A Scaling Analysis of Linux I/O Performance

Yannis Klonatos, Manolis Marazakis, and Angelos Bilas

Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
{klonatos, maraz, bilas}@ics.forth.gr

I. Introduction

The widespread availability of multicore processors in server configurations with aggressive I/O resources (I/O controllers along with multiple storage devices, such as solid-state drives) is very promising for increasing levels of I/O performance. However, in this work, we raise concerns about the actual scalability of I/O performance observed in current server configurations. We provide experimental evidence of scalability issues, using the latest stable Linux kernel release (v.2.6.37). We identify scaling limitations based on extensive measurements and analysis of overheads, mostly lock contention, using a synthetic workload, which consists of basic filesystem tasks and employs a high degree of I/O concurrency.

Starting with a fairly common system configuration as a baseline, we find that for several cases I/O performance does not scale adequately with the number of available hardware threads, for a variety of filesystem operations. Moreover, we observe that performance may actually deteriorate when more CPU cores become available. We attempt to mitigate overheads, as identified in our analysis, with targeted tuning. Still, significant scaling limitations remain, motivating a critical re-examination of the entire I/O path in the Linux kernel.

II. Related Work

Scalability in multicore environments, including filesystem performance, is nowadays a hot topic. Extensive analysis and experimentation is currently being performed by the Linux kernel developers themselves, as evident by numerous mail-threads in the Linux Kernel Mailing List. An important result of this ongoing debate has been the introduction in modern kernels of the Read-Copy-Update (RCU) [1] mechanism. The most recent analysis of Linux Scalability is to our knowledge [2]. However, in this publication the authors do not: 1) Examine the scalability of commonly deployed filesystems, such as XFS and EXT4, 2) Consider block-level I/O, as they performed all their storage-related experiments using *tmpfs*, which operates entirely in the DRAM-based buffer-cache. Our work focuses on these two specific aspects, presenting further scalability issues in the common I/O path.

III. Experimental Methodology

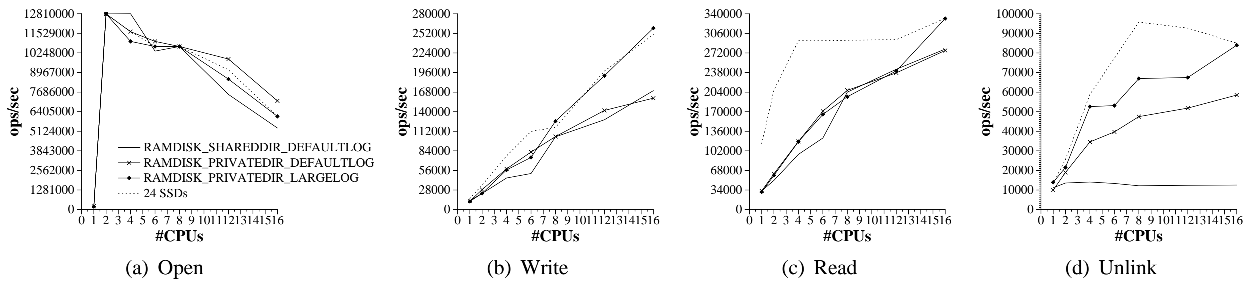
We perform our evaluation on a x86-based Linux server, consisting of the following components: a dual-socket Tyan S7025 motherboard with two 4-core Intel Xeon 5520 64-bit processors running at 2.26 GHz, with two hardware threads per core, 12 GB of DDR-III DRAM, and 24 32-GB enterprise-grade Intel X25-E SLC NAND-Flash SSDs, connected to four LSI MegaSAS 9260 storage controllers. The SSDs are organized in a RAID-0 configuration using the default `md` Linux driver, with a chunk-size of 64KB. The Linux distribution installed is CentOS, v.5.5, with version 2.6.37 of the mainline ("vanilla") kernel. We select this specific kernel because it includes numerous recent scalability related patches. We use two filesystems, XFS and EXT4 for our experiments.

For our evaluation, we use a modified version of the fsmark [3] benchmark. fsmark is widely used by Linux kernel developers because it issues low-level filesystem-intensive operations, a property that makes it suitable for studying filesystem scalability. The original version of the benchmark provides support for open, write, read, unlink and fsync system calls; we have added support for create, seek and stat system calls. We have also modified the benchmark to report operations per second for each system call, instead of average execution times the original version reported. For all our experiments we use 128 application threads, each one executing operations on 128 private files. Each file contains 512KB of data. Finally, we vary the number of CPU threads in the range from 1 to 16.

We collect lock contention statistics using the lock_stat [4] mechanism provided by recent Linux kernels. Lock_stat reports for each lock: i. It's average contention (how many times threads failed to acquire the lock), and ii. The total wait time associated with it. We use the aforementioned information to locate contention bottlenecks.

IV. Experimental Results

We make our experimental analysis in three parts. First, we present scalability issues with the XFS filesystem running on top of a ramdisk. We believe it is a fairly common system configuration to use as a baseline, given the performance gap between DRAM and SSDs. However, since modern server systems can not always rely on DRAM-based caches to achieve the highest performance possible, we then present our scalability concerns using 24SSDs, which provide high throughput and IOPS performance, resulting in the block-devices never being the performance bottleneck. Finally, we show that our observations also hold when using another filesystem like EXT4, instead of XFS.



(a) Open (b) Write (c) Read (d) Unlink

Fig. 1. Operations per second for the open, write, read and unlink system calls, for varying number of cores. We run four configurations: i) Using ramdisk with all threads operating on the same directory (RAMDISK-SHAREDDIR-DEFAULTLOG), ii) Using ramdisk with each threads operating on its own directory (RAMDISK-PRIVATEDIR-DEFAULTLOG), iii) Using ramdisk with each thread operating on its own directory and 2GB log (instead of the default 128MB, RAMDISK-PRIVATEDIR-LARGELOG) and, finally iv) Using 24 SSDs.

To begin with, for the open system call using the ramdisk, we observe the same behavior for all ramdisk configurations. As shown in Figure 1, performance is first increased when moving from 1 to 2 cores, however it then begins to drop as we further increase the available CPU count. The main contention point is the request queue lock, and the wait time is increased significantly (up to 800%) when moving to a higher number of available cores. The directory lock (dlock), used in path resolution, exhibits the same behaviour when moving to more than 4 cores.

Furthermore, for the read and writes system calls using the ramdisk, we observe that there is high wait time for the zone lock. This lock is responsible for holding the buffer cache radix tree organization consistent, when it is accessed through multiple threads. Although we notice that the wait time for this thread is not decreased when we operate on different directories instead of a single one (RAMDISK-SHAREDDIR-DEFAULTLOG vs RAMDISK-PRIVATEDIR-DEFAULTLOG), we notice that this time is significantly reduced when using the larger log option (RAMDISK-PRIVATEDIR-LARGELOG).

Then, for the unlink system call, we observe high contention (>90% of the threads are continuously waiting) for the superblock lock employed by XFS, when using a shared directory. We argue this is due to the fact that the superblock is updated in each unlink, causing the threads to be synchronized at this point. When moving to different directories, XFS spreads out the directories throughout the SSD address space, in partitions called allocation groups, and uses a single superblock for each of them. Thus, less contention is observed for each superblock, and consequently each thread must wait for less time in order to acquire a lock. However, at this point, the XFS log begins to be a significant performance bottleneck. However, if we increase the log size to 2GB, and although the log lock does not disappear from lock_stat output, the wait time observed for this lock is significantly reduced, resulting in significant performance benefits.

Secondly, for the SSD setup, we observe that SSDs exhibit the same performance behaviour for the open and write system calls with the RAMDISK-PRIVATEDIR-LARGELOG setup. However, for the read and unlink system calls, the SSDs reach their peak performance number with less CPUs (4 and 8 respectively), while after this point the request queue lock is the main performance bottleneck. This is important, since resolving the request queue contention point will result in helping both ramdisk and I/O subsystems found in commonly found in Linux server environments.

Finally, we wanted to observe what happens when using another filesystem, rather that the XFS used so far. For this purpose, we run again our benchmark with the EXT4 filesystem. We notice, that although EXT4 achieves significantly less performance compared to XFS for all systems calls (especially for writes), the most significant observation is that EXT4 exhibits the same contention behaviour for the same locks as XFS. This is important, since it reveals that if these contentions are solved or mitigated, both filesystems (and most probably others as well) will significantly benefit.

V. Conclusions and Future Work

Having collected experimental evidence of I/O scaling limitations on modern server configurations, as presented so far, our current work aims in touching several layers along the I/O path in the Linux kernel, including caching for filesystem data and metadata, and efficient submission and scheduling of I/O requests. We are also using more complex benchmarks to emulate server I/O workloads, operating at high I/O concurrency levels. With the rapidly growing gap between processor and I/O subsystem performance, we expect this research direction to become critical for achieving adequate performance levels for data-intensive application workloads.

Acknowledgments

We thankfully acknowledge the support of the European Commission under the 6th and 7th Framework Programs through the IOLanes (FP7-STREP-248615), HiPEAC (NoE-004408), and HiPEAC2 (FP7-ICT-217068) projects.

References

[1] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni, “Read-copy update,” in *In Ottawa Linux Symposium*, pp. 338–367, 2001.

[2] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, “An analysis of linux scalability to many cores,” in *Proceedings of OSDI ’10*, (Vancouver, Canada), October 2010.

[3] R. Wheeler, “fsmark: a file-system stress test,” 2008.

[4] “Lock_stat: a mechanism providing statistics on locks..” http://lxr.linux.no/linux+v2.6.37.2/Documentation/lockstat.txt.

A Scaling Analysis of Linux I/O Performance

Yannis Klonatos, Manolis Marazakis, and Angelos Bilas

{klonatos, maraz, bilas}@ics.forth.gr

Motivation & Previous work

Motivation:

- Widespread availability of multi-core processors
- Scalability is a very hot topic:
  - → **Does the system performance scale with the number of available CPU cores?**
- Server configurations with aggressive I/O resources
  - I/O controllers with multiple storage devices, such as SSDs
  - → **Does system scale when I/O is not the bottleneck?**

Concerns:

- Lock contention / Serialization points
- Percentage of Idle CPUs
- Mitigate overheads with targeted tuning

Related Work:

- Extensive analysis and experimentation by the Linux kernel developers in kernels 2.6.37+
  - Introduction of Read-Copy-Update (RCU)
  - Parallel name lookups
  - Improved file-system level scalability (shown with XFS, EXT4)
- "An analysis of linux scalability to many cores" (OSDI' 10): S, Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M.F. Kaashoek, R. Morris, N. Zeldovich.
  - ➢ Uses tmpfs, a file-system that operates completely in the DRAM-based buffer-cache.
  - ➢ In addition, in our work:
    1. **We examine the scalability of commonly deployed file-systems**
    2. **We consider block-level I/O**
    3. **Perform lock contention analysis**

Lock contention analysis

Questions we address in this work:

1. **Which locks are responsible for scalability problems in the Linux Kernel?**
2. **Are the scalability problems file-system specific?**
   - ➢ No. Both XFS and EXT4 file-systems suffer from the same problems.
3. **Do storage scalability problems exist (even if storage is not the bottleneck)?**
   - ➢ Unlikely, since our SSD raid scales roughly equally with a RAMdisk.
4. **Are there any preliminary & "easy" work-arounds?**

System call	Contention Point	Subsystem in the Linux Kernel	Description
Open	Run queue lock	CPU	Responsible for keeping consistent the list of processes that will run in this CPU
Open	Directory lock	VFS	Used in path resolution, keeps all paths in DRAM for quick lookup
Write	Zone lock	Buffer-cache	Responsible for holding the buffer cache radix tree organization consistent
Read	Zone lock	Buffer-cache	
Unlink	Superblock lock	XFS	Keeps superblock consistent. Superblock is updated in each unlink system call.
Unlink	Log lock	XFS/EXT4	Keeps the FS journal consistent.

Locks of Linux Kernel with high contention (RAMdisk)

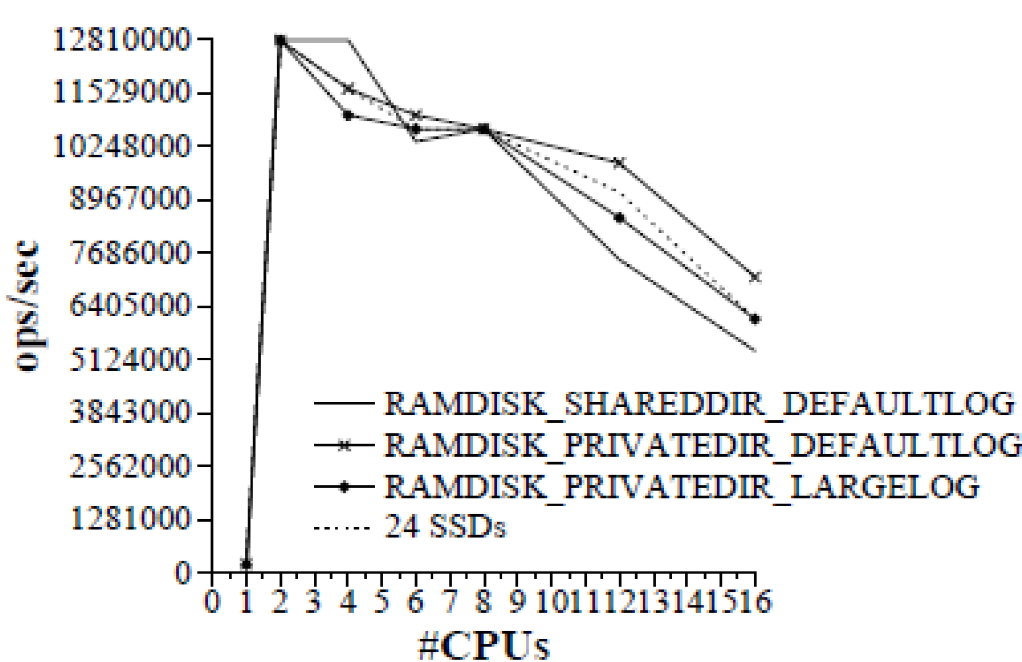
Experimental Setup

- Two 4-core Intel Xeon 5520 64-bit @ 2.26GHz
  - → 2 Hardware threads per core
- 12 GB of DDR-III DRAM
- 24 32GB Intel X25-E SLC NAND-Flash SSDs
- 4 LSI MegaSAS 9260 storage controllers
- RAID 0 Configuration for SSDs with 64KB chunk-size
- CentOS 5.5, vanilla kernel 2.6.37.2
- Two File-systems: XFS and EXT4

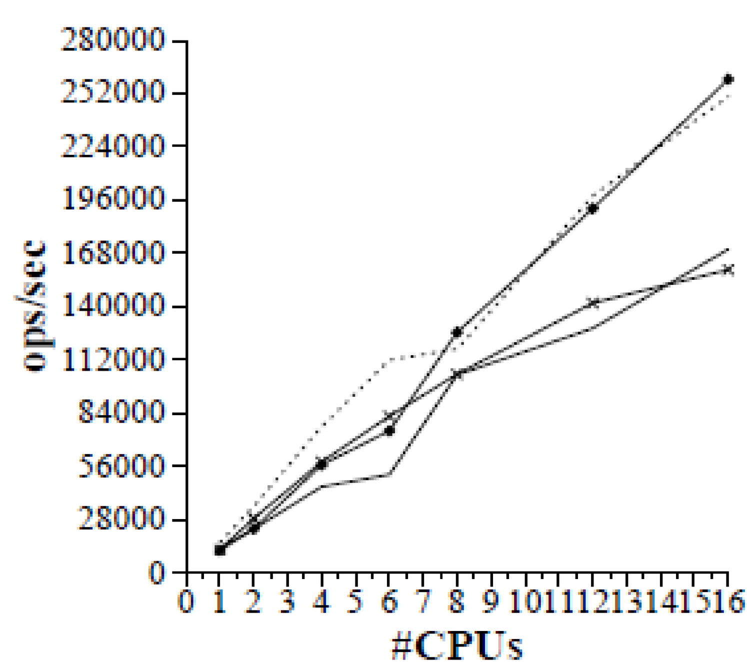
Experimental Methodology

- **Use of modified version of fs_mark benchmark**
  - ➢ Widely used by Linux kernel developers
  - ➢ Issues low-level file-system intensive operations
  - ➢ Original version supports opens, writes and unlinks
  - ➢ Added support for create system call
  - ➢ Our version reports operations per second
- **128 application threads, 128 files of 512 KB data**
- **Vary number of CPUs from 1 to 16**
- **Collect lock contention statistics using lock_stat**

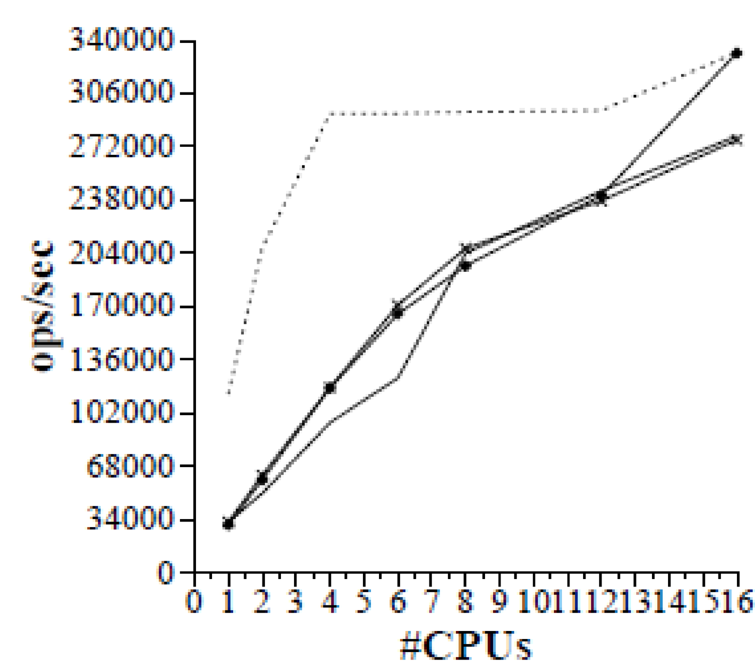
Measuring Kernel Scalability – Preliminary solutions



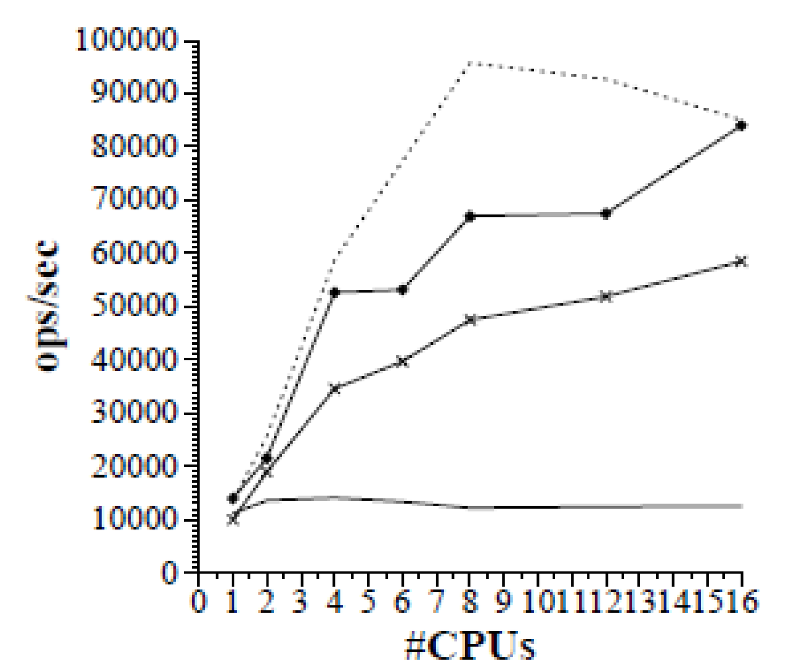
(a) Open



(b) Write



(c) Read



(d) Unlink

Preliminary solutions and work-arounds:

- ➢ Superblock and directory locks:
  - ✓ Using private directory per instead to one shared directory for all threads.
- ➢ Log lock ➔ Using larger lock size reduces contention

Storage – based (24 SSDs, direct I/O) observations:

- Request queue of the RAID device is the main scalability bottleneck.
- Roughly equal performance with RAMdisk for open and write system calls.
- Better performance than RAMdisk for < 16 CPUs for read and unlink system calls.

Open question: Do real applications (databases, file-servers) suffer from the same scalability problems?

Acknowledgements

IOLanes (FP7-STREP-248615), HiPEAC (NoE-004408), and HiPEAC2 (FP7-ICT-217068)

Computer Architecture & VLSI Systems Laboratory (CARV)