2010 Linux Plumbers Conference

KVM / QEMU Storage Stack Performance Discussion

Speakers:

Khoa Huynh – khoa@us.ibm.com

Stefan Hajnoczi – stefan.hajnoczi@uk.ibm.com

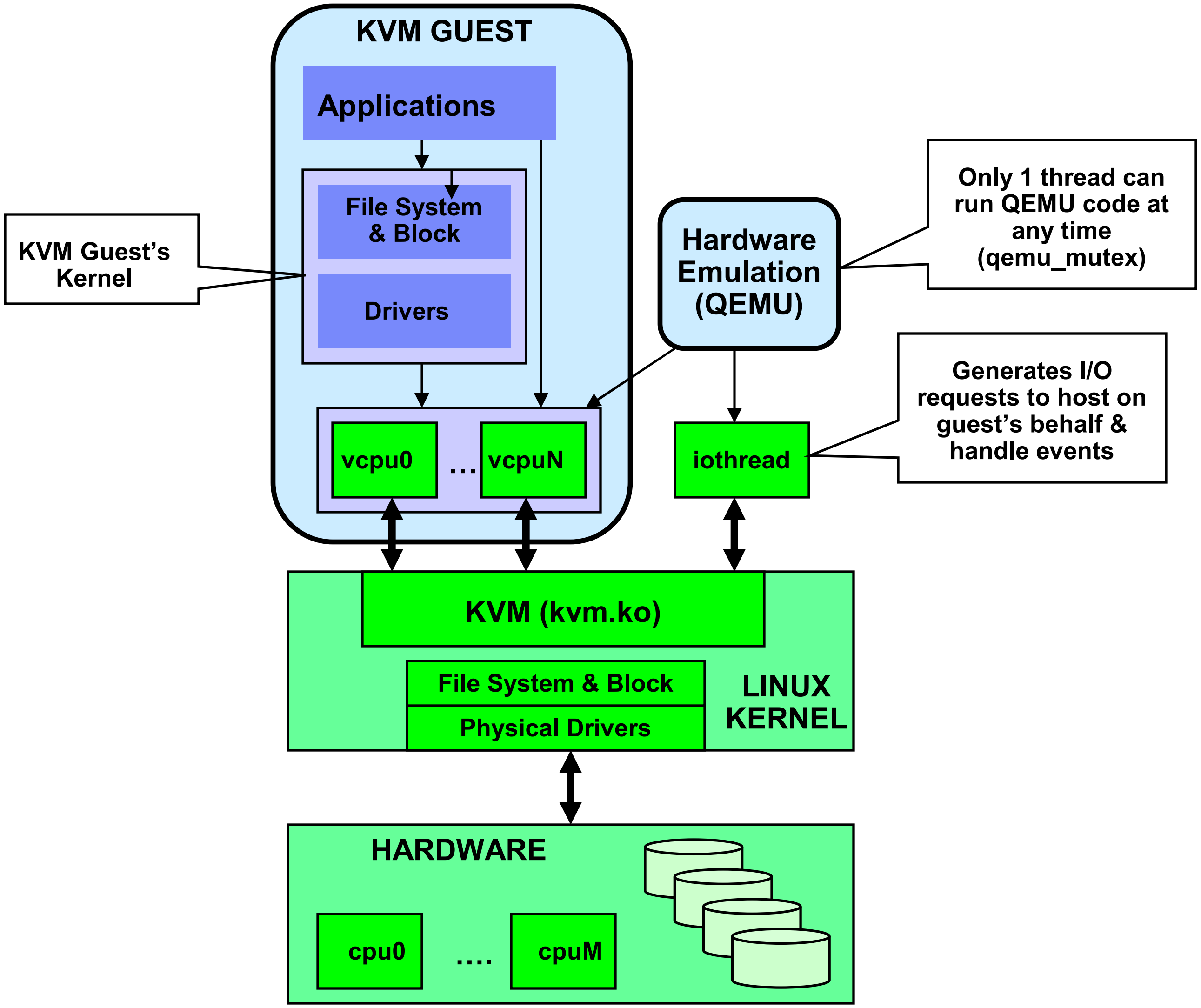
IBM Linux Technology Center

ON DEMAND BUSINESS

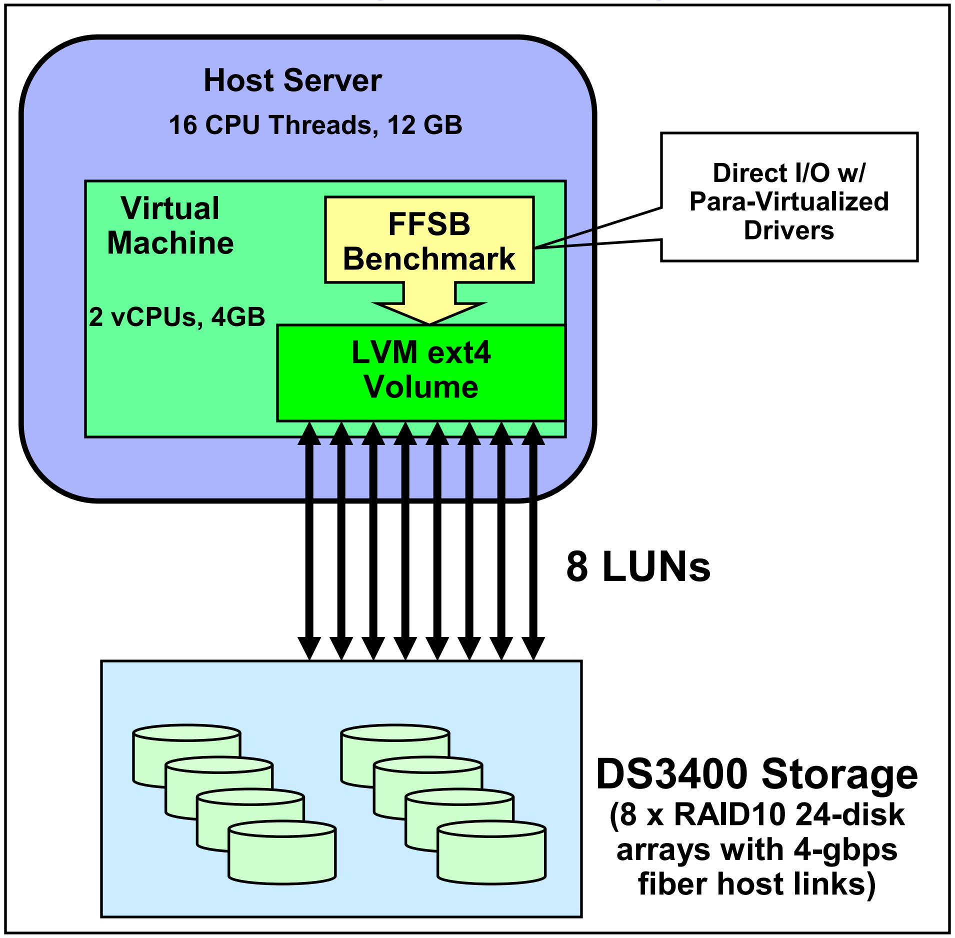
Agenda

- **<u>The State of KVM Block I/O Performance</u>: Where We Are**
  - The 50,000-foot View – QEMU Storage Stack
  - Where We Are Against Another “Popular” Hypervisor
- **<u>Discussion</u>: KVM / QEMU Settings**
  - Virtio vs. IDE emulation
  - Caching Options
  - AIO vs. threads
  - File Systems
  - I/O Schedulers
- **<u>Discussion</u>: KVM Block I/O Performance Issues**
  - Low Throughput
  - High (Virtual) CPU Usage In KVM Guests
  - Virtual Disk Image Formats
  - Others ?

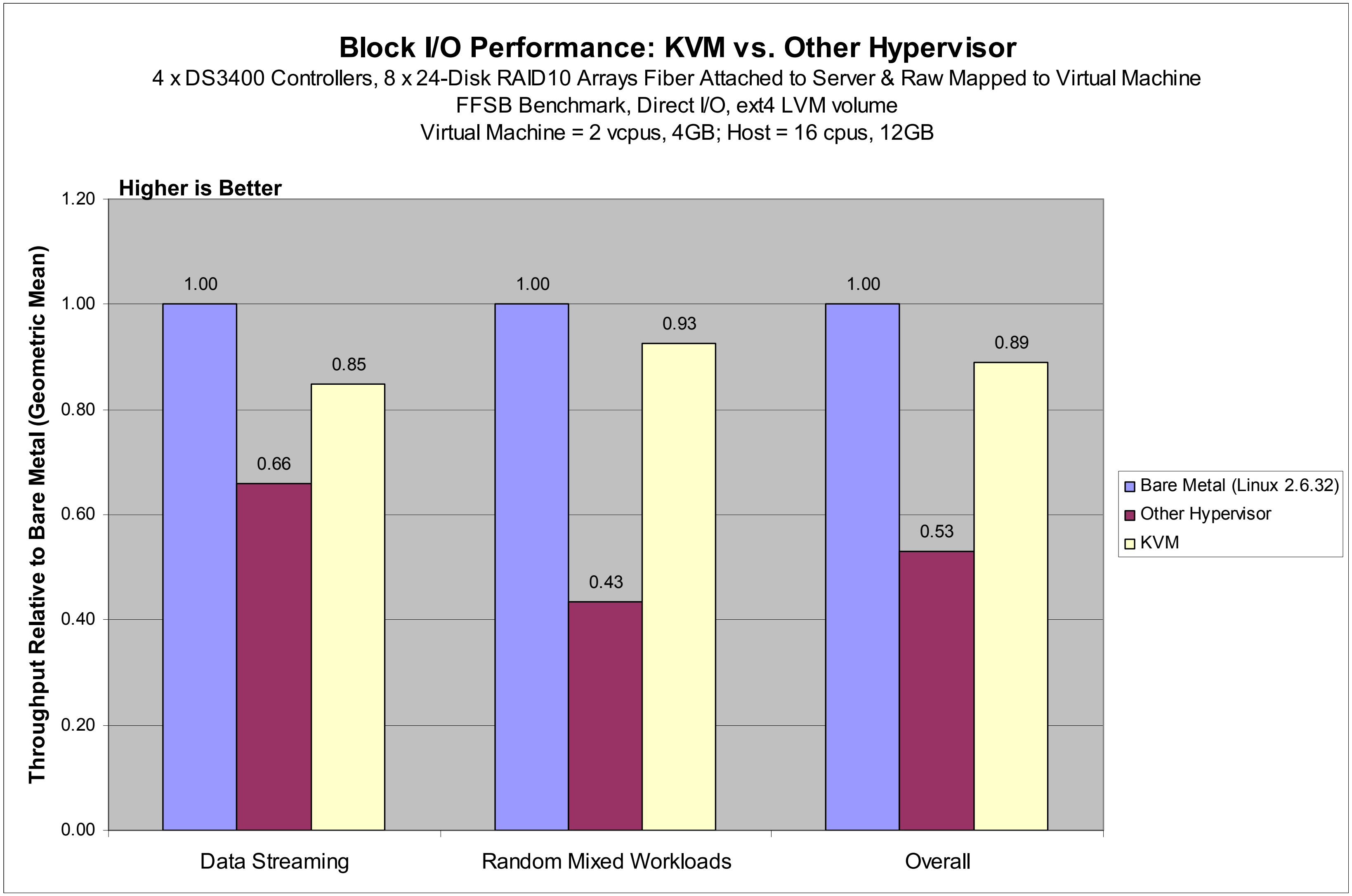
The View @ 50,000 Feet – KVM / QEMU Stack



So…Where We Are ? Let's Take a Look @ A Typical High-End Storage Configuration….

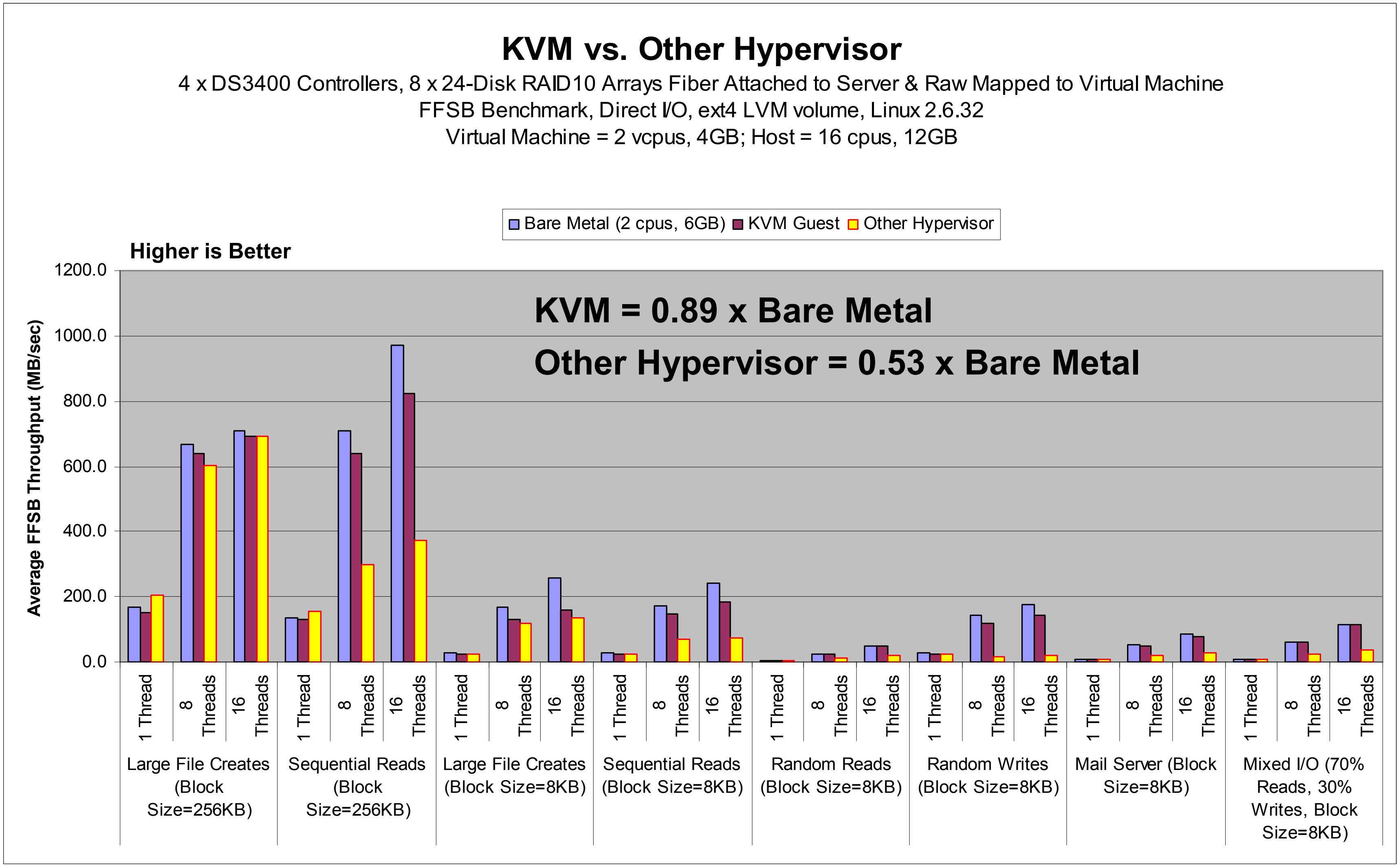


- Host Server: IBM x3650 M2 with E5530 @ 2.40GHz, 8 Cores (16 CPU Threads), 12 GB memory, Chelsio 10-GbE, Broadcom 1-GbE.



Notes:

- Data Streaming (sequential reads, sequential writes) with block sizes of 8KB and 256KB
- Random Mixed Workloads = random reads, random writes, mail server, mixed DB2 workloads (8KB block size)



Notes:

- *Mail Server: random file operations (including file creates, file opens, file deletes, random reads / writes, etc.) to 100,000 files in 100 directories with file sizes ranging from 1 KB to 1 MB.*

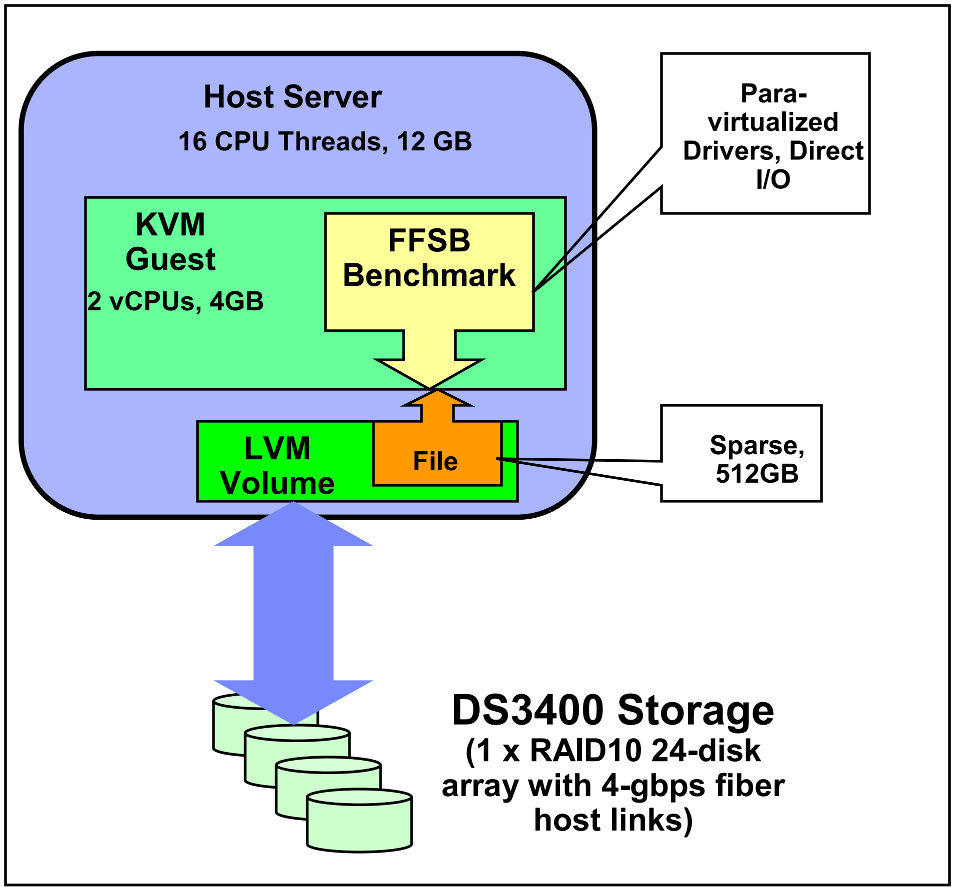
Another Important Metric is the CPU (Virtual and Physical) Usage …

LOCAL FFSB Scenarios	Threads	KVM Guest (2 vcpus, 4GB, Linux 2.6.32) on Host (16 cpus, 12GB)						Other Hypervisor (Guest = 2 vcpus, 4GB, Linux 2.6.32; Host = 16 cpus, 12GB)				
		IOPS	Throughput (MB/sec)	vcpu0%, vcpu1%	Total % (both vcpus)	%vcpu per MB/sec	Host CPU% (Average)	IOPS	Throughput (MB/sec)	vcpu0%, vcpu1%	Total % (both vcpus)	%vcpu per MB/sec
Large File Creates (Block Size=256KB)	**1 Thread**	605.5	151.0	18%, 0%	18%	**0.12%**	2%	824.1	206.0	9%, 2%	11%	**0.05%**
	8 Threads	2556.2	639.0	76%, 28%	104%	**0.16%**	8%	2410.5	603.0	23%, 3%	26%	**0.04%**
	16 Threads	2765.0	691.0	75%, 43%	118%	**0.17%**	9%	2775.7	694.0	23%, 2%	25%	**0.04%**
Sequential Reads (Block Size=256KB)	**1 Thread**	522.4	131.0	14%, 0%	14%	**0.11%**	2%	627.1	157.0	8%, 2%	10%	**0.06%**
	8 Threads	2552.0	638.0	71%, 21%	92%	**0.14%**	7%	1196.1	299.0	12%, 2%	14%	**0.05%**
	16 Threads	3294.2	824.0	86%, 64%	150%	**0.18%**	11%	1488.8	372.0	13%, 2%	15%	**0.04%**
Large File Creates (Block Size=8KB)	**1 Thread**	3090.4	24.1	26%, 0%	26%	**1.08%**	3%	3199.5	25.0	21%, 2%	23%	**0.92%**
	8 Threads	16671.3	130.0	90%, 4%	94%	**0.72%**	8%	15023.6	117.0	54%, 8%	62%	**0.53%**
	16 Threads	20444.4	160.0	98%, 59%	157%	**0.98%**	11%	17153.4	134.0	55%, 13%	68%	**0.51%**
Sequential Reads (Block Size=8KB)	**1 Thread**	3084.7	24.1	16%, 0%	16%	**0.66%**	3%	3112.1	24.3	17%, 2%	19%	**0.78%**
	8 Threads	18648.9	146.0	80%, 2%	82%	**0.56%**	8%	9072.1	70.9	34%, 7%	41%	**0.58%**
	16 Threads	23677.8	185.0	98%, 53%	151%	**0.82%**	11%	9536.5	74.5	35%, 7%	42%	**0.56%**
Random Reads (Block Size=8KB)	**1 Thread**	466.3	3.6	3%, 0%	3%	**0.82%**	1%	449.2	3.5	3%, 0%	3%	**0.85%**
	8 Threads	3252.4	25.4	16%, 0%	16%	**0.63%**	3%	1709.0	13.4	9%, 1%	10%	**0.75%**
	16 Threads	6185.0	48.3	32%, 1%	33%	**0.68%**	4%	2629.1	20.5	12%, 2%	14%	**0.68%**
Random Writes (Block Size=8KB)	**1 Thread**	3134.2	24.5	17%, 0%	17%	**0.69%**	3%	3157.4	24.6	20%, 3%	23%	**0.93%**
	8 Threads	15222.7	118.9	82%, 3%	85%	**0.71%**	8%	1993.4	15.6	8%, 1%	9%	**0.58%**
	16 Threads	18307.3	143.0	91%, 29%	120%	**0.84%**	9%	2780.8	21.7	10%, 2%	12%	**0.55%**
Mixed I/O (70% Reads, 30% Writes) (Block Size=8KB)	**1 Thread**	550.6	7.2	3%, 0%	3%	**0.41%**	1%	606.4	9.3	4%, 1%	5%	**0.54%**
	8 Threads	4172.6	60.9	22%, 0%	22%	**0.36%**	3%	1743.4	24.8	9%, 1%	10%	**0.40%**
	16 Threads	7724.0	114.6	43%, 1%	44%	**0.38%**	5%	2531.7	36.1	12%, 1%	13%	**0.36%**
Mail Server (Block Size=8KB)	**1 Thread**	1081.2	8.5	11%, 0%	11%	**1.30%**	2%	1021.3	8.0	9%, 1%	10%	**1.25%**
	8 Threads	6507.9	50.8	60%, 2%	62%	**1.22%**	6%	2736.6	21.4	17%, 2%	19%	**0.89%**
	16 Threads	9810.4	76.6	77%, 33%	110%	**1.44%**	9%	3756.7	29.3	20%, 4%	24%	**0.82%**

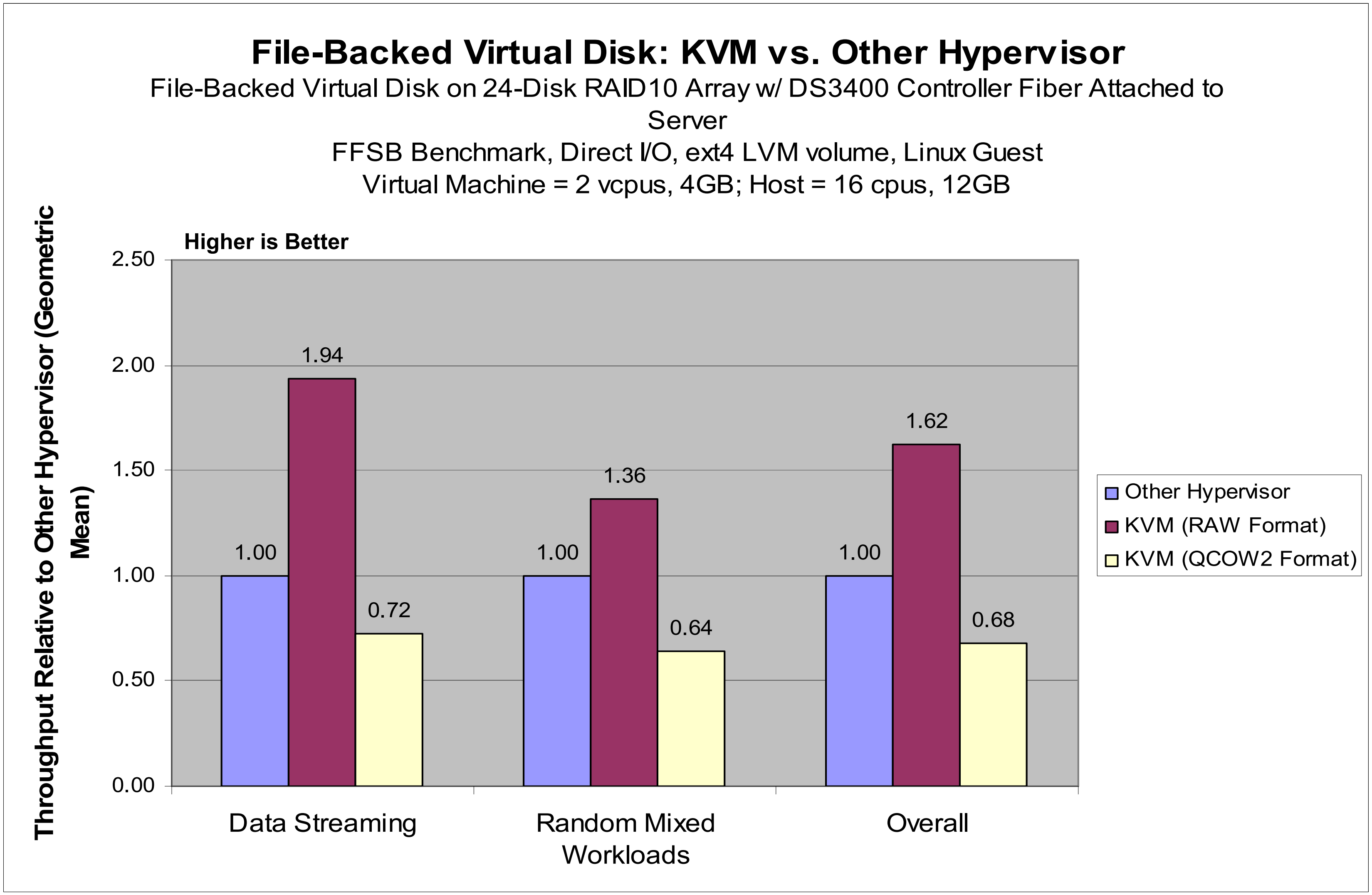
Notes:

- *For virtual CPU usage in the VM, the other hypervisor appears to be better than KVM only for data streaming scenarios; for random and mixed I/O scenarios, KVM is very competitive (even with higher throughput).*

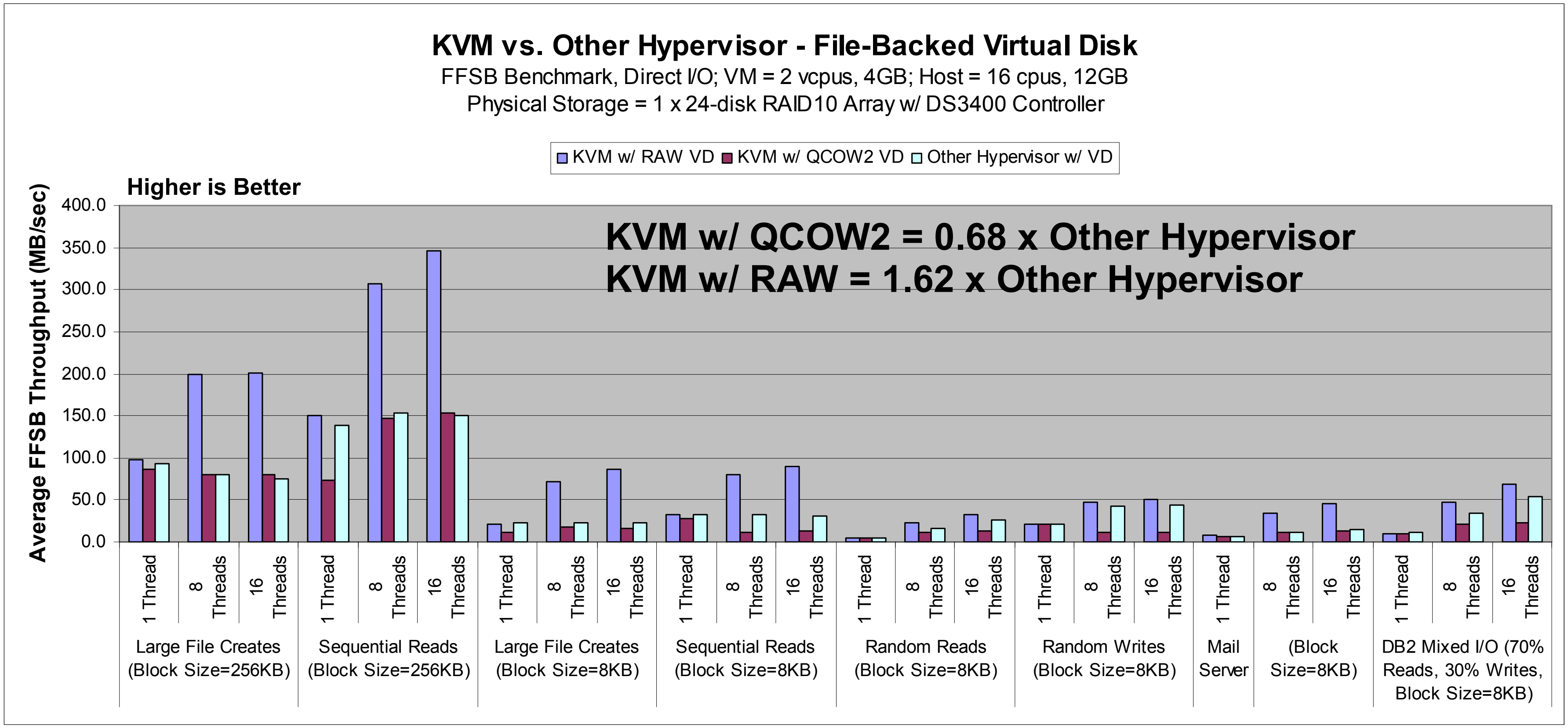
What About File-Backed Virtual Disks ? Let's Take A Look At A Typical Configuration…



- *Storage Node: x3650 M2 (8 x E5530 @ 2.40GHz, 16 Threads, 12 GB memory, Chelsio 10-GbE, Broadcom 1-GbE)*



Notes:
- *Data Streaming (sequential reads, sequential writes) with block sizes of 8KB and 256KB*
- *Random Mixed Workloads = random reads, random writes, mail server, mixed DB2 workloads (8KB block size)*



Notes:

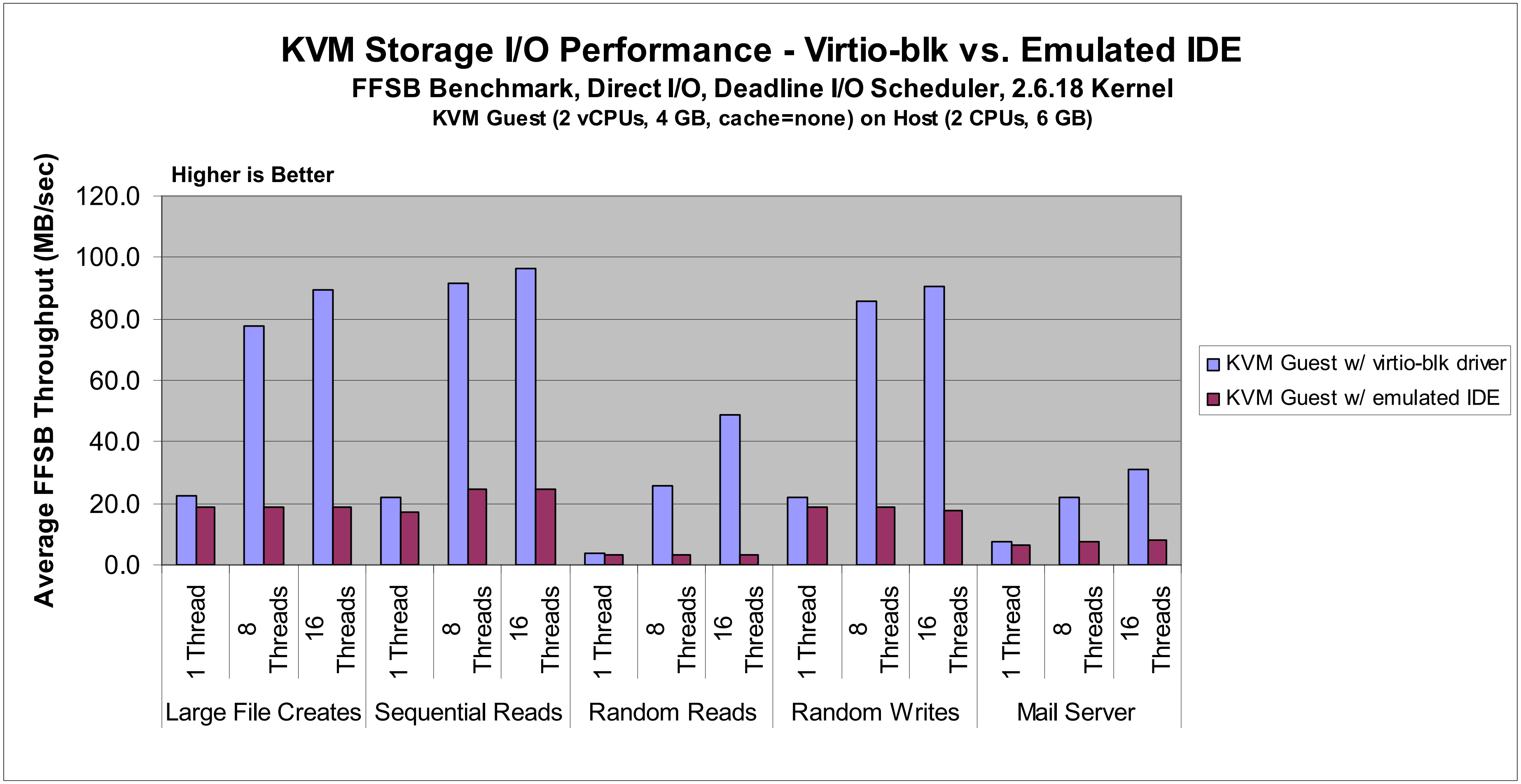
- *KVM w/ QCOW2 performance is worse than other hypervisor in many scenarios.*
- *We proposed a solution – QEMU Enhanced Disk (QED) format – to deliver better performance than QCOW2 (more on this later).*

DISCUSSION – KVM / QEMU Settings For Block I/O

KVM / QEMU Settings For Block I/O

- VirtIO vs. IDE emulation
- KVM caching (cache=none vs. cache=writethrough)
- Linux AIO support
- No Barrier (barrier is enabled by default in ext4)
- Deadline I/O scheduler vs. CFQ
- x2APIC support

IDE Emulation Just Does NOT Scale….

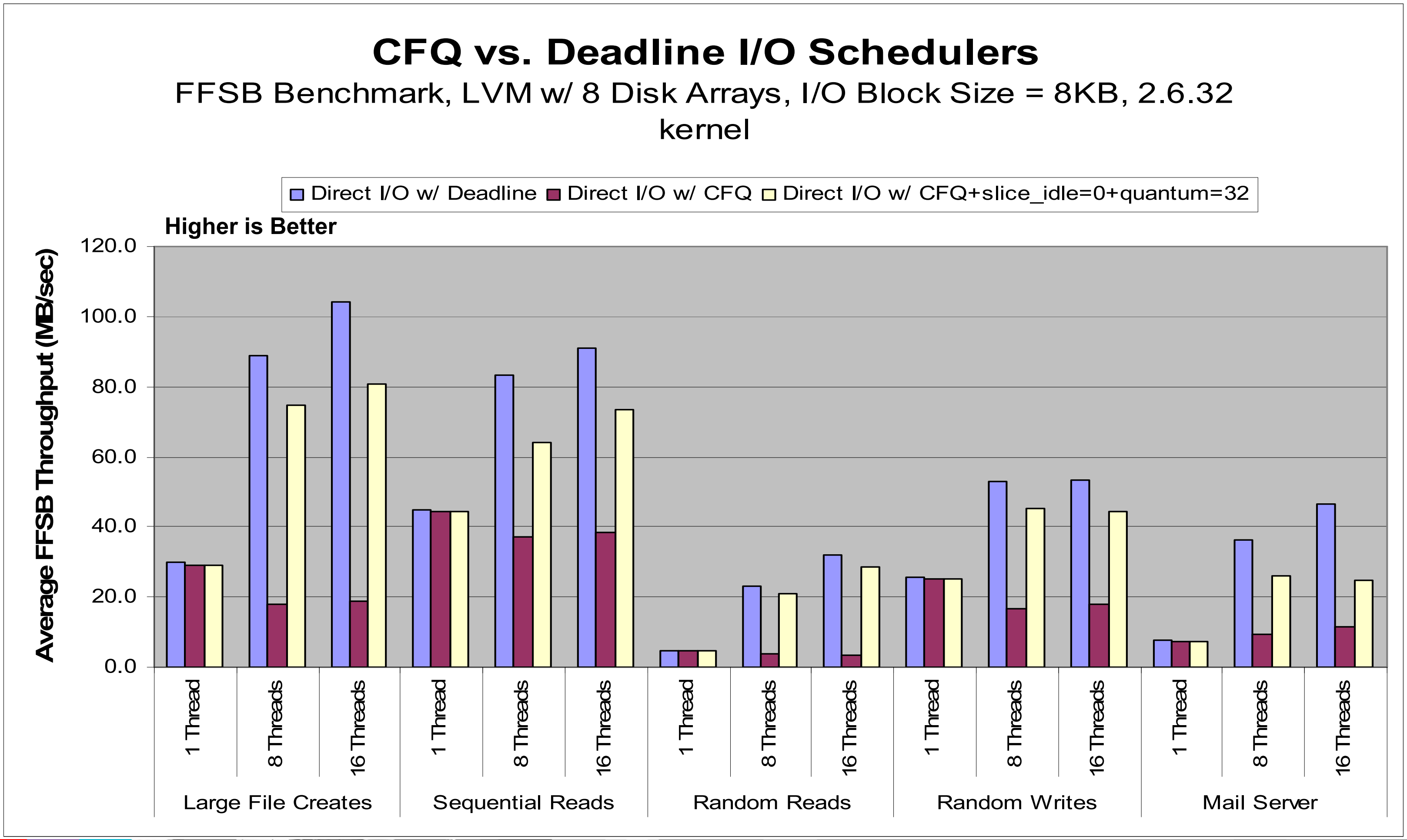


Notes:

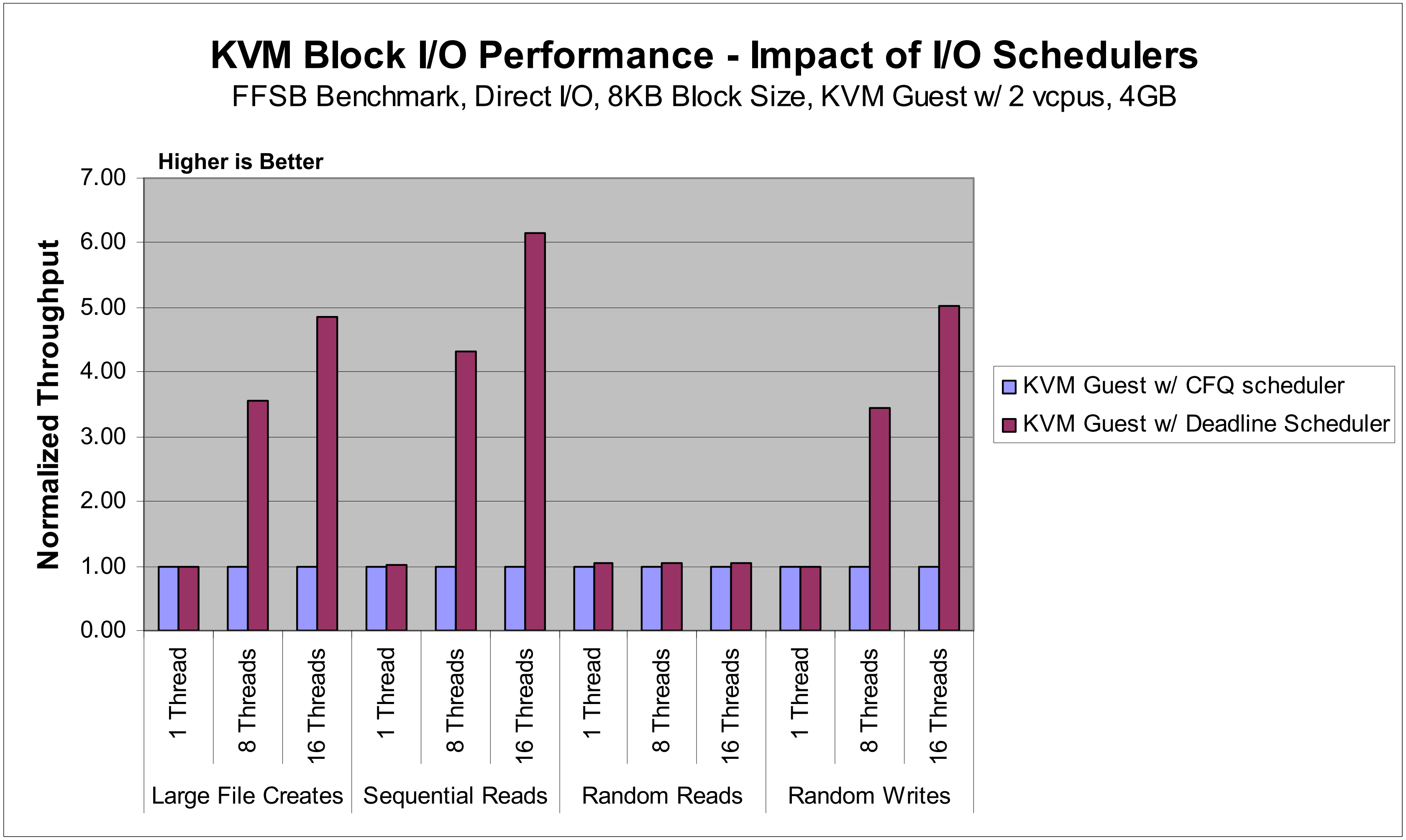
- *IDE emulation does NOT scale because it can only support one outstanding I/O request at a time.*

Deadline Scheduler Is Better (Scales Better) For Enterprise Storage Systems…

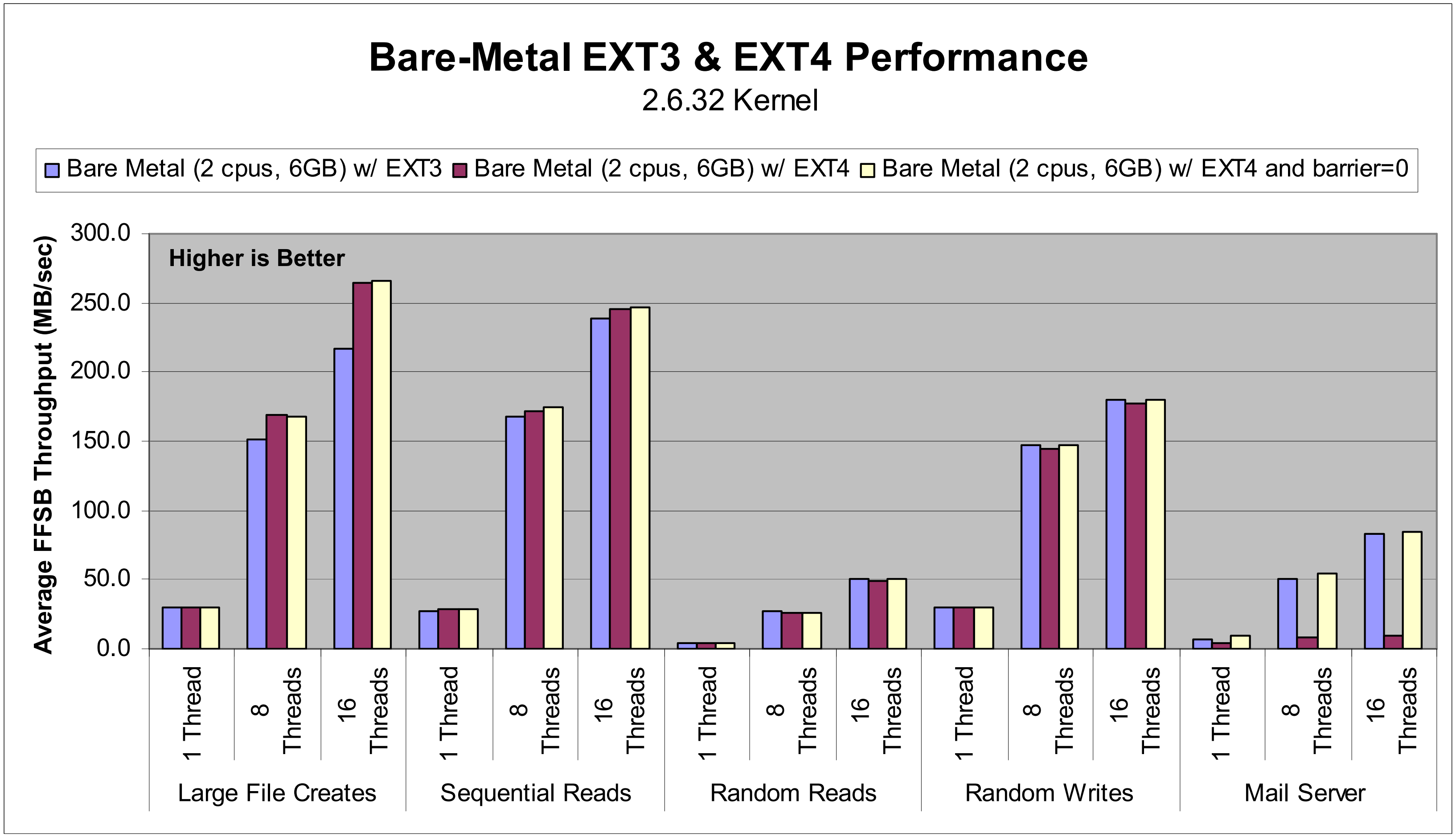
(Red Hat → *# tuned-adm profile enterprise-storage*)



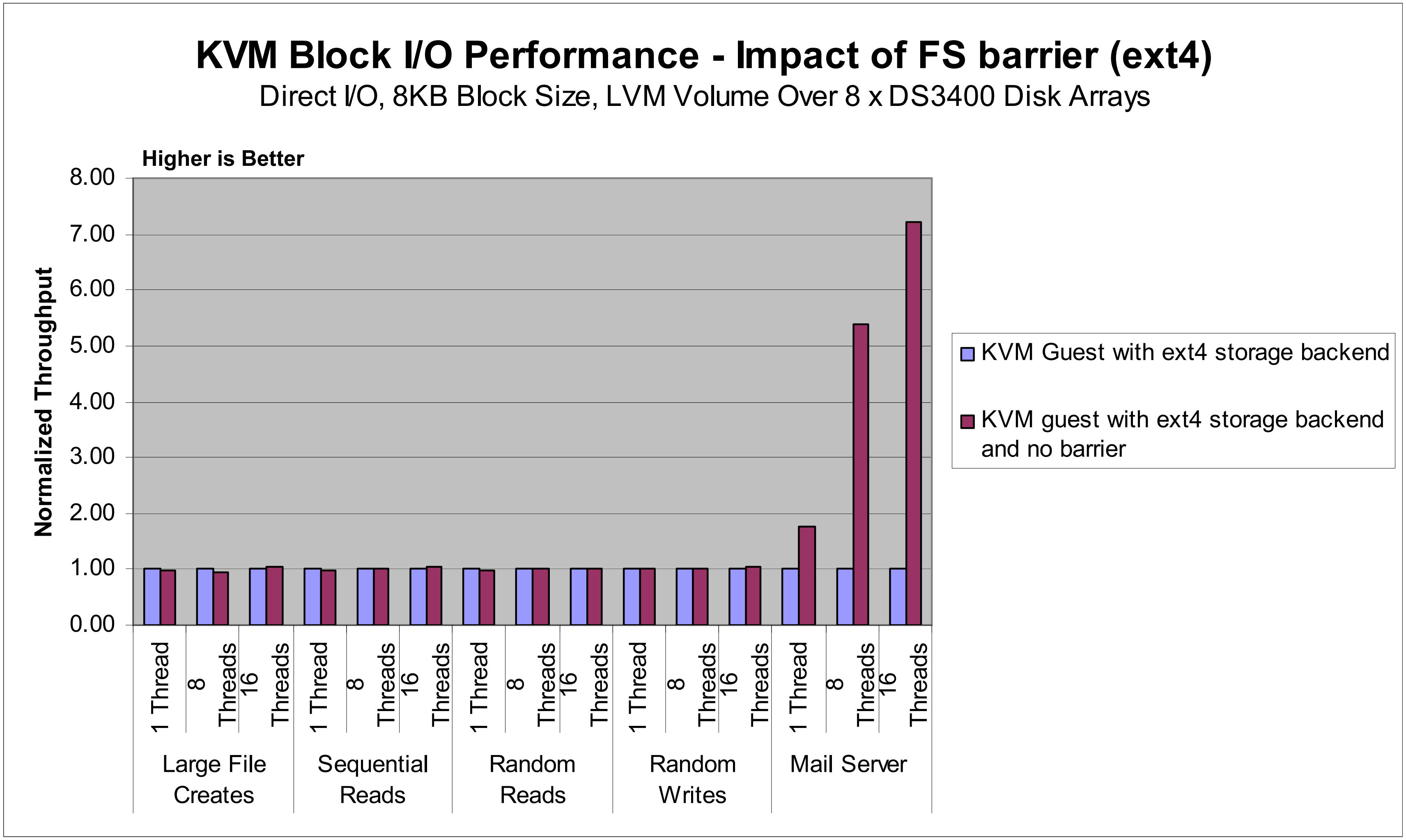
…But What About From Within KVM Guest ? Well, Deadline Scheduler Is Better There, Too….



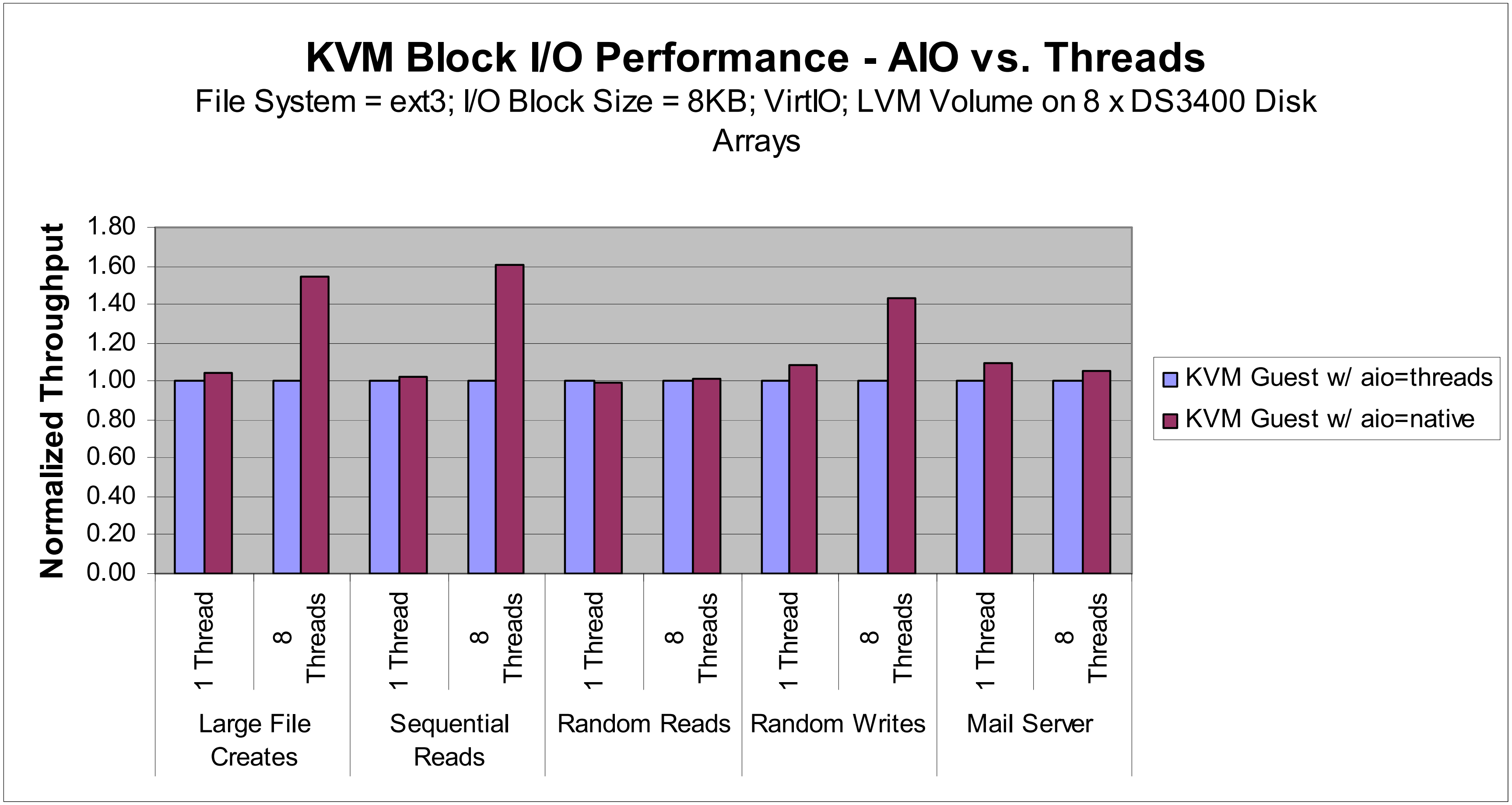
What About File Systems ? Barrier (ON By Default For ext4) Only Affects One Scenario Tested…



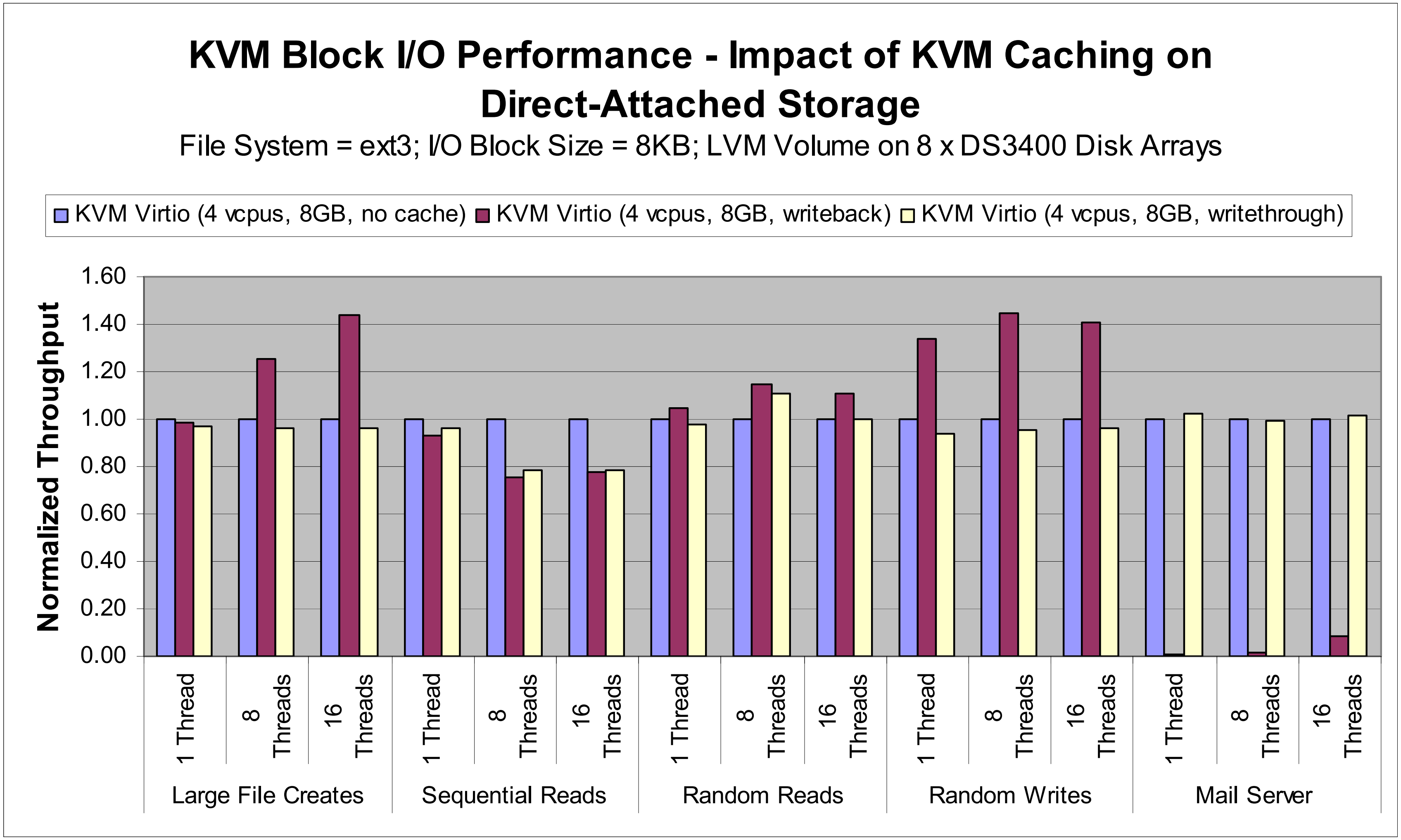
From KVM Guest, FS Barrier (ext4) Only Affects Mail Server Scenario…



Linux AIO Support for KVM / QEMU Is Good For Multiple Threads



For Many I/O Workloads (Databases), We Generally Recommend Bypassing Host Cache (cache=none)



One More Thing: x2APIC Support Has Been Found Beneficial For Many I/O Workloads…

- ***What is it ?***
  - This support implements x2APIC emulation for KVM. x2APIC is an MSR interface to a local APIC with performance and scalability enhancements. It brings 32-bit apic ids, 64-bit ICR access, and reading of ICR after IPI is no longer required.
- ***Author***: Gleb Natapov
- ***Performance Impact***: 2% to 5% throughput improvement for many I/O workloads

DISCUSSION – KVM / QEMU Block I/O Performance Issues

KVM /QEMU Block I/O Performance Issues

- **High (virtual) CPU usage in the KVM guest, preventing other work from being done in the guest**
  - *Verified that the high CPU usage in the guest is REAL (cyclesoak)*
  - *Profiling data (More on this later if time permits)*
    - Spin lock issue in QEMU – vblock->lock()
      - Proposed solution: release vblock->lock() before doing "kick" to the host; re-acquire lock upon return
  - *Path length analysis (Stefan Hajnoczi will go into all gory details during his talk on Friday 11/5)*
    - Some QEMU work (including I/O submission) is done inside the guest (vcpu threads)
      - Proposed solution: Move I/O submission work to iothread, freeing up vcpu threads for guest
  - *Any other suggestions* ?

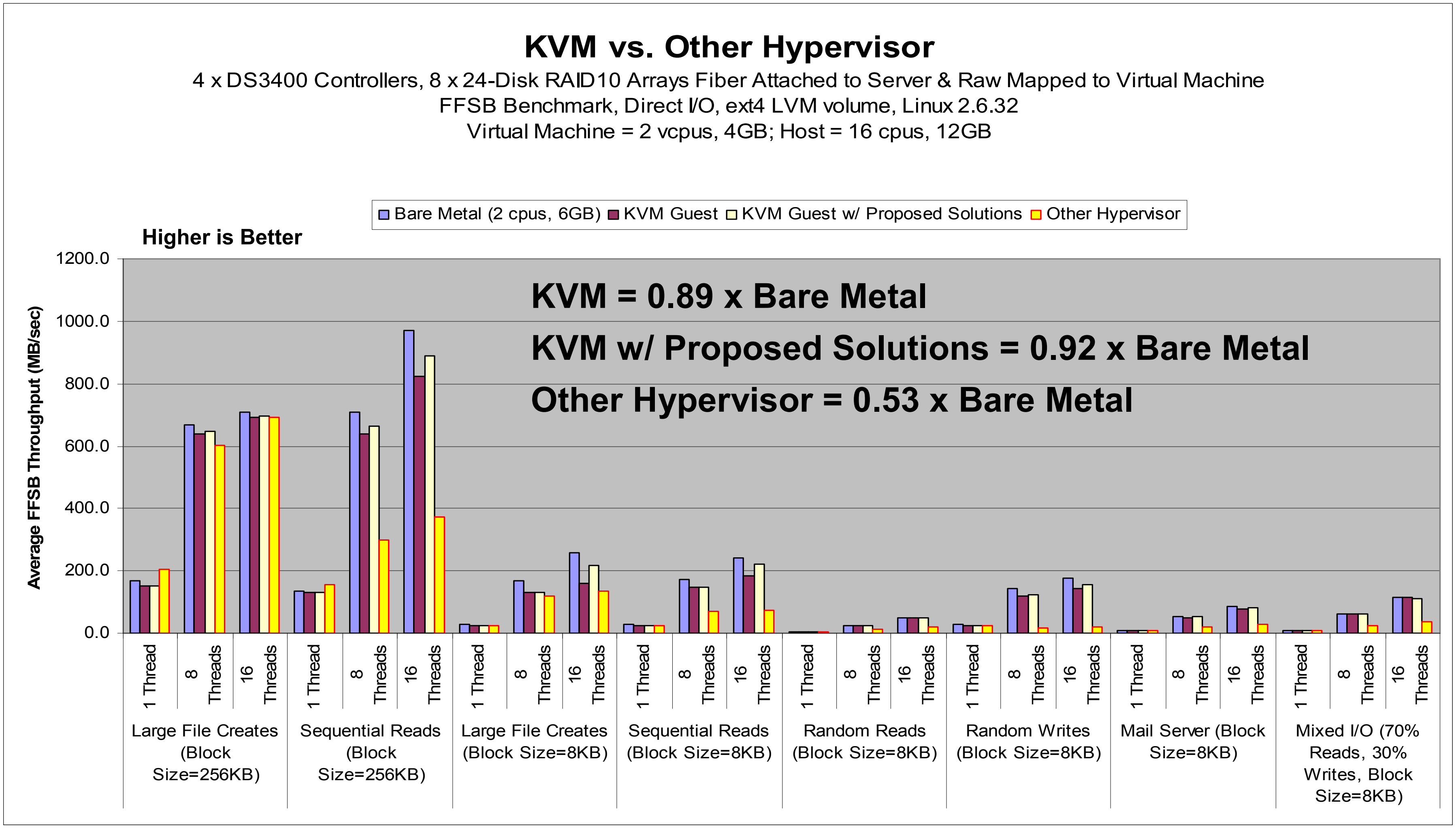
Discussion: KVM Block I/O Performance Issues (Cont'd)

- **(Relatively) Low Throughput Against Bare Metal**
  - *Path length analysis*
    - Stefan Hajnoczi will go into more details during his talk on Friday 11/5
      - Proposed solution: Use ioeventfd for asynchronous virtqueue processing (virtio-blk)
  - *Improving file-system efficiency between KVM guest and host*
    - VirtFS – JV Rao (presentation on Wednesday 11/3)
  - *Any other suggestions ?*

Discussion: KVM Block I/O Performance Issues (Cont'd)

- **Virtual Disk Image Format**
  - *RAW (flat file) has good performance & data integrity, but lacks advanced features (snapshots, delta images, …)*
  - *QCOW2 has all advanced features, provides strong data integrity at the expense of performance*
    - Proposed solution: QEMU Enhanced Disk (QED) – simpler design (eliminating rarely used features), better performance
    - On-going QCOW2 improvements ?
- **Others ?**

So What Do Proposed Solutions Buy Us ? Let's Look At The Throughput …



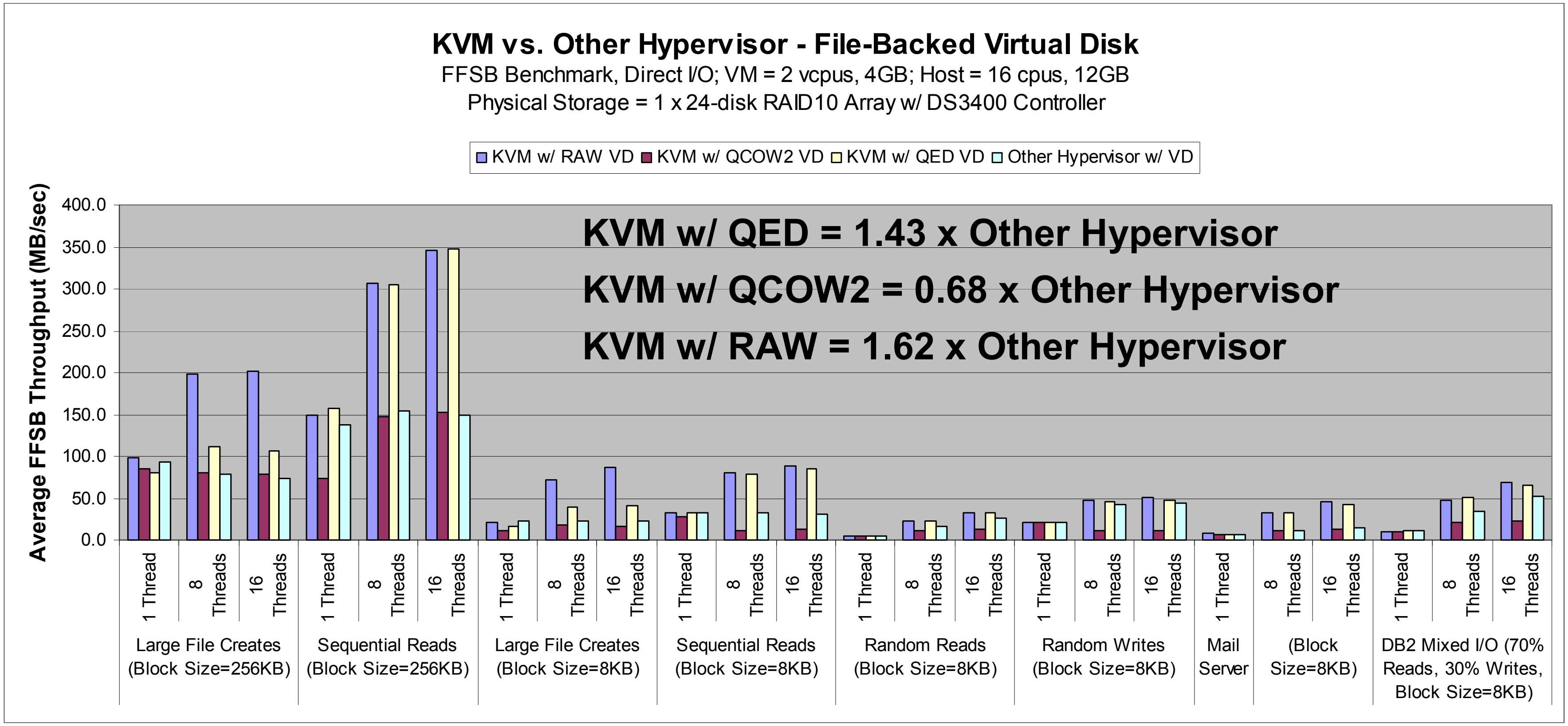
… And (Virtual) CPU Usage in Guest: 50% Reduction For Multi-Threaded I/O Scenarios

LOCAL FFSB Scenarios	Threads	KVM Guest (2 vcpus, 4GB) on Host (16 cpus, 12GB)						KVM guest (2 vcpus, 4GB) on Host (16 cpus, 12GB Host) w/ Proposed Solutions					
		IOPS	Throughput (MB/sec)	vcpu0%, vcpu1%	Total % (both vcpus)	%vcpu per MB/sec	Host CPU% (Average)	IOPS	Throughput (MB/sec)	vcpu0%, vcpu1%	Total % (both vcpus)	%vcpu per MB/sec	Host CPU% (Average)
Large File Creates (Block Size=256KB)	**1 Thread**	605.5	151.0	18%, 0%	18%	**0.12%**	2%	606.9	152.0	11%, 0%	11%	**0.07%**	2%
	8 Threads	2556.2	639.0	76%, 28%	104%	**0.16%**	8%	2589.8	647.0	46%, 8%	54%	**0.08%**	8%
	16 Threads	2765.0	691.0	75%, 43%	118%	**0.17%**	9%	2792.8	698.0	47%, 13%	60%	**0.09%**	9%
Sequential Reads (Block Size=256KB)	**1 Thread**	522.4	131.0	14%, 0%	14%	**0.11%**	2%	528.1	132.0	8%, 0%	8%	**0.06%**	2%
	8 Threads	2552.0	638.0	71%, 21%	92%	**0.14%**	7%	2650.3	663.0	44%, 7%	51%	**0.08%**	7%
	16 Threads	3294.2	824.0	86%, 64%	150%	**0.18%**	11%	3546.1	887.0	56%, 18%	74%	**0.08%**	10%
Large File Creates (Block Size=8KB)	**1 Thread**	3090.4	24.1	26%, 0%	26%	**1.08%**	3%	3049.8	23.8	11%, 0%	11%	**0.46%**	3%
	8 Threads	16671.3	130.0	90%, 4%	94%	**0.72%**	8%	16716.6	131.0	54%, 1%	55%	**0.42%**	9%
	16 Threads	20444.4	160.0	98%, 59%	157%	**0.98%**	11%	27642.3	216.0	74%, 3%	77%	**0.36%**	11%
Sequential Reads (Block Size=8KB)	**1 Thread**	3084.7	24.1	16%, 0%	16%	**0.66%**	3%	3067.6	24.0	7%, 0%	7%	**0.29%**	3%
	8 Threads	18648.9	146.0	80%, 2%	82%	**0.56%**	8%	18706.5	146.0	37%, 0%	37%	**0.25%**	9%
	16 Threads	23677.8	185.0	98%, 53%	151%	**0.82%**	11%	28585.4	223.0	53%, 1%	54%	**0.24%**	10%
Random Reads (Block Size=8KB)	**1 Thread**	466.3	3.6	3%, 0%	3%	**0.82%**	1%	433.7	3.4	1%, 0%	1%	**0.29%**	1%
	8 Threads	3252.4	25.4	16%, 0%	16%	**0.63%**	3%	3231.1	25.2	7%, 0%	7%	**0.28%**	3%
	16 Threads	6185.0	48.3	32%, 1%	33%	**0.68%**	4%	6184.7	48.3	14%, 0%	14%	**0.29%**	5%
Random Writes (Block Size=8KB)	**1 Thread**	3134.2	24.5	17%, 0%	17%	**0.69%**	3%	3074.0	24.0	8%, 0%	8%	**0.33%**	4%
	8 Threads	15222.7	118.9	82%, 3%	85%	**0.71%**	8%	15526.5	121.3	36%, 0%	36%	**0.30%**	8%
	16 Threads	18307.3	143.0	91%, 29%	120%	**0.84%**	9%	19747.7	154.3	45%, 2%	47%	**0.30%**	10%
Mixed I/O (70% Reads, 30% Writes) (Block Size=8KB)	**1 Thread**	550.6	7.2	3%, 0%	3%	**0.41%**	1%	584.3	8.7	2%, 0%	2%	**0.23%**	1%
	8 Threads	4172.6	60.9	22%, 0%	22%	**0.36%**	3%	4199.0	61.4	10%, 0%	10%	**0.16%**	4%
	16 Threads	7724.0	114.6	43%, 1%	44%	**0.38%**	5%	7647.3	111.2	18%, 0%	18%	**0.16%**	5%
Mail Server (Block Size=8KB)	**1 Thread**	1081.2	8.5	11%, 0%	11%	**1.30%**	2%	1093.3	8.5	5%, 0%	5%	**0.59%**	1%
	8 Threads	6507.9	50.8	60%, 2%	62%	**1.22%**	6%	6727.3	52.5	30%, 0%	30%	**0.57%**	6%
	16 Threads	9810.4	76.6	77%, 33%	110%	**1.44%**	9%	10378.2	81.0	45%, 2%	47%	**0.58%**	8%

Some Raw Data Comparing KVM w/ Proposed Solutions To Other Hypervisor …

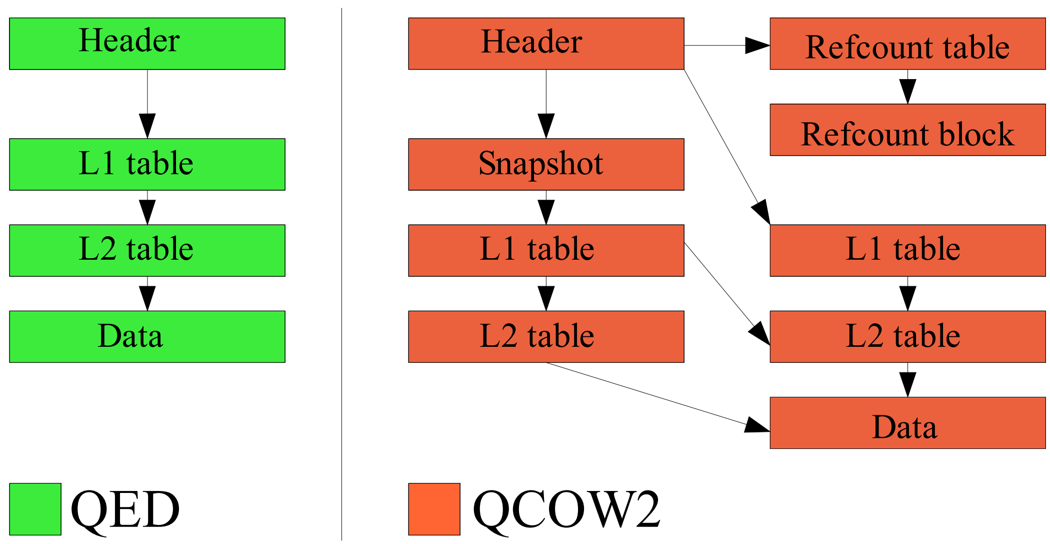
LOCAL FFSB Scenarios	Threads	KVM guest w/ Proposed Solutions				Bare Metal (2 cpus, 6GB, Linux 2.6.32)				Other Hypervisor			
		Throughput (MB/sec)	vcpu0%, vcpu1%	Total % (both vcpus)	%vcpu per MB/sec	Throughput (MB/sec)	vcpu0%, vcpu1%	Total % (both vcpus)	%vcpu per MB/sec	Throughput (MB/sec)	vcpu0%, vcpu1%	Total % (both vcpus)	%vcpu per MB/sec
Large File Creates (Block Size=256KB)	**1 Thread**	152.0	11%, 0%	11%	**0.07%**	168.0	7%, 1%	8%	**0.05%**	206.0	9%, 2%	11%	**0.05%**
	8 Threads	647.0	46%, 8%	54%	**0.08%**	669.0	28%, 10%	38%	**0.06%**	603.0	23%, 3%	26%	**0.04%**
	16 Threads	698.0	47%, 13%	60%	**0.09%**	707.0	31%, 11%	42%	**0.06%**	694.0	23%, 2%	25%	**0.04%**
Sequential Reads (Block Size=256KB)	**1 Thread**	132.0	8%, 0%	8%	**0.06%**	137.0	2%, 5%	7%	**0.05%**	157.0	8%, 2%	10%	**0.06%**
	8 Threads	663.0	44%, 7%	51%	**0.08%**	708.0	26%, 10%	36%	**0.05%**	299.0	12%, 2%	14%	**0.05%**
	16 Threads	887.0	56%, 18%	74%	**0.08%**	971.0	36%, 14%	50%	**0.05%**	372.0	13%, 2%	15%	**0.04%**
Large File Creates (Block Size=8KB)	**1 Thread**	23.8	11%, 0%	11%	**0.46%**	30.1	12%, 2%	14%	**0.47%**	25.0	21%, 2%	23%	**0.92%**
	8 Threads	131.0	54%, 1%	55%	**0.42%**	166.0	49%, 7%	56%	**0.34%**	117.0	54%, 8%	62%	**0.53%**
	16 Threads	216.0	74%, 3%	77%	**0.36%**	256.0	72%, 13%	85%	**0.33%**	134.0	55%, 13%	68%	**0.51%**
Sequential Reads (Block Size=8KB)	**1 Thread**	24.0	7%, 0%	7%	**0.29%**	28.0	3%, 3%	6%	**0.21%**	24.3	17%, 2%	19%	**0.78%**
	8 Threads	146.0	37%, 0%	37%	**0.25%**	170.0	30%, 3%	33%	**0.19%**	70.9	34%, 7%	41%	**0.58%**
	16 Threads	223.0	53%, 1%	54%	**0.24%**	241.0	43%, 4%	47%	**0.20%**	74.5	35%, 7%	42%	**0.56%**
Random Reads (Block Size=8KB)	**1 Thread**	3.4	1%, 0%	1%	**0.29%**	3.6	0%, 1%	1%	**0.28%**	3.5	3%, 0%	3%	**0.85%**
	8 Threads	25.2	7%, 0%	7%	**0.28%**	25.5	5%, 1%	6%	**0.24%**	13.4	9%, 1%	10%	**0.75%**
	16 Threads	48.3	14%, 0%	14%	**0.29%**	47.7	9%, 2%	11%	**0.23%**	20.5	12%, 2%	14%	**0.68%**
Random Writes (Block Size=8KB)	**1 Thread**	24.0	8%, 0%	8%	**0.33%**	30.2	6%, 2%	8%	**0.27%**	24.6	20%, 3%	23%	**0.93%**
	8 Threads	121.3	36%, 0%	36%	**0.30%**	145.1	32%, 5%	37%	**0.25%**	15.6	8%, 1%	9%	**0.58%**
	16 Threads	154.3	45%, 2%	47%	**0.30%**	177.0	38%, 8%	46%	**0.26%**	21.7	10%, 2%	12%	**0.55%**
Mixed I/O (70% Reads, 30% Writes) (Block Size=8KB)	**1 Thread**	8.7	2%, 0%	2%	**0.23%**	8.5	4%, 1%	5%	**0.59%**	9.3	4%, 1%	5%	**0.54%**
	8 Threads	61.4	10%, 0%	10%	**0.16%**	54.3	27%, 6%	33%	**0.61%**	24.8	9%, 1%	10%	**0.40%**
	16 Threads	111.2	18%, 0%	18%	**0.16%**	84.2	39%, 11%	50%	**0.59%**	36.1	12%, 1%	13%	**0.36%**
Mail Server (Block Size=8KB)	**1 Thread**	8.5	5%, 0%	5%	**0.59%**	8.0	1%, 1%	2%	**0.25%**	8.0	9%, 1%	10%	**1.25%**
	8 Threads	52.5	30%, 0%	30%	**0.57%**	61.8	7%, 1%	8%	**0.13%**	21.4	17%, 2%	19%	**0.89%**
	16 Threads	81.0	45%, 2%	47%	**0.58%**	115.5	13%, 2%	15%	**0.13%**	29.3	20%, 4%	24%	**0.82%**

What About Virtual Disk Image Format ? Introducing new QED format (see next slide)…



QEMU Enhanced Disk format

- New format with an **open specification**.
- Designed for **strong data integrity** while achieving **good performance**.
- Significantly **simpler design**:



Summary – Good Things

- KVM block I/O performance to device-backed virtual disks
- QED for file-backed virtual disks
- VirtIO drivers
- cache = none
- Linux AIO support
- Deadline I/O scheduler
- Outstanding patches
  - Move some I/O submission work from vcpu threads to iothread
  - Reduce time in spin lock (guest kernel)
- Others ?

Credits & Thanks

- Anthony Liguori
- Andrew Theurer
- Badari Pulavarty
- Ryan Harper
- Frank Novak

धन्यवाद

Hindi

多謝

Traditional Chinese

ขอบคุณ

Thai

Спасибо

Russian

Gracias

Spanish

Thank You

English

Obrigado

Brazilian Portuguese

شكراً

Arabic

多谢

Simplified Chinese

Danke

German

שָׁלוֹם

Hebrew

Grazie

Italian

Merci

French

நன்றி

Tamil

ありがとうございました

Japanese

감사합니다

Korean

BACK-UP

First Thing … Get Profiling Data In KVM Guest … PERF Tool

- **KVM guest = 2 vcpus, 2.6.18 kernel, virtio-blk w/ cache=none; Host = 8 cores (16 cpu threads), 2.6.35-rc2+ kernel**
- **Scenario = Large File Creates (w/ 16 threads)**

```
# Events: 456K cycles #
# Overhead Command Shared Object Symbol #
........ ............... ....................... ...... #
51.94% qemu-kvm [guest.kernel.kallsyms] [g] .text.lock.spinlock
2.59% qemu-kvm 3b698bb8d2 [u] 0x00003b698bb8d2
1.13% qemu-kvm [guest.kernel.kallsyms] [g] __blockdev_direct_IO
1.10% qemu-kvm [guest.kernel.kallsyms] [g] __find_get_block
1.03% qemu-kvm [guest.kernel.kallsyms] [g] kmem_cache_free
1.03% qemu-kvm [guest.kernel.kallsyms] [g] kmem_cache_alloc
0.83% qemu-kvm [ext3] [g] __ext3_get_inode_loc
0.82% qemu-kvm [jbd] [g] do_get_write_access
0.74% qemu-kvm [jbd] [g] journal_add_journal_head
0.73% qemu-kvm [guest.kernel.kallsyms] [g] __make_request
0.58% qemu-kvm [ext3] [g] ext3_mark_iloc_dirty
0.58% qemu-kvm [guest.kernel.kallsyms] [g] _spin_lock
0.57% qemu-kvm [guest.kernel.kallsyms] [g] ioread8
0.56% qemu-kvm [guest.kernel.kallsyms] [g] schedule
0.56% qemu-kvm [guest.kernel.kallsyms] [g] radix_tree_lookup
0.54% qemu-kvm [guest.kernel.kallsyms] [g] kfree
0.54% qemu-kvm [guest.kernel.kallsyms] [g] bit_waitqueue
```

Digging Deeper Into Spin Locks … With Lockstat

- Installed debug kernel in the guest to run lockstat tool
- **KVM guest = 4 vcpus, 2.6.18 kernel (debug kernel), virtio-blk w/ cache=none; Host = 8 cores (16 cpu threads), 2.6.35-rc2+ kernel**
- **At the top of LOCKSTAT output:**

```
&vblk->lock          1265          [<ffffffff8000c435>] __make_request+0x73/0x402
&vblk->lock       3627248          [<ffffffff8000c720>] __make_request+0x35e/0x402
&vblk->lock        531284          [<ffffffff8005d9c1>] generic_unplug_device+0x1a/0x31
&vblk->lock          8778          [<ffffffff88097310>] blk_done+0x1d/0xbd [virtio_blk]
```

Observations:

- The most “popular” lock is **&vblk->lock**. The number of contentions is high and the wait time is high - an average wait time of 190.10 per contention. Those are most likely so high because the lock is held for such a long time - an average of 80.21 per acquisition.

After Releasing vblock->lock() Before "Kicking" To Host, Time Spent In Spin Locks Are Reduced …

- **KVM guest = 2 vcpus, 2.6.18 kernel, virtio-blk w/ cache=none; Host = 16 cpu threads, 2.6.35-rc2+ kernel**
- **Scenario = Large File Creates (w/ 16 Threads)**

```
# Events: 293K cycles #
# Overhead Command Shared Object Symbol
# ........ ............... ...................... ...... #
5.65% qemu-kvm 3b6787aaa9 [u] 0x00003b6787aaa9
3.73% qemu-kvm [guest.kernel.kallsyms] [g] .text.lock.spinlock
2.19% qemu-kvm [guest.kernel.kallsyms] [g] __blockdev_direct_IO
2.14% qemu-kvm [guest.kernel.kallsyms] [g] kmem_cache_free
2.13% qemu-kvm [guest.kernel.kallsyms] [g] __find_get_block
2.00% qemu-kvm [guest.kernel.kallsyms] [g] kmem_cache_alloc
1.63% qemu-kvm [ext3] [g] __ext3_get_inode_loc
1.62% qemu-kvm [jbd] [g] do_get_write_access
1.57% qemu-kvm [jbd] [g] journal_add_journal_head
1.46% qemu-kvm [guest.kernel.kallsyms] [g] _spin_lock
1.17% qemu-kvm [guest.kernel.kallsyms] [g] schedule
1.17% qemu-kvm [ext3] [g] ext3_mark_iloc_dirty
1.09% qemu-kvm [guest.kernel.kallsyms] [g] iowrite16
1.08% qemu-kvm [virtio_ring] [g] vring_kick
1.06% qemu-kvm [guest.kernel.kallsyms] [g] radix_tree_lookup
1.06% qemu-kvm [guest.kernel.kallsyms] [g] bit_waitqueue
1.06% qemu-kvm [guest.kernel.kallsyms] [g] kfree
```

… BUT Oprofile Data Shows We Still Spend Much Time In *make_request()* → Path Length Analysis

- **KVM guest = 2 vcpus, 2.6.18 kernel, virtio-blk w/ cache=none; Host = 16 cpu threads, 2.6.35-rc2+ kernel**
- **Scenario = Large File Creates (w/ 8 Threads)**

```
Overflow stats not available
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
samples    %     app name symbol name
567149  62.2261 vmlinux __make_request ← This still consumes much time
99605   10.9284 uhci-hcd.ko uhci_irq
61150    6.7092 vmlinux ioread8
22047    2.4189 vmlinux default_idle  ← Virtual CPUs are pretty busy
15704    1.7230 virtio_pci.ko vp_interrupt
6341     0.6957 vmlinux thread_return
5574     0.6116 vmlinux __blockdev_direct_IO
5426     0.5953 vmlinux get_request
4250     0.4663 vmlinux kmem_cache_alloc
4228     0.4639 vmlinux __find_get_block
4110     0.4509 vmlinux handle_IRQ_event
3501     0.3841 ext3.ko __ext3_get_inode_loc
3207     0.3519 vmlinux find_get_page
3150     0.3456 jbd.ko journal_add_journal_head
3105     0.3407 jbd.ko do_get_write_access
2924     0.3208 vmlinux kmem_cache_free
2566     0.2815 ext3.ko ext3_mark_iloc_dirty
2215     0.2430 vmlinux bit_waitqueue
2141     0.2349 libpthread-2.5.so __write_nocancel
```

What If We Could Bypass Current QEMU Entirely (e.g. Implementing separate virtio-blk threads using Linux AIO and direct virtio-ring access) ?