Predicting (Disk Scheduling) Performance with Virtual Machines

Robert Geist, Zach Jones, James Westall

Clemson University

Motivation

- CPSC 822: Operating System Design: Case Study
  - second level, graduate OS course at Clemson
  - since 1985: walk through source of a UNIX derivative (this semester: Linux 2.6.30)
  - modify schedulers for performance
  - build new kernels
  - write drivers for real devices

Motivation

- CPSC 822: Operating System Design: Case Study
  - second level, graduate OS course at Clemson
  - since 1985: walk through source of a UNIX derivative (this semester: Linux 2.6.30)
  - modify schedulers for performance
  - build new kernels
  - write drivers for real devices
- dedicated hardware required (usually crashed) → limited enrollment → waiting list, every semester

Motivation

- CPSC 822: Operating System Design: Case Study
  - second level, graduate OS course at Clemson
  - since 1985: walk through source of a UNIX derivative (this semester: Linux 2.6.30)
  - modify schedulers for performance
  - build new kernels
  - write drivers for real devices
- dedicated hardware required (usually crashed) → limited enrollment → waiting list, every semester
- standard evaluation (5 yrs. out): *the* most valuable course of educational career (e.g. Satish Dharmaraj)

Virtualization?

- a large part of the course could be virtualized (VMWare, XEN, KVM)

Virtualization?

- a large part of the course could be virtualized (VMWare, XEN, KVM)
- oops! two important projects resist this:

Virtualization?

- a large part of the course could be virtualized (VMWare, XEN, KVM)
- oops! two important projects resist this:
  - write a driver for non-trivial graphics card (interrupts, DMA, buffer handling, mem. map.)

Virtualization?

- a large part of the course could be virtualized (VMWare, XEN, KVM)
- oops! two important projects resist this:
  - write a driver for non-trivial graphics card (interrupts, DMA, buffer handling, mem. map.) See *Proc. IBM CASCON 2009, Toronto, CA.*

Virtualization?

- a large part of the course could be virtualized (VMWare, XEN, KVM)
- oops! two important projects resist this:
  - write a driver for non-trivial graphics card (interrupts, DMA, buffer handling, mem. map.) See *Proc. IBM CASCON 2009, Toronto, CA.*
  - design a new disk scheduler that outperforms default Linux schedulers

Virtualization?

- a large part of the course could be virtualized (VMWare, XEN, KVM)
- oops! two important projects resist this:
  - write a driver for non-trivial graphics card (interrupts, DMA, buffer handling, mem. map.) See *Proc. IBM CASCON 2009, Toronto, CA.*
  - design a new disk scheduler that outperforms default Linux schedulers

Yeow! How do I measure that?

Goals

- provide a method for predicting the performance of disk scheduling algorithms on real machines using only their performance on virtual machines

Goals

- provide a method for predicting the performance of disk scheduling algorithms on real machines using only their performance on virtual machines
- provide a new, high-performance, disk scheduling algorithm as a case study

Goals

- provide a method for predicting the performance of disk scheduling algorithms on real machines using only their performance on virtual machines
- provide a new, high-performance, disk scheduling algorithm as a case study
- describe the *iprobe*, a key kernel modification tool, which should have wide application

Background - Kernel Probes

- intended as dynamically-loaded debugging tools
- Linux *kprobe*

Background - Kernel Probes

- intended as dynamically-loaded debugging tools
- Linux *kprobe*
    - target instruction, pre-handler, post-handler
    - save target, replace with breakpoint
    - upon break:

Background - Kernel Probes

- intended as dynamically-loaded debugging tools
- Linux *kprobe*
  - target instruction, pre-handler, post-handler
  - save target, replace with breakpoint
  - upon break:
    - pre-handler;
    - target (single step mode);
    - post-handler;
    - resume;

Background - Kernel Probes

- Linux *jprobe*
    - target function, (second-stage) pre-handler
    - copy first instruction, replace with breakpoint
    - upon break:

Background - Kernel Probes

- Linux *jprobe*
  - target function, (second-stage) pre-handler
  - copy first instruction, replace with breakpoint
  - upon break:
    - run fixed, first-stage pre-handler:

Background - Kernel Probes

- Linux *jprobe*
  - target function, (second-stage) pre-handler
  - copy first instruction, replace with breakpoint
  - upon break:
    - run fixed, first-stage pre-handler:
      - copy registers and stack;
      - load saved IP with address of ss handler;
      - return (passes control to ss handler);

Background - Kernel Probes

- Linux *jprobe*
  - target function, (second-stage) pre-handler
  - copy first instruction, replace with breakpoint
  - upon break:
    - run fixed, first-stage pre-handler:
      - copy registers and stack;
      - load saved IP with address of ss handler;
      - return (passes control to ss handler);
    - execute second-stage handler;

Background - Kernel Probes

- Linux *jprobe*
  - target function, (second-stage) pre-handler
  - copy first instruction, replace with breakpoint
  - upon break:
    - run fixed, first-stage pre-handler:
      - copy registers and stack;
      - load saved IP with address of ss handler;
      - return (passes control to ss handler);
    - execute second-stage handler;
    - jprobe return (restore stack and state);

Background - Kernel Probes

- Linux *jprobe*
  - target function, (second-stage) pre-handler
  - copy first instruction, replace with breakpoint
  - upon break:
    - run fixed, first-stage pre-handler:
      - copy registers and stack;
      - load saved IP with address of ss handler;
      - return (passes control to ss handler);
    - execute second-stage handler;
    - jprobe return (restore stack and state);
    - first instruction in single-step mode;
    - remainder of function (empty post-handler);

iprobe

- dynamically replace any kernel function!

iprobe

- dynamically replace any kernel function!
- target function, replacement function
- function prototypes must match
- built on jprobe framework:

iprobe

- dynamically replace any kernel function!
- target function, replacement function
- function prototypes must match
- built on jprobe framework:
    - custom second-stage pre-handler
    - custom (non-empty) post-handler

iprobe

- copy first instruction, replace with breakpoint
- upon break:

iprobe

- copy first instruction, replace with breakpoint
- upon break:
  - run jprobe first-stage pre-handler;
  - execute custom second-stage handler:

iprobe

- copy first instruction, replace with breakpoint
- upon break:
  - run jprobe first-stage pre-handler;
  - execute custom second-stage handler:
    - backup saved instruction;
    - overwrite saved instruction with no-op;
  - jprobe return;

iprobe

- copy first instruction, replace with breakpoint
- upon break:
  - run jprobe first-stage pre-handler;
  - execute custom second-stage handler:
    - backup saved instruction;
    - overwrite saved instruction with no-op;
  - jprobe return;
  - first instruction (no-op) in single-step mode;
  - post-handler:

iprobe

- copy first instruction, replace with breakpoint
- upon break:
  - run jprobe first-stage pre-handler;
  - execute custom second-stage handler:
    - backup saved instruction;
    - overwrite saved instruction with no-op;
  - jprobe return;
  - first instruction (no-op) in single-step mode;
  - post-handler:
    - load IP with replacement function address;
    - overwrite no-op with backup copy;
    - return;

Background - Disk Scheduling

- heavily-loaded system: non-empty queue of pending disk requests likely; schedule in which order?

Background - Disk Scheduling

- heavily-loaded system: non-empty queue of pending disk requests likely; schedule in which order?
- such algorithms studied for decades (at least 4!)

Background - Disk Scheduling

- heavily-loaded system: non-empty queue of pending disk requests likely; schedule in which order?
- such algorithms studied for decades (at least 4!)
- increasing importance:
  - 20 years ago: CPU speed in μs, disk speed in ms

Background - Disk Scheduling

- heavily-loaded system: non-empty queue of pending disk requests likely; schedule in which order?
- such algorithms studied for decades (at least 4!)
- increasing importance:
  - 20 years ago: CPU speed in μs, disk speed in ms
  - today: CPU speed in ns, disk speed still in ms

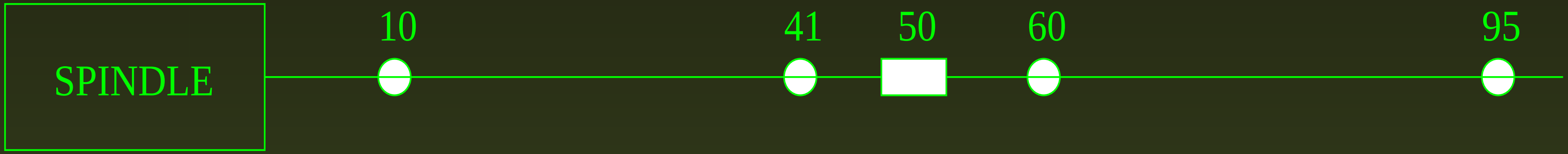
Background - Disk Scheduling

- heavily-loaded system: non-empty queue of pending disk requests likely; schedule in which order?
- such algorithms studied for decades (at least 4!)
- increasing importance:
  - 20 years ago: CPU speed in μs, disk speed in ms
  - today: CPU speed in ns, disk speed still in ms
  - disks are performance bottlenecks

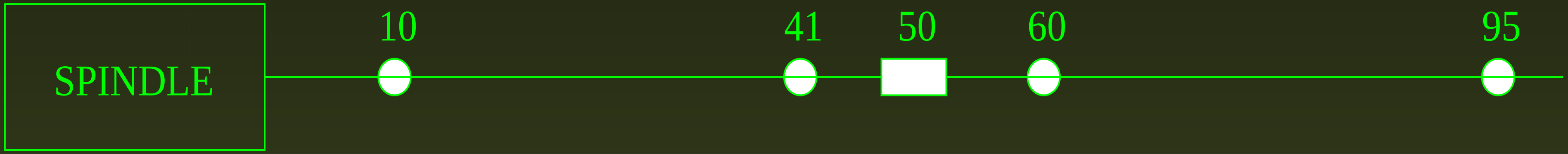
Background - Disk Scheduling

- heavily-loaded system: non-empty queue of pending disk requests likely; schedule in which order?
- such algorithms studied for decades (at least 4!)
- increasing importance:
  - 20 years ago: CPU speed in μs, disk speed in ms
  - today: CPU speed in ns, disk speed still in ms
  - disks are performance bottlenecks
- algorithms not constrained to be *work-conserving*

An Example

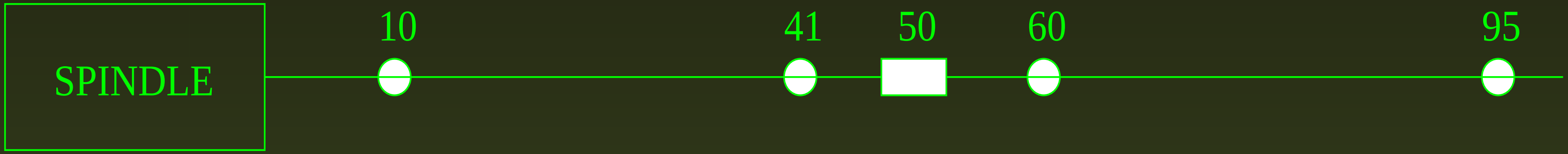


An Example



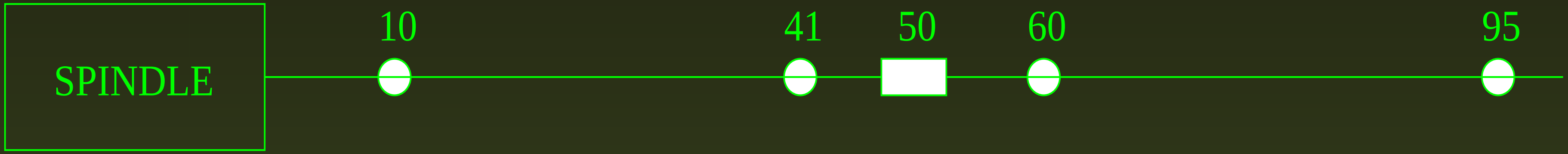
- order of arrival is 95, 10, 60, 41 (r/w head at 50)

An Example



- order of arrival is 95, 10, 60, 41 (r/w head at 50)
- travel time constant per unit distance

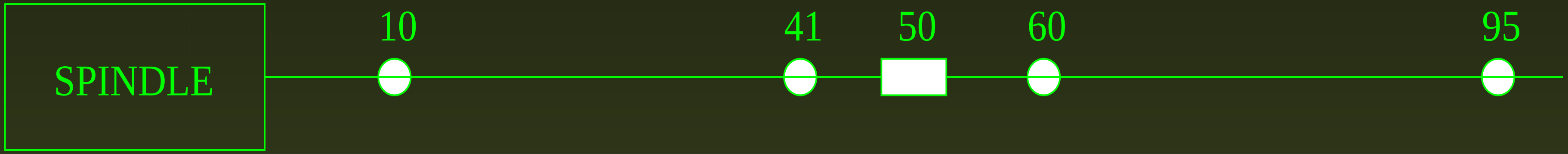
An Example



- order of arrival is 95, 10, 60, 41 (r/w head at 50)
- travel time constant per unit distance

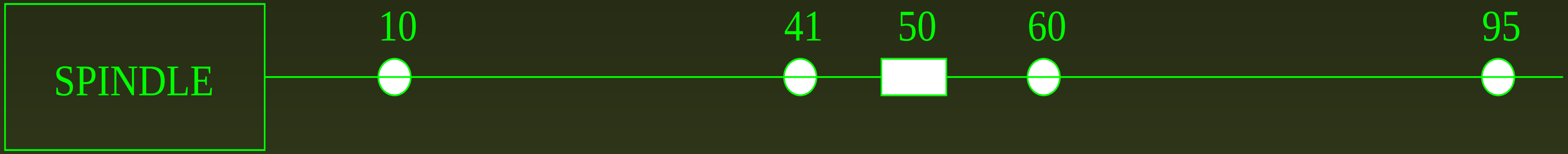
schedule	service	wait	response
95	45	0	45
10	85	45	130
60	50	130	180
41	19	180	199
mean	49.75	88.75	138.50

An Example (Continued)



Greedy or *shortest access time first (SATF)* schedule:

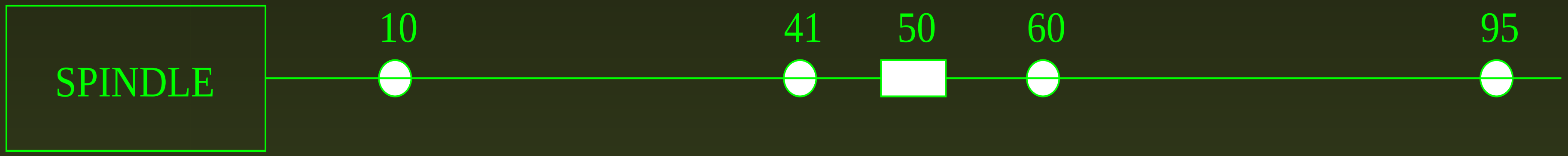
An Example (Continued)



Greedy or *shortest access time first (SATF)* schedule:

schedule	service	wait	response
41	9	0	9
60	19	9	28
95	35	28	63
10	85	63	148
mean	37.00	25.00	62.00

An Example (Continued)

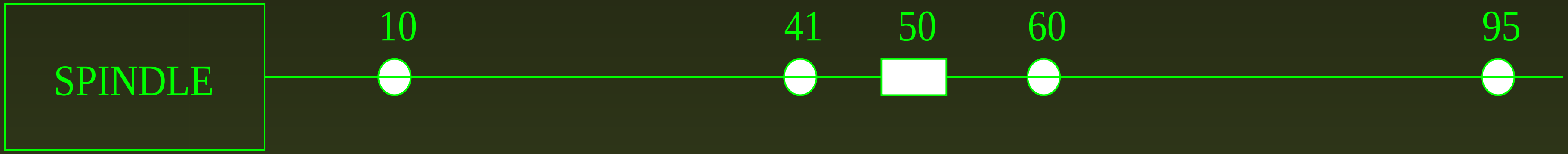


SPINDLE

10 41 50 60 95

SATF often claimed to be optimal, but ...

An Example (Continued)



SATF often claimed to be optimal, but ...

schedule	service	wait	response
60	10	0	10
41	19	10	29
10	31	29	60
95	85	60	145
mean	36.25	24.75	61.00

Schedulers Supplied with Linux 2.6

- No-op
- Anticipatory
- Deadline
- Completely Fair Queueing

Cache-Aware Table Scheduler (CATS)

- separate reads and writes; reads have priority

Cache-Aware Table Scheduler (CATS)

- separate reads and writes; reads have priority
- writes use CSCAN with *request coalescing*

Cache-Aware Table Scheduler (CATS)

- separate reads and writes; reads have priority
- writes use CSCAN with *request coalescing*
- writes served in bursts (MAX/MIN WRITEDELAY)

Cache-Aware Table Scheduler (CATS)

- separate reads and writes; reads have priority
- writes use CSCAN with *request coalescing*
- writes served in bursts (MAX/MIN WRITEDELAY)
- reads use algorithm T with *request coalescing*:

Cache-Aware Table Scheduler (CATS)

- separate reads and writes; reads have priority
- writes use CSCAN with *request coalescing*
- writes served in bursts (MAX/MIN WRITEDELAY)
- reads use algorithm T with *request coalescing*:
  - for any collection of n requests, find optimal (minimum response time) completion sequence in worst-case $O(n^2)$ time

Cache-Aware Table Scheduler (CATS)

- separate reads and writes; reads have priority
- writes use CSCAN with *request coalescing*
- writes served in bursts (MAX/MIN WRITEDELAY)
- reads use algorithm T with *request coalescing*:
  - for any collection of n requests, find optimal (minimum response time) completion sequence in worst-case $O(n^2)$ time
  - serve first request from optimal list

Cache-Aware Table Scheduler (CATS)

- separate reads and writes; reads have priority
- writes use CSCAN with *request coalescing*
- writes served in bursts (MAX/MIN WRITEDELAY)
- reads use algorithm T with *request coalescing*:
  - for any collection of n requests, find optimal (minimum response time) completion sequence in worst-case $O(n^2)$ time
  - serve first request from optimal list
  - re-compute optimal list, if new arrivals

Cache-Aware Table Scheduler (CATS)

- separate reads and writes; reads have priority
- writes use CSCAN with *request coalescing*
- writes served in bursts (MAX/MIN WRITEDELAY)
- reads use algorithm T with *request coalescing*:
  - for any collection of n requests, find optimal (minimum response time) completion sequence in worst-case $O(n^2)$ time
  - serve first request from optimal list
  - re-compute optimal list, if new arrivals
- out-wait deceptive idleness (5 ms)

Cache-Aware Table Scheduler (CATS)

- cache model:

Cache-Aware Table Scheduler (CATS)

- cache model: number of segments, sectors per segment, pre-fetch size (sectors)

Cache-Aware Table Scheduler (CATS)

- cache model: number of segments, sectors per segment, pre-fetch size (sectors)
- cache model assumptions:

Cache-Aware Table Scheduler (CATS)

- cache model: number of segments, sectors per segment, pre-fetch size (sectors)
- cache model assumptions: fully associative, FIFO replacement, wrap-around within segments

Cache-Aware Table Scheduler (CATS)

- cache model: number of segments, sectors per segment, pre-fetch size (sectors)
- cache model assumptions: fully associative, FIFO replacement, wrap-around within segments
- scheduling:

Cache-Aware Table Scheduler (CATS)

- cache model: number of segments, sectors per segment, pre-fetch size (sectors)
- cache model assumptions: fully associative, FIFO replacement, wrap-around within segments
- scheduling:
  - maintain shadow cache within scheduler
  - on each dispatch, check entire queue for predicted cache hit
  - if predicted hit, schedule immediately

Virtual Performance Throttle

Predict real performance from virtual performance?

Virtual Performance Throttle

Predict real performance from virtual performance?
Use *iprobe* in virtual SCSI path to force virtual service times to be proportional to real ones.

Virtual Performance Throttle

Predict real performance from virtual performance?
Use *iprobe* in virtual SCSI path to force virtual service times to be proportional to real ones.

- real service time model: $X_r = R_r/2 + S_r(d_r/D_r)$
  R_r is rotation time, S_r is maximum seek time, D_r is maximum seek distance

Virtual Performance Throttle

Predict real performance from virtual performance?
Use *iprobe* in virtual SCSI path to force virtual service times to be proportional to real ones.

- real service time model: $X_r = R_r/2 + S_r(d_r/D_r)$
  R_r is rotation time, S_r is maximum seek time, D_r is maximum seek distance
- force virtual service time kX_r, where k is constant

Virtual Performance Throttle

Predict real performance from virtual performance? Use *iprobe* in virtual SCSI path to force virtual service times to be proportional to real ones.

- real service time model: $X_r = R_r/2 + S_r(d_r/D_r)$ R_r is rotation time, S_r is maximum seek time, D_r is maximum seek distance
- force virtual service time kX_r, where k is constant
- observed virutal service time is X_v
- *iprobe*: delay virtual request completion $kX_r - X_v$

Virtual Performance Throttle

Predict real performance from virtual performance? Use *iprobe* in virtual SCSI path to force virtual service times to be proportional to real ones.

- real service time model: $X_r = R_r/2 + S_r(d_r/D_r)$ R_r is rotation time, S_r is maximum seek time, D_r is maximum seek distance
- force virtual service time kX_r, where k is constant
- observed virutal service time is X_v
- *iprobe*: delay virtual request completion $kX_r - X_v$
- oops! k is unknown

Virtual Performance Throttle

Need self-scaling k! Rules:

1. virtual request completes after target time?
   k too small $\longrightarrow$ *iprobe* increases k

Virtual Performance Throttle

Need self-scaling k! Rules:

1. virtual request completes after target time?
   k too small $\longrightarrow$ *iprobe* increases k
2. *iprobe* queue of completions too large?
   k too large $\longrightarrow$ *iprobe* decreases k

Virtual Performance Throttle

Need self-scaling k! Rules:

1. virtual request completes after target time?
   k too small → *iprobe* increases k
2. *iprobe* queue of completions too large?
   k too large → *iprobe* decreases k

System reports current k
Rule 1: accuracy
Rule 2: simulation run-time

Test Platforms

Real:

- Linux 2.6.30
- dual Intel Xeon 2.80GHz CPUs
- Western Digital IDE system drive
- dual 73.4 GB Seagate Cheetah 15.4K SCSI drives
- dual Adaptec 39320A Ultra320 SCSI controllers
- tests restricted to single SCSI drive

Test Platforms

Virtual:

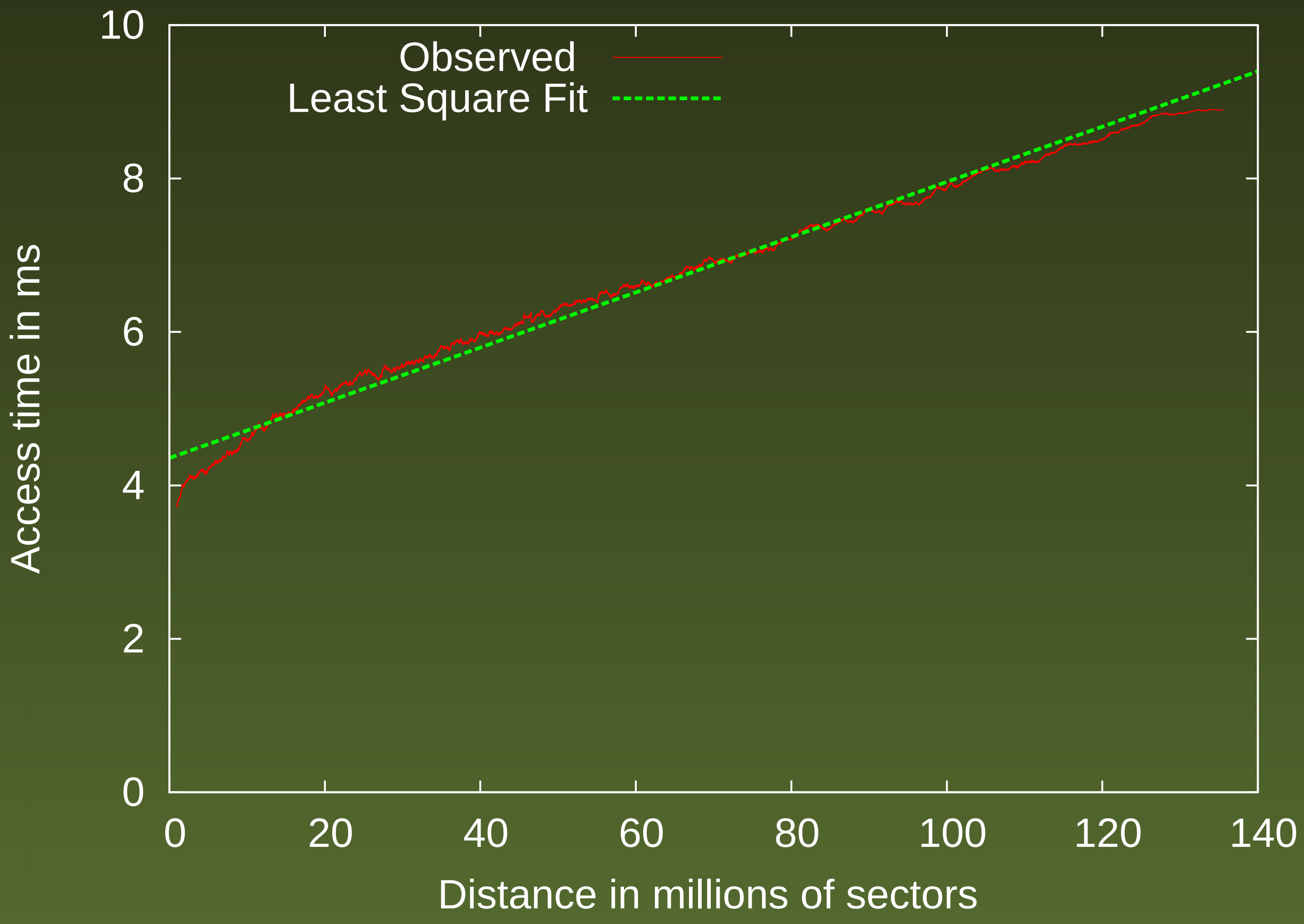
- KVM-based, virtual Linux 2.6.30
- hosted on IBM 8853AC1 dual 2.83GHz Xeon blade
- virtual 73.4 GB SCSI disk
- virtual disk on NetApp FAS960c, access NFS

Test Platforms

Service Time Model: disable cache, open O_DIRECT, measure 100,000 random page reads, linear fit

Test Platforms

Service Time Model: disable cache, open O_DIRECT, measure 100,000 random page reads, linear fit



Test Platforms

Cache Model (Seagate manual and *sdparm*):

- 64 segments
- 221 sectors per segment
- 64-sector pre-fetch

Workload

- Barford and Crovella: (SURGE) tool

Workload

- Barford and Crovella: (SURGE) tool
- 64 processes, each executes:

```
forever{






          }
```

Workload

- Barford and Crovella: (SURGE) tool
- 64 processes, each executes:

```
forever{
        generate a file count, $n$, from Pareto($\alpha_1$,$k_1$);









          }
```

Workload

- Barford and Crovella: (SURGE) tool
- 64 processes, each executes:

```
forever{
        generate a file count, $n$, from Pareto($\alpha_1$,$k_1$);
        repeat($n$ times){







            }
        }
```

Workload

- Barford and Crovella: (SURGE) tool
- 64 processes, each executes:

```
forever{
        generate a file count, $n$, from Pareto($\alpha_1$,$k_1$);
        repeat($n$ times){
            select file from $L$ files using Zipf($L$);



            }
        }
```

Workload

- Barford and Crovella: (SURGE) tool
- 64 processes, each executes:

```
forever{
        generate a file count, $n$, from Pareto($\alpha_1$,$k_1$);
        repeat($n$ times){
            select file from $L$ files using Zipf($L$);
            while(file not read){
                read one page;
                generate $t$ from Pareto($\alpha_2$,$k_2$);
                sleep $t$ ms;
                }
            }
        }
```

Distributions

- Pareto (file count, time delays, large file sizes)

$$F_X(x) = 1 - (k/x)^\alpha \qquad x \geq k$$

Distributions

- Pareto (file count, time delays, large file sizes)

$$F_X(x) = 1 - (k/x)^{\alpha} \qquad x \geq k$$

- Zipf (file popularity)

$$p(i) = k/(i+1), \qquad i = 0, 1, \ldots, L$$

Distributions

- Pareto (file count, time delays, large file sizes)

$$F_X(x) = 1 - (k/x)^\alpha \qquad x \geq k$$

- Zipf (file popularity)

$$p(i) = k/(i+1), \qquad i = 0, 1, ..., L$$

- Lognormal (small file sizes)

$$F_Y(y) = \int_0^y e^{-\frac{(log_e t-\mu)^2}{2\sigma^2}} / (t\sigma\sqrt{2\pi})dt \qquad y > 0$$

Distributions

- Pareto (file count, time delays, large file sizes)

$$F_X(x) = 1 - (k/x)^{\alpha} \qquad x \geq k$$

- Zipf (file popularity)

$$p(i) = k/(i+1), \qquad i = 0, 1, ..., L$$

- Lognormal (small file sizes)

$$F_Y(y) = \int_0^y e^{-\frac{(log_e t - \mu)^2}{2\sigma^2}} / (t\sigma\sqrt{2\pi}) dt \qquad y > 0$$

Parameters from Barford-Crovella study

Results

- 64 processes, 50,000 requests, O_DIRECT

Results

- 64 processes, 50,000 requests, O_DIRECT

	real			virtual (k=8)		
sched.	*cats*	*dline*	*cfq*	*cats*	*dline*	*cfq*
μ_S	1.96	2.71	1.39	2.58	3.24	2.36
σ_S^2	8.51	9.76	5.85	9.03	8.23	7.78
μ_R	37.35	59.87	124.70	53.79	78.27	117.13
tput	8.19	6.08	2.19	6.15	5.06	3.38

- S and R in ms
- throughput in sectors/ms

Results

- 64 processes, 50,000 requests, non-O_DIRECT

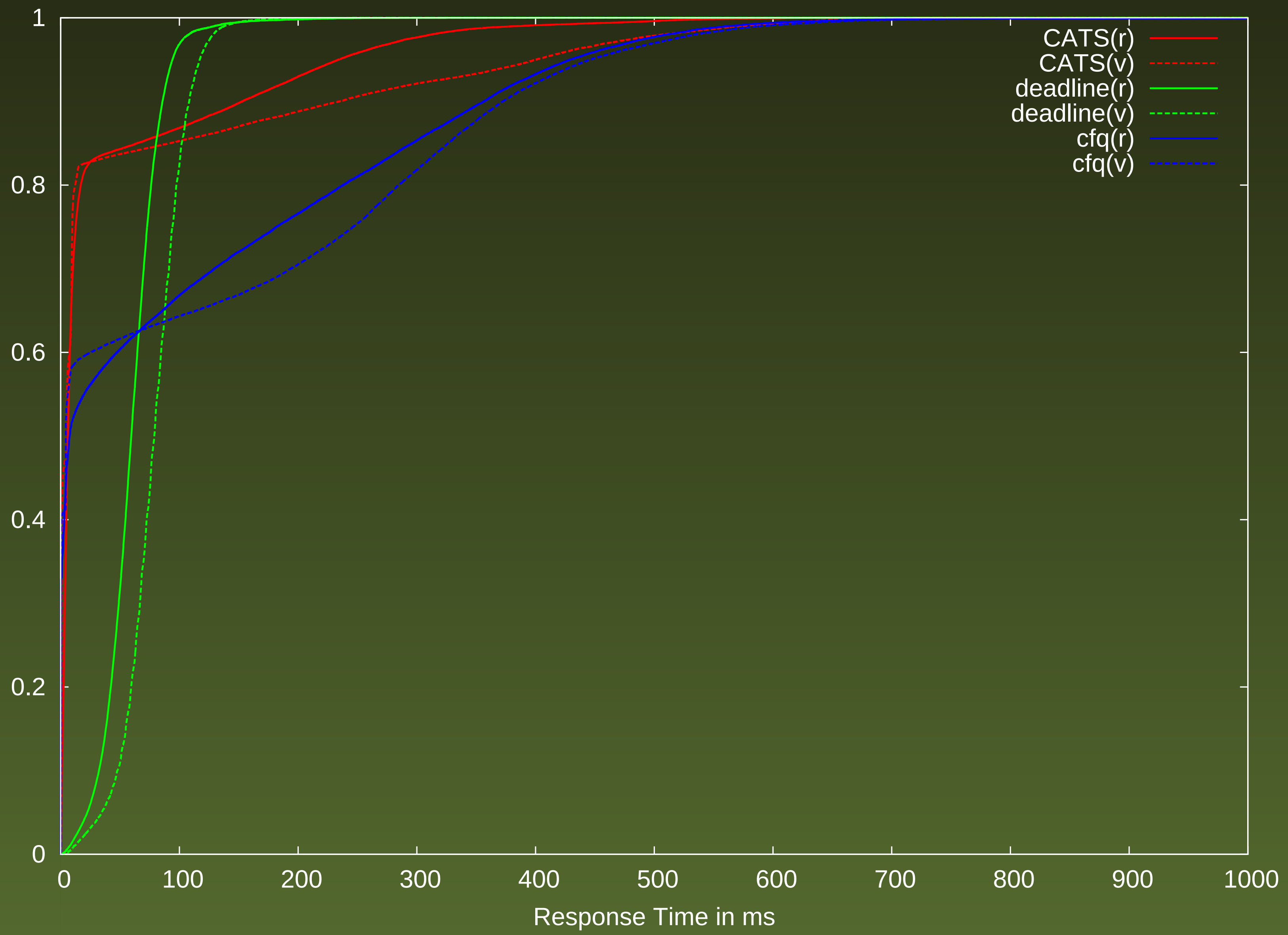
Results

- 64 processes, 50,000 requests, non-O_DIRECT

	real			virtual (k=8)		
	cats	*dline*	*cfq*	*cats*	*dline*	*cfq*
μ_S	6.53	7.41	7.80	7.15	7.60	8.57
σ_S^2	11.13	8.80	17.62	6.22	6.05	10.30
μ_R	114.91	121.87	179.17	189.45	198.33	258.75
tput	12.00	12.04	9.08	11.44	11.68	8.82

- S and R in ms
- throughput in sectors/ms

Results



1
0.8
0.6
0.4
0.2
0
0
100
200
300
400
500
600
700
800
900
1000
Response Time in ms
CATS(r)
CATS(v)
deadline(r)
deadline(v)
cfq(r)
cfq(v)

Conclusions

- New method for predicting (real) disk scheduler performance using only performance on virtual machines

Conclusions

- New method for predicting (real) disk scheduler performance using only performance on virtual machines
- Method uses new *iprobe* to force virtual service times to match simple service model

Conclusions

- New method for predicting (real) disk scheduler performance using only performance on virtual machines
- Method uses new *iprobe* to force virtual service times to match simple service model
- New disk scheduler (CATS) provided as case study

Conclusions

- New method for predicting (real) disk scheduler performance using only performance on virtual machines
- Method uses new *iprobe* to force virtual service times to match simple service model
- New disk scheduler (CATS) provided as case study
- Absolute performance predictions not yet accurate, but relative predictions are quite accurate

Conclusions

- New method for predicting (real) disk scheduler performance using only performance on virtual machines
- Method uses new *iprobe* to force virtual service times to match simple service model
- New disk scheduler (CATS) provided as case study
- Absolute performance predictions not yet accurate, but relative predictions are quite accurate
- Fair criticism: just using virtual Linux as elaborate simulator

Conclusions

- New method for predicting (real) disk scheduler performance using only performance on virtual machines
- Method uses new *iprobe* to force virtual service times to match simple service model
- New disk scheduler (CATS) provided as case study
- Absolute performance predictions not yet accurate, but relative predictions are quite accurate
- Fair criticism: just using virtual Linux as elaborate simulator
- True, but good results with almost zero programming effort!

Where has he been ... ?