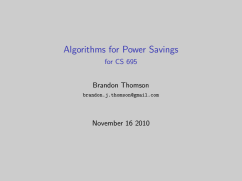
Algorithms for Power Savings

for CS 695

Brandon Thomson

brandon.j.thomson@gmail.com

November 16 2010



This talk was presented at George Mason University on Nov 16 2010.

Talk Overview

Optimizing job scheduling and hardware state to reduce energy use

Talk Overview

Optimizing job scheduling and hardware state to reduce energy use

This talk is about some of the hardware features that CPUs provide to save power, and algorithms that we can use to take advantage of those features.

First I'll tell you why we care about saving power.

Unlike some of the other algorithms that we've looked at, this one is really setup the way it is because of the way hardware is designed. If CPUs were designed in a different way, we'd be learning something different. So I want to show you how CPUs work so you understand why we model the problem the way we do.

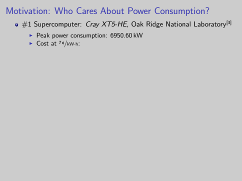
We'll also look at some variations on the problem that have been written about in other papers.

We won't have time to look at everything in today's paper, but we'll get as far as we can.

So first, let me tell you why we care about this problem.

Motivation: Who Cares About Power Consumption?

- #1 Supercomputer: *Cray XT5-HE*, Oak Ridge National Laboratory[3]
  - Peak power consumption: 6950.60 kW
  - Cost at 7 ¢/kW·h:



This supercomputer consumes almost 7 MEGAWATTS.

For comparison, an average nuclear fission plant generates about 700 megawatts. This computer uses 1% of a nuclear plant's capacity.

So if you wanted to run this thing full-bore for a year, it would only cost you. . . oh, I don't know. . . 4 MILLION DOLLARS

Motivation: Who Cares About Power Consumption?

- #1 Supercomputer: *Cray XT5-HE*, Oak Ridge National Laboratory[3]
  - Peak power consumption: 6950.60 kW
  - Cost at 7 ¢/kW·h: $4,261,740 per year

Motivation: Who Cares About Power Consumption?

- #1 Supercomputer: *Cray XT5-HE*, Oak Ridge National Laboratory[3]
  - Peak power consumption: 6950.60 kW
  - Cost at 7¢/kW·h: $4,261,740 per year
- For large-scale systems, reducing operational cost is important
- Case study: $200,000/year saved at Kyoto University[4]

Motivation: Who Cares About Power Consumption?

- #1 Supercomputer: *Cray XT5-HE*, Oak Ridge National Laboratory[3]
  - Peak power consumption: 6950.60 kW
  - Cost at 7¢/kW·h: $4,261,740 per year
- For large-scale systems, reducing operational cost is important
- Case study: $200,000/year saved at Kyoto University[4]

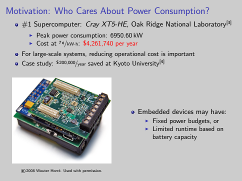
When you're talking about this much money, even if you have to pay someone to work on it for a year, it's not a bad deal.

Also, this is not just supercomputers... Any company that runs datacenters with lots of servers is interested in saving money through better power management.

Motivation: Who Cares About Power Consumption?

- #1 Supercomputer: *Cray XT5-HE*, Oak Ridge National Laboratory[3]
  - Peak power consumption: 6950.60 kW
  - Cost at 7 ¢/kW·h: $4,261,740 per year
- For large-scale systems, reducing operational cost is important
- Case study: $200,000/year saved at Kyoto University[4]

- Embedded devices may have:
  - Fixed power budgets, or
  - Limited runtime based on battery capacity



So that's the large scale. Huge systems. On the other side we have embedded devices.

It's not always practical to change batteries, especially when you have large deployments of wireless devices. So this is another case we care about.

We can also talk about the environment, or any other number of reasons, but suffice it to say that this is not just academic.

Background: Processor Speed Scaling

- CPUs support a fixed set of clock frequencies
  - Lower frequency $\rightarrow$ Lower voltage $\rightarrow$ Lower energy use
  - Examples: Intel's "SpeedStep," AMD's "PowerNOW"

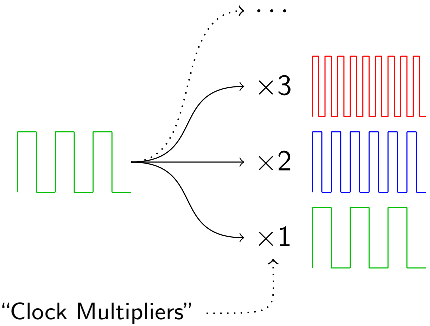
Background: Processor Speed Scaling

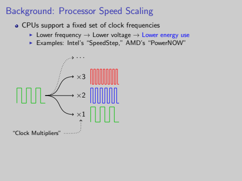
- CPUs support a fixed set of clock frequencies
  - Lower frequency → Lower voltage → Lower energy use
  - Examples: Intel's "SpeedStep," AMD's "PowerNOW"

We're going to be looking at an individual processor, so it's important to understand how they work.

Background: Processor Speed Scaling

- CPUs support a fixed set of clock frequencies
  - Lower frequency → Lower voltage → Lower energy use
  - Examples: Intel's "SpeedStep," AMD's "PowerNOW"



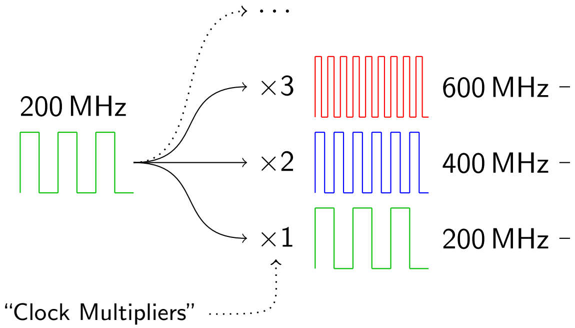


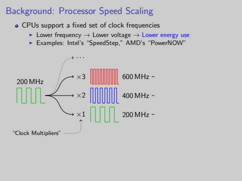
Usually there's a fixed set of frequencies that you can change to. The processor multiplies a slow input clock internally, thereby allowing it to run faster than the bus but still remaining synchronized to it.

Sometimes you can change this input bus clock on the fly and get arbitrary speeds for the CPU. but now you have to make sure all your other hardware supports the new clock speed too. And then you're changing two things at the same time. So just changing the CPU multiplier is much more widely supported. Much less messy.

Background: Processor Speed Scaling

- CPUs support a fixed set of clock frequencies
  - Lower frequency → Lower voltage → Lower energy use
  - Examples: Intel's "SpeedStep," AMD's "PowerNOW"

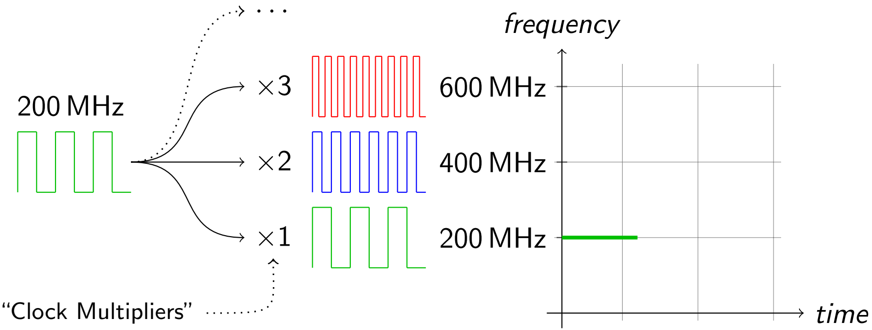




Let's put up some example speeds and see how the frequency-changing process works. First we start out at some speed, say 200 MHz.

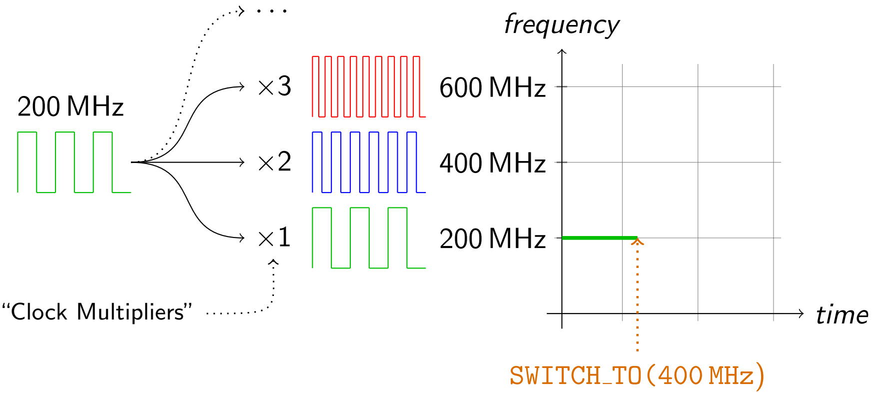
Background: Processor Speed Scaling

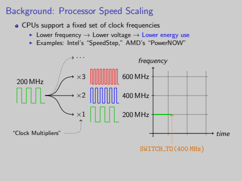
- CPUs support a fixed set of clock frequencies
  - Lower frequency → Lower voltage → Lower energy use
  - Examples: Intel's "SpeedStep," AMD's "PowerNOW"



Background: Processor Speed Scaling

- CPUs support a fixed set of clock frequencies
  - Lower frequency → Lower voltage → Lower energy use
  - Examples: Intel's "SpeedStep," AMD's "PowerNOW"

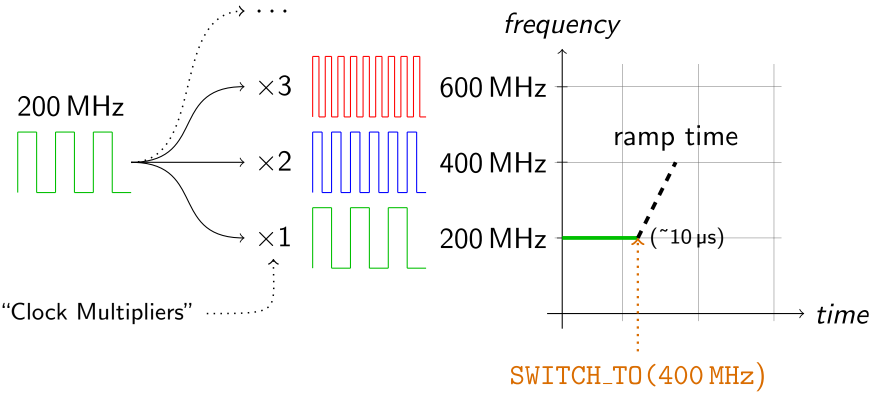




Then the OS issues a command to switch. There's a delay while all sorts of fun electrical stuff is happening. You can't get work done during that period.

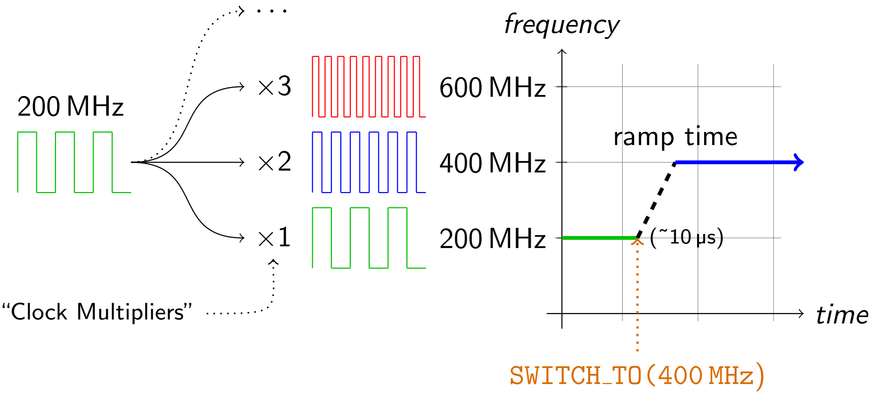
Background: Processor Speed Scaling

- CPUs support a fixed set of clock frequencies
  - Lower frequency → Lower voltage → Lower energy use
  - Examples: Intel's "SpeedStep," AMD's "PowerNOW"

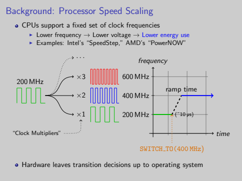


Background: Processor Speed Scaling

- CPUs support a fixed set of clock frequencies
  - Lower frequency → Lower voltage → Lower energy use
  - Examples: Intel's "SpeedStep," AMD's "PowerNOW"



- Hardware leaves transition decisions up to operating system



Note that the hardware doesn't manage it's own speeds; the OS has to do that. This is also true for sleep states, which we're going to look at next.

Background: Processor Sleep States

- Many CPUs support fixed set of "sleep states"
- Deeper sleep states:
  - Save more power
  - Have higher "return-to-service" latency
- Non-trivial transition delay (compared to speed scaling)

Background: Processor Sleep States

- Many CPUs support fixed set of "sleep states"
- Deeper sleep states:
  - Save more power
  - Have higher "return-to-service" latency
- Non-trivial transition delay (compared to speed scaling)

The general principle is, the more stuff you turn off, the longer it takes to resynchronize and get you back to a state where you can execute. Delays here are much more likely to be significant compared to speed scaling. The deeper sleep states are on the order of ms. Some papers talk about suspending or hibernating an entire computer, which is on the order of seconds.

Background: Processor Sleep States

- Many CPUs support fixed set of "sleep states"
- Deeper sleep states:
  - Save more power
  - Have higher "return-to-service" latency
- Non-trivial transition delay (compared to speed scaling)
- Intel sleep state examples[5]:
  - C0 - Active: CPU on.
  - C1 - Auto Halt: no execution; can return to executing state quickly.
  - C2 - Stop Clock: core and bus clocks off.
  - C3 - Deep Sleep: all clock circuitry off, cache flushed to main memory.
  - C4 - Deeper Sleep: reduced voltage.
- Ugly details. Sometimes hardware:
  - has to be at slowest speed to go to sleep
  - always wakes in slowest speed
  - behaves abnormally in sleep states
  - ...

Background: Processor Sleep States

- Many CPUs support fixed set of "sleep states"
- Deeper sleep states:
  - Save more power
  - Have higher "return-to-service" latency
- Non-trivial transition delay (compared to speed scaling)
- Intel sleep state examples[5]:
  - C0 - Active: CPU on.
  - C1 - Auto Halt: no execution; can return to executing state quickly.
  - C2 - Stop Clock: core and bus clocks off.
  - C3 - Deep Sleep: all clock circuitry off, cache flushed to main memory.
  - C4 - Deeper Sleep: reduced voltage.
- Ugly details. Sometimes hardware:
  - has to be at slowest speed to go to sleep
  - always wakes in slowest speed
  - behaves abnormally in sleep states
  - . . .

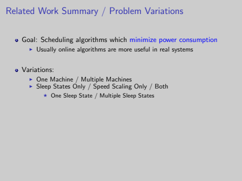
The general principle is, the more stuff you turn off, the longer it takes to resynchronize and get you back to a state where you can execute. Delays here are much more likely to be significant compared to speed scaling. The deeper sleep states are on the order of ms. Some papers talk about suspending or hibernating an entire computer, which is on the order of seconds. Often individual hardware has its own quirks. So as an OS programmer, if you want an algorithm that supports everything, that can be difficult

If I enable C4 sleep on my laptop, every time I go to move the cursor there's a delay. The USB interrupt comes in and then the thing has to wakeup and repopulate the cache, and it takes long enough that it's noticeable.

Related Work Summary / Problem Variations

- Goal: Scheduling algorithms which minimize power consumption
  - Usually online algorithms are more useful in real systems

- Variations:
  - One Machine / Multiple Machines
  - Sleep States Only / Speed Scaling Only / Both
    - One Sleep State / Multiple Sleep States



Systems handle scheduling differently, so there's value in looking at many ways of setting up the problem.

Especially in this simplest single sleep state case, we don't have to be talking about a CPU. This could be hibernating an entire computer, or turning off the wireless radio on a laptop, or... whatever you can think of that can be turned off when it's idle.

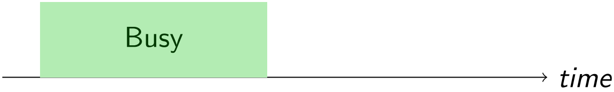
Related Work Summary / Problem Variations

- Goal: Scheduling algorithms which minimize power consumption
  - Usually online algorithms are more useful in real systems

- Variations:
  - One Machine / Multiple Machines
  - Sleep States Only / Speed Scaling Only / Both
    - One Sleep State / Multiple Sleep States

Repeated Continuous Ski-Rental Problem

Repeated Continuous Ski-Rental Problem

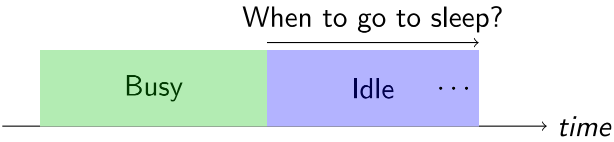


Repeated Continuous Ski-Rental Problem

Repeated Continuous Ski-Rental Problem

- If idle period is long enough, sleeping is “worth it”
- Should sleep immediately after busy if upcoming idle period is “worth it”

Repeated Continuous Ski-Rental Problem



- If idle period is long enough, sleeping is “worth it”
- Should sleep immediately after busy if upcoming idle period is “worth it”
- Repeated:

Repeated Continuous Ski-Rental Problem

- If idle period is long enough, sleeping is “worth it”
- Should sleep immediately after busy if upcoming idle period is “worth it”
- Repeated:

- More advanced versions:
  - Assume idle periods conform to known probability distribution
  - “Learn” and change strategy based on recent idle period lengths

Related Work Summary / Problem Variations

- Goal: Scheduling algorithms which minimize power consumption
  - Usually online algorithms are more useful in real systems

- Variations:
  - One Machine / Multiple Machines
  - Sleep States Only / Speed Scaling Only / Both
    - ★ One Sleep State / Multiple Sleep States

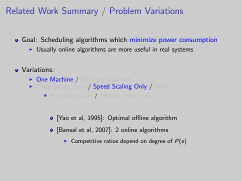
Repeated Continuous Ski-Rental Problem

Related Work Summary / Problem Variations

- Goal: Scheduling algorithms which minimize power consumption
  - Usually online algorithms are more useful in real systems

- Variations:
  - One Machine / Multiple Machines
  - Sleep States Only / Speed Scaling Only / Both
    - ★ One Sleep State / Multiple Sleep States

- [Yao et al, 1995]: Optimal offline algorithm
- [Bansal et al, 2007]: 2 online algorithms
  - Competitive ratios depend on degree of $P(s)$



The speed scaling case is not necessarily exclusive to CPUs... For example some hard disks support multiple speeds... pretty much any device that is clocked... but the CPU is by far the most common case.

Related Work Summary / Problem Variations

- Goal: Scheduling algorithms which minimize power consumption
  - Usually online algorithms are more useful in real systems

- Variations:
  - One Machine / Multiple Machines
  - Sleep States Only / Speed Scaling Only / Both
    - One Sleep State / Multiple Sleep States

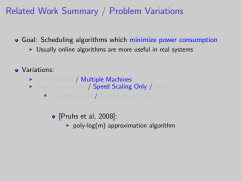
"Optimal Powerdown Strategies[2]"

"Online Strategies for Dynamic Power Management in Systems with Multiple Power-Saving States[7]"

Related Work Summary / Problem Variations

- Goal: Scheduling algorithms which minimize power consumption
  - Usually online algorithms are more useful in real systems
- Variations:
  - One Machine / Multiple Machines
  - Sleep States Only / Speed Scaling Only / Both
    - One Sleep State / Multiple Sleep States

- [Pruhs et al, 2008]:
  - poly-log(m) approximation algorithm



You can come up with other variations here... For example different papers treat job scheduling differently... but I want to spend at least a little time looking at the algorithm setup from the paper.

Related Work Summary / Problem Variations

- Goal: Scheduling algorithms which minimize power consumption
  - Usually online algorithms are more useful in real systems
- Variations:
  - One Machine / Multiple Machines
  - Sleep States Only / Speed Scaling Only / Both
    - One Sleep State / Multiple Sleep States

Tonight:

- "Algorithms for Power Savings[1]"
  - offline algorithm: within 2x of optimal
  - online algorithm: constant competitive ratio

Problem Definition: Input

- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j

Problem Definition: Input

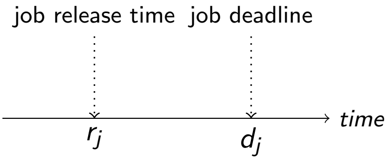
- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j

This is fairly similar to single machine scheduling so far. Note that we have work units instead of duration or processing time.

This setup with a release time and deadline is pretty standard... but obviously we only have those in real-time systems. In multi-user operating systems we're more interested in things like fairness and lack of starvation.

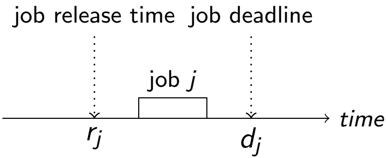
Problem Definition: Input

- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j



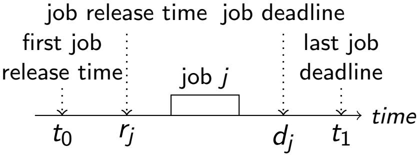
Problem Definition: Input

- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j



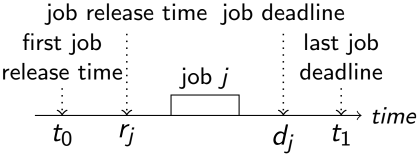
Problem Definition: Input

- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j



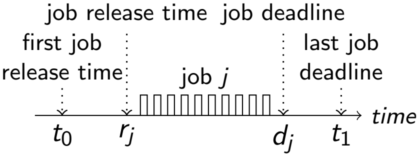
Problem Definition: Input

- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j
- Online algorithm learns of job at r_j
- One job at a time
- No suspend/resume delay
- No state transition delay



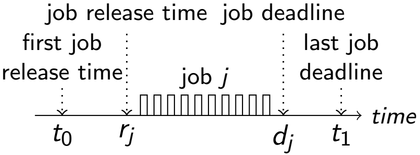
Problem Definition: Input

- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j
- Online algorithm learns of job at r_j
- One job at a time
- No suspend/resume delay
- No state transition delay



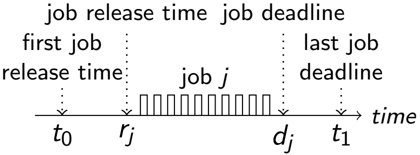
Problem Definition: Input

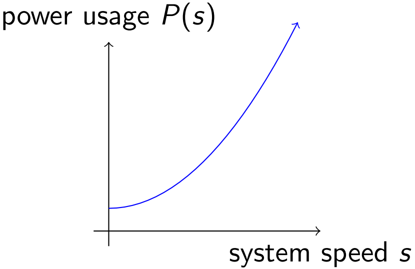
- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j
- Online algorithm learns of job at r_j
- One job at a time
- No suspend/resume delay
- No state transition delay
- function $P(s)$ is:
  - non-decreasing
  - unbounded
  - convex
  - continuous
- $P(0) > 0$



Problem Definition: Input

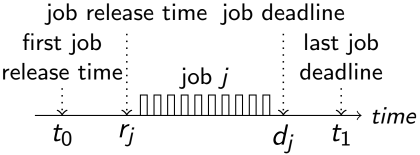
- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j
- Online algorithm learns of job at r_j
- One job at a time
- No suspend/resume delay
- No state transition delay
- function $P(s)$ is:
  - non-decreasing
  - unbounded
  - convex
  - continuous
- $P(0) > 0$

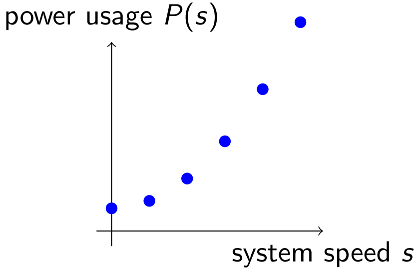


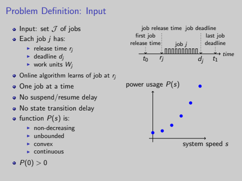


Problem Definition: Input

- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j
- Online algorithm learns of job at r_j
- One job at a time
- No suspend/resume delay
- No state transition delay
- function $P(s)$ is:
  - non-decreasing
  - unbounded
  - convex
  - continuous
- $P(0) > 0$



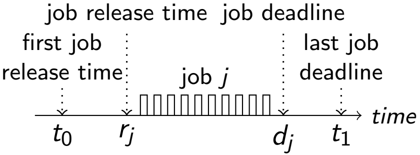


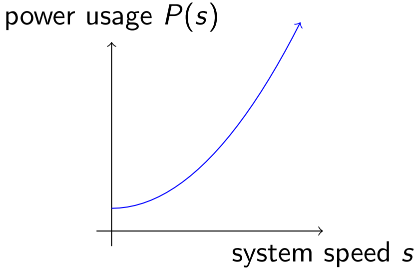


Note that since real CPUs usually support discrete speed states, this would be more realistically modeled as a set of points. Some papers do it that way, but then you lose the ability to integrate etc so it's a tradeoff. Here, we'll treat it as continuous.

Problem Definition: Input

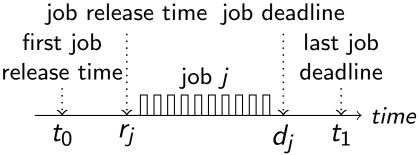
- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j
- Online algorithm learns of job at r_j
- One job at a time
- No suspend/resume delay
- No state transition delay
- function $P(s)$ is:
  - non-decreasing
  - unbounded
  - convex
  - continuous
- $P(0) > 0$

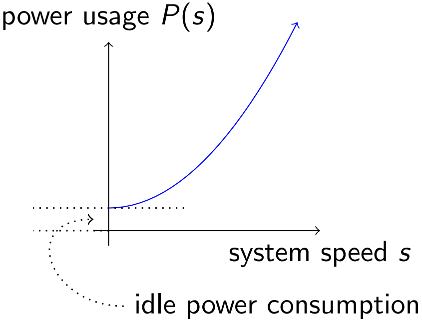




Problem Definition: Input

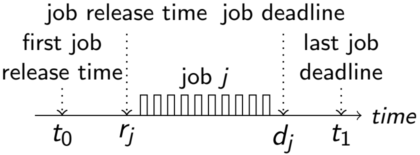
- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j
- Online algorithm learns of job at r_j
- One job at a time
- No suspend/resume delay
- No state transition delay
- function $P(s)$ is:
  - non-decreasing
  - unbounded
  - convex
  - continuous
- $P(0) > 0$

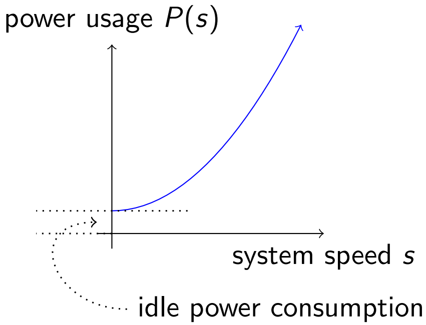




Problem Definition: Input

- Input: set $\mathcal{J}$ of jobs
- Each job j has:
  - release time r_j
  - deadline d_j
  - work units W_j
- Online algorithm learns of job at r_j
- One job at a time
- No suspend/resume delay
- No state transition delay
- function $P(s)$ is:
  - non-decreasing
  - unbounded
  - convex
  - continuous
- $P(0) > 0$, $P(sleeping) = 0$





Problem Definition: Output

- Output: Schedule $\mathcal{S} = (s, \phi, job)$

$$\begin{aligned} s\left(t\right) &: \text{system speed at time } t \\ job\left(t\right) &: \text{job executing at time } t \\ \phi\left(t\right) &: \text{sleep status at time } t \end{aligned}$$

Problem Definition: Output

- Output: Schedule $\mathcal{S} = (s, \phi, job)$

$$\begin{aligned} s\,(t) &: \text{system speed at time } t \\ job\,(t) &: \text{job executing at time } t \\ \phi\,(t) &: \text{sleep status at time } t \end{aligned}$$

- $\mathcal{S}$ is *feasible* if all jobs completed between *release* and *deadline*.

Problem Definition: Output

- Output: Schedule $\mathcal{S} = (s, \phi, job)$

$$
\begin{aligned}
s(t) &: \text{system speed at time } t \\
job(t) &: \text{job executing at time } t \\
\phi(t) &: \text{sleep status at time } t
\end{aligned}
$$

- $\mathcal{S}$ is *feasible* if all jobs completed between *release* and *deadline*.

$$
\texttt{cost}(\mathcal{S}) = k + \int_{t_0}^{t_1} P(s(t), \phi(t)) \text{ dt}
$$

- Goal: Find a feasible $\mathcal{S}$ that minimizes $\texttt{cost}(\mathcal{S})$.

Problem Definition: Output

- Output: Schedule $\mathcal{S} = (s, \phi, job)$

$$s(t) : \text{system speed at time } t$$
$$job(t) : \text{job executing at time } t$$
$$\phi(t) : \text{sleep status at time } t$$

- $\mathcal{S}$ is *feasible* if all jobs completed between *release* and *deadline*.

$$\texttt{cost}(\mathcal{S}) = k + \int_{t_0}^{t_1} P(s(t), \phi(t)) \text{ dt}$$

- Goal: Find a feasible $\mathcal{S}$ that minimizes $\texttt{cost}(\mathcal{S})$.

Problem Definition: Output

- Output: Schedule $\mathcal{S} = (s, \phi, job)$

$$s(t) : \text{system speed at time } t$$
$$job(t) : \text{job executing at time } t$$
$$\phi(t) : \text{sleep status at time } t$$

- $\mathcal{S}$ is *feasible* if all jobs completed between *release* and *deadline*.

$$\texttt{cost}(\mathcal{S}) = k + \int_{t_0}^{t_1} P(s(t), \phi(t)) \text{ dt}$$

- Goal: Find a feasible $\mathcal{S}$ that minimizes $\texttt{cost}(\mathcal{S})$.

Example

- $P(s) = s^3 + 16$

Example

- $P(s) = s^3 + 16$
  $\rightarrow$ Running Power Consumption
- "cube-root-rule"

Example

- $P(s) = s^3 + 16$
  → Running Power Consumption
- "cube-root-rule"

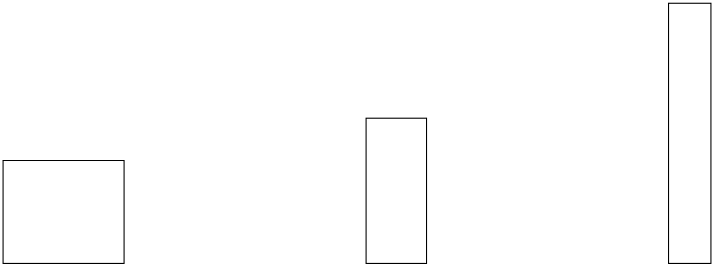
The cube-root-rule says that a cubic function is a pretty good approximation for power usage at a given speed.
(By the way, that's why we're stuck around 3 GHz... The power usage is increasing with the cube, so it starts getting ridiculous beyond that point)

Example

- $P(s) = s^3 + 16$
  $\rightarrow$ Idle Power Consumption
- "cube-root-rule"

Example

- $P(s) = s^3 + 16$
- “cube-root-rule”
- Power usage/duration of job at different speeds?



$s = 1$ $s = 2$ $s = 3$

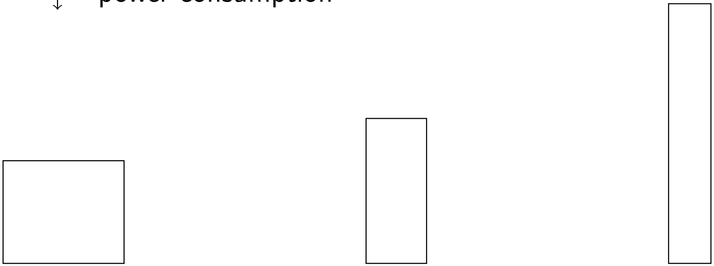
Example

- $P(s) = s^3 + 16$
- "cube-root-rule"

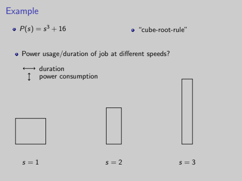
- Power usage/duration of job at different speeds?

$\longleftrightarrow$ duration

$\updownarrow$ power consumption



$s = 1$ $s = 2$ $s = 3$



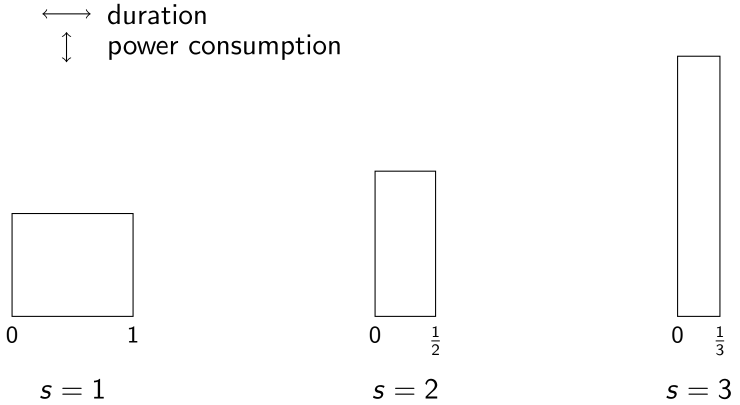
The fact that the idle power consumption is decreasing while the running power consumption is increasing means there's going to be a critical point somewhere in this middle.

In this example, that's at $s = 2$. If you sum up the area of the boxes, the center one is only 12, whereas both of the ones on the ends are larger.

Example

- $P(s) = s^3 + 16$
- "cube-root-rule"

- Power usage/duration of job at different speeds?

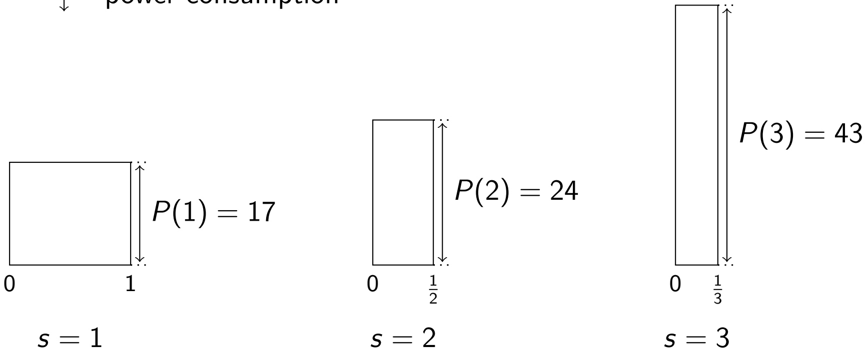


Example

- $P(s) = s^3 + 16$
- "cube-root-rule"

- Power usage/duration of job at different speeds?

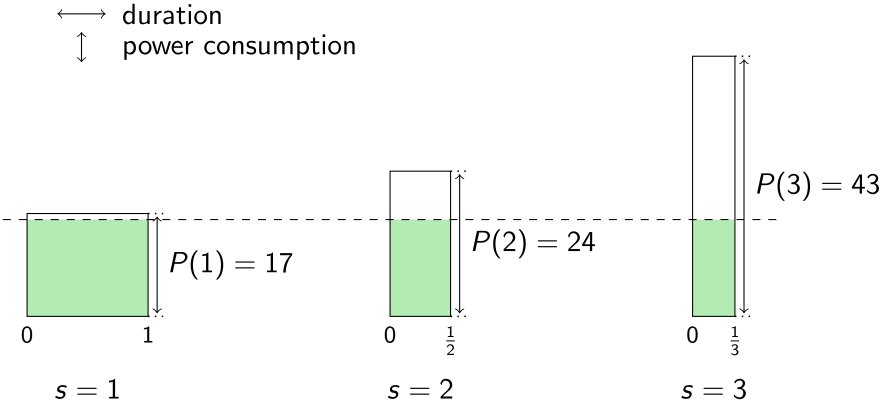
$\longleftrightarrow$ duration
$\updownarrow$ power consumption



Example

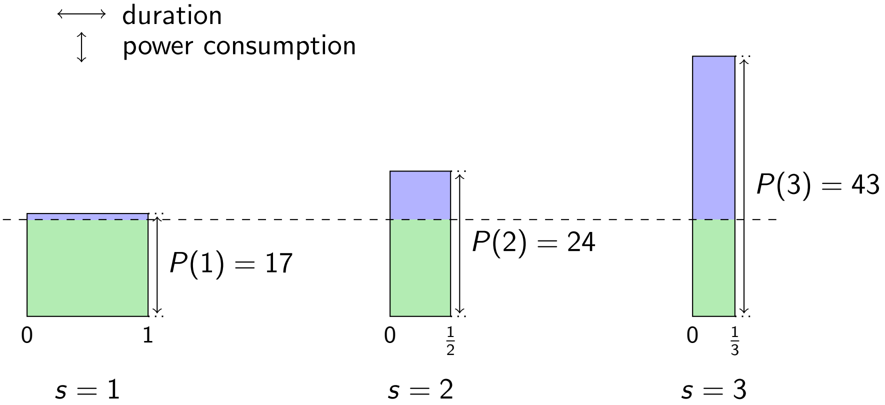
- $P(s) = s^3 + 16$
  $\rightarrow$ Idle Power Consumption
- "cube-root-rule"

- Power usage/duration of job at different speeds?



Example

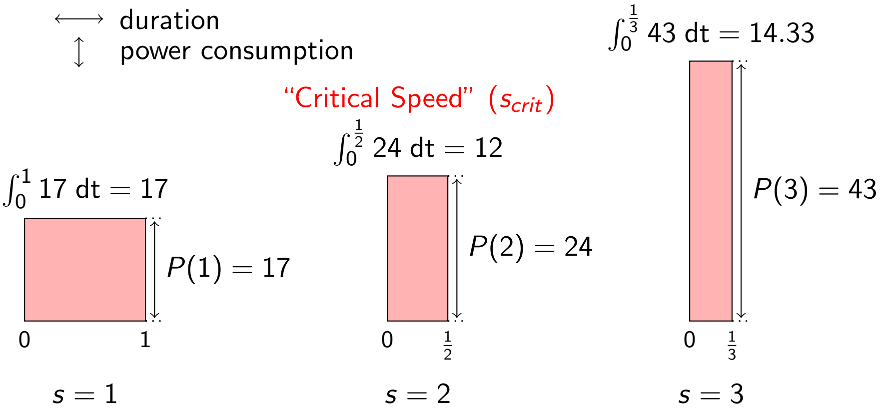
- $P(s) = s^3 + 16$
  $\rightarrow$ Running Power Consumption
- "cube-root-rule"
- Power usage/duration of job at different speeds?



Example

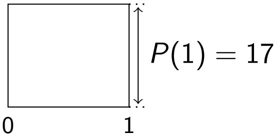
- $P(s) = s^3 + 16$
- "cube-root-rule"

- Power usage/duration of job at different speeds?

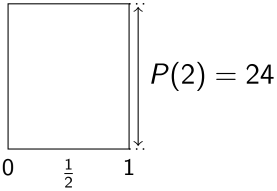


Critical Speed = Optimal Speed?

- No. Sometimes we may want to run slower:



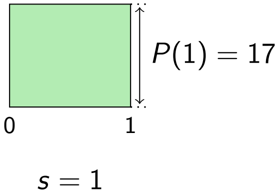
$s = 1$

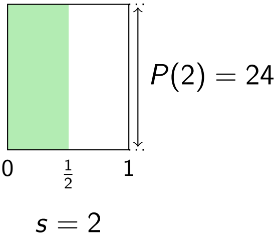


$s = 2$

Critical Speed = Optimal Speed?

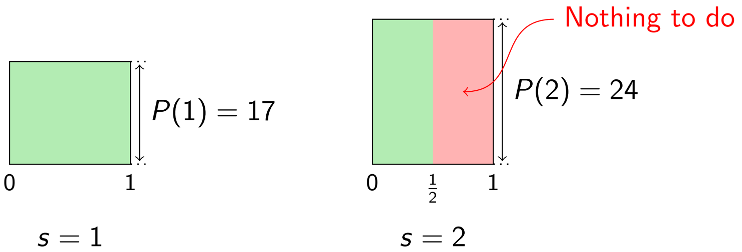
- No. Sometimes we may want to run slower:





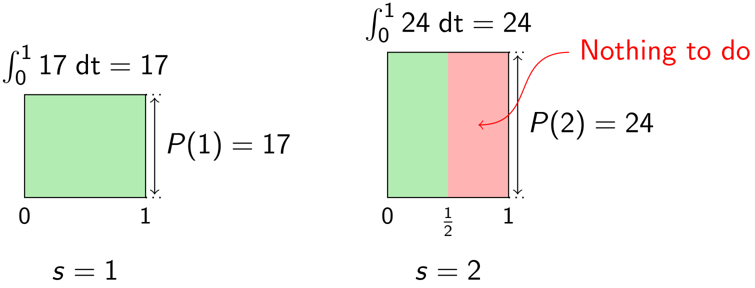
Critical Speed = Optimal Speed?

- No. Sometimes we may want to run slower:



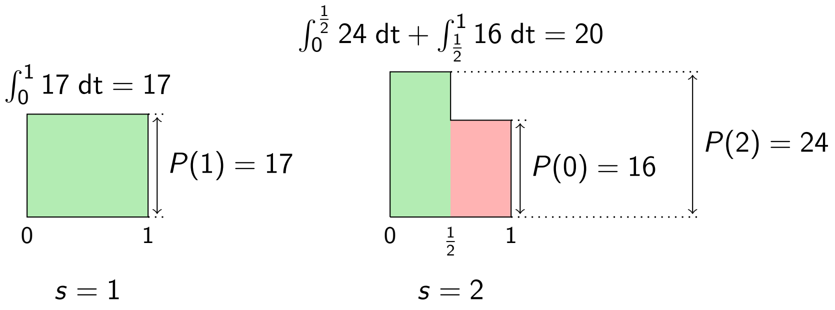
Critical Speed = Optimal Speed?

- No. Sometimes we may want to run slower:



Critical Speed = Optimal Speed?

- No. Sometimes we may want to run slower:

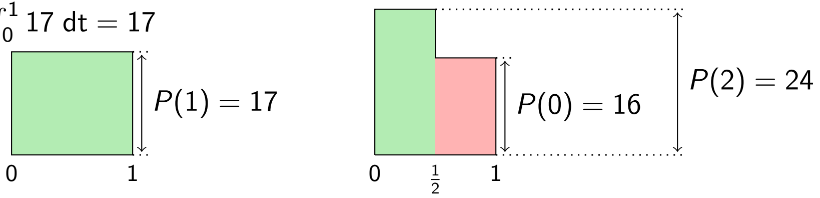


- Running at constant minimum constant speed to finish job in interval is better than running at s_{crit} and then dropping to idle

Critical Speed = Optimal Speed?

- No. Sometimes we may want to run slower:

$$\int_0^1 17\ dt = 17 \qquad \int_0^{\frac{1}{2}} 24\ dt + \int_{\frac{1}{2}}^1 16\ dt = 20$$



$s = 1$ $\qquad$ $s = 2$

- Running at constant minimum constant speed to finish job in interval is better than running at s_{crit} and then dropping to idle
- Running faster than s_{crit} is always wasteful
  - use only if required to meet deadlines

Finding the Critical Speed

- s_{crit}: first zero of $\left(\frac{P(s)}{s}\right)'$.

- (details about perverse cases omitted)

Finding the Critical Speed

- s_{crit}: first zero of $\left(\frac{P(s)}{s}\right)'$.
- For our example $P(s) = s^3 + 16$:

$$P'(s) = 3s^2$$

$$\left(\frac{P(s)}{s}\right)' = \frac{sP'(s) - P(s)}{s^2} = \frac{2s^3 - 16}{s^2}$$

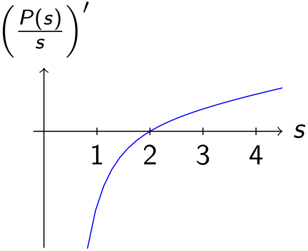
- (details about perverse cases omitted)

Finding the Critical Speed

- s_{crit}: first zero of $\left(\frac{P(s)}{s}\right)'$.
- For our example $P(s) = s^3 + 16$:

$$P'(s) = 3s^2$$

$$\left(\frac{P(s)}{s}\right)' = \frac{sP'(s) - P(s)}{s^2} = \frac{2s^3 - 16}{s^2}$$

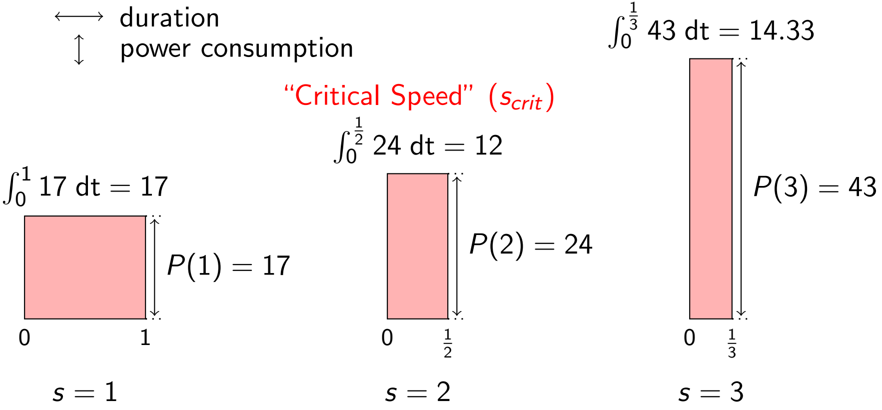


- (details about perverse cases omitted)

Example

- $P(s) = s^3 + 16$
- "cube-root-rule"

- Power usage/duration of job at different speeds?



Summary

- Proper power management saves money and the environment
- CPUs support software-controlled:
  - clock speeds
  - sleep states
- Varying hardware configurations inspire many different algorithms
  - Sleep-state algorithms can be used with many kinds of devices
- "Algorithms for Power Savings"
  - Online/Offline algorithms for single machine with speed scaling and a single sleep state

Irani, S. Shukla, S., and Gupta, R.
Algorithms for power savings.
ACM Trans. Algor. 3, 4, Article 41 (November 2007), 23 pages.

Yao, F., Demers, A., and Shenker, S. 1995.
A scheduling model for reduced CPU energy.
In Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS).
IEEE Computer Society, Washington, DC, 374.

John Augustine, Sandy Irani, and Chaitanya Swamy.
Optimal Power-Down Strategies.
In Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS '04).
IEEE Computer Society, Washington, DC, USA, 530-539.

TOP500 List
http://www.top500.org/list/2010/06/100, June 2010.

Hikita, J.; Hirano, A.; Nakashima, H.;
Saving 200kW and $200 K/year by power-aware job/machine scheduling
Parallel and Distributed Processing, 2008.
IPDPS 2008. IEEE International Symposium on, pp.1-8.

What is difference between deep and deeper sleep states?
http://www.intel.com/support/processors/sb/cs-028739.htm, Oct 2010.

Birks, Martin and Fung, Stanley.
Temperature Aware Online Scheduling with a Low Cooling Factor
Theory and Applications of Models of Computation.
Lecture Notes in Computer Science, 2010

Sandy Irani, Sandeep Shukla, and Rajesh Gupta.
Online strategies for dynamic power management in systems with multiple power-saving states.
ACM Trans. Embed. Comput. Syst. 2, 3 (August 2003), 325-346.

Bansal, N., Kimbrel, T., and Pruhs, K. 2007.
Speed scaling to manage energy and temperature.
J. ACM 54, 1, 1.

Pruhs, Kirk and van Stee, Rob and Uthaisombut, Patchrawat,
Speed Scaling of Tasks with Precedence Constraints,
Theory of Computing Systems, vol 43. 67-80, Springer New York.