

Linux Whole-System Profiling Study

Brian Greskamp

December 10, 2004

1 Introduction

“Optimize the common case” (Amdahl’s Law) is the mantra of software optimization. To realize the most performance improvement for a given amount of development effort, that effort should be focused on the most frequently executed segments of code. In UNIX, there are three types of code that must be considered: application, kernel, and library code. Any reasonably complex system will exercise all three, usually on behalf of multiple concurrent processes and threads. Traditional profiling tools such as *gprof* are only capable of producing data for a single application at a time and can not capture kernel-level activity. It is often difficult to understand the performance of the entire system from these individual profiles. Consequently, it is useful to profile all three types of code simultaneously across all running processes and threads to provide a complete perspective — a technique called *whole-system profiling*. With whole-system profiling, we can do a single profiling run and capture information about *all* of the code that is executing on the machine. This information can then be used to direct optimization efforts.

Ultimately, the important questions are: Where does the most potential for optimizing whole-system performance lie? Which applications would benefit most from optimizing specific parts of the kernel or libraries? More fundamentally, are these optimizations necessary and economical? This paper attempts to reach some conclusions about these important questions.

This work uses the open-source *OProfile*[1] tool to generate profiles for several common workloads under Linux. Four application domains were chosen for investigation. First, the *video* benchmark consists of playing full-screen DVD video. Second, *graphics* comprises an interactive 3D game that uses the machine’s graphics accelerator. Third, *compilation* builds a large C language program. Fourth, *HTTP* includes three different web-serving benchmarks that measure the behavior of the *apache* server under different loading conditions. Finally, a synthetic “desktop” benchmark contains a random assortment of applications that the author ran during a typical 24-hour period. Together, these benchmarks give performance numbers for a wide range of applications facilitating comparison.

The rest of the paper is organized as follows. Section 2 describes the experimental method. Subsequently, 3 presents the data from each experiment in turn, along with an analysis of each benchmark’s behavior. Later in the section, the potential for optimizing key kernel routines is discussed. Finally, section 4 concludes.

2 Experiment Setup

The results from seven separate profiling sessions are discussed in this paper. For each session, the OProfile daemon is started and an application or collection of applications is executed. The profiling daemon is then stopped and the raw profile data is dumped to disk. Finally, a collection of custom PERL scripts processes the raw data to produce the charts and graphs presented here. For convenience, all tests are performed from within a graphical desktop environment (KDE) on a network-connected machine. Cases where these facts are likely to affect the results are clearly distinguished in the analysis.

The full specifications of the machine used in the experiments appear in Table 1. It is worth noting that the machine is quite old, but the data presented here should be independent of CPU speed since the profiler measures the *fraction* of CPU time spent in each function. Except where noted, all software components are

CPU	550 MHz Pentium III
Memory	256 MB PC100
Graphics	NVidia TNT2 (16 MB)
Network	DEC Tulip 10/100 ethernet
OS	Debian Linux 3.0 (“Woody”)
Filesystem	ReiserFS v3
Kernel	Linux 2.4.18
Compiler	GCC 2.95

Figure 1: Test system specifications

the stock binary packages from the Debian Linux 3.0 distribution. The kernel was custom-compiled for the machine, with device drivers built as modules. All profiling data were gathered with OProfile version 0.8.1.

2.1 Benchmarks

The five application categories were selected to cover a wide variety of computing tasks, from interactive desktop use, to web serving, to multimedia playback. Descriptions of each benchmark follow.

Video: A DVD trailer from *2001: A Space Odyssey* is played in full-screen mode with MPlayer version 0.92

Graphics: This benchmark consists of playing the first couple of courses in TuxRacer, exercising the computer’s 3D video hardware

Compilation: Linux kernel 2.6.9 is compiled with GCC 2.95. Only the kernel image is built (`make bzImage`)

HTTP The Apache HTTP daemon is exercised under three different load models. In the first case, `cached-longfile`, the same 3 MB file is repeatedly downloaded from the server. In the `cached-shortfile` test, a short file is repeatedly downloaded. Finally, in the `unchached` test, the client walks and downloads an entire 300 MB document tree, which contains a mix of large and small files. In all but the `uncached` test, ten simultaneous clients are actively downloading at all times. In `unchached`, only one client is active at any time. Thus the first two tests place the scheduler under high load, while the last does not.

Desktop This workload comprises a typical day’s desktop usage. Tasks include streaming audio, web browsing, playing video, viewing PDF documents, some light console work, and watching TV. No effort is made to account the amount of time spent in each task. Instead, this workload is intended to give an ‘average’ profile over all of the applications a typical user might run. Obviously, this can vary greatly from user-to-user. This workload is not representative of the user who edits video all day, but it should be a reasonable approximation of typical usage.

2.2 Accounting Method

The profiling tool gathers data on a per-function basis for the kernel and for all binaries compiled with debugging support. Such fine granularity can be a distraction when we want a high-level view of system performance. Consequently, we must summarize the data. For this purpose, two different profiles are reported for each benchmark. The “top-level profile” records execution under four categories: user-space applications (*apps*), the C library (*libc*), other libraries (*libs-other*), and all kernel-space code including device drivers (*kern*). The “kernel profile” deals only with kernel-space code (*i.e.* that accounted under the “kernel” category in the top-level profile). In the kernel profile, code is categorized according to where it appears in the source code tree. The five categories for the kernel profile are: filesystem functions (*fs*), network functions including UNIX and IP sockets (*net*), device drivers (*drivers*), architecture-dependent code (*arch*), and everything else (unceremoniously dubbed *kernel*).

Benchmark	kern	apps	libs-other	libc
Video	11.4	76.6	9.0	2.9
Graphics	6.2	53.7	38.8	1.3
Compilation	10.8	73.1	2.5	13.6
Desktop	32.4	19.8	41.3	6.5
HTTP-short	53.3	22.8	1.9	21.9
HTTP-long	86.9	3.8	7.2	2.1
HTTP-uncached	74.0	11.0	6.7	8.3

Figure 2: Top-level profiles for each benchmark

Benchmark	drivers	net	arch	fs	kernel
Video	29.1	9.4	21.9	18.5	20.8
Graphics	39.4	7.6	17.2	16.9	18.6
Compilation	6.7	5.9	16.9	23.8	46.2
Desktop	13.2	17.1	27.8	22.5	19.1
HTTP-short	8.1	34.0	13.9	26.2	17.5
HTTP-long	11.0	39.0	37.8	3.1	8.9
HTTP-uncached	11.9	28.2	21.4	15.4	23.0

Figure 3: Kernel profiles for each benchmark

A bit more clarification of the kernel profile categories is in order since some code could logically fall into more than one class. One can use the following prioritization — starting from highest priority — to resolve any ambiguities: drivers, arch, net, fs, kernel. For example, the driver for the ethernet card is accounted under “drivers” rather than “net”. Similarly, the real-time clock is counted as “drivers” rather than “kernel”. Finally the PERL script used for report extraction leaves uncategorized a small fraction of functions (less than 0.3% in all cases).

3 Results

This section presents the results for each benchmark. One subsection is dedicated to the detailed discussion of each, but first we present Figures 2 and 3 which constitute the most important data. From these we observe that only the HTTP and *desktop* benchmarks spend a significant proportion of time in the kernel. Additionally, for all benchmarks, kernel time is spread broadly across all of the kernel categories. We will see later that very few kernel hotspots exist for any applications.

3.1 Video

As seen in Figure 4, most of the computation for the video playback benchmark occurs in the *mplayer* executable itself. Kernel involvement, shown in Figure 5, is minimal and spread evenly across all categories. One might expect to see more activity in the video drivers here, but they are mostly circumvented due to the way *mplayer* accesses the display. The player simply memory maps the display memory into its own address space and then writes into it directly to display the image. The *driver* time in the kernel is due in small part to the display driver and also to the real-time clock, which the player uses to time the playback of frames. The audio driver registers essentially no processor time.

It is also interesting to note that the amount of time in *fs* is extremely low even though the player is continuously reading data from the DVD at a rate of approximately 1 MB/s. Since the data is read from the disk sequentially and in large chunks, a `read()` call consists mainly of performing a DMA operation to copy a series of contiguous blocks from the drive into host memory. Actually, most of the *fs* time is probably due to the KDE GUI, which was running in the background.

Also of interest is the *net* time, which is non-zero although the video player doesn't use the network at all. Nevertheless, the KDE GUI running in the background makes extensive use of UNIX sockets to pass messages among its components. Additionally, the machine is network-connected, so a steady trickle of packets arriving at the ethernet interface must be processed. For example, other machines on the local network are regularly sending ARP queries to which the machine must respond.



Figure 4: Video benchmark top-level profile

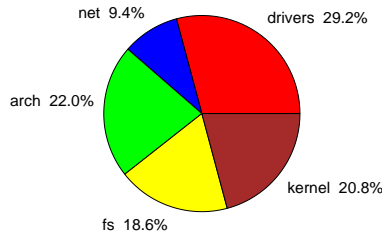


Figure 5: Video benchmark kernel profile

% kern	Function
9.5	mask_and_ack_8259A
5.9	file_read_actor
5.0	unix_poll
3.6	fget
3.5	schedule
3.4	do_select
3.0	enable_8259A_irq
2.9	sock_poll
2.2	IRQ0x08_interrupt
2.1	system_call

Figure 6: Kernel hotspots for the Video benchmark

Figure 6 shows the top ten kernel-space CPU hogs for this benchmark. Consistent with the other benchmarks, `mask_and_ack_8259A` and its companion `enable_8259A_irq` are two major CPU users. The first function is responsible for informing the interrupt controller chip that the kernel has received an interrupt and is processing it. The second function re-enables interrupts after the kernel has finished processing the current interrupt. Such handshaking is crucial because the Linux kernel is not preemptible. Unfortunately, communication with the controller must take place through a series of IO-port reads and writes, which are extremely slow. In fact, the time required to inform the interrupt controller that an interrupt is being processed dominates the time required to actually process an interrupt!

Other time consuming functions include `file_read_actor()` and `fget()`, both of which are invoked in response to `read()` system calls. As already discussed, it is reasonable to expect that they would be very active given the large amount of data being read. Nevertheless, together, they consume only one percent of the total system's CPU cycles.

3.2 Graphics

The graphics benchmark is intended to exercise the graphics hardware. In Figures 7 and 8, we see that although little time is spent in the kernel, an inflated portion of kernel time falls under the *drivers* category. Almost all of the *driver* time is spent within the NVidia kernel module. However, due to the design of X11, most of the OpenGL magic happens in user-space. About 32% of the total CPU time is spent inside of the user-space OpenGL libraries bundled with the video driver. All told, about 34% of the total CPU time was due to the graphics kernel module and its associated user-space libraries.

From the list of kernel hotspots in Table 9, we see a familiar picture. The one notable addition is `do_anonymous_page()`, a function called from within the page fault handler. Counting the OpenGL libraries, the Tuxracer application has a relatively large memory profile, so swapping is occasionally necessary.

3.3 Compilation

This benchmark involves the compilation of hundreds of C source files. Each invocation of `gcc` consists of reading the source file from disk, building and transforming a representation of the program in main memory, and writing out the object file to disk. As shown in Figures 10 and 11, kernel time is in line with that of the previous two benchmarks. In the top-level profile, we notice a large amount of *libc* time because this benchmark involves a lot of dynamic memory allocation (*e.g.* `malloc()`). Accordingly, we

observe that a lot of the kernel-space processing is dealing with the virtual memory subsystem. All of the architecture-independent virtual memory functions are categorized as *kernel* in Figure 11.

Consistent with heavy utilization of the VM subsystem, we see that the kernel hotspots in Table 12 include `page_fault()`, `do_wp_page()`, `do_no_page()`, and the familiar `do_anonymous_page()`. This is to be expected since the compiler is building many large, short-lived data structures which must be paged in on first use. Also of note, the interrupt-handling routines which ranked highly in the previously two benchmarks barely place here. This is because the compilation benchmark does not perform any I/O to the video or sound cards, which generate frequent interrupts.

3.4 Desktop

At first, it is somewhat surprising that the proportion of kernel time in this benchmark is much greater than any of the previous three. Figures 13 and 14 reveal that more time is spent in the kernel than in the application binaries! A large amount of time is also spent in dynamically loaded libraries. In a desktop environment such as KDE, the libraries include a large amount of shared functionality that almost all applications use, such as drawing common widgets and handling the majority of user input events. The large proportion of kernel time is partly an illusion caused by the low over-all CPU utilization. Unlike in the previous benchmarks, the CPU is often idle during typical desktop usage. Average utilization over the profiling period was about 10%. That means kernel functions whose invocation frequency does not change with load are accentuated in the profile. For example, the familiar `mask_and_ack_8259A` function accounts for fully 3% of the non-idle CPU time.

Table 15 reflects the low overall CPU utilization. We see `timer_interrupt()` in the list of kernel hotspots. This function is invoked only 100 times per second. The `schedule()` function consumes only slightly more time than `timer_interrupt`, even though each invocation of `schedule()` is usually more costly than `timer_interrupt()`. This is a consequence of the fact that most of the time, no processes are in the ready list when the timer interrupt arrives.

3.5 HTTP

The HTTP benchmarks stress the network subsystem, and in the case of uncached transfers, also the filesystem. In the two benchmarks involving document trees that fit entirely into memory, the scheduler is also exercised as a load factor of 5 – 10 is maintained at all times.

3.5.1 Uncached Transfers

In this benchmark, the document tree is too large to fit in main memory, and each file is downloaded exactly once, by a single client. The scheduler load factor is less than unity. Figures 16 and 17 show that almost 75% of the time is spent in kernel space. A notable fraction of that occurs in the *net* category. The standard C library is also heavily used. The Apache server makes many system calls, creating sockets, waiting for

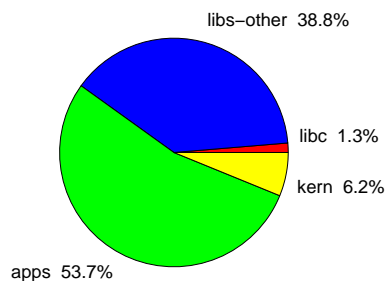


Figure 7: Graphics benchmark top-level profile

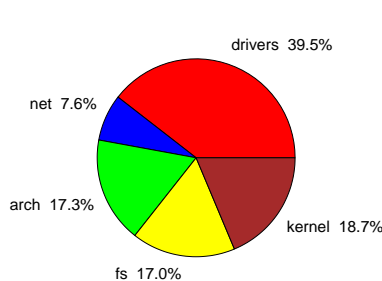


Figure 8: Graphics benchmark kernel profile

% kern	Function
6.7	mask_and_ack_8259A
5.4	unix_poll
4.0	schedule
3.7	do_select
3.6	file_read_actor
3.5	fget
3.0	do_anonymous_page
2.9	sock_poll
2.3	sys_select
2.2	system_call

Figure 9: Kernel hotspots for the Graphics benchmark

data to arrive on the sockets, and forking new processes to handle the incoming requests. The standard C library participates in all of these activities. The apache binary itself consumes little CPU time. It essentially performs a listen-fork-send loop, the substance of which is implemented in the C library and in the kernel. The effort to parse an HTTP request is small.

Table 18 shows some new functions. `search_by_key()` is used within the ReiserFS filesystem to locate the inode for a newly opened file from the path's hash key. The `link_path_walk()` function is also involved in finding a directory entry for a given full path. Frequent calls to `open()` on many small files justify the extra time spent in these functions.

Several new network functions also appear as hotspots. `tcp_sendmsg()` and `tcp_transmit_skb()` are both involved in transmitting TCP packets. `csum_partial_copy_generic()` calculates CRC-16 checksums for outgoing TCP packets and verifies checksums on incoming packets. The checksum generator loops over each byte in each packet, so the amount of time spent checksumming should be directly proportional to the number of bytes transmitted and received via TCP.

3.5.2 Cached Long File Transfers

The CPU usage of the long-file cached HTTP benchmark is summarized in Figures 19 and 20. The fraction of kernel time is even higher than in the uncached case. The proportion of time spent parsing HTTP requests, opening files, and opening sockets is decreased. The raw TCP throughput is much increased, as reflected by the extra time in `csum_partial_copy_generic()`, the TCP checksumming function. In the kernel profile, the checksumming function appears in the *arch* category, since it is architecture-dependent. The reason for having an optimized implementation for each architecture is obvious; in high-throughput server applications, it consumes a large fraction of the total CPU time.

Table 21 shows some differences from the uncached case. First, `system_call()` is no longer a hot spot because fewer system calls are made. Since the requested file is large, the server can pack data into a large user-space buffer before calling `send()` to transmit it all at once rather than performing a smaller transfer to answer each small file request in the uncached case. Moreover, the sustained network throughput is much higher, as indicated by the additional checksumming time and by the appearance of `IRQ_0x09_interrupt`, responsible for servicing interrupts from the network card.

3.5.3 Cached Short File Transfers

In the case that the same small (< 1 KB) file is requested repeatedly, the amount of effort dedicated to receiving queries and to sending responses is almost equal. The overhead of parsing HTTP requests becomes significant and the apache executable itself takes much more time than in either of the previous two cases (see Figure 22). System calls are more frequent, resulting in more time within the C libraries. Even though the raw bitrate is small with such small transfers, the *net* category time in the kernel remains high. The overhead of passing socket buffers through the kernel in the large transfer case has been largely replaced by the overhead of repeatedly setting up and tearing down TCP connections for short transfers.

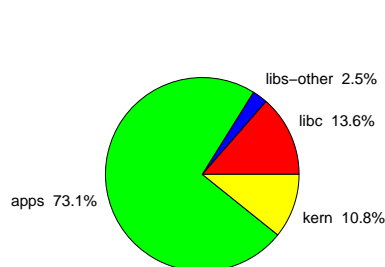


Figure 10: Compilation benchmark top-level profile

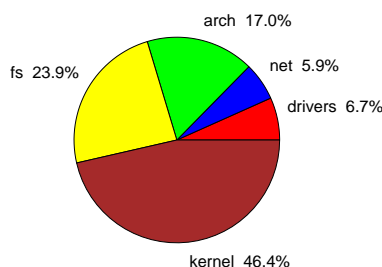


Figure 11: Compilation benchmark kernel profile

% kern	Function
15.4	do_anonymous_page
4.4	file_read_actor
2.5	unix_poll
2.3	do_no_page
2.3	do_wp_page
2.1	schedule
2.1	__generic_copy_from_user
2.0	mask_and_ack_8259A
1.9	do_select
1.9	page_fault

Figure 12: Kernel hotspots for the Compilation benchmark

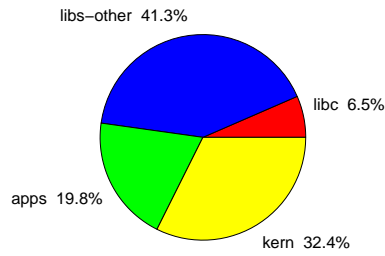


Figure 13: Desktop benchmark top-level profile

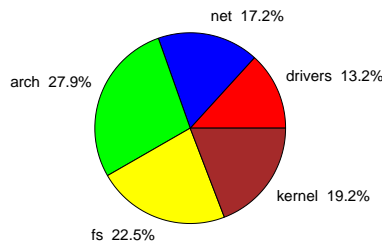


Figure 14: Desktop benchmark kernel profile

% kern	Function
8.9	mask_and_ack_8259A
4.1	unix_poll
3.2	do_select
3.2	fget
3.2	schedule
2.8	sock_poll
2.8	timer_interrupt
2.6	enable_8259A_irq
2.2	system_call
2.2	link_path_walk

Figure 15: Kernel hotspots for the Desktop benchmark

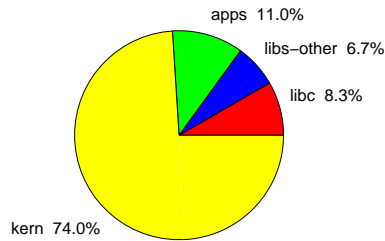


Figure 16: HTTP uncached transfer top-level profile

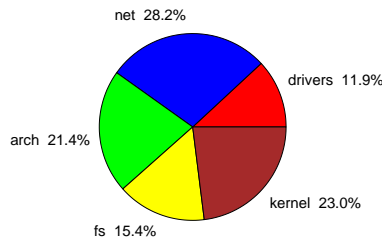


Figure 17: HTTP uncached transfer kernel profile

% kern	Function
11.4	file_read_actor
10.4	csum_partial_copy_generic
5.6	mask_and_ack_8259A
2.0	search_by_key
1.6	tcp_sendmsg
1.6	enable_8259A_irq
1.5	link_path_walk
1.3	tcp_transmit_skb
1.3	IRQ0x09_interrupt
1.2	kmalloc

Figure 18: Kernel hotspots for the HTTP-uncached benchmark

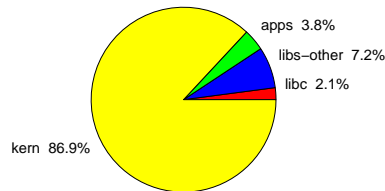


Figure 19: HTTP long file transfer top-level profile

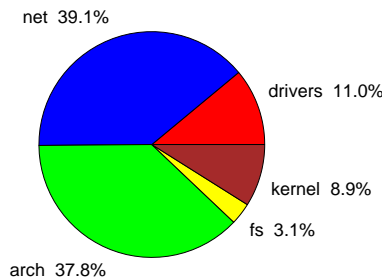


Figure 20: HTTP long file transfer kernel profile

% kern	Function
30.0	csum_partial_copy_generic
7.7	mask_and_ack_8259A
2.6	skb_clone
2.5	_kfree_skb
2.2	tcp_sendmsg
2.2	kfree
2.1	tcp_transmit_skb
2.1	IRQ0x09_interrupt
2.1	alloc_skb
2.0	tcp_clean_rtx_queue

Figure 21: Kernel hotspots for the HTTP-long-cached benchmark

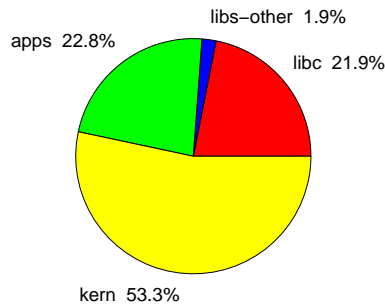


Figure 22: HTTP short file transfer top-level profile

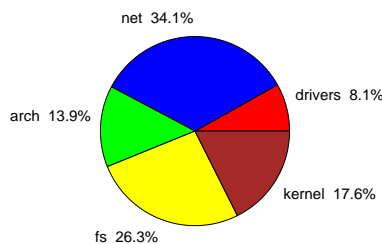


Figure 23: HTTP short file transfer kernel profile

% kern	Function
4.9	mask_and_ack_8259A
3.1	link_path_walk
3.0	search_by_key
2.1	do_wp_page
2.1	system_call
2.1	d_lookup
1.8	reiserfs_readdir
1.7	IRQ0x09_interrupt
1.6	do_anonymous_page
1.4	enable_8259A_irq

Figure 24: Kernel hotspots for the HTTP-short-cached benchmark

Table 24 demonstrates that the biggest kernel bottlenecks here are in the filesystem. The apache server is opening and reading the same file hundreds of times per second. Each time it opens the file, the kernel must invoke the `link_path_walk()`, `d_lookup()`, `search_by_key()`, and `reiserfs_readdir()` functions to locate the inode for the given path.

3.6 Locality Study

To evaluate the potential for optimization in the kernel, it is critical to understand the *locality* of the code. If a large percentage of CPU time is associated with a few functions, then locality is high; if there are no real hotspots then locality is low. To study this relationship, one can use a locality plot such as that shown in Figure 25. It is a cumulative plot showing how many kernel functions (x axis) are needed to account for a given percentage of the kernel time excluding device drivers (y axis). The x axis is logarithmic and includes only functions that were executed at least once for each benchmark. The steeper the curve, the more locality is present.

It may be surprising that the *desktop* benchmark, which consists of a wide variety of applications, has the second highest kernel locality. As already explained, this is mainly because CPU utilization is so low that background functions like the timer interrupt dominate. Therefore, it is not a good candidate for optimization. One area that does look ripe for optimization is the HTTP cached long-file benchmark; it spends about 25% of its total CPU time in `partial_csum_generic()`, which provides a single self-contained optimization target. Unfortunately, that function has already been optimized for each architecture, so it is unlikely that significantly better performance can be achieved for that function. If we could speed up 10% of the active kernel code by 20% (an enormous undertaking), we would succeed in speeding up the system by about 12% for HTTP-long.

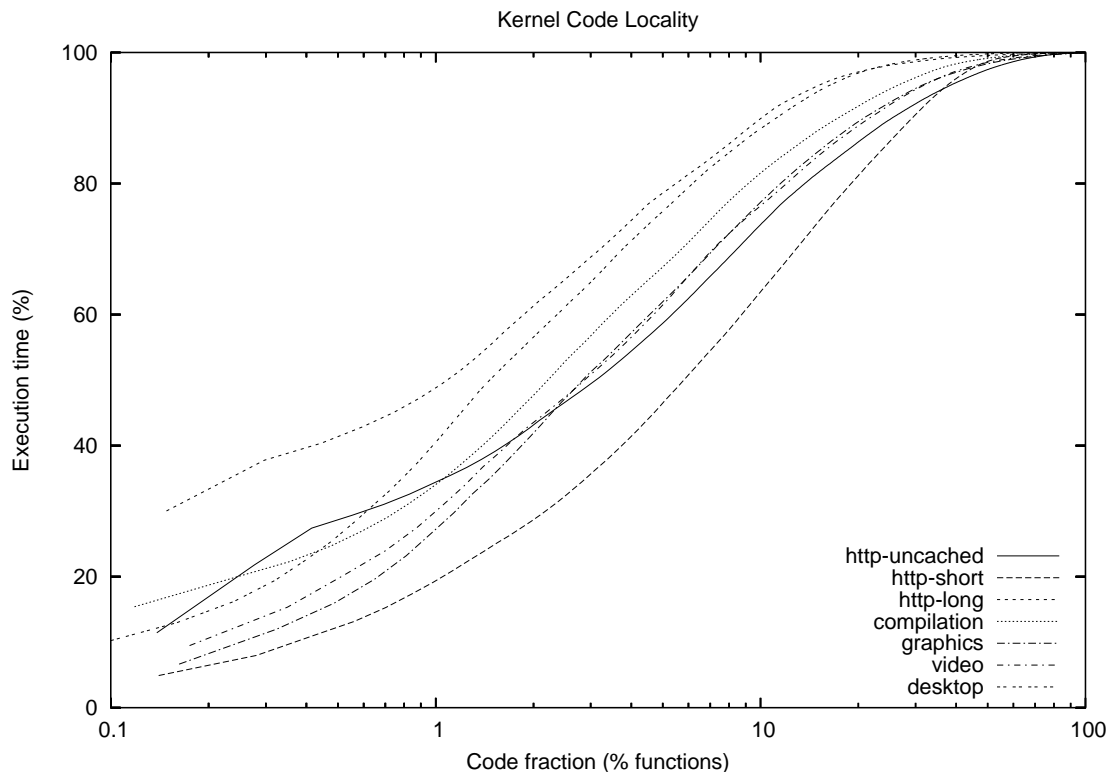


Figure 25: Kernel code locality. y -axis is the cumulative percentage of kernel CPU time excluding device drivers. Note the log scale on the x axis.

4 Conclusions

It is apparent that for the benchmarks this paper has considered, there are few hotspots in a uniprocessor Linux kernel. Any optimization effort would therefore necessarily be spread across many pieces of kernel code. However, it is important to remember that we have been considering a uniprocessor system whereas most of the performance enhancement effort for Linux is now focused on scalability (*i.e.* making enterprise applications perform well on large SMP or NUMA machines). In that field, great strides have been made. In [2], IBM modified the Linux scheduler and synchronization primitives to achieve a ten-fold improvement in SPECweb99 performance on a large SMP system. They also achieved a five-fold improvement in database query performance by making modifications to the I/O subsystem. Many of these optimizations are detailed in [3].

This paper has shown how different classes of applications exercise the kernel and has demonstrated that Linux is well-optimized for uniprocessor performance. It has shown that the kernel is exercised most in server applications and that it accounts for only about 10% of the CPU time in gaming or multimedia applications. The data also shows that the standard C library is not a bottleneck, as it consumes even less time than the kernel in all benchmarks except *compilation*. So, while it appears there is little that can be done to improve uniprocessor performance, it is important that benchmarks such as those in this paper be re-run and evaluated continually to ensure that ongoing scalability work does not adversely affect the performance of small systems.

References

- [1] <http://oprofile.sourceforge.net>
- [2] Sandra Johnson, B. Hartner, and B. Brantley, Jan 2003, *Improving Linux Performance and Scalability*, <http://www-106.ibm.com/developerworks/linux/library/l-kperf/>
- [3] Peter Wai Yee Wong, *et. al.*, *Improving Linux Block I/O for Enterprise Workloads*, OLS 2002.