7.a Multicore systems: initial reckoning

Where we enter into the world of multicore processors and see that everything has changed. To make sense of it, we first look inside the processor and see what has happened there (and still is), and then begin to reflect on what the scheduling problem becomes when parallelism enters the picture

A reconnaissance taxonomy /1

- Distributed systems are *loosely coupled*
  - They do *not* share memory: capturing the global status of computation is exceedingly costly
  - Scheduling decisions are strictly per-processor
- Multiprocessors (nowadays multi-core) are *tightly coupled*
  - They share memory: capturing global status and workload information on all CPUs is cheap and straightforward
  - They bring *application-level parallelism* to the fore
  - Their architecture enables several variants of scheduling
- Modern multiprocessors are either *homogeneous* (aka symmetric) or *heterogeneous*
  - The former make for a much simpler problem
  - The latter are the new normality and a harder problem

The origin of multicore processors

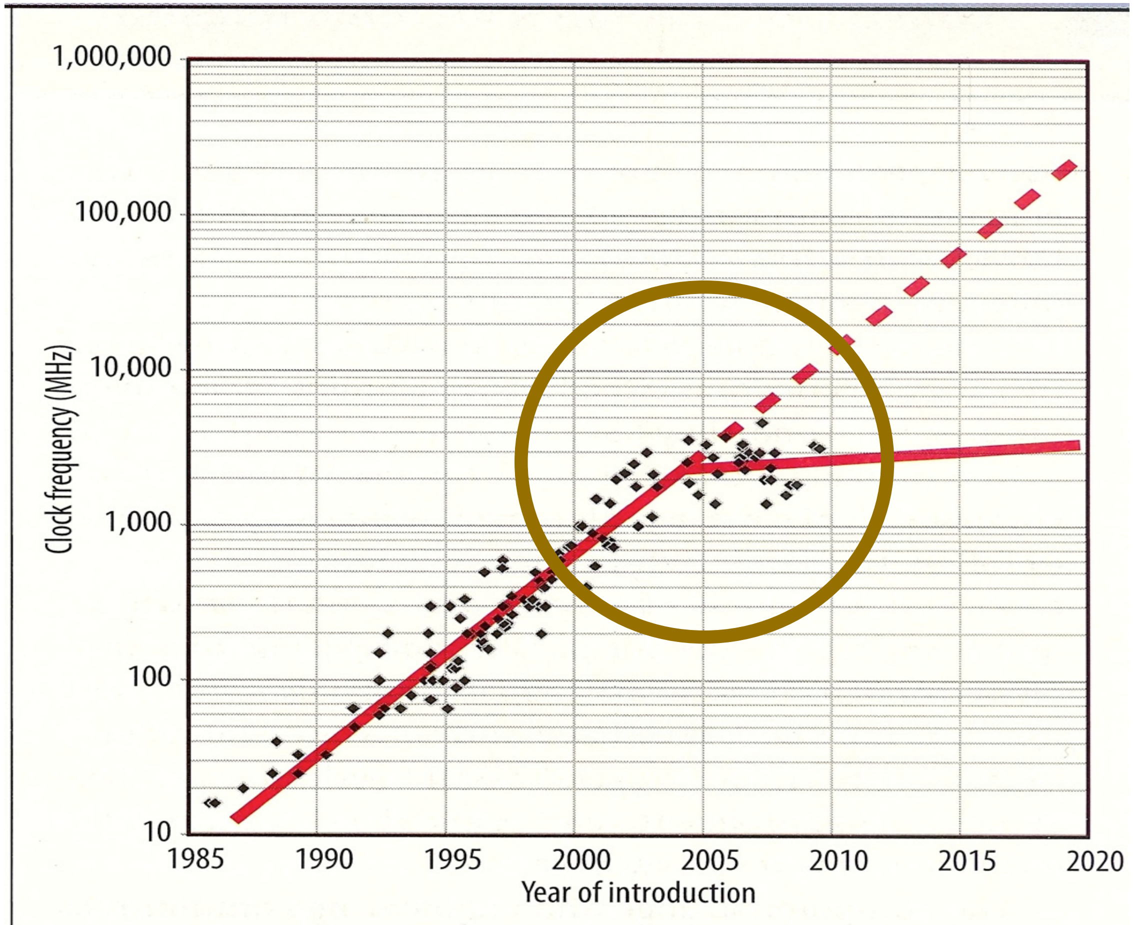
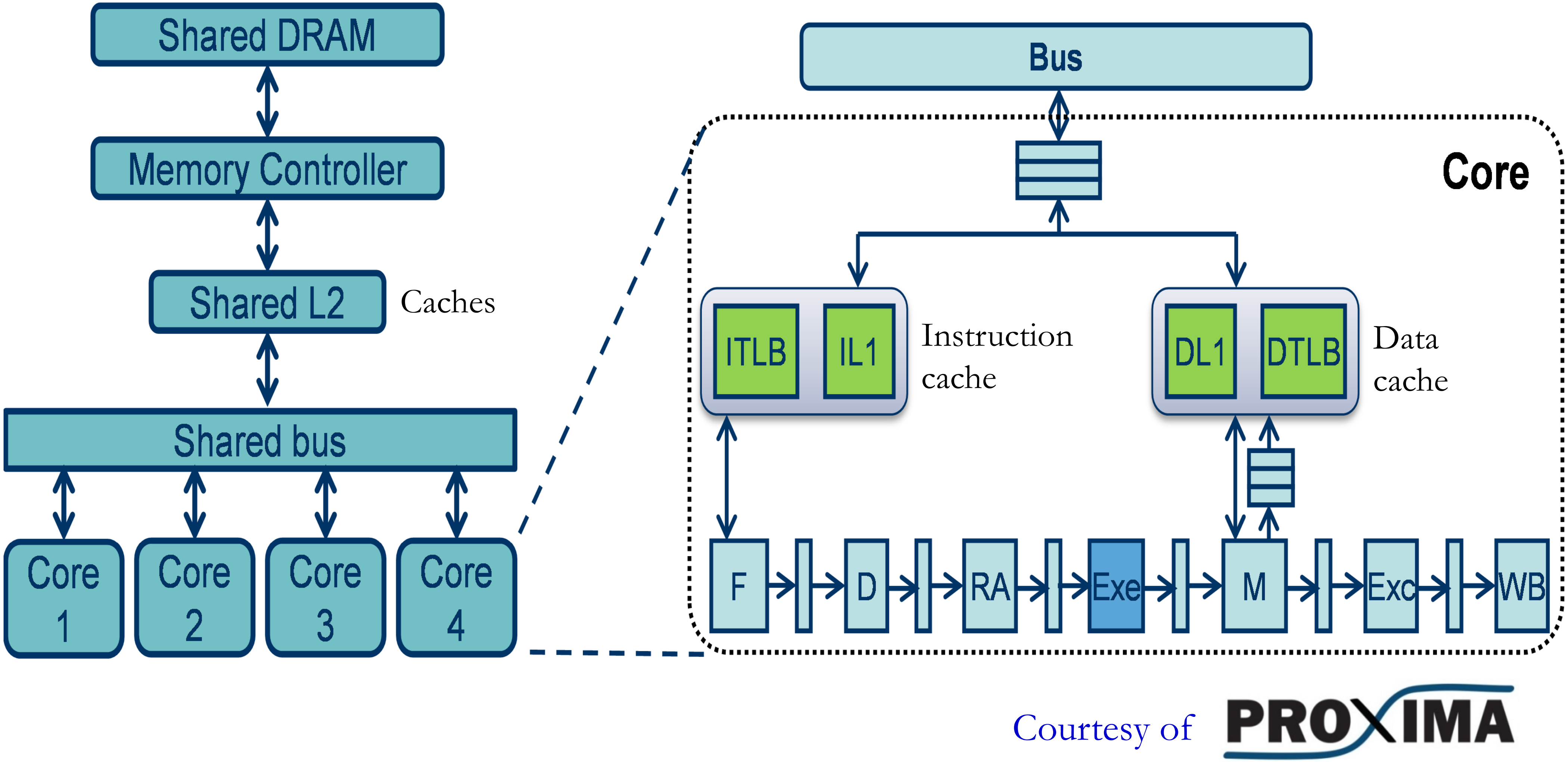


Figure 2. Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

Courtesy of IEEE Computer, January 2011, page 33

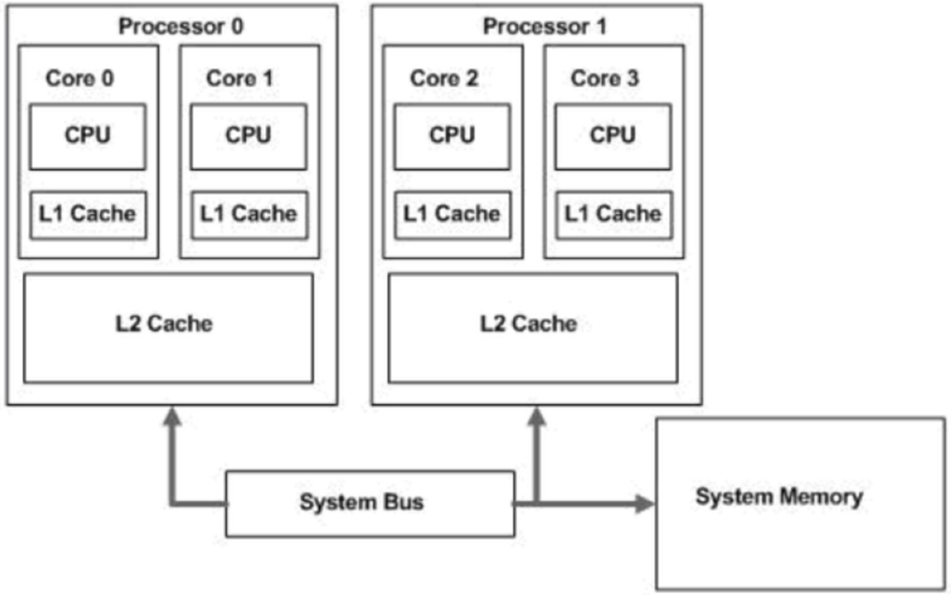
Understanding multicore hardware



Courtesy of PROXIMA

Cache coherence /1

- Now that cores have their own *private* L1 cache …



- … when jobs share data across cores, R/W operations on the same memory location may see *different* copies of it in their respective L1 cache

Cache coherence /2

- Naïve thoughts …
  - Renounce caches
    - Nay, that would bog performance
  - Sharing L1 across cores
    - Nay, parallelism would smash locality
  - Use write-through caches
    - Nay, local reads would lose remote writes

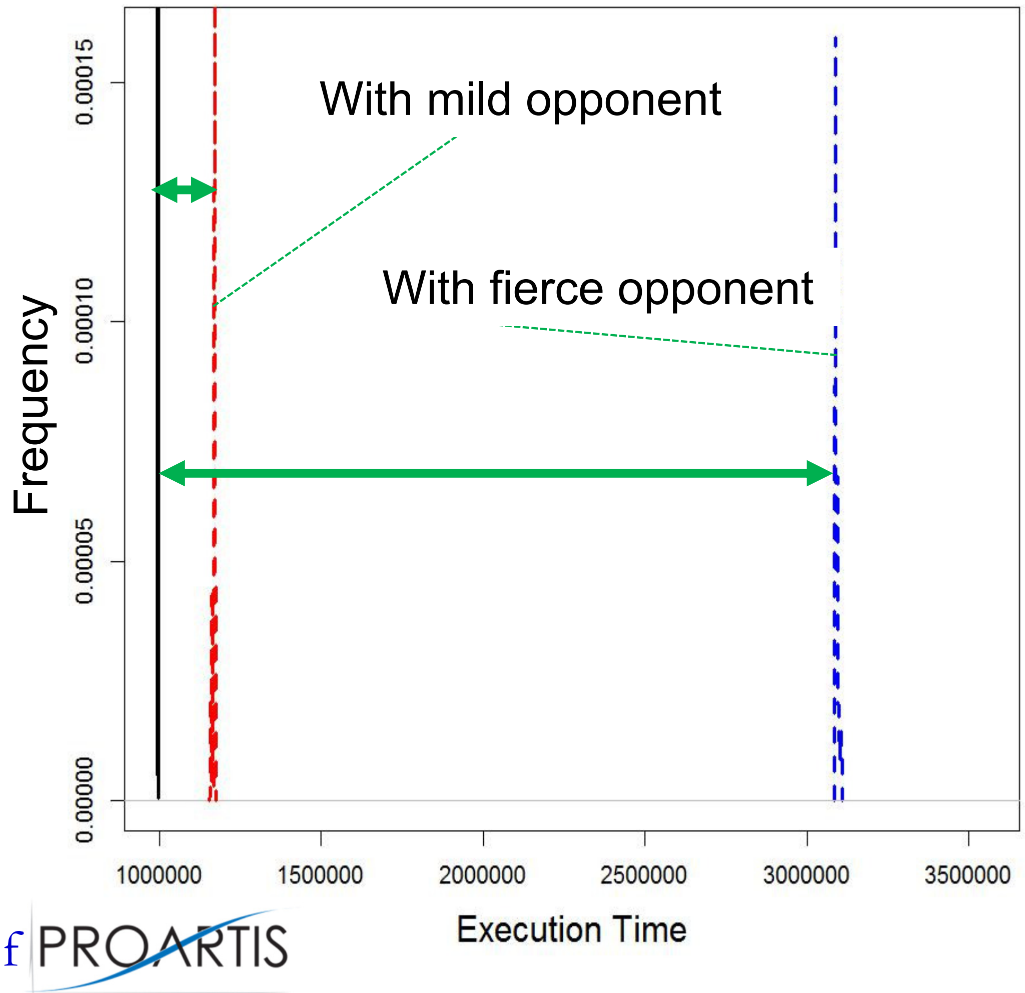
- Req-1: every read must see the effect of every write
  - Either every write updates every L1 (*write update*)
  - Or every write invalidates all L1 copies of same ref (*write invalidate*)
- Req-2: all reads must see the same order of writes
  - Write requests' propagation on the bus tells the order (*snooping*)

Hardware interference /1

- Parallel execution on a multicore processor causes opportunities of contention for the hardware resources shared among the cores
  - This phenomenon did *not* occur on single-core systems
- Such contention *increases* the WCET of running jobs by causing them to *hold the CPU without progressing* (!)
  - This is called *stalling*
  - In single-core processors, a job may be held from running while being ready, but is king when it runs

Hardware interference /2

- The WCET of even the simplest single-path program running alone on a CPU *does not stay the same* when other programs run on other CPUs
- The extent of slow-down is proportional to the amount of off-core work that the programs happen to do
- *The WCET no longer is a composable value!*



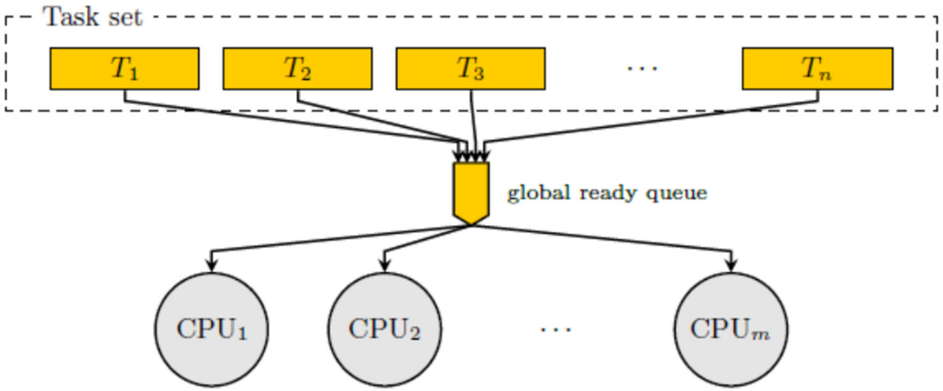
Courtesy of PROARTIS

A reconnaissance taxonomy /2

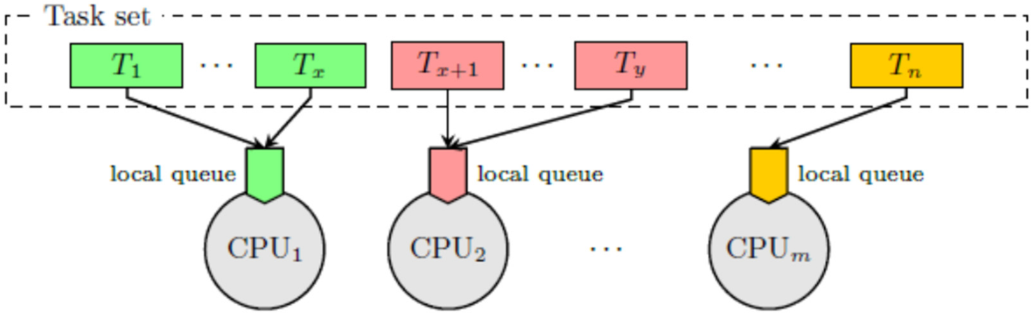
- What scheduling choices do multiprocessors enable?
  - *Global* vs. *partitioned*, or alternatives between them
    - Global scheduling allows jobs to run on *any* core and move across them freely during execution
    - Partitioned scheduling translates into a task-to-core *static* assignment problem, followed by single-core scheduling
- The good-old world of optimality falls apart
  - EDF *no longer* optimal and *not always* better than FPS
- Global scheduling *not always* better than partitioned scheduling
  - Counterintuitive: having multiple assignment choices does *not* beat having just one

A reconnaissance taxonomy /3

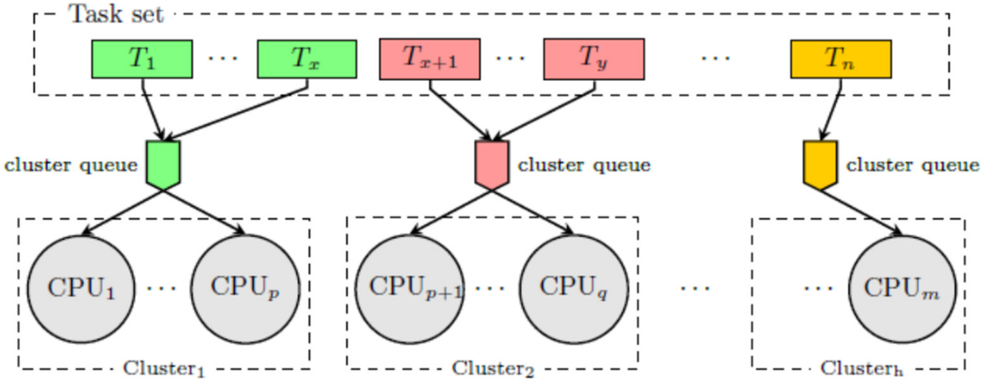
Global



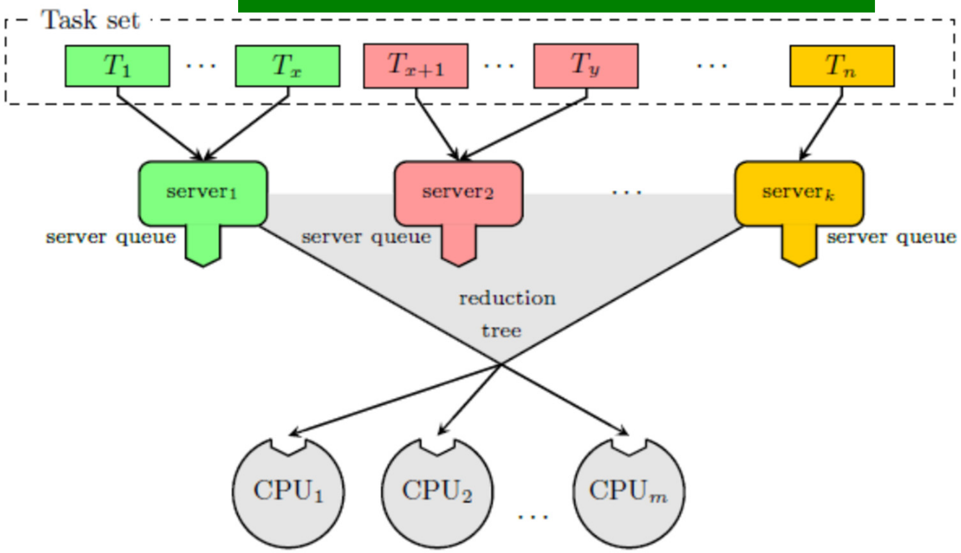
Partitioned



Clustered



Hybrid (semi-partitioned)



Intermission: what is going on?

Credits to Tucker Taft AdaCore The GNAT Pro Company

Let us listen to the words of a language designer, who explains what is changing in the hardware space and what that implies for the software. We will return to this line of argument in the last lecture of this course

What’s the matter with the processor HW?

- **Major, unstoppable shift to multicore, manycore, heterogeneous (e.g. GPGPU) processors, cloud computing**
- **Associated challenge**
  - It is already hard to write safe, correct sequential programs for single-core processors
  - *Will programming for multicores exceed our abilities?*
- **Very opportune goal: provide programming language support to make it easy and natural to write safe (including predictable), correct parallel programs**
  - Perhaps even easier than it is to write safe, correct sequential programs in many existing languages
- **Is that possible?**

Why are they all moving to multi/manycore?

- **Power, power, power**
  - Speeding clock rates above 3 GHz increased power density beyond what the chips (and customer pocketbooks) can bear
  - More and more computing is moving to battery-operated mobile platforms where low power is king
- **With multi/manycore, the theoretical computing performance-per-watt (PPW) can be increased by adding cores, perhaps slowing clock rate a bit**
  - With single-core processor technology, PPW began to *decrease* with increasing clock rates, due to increased power dissipation (source-to-drain leakage)
- **Clock rate doubling (one ramification of Moore's law) came to a screeching halt by the year 2005**

The implications of going "multicore"

- **Clock rate**
  - Clock rates that were doubling about every 2 years since 1985, stalled at about 3 GHz by 2005
  - Had they continued doubling, we would now be buying laptops with clocks at about 50 GHz
- **Cores/chip**
  - Scaling to smaller features has continued
  - Now using added chip real estate for additional CPU cores
  - The number of cores/chip has started doubling since 2005
  - After that (15 years), mainstream commercial x86 chips came at 20-32 cores/chip, Xeon Phi at 70$^+$, GPUs/Adapteva at 1000$^+$
- **Almost back on Moore's Law exponential rocket**
  - But only if considering cores/chip x performance/core

What else is happening to the HW?

- **HW is getting more complicated**
- **Not just a handful of really fast processors**
- **Today's fastest computers have**
  - A giant network of nodes
  - Each node is itself a heterogeneous conglomeration
    - Multiple cores
    - Vector units
    - GPUs or other accelerators
- **Our challenge is to figure how to program these beasts**
  - Ideally we want our programs to *scale without rewriting*, from one core up to a giant server farm or supercomputer
  - Our basic approach is to *eliminate* barriers to parallelization, and remove the *sequential* bias of our programming languages

Concurrency vs. Parallelism

Concurrency

- ***Concurrent*** programming allows the programmer to *simplify the application architecture* by using multiple logical threads of control to reflect the natural patterns of collaboration in the problem domain
  - *Heavier-weight* constructs can be acceptable as they used rarely

Parallelism

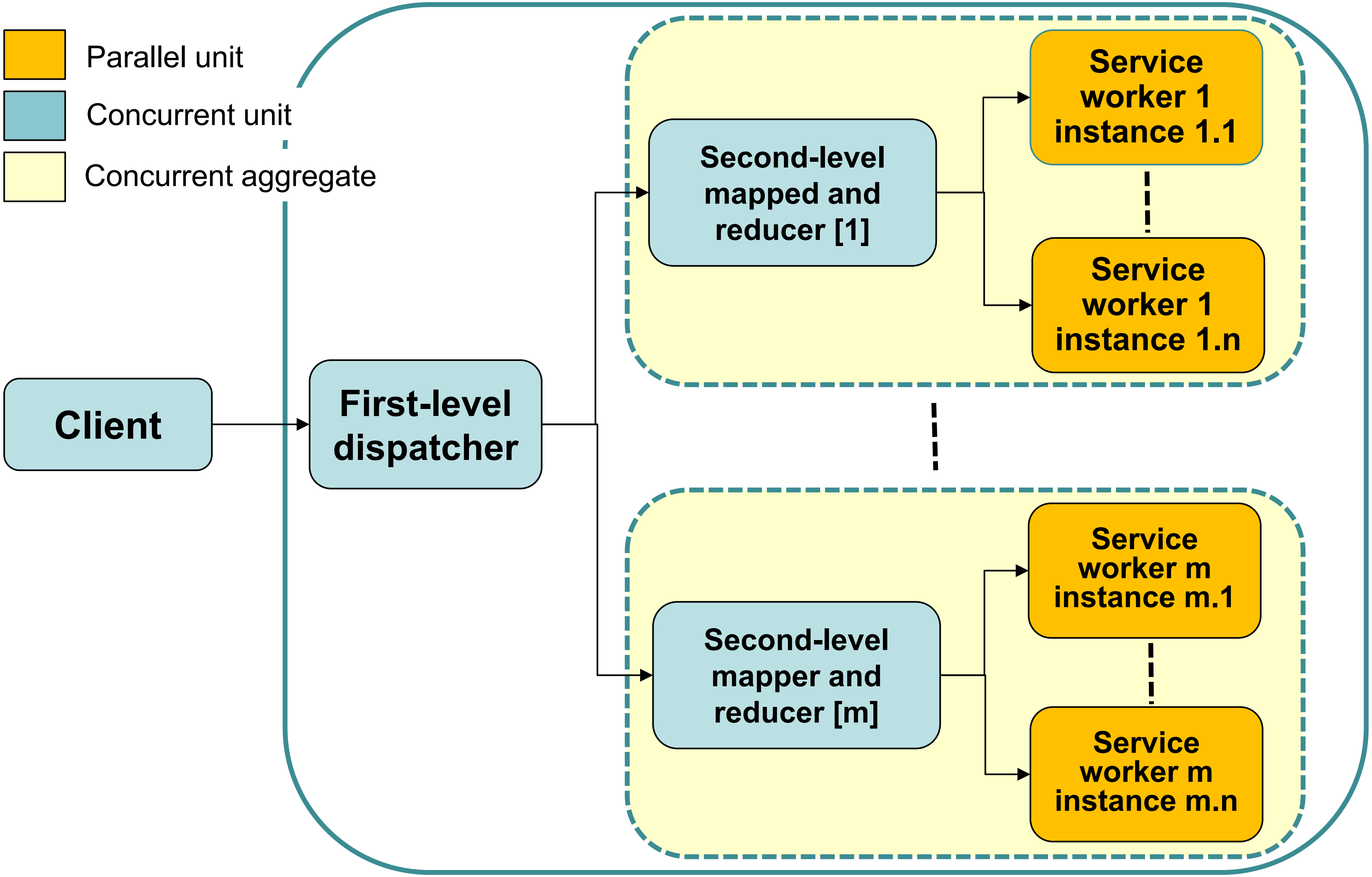
- ***Parallel*** programming allows the programmer to *divide-and-conquer* the problem space, using multiple threads to work in parallel on independent parts of it
  - Constructs should be *light-weight* syntactically *and* at run time as they are used very frequently

Collaboration

Independence

We are heading toward parallelism *within* concurrency

Parallelism within concurrency (example)



Parallel unit
Concurrent unit
Concurrent aggregate
Client
First-level dispatcher
Second-level mapped and reducer [1]
Service worker 1 instance 1.1
Service worker 1 instance 1.n
Second-level mapper and reducer [m]
Service worker m instance m.1
Service worker m instance m.n

All falls apart

- In the multiprocessor world, low-utilization tasksets may be deemed unfeasible: “ the *Dhall’s effect* ” [Dhall & Liu, 1978]
- The known exact schedulability tests have exponential complexity
  - The known sufficient tests with polynomial complexity are pessimistic
- Single-core optimality criteria do not apply anymore
- Global scheduling is not always better than partitioned
- RM or DM priority assignments are not optimal for it
  - The same priority level may have different effects on different cores
- No optimal priority assignment has been found to exist with polynomial complexity

Dhall's effect /1

Task	*T*	*D*	*C*	*U*
a	10	10	5	0.5
b	10	10	5	0.5
c	12	12	8	0.67

$\boldsymbol{m} = 2$

$$\sum_{i=1,..,\boldsymbol{n}=3} U_i = 1.67 < m$$

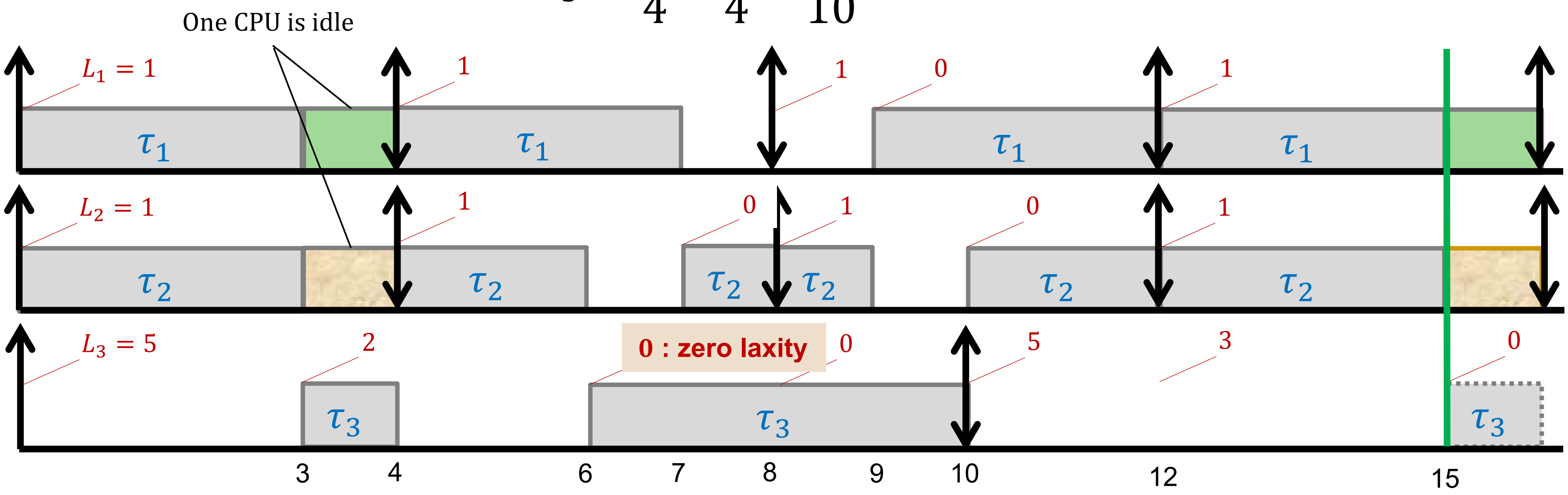
- Under EDF or FPS, *global* scheduling, would run ***a*** and ***b*** first on either of the $m = 2$ processors respectively
- But this would not leave sufficient time for ***c*** to complete
  - 7 time units would be available on each processor, but 8 on neither
- Deadline miss even if the total system is underutilized (**!**)

G-LLF fails too …

$(\varphi_i, \boldsymbol{p_i}, \boldsymbol{e_i}, D_i)$

$$S = \{\tau_1 = (4{,}3), \tau_2 = (4{,}3), \tau_3 = (10{,}5)\}, H_S = 20$$

$$U_s = \frac{3}{4} + \frac{3}{4} + \frac{5}{10} = 2 = \boldsymbol{m}$$



- At $t = 15$, the remaining CPU time is $T_R = m \times (H_S - t) = \mathbf{10}$
- Yet, the time needed is $T_N = e_1 + e_2 + e_3 = \mathbf{11}$

Why does this happen?

Theorem (stating the obvious)

When the total utilization of a periodic task set is equal to the number of processors, and all tasks have the same initial release time ($t = 0$), then no feasible schedule can allow any processor to remain idle for any length of time

- In the LLF example, at times $t = 3$ and $t = 15$, one CPU is left idle for 1 time unit
  - Scheduling was greedy on an ill-based sense of urgency
- That waste will be missed out at time $t = 18$, when $n = 3$ tasks will have laxity $L = 0$ with just $m = 2$ CPUs available
- A "proper" scheduling algorithm should have noticed this problem already at $t = 3$!

Dhall's effect /2

Task	T	D	C	U
$\boldsymbol{d}$	10	10	9	0.9
$\boldsymbol{e}$	10	10	9	0.9
$\boldsymbol{f}$	10	10	2	0.2

$$\boldsymbol{m} = 2$$

$$\sum_{i=1,..,\boldsymbol{n}=3} U_i = m$$

- *Partitioned* scheduling does not work well either
- After $\boldsymbol{d}$ and $\boldsymbol{e}$ are assigned to a CPU, $\boldsymbol{f}$ has no place to run
  - To find room for execution, $\boldsymbol{f}$ would have to migrate from one CPU to the other
  - And $\boldsymbol{d}$ and $\boldsymbol{e}$ should also be willing to yield for $\boldsymbol{f}$ to complete in time

The oddity of software interference /1

- What does the software interference I_i suffered by task τ_i in its busy period become on a multiprocessor?
  - For partitioned scheduling, it reduces to the single-processor case, so it poses no problem
  - For global scheduling on an m-processor system, it occurs *only* when $k \geq m$ tasks are ready simultaneously
    - This means that harmonic periods may be bad news!
- Multiprocessor interference for τ_i can be computed as the sum of all time intervals when m higher-priority tasks execute *in parallel* on all m processors
  - Not the easiest of things to determine …

The oddity of software interference /2

- A very pessimistic bound for G-scheduling considers *all* higher-priority tasks to interfere *always*

$$\boldsymbol{R_k^{max}} = \boldsymbol{C_k} + \boxed{\frac{\mathbf{1}}{\boldsymbol{m}} \sum_{\boldsymbol{\tau_j} \in \boldsymbol{hp}(\boldsymbol{k})} \left(\left\lceil \frac{\boldsymbol{R_k^{max}}}{\boldsymbol{T_j}} \right\rceil \boldsymbol{C_j} + \boldsymbol{Cj} \right)}$$

- This naïve bound however is extremely pessimistic
  - It can be improved, and has been, but for great computational complexity, still without becoming exact

Global scheduling anomalies

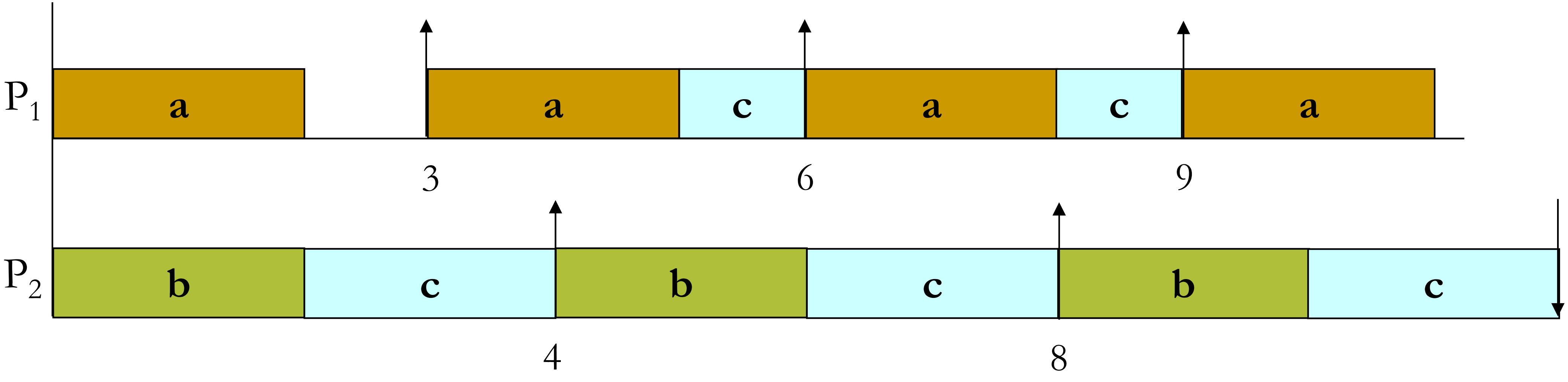
Credits to to B. Andersson and J. Jonsson
Proc. of RTSS WiP Session, 2000, pp. 53–56

- In single-core processor scheduling, the deadline-miss ratio often depends on system load
  - Ergo, increasing tasks' period should decrease utilization and thus decrease the deadline-miss ratio too
- **Multiprocessor anomaly 1**
  - A *decrease* in processor demand from ***hp*** tasks can *increase* the interference on ***lp*** tasks by changing the time windows in which those tasks execute
- **Multiprocessor anomaly 2**
  - A *decrease* in one task's own processor demand may *increase* the interference that it suffers

Anomaly 1: decrease in *hp* utilization

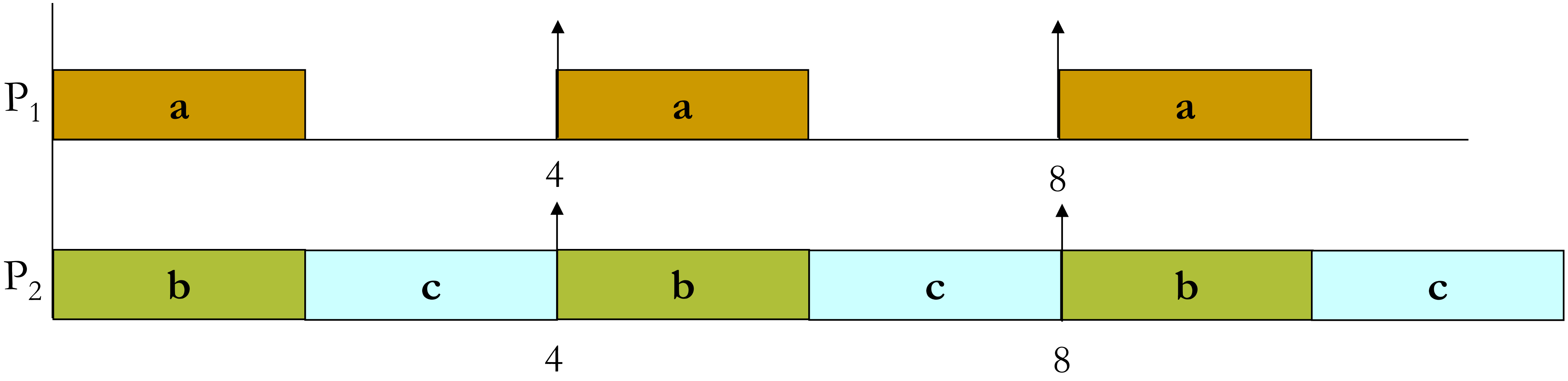
Task	*T*	*D*	*C*	*U*
a	3	3	2	0.67
b	4	4	2	0.50
c	12	12	8	0.67

$m = 2$ processors, $\sum_{i=1,..,n=3} U_i = 1.83 < m$, τ_c is *saturated* ($C_c + I_c = D_c$): any increase in I_c for the same C_c would render τ_c unfeasible



Anomaly 1: continued

- With $T_{a'} = 4 > T_a = 3, U = 1.67$ *decreases*
- But now periods are harmonic and cause τ_c to suffer more interference
  - $I_{c'} = 6 > I_c = 4$, which causes τ_c to miss its deadline

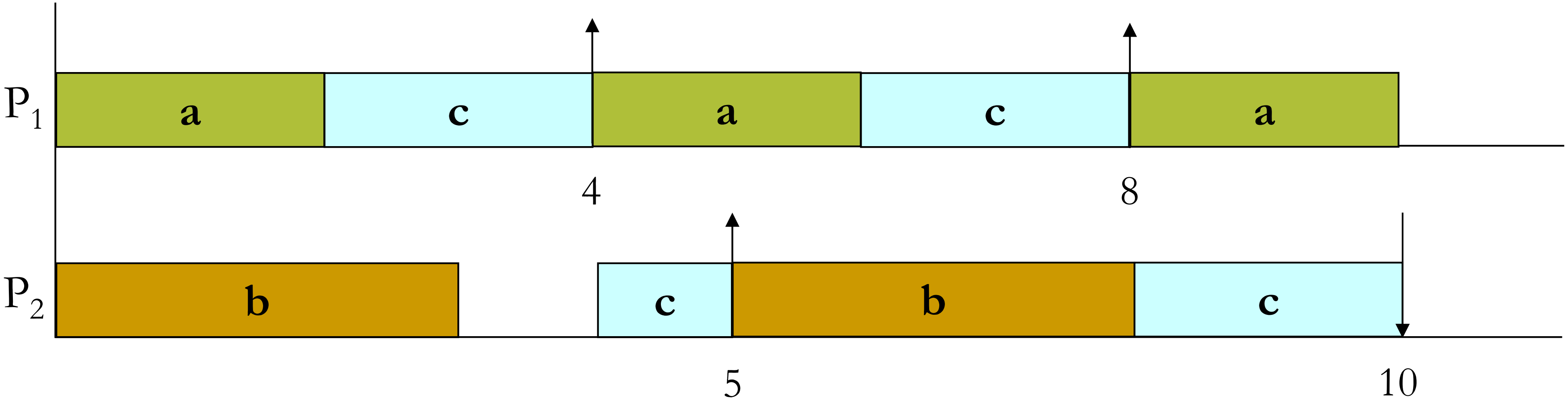


Anomaly 2: decrease in own demand

Task	*T*	*D*	*C*	*U*
a	4	4	2	0.5
b	5	5	3	0.6
c	10	10	7	0.7

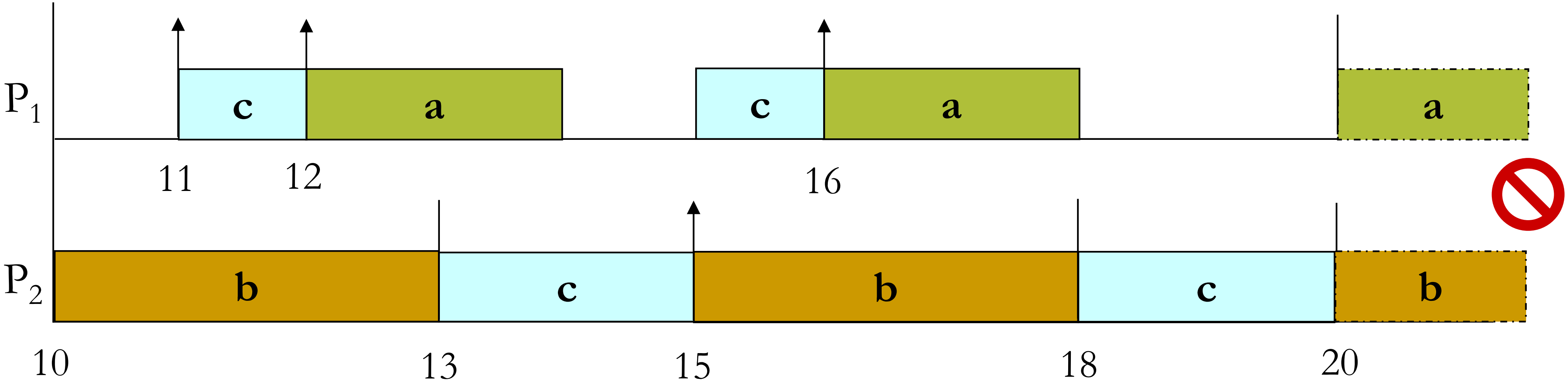
$m = 2$ processors and $U = 1.8$

τ_c with $I_c = 3$ is *saturated*



Anomaly 2: continued

- With $T_{c'} = 11 > T_c = 10$, $U = 1.74$ *decreases*
- But then $I_{c'} = 5 > I_c = 3$, *increases*, for τ_c's 2nd job
  - Which also shows that the critical-instant hypothesis no longer holds!



The defeat of greedy schedulers

- Greedy algorithms are easy to explain, study, and implement
- They work very well on single-core processors, where *the urgency of a job collapses into a single value*, which can be used to schedule jobs greedily
- Greedy algorithms fail on multiprocessors, where *computation* (one's own progress) *and parallelism* (use of all cores) *are distinct dimensions*
- Optimality in multicore scheduling needs to use different principles altogether

Enters proportionate fairness

- An airline has m planes and n flight crews, with $n > m$
  - All planes and crews are based in the same city
- Exactly m crews are scheduled to work on any given days
  - Due to seniority, job performance, or other factors, it may be desirable to schedule some crews *more often* than others
    - This notion reflects the crew work period
- For each crew k, W_k is the fraction of all days that crew x is desired to work, such that $\sum_k W_k = m$
- The airline wants a scheduler that produces a schedule in which every crew works at a balanced rate
  - One where, after t workdays (the hyperperiod), crew k will have worked either $\lfloor W_k \times t \rfloor$ or $\lceil W_k \times t \rceil$ workdays

Unveiling the analogy

- The airplanes $\{A\}$ are the CPUs $(m > 1)$
- The crews $\{C\}$ are the tasks $(n \geq m)$
  - Assigning $\{C\}$ to $\{A\}$ is a multiprocessor scheduling problem
- The contract with the crews is that each i of them will receive work according to their privilege w_i
  - For tasks, this is the utilization rate
  - This is the first dimension of the multiprocessor scheduling problem (*progress*)
- The contract with the airplanes is that they will all be given a crew, $\sum_{i=1,..,n} W_i = m$
  - This is the second dimension of the multiprocessor scheduling problem (*parallelism*)

P-fair scheduling [Baruah et al. 1996]

- *Proportional progress* is a form of proportionate fairness also known as ***P-fairness***
  - Each task τ_i is assigned processing resources in proportion to its *weight* $W_i = \frac{C_i}{T_i}$ so that its computation may progress steadily
    - Think of real-time multimedia applications …
- At every time $t > 0$, task τ_i must have been scheduled either $\lfloor W_i \times t \rfloor$ or $\lceil W_i \times t \rceil$ time units
  - Perfectly analogous to the airline crew schedule problem
  - Without loss of generality, preemption is assumed to occur solely at integral time units
- The workload model is assumed to be periodic with implicit deadlines

P-fair scheduling /2

- $\boldsymbol{lag}(S, \tau_i, t)$ is the delta between the total resource allocation that task τ_i should have received in $[0, t)$ and what schedule S gave it
- For a P-fair schedule S, at time t
  - τ_i is *ahead* if and only if $\boldsymbol{lag}(S, \tau_i, t) < 0$
  - τ_i is *behind* if and only if $\boldsymbol{lag}(S, \tau_i, t) > 0$
  - τ_i is *punctual* if and only if $\boldsymbol{lag}(S, \tau_i, t) = 0$
- Scheduling occurs at “integral” units of time
  - This reflects the analogy that the assignment of one crew is for a full (integral) airplane service

P-fair scheduling /3

- $\boldsymbol{\alpha}(x)$ is the *characteristic* (infinite) *string* of task τ_x over $\{-, 0, +\}$ for $t \in \mathbb{N}$ with

  $$\boldsymbol{\alpha}_t(x) = \boldsymbol{sign}(W_x \times (t+1) - \lfloor W_x \times t \rfloor - 1)$$

  Above or below the integral approximation of the ***fluid rate curve***

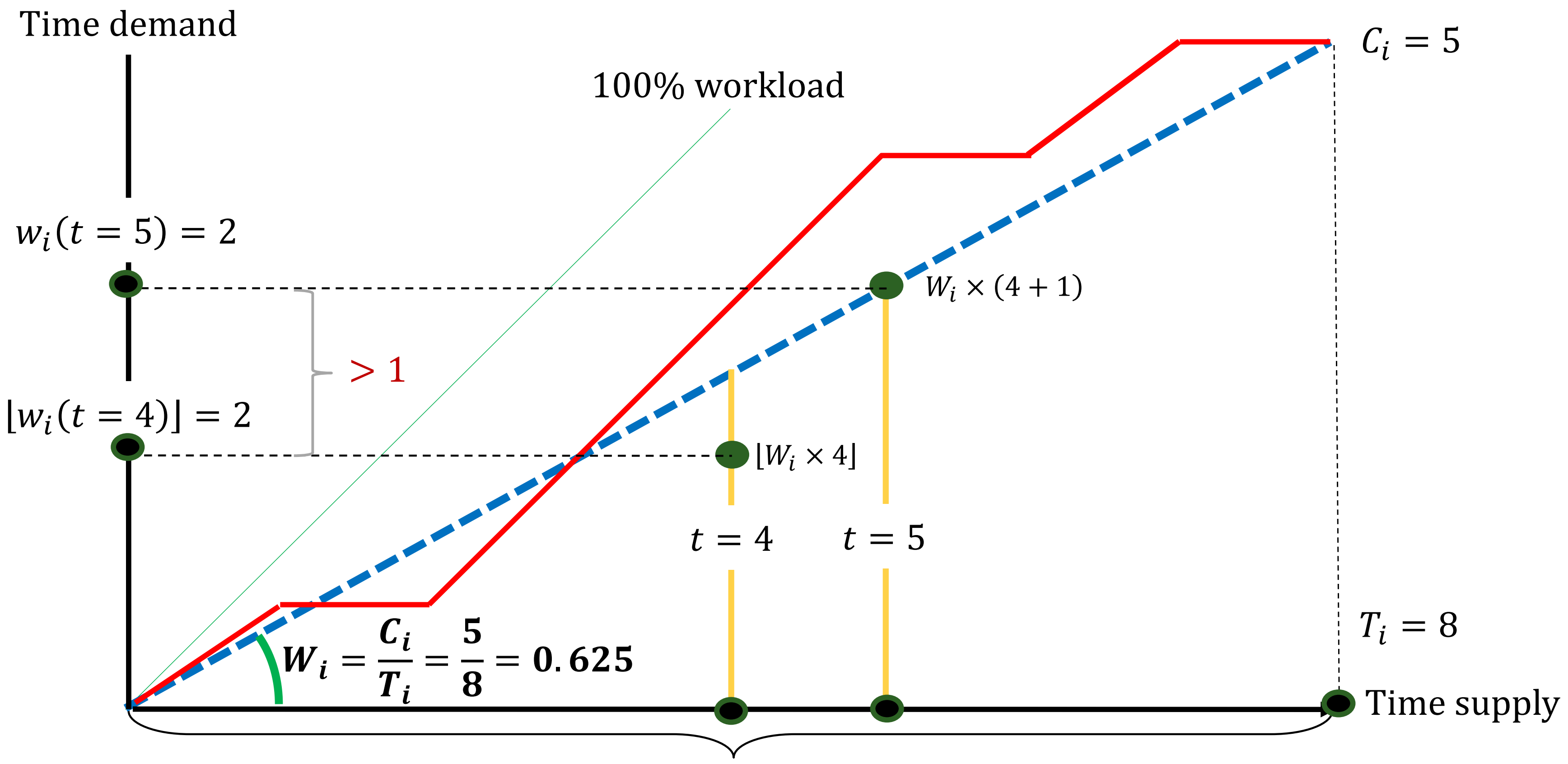
- $\boldsymbol{\alpha}(x, t)$ is the *characteristic substring*

  $$\boldsymbol{\alpha}_{t+1}(x)\boldsymbol{\alpha}_{t+2}(x) \ldots \boldsymbol{\alpha}_{t'}(x) \text{ of task } \tau_x \text{ at time } t$$

  where $t' = min\, i : i > t : \boldsymbol{\alpha}_i(x) = 0$

- For a P-fair schedule S at time t, task τ_i is
  - *Urgent* : τ_i is *behind* and $\boldsymbol{\alpha}_t(\tau_i) \neq -$ (τ_i has credits to claim)
  - *Tnegru* : τ_i is *ahead* and $\boldsymbol{\alpha}_t(\tau_i) \neq +$ (τ_i has stolen from others)
  - *Contending* otherwise

The fluid rate curve



$$\boldsymbol{\alpha}_{t=4}(i) = \boldsymbol{sign}(W_i \times (4 + 1) - \lfloor W_i \times 4 \rfloor - 1) = 3.125 - 2 - 1 = +$$

At time $t = 4 + 1$, if not scheduled at $t = 4$, task τ_i might have a *credit* that could not be satisfied in one single round of scheduling:

Properties of a P-fair schedule S

- For task τ_i *ahead* at time t under S

  tnegru

  - If $\boldsymbol{\alpha_t}(\tau_i) = -$ and τ_i not scheduled at t then τ_i is *ahead* at $t + 1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = 0$ and τ_i not scheduled at t then τ_i is *punctual* at $t + 1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = +$ and τ_i not scheduled at t then τ_i is *behind* at $t + 1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = +$ and τ_i scheduled at t then τ_i is *ahead* at $t + 1$
- For task τ_i *behind* at time t under S
  - If $\boldsymbol{\alpha_t}(\tau_i) = -$ and τ_i scheduled at t then τ_i is *ahead* at $t + 1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = -$ and τ_i not scheduled at t then τ_i is *behind* at $t + 1$

  urgent

  - If $\boldsymbol{\alpha_t}(\tau_i) = 0$ and τ_i scheduled at t then τ_i is *punctual* at $t + 1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = +$ and τ_i scheduled at t then τ_i is *behind* at $t + 1$

P-fair scheduling /4

- To preserve P-fairness
  - Every task *urgent* at time t *must* be scheduled at t so that P-fairness can be preserved
  - *No* task *tnegru* at time t can be scheduled at t
- With m resources, n tasks, and n_0 *tnegru*, n_1 *contending*, n_2 *urgent* tasks at time t $(n = n_0 + n_1 + n_2)$, two situations must be avoided
  - $\boldsymbol{n_2 > m}$, when all *urgent* tasks *cannot* be scheduled: some tasks will never be able to catch up
  - $\boldsymbol{n_0 > n - m}$, when some *tnegru* tasks will be scheduled wasting CPU time on them that will be regretted later

P-fair scheduling /5

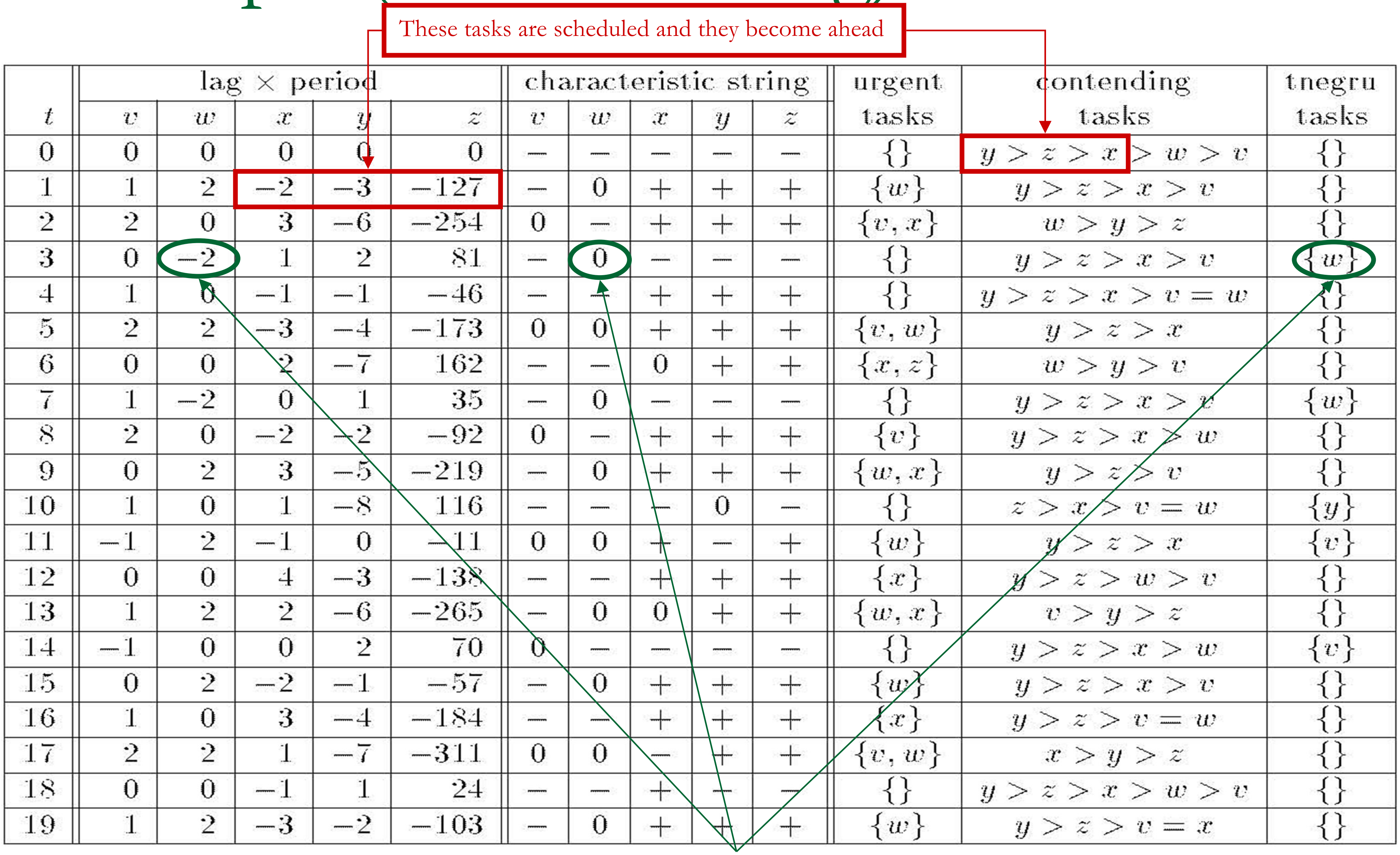
- The commandments of the **PF** scheduling algorithm
  - Always schedule all *urgent* tasks
  - Allocate the remaining resources to the hp *contending* tasks according to the total order function $\sqsupseteq$ with ties broken arbitrarily
    - At time t, $x \sqsupseteq y : \boldsymbol{\alpha}(x,t) \geq \boldsymbol{\alpha}(y,t)$, where $- < 0 < +$
- With PF, we have $\sum_{x \in [0,n]} W_x = m$
  - Dummy task added to task set to fill utilization up to m
- No problematic situation can occur with the PF algorithm
  - PF always has $n_2 \leq m$ and $n_0 \leq n - m$
  - A property of seeking the closest approximation of the fluid rate curve for all tasks

Example (PF scheduling) /1

Task	C	T	W
τ_v	1	3	$\frac{1}{3}$
τ_w	2	4	$\frac{1}{2}$
τ_x	5	7	$\frac{5}{7}$
τ_y	8	11	$\frac{8}{11}$
τ_z	$(3 - U) \times \frac{H}{2}$	$\frac{H}{2}$	$m - U$

- $m = 3$ processors
- $n = 4$ tasks
- $U = \sum_{i=v,w,x,y} \frac{C_i}{T_i} = 2.27489 \ldots$
- τ_z is a dummy task used to top up system utilization to m
  - $U_z = m - U$
- τ_z's period is set to the system hyperperiod H
  - This time we just halved it as T_z and C_z happen to be even

Example (PF scheduling) /2



These tasks are scheduled and they become ahead

t	lag × period v	w	x	y	z	characteristic string v	w	x	y	z	urgent tasks	contending tasks	tnegru tasks
0	0	0	0	0	0	−	−	−	−	−	{}	$y > z > x > w > v$	{}
1	1	2	−2	−3	−127	−	0	+	+	+	{w}	$y > z > x > v$	{}
2	2	0	3	−6	−254	0	−	+	+	+	{v, x}	$w > y > z$	{}
3	0	−2	1	2	81	−	0	−	−	−	{}	$y > z > x > v$	{w}
4	1	0	−1	−1	−46	−	−	+	+	+	{}	$y > z > x > v = w$	{}
5	2	2	−3	−4	−173	0	0	+	+	+	{v, w}	$y > z > x$	{}
6	0	0	2	−7	162	−	−	0	+	+	{x, z}	$w > y > v$	{}
7	1	−2	0	1	35	−	0	−	−	−	{}	$y > z > x > v$	{w}
8	2	0	−2	−2	−92	0	−	+	+	+	{v}	$y > z > x > w$	{}
9	0	2	3	−5	−219	−	0	+	+	+	{w, x}	$y > z > v$	{}
10	1	0	1	−8	116	−	−	−	0	−	{}	$z > x > v = w$	{y}
11	−1	2	−1	0	−11	0	0	+	−	+	{w}	$y > z > x$	{v}
12	0	0	4	−3	−138	−	−	+	+	+	{x}	$y > z > w > v$	{}
13	1	2	2	−6	−265	−	0	0	+	+	{w, x}	$v > y > z$	{}
14	−1	0	0	2	70	0	−	−	−	−	{}	$y > z > x > w$	{v}
15	0	2	−2	−1	−57	−	0	+	+	+	{w}	$y > z > x > v$	{}
16	1	0	3	−4	−184	−	−	+	+	+	{x}	$y > z > v = w$	{}
17	2	2	1	−7	−311	0	0	−	+	+	{v, w}	$x > y > z$	{}
18	0	0	−1	1	24	−	−	+	−	−	{}	$y > z > x > w > v$	{}
19	1	2	−3	−2	−103	−	0	+	+	+	{w}	$y > z > v = x$	{}

Summary

- Multicore processors are the processor makers' escape route to the doom of Moore's law, yet their advent shakes the foundations of real-time systems theory that rest on the single-runner assumption
- We are confounded between the urge to schedule greedily and the actual inanity of it
- We begin to see that optimality here is a very different story