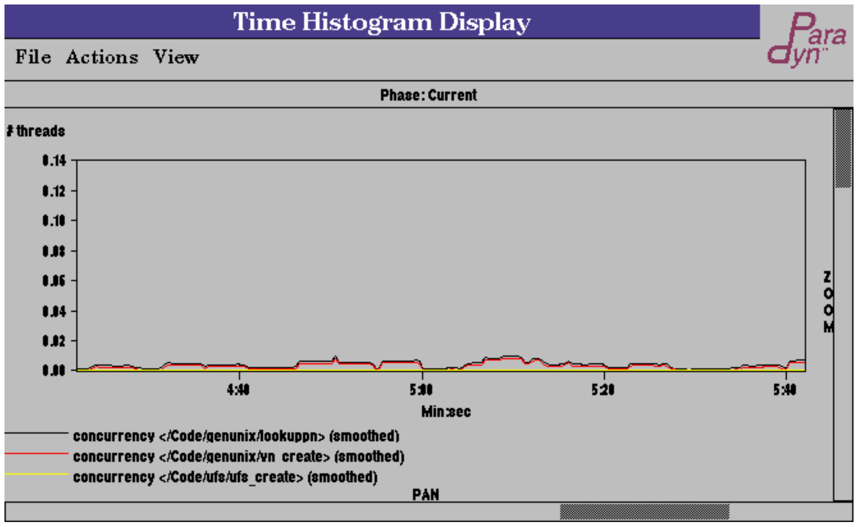

Time Histogram Display
Para dyn
File Actions View
Phase: Current
# threads
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
4:40
5:00
5:20
5:40
Min:sec
ZOOM
PAN
concurrency </Code/genunix/lookuppn> (smoothed)
concurrency </Code/genunix/vn_create> (smoothed)
concurrency </Code/ufs/ufs_create> (smoothed)

Figure 12: vn_create time with both the file truncation and DNLC optimizations applied
vn_create once took 40% of Squid's run time (Figure 4); it now takes less than 1%

file per cached HTTP object, Squid should use one huge fixed-sized file for its disk cache, and manage the blocks of this file using its own policies, thus effectively bypassing the overhead of UFS operations.

This optimization would eliminate meta-data update bottlenecks, for several reasons. First, there is no need to synchronously update meta-data in a web *proxy* server because any file corruption, assuming it can be detected, can be worked around by re-fetching the affected file from the server. Similarly, Squid has no use for the UFS feature of synchronously updating time of last modification when writing files. Our measurements suggest that bypassing UFS by managing disk space manually would yield major performance improvements—though at the cost of significantly increasing Squid's code complexity.

5 Related Work

KernInst can insert any dynamically generated code into an unmodified commodity kernel's code space at runtime. Extensible operating systems [3, 6, 15] allow processes to download code into a kernel, but differ from our approach in several ways. First, they are not unmodified commodity kernels. Second, they perform coarse-grained instrumentation; for example, VINO allows C++ classes to customize object methods [16]. Third, they have a limited number of instrumentation points, that must be pre-coded in a manner that allows easy instrumentation; for example, Synthetix [13] replaces a function that is called through a level of indirection by overwriting the appropriate function pointer. We note that KernInst can complement extensible kernels.

Digital's Continuous Profiling (dcpi) [2] measures detailed performance metrics (such as cycles and instruction cache misses) at a fine grain (instruction level) of a commodity kernel, Digital UNIX. Unlike KernInst, dcpi does instrument kernel code in any way. This precludes performance metrics that cannot be readily sampled. KernInst can use dynamic instrumentation to create software-based metrics, which can then be sampled. Furthermore, such metrics gather exact performance data, not statically accurate samples; the accuracy of performance data gathered by KernInst depends on the instrumentation code that writes to various software timers and counters, not on the rate at which they are sampled. KernInst could be used in concert with continuous profiling to provide access to a greater range of performance data.

Paradyn [9] dynamically instruments user programs. Our work differs from Paradyn in several ways: it applies to kernels; instrumentation is fine-grained, whereas Paradyn limits instrumentation points to procedure entry, exit, and call sites; and KernInst instruments without pausing, whereas Paradyn incurs substantial overhead by pausing the application and walking the stack to ensure safe splicing for each instrumentation request.

Static binary rewriters such as EEL [11] and ATOM [17] are fine-grained and allow arbitrary code to be inserted into user programs (and potentially to kernels). However, static rewriting requires the program being instrumented to be taken off-line, instrumented, and re-run for instrumentation to take effect. KernInst, by contrast, is dynamic; instrumentation is deferred until it is needed. Static instrumentation requires instrumenting everything in case it may turn out to be of interest. Dynamic instrumentation allows the user to refine, during runtime, what instrumentation code is of interest.

Kitrace [10] traces kernel code locations. It replaces instructions being traced with a trap, which transfers control to a custom handler. The handler appends an entry to the trace log and resumes execution. Because trap instructions can be inserted at most instructions, kitrace is fine-grained. KernInst differs from kitrace is several ways. First, KernInst does not require a kernel recompile, as does the most recent version of kitrace. Second, kitrace does not insert any code into the kernel. Third, resuming execution after a kitrace trap is more expensive than in dynamic instrumentation. Fine-grained dynamic instrumentation subsumes kitrace because it can insert arbitrary code, instead of just trace-gathering code.

6 Conclusion

We have used dynamic kernel instrumentation to understand two bottlenecks caused by running a heavily loaded Squid web proxy server on Solaris. This case study demonstrates a two-way benefit from kernel measurement, providing information useful for *both* kernel and application tuning. One of the bottlenecks we found, poor DNLC performance, was addressed by changing the kernel (increasing the DNLC size); this shows that kernel profiling is useful to kernel developers. Another bottleneck, superfluous file truncation, was addressed by changing application code; this shows that kernel profiling is useful to application developers. In both cases, optimization was made possible only through the detailed understanding of the kernel's inner workings provided by KernInst; without knowing *why* the open system call had such high latency, we would not have thought of either optimization.

Acknowledgments

We thank Pei Cao and Kevin Beach of the WisWeb group for supplying the Wisconsin Proxy Benchmark; Stephen Chessin and Madhusudhan Talluri of Sun Microsystems for technical assistance and suggestions; and Matt Cheyney, Carlos Figuiera, Karen Karavanic, Tia Newhall, Brian Wylie, and Zhichen Xu for their comments on this manuscript.

References

[1] J. Almeida and P. Cao. Wisconsin Proxy Benchmark 1.0. `http://www.cs.wisc.edu/~cao/wpb1.0.html`.

[2] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T. A. Leung, R.L. Sites, M.T. Vandervoorde, C.A. Waldspurger, and W.E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *16th ACM Symp. on Operating Sys. Principles*, Saint-Malo, France, Oct. 1997.

[3] B.N. Bershad, S.Savage, P. Pardyak, E. Sirer, M. Fiucynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. *15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[4] P. Cao. Personal Communication. July 27, 1998.

[5] A. Cockcroft and R. Pettit. *Sun Performance and Tuning: Java and the Internet.* SunSoft Press, 1998.

[6] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[7] **The GRID: Blueprint for a New Computing Infrastructur**e. I. Foster and C. Kesselman, eds. Morgan Kaufmann, 1999.

[8] J.K. Hollingsworth, B.P. Miller and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools, *Scalable High Performance Computing Conference*, Knoxville, May 1994.

[9] J.K. Hollingsworth, B.P. Miller, M.J.R. Gonçalves, O. Naim, Z. Xu and L. Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. *International Conference on Parallel Architectures and Compilation Techniques,* San Francisco, CA. Nov., 1997.

[10] G. H. Kuenning. Precise Interactive Measurement of Operating Systems Kernels, *Software—Practice & Experience* **25**, 1 (January 1995).

[11] J.R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI),* La Jolla, CA. June, 1995.

[12] National Laboratory for Applied Network Research. Squid Web Proxy Server. http://squid.nlanr.net/Squid/

[13] C. Pu, T. Audrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. *15th ACM Symposium on Operating Sys. Principles,* Copper Mountain, CO. Dec., 1995.

[14] Rational Software. Quantify web site. http://www.rational.com/products/quantify

[15] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI),* Seattle, WA, Oct. 1996.

[16] C. Small. A Tool for Constructing Safe Extensible C++ Systems. 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe, April 1998.

[17] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. *SIGPLAN 1994 Conference on Programming Language Design and Implementation,* June 1994.

[18] A. Tamches and B. P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. *Third Symposium on Operating System Design and Implementation,* February, 1999.