7.a Multicore systems – initial reckoning

Credits to various authors (acknowledged in place)

Fundamental issues

- Hardware architecture taxonomy
  - Homogeneous vs. heterogeneous processors
    - Research focused first on SMP (*symmetric multiprocessors*) which make a much simpler problem
    - Attention is now shifting to heterogeneous processors, which are becoming dominant in a variety of application domains
- Scheduling approach
  - Global / partitioned, or alternatives between these extremes
    - Partitioning = allocation problem followed by single-CPU scheduling
- Optimality criteria are shattered
  - EDF no longer optimal and not always better than FPS
  - Global scheduling not always better than partitioned

Hardware architecture taxonomy

- A multiprocessor (or multi-core) is *tightly coupled*
  - Global status and workload information on all processors (cores) can be kept current at low cost
  - The system may use a centralized dispatcher and scheduler
  - When each processor (core) has its own scheduler, the decisions and actions of all schedulers are coherent
    - Scheduling in this model is an NP-hard problem
- A distributed system is *loosely coupled*
  - It is too costly to keep global status
  - There usually is a dispatcher / scheduler per processor

What is changing in the HW world?

Credits to Tucker Taft

AdaCore
The GNAT Pro Company

What's the matter with the processor HW?

- **Big, unstoppable shift to multicore, manycore, heterogeneous (e.g. GPGPU), cloud computing**
- **Associated challenge**
  - It is already hard to write safe, correct sequential programs for single-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 all moving to multi / manycore?

- **Power, power, power**
  - Speeding clock rates past 3 GHz increased power density beyond what the chips (and customer pocketbooks) could 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, PPW began to *decrease* with increasing clock rates, due to increased source-to-drain leakage
- **Clock rate doubling came to a screeching halt by the year 2005**

The *right turn* in processor performance

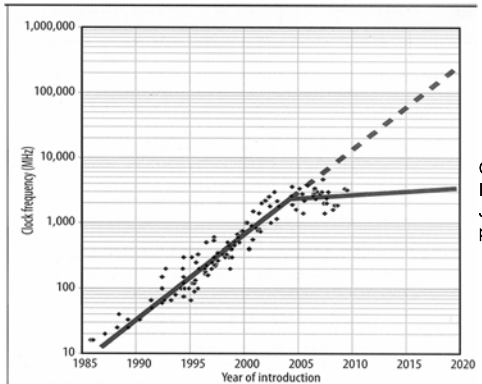


Figure 2. Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

Courtesy IEEE Computer, January 2011, page 33

What are the implications of this right turn?

- **Clock rate**
  - Clock rates that were doubling about every 2 years, 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
  - In those 10+ 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 constructs allow the programmer to *simplify the program* by using multiple logical threads of control to reflect the natural concurrency in the problem domain
  - Heavier-weight constructs can be acceptable as they used rarely

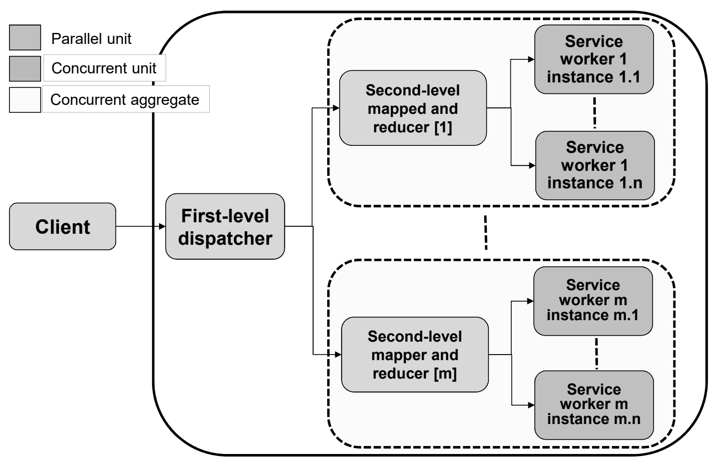
Parallelism

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

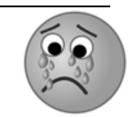
Cooperation ***Independence***

We are heading toward parallelism *within* concurrency

Parallelism within concurrency (example)



State of the art: what a loss!



- Low-utilization task sets may be deemed not schedulable
  - Long-known as the *Dhall's effect* [Dhall & Liu, 1978]
- The known exact schedulability tests have exponential complexity
  - The known sufficient tests with polynomial complexity are pessimistic
- Several routes for scheduling
  - Global, partitioned, or hybrids of them
  - Partitioned scheduling corresponds to an allocation problem followed by single-CPU scheduling (some like it better …)
- Single-processor optimality criteria do *not* apply
  - EDF no longer optimal and not always better than FPS
  - Global scheduling not always better than partitioned
- Rate-monotonic priority assignment is *not* optimal
  - The same priority level may have different meaning on different cores
  - No known optimal priority assignment with polynomial time 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

$m = 2$

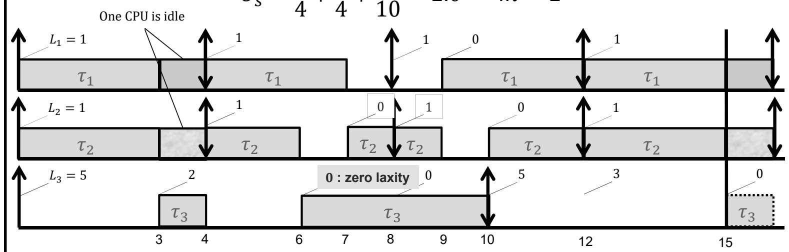
$\sum_i U_i = 1.67 < m$

- Under *global* scheduling, G-EDF and G-FPS would run a and b first on each of the 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 also fails …

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

$U_s = \frac{3}{4} + \frac{3}{4} + \frac{5}{10} = 2.0 \rightarrow m = 2$



- 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

- At time $t = 3$ (and then again at $t = 15$) in the LLF example, one CPU is left idle for 1 time unit
- That time will be missed out later, at time $t = 18$, when *all three tasks* will have laxity $L = 0$ and only two CPUs are available
- A proper scheduling algorithm should have noticed this problem already at $t = 3$!

Dhall's effect /2

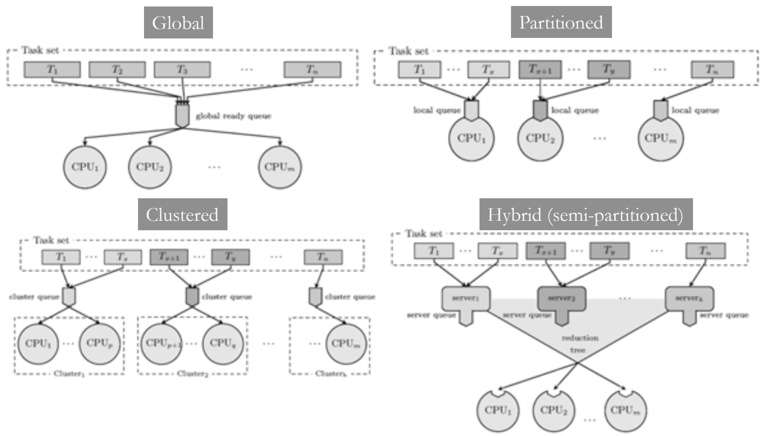
Task	T	D	C	U
d	10	10	9	0.9
e	10	10	9	0.9
f	10	10	2	0.2

$m = 2$

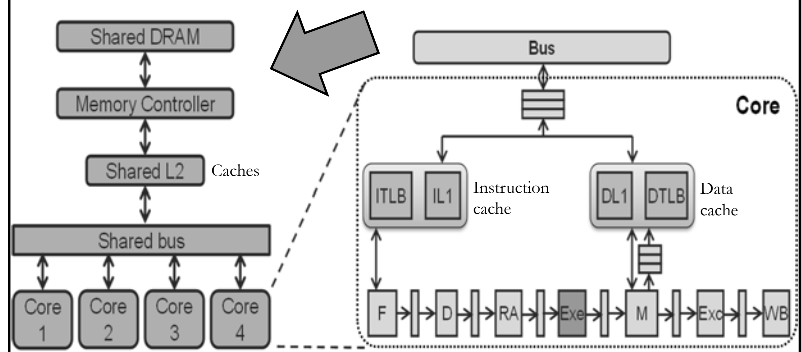
$\sum_i U_i = m$

- *Partitioned* scheduling does not work here any better
- After d and e are assigned, f has no place to run
  - It needs to migrate from one CPU to the other to find room for execution
- And it also needs that d and e are willing to yield for f to complete in time

The multicore scheduling landscape



Understanding the hardware



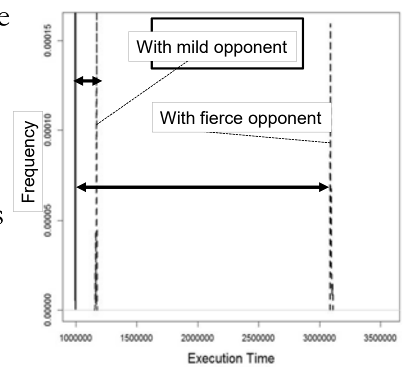
Courtesy of PROXIMA

Hardware interference /1

- Parallel execution on a multiprocessor causes many opportunities of contention for hardware resources that are shared among the cores
- This phenomenon increases the execution time of running threads by causing them to hold the CPU *without* progressing (**!**)
  - Unlike software interference on single CPU, where a thread may be held from running when being ready

Hardware interference /2

- The WCET of even the simplest (single-path) program running alone does not stay the same when other programs execute on other CPUs



Courtesy of PROARTIS

Software interference /1

- What does the SW interference I_i suffered by task τ_i in its busy period become on a multiprocessor?
  - For partitioned scheduling, obviously it reduces to the single-processor case
  - For global scheduling on an m-processor system, instead, interference occurs *only* when more than $\mathrm{k} \geq m$ tasks are ready simultaneously
- Multiprocessor interference can be computed as the sum of all intervals when m higher-priority tasks execute in parallel on all m processors

Software interference /2

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

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

- 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

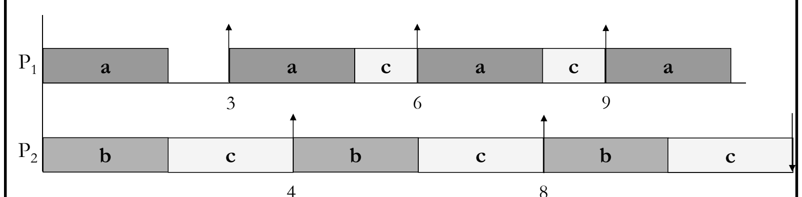
Credits to to B. Andersson and J. Jonsson for their work in proc. of RTSS WiP Session, 2000, pp. 53–56

- In single-processor scheduling, the deadline miss ratio often highly depends on the system load
  - This suggests that increasing tasks' period should decrease the utilization and thus decrease the deadline miss ratio
- **Anomaly 1**
  - A *decrease* in processor demand from higher-priority tasks can *increase* the interference on lower-priority tasks because of the change in the time windows in which those tasks execute
- **Anomaly 2**
  - A *decrease* in one task's processor demand may *increase* the interference that it suffers

Anomaly 1: decrease in *hp* demand

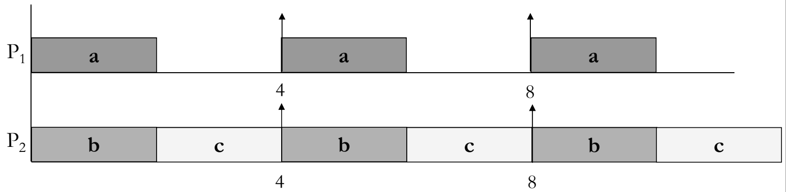
Task	*T*	*D*	*C*	*U*
a	3	3	2	0.67
b	4	4	2	0.50
c	12	12	8	0.67

$m = 2$ processors and $\sum_i U_i = 1.83$, but τ_c is *saturated* because $C_c + I_c = D_c$, hence any increase in I_c for the same C_c would make τ_c unschedulable



Anomaly 1/b

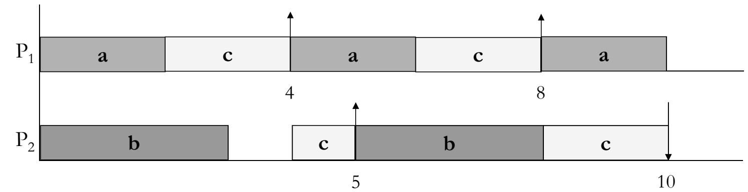
- With $T_a = 4$, longer period, the overall load decreases to $U = 1.67$
- But in this way I_c *increases* from 4 to 6 and τ_c misses its deadline (!)



Anomaly 2: decrease in own demand

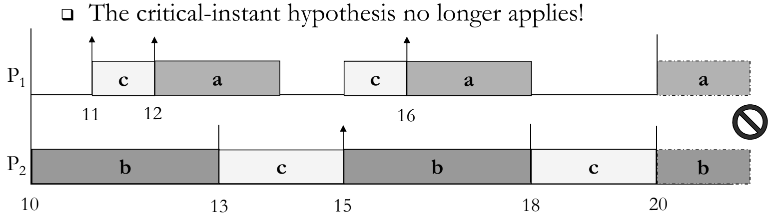
Task	*T*	*D*	*C*	*U*
a	4	4	2	0.5
b	5	5	3	0.6
c	10	10	7	0.7

$m = 2$ processors and $U = 1.8$, but τ_c with $I_c = 3$ is *saturated*



Anomaly 2/b

- With $T_c = 11$, longer period, the overall load decreases to $U = 1.74$
- But in this way I_c *increases* from 3 to 5 (!) as it becomes visible in the <u>second</u> job of τ_c
  - The critical-instant hypothesis no longer applies!



The defeat of greedy schedulers

- Greedy algorithms are easy to explain, study, and implement
- They work very well on single-core processors, where *they collapse the urgency of a job into a single value* and use it to schedule jobs greedily
- But greedy algorithms fail on multiprocessors, where *computation and parallelism are distinct dimensions*
- Optimality in multicore scheduling needs to use different principles …

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 it progresses steadily
  - Useful, e.g., for real-time multimedia applications
- At every time t, task τ_i must have been scheduled either $\lfloor W_i \times t \rfloor$ or $\lceil W_i \times t \rceil$ time units
  - Without loss of generality, preemption is assumed to only occur at integral time units
  - The workload model is assumed to be periodic

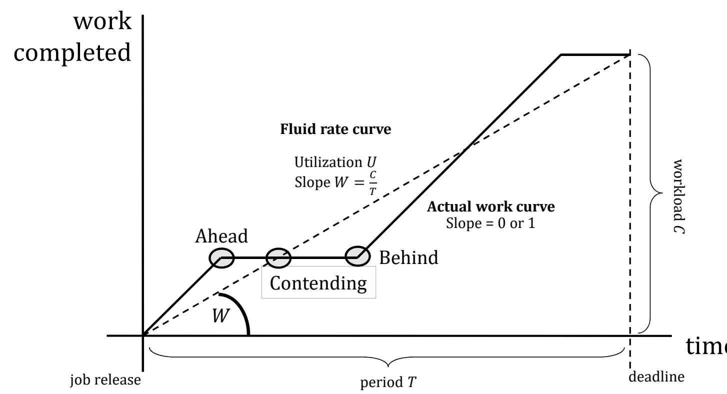
P-fair scheduling /2

- $\boldsymbol{lag}(S, \tau_i, t)$ is the difference between the total resource allocation that task τ_i should have received in $[0, t)$ and what it received under schedule S
- For a P-fair schedule S at time t
  - τ_i is *ahead* iff $\boldsymbol{lag}(S, \tau_i, t) < 0$
  - τ_i is *behind* iff $\boldsymbol{lag}(S, \tau_i, t) > 0$
  - τ_i is *punctual* iff $\boldsymbol{lag}(S, \tau_i, t) = 0$

P-fair scheduling /3

- $\boldsymbol{\alpha}(x)$ is the *characteristic* (infinite) *string* of task τ_x over $\{-, 0, +\}$ for $t \in \mathbb{N}$ with
  - $\boldsymbol{\alpha}_t(x) = \boldsymbol{sign}(W_x \cdot (t+1) - \lfloor W_x \cdot t \rfloor - 1)$
    - Distance from the integral approximation of ***fluid rate curve***
  - $\boldsymbol{\alpha}(x, t)$ is the *characteristic substring* $\boldsymbol{\alpha}_{t+1}(x)\boldsymbol{\alpha}_{t+2}(x) \ldots \boldsymbol{\alpha}_{t'}(x)$ of task τ_x 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* iff τ_i is *behind* and $\boldsymbol{\alpha_t}(\tau_i) \neq -$
  - *Tnegru* iff τ_i is *ahead* and $\boldsymbol{\alpha_t}(\tau_i) \neq +$
  - *Contending* otherwise

Fluid Rate Curve



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

- General principle of 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 without breaking P-fairness
- Breakage with n_0 *tnegru*, n_1 *contending*, n_2 *urgent* tasks at time t, with m resources and $n = n_0 + n_1 + n_2$ tasks
  - If $n_2 > m$, the scheduling algorithm cannot schedule all *urgent* tasks → some of them will never be able to catch back
  - If $n_0 > n - m$, the scheduling algorithm is forced to schedule some *tnegru* tasks and consequently waste CPU time on them

P-fair scheduling /5

- The commandments of the **PF** scheduling algorithm
  - Schedule all *urgent* tasks
  - Allocate the remaining resources to the highest-priority *contending* tasks according to the total order function $\supseteq$ with ties broken arbitrarily
    - $x \supseteq y$ iff $\boldsymbol{\alpha}(x,t) \geq \boldsymbol{\alpha}(y,t)$
    - And the comparison between the characteristics substrings is resolved lexicographically with $- < 0 < +$
- With PF we have $\sum_{x \in [0,n]} W_x = m$
  - A dummy task may need to be added to the task set to top utilization up
- No problem situation can occur with the PF algorithm

Example (PF scheduling) /1

Task	C	T	W
τ_v	1	3	0.333...
τ_w	2	4	0.5
τ_x	5	7	0.714...
τ_y	8	11	0.727...
τ_z	335	462	3-U

- $m = 3$ processors
- $n = 4$ tasks
- τ_z is a dummy task used to top up system utilization to m
- In general, τ_z's period is set to the system hyperperiod
  - This time we halved it
- With PF we always have $n_2 > m$ and $n_0 \leq n - m$

Example (PF scheduling) /2

These tasks are scheduled and they become ahead

t	lag × period					characteristic string					urgent tasks	contending tasks	tnegru tasks
	v	w	x	y	z	v	w	x	y	z			
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	+	+	+	$\{w\}$	$y > z > v = x$	{}