CITI Technical Report 99-4

The Linux Scalability Project

Peter Honeyman
Chuck Lever
Stephen Molloy
Niels Provos

`linux-scalability@citi.umich.edu`
`http://www.citi.umich.edu/projects/linux-scalability`

ABSTRACT

The Linux Scalability Project is adapting Linux for use in enterprise-scale networking environments. We focus on kernel algorithms and data structures that scale poorly when presented with thousands or tens of thousands of simultaneous service requests. For example, we uncovered a “thundering herd” problem in the `accept` system call. A few dozen lines of code corrects this behavior to awaken only one, instead of all, waiting threads. This small change improves macro-benchmark performance by over 50% on high-performance hardware.

Other examples include improving `poll` performance, adding read-ahead support for `sendfile` and `mmap`ed files, and identifying areas of significant kernel SMP lock contention. The project is also implementing an open source, reference implementation of NFSv4, a highly scalable evolution of the popular distributed file system.

Building relationships between academia, industry, and open source communities is a primary goal of our effort. While our research-oriented, “cathedral” approach is sometimes at odds with the “bazaar” style of software development prevalent in the Linux community, we find ways to build reputation and influence among Linux kernel developers and the growing Linux commercial sector.

November 11, 1999

Center for Information Technology Integration
University of Michigan
519 W. William St.
Ann Arbor, MI 48103-4943

The Linux Scalability Project

Peter Honeyman
Chuck Lever[1]
Stephen Molloy
Niels Provos

Introduction

The Linux operating system is an open-source POSIX-compliant operating system that runs particularly well on commodity Intel-compatible PC hardware. Linux is not only free, it also happens to be very stable, crashing infrequently in a variety of environments. As such, it is widely deployed as a platform for network services, e.g., mail and web servers. Despite high acclaim and ultra-stability, Linux remains a work in progress: Linux continues to be improved in many ways to better position it for enterprise service.

For peculiar reasons, Linux has focused on the desktop, paying special attention to hardware compatibility, user interfaces, and application development. These emphases are critical to Linux' broad acceptance by a worldwide community of users, but have little influence on its usability or appropriateness in an enterprise service environment. The Linux Scalability Project (LSP) at the University of Michigan's Center for Information Technology Integration (CITI) is adapting Linux to meet the needs of enterprise-scale Internet service providers.

LSP is specifically interested in finding immediate and practical improvements to Linux that increase the performance of commercial enterprise servers, such as Sun-Netscape's iPlanet server suite (a collection of web, directory, messaging, and security services). To achieve our goals, our strategy is to select a handful of areas of potential improvement to the Linux operating system, prioritize the areas based on their estimated improvement pay-off versus their implementation cost, then implement the highest priority improvements. We evaluate each improvement using server and OS benchmarking methodologies that are as close to standard as possible to allow scientific comparison with other research in this area.

The specific areas in which we are interested are:

- Reliability — improving system recovery mechanisms, backup/restore, and fault-tolerance.
- Performance — getting the most out of high-performance systems: fast CPUs, RAID systems, and high-speed networking such as ATM and gigabit Ethernet.
- Scalability — improving system throughput, overload characteristics, relieving architectural constraints, and enhancing administration of large installations.
- Security — improving resistance to network and local attacks, reducing or eliminating the risk of buffer overflows, continuous security testing of all bundled applications and utilities.
- Standards compliance — assuring that network implementations are well behaved and that useful and common APIs maintain standards compliance (e.g., POSIX).
- Quality assurance — reducing defect rate (and defect re-introduction).

Improving existing features and adding new ones to an operating system to boost application-specific performance via independent research is an opportunity afforded by Linux' open source distribution. A major challenge for LSP is to find the right ways to work with Linux developers so that our kernel improvements can be easily incorporated into the baseline Linux source code. In the following sections, we outline the areas where we have had the most success and those where we intend to continue our efforts. In addition to technical milestones, we discuss building collaborative relationships among open source advocates, and with system and software vendors.

---

[1] Chuck Lever is a Sun-Netscape Alliance employee resident at CITI.

Prioritizing our work

To decide which improvements provide the most benefit, we assess potential improvements in the following categories.

- Measurable throughput, performance, scalability improvements.

These are the most important gains we hope for. We can estimate these benefits by using research of preexisting literature and simple microbenchmarks.

- Added stability during overload.

While some improvements may provide little, none, or even slightly negative performance or scalability gains, they might offer significant enhancements of system behavior during overload conditions.

- Synergy with other potential improvements.

Several potential improvements to Linux can be accomplished in different ways. Choosing to implement one improvement may make others much simpler to implement.

- Estimated resource costs.

We want to make sure the work we plan is feasible for our developers, and can be completed successfully with the resources available to CITI.

- Estimated complexity.

Complexity relates directly to the amount of testing required, for example, and increases the likelihood that improvements may introduce new bugs. We are also concerned about introducing improvements that require significant changes to applications, especially changes that are not backward compatible.

- Potential introduction of security or scalability problems.

While an improvement might be easily implemented, it also might introduce other problems that make it unsuitable, such as unstable overload behavior or unacceptable security exposures.

- Amount of server re-engineering required.

We favor work that requires few modifications to system interfaces. This lets everyone take advantage of our changes immediately.

- Expectation of acceptance by Linux developers.

We want to see our patches applied to the distributed Linux kernel. Our improvements lose value if they have to be installed or included separately.

- Coordination efforts

These efforts help build collaborative relationships among research and corporate entities to forward the mission of our research.

- Input from industry partner server product teams

We consult with technical staff among our industry partners, such as Netscape's server product teams, to itemize, prioritize, and coordinate LSP plans and efforts.

- Collaboration with Linux development community

We cooperate with members of the Linux development community to determine the current state of Linux, and determine how that work affects high-end server performance. We offer development resources for work on scalability and performance issues.

Network server performance issues

In this section, we summarize some of the performance and scalability issues we have addressed.

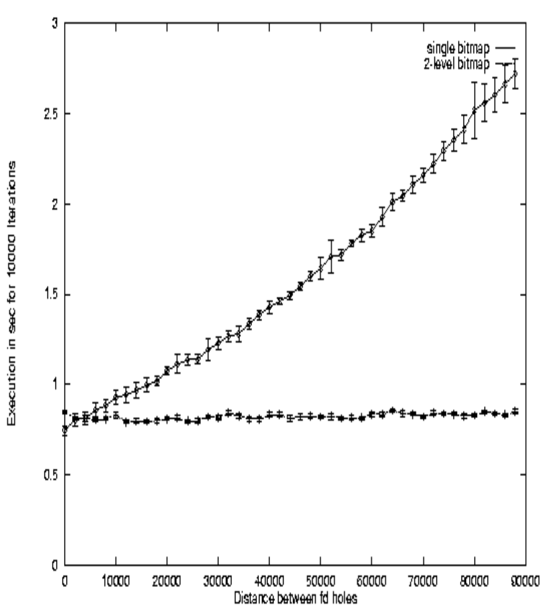
File descriptor scalability

As the number of network users and clients grows, the number of concurrent open file descriptors on network servers can easily exhaust system limits. The number of file descriptors maintained on servers often grows proportionally with the number of concurrent clients served. For example, IMAP servers need to maintain a socket to connect with each client, and an open file descriptor for each client's mailbox. Until recently, a system-wide limit of 1024 file descriptors prevented such a server from supporting more than about 500 concurrent users.

Although the file descriptor limit was extended to 4096 open files, vestigial limits to `poll` and `select` precluded their use with more than 1000 file descriptors. When calling `poll` or `select` each process must allocate a fixed size `wait_table` that can be used by the underlying device drivers to store information about the processes that need to be awakened when an event occurs. Each device driver adds information to the `wait_table` via the `poll_wait` interface. We extended the interface to allocate more storage space dynamically for the `wait_table` if needed.

We also encountered a performance problem for applications that use a large number of file descriptors. In applications with many open file descriptors, the `open` system call searches a linear list in the form of a single bitmap to find a free file descriptor. The result is that the time to find an open file descriptor increases linearly with the number of open files. Linux and *BSD memorize the last closed file descriptor, which allows a single `open` to return quickly. However, when `open` and `close` system calls are interleaved, performance degradation is apparent.

A solution to this problem was suggested by Banga et al. [BM, BDM]: instead of using a single bitmap, a two-bitmap structure is used to maintain the list of free file descriptors. The first bitmap points to free entries in the second bitmap, and the second bitmap points to the free file descriptors themselves. The graph shown below shows the result of a micro-benchmark that opens 100,000 file descriptors and measures the time between closing the first and the n^{th} file descriptors and two subsequent calls to `open`. There is a clear difference between the single bitmap allocation of file descriptors and the two-bitmap allocation.



Hints for `select` and `poll`

A process registers interest in file descriptor events with the `poll` system call. The kernel passes this information to all concerned device drivers and puts the process to sleep until a relevant event occurs. Even though the status of only one file descriptor might have changed, when the process awakens, `poll` is forced to examine all of the selected file descriptors for status changes.

It would be useful if the device drivers could tell — *hint* — each process as to which file descriptors changed their status. To make this possible, we extended `poll` to maintain a list of selected file descriptors and their associated events and processes. When a relevant event occurs, e.g., an interrupt, the driver searches this list to find the associated process and file descriptor. `Poll` then sets the bitmap of changed file descriptors directly, obviating the need to check each driver individually.

Because in general the hinting system requires each device driver to be rewritten, each device driver is given the ability to indicate whether it supports hinting. This way, only the essential drivers — generally speaking, the network devices — need to be modified.

Memory management

Server applications make heavy use of shared regions, anonymous maps, and mapped files. Special features like locking down regions so they aren't swapped, fast `mmap`, support for allocating very large shared regions and memory areas, and efficient memory allocation are especially useful. Many modern network servers use multithreading to take advantage of I/O concurrency and multiple CPUs. As network services scale to tens of thousands of clients per server, their architecture depends more and more on the ability of the underlying operating system to support multithreading efficiently. This especially true for library routines that are provided not by the applications' designers, but by the OS.

An example of a heavily used programming interface that needs to scale well with the number of threads is the memory allocator, known in UNIX as `malloc`. `Malloc` makes use of several important system facilities, including mutex locking and virtual memory page allocation. Thus, analyzing the performance of `malloc` in a multithreaded and multi-CPU environment can provide important information about potential system inefficiency. Finding ways to improve the performance of `malloc` can benefit the performance of any sophisticated multithreaded application, such as network servers.

To test `malloc`'s ability to divide its work efficiently among multiple threads and processors, we wrote a simple benchmark that drives multithreaded loads. In the next subsections, we discuss initial results of the benchmarks and analyze the results with an eye towards identifying areas of the implementation of `malloc` and Linux itself that can be improved.

A look at glibc's `malloc`

Most modern distributions of Linux use glibc version 2.0 as their C library. Glibc's implementers have adopted Wolfgang Gloger's `ptmalloc` [WG] as the glibc implementation of `malloc`. `Ptmalloc` has many desirable properties, including multiple heaps to reduce heap contention among threads sharing a single C library invocation.

`Ptmalloc` is based on Doug Lea's original implementation of `malloc` [DL]. Lea's `malloc` has several goals, including improving portability, space and time utilization, and adding tunable parameters to control allocation behavior. Lea is also greatly concerned about software re-use, because very often, application developers, frustrated by inappropriate behavior of memory allocators, often write "yet another" specialized memory allocation scheme rather than re-use an existing one.[2]

Gloger's update to Lea's work retains these desirable behaviors, and adds multithreading ability and some nice debugging extensions. Nonetheless, because the C library is pre-built on most Linux distributions with debugging extensions and tunability compiled out, it is necessary to rebuild the C library or pre-load a separate version of `malloc` to take advantage of these features. `Ptmalloc` also makes use of both `mmap` and `sbrk` on Linux when allocating arenas. Of course, these system calls are essentially the same under the covers, using anonymous maps to offer processes large, pageable virtual memory areas. Optimizing the allocation of anonymous maps and reducing the overhead of these calls by having `malloc` ask for larger chunks at a time are two possible ways of helping performance in this area.

Benchmark description

We wrote a simple multithreaded program that invokes `malloc` and `free` in a loop, and timed the results on a dual processor 200Mhz Pentium Pro with 128Mb of RAM and an Intel i440FX mainboard. The operating system is Red Hat 5.1, which comes with glibc 2.0.6. We replaced the kernel with 2.2.0-pre4. `Gettimeofday`'s resolution on this hardware is 2-3 microseconds. During the tests, the machine was at runlevel 5, but was otherwise quiescent.

To measure the effects of multithreading on heap accesses, we compare the results of running this program on a single process with the results of two processes running this program on a dual processor, and one process running this test in two threads on a dual processor. This answers several questions:

- How well does thread scheduling compare with process scheduling?
- How well does more than one thread in a process utilize multiple CPUs?
- How well does `malloc` scale with multiple threads accessing the same library and heaps?
- How heavyweight are thread mutexes?

If a `malloc` implementation is efficient, we expect that the two-thread run will work as well as the two-process run. Typically, we find that a poorly performing implementation uses a significant amount of kernel time with a high context switch count as a result of contention for mutexes protecting the heap and other shared resources.

We are also interested in the behavior of `malloc` and the system on which it's running as we increase the number of threads past the number of physical CPUs present in the system. We conjecture that the most efficient way to run heavily loaded servers is to keep the ratio of busy threads to physical CPUs as close to 1:1 as possible. We would like to know the penalty as the ratio increases.

[2] One of the authors of this paper encountered this frustration many, many years ago; see [HB].

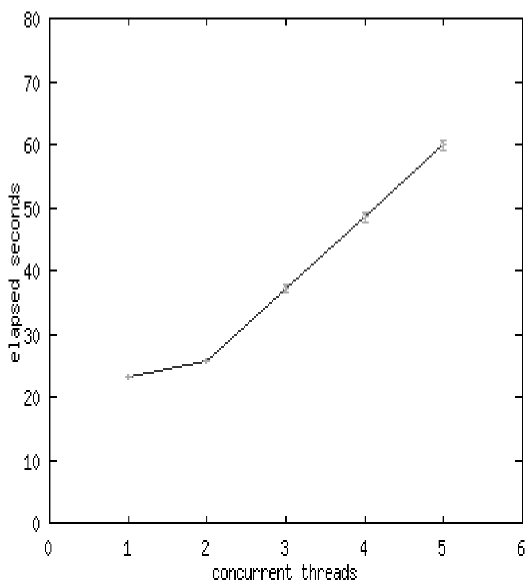
For each test, the benchmark makes 10 million balanced `malloc` and `free` requests. It does this because:

- Increasing the sample size increases the statistical significance of the average results.
- Short timings are hard to measure precisely, so running the test over a longer time allows elapsed time measurements with greater precision.
- Start-up costs (e.g., library initialization) are amortized over the huge number of requests.

Specific tests and results

First, we compare the performance of two threads sharing the same C library with the performance of two threads using their own separate instances of the C library. We hope to learn whether sharing a C library (and thus "sharing" the heap) scales as well as using separate instances of the C library. On our host, the threaded test did almost as well as the process test, losing only about 10% of elapsed time. This indicates that `malloc` scales well as the number of threads sharing the same C library increases.

Next, we examine the behavior of `malloc` as we increase the number of working threads past the

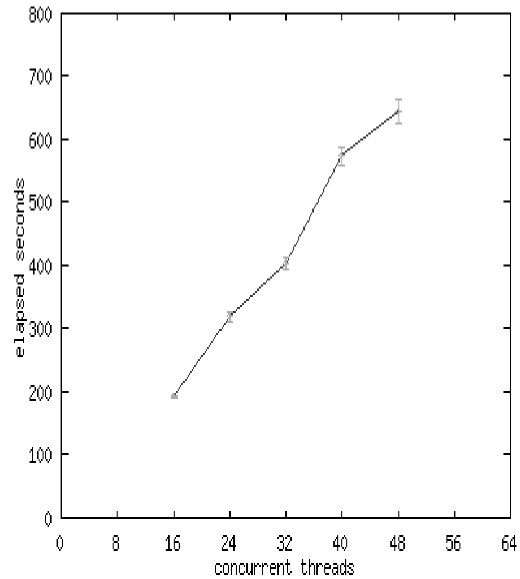


number of physical CPUs in the system. The results are summarized in the following graph.

So far, so good: the average elapsed time increases linearly with the number of threads, with a constant slope of 1/N * M, where N is the number of processors (N = 2 in our case) and M is the number of seconds for a single thread run (23 seconds in our case).

Next, we examine the linearity over a much greater number of threads. This tells us how the library scales with increasing thread count.

The graph shows that the increase in elapsed time is approximately linear with increasing thread counts for counts much larger than the number of configured physical CPUs on the system.



Discussion

We are satisfied that the `malloc` implementation used in glibc 2.0 effectively handles increasing numbers of threads with low overhead, even for a comparatively large number of threads. We found performance to respond linearly to increased offered load.

In the future, we'd like to examine the multi-threaded capabilities of a commercial vendor's `malloc` implementation, e.g., Solaris 2.6 or Solaris 7. Initial tests on single and dual processor Ultra 2s indicate that the Solaris pthread implementation serializes all of the threads created by the `malloc`-test program.

We are also examining the performance relationship between the C library's memory allocator and OS primitives such as mutexes and `sbrk`. We recently removed a global kernel lock from `sbrk`, allowing greater concurrency in memory-intensive multi-threaded applications. (This patch is incorporated in the 2.3.6 kernel.)

We would like to study the effects of allocating in one thread while freeing the same area in another. In addition, we would like to investigate whether memory allocator performance benefits from knowledge about level-2 cache and main memory characteristics. Page coloring or alignment refinements could help promote cache-friendly heap storage behavior.

Buffer management

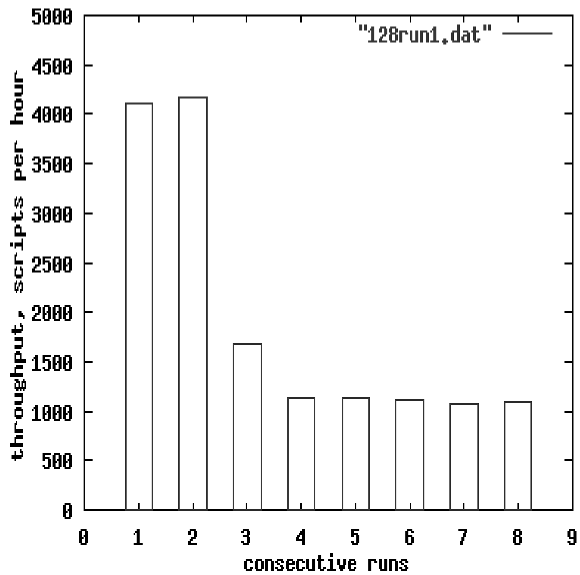
This section describes a bug in the buffer cache that was identified and corrected by LSP staff. Recent releases of Linux feature the ability to self-tune system parameters, such as buffer cache size, according to offered system load. Part of this self-tuning ability is implemented in a varying partitioning of physical RAM between the traditional-style buffer cache and the virtual memory system. However, allowing the buffer cache to grow and shrink on-demand introduces some interesting design problems, and, in this case, a significant bug.

In this section, we describe the bug and how it was identified. We provide performance measurements to show the significance of the problem. Finally, we detail the fix as we proposed it and as it was adopted into the Linux kernel, and report on the performance improvement.

Along with several other developers, we noticed that early releases of Linux 2.2.x were experiencing a memory leak of some kind under heavy file system and VM loads. Symptoms included poor system performance after copying or removing large files, sporadic “out of memory” errors, and performance degradation on long-running jobs. To stress the system and help identify the cause of the problem, we used the SPEC S-DET benchmark suite [SDM] to generate significant loads on a 4-way Dell PowerEdge 6300-450 with 512M of RAM and an 18G Ultra2 LVD SCSI hard drive.

The SPEC S-DET benchmark consists of a script that is designed to emulate a software developer by running programs such as `cc`, `nroff`, `cpio`, and `ed`. The software developer workload stresses many aspects of an operating system, including the file and VM subsystems. Multiple concurrent instances of the script can run to simulate reproducible and increasing amounts of system load. The S-DET benchmark is carefully designed to provide a consistent workload across runs, and to error-check the output of each script to quickly catch problems.

It became quickly apparent that running S-DET with a large number of scripts could reproduce the performance degradation scenario quickly, as illustrated in the following graph, representing performance of consecutive runs of 128 scripts on Linux 2.2.3.



We observed (using `vmstat`) that the buffer cache continues to grow without bounds during the benchmark runs. Eventually, the buffer cache causes the system to begin flushing pages unnecessarily. This pressure on the VM system results in, among other things, pages being stolen back from the buffer cache. Remarkably, the buffer cache doesn't have any aging or replacement policy, so any buffer can be stolen, even buffers for heavily-used data. The system doesn't usually recover from this condition until it is rebooted.

Other features of this scenario include low CPU utilization and a large number of blocked processes. The size of the buffer cache and the size of the free memory list are continually fluctuating, suggesting a high flow of pages into and out of the buffer cache.

After some number of unsuccessful guesses at what might be the problem, it was noticed that the rate at which blocks were being read was low during benchmark runs with good performance, but increased significantly during poorly performing runs. This suggested that the buffer cache was somehow becoming ineffective over time. Linux prioritizes read requests over write requests, assuming that a read request is usually more time-critical, so the elevated read rate interferes with disk write bandwidth.

LSP staff and Andrea Arcangeli, a European Linux developer working for SuSE, independently discovered that buffers were being orphaned. Many buffers were ending up unlinked from the hash, so that they would not be found during a subsequent `find_buffer` request. Each time a buffer is orphaned, another buffer is allocated for the same logical block of data, and another read operation is requested to pull the same data in from the disk. This also causes the buffer cache to grow in size as more and more copies of the same data blocks appear in the cache.

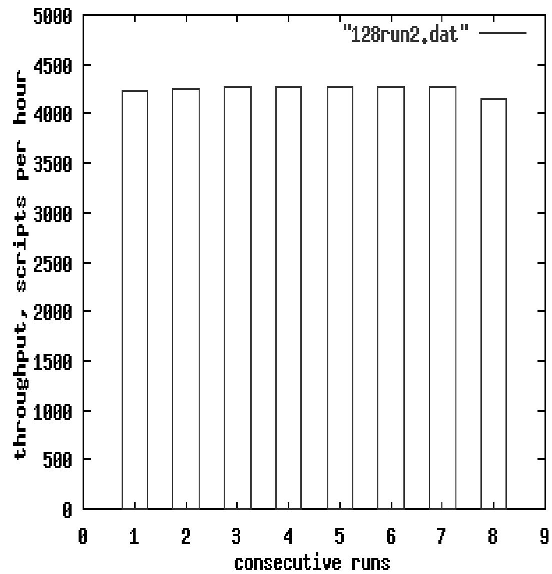
A review of the source that manages buffers (`linux/fs/buffer.c`) revealed that the `bforget` function, used by ext2 when files are truncated or deleted, removes buffers from the hash table, but then abandons them without recycling them.

In Linux 2.0, `refile_buffer` was probably responsible for ensuring that the buffer was properly added to the free list. However, rewrites of the VM and buffer cache subsystems in Linux have since removed that functionality from `refile_buffer`.

A patch was proposed by LSP and Andrea Arcangeli, subsequently modified slightly by Linus Torvalds. The revised `bforget` ensures that a buffer's usage count is zero before inserting it into the free list. In general, a non-zero usage count prevents `try_to_free_buffers` from releasing pages containing buffers. If buffers with non-zero usage counts do appear in the buffer free lists, they are skipped over by `try_to_free_buffers` in favor of buffers that may still have useful data. Having a large number of these buffers in the free list can even cause a severe system-wide memory shortage.

The revised `bforget` corrects the performance degradation observed in the buggy version. `Vmstat` confirms that CPU utilization is maximized, few processes are blocked, block read rate is low, and the buffer cache size expands to a reasonable working set, then remains at a steady size.

The following graph shows that performance is flat for each consecutive run of the 128-script benchmark.



The corrected version of `bforget` is contained in Linux 2.2.5 and later.

Kernel hash table analysis

Hash tables are a venerable and well-understood data structure favored for high-performance applications because of their excellent average search time. The Linux kernel relies on hash tables to manage pages, buffers, inodes, and other data objects. Several hash tables in the Linux kernel are in performance-critical paths. As hardware size and offered load increases on servers, proper allocation of resources to kernel hash tables becomes important to overall system performance and scalability.

For example, on a small machine with 32M of physical RAM, a page cache hash table with 2048 buckets is probably enough to hold all the pages that could be hashed, in chains of less than three. However, this hash table couldn't possibly hold all the pages on a large machine with, say, 512M of physical RAM and continue to maintain short chains to keep lookup times quick. This is an issue for older Linux kernels because these hash tables are statically allocated in fixed sizes for all hardware types.

It is important to understand why hash tables are used in preference to a more sophisticated data structure, such as a tree. Insertion into and deletion from a hash table is **O**(1) if the hashed objects are simply maintained in LIFO order in each bucket. A tree insertion or deletion is **O**(log(n)). Hash table lookup operations are often **O**(n/m) (where n is the number of objects in the table and m is the number of buckets), which is close to **O**(1), especially when the hash function has spread the hashed objects evenly through the hash table and there are more hash buckets than objects to be stored. Finally, if we are careful about our hash table design, we can keep the average lookup time for both successful and unsuccessful lookups low — i.e., less than **O**(log(n)) — by using a large hash table and a hash function that does a good job of distributing the hash key.

Hash tables depend on good average behavior to perform well. This average behavior relies on the actual input data more often than we like to admit, especially if simple shift-add hash functions are used. Therefore, statistical examination of specific hash functions, in combination with specific real world data, can reveal surprising behavior, and can expose opportunities for performance improvement.

Our analysis focuses on several aspects of kernel hash table behavior:

- Statistical "goodness" of hash functions.
- Size of hash table relative to the number of objects it must store.
- Lookup and insertion efficiency.
- Overall system throughput as it changes with hash table parameters.
- Worst-case behavior, which can expose denial-of-service vulnerabilities.

Our methodology measures standard benchmark throughput, as well as statistical behavior of the various hash tables, via instrumentation we added to the Linux kernel. We used the SPEC S-DET workload described above to offer fixed load levels to the system as we varied specific parameters of each hash table. For each table we are interested in, we recorded hash table histogram information to measure the number empty buckets, the percentage of hashed objects contained in small buckets, the largest bucket size, and the bucket size. We are also curious about how hash function randomness changes as table sizes change. We want to demonstrate the positive effects of using dynamic table sizes determined by hardware characteristics, while keeping the simplicity and compute-efficiency of existing hash functions.

There are four hash tables we were especially interested in measuring:

- page cache
- buffer cache
- inode cache
- dentry cache

In Linux, the page cache holds data in active use by processes. The buffer cache holds data moving to and from disk. The inode cache holds VFS inodes (file system metadata), and the dentry cache is a tree representing the file system directory structure. Our analysis shows that increasing the size of these tables significantly improves scalability on large-memory hardware.

We also studied hash function behavior and compare the results of benchmarks where we fix the size of the hash table but vary the hash function. Hash function alternatives include:

- Untransformed key.
- Modulus hashing.
- Multiplicative hashing.
- Shift-add hash function.
- Random table-driven hash function.
- Architecture-specific hash functions.

The result of our work is that the page and buffer caches in the Linux 2.3 kernel are now dynamically sized during system initialization, and the buffer cache hash function has been significantly enhanced to improve the distribution of buffers in the hash table. The inode and dentry caches are undergoing some evolution in the 2.3 kernel and have yet to be retrofitted with dynamic hash table allocation.

`Accept` scalability

This section explores the effects of a "thundering herd" problem associated with the Linux implementation of the POSIX `accept` system call. We discuss the nature of the problem and how it affects the scalability of the Linux kernel. In addition, we identify candidate solutions and considerations to keep in mind. Finally, we present a solution and benchmark it, giving a description of the benchmark methodology and the results of the benchmark.

Offered loads on network servers that use TCP/IP to communicate with their clients is rapidly increasing. A service may elect to create multiple threads or processes to wait for increasing numbers of concurrent incoming connections. By pre-creating these multiple threads, a network server can handle connections and requests at a faster rate than with a single thread.

In Linux, when multiple threads call `accept` on the same TCP socket, they get put on the same wait queue, waiting for an incoming connection to wake them up. In the Linux 2.2.9 kernel (and earlier), when an incoming TCP connection is accepted, the `wake_up_interruptible` function is invoked to awaken waiting threads. This function walks the wait queue and awakens everybody. All but one of the threads, however, will go back to sleep, waiting for the next connection. This unnecessary awakening is commonly referred to as a “thundering herd” problem and creates scalability problems for network server applications.

In the remainder of this section, we explore the effects of the thundering herd problem associated with the `accept` system call as implemented in the Linux kernel. We then discuss the nature of the problem and how it affects the scalability of network server applications running on Linux. Finally, we benchmark the solutions and give the results and description of the benchmark. All benchmarks and patches are against the Linux 2.2.9 kernel.

Current practice

The socket structure in Linux contains a virtual operations vector, similar to VFS inodes, that lists six methods (referred to as callbacks in some kernel comments). These methods are initially pointed at a set of default functions for generic sockets. Each socket protocol family (e.g., TCP) has the option to override these default functions and point the method to a function specific to the protocol family. TCP provides only one of these for TCP sockets. The four most commonly used socket methods for TCP are:

`sk->state_change`
    bound to `sock_def_wakeup`
`sk->data_ready`
    bound to `sock_def_readable`
`sk->write_space`
    bound to `tcp_write_space`
`sk->error_report`
    bound to `sock_def_error_report`

Each of these methods invokes the `wake_up_interruptible` function. This means that extra tasks may be unnecessarily awakened in other sections of the TCP code during the processing of other system calls or protocol states. In fact, while processing `accept`, three methods — `tcp_write_space`, `sock_def_readable`, and `sock_def_wakeup` — are invoked every time, essentially tripling the thundering herd problem.

Because the most frequently invoked socket methods use `wake_up_interruptible`, the thundering herd problem extends beyond the `accept` system call into the rest of the TCP code. In reality, it is wasteful for most of these methods to awaken the entire wait queue. Thus, almost any TCP socket operation unnecessarily awakens tasks and returns them to sleep. This inefficient practice robs valuable CPU cycles from server applications.

Methodology

Our focus is on improving system throughput by eliminating unnecessary kernel state CPU activity. Two metrics can be used to evaluate our solution. The first is the amount of time it takes from the initiation of the TCP connection until all tasks are back on the wait queue. The second is a measurement of throughput under a high load macro-benchmark.

Guidelines

Don't break any existing system calls

If the changes affect the behavior of any other system calls in an unexpected way, then the solution is unacceptable.

Preserve “wake everybody” behavior for calls that rely on it. Some calls — notably `select` — rely on the “wake everybody” behavior of `wake_up_interruptible`. Without this behavior, `select` does not conform to POSIX specifications.

Make the solution as simple as possible without adding too much new code in too many places.

The more complicated the solution, the more likely it is to break something, or have bugs. In addition, we want to keep the changes local to the TCP code insofar as possible so other parts of the kernel don't have to worry about tripping over the changed behavior.

Do not change any familiar/expected interfaces. Do not add extra arguments to existing function calls.

Make the solution general so that it can be used by the entire kernel.

Solutions

One proposed solution to this problem was suggested by the Linux community after the `accept` thundering herd problem was brought to their attention. The idea is to add a flag in the kernel's task structure and change the handling of wait queues in the `__wake_up` and `__add_wait_queue_tail` functions. First, a bit in the state variable of the task structure is reserved for an "exclusive" marking. The `accept` system call is then responsible for setting the "exclusive" flag in the task's state variable and calling `add_wait_queue_exclusive` to add the task to the wait queue.

In handling the wait queue, `__wake_up` walks the wait queue, waking tasks as it goes until it runs into its first "exclusive" task. It wakes this task and then exits, leaving the rest of the queue waiting. To ensure that all tasks that are not marked exclusive are awakened, `add_wait_queue` is complemented by `add_wait_queue_exclusive`, which adds an exclusive task to the end of the wait queue, past all non-exclusive waiters, to ensure that all "normal" tasks are considered first.

The solution developed at CITI stems from the idea that deciding whether a task should be exclusive should not occur when the task is put on a wait queue. The process or interrupt that awakens tasks on the wait queue is better able to determine if it wants to awaken one task or all of them. With these considerations in mind, we added new calls to complement `wake_up` and `wake_up_interruptible`. These new calls are `wake_one` and `wake_one_interruptible`. They are `#defined` macros, just like `wake_up` and `wake_up_interruptible`, and take exactly the same arguments. The only difference is that an extra flag is sent to `__wake_up`, indicating "wake one" as opposed to the default "wake all." This way, it's up to the waker whether it wants to wake one (e.g., to accept a connection) or wake all (e.g., to tell everyone the socket is closed).

For this "wake one" solution we examined each of the methods used with TCP sockets and decided which should call `wake_up_interruptible` and which should call `wake_one_interruptible`. Where we elected to use `wake_one_interruptible`, and the method was the socket default, we created a small function just for TCP to be used instead of the default. We did this so the changes would affect only the TCP code, and not affect any other working socket protocols. If at some point later it is decided that `wake_one_interruptible` should be the socket default, then the new TCP specific methods can be eliminated. Based on our interpretation of how each socket method is used, we arrived at the following solution:

`sk->state_change`
    bound to `tcp_wakeup`
`sk->data_ready`
    bound to `tcp_data_ready`
`sk->write_space`
    bound to `tcp_write_space`
`sk->error_report`
    bound to `sock_def_error_report`

When the LSP patch is applied, all three of the methods used in `accept` call `wake_one_interruptible` instead of `wake_up_interruptible`.

Benchmarks

We took two different approaches to benchmarking the performance impact of the "wake one" and "task exclusive" patches. The first is a simple micro-benchmark that is easy to set up and quick to run. We ran this to get an idea of the "best case" performance improvement. To see if the patch improves performance under high loads, we also ran a large-scale macro-benchmark on the patched kernels.

Micro-Benchmark

The micro-benchmark measures the time for wait queue activity to settle down after a connection is made. A server generates a large number of threads and has each of them accept on the same port. A client program creates a socket and connects to the server. We issue a `printk` from the kernel every time a task is put on or removed from the wait queue. After the client "taps" the server, we examine the output of the `printk`s and identify the point where the connection was first acknowledged (in terms of wait queue activity) and when all tasks finally settled back into the wait queue.

The results are reported as an estimated elapsed time for the wait queue to settle down after an `accept` call. The measurements are not exact, as we did not take any precautions with regard to concurrency control in the `printk`s. Each data point is measured only once as we need only an estimate of what it looks like.. The server was running Linux 2.2.9 on a Dell PowerEdge 6300 with four 450 MHz Xeon processors and 512M of RAM.

Macro-Benchmark

To set up the test harness for this benchmark, we purchased four machines for use as clients against the web server. The four machines are equipped with AMD K6-2's running at 400 MHz and a 100 Mbps Ethernet card. The server is a Dell PowerEdge 6300 with 4 Pentium II Xeon processors and a 100Mbps Ethernet card. The clients and the server are all connected to the server through a 100 Mbps Ethernet switch. All client machines used in the test harness ran the stock 2.2.9 Linux kernel. The server runs Red Hat Linux 5.2 with a stock 2.2.9 kernel as well as our patched 2.2.9 kernel.

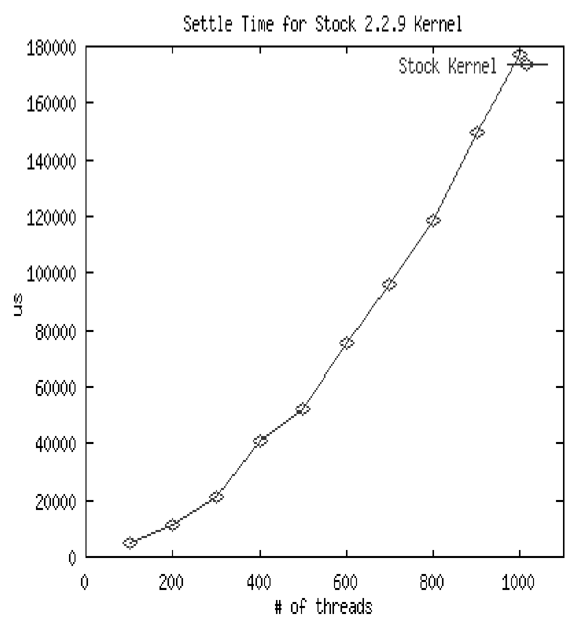
We elected to use the Apache web server on the server host because it's open source and is easily modified to make this test more useful. Stock Apache 1.3.6 uses a locking system to prevent multiple `httpd` processes from calling `accept` on the same port at the same time, which is intended to improve performance and reduce errors in production web servers. For our purposes, we want to see how the web-serving machine will react when multiple `httpd` processes all call `accept` at once. So we modified Apache so that it doesn't wait to obtain a lock before calling `accept`.

To stress test our web server, we used a pre-release version of SPEC's SpecWeb99 benchmark, courtesy of Netscape's web server development team. Because we modified the benchmark's static-dynamic content ratio specifically to hammer the `accept` system call and because the benchmark is pre-release, SPEC rules constrain us from publishing detailed throughput results. However, we are able to report statistically significant throughput improvements.

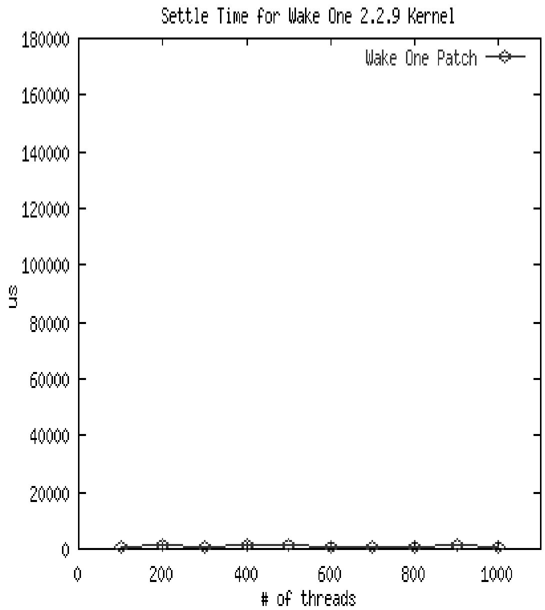
Running the benchmark establishes n simultaneous connections to the web server from the client machines. Each connection requests a web page and then dies while new connections are generated to take their place. These runs of the benchmark request static pages as that will allow it to create more TCP/IP connections per second rather than consuming excess server cycles by running cgi-scripts. This helps generate a higher stress on `accept`. The Apache web server starts 1000 HTTP daemons and increases the number if it deems necessary (which it does occasionally due to lingering connections). All of these daemons accept on the same port.

Benchmark Results

The following micro-benchmark result shows the initial state of the thundering herd problem in unmodified 2.2.9 kernels. The problem is evident: as the number of threads increases, so does the time required to process a call to `accept`.



Running the micro-benchmark on a 2.2.9 kernel with our patch produces the following graph:



The performance improvement is clear: the cost of processing a call to `accept` is constant. We also benchmarked the "task exclusive" approach, and found similar improvements.

The micro-benchmark results show great promise in improving Linux performance by addressing the thundering herd problem in `accept`. However, we also elected to examine overall system performance improvement by running an industry standard macro-benchmark on the modified kernel.

Macro-Benchmark

The results of the macro-benchmark are similarly encouraging. Running with a stable load between 100 and 1500 simultaneous connections to the web server, the number of requests serviced per second increased dramatically with both the "wake one" and "task exclusive" patches. While the performance impact is not as powerful as that evidenced in the micro-benchmark, a considerable gain is evident in the testing. Whether the number of simultaneous connections is at a low level, or reaching the upper bounds of the test, the performance increase due to either patch remains steady at just over 50%. There is no discernable difference between the two patches.

Discussion

We have shown that the thundering herd problem in `accept` is indeed a bottleneck in high-load server performance, and that both the "task exclusive" and "wake one" patches significantly improves the performance of a high-load server. It is our opinion that the "wake one" patch has two main advantages over the "task exclusive" approach.

First, "wake one" does not commit a task to "exclusive" status before it is awakened, obviating the need to handle special cases to completely empty the wait-queue. In this regard, the "wake one" patch can solve any thundering herd problems locally, while the "task exclusive" method may require changes in multiple places.

Second, "wake one" is somewhat cleaner and easier to incorporate into new or existing code.

Read-ahead for `mmap` and `sendfile`

To support applications that deliver streaming media such as audio or video efficiently, an operating system must be aware of some of the unique characteristics of streaming data:

- Usually each page of the stream is read once, so data caching is unnecessary and wasteful.
- Pages are read sequentially from the beginning of the stream to the end, in strict order.
- Streaming data is often larger than a server's main memory.
- Delivery of the pages in a stream is often time-critical.

Mail servers, web servers, and networked database servers fall in the class of servers that can be asked to serve streaming data.

Applications that serve streaming data often use `mmap` to access the data to be streamed. On Linux, the `mmap` implementation reads a cluster of pages out of any file on 64-kilobyte boundaries, but does no prediction about which pages might be accessed next. Linux's `read` implementation can read single pages at a time. Speculative read-ahead helps minimize application latency; we are engineering `mmap` to share this advantage.

Our focus is to provide speculative read-ahead support in two places that can be of great benefit to network server applications: in `mmap`, and in the new `sendfile` system call. It is obvious from its definition that the `sendfile` system call reads data from a file from beginning to end in order, and thus can benefit from nonspeculative read-ahead. The `mmap` system call can be used both for random data access, and for sequential access, thus some speculative heuristics must be used to determine whether to read in small chunks or use read-ahead to help improve read latency. Our implementations should not increase the average latency overhead of `mmap` page faults, the penalty of reading ahead when the data is not used, nor cause significant pollution of the page cache with pages read ahead too far in advance.

Applications that benefit from read-ahead, such as database servers or specialized web servers, may also directly exercise rather precise control over data caching in the operating system. On some POSIX-compliant UNIX systems, the `madvise` and `mincore` system calls provide applications access to information that was formerly exclusively in the domain of operating system caching algorithms. It is a short jump from teaching `mmap` to recognize sequential page faults and invoke a simple read-ahead mechanism, to providing these POSIX APIs to applications.

So far, we've been able to implement `mmap` read-ahead and `madvise` in several versions of the Linux 2.3 kernel. Studying Linux's read-ahead support and `mmap` implementation has allowed us to provide the Linux kernel community with a cleaner implementation of the generic `mmap` logic, and this work appears in revisions of the 2.3 kernel series. Because of recent significant modifications to this area of the kernel to provide support for large-memory hardware and the new write-through page cache, we've been passing our modifications to the Linux community at a rather determined pace. Future work will include:

- Studying the improvement of web serving using our modified kernels.
- Studying the behavior of our modified kernels with artificially created malicious applications.
- Adding read-ahead and network transmission scheduling to `sendfile.`
- Adding `mincore` support.

Scalable distributed filesystems

A related effort of the LSP is the implementation of an open source, reference implementation of the new version of the NFS distributed file system, NFSv4. The NFSv4 protocol standard, under development in the IETF at this writing, holds great promise for highly scalable file systems. Notably, NFSv4 provides a mechanism for multiple requests to be issued in the same remote procedure call, which promises to reduce the latency of many compound NFS requests. In addition, NFSv4 includes scalable mechanisms for insuring consistent access to shared data as well as advanced security features, based on GSS API [GSS].

Our implementation effort is still young, but interoperability testing at an NFSv4 “bake off” in October, 1999 was very promising. Our NFSv4 reference implementation will be available in “alpha” form before the end of 1999; we anticipate a fully functional, well-tested and -documented version in 2000, as the IETF standardization process moves forward.

Current work

Many of the issues discussed above require further exploration. In addition, we have identified several other challenging problems in Linux. In this section, we briefly describe some of these issues.

Hybrid poll/interrupt device driver

We're formulating plans to implement a device driver that intelligently switches back and forth between interrupt mode and polling for work. We intend to modify a stock gigabit-Ethernet driver and create microbenchmarks to measure the improvement.

POSIX RT signals

Traditional methods of managing asynchronous event notification in UNIX, such as `select` and non-blocking I/O, are rapidly approaching, or have passed, their limits of efficiency and usefulness in high-performance environments such as today's network servers. Network server application developers, as well as client developers, seek alternatives to traditional paradigms to address some of the challenges brought on by having to support orders of magnitude more clients per server than ever before.

In UNIX-like operating systems, I/O-ready and -completion events are accomplished via signals. While traditional UNIX signals don't carry a data payload, new POSIX real-time (RT) signals do. This payload can indicate, for example, the identity of the file descriptor that just completed. This would obviate an additional `poll` invocation in order for an application to discover which file descriptor is ready for more I/O. An added benefit of RT signals is that they can be queued in the kernel and delivered to an application in order, one at a time, leaving an application free to pick up the events when it is convenient.

Because the POSIX RT API is fairly new, it is unfamiliar to many application developers. There is significant apprehension, and even some misunderstanding, about its capabilities and limitations. We are studying the few existing applications that have successfully employed the POSIX RT API and creating a primer or road-map for mainstream application developers to use as they discover and begin to use POSIX RT signals. We are also measuring performance improvement, and analyzing these new applications to see how the new architecture compares to other types of server architectures [HPS].

Socket efficiency

Currently it takes much longer to create sockets than it does to simply open files. Furthermore, the memory overhead per socket makes it prohibitive to manage thousands and thousands of sockets per server application. We are investigating ways to make kernel socket management cheaper in terms of CPU and memory resources so that applications can open more sockets. This work extends what we've done on thundering herds and `poll`.

SMP locking strategy

During the most recent kernel development cycle, many areas of the kernel have been freed from the global kernel lock. Consequently, some operations are no longer safe, resulting in new deadlocks or incorrect behavior. In addition, performance is not consistent because of newly introduced timing dependent behavior. We are investigating inter-run performance variance as well as to discover new ways to debug and measure locking problems in the kernel.

madvise

We're developing a Linux version of the POSIX `madvise` system call and plan to compare its performance and behavior to the same call on other systems, e.g., Solaris. This will involve developing test applications and microbenchmarks to measure application behavior while using `madvise`, as well as research into how `madvise` behaves on other open and closed source operating systems.

Future work

In addition to our current development activities, we plan to address a number of critical scalability issues in the Linux kernel. In this section, we discuss some of these plans.

Data throughput

Service scalability depends on the ability of network servers to deliver more and more data at higher and higher rates. Operating system architecture and implementation can have significant effects on data bandwidth. To improve a server's effectiveness, we need to address issues in the operating system and application that limit the amount and rate of data flowing from the server's disk to the network.

On some types of network servers, such as mail servers, the disk read-write ratio is significantly skewed towards writes. Metadata updates and data writes are among the most expensive disk and file system operations. Careful analysis of these operations is of great benefit.

Memory bandwidth is also important in this regard. Memory allocation and system memory management can be optimized to make good use of hardware memory caches. As well, maintaining I/O data cached in main memory can improve overall server efficiency.

Network traffic generated by heavily used network servers exhibits unique characteristics not easily reproduced when analyzing server performance. Clients are often situated behind high latency network connections, resulting in a high degree of server packet retransmission. Packet retransmission creates unnecessary levels of network congestion. Furthermore, servers often maintain an increasingly large number of concurrent connections because most clients are slow to retrieve data, and thus maintain their connection for a longer time.

Research has suggested ways to improve TCP congestion management and startup behavior. The good news is that these changes can be implemented on the server, benefiting server network data throughput without dependencies on client networking software.

Specific OS issues

Lock contention

Locks are used extensively in server applications, so the performance of an operating system's lock primitives is very important. In addition, support for mutexes that can be shared among processes is required.

Multi-processor scalability

To provide more processing power to a server application, we can add more CPUs to a server. First, we must be sure that the operating system and the server application can take full advantage of more than one or two CPUs at a time.

Network servers are generally I/O bound, but increasing the number of CPUs while not directly increasing the I/O bandwidth of a system may have other benefits, such as increasing the amount of CPU available for handling interrupts and processing network protocols. The very latest versions of Linux use MP hardware significantly more efficiently than some earlier versions do. However, there is still room to improve.

Asynchronous events and thread dispatching

Network servers require an integrated approach to asynchronous I/O and thread dispatching. Most modern server architectures make heavy use of both asynchronous I/O and threads. Asynchronous I/O support helps keep the amount of kernel resources and number of outstanding read buffers to a minimum. Having an asynchronous I/O model that is easy to program and allows reuse of server software among various OS platforms is a big win. Most importantly, an OS-provided integrated asynchronous I/O and event-dispatching facility has been shown by researchers to be critical to the performance and scalability of Internet servers.

More flexible and efficient system call interfaces

Under some circumstances, enterprise server products appear to perform better on Windows NT than on UNIX platforms. Many have conjectured that NT has better system call interfaces for network servers than UNIX. A way of improving server performance and scalability is to help the server application itself make more efficient use of the operating system and the resources it provides.

We can do this by adding improved interfaces, or by making the current interfaces, such as `poll`, more efficient. System interfaces should also support 64-bit files and filesystems, as well as very large address spaces and more than a few gigabytes of physical RAM.

Improving memory bandwidth

We plan to implement and measure new versions of `memset` and `memcpy` in the kernel and in the C run-time library that can approach hardware memory bandwidth limits. We will also tune `malloc` in the kernel and the C run-time library to help mitigate latencies in underlying system resource providers, and to help these routines layout memory in a more hardware cache-friendly way.

Improving TCP bandwidth

Several interesting innovations, such as TCP Vegas [BP], can help boost TCP throughput, and reduce latency due to lost packets. We will implement and study a new mechanism that connects recovery processing on one connection to all other connections between the server and a particular client. We will also hope to tune and improve current TCP recovery mechanisms, including SACK [MMF], duplicate ACK, slow start, and fast retransmit. Finally, we plan to analyze Linux's current TCP implementation to check its compliance with TCP standards, e.g., to verify that its congestion behavior is neighborly.

Continuing to extend and benchmark our test harness

To measure our improvements accurately and reproducibly, we will need to extend our test harness to meet the needs of experiments that stress and benchmark Linux and the various network server applications.

Implementing a caching `sendfile`

A Linux implementation of `sendfile` exists, but there may be room for improvement. For example, integrating `sendfile` with the kernel network buffers may improve `sendfile` performance significantly. There may also be opportunities to improve `sendfile` throughput by adjusting the scheduler to process short jobs first [CFH].

Discovering Linux scalability limits

Linux may have some unfortunate system limits that we will need to discover in order to address them within enterprise server software. Examples of such limits might be:

- Small process address space size.
- Small kernel address space size.
- Inability to page most kernel data structures.
- Fixed limits on kernel data structure size.
- 32-bit limits on file system interfaces (like VFS).

This work will attempt to stress various parts of the Linux kernel to determine where its limits lie. We will also engage in research and communication with Linux developers to uncover architected limits, and find ways to relieve the limits.

High-performance filesystems

In the near future, Linux is expected to have a journaling file system, as well as support for 64-bit files. It is important that the underlying filesystem implementations can realistically scale to provide these features. Some such areas might include boosting the ability to create many files in the same directory, providing support for swapping memory-based filesystems, improving the efficiency of metadata operations and data writes, and supporting very large filesystems via variable block sizes (for use with RAID subsystems).

Optimizing PCI performance on multiple buses

This work will require a server configuration with high memory bandwidth and multiple PCI buses. We will study the interaction between the operating system and multiple PCI buses, and try some improvements based on our analysis. In addition, we will explore ways of improving the efficiency of SCSI drivers by increasing the capacity of the device driver to handle overlapped I/O and RAID.

Linux device driver support for ATM cards

ATM networking can help increase server throughput over and above FDDI or fast Ethernet technologies. Therefore, we can explore much further the edges of server performance with ATM. It is not clear, though, how well ATM and other types of high-performance networking are supported in the Linux kernel.

Improved interrupt handling

This work will combine SMP enhancements with the addition of a hybrid polling/interrupt-driven interrupt model to the kernel to allow device drivers to handle batches of interrupts rather than one interrupt at a time. Such support already exists for serial devices; we may find that it significantly improves the performance of disk and high-bandwidth network devices, too.

Zero-copy networking

Reducing or eliminating data copy operations that result from processing a network packet can help improve application data and request bandwidth [MZA]. Mechanisms for improving networking efficiency include checksum caching, reducing the number of data copy operations, and moving data directly from one driver to another without context switches (using a mechanism such as IO-Lite [PDZ]). Some of these changes are easy, but something like IO-Lite would be a significant undertaking.

Benchmark methodologies

To provide truly useful measurements of performance and scalability, we endeavor to select benchmarking systems that academic researchers and industrial engineers use most often. This permits comparison and repetition of our work, increasing its value over time. At the same time, we recognize that no benchmark is able to measure all types of performance problems, so we use other benchmarks as well, usually crafted locally.

There are also cases where we need to examine directly the effects of certain modifications to operating system features. For analyzing OS-specific modifications, McVoy's microbenchmarks and the Byte Linux Benchmarks are very useful. File system benchmarks, such as Bonnie, the Modified Andrew Benchmark, and SPEC’s SDET and KENBUS benchmarks, provide cross-sections of overall system performance.

We are especially interested in application performance, so application-specific benchmarks are typically used to measure our progress. Webstone and SPECweb96 appear to be the standard web server benchmarks. However, S-client and httperf have features that exercise pathological network behavior, and are often useful in judging networking improvements.

For several reasons, we have a strong bias towards web-server benchmarks:

- Using freeware web servers and benchmarks means LSP and others can remain without nondisclosure agreements while still making significant contributions.
- Many web server performance issues are common to other types of network services .
- There are numerous web servers and web server benchmarks available, as well as a body of research literature describing the challenges and pitfalls of measuring web server performance.
- Hardware.

High-speed networking technologies are an integral part of our test harness. Currently, switched fast Ethernet comprises our test harness network. We are preparing to install gigabit Ethernet, and hope to experiment with ATM.

We have multiprocessor CPU hardware on hand to implement and test SMP changes. It may be advisable to use the more powerful machines to drive server loads on smaller machines in the test harness to approach server performance limits more quickly and repeatably.

Testing and evaluation of large-scale server configurations is beyond the scope of this project. We can go as far as understanding compatibility and Linux-specific performance issues with large-scale and esoteric configurations, but our expertise is focused on software optimization. We believe that our operating system optimizations will benefit moderate and large-scale server deployments. Moreover, as our work progresses, we will be better positioned later to investigate large-scale server performance issues.

LSP and the Linux community

As universities and businesses increase their reliance on information technology products and services, the cost of providing infrastructure services also increases, especially as technologies reach the limit of their scalability. Linux offers new hope to many service providers because it provides paradigms that escape the conventional models of software licensing and technical support.

Building relationships between academia, industry, and open source communities is a primary goal of our effort. Yet, our approach is characteristic of academic researchers: we are "cathedral" people [ESR], "individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time." Recognizing that our academic approach is in many ways at odds with the "bazaar" style of software development prevalent in the Linux community, we are pressed to reach beyond the technical challenges and establish a mechanism that influences the design and implementation of Linux in areas where our goals dovetail with those of the Linux core.

LSP nurtures its involvement in the Linux development community by serving as a source of good ideas and by backing up its faith in and respect for open source with source code, patches, and detailed benchmark results. To best serve the Linux community, LSP is forced to take certain risks. For example, when we undertook our analysis of `malloc`, we recognized that the results might not provide LSP opportunities for improvement, as indeed they did not. Yet, our detailed analysis and confirmation of a sound implementation have inherent value to us as researchers and to the broader Linux community.

Similarly, LSP was able to contribute to addressing the thundering herd issue in `accept` by helping to identify the existence and severity of the problem. Although a patch was suggested and integrated by the Linux core almost immediately, we continued to study the problem. Our analysis serves as a means of documenting the issue for current developers and educating future developers. Furthermore, the LSP-suggested patch has architectural features not present in the patch that was adopted, and we continue to press (lightly) to see our solution embraced by others.

In other instances, LSP has succeeded in integrating improvements into the formal Linux distribution channel by applying focused efforts that address specific performance and scalability problems. LSP also serves as a focal point for industry's and academia's mutual agendas by forming a coalition of interested parties that benefit from the leverage gained by our open source policy. Still young, LSP has established and continues to grow a strong reputation for its fundamental contributions to Linux.

Acknowledgements

We gratefully acknowledge input and contributions from many members of the Linux developer community, especially Andrea Arcangeli, Linus Torvalds, and Stephen C. Tweedie. We thank Charles Antonelli and Gary Tyson for providing access to hardware.

This work was partially supported by the Sun-Netscape Alliance, Intel, and Sun Microsystems.

References

[BDM] G. Banga, P. Drushel, J.C. Mogul, "Better operating system features for faster network servers," in *Proc. SIGMETRICS Workshop on Internet Server Performance* (June 1998).

[BM] G. Banga, J. C. Mogul, "Scalable kernel performance for Internet servers under realistic load," in *Proc. of USENIX Annual Technical Conference*, New Orleans (June 1998).

[BP] L.S. Brakmo, L.L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet," *IEEE Journal on Selected Areas in Communication* 13(8) (October 1995).

[CFH] M. Crovella, R. Frangioso, M. Harchol-Balter, "Connection Scheduling in Web Servers," in *Proc. USENIX Conference on Internet Technologies* (October 1999)

[DL] D. Lea, "A Memory Allocator," *unix/mail*, December 1996. See also `http://g.oswego.edu/dl/html/malloc.html`.

[ESR] E.S. Raymond, *The Cathedral & the Bazaar*, O'Reilly & Assoc. (October 1999).

[HB] P. Honeyman and S. Bellovin, "PATHALIAS or The Care and Feeding of Relative Addresses," in *Proc. Summer USENIX Conf.*, Atlanta (June 1986).

[HPS] J.C. Hu, I. Pyarali, D.C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-Speed Networks," *in Proc. 2nd IEEE Global Internet Conference* (November 1997).

[MMF] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018 (October 1996).

[MZA] B.J. Murphy, S Zeadally, C.J. Adams, "An Analysis of Process and Memory Models to Support High-Speed Networking in a UNIX Environment," in *Proc. USENIX Technical Conference* (January 1996).

[PDZ] V.S. Pai, P. Druschel, W. Zwaenepoel, "IO-Lite: A Unified I/O Buffering and Caching System," in *Proc. 3rd Symposium on Operating Systems Design and Implementation* (February 1999).

[SDM] Standard Performance Evaluation Corporation, *System Development Multi-task Benchmark*, `http://www.spec.org/osg/sdm91`, (1991).

[WG] W. Gloger, "Dynamic memory allocator implementations in Linux system libraries," `http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html`.