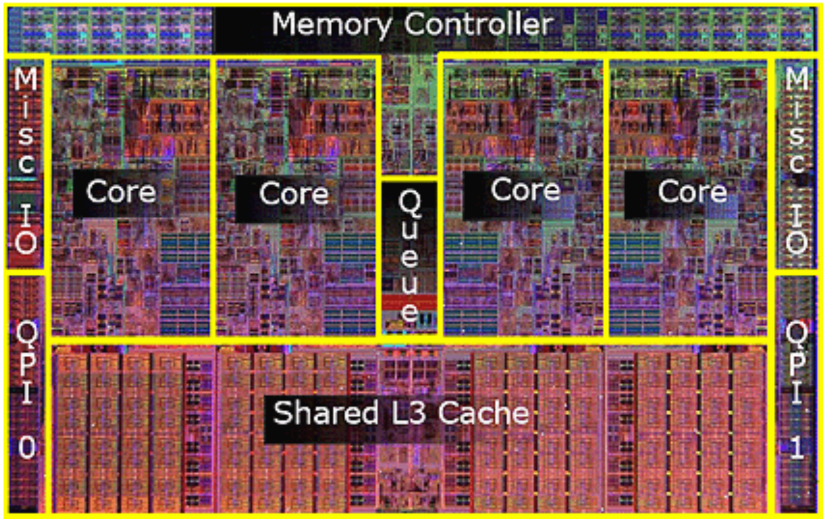
# What to make of multi-core processors for reliable real-time systems?



## Short version

- More
  - complexity, choices, variability, uncertainty
- Less
  - safe generalizations, reliability?

## Overview

- Scheduling theory foundations & results
- How is the theory affected by multiple processors?
- How valid is the theory for real machines?
- Disaster brewing?
- Survival Plans

## Why do we have multi-core processors?

- Can't make clocks faster
  - Energy usage grows with cube of speed
  - So does heat
- Can pack circuits denser

## Advantages

- More processors
- More processing power, when we need it
- Fewer preemptions
- Can switch off unused cores

## Problems

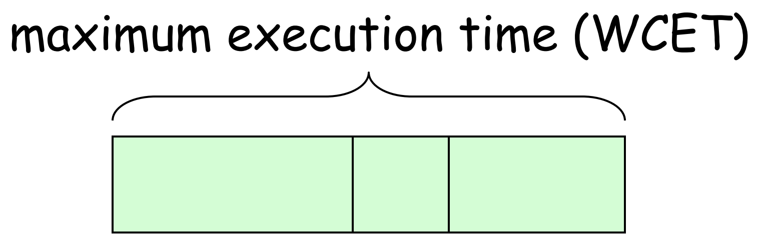
- We don’t know how to write good scalable parallel programs
- More complexity

## SMP Scheduling Theory Foundations

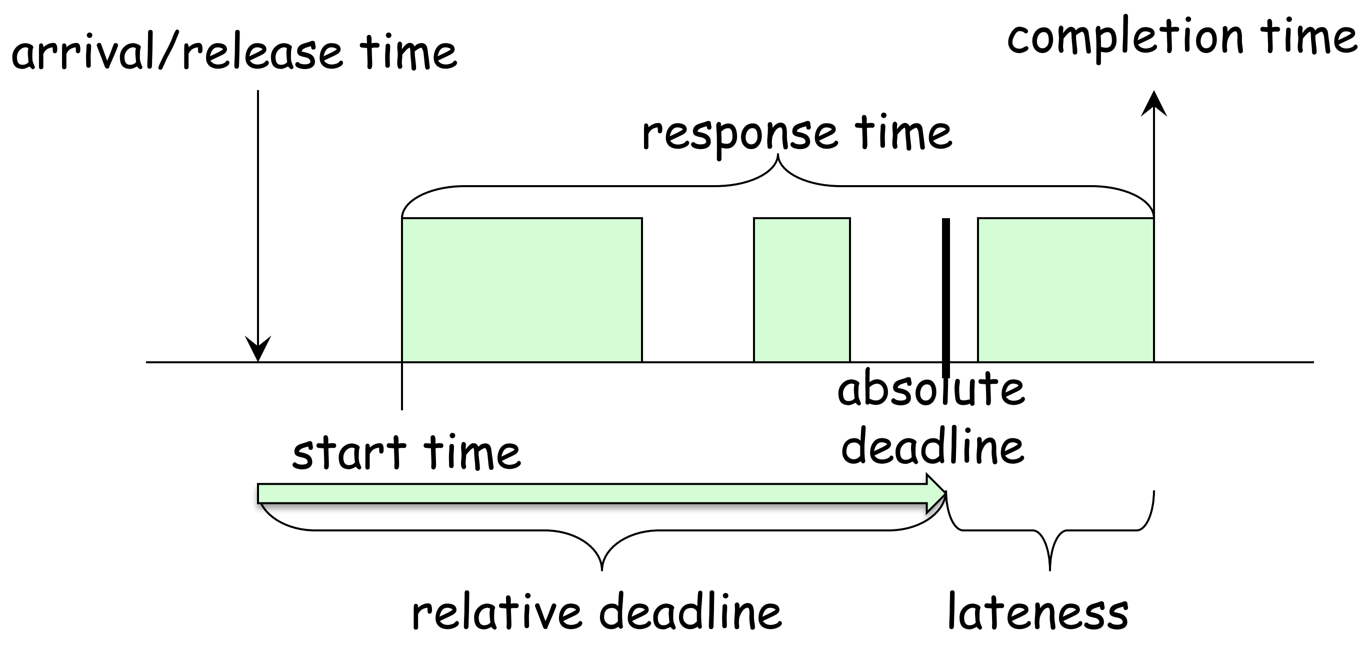
- Workload model
  - jobs
  - tasks
- Processor model

# Job

- A procedure to execute, with
- Known maximum execution time
  - assumed to be "worst case" (WCET)
  - actually, a property of scheduling algorithm
- Release/arrival time, deadline
- Possibly other attributes



# Job terms


arrival/release time
completion time
response time
absolute deadline
start time
relative deadline
lateness

# Schedule

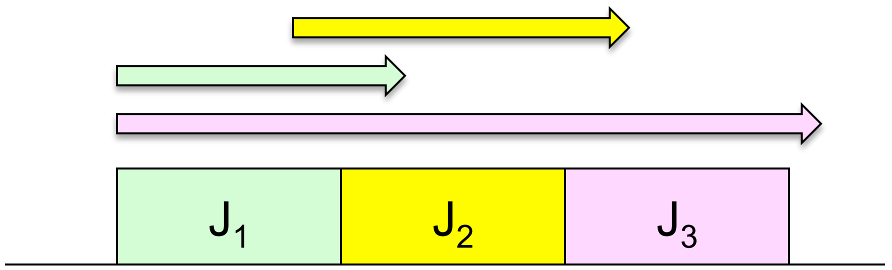
- Maps jobs to processor(s)
  - over time
- Feasible if timing constraints all satisfied
  - no early releases, no missed deadlines

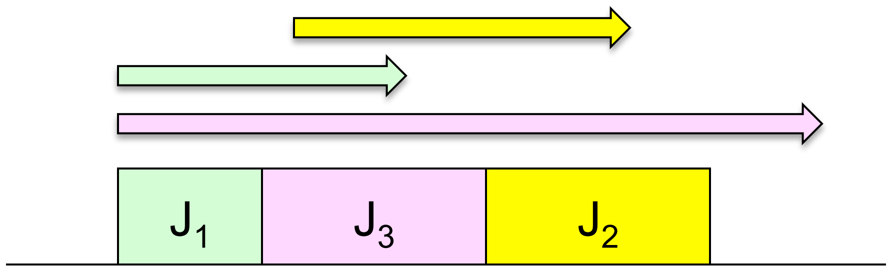
## Scheduling Algorithm

- Finds schedule for jobs
- May be static, determined off-line
- May be dynamic, determined on-line
- Reliable algorithms sustain ability to find feasible schedules under expected variations from model, especially job parameter "improvements" including
  - Shorter execution times
  - Longer relative deadlines

## Non-Preemptive Scheduling Anomaly

- Shorter execution time ⇨missed deadline
- Suppose priorities of $J_1 > J_2 > J_3$



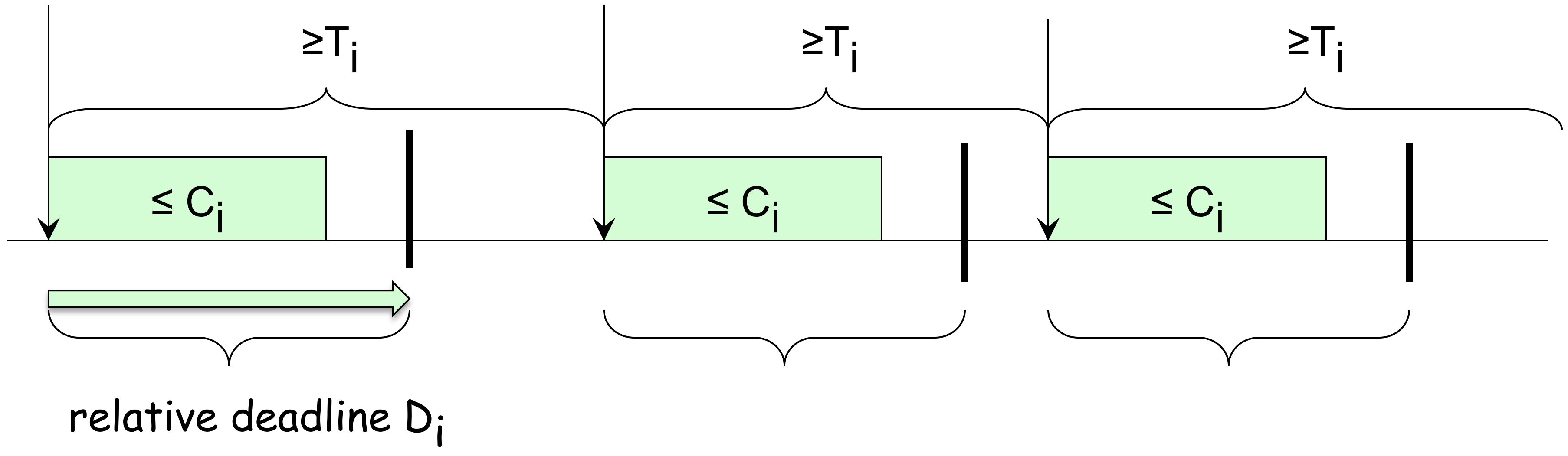


- Motivates preemptive scheduling

## Task

- Defines collection of sequences of jobs
  - Sequences generally assumed to be unbounded
- Serial execution usually required
- Various constraints on job characteristics
  - e.g., on arrivals: periodic, sporadic
- Constraints enable schedulability analysis
- Example: sporadic task system $\tau_1 \ldots \tau_n$

# Sporadic task $\tau_i = (C_i, T_i, D_i)$



Implicit deadline : $D_i = T_i$
Constrained deadline : $D_i \leq T_i$
Unconstrained : arbitrary $T_i$

utilization $u_i = C_i/T_i$
density $\delta_i = C_i/\min(T_i,D_i)$

## Schedulability Test

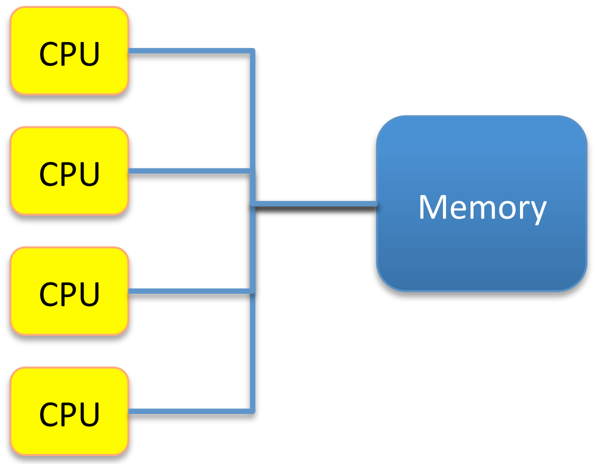
- Tells whether given algorithm will find feasible schedules
  - For job sequences of a given set of tasks
- Exact
  - "Yes" means it will succeed always
  - "No" means it will fail sometimes
- Sufficient
  - "Yes" means it will succeed always
  - "No" provides no information

## For Reliable Systems

- Scheduling algorithm is only as good as its tests

# SMP Processor Model

- All processors have same speed
  - or linearly related speeds, for "uniform" model
- Any processor can execute any task



# What makes many processors different from one?

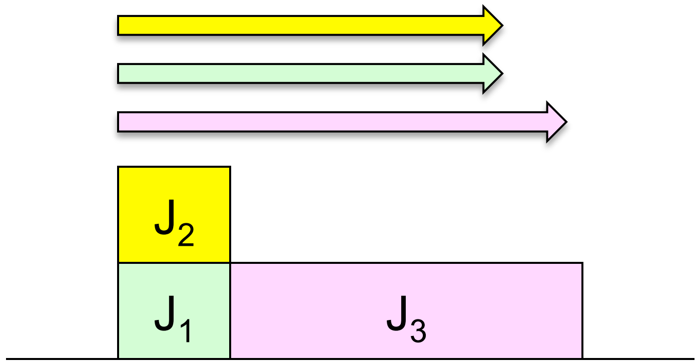
# Differences

1. Some  covered by the model
2. More not in the model, but in real multi-core systems (later)

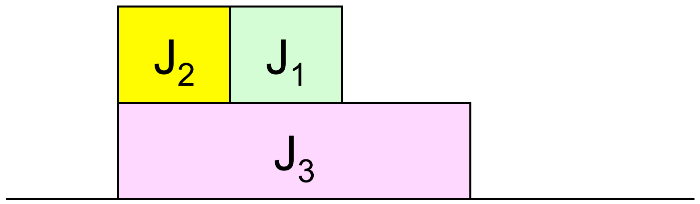
## Single-Processor Wisdom

- Priority does not affect total completion time
- EDF scheduling optimal for deadlines
- Deadline Monotonic (DM) optimal for fixed task priority
- Critical Zone:
  *Worst case response time occurs when all tasks are started together*
- Not valid for SMP scheduling

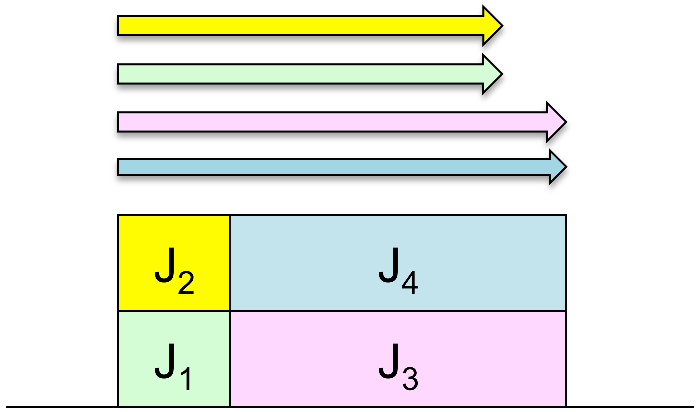
# Examples

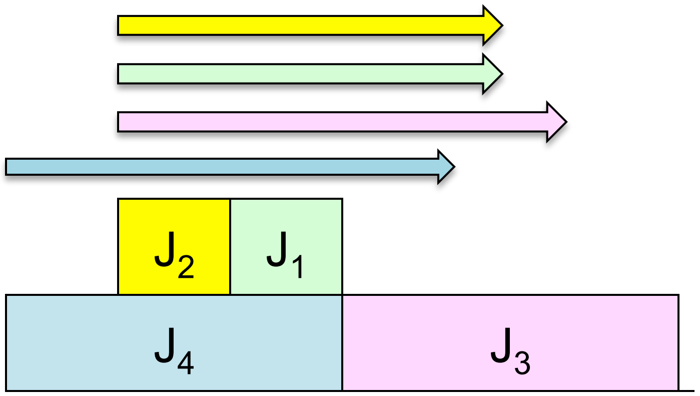


EDF, DM



Optimal



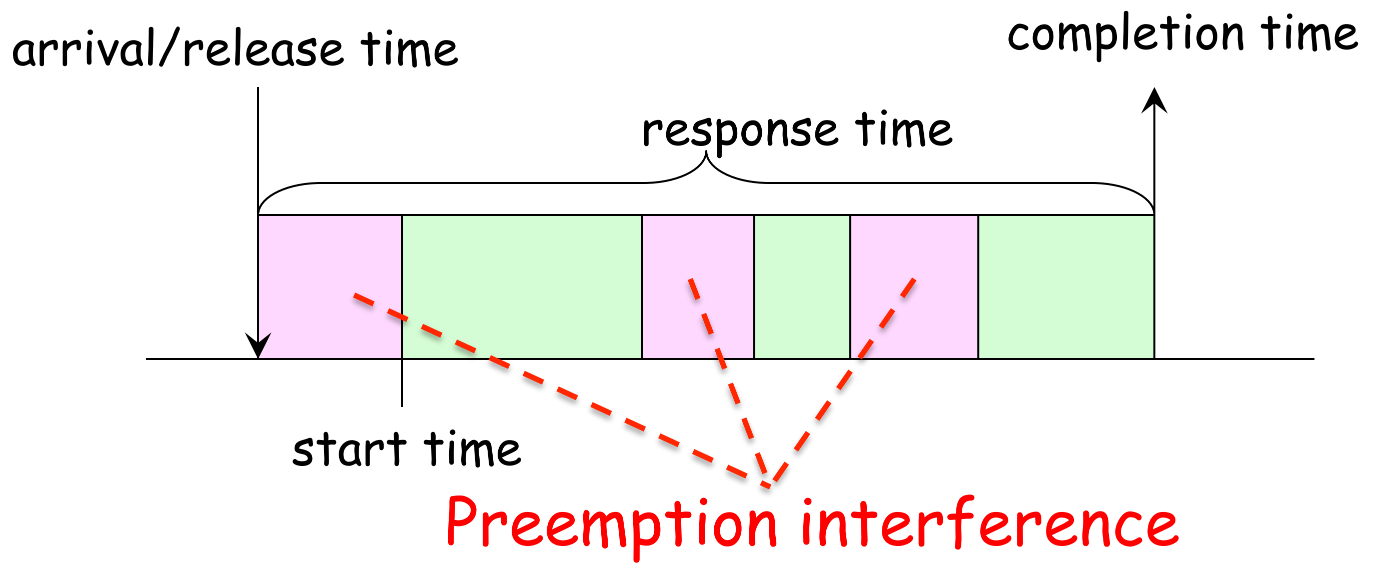


Simultaneous Releases
Not Critical Zone

# Demand Analysis

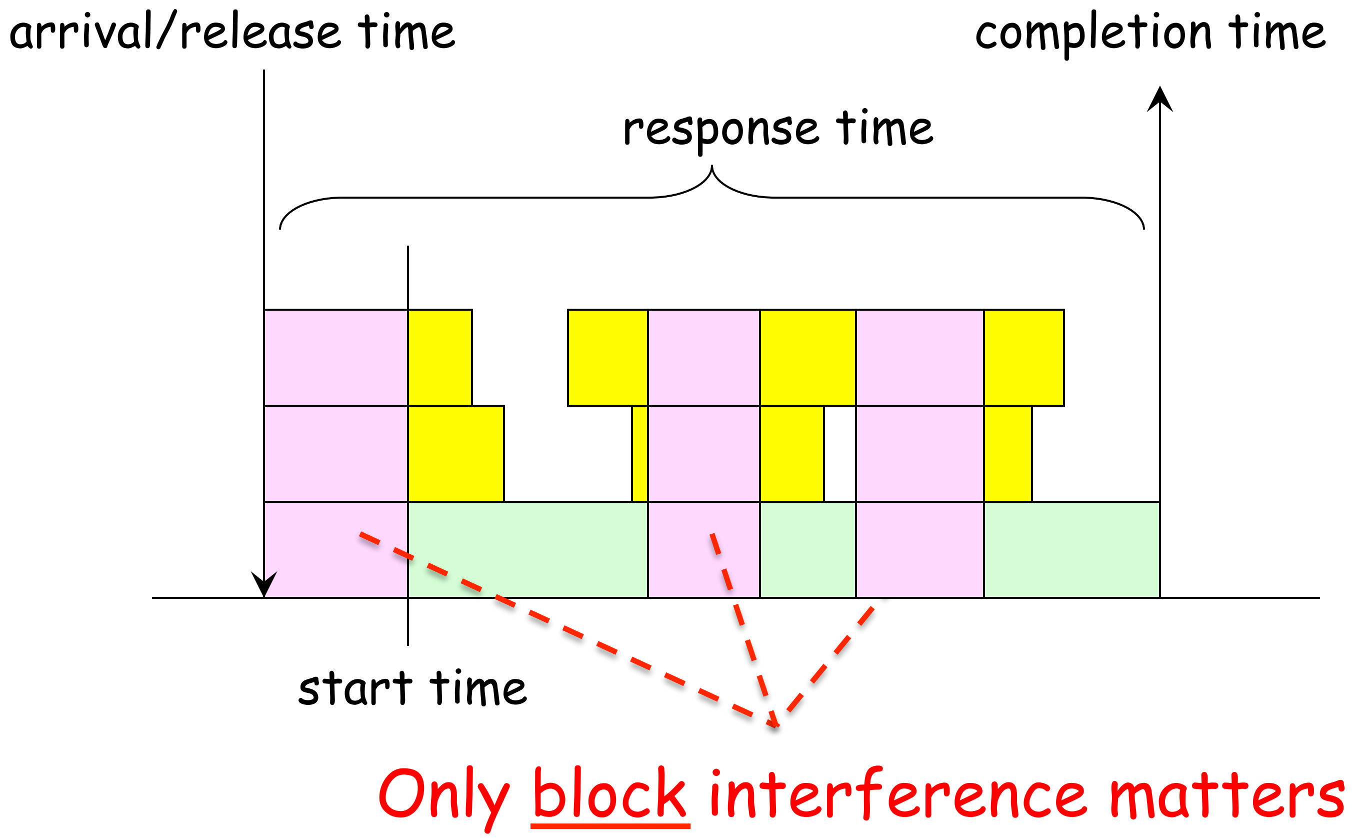
- Bound max <u>demand</u> for processor time of job in its scheduling window
- Bound maximum <u>interference</u> in window
  - Time that the job cannot run
  - Caused by other jobs not yet completed
  - Preemption, blocking, etc.
- Supply = window size - interference
- If supply > demand, job will complete within its deadline

# Single-Processor Preemption



- "Work conserving" (no idle when job waiting)
  ⇨ no idle time

# Multi-Processor Preemption



## Changes from Single Processor

- Two dimensions (processors, time)
- "Work-conserving" is not enough
  - Optimal schedule for near deadlines can create idle time, increasing backlog
  - Causing avoidable future delays
  - In the long run, keeping all processors busy wins
- Locks have larger impact
  - Can force idling of processors
  - Priority ceiling does not prevent deadlock

## Migration Costs

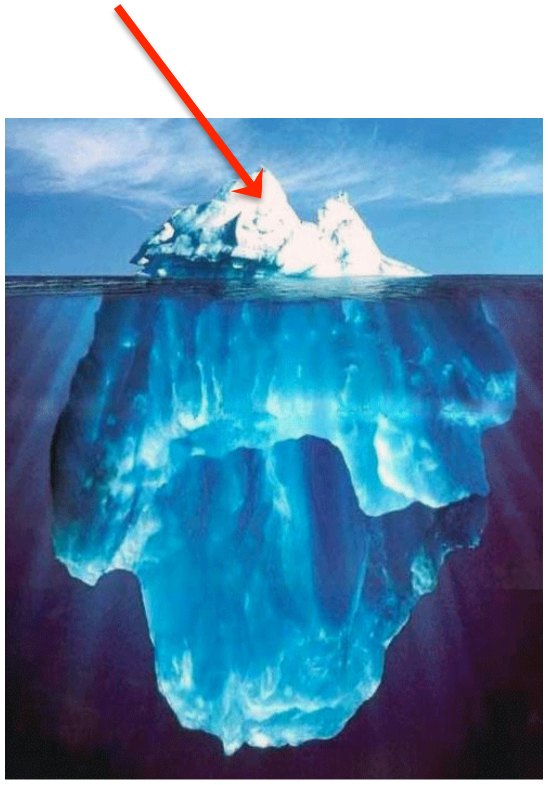
- Additional cost of resuming preempted job, or next of job of task, on a different processor
  - Communication delay
  - Evicting preempted task/job (if any)
  - Loading cache on new processor
- Can be avoided by partitioned scheduling
- Often cited as problem with global scheduling, probably exaggerated

## Migration Costs

- Can be modeled by adding constant to WCET of each job, like preemption costs
- With shared cache, may not cost more than preemption

## Migration Costs

- Highly architecture-dependent
  - How much cache is shared?
  - How fast is interconnect?
- Highly context-dependent
  - What has transpired in old processor's cache since preemption or last job execution?



This is just one among several more serious architecture-dependent sources of execution-time variation.

## SMP Scheduling Theory Results

## Sample Results: Static Scheduling

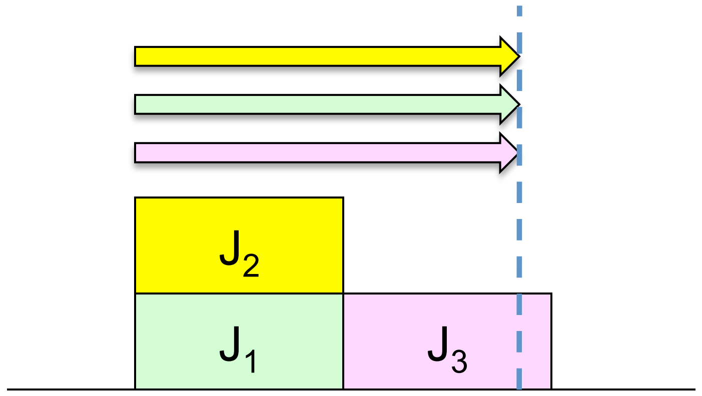
- Optimal scheduling NP-hard
- A form of bin packing problem
- Optimal is not necessary
- Greedy heuristics within 2x optimal, in worst case, very good on average
- Can handle complex constraints
  - precedence, task interdependences
  - additional resources

## Extending EDF & DM to SMP

- Partitioned
  - Assign tasks to processors statically
  - Schedule tasks on each processor dynamically
    - Fewer combinations of interference effects
    - Allows cheaper local locks
- Global
  - Assign tasks to processors dynamically
    - Less idle time, better average throughput

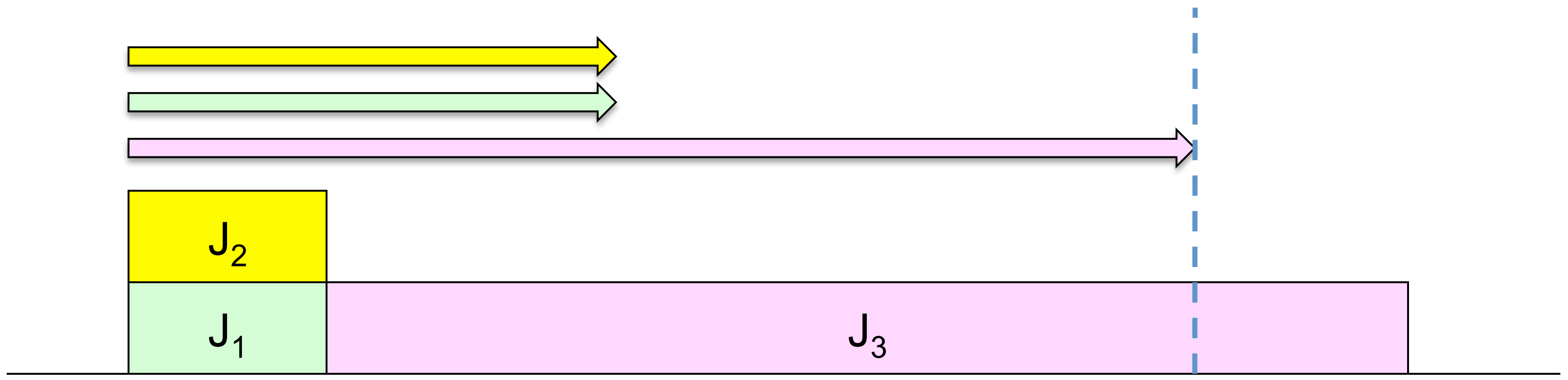
## Some Results: Partitioned EDF & RM

- Optimal partitioning NP-hard
  - Still bin-packing variants
  - Optimal is not necessary
  - Greedy heuristics good
- Worst-case utilization bound = 50%



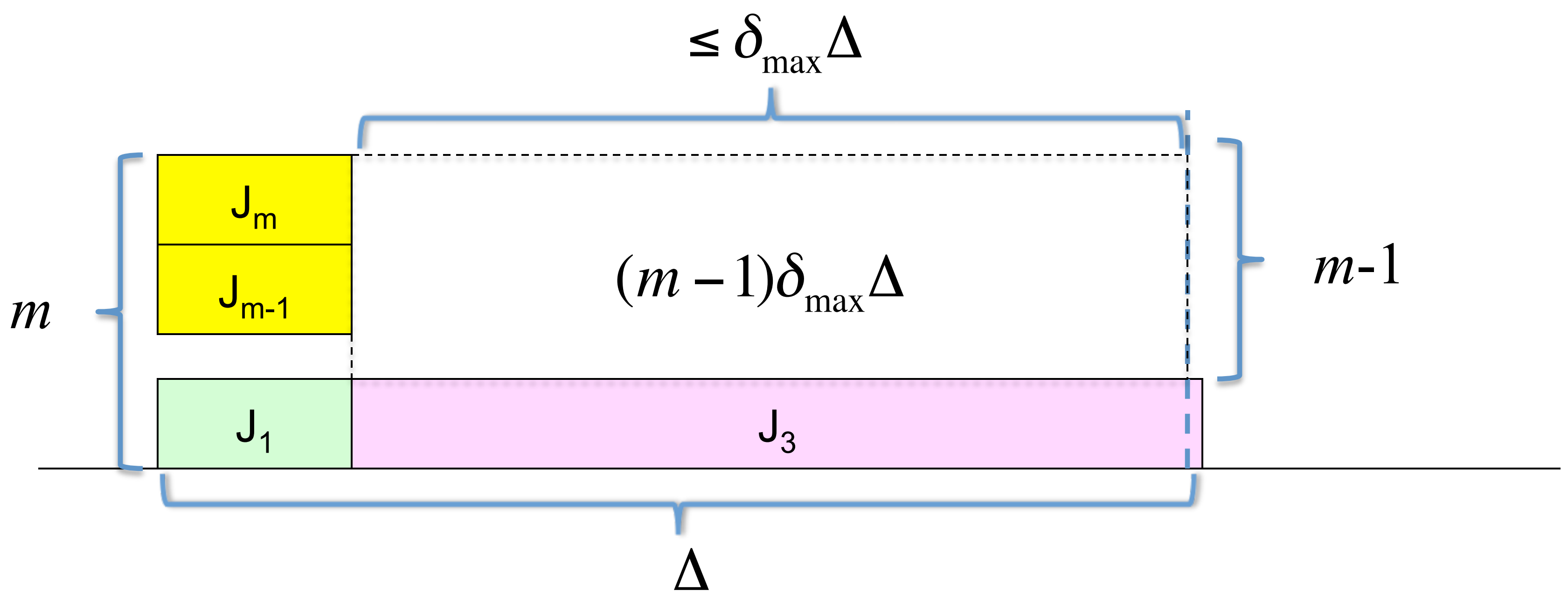
## Some Results: Global EDF & RM

- "Heavy" tasks cause problems
- Worst-case utilization bound = $1/\delta_{max}$



## Density Bounds

- Sufficient schedulability conditions
- For EDF: $$\sum_{i=1}^{n} \delta_i \leq m - (m-1)\delta_{\max}$$



## Sufficient Tests for Global Scheduling

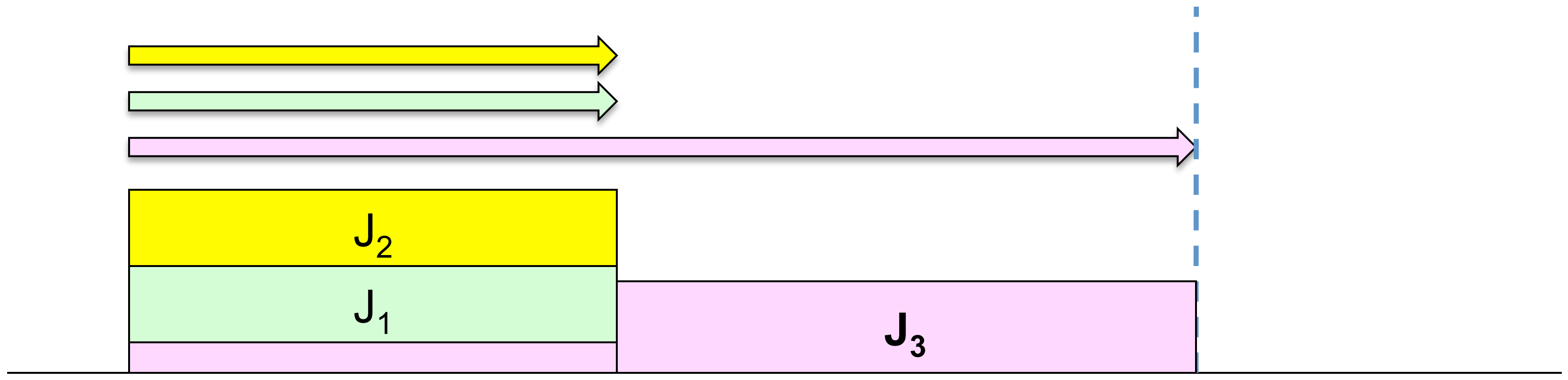
- Density bounds simplest, and most conservative
- Capture the significance of "heavy" vs. "light" tasks
- There are about a dozen tests that are less conservative (more accurate)

# Global Hybrids

- Assign top priority to tasks with $\delta_i$ > cut-off
- Apply global EDF or DM to the rest
- Intuition
  - longer job = more opportunity for parallelism
  - and more need to start early
- Can achieve higher worst-case utilization bound
- For EDF with 50% cut-off: $\frac{m+1}{2}$

## Processor Sharing (Pfair)

- Approximates "fluid" scheduling
- Utilization bound 100% for implicit-deadline periodic tasks
- Limited by time-slicing overhead



# Many Algorithms, Tests

- e.g, EDZL, task-splitting
- Growing set of sufficient tests
  - not simply comparable
  - difficult to choose one that is best
  - all quite usable
- *See paper for more detail*

## Extensions

- Aperiodic servers
- Locking protocols

## Aperiodic Servers

- Not much published
- Generalizations to SMP seem fairly simple
- Group budgets seem to be a problem

# Locking Protocols

- Not yet well understood
  - Results not as satisfying as for single processor
- Some spinning necessary for general solutions
  - When is blocking lock worth the overhead?
- One size does not fit all

## Locking Protocols

- Impact of global locks grows with number of processors
- Periodicity and parallel decomposition can increase contention
- A weakness of global scheduling
  - Partitioning allows optimization of local locks
  - Static scheduling can eliminate locks

## Lock-Free Methods

- Circular buffers, Read-Copy-Update, Atomic Queues, Software Transactional Memory
- Again, one size does not fit all

## How solid is the foundation?

## Dangerous Assumption

- That actual workloads and processors fit the models

# Dataflow Blocking

- Execution of one task must wait for results of computation by another task
- Not a big problem for single processor system
  - consumer must wait anyway, since CPU is shared
- Results in idle processors in SMP system
- Tasks are not independent

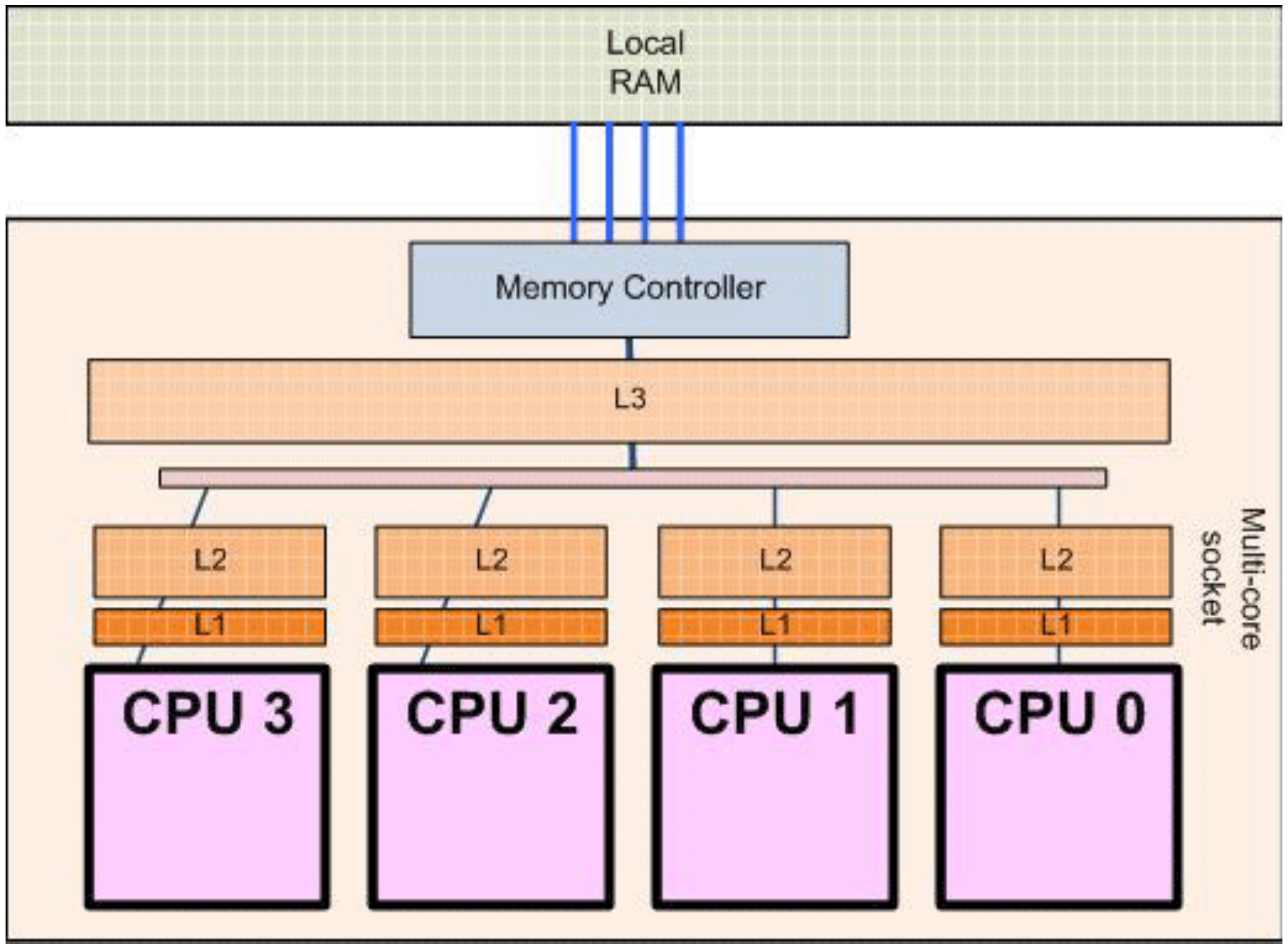
## WCET Myth

- Already a growing problem for single processors
- Cache & DMA I/O effects problematic

## Execution Time Dependences

- Effects of other concurrent tasks
- Differences between "identical" processors
  - Heat protection mechanisms
  - Bus priorities
- System elements other than the processors
  - memory hierarchy
  - component interconnects
  - I/O devices

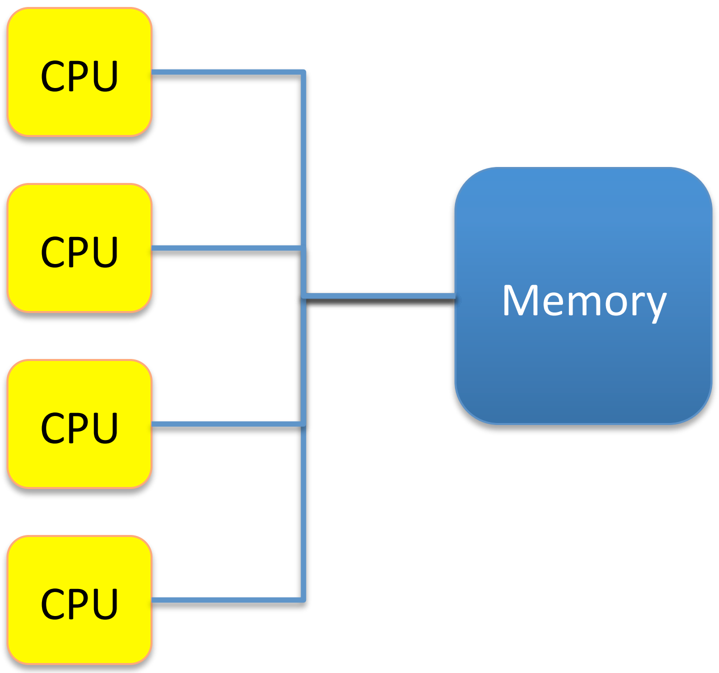
## A simple example

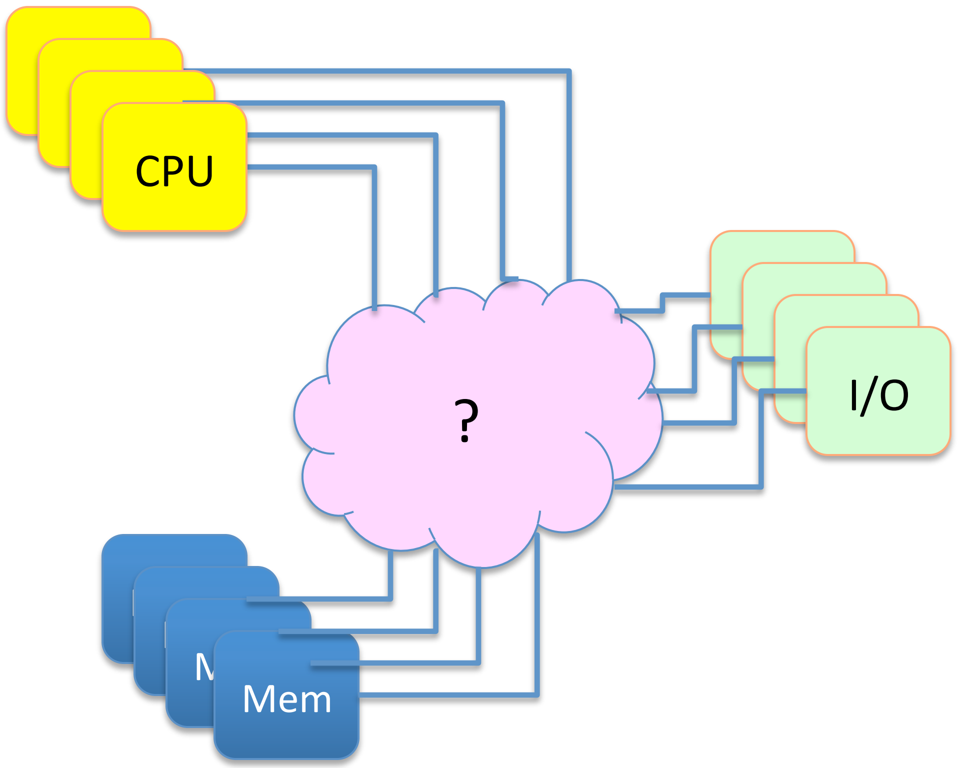


## Task Interference

- New ways for tasks to interfere
  - shared cache eviction conflicts – all the time, not just at context switches
  - cache snooping delays
  - cache and memory access path (bus) conflicts
  - dataflow blocking (see next slide)
- Interference is dynamic, hard to model
  - Reports of execution time variations up to 100%

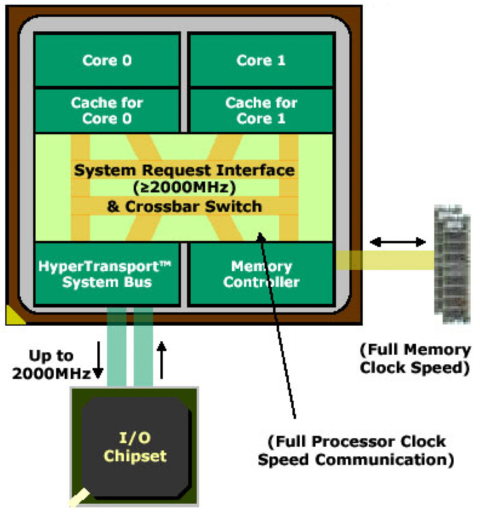
# Myth Grows Worse with Multicore

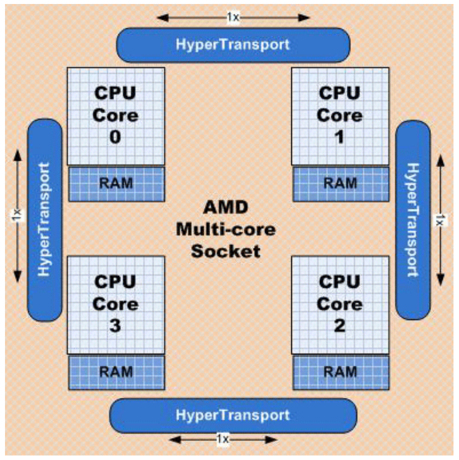


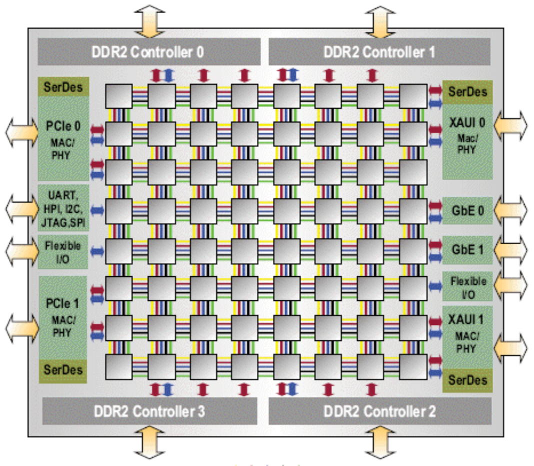


# Many Variations

- Designs continue to evolve
- There is no single common architecture


Core 0
Core 1
Cache for Core 0
Cache for Core 1
System Request Interface (≥2000MHz) & Crossbar Switch
HyperTransport™ System Bus
Memory Controller
Up to 2000MHz
(Full Memory Clock Speed)
I/O Chipset
(Full Processor Clock Speed Communication)





## Processor Dependence

- Different processors in same chip may have different internal cache access priorities
  - see reports of variations up to 400%
- Seems certain to become more variable with larger numbers of cores

## Apparent Trends

- More cores
- Serial connections between modules on chip
- On-chip networks: grid, ring, etc.
- Packetized routing of data
- Less cache coherency
- More opportunity for tasks to interfere
- More variation in execution time

## We have seen these before, in HPC

- In larger scale (not on one chip)
- "Supercomputers"
- Architectural dependence of code
- Lack of standard architecture
- Exploiting potential of HW took lots of programming time
- Results were not portable
- SW development could take longer than time to next HW generation

## Network Dependence

- Can no longer ignore data paths between cores, caches, memory
- These may be the real bottlenecks
- Delays depend on dynamic interactions
- Nominal (single core) WCET becomes irrelevant

## System-on-a-Chip

- Chip needs to be viewed as “distributed”
- Routing algorithms and message transmission delays need to be taken into account
- But … *(next slide)*

## Network analysis?

- SW cannot control on-chip routing
- Can it be modeled?
  - Is enough information available?
  - Is it portable?
  - Does it change between instances of the same part number?
- Granularity of transactions and micro-task complexity seem to make detailed analysis impracticable

## Doom?

## Scenario

- Complex architectures

+ Extreme execution time variability

+ Need to consider entire processor network

+ Dynamic dependence on core interactions

+ Lack of documentation

+ Lack of standard architectural models

= No meaningful WCET bounds

⇒ End of “hard” real time analysis?

## Survival

## Need to Cope with

- Inter-task data flow delays
- Inter-processor data flow bottlenecks
- Wide variation in execution times
- Variations in architecture
  - Cache, intra/inter-chip data paths
  - Poorly documented, hard to model

## Manage Data Flows

- Avoid global data wherever possible
- Divide work into units with explicit input/ output parameters
- Use data flow design constructs, e.g.
  - Pipelines
  - Work queues

## Allow for Execution Time Variation

- Design to avoid hard deadlines
- For unavoidable cases
  - Reserve resources
  - Overprovision
  - Apply static scheduling
- Focus on throughput
  - Apply HPC techniques

## Contain Architectural Variation

- Separate concurrency design from functional design
- Look for abstractions that can hide optimizations to fit hardware
  - e.g., cache line size, sharing, coherency
- Break free of thread model
  - Adopt message/event-handler model

## What to do in Ada?

- Reduce casual memory sharing
- Design to run on variable number of cores, without recoding
- Reduce focus on tasks as semantic units
- Move toward event-driven model
  - Example: work queues & servers
- Consider optimizable standard packages
  - Example: Atlas linear algebra library
- Apply distributed systems annex?

## Problems with Tasks/Threads

- Implicitly share access to global data
  - encourages undisciplined sharing
  - hides data flow within internal task logic
- Mix concerns that should be separable
  - semantics vs. performance
- Limit concurrency, ability to use more cores
  - hard coded
- Limit fine-grained concurrency
  - single thread of control, heavy weight

## Problems with Protected Objects

- Implicitly share access to global data
  - same as with tasks
- Overly general & overly complex semantics
  - limit cache-friendly optimization

## Summary

- More complexity, choices, variability, uncertainty
- Less safe generalizations, reliability?

# THE END

Debate?

## Appendices

- The real story of the tortoise & the hare
- An event-driven design example

## Who really won the race?

- Each was ahead at times
- Positions reversed, several times
- This seems to be true of technological choices, also

## Re*versals

- CPU vs. memory as bottleneck
- Global vs. partitioned scheduling superiority
- Static vs. dynamic scheduling
- Hashed vs. sequential access to data
- Interrupts vs. polling for I/O
- etc.

## Work Queues & Servers

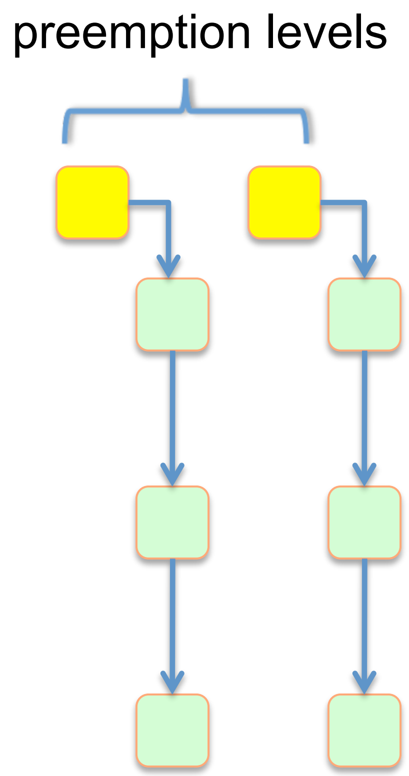
- An illustrative example, **not a panacea**
  - In particular, cannot handle "joins" of work flows

## Work Queues*: Goals

- Fit collection of servers to available processors, transparent to program logic
- Make data flows visible enough to manage & analyze
- Provide deadline service with fixed task priorities
- Do without new language features

** As described in my 1993 Washington Ada Symposium tutorial*

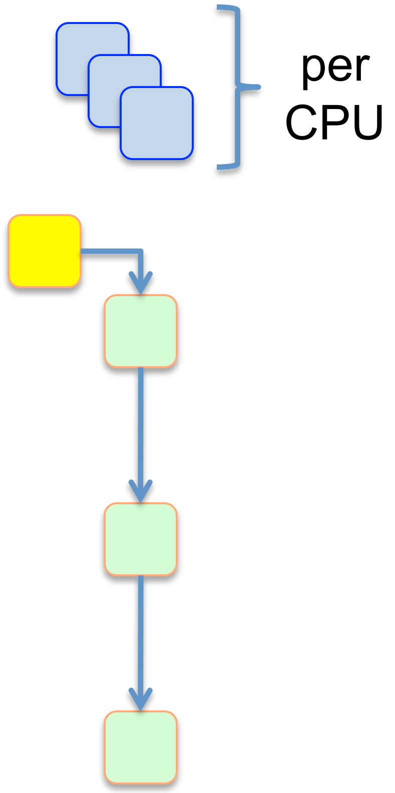
# Work Queue



- A list of work items
- Has associated priority or preemption level
- Has associated set of processors
- One server task per processor
- Has a specified queuing discipline
  - *e.g. FIFO or deadline*
- Data flows between queues

# Server

- A general-purpose task
- Serves a single work queue
- Has a fixed priority
  - to match its queue
- Is assigned to a specific CPU
- Suspends while queue is empty
- Executes the service methods of items in the queue



## Work Item

- Derived from base work_item class
- Has associated service method
- Visibility limited to explicit parameters
  - Inputs: constant components, or access-constant components
  - Outputs: copied to another item, or updated via access-variable components
- Preemption level matches queue