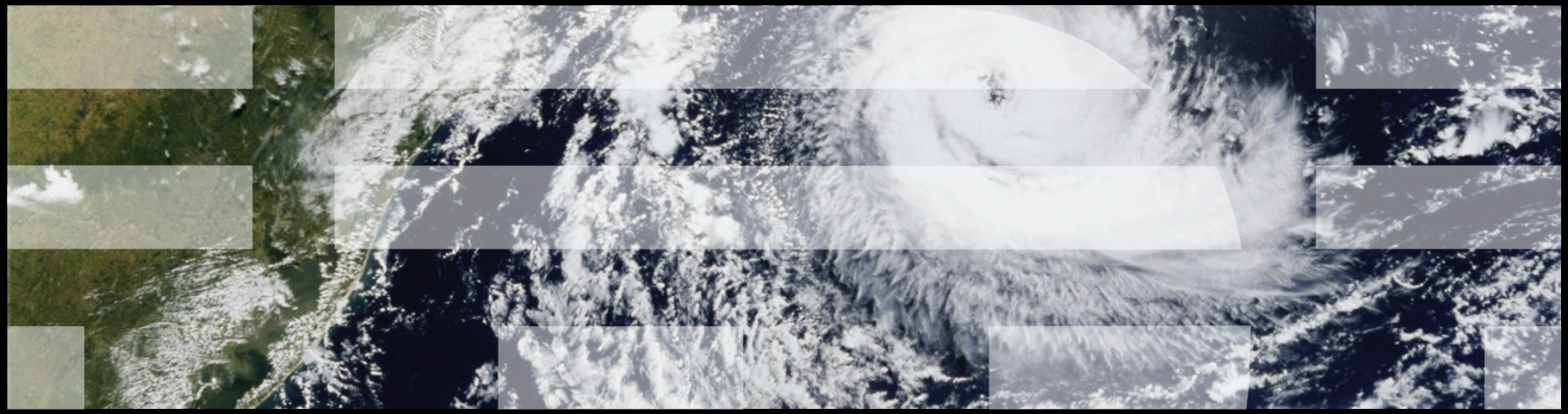
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center

21 October 2011 (Revised)

On migrate_disable() and Latencies



Overview

- Approach
- Disabling Preemption vs. Disabling Migration
- Description of Algorithms
- Results
- Remaining Challenges
- Summary and Conclusions

Approach

Approach

- Decisions, decisions...
- Door #1: Empirical approach
  - Which gives exact results for a workload that nobody really uses
  - (These workloads are called “benchmarks”)
- Door #2: Analytic approach
  - Obtain general results, but using hopelessly unrealistic assumptions
  - (Just in case you want the solution in finite time and space)
- This presentation uses Door #2
  - And therefore makes a number of simplifying assumptions...

Simplifying Assumptions

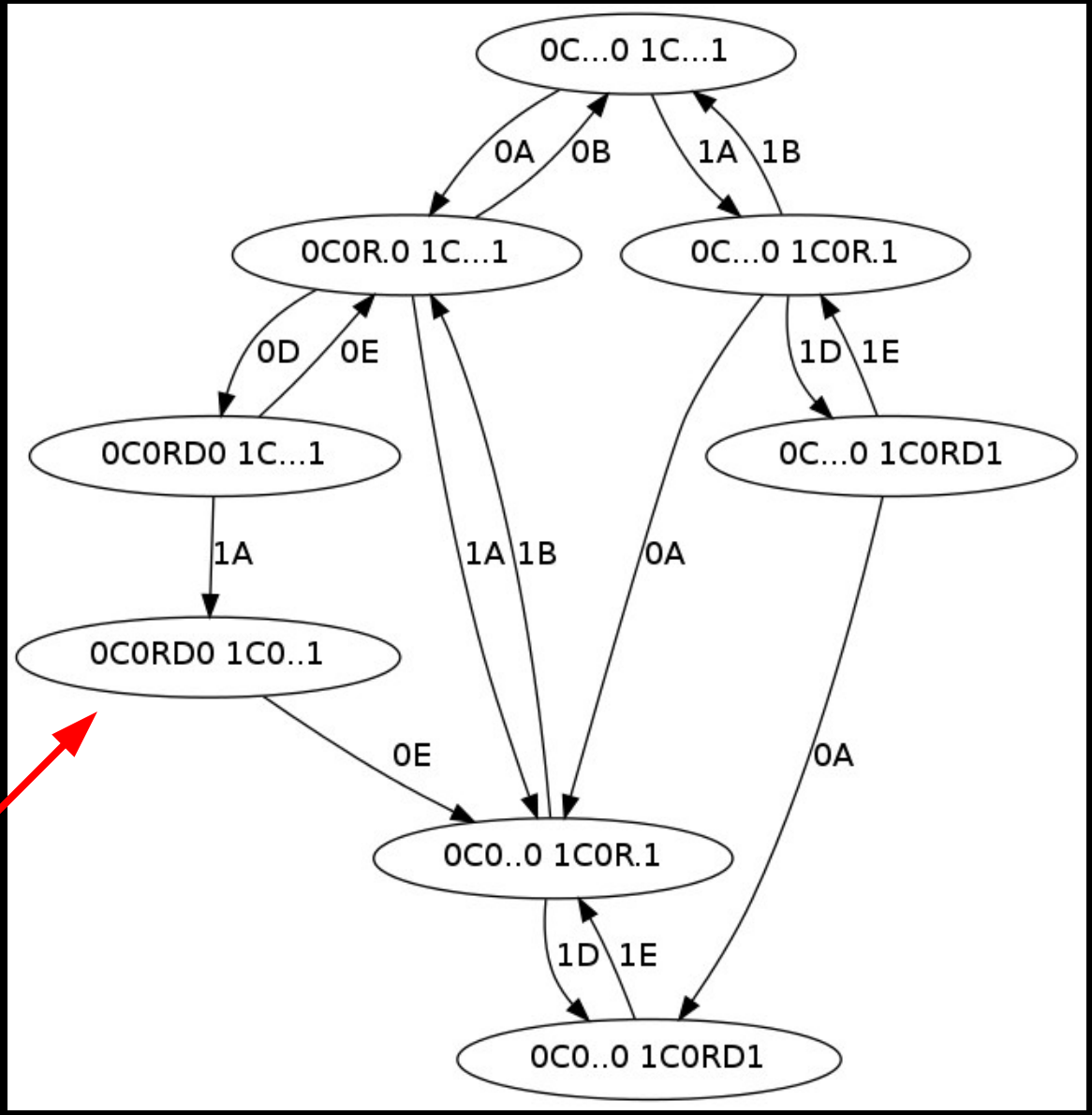
- The periods between a given task's events are memoryless
  - Exponentially distributed “interarrival rates”
  - Think in terms of how many times a task disables preemption per second of CPU time that it consumes while not being preempted
  - In reality history really does matter
- CPUs are interchangeable at all points in time
  - Completely ignore cache-affinity effects
  - Critically important for state-space reduction
    - For example, all preempted tasks are associated with CPU 0
- Each task is running identical “workload” at a unique priority
  - And lower-priority tasks are real-fast rather than real-time
- Omniscient “scheduler”
  - For example, full rebalancing on each enable event
- Interarrival rates within a few orders of magnitude of each other
  - Otherwise roundoff error kills double-precision floating point accuracy

Disabling Preemption vs. Disabling Migration

Disabling Preemption vs. Disabling Migration

- A task that disables preemption delays higher-priority tasks
  - That would be what “disables preemption” means, after all...
- Disabling preemption has numerous uses:
  - Protect access to per-CPU variables
  - Block scheduling
  - Block CPU-hotplug operations
- Disabling preemption sometimes bigger hammer than needed
- So recent -rt kernels typically disable migration rather than preemption
- The difference is illustrated on the next two slides
  - Two tasks running on a single CPU

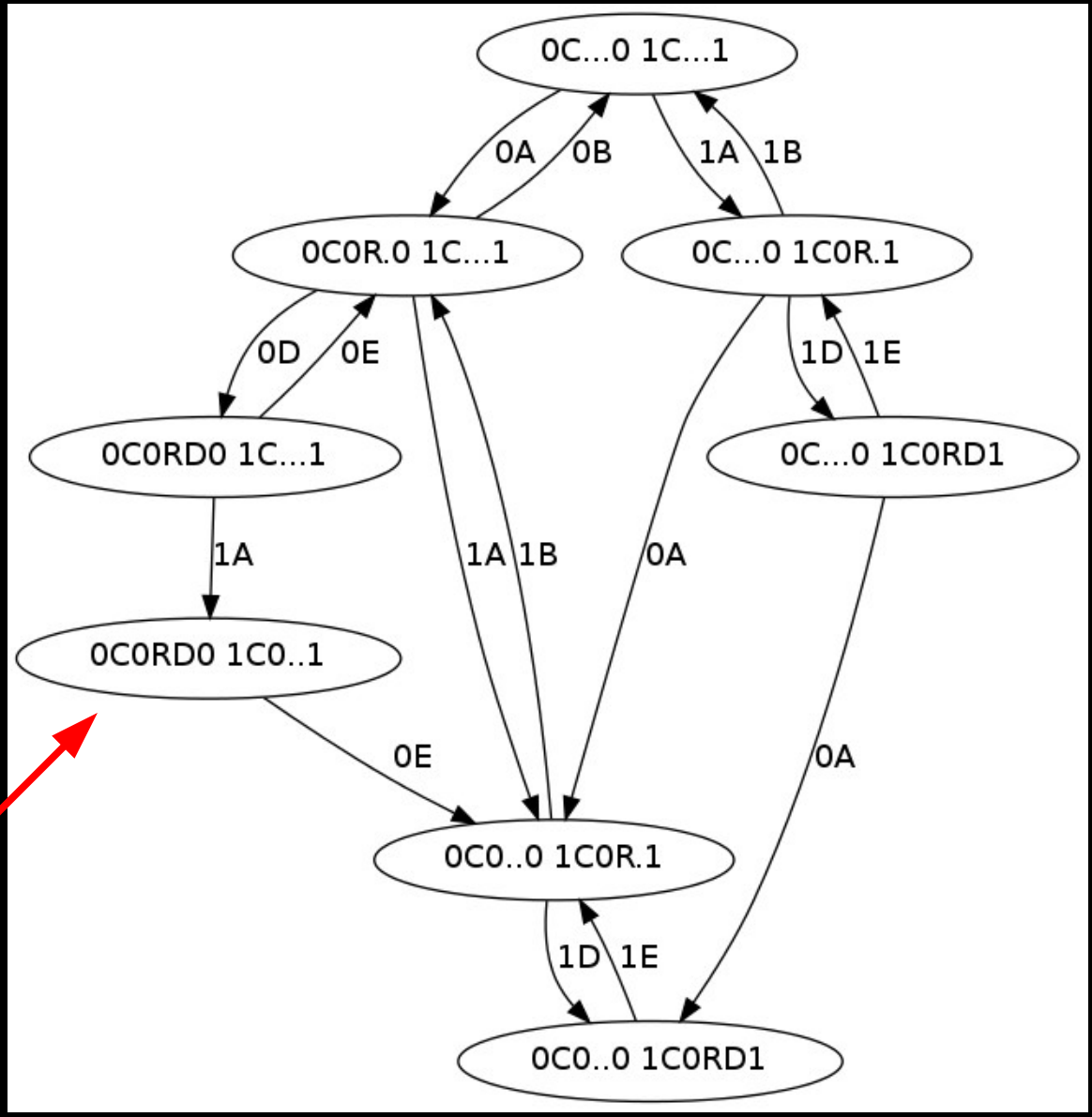
Disabling Preemption



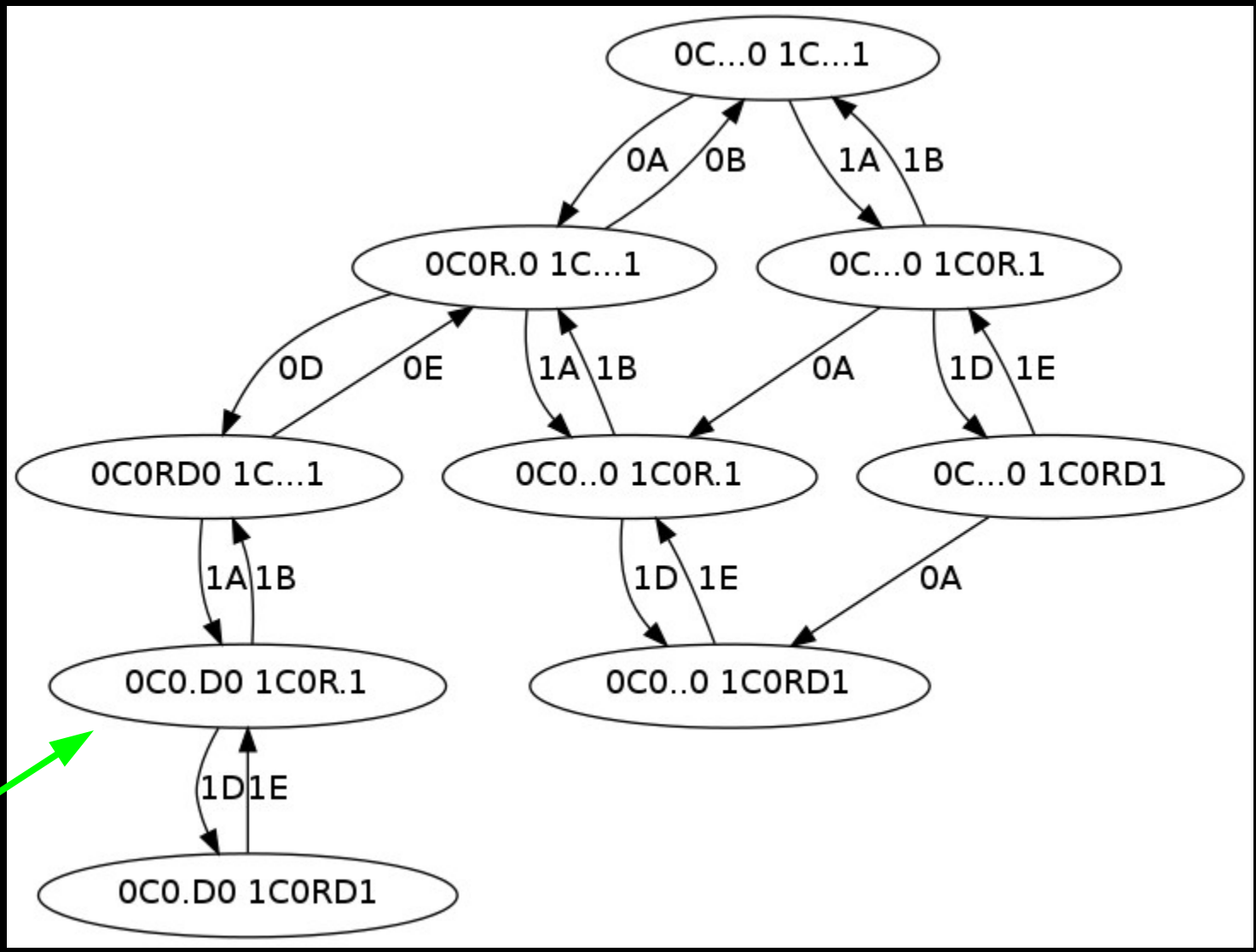
Key to State and Transition Labels

- State label example: “0C0RD0 1C...1” (two tasks)
  - First character: task #
  - Second character: “C” for “CPU”
  - Third character: Task's CPU or “.” if no CPU
  - Fourth character: “R” if running, “.” otherwise
  - Fifth character: “D” if disabled, “.” otherwise
  - Final character: Priority ranging from 0 to 9
    - Task 0 is running disabled on CPU 0 at priority 0, while task 1 is blocked (but would run at priority 1 if it was running)
- Transition label example: “1A*”
  - First character: affected task (in this case, task 1)
  - Second character: A for awaken, B for block, D for disable, E for enable
  - Optional final character: mapped to mathematically equivalent state
- Back to the two-task, single-CPU, preempt-disable diagram...

Disabling Preemption (Take 2)



Disabling Migration



The Diagram for Four Tasks on Two CPUs is Larger

Way larger

Preemption disabling: 95 states
Migration disabling: 140 states

But no problem in memory
Or on flatbed plotter

As for Six Tasks Running on Three CPUs...

- A segmentation violation from graphviz when plotting diagram
- More than 1,000 states, up to 12 transitions from each state
  - Please note that this is after significant state-space reduction techniques have been applied
  - Otherwise, it would have more than 10,000 states
- I strongly recommend against attempting to carry out the Markov-model analysis while running on battery power
  - But the matrix to be inverted is only in the tens-of-megabyte range, so should be eminently doable
    - Give or take roundoff-error issues
- This presentation focuses on four tasks and two CPUs

State-Space Size for Dual-CPU Cases

# Tasks	DISABLE_MIGRATE	DISABLE_PREEMPT
1	3	3
2	9	9
3	37	30
4	140	95
5	479	278
6	1540	763
7	4787	1998
8	14624	5055
9	44287	12462

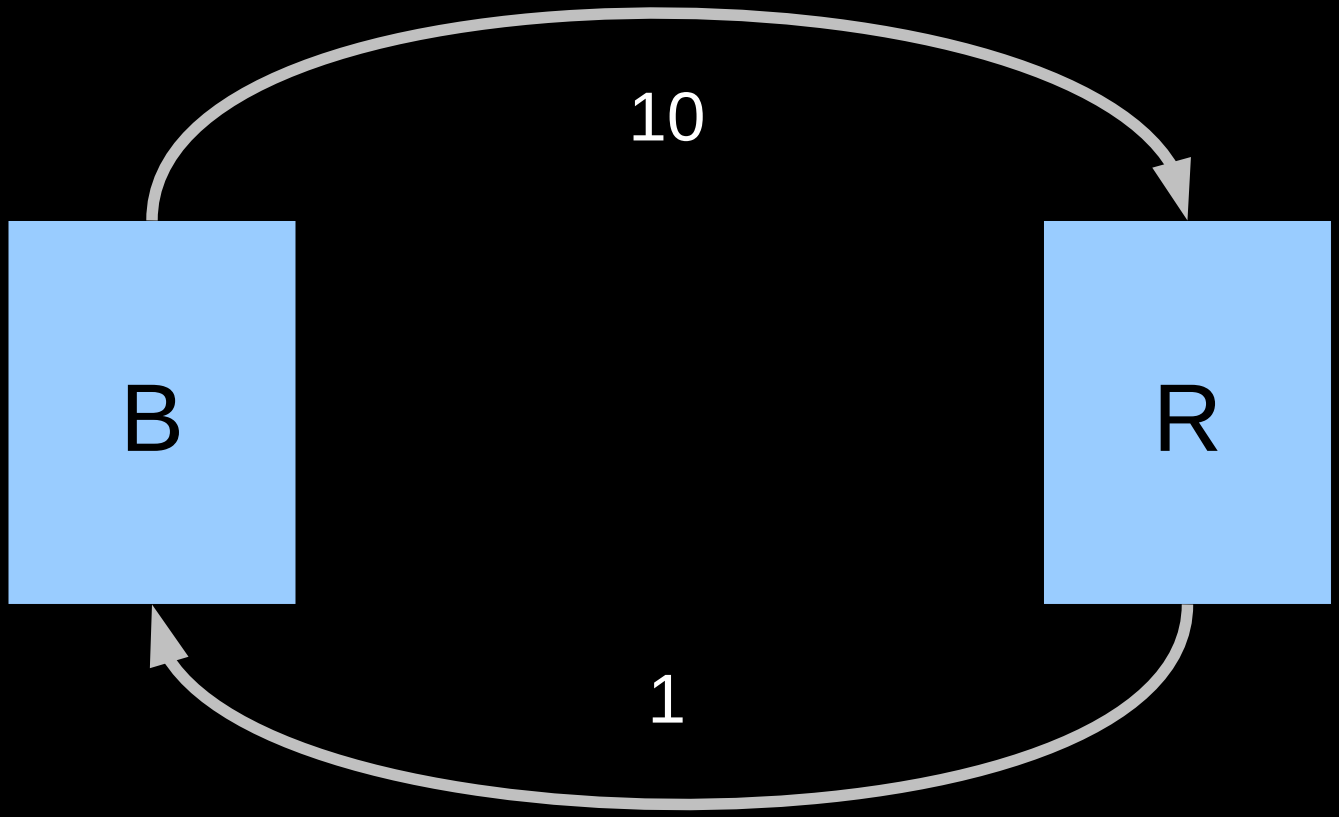
Solution is $O(N^3)$, where N is the number of states.
Plus cache and TLB behavior further degrades performance.
Problem size is also sharply limited by roundoff error.
More than 30 minutes required just to generate 44,287 states.

Description of Algorithms

Theoretical Background

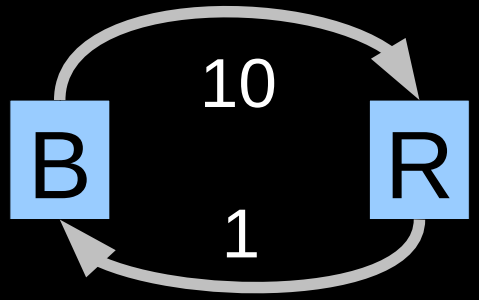
- Markov model
- States and edges, where the edges represent probabilistic transitions between states
- Exponential distribution to permit reasonable solution
  - Other distributions are possible, but how good are our measurements, anyway???
- Probability of being in a given state depends on the probability of being in states feeding into that state as well as the probabilities of the corresponding transitions

Theoretical Background: Model Representation



Roughly: 10 transitions per unit time from B to R, 1 transition per unit time back
More precisely: Interarrival times for transitions from B to R drawn from an exponential distribution with parameter 10, namely: $10e^{-10t}$

Theoretical Background: General Solution



- B(t) is probability of being in state B at time t.
- R(t) is probability of being in state R at time t.
- B(t)+R(t)=1 for all t.
- B'(t)=-10B(t)+R(t)
- R'(t)=-R(t)+10B(t) – but this is a redundant equation, ignore!
  - R(t)=1-B(t)
  - B'(t)=-10B(t)+1-B(t)=1-11B(t)
  - B'(t)+11B(t)=1
  - $e^{11t}B'(t)+11e^{11t}B(t)=e^{11t}$
  - $e^{11t}B(t)=(e^{11t}+C)/11$
  - $B(t)=(1+Ce^{-11t})/11$
  - $R(t)=(10-Ce^{-11t})/11$

  } General Solution

Theoretical Background: General Solution: Problems

- It is hard enough to get decent measurements of the interarrival rates, let alone all the initial states
- 140 states means 140 coupled differential equations with 140 unknown functions
  - The form of the differential equations is particularly simple, but...
  - The standard solution technique involves finding all roots of a 140^{th}-degree polynomial, which is just asking for trouble
    - Especially given that multiple roots need special treatment...
- Most benchmarks (err... workloads) ignore the startup transients anyway, allowing “warm-up periods”
- So we usually care only about steady-state operation
  - And during steady state, all derivatives are zero by definition!

Theoretical Background: Steady-State Solution

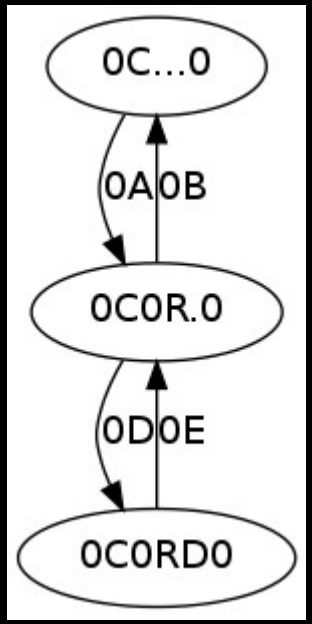
- General equations:
  - B(t)+R(t)=1
  - B'(t)=-10B(t)+R(t)
  - R'(t)=-R(t)+10B(t)
- Set all deriviatives to zero to get steady-state equations:
  - B+R=1
  - 0=-10B+R
  - 0=-R+10B – and this is still a redundant equation, ignore!
- This is a simple linear systems of equations
  - Solution is a simple matter of matrix inversion
  - Which has its own challenges, but far fewer pitfalls than systems of differential equations

Some Simplifying Assumptions

- Each task is running the same workload, so we only need to estimate four interarrival rates:
  - Wakeups per unit time spent blocked (as opposed to preempted)
  - Blocks per unit CPU time spent running while enabled
  - Disables per unit CPU time spent running while enabled
  - Enables per unit CPU time spent running while disabled
- CPUs are interchangeable
  - So task 0 running on CPU 0 is modeled as the same state as task 0 running on CPU 1
  - As long as the pattern of mappings of tasks to CPUs is isomorphic under some permutation, the relevant states are collapsed
  - Preempted tasks are always associated with CPU 0
    - Reduces state space, and doesn't hurt because we are not modeling cache

Generating the Transition Graph

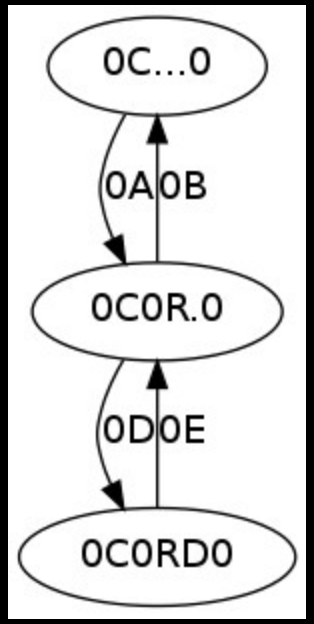
- Generate an initial state with all tasks blocked
- For each state in existence, generate all legal transitions out of that state, creating new states as needed
  - Scheduler model: find_idle_cpu(), find_lowest_prio_cpu(), find_best_task(), schedule_awakened_task(), deschedule_task(), schedule_enabled_task()
  - See for example the single-task/single-CPU case shown below.



Generating the Transition Matrix

- Create a matrix with N rows and N+1 columns, where N is the number of states
- Each state corresponds to one row of the matrix
  - One state is omitted due to redundancy, doesn't matter which
  - Each diagonal element is the negative of the sum of the interarrival rates of the transitions leaving the corresponding state
  - Each non-diagonal element of a given state's row contains the sum of the interarrival rates for all transitions from the column's state to the row's state
- All entries of final row of matrix are 1.0

Generating the Transition Matrix



-A	B	0	0
A	-B-D	E	0
1	1	1	1

Easily solved by Gaussian elimination.

Results

Results: Comparing Priority Inversion

- Note that preempt_disable() causes priority inversion:
  - Task 0 at priority 0 disables preemption on single-CPU system
  - Task 1 at priority 1 awakens, and is “born preempted” due to task 0's disabling of preemption
- Disabling migration permits preemption, but consider:
  - Task 1 at priority 1 running on CPU 0 disables migration
  - Task 2 at priority 2 awakens and runs on CPU 1
  - Task 3 at priority 3 awakens and preempts task 0 on CPU 0
  - Task 3 disables migration
  - Task 2 blocks, but neither task 1 nor task 3 can be migrated to CPU 1
  - This is a priority inversion involving the idle loop
  - Similar sequences result in more typical priority-inversion situations

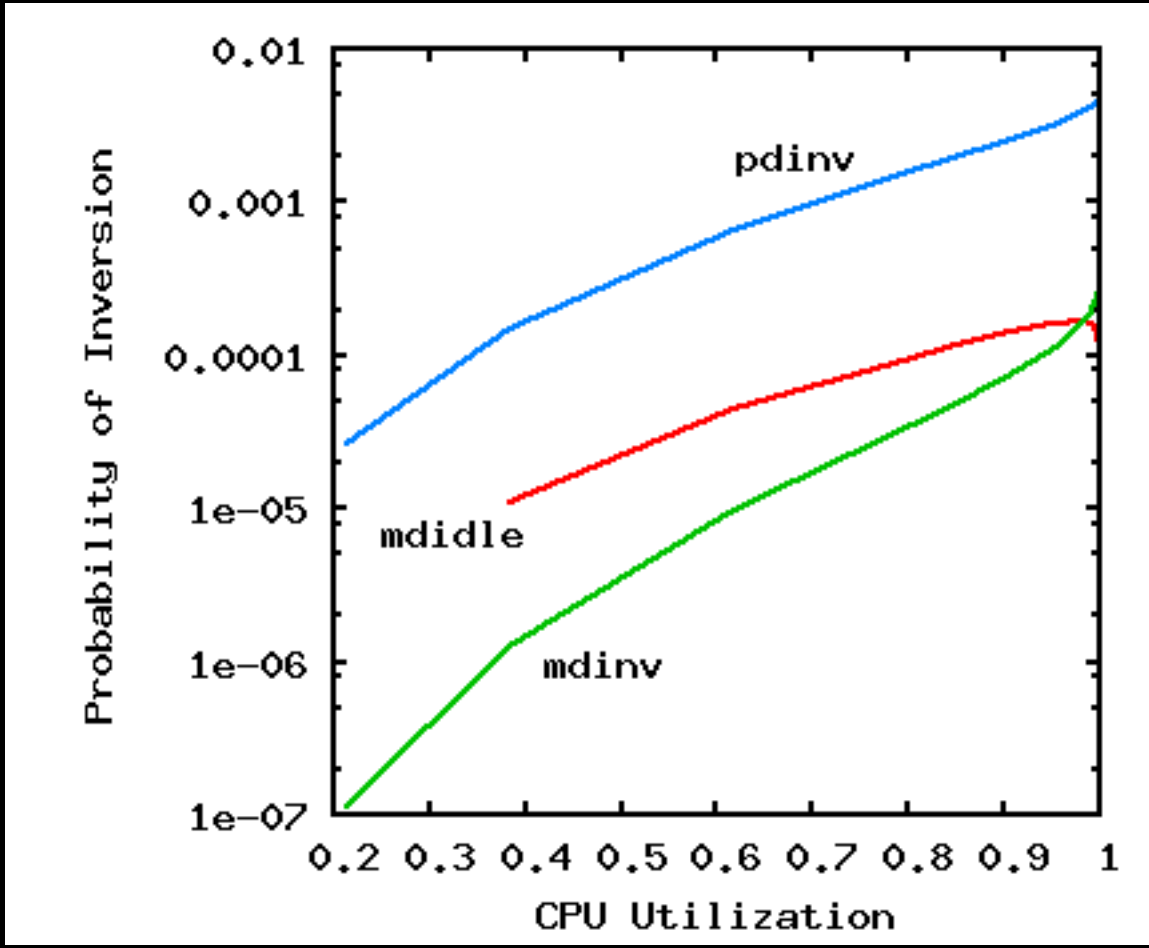
Results: Comparing Priority Inversion

- Measured quantities:
  - pdinv: preempt_disable()-induced priority inversion
  - mdinv: migration-disable-induced priority inversion
  - mdidle: migration-disable-induced priority inversion involving idle loop
- These results assume migration of high-priority tasks:
  - Task 0 at priority 0 runs on CPU 1
  - Task 1 at priority 1 disables migration on CPU 0
  - Task 2 at priority 2 awakens and runs on CPU 0, preempting task 0
  - Task 3 at priority 3 awakens and runs on CPU 1, preempting task 1
  - Task 2 blocks, allowing task 0 to run: The model assumes that task 3 will migrate to CPU 1 (again preempting task 0) in order to allow the higher-priority task 1 to run
  - (It would not be hard to modify the model to measure the effect.)

Results: Experimental Setup

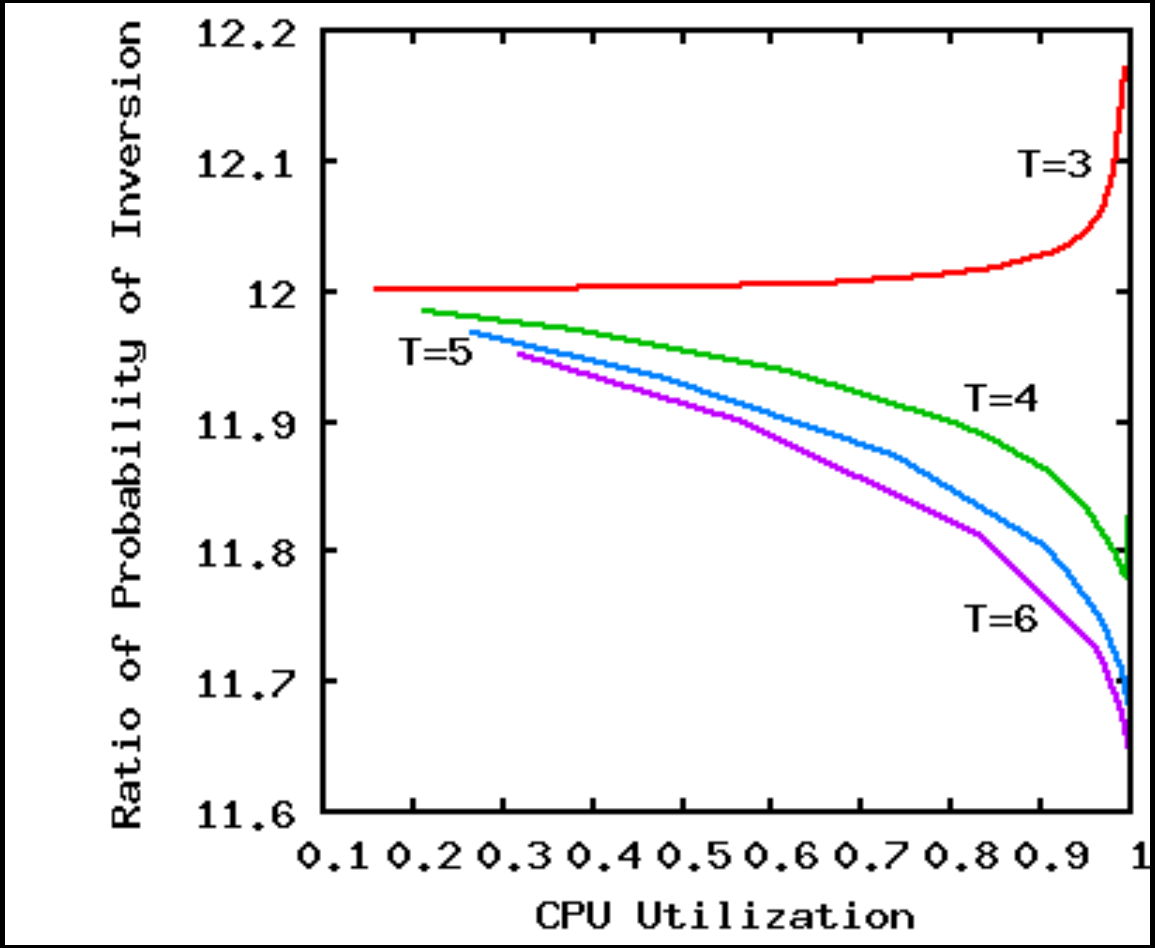
- Two CPUs
- Four tasks
- Interarrival rates:
  - Wakeup interarrival rate for sleeping task: Vary from 1 to 100
  - Blocking interarrival rate for non-disabled running task: 10
  - Disable interarrival rate for non-disabled running task: 100
  - Enable interarrival rate for disabled running task: 500
- Rationale: One wakeup per millisecond, average CPU burst duration of 100 microseconds, disable every ten microseconds, remain disabled for two microseconds
  - Vary wakeup rate in order to vary CPU utilization
  - Preliminary results: data from actual workload would be good

Results: Four Tasks Running on Two CPUs



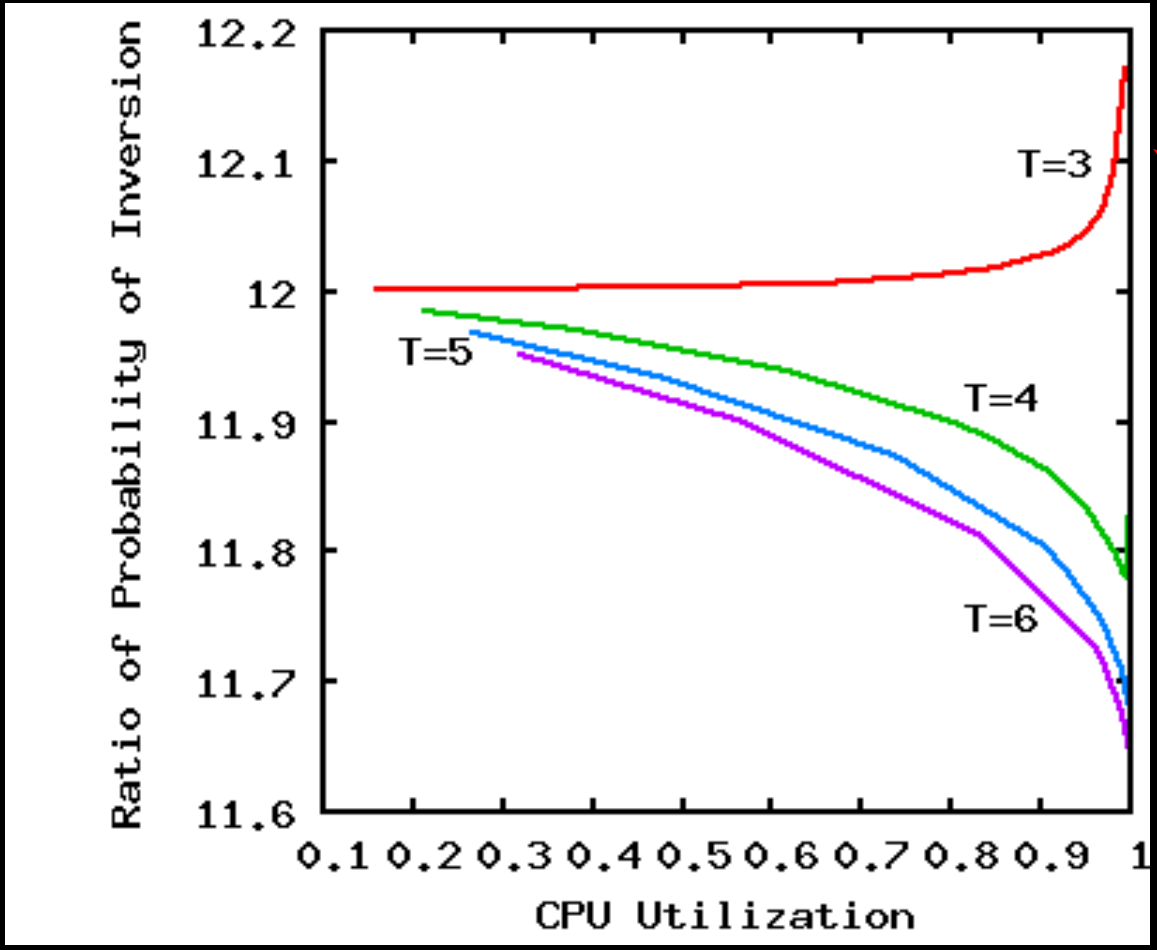
Reduces inversion, and tends to preempt low-priority tasks

Results: Three to Six Tasks Running on Two CPUs



Ratio varies little with number of tasks and CPU utilization

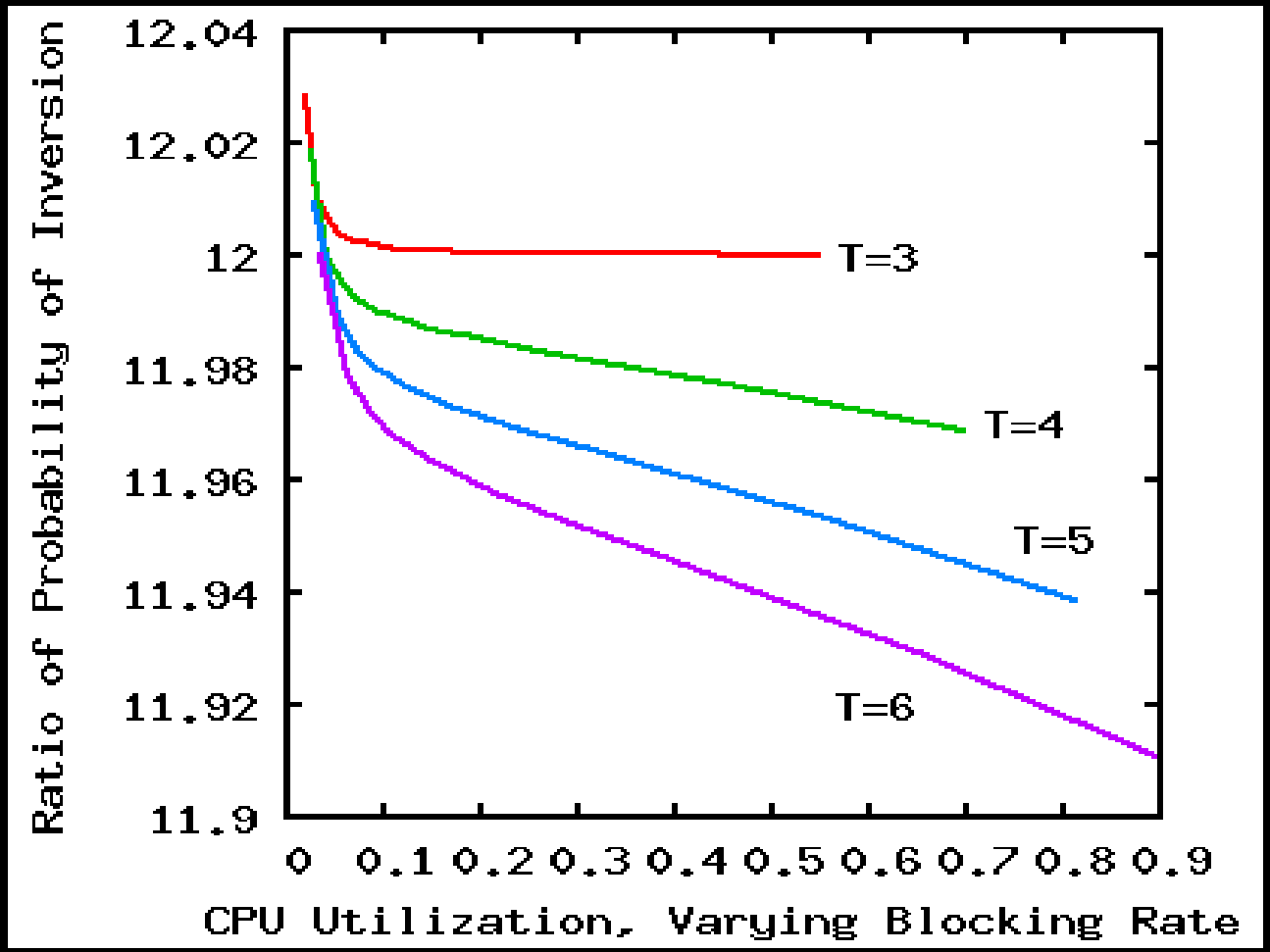
Results: Three to Six Tasks Running on Two CPUs



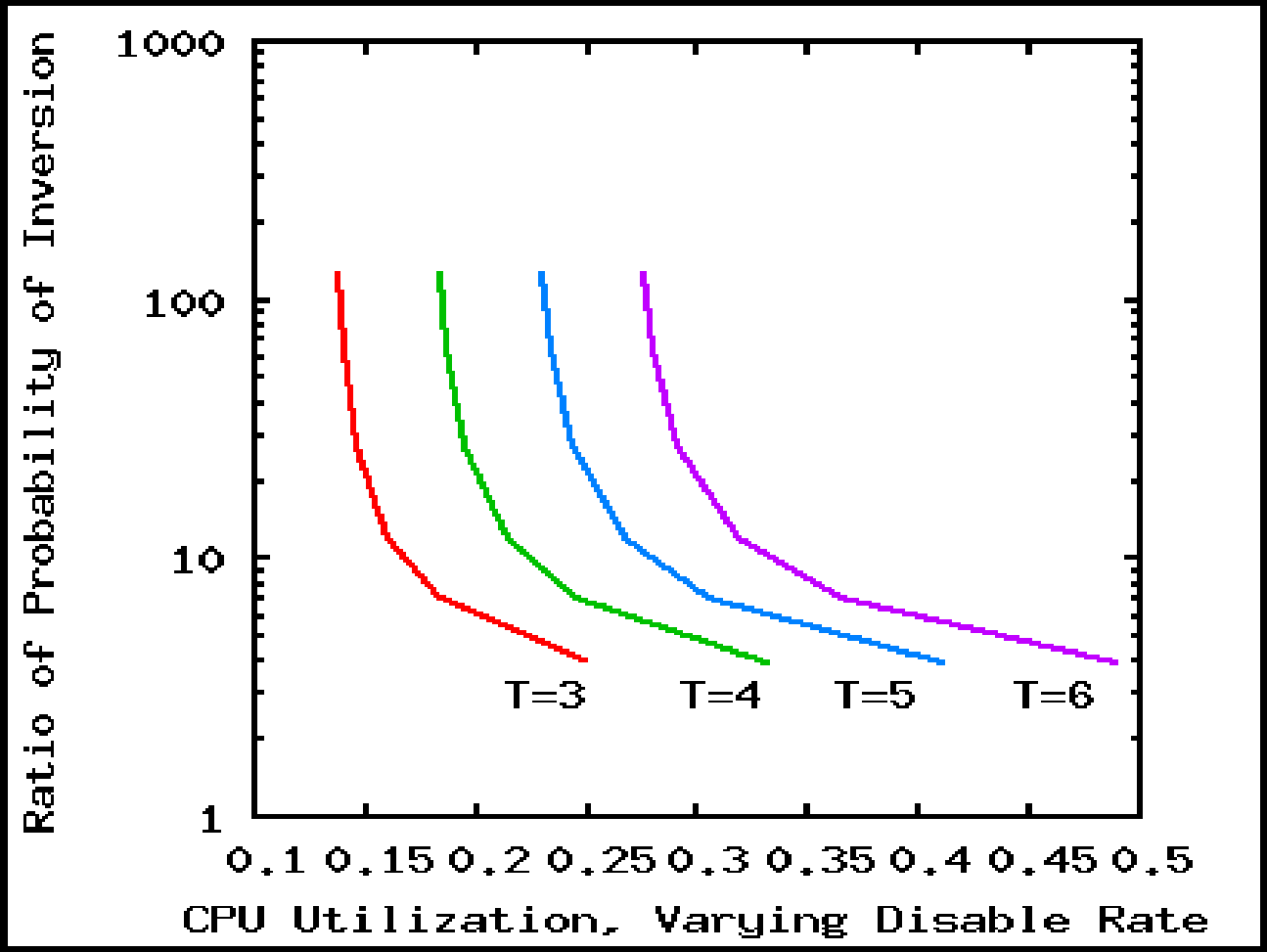
Only one more task than CPU

Ratio varies little with number of tasks and CPU utilization

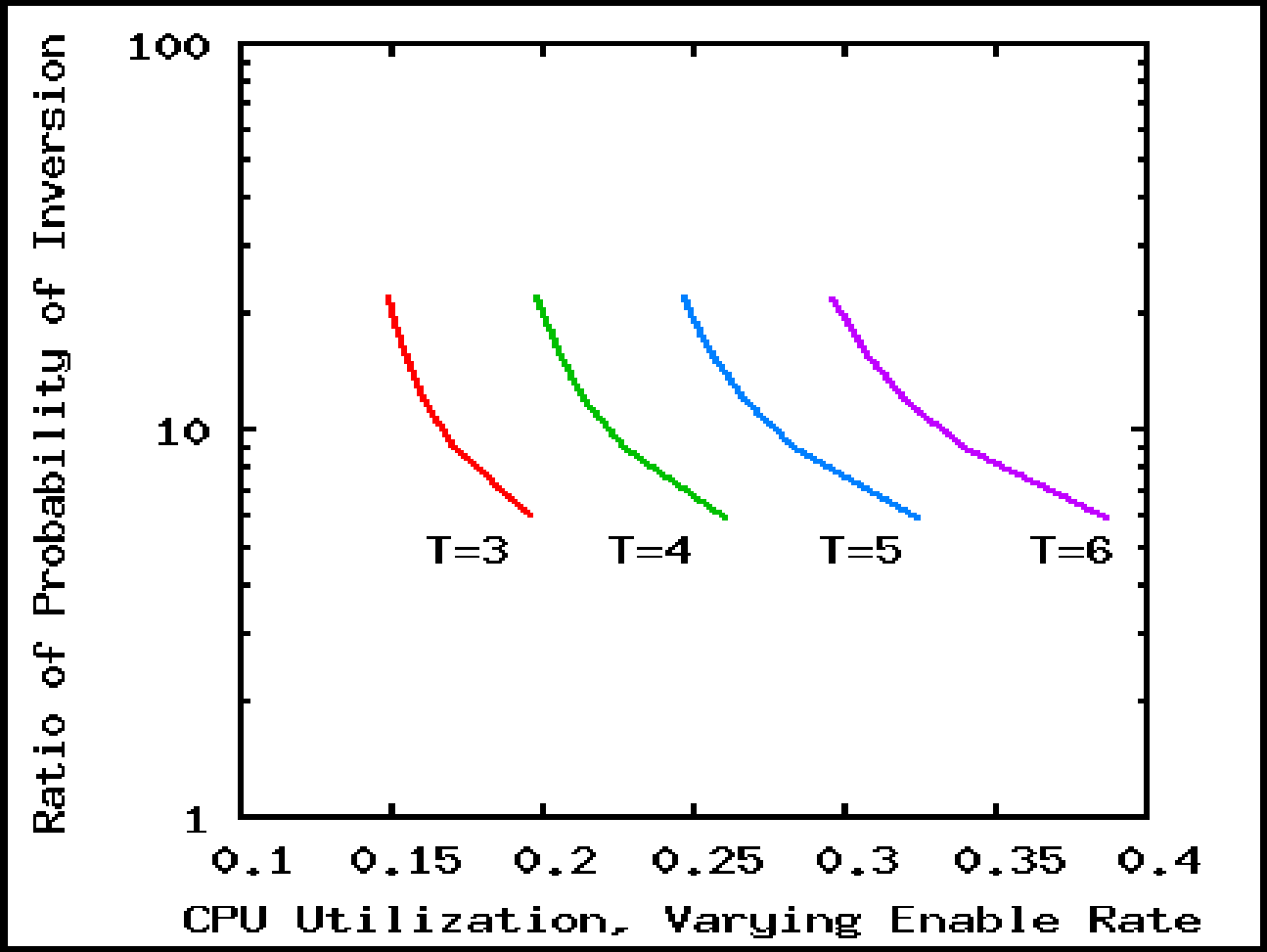
Results: Varying Blocking Rate



Results: Varying Disable Rate



Results: Varying Enable Rate



Results: Discussion

- Ratio insensitive to CPU utilization
- Main sensitivity is to fraction of time disabled
  - The greater the fraction of time disabled, the less the benefit of migrate-disable over preempt-disable
- Lesson: If you disable long enough, bad things are probable
- Disabling migration produces better results than does disabling preemption in all scenarios analyzed

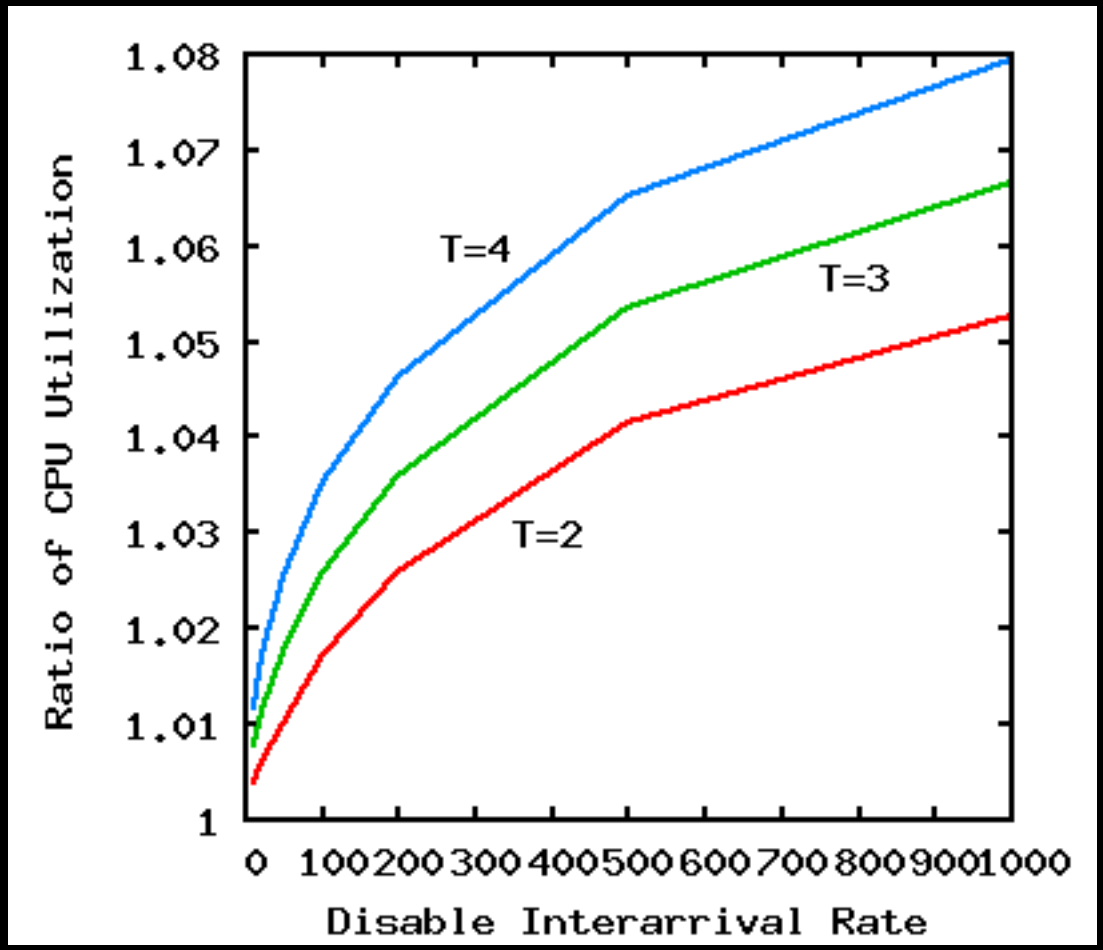
But About The Real Linux Kernel Scheduler...

- Tasks can block when disabled in -rt
  - The model described earlier in this presentation does not allow this
  - The model was updated to cover this behavior, which resulted in greatly increased probabilities of blocking low-priority tasks
- The scheduler is not omniscient
  - There are sequences of events that can leave low-priority tasks preempted in ways that are not strictly necessary
  - These situations could be avoided by migrating high-priority tasks in order to allow lower-priority migrate-disabled tasks to run
- But are such changes worthwhile?
  - Compare CPU utilizations in scheduler model to evaluate

Does It Help To Kick High-Priority Task Off of CPU?

- Consider the following sequence of events:
  - Task 0 runs on CPU 0 and disables migration
  - Task 1 runs on CPU 1
  - Task 2 preempts Task 0
  - Task 1 blocks
- Should Task 2 migrate to CPU 1 to allow Task 0 to run?
- Consider also the following:
  - Task 0 runs on CPU 0 and disables migration
  - Task 1 runs on CPU 1 and disables migration
  - Task 2 preempts Task 0
  - Task 3 preempts Task 1
  - Task 2 blocks
- Should Task 3 migrate to CPU 0 so Task 1 (instead of Task 0) may run?
- The scheduler is reported to currently do no such migrations

Does It Help To Kick High-Priority Task Off of CPU?



There is some benefit to migrating high-priority tasks to allow low-priority tasks to run

Remaining Challenges

Remaining Challenges

- Roundoff and numerical stability for larger problem sizes
  - Perhaps use a production-quality linear-system solver or indefinite precision (slow!)
  - Or various refinement techniques to “polish” a roundoff-degraded initial solution
- Results show probability, not worst-case inversion times
  - Memoryless assumption gives theoretical worst case of infinity
  - Could potentially switch to discrete time approach, but straightforward approaches either restrict residency times or blow up state space
  - But given that inversion is now hitting lower-priority tasks, throughput-based measures from current model probably what we want anyway
- Numerous tweaks to scheduling algorithm could be modeled
  - Also drive interarrival rates from real workloads
- The model does not take locking and priority boosting into account
  - Holy state-space explosion, Batman!!!
- There are almost certainly still bugs remaining in the model and code

Summary and Conclusions

Summary and Conclusions

- Disabling migration produces order-of-magnitude reductions in probability of priority inversion
  - More effective at lower probabilities of disabling
- Any number of refinements possible
- Lessons relearned
  - State-space-reduction techniques: don't leave home without them!
  - Never forget about roundoff error: I chased what I thought were bugs that turned out to be a too-large “epsilon” value
  - Bring reference material, otherwise you too may find yourself deriving steady-state solutions to Markov models somewhere over Siberia

Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.

Questions?