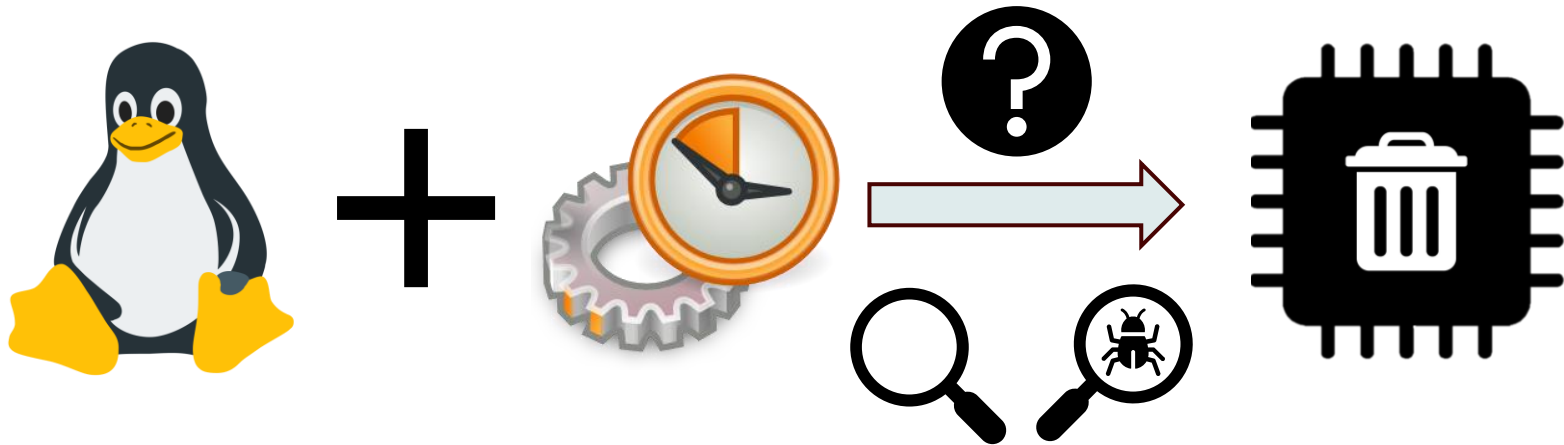
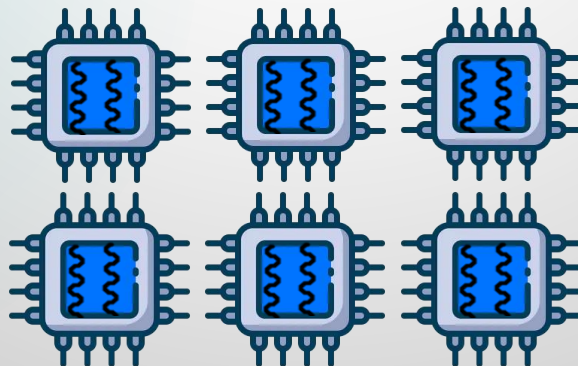
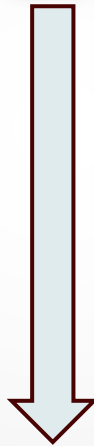
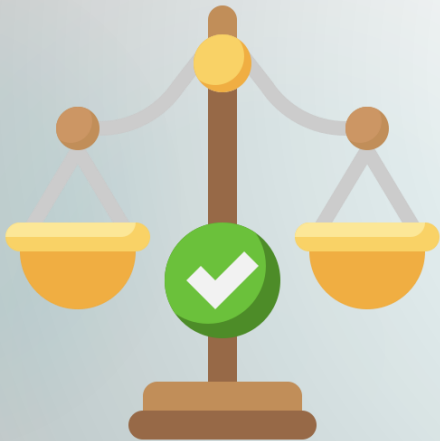
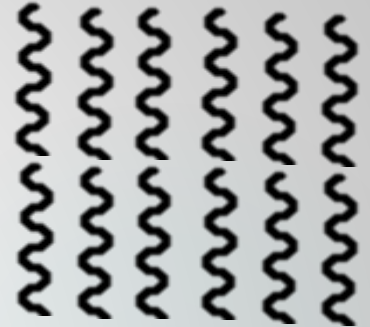
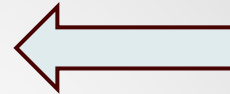
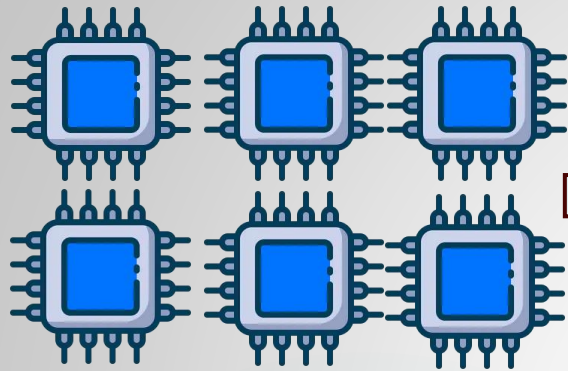


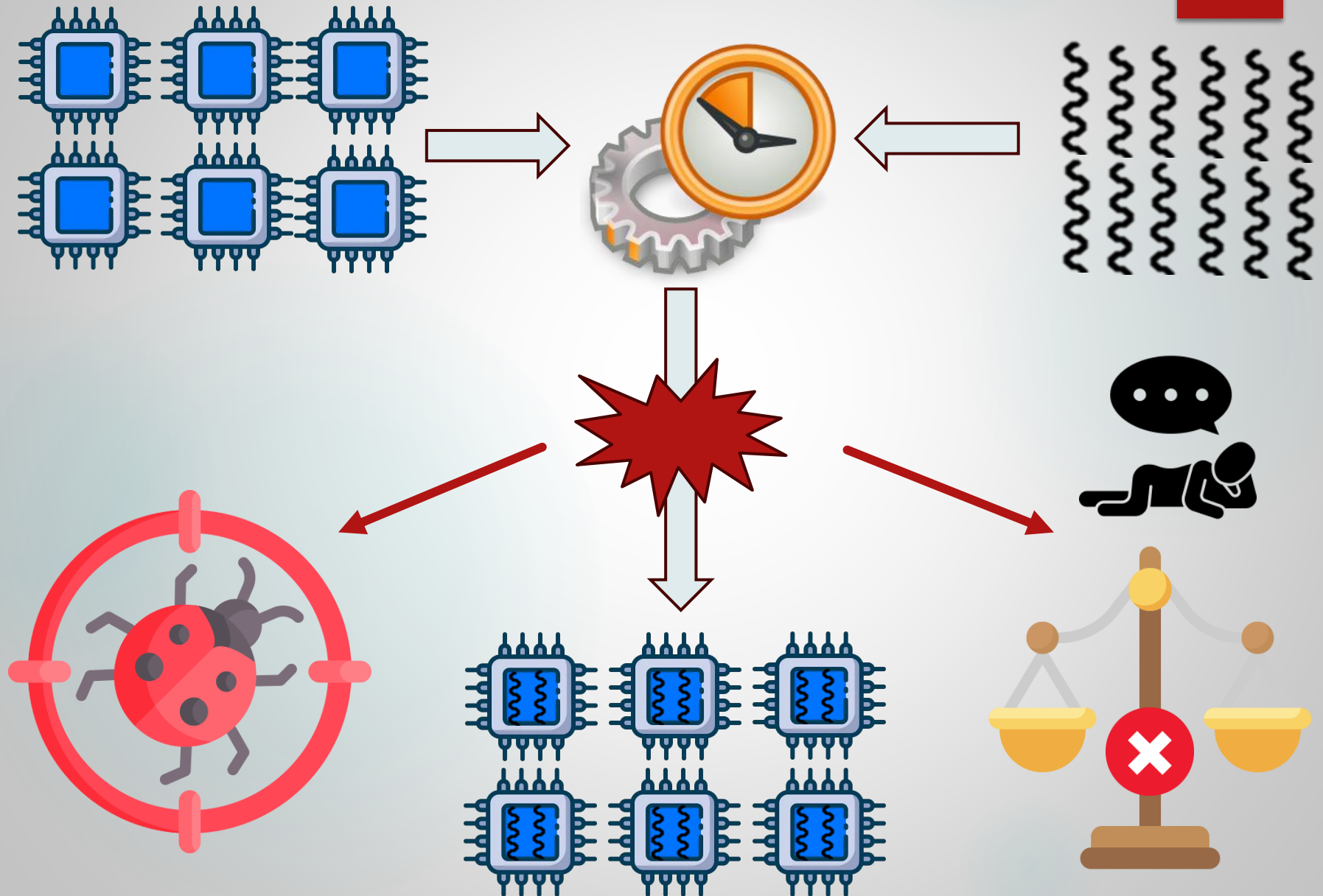
The Linux Scheduler: *A Decade of Wasted Cores*




Expectativa



Realidad



Objetivos del *paper*

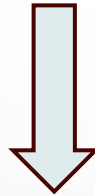
- 
- ***Razonar pérdidas de rendimiento observadas:***
 - Degradación considerable en tareas intensas en sincronización.
 - Compilaciones del *kernel* hasta un 13% más lentas.
 - Penalizaciones de incluso un 23% en *benchmarks* de BBDD (TPC-H).
 - ***Describir bugs encontrados en el planificador:***
 - Group Imbalance
 - Scheduling Group Construction
 - Overload-on-Wakeup
 - Missing Scheduling Domains
 - ***Desarrollo de herramientas.***
 - ***Plantear una solución:***
 - Valorar posibles parches.
 - Observar la diferencia de rendimiento.

Introducción

“And you have to realize that there are not very many things that have aged as well as the scheduler.”

Introducción

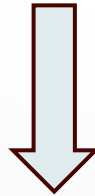
“And you have to realize that there are not very many things that have aged as well as the scheduler.”



“Which is just another proof that scheduling is easy.”

Introducción

“And you have to realize that there are not very many things that have aged as well as the scheduler.”



“Which is just another proof that scheduling is easy.”



Linus Torvalds, 2001

Introducción

La complejidad del planificador



Otorgar un *quantum* adecuado.

Gestionar efectivamente las colas.

Gestionar tareas por lotes e interactivas.

2004

Fin de la escala de Dennard (MOSFET)

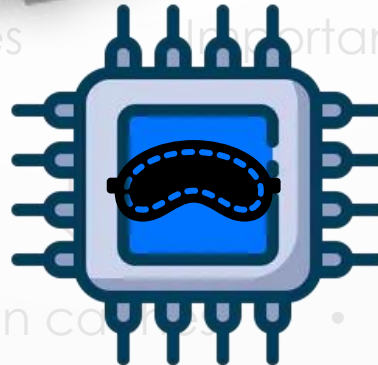
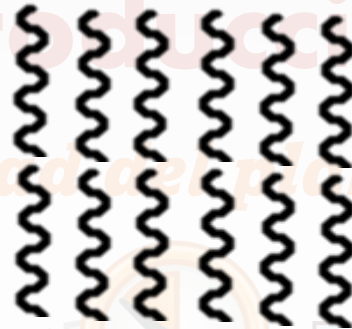
Mayor enfoque en procesadores
multinúcleo.

Importancia de la eficiencia
energética.

- NUMA
- Coherencia en cachés
- Velocidad memoria vs CPU

Introducción

La complejidad del problema



Otorgar un quantum adecuado para efectivamente las colas.

Gestionar las actividades.

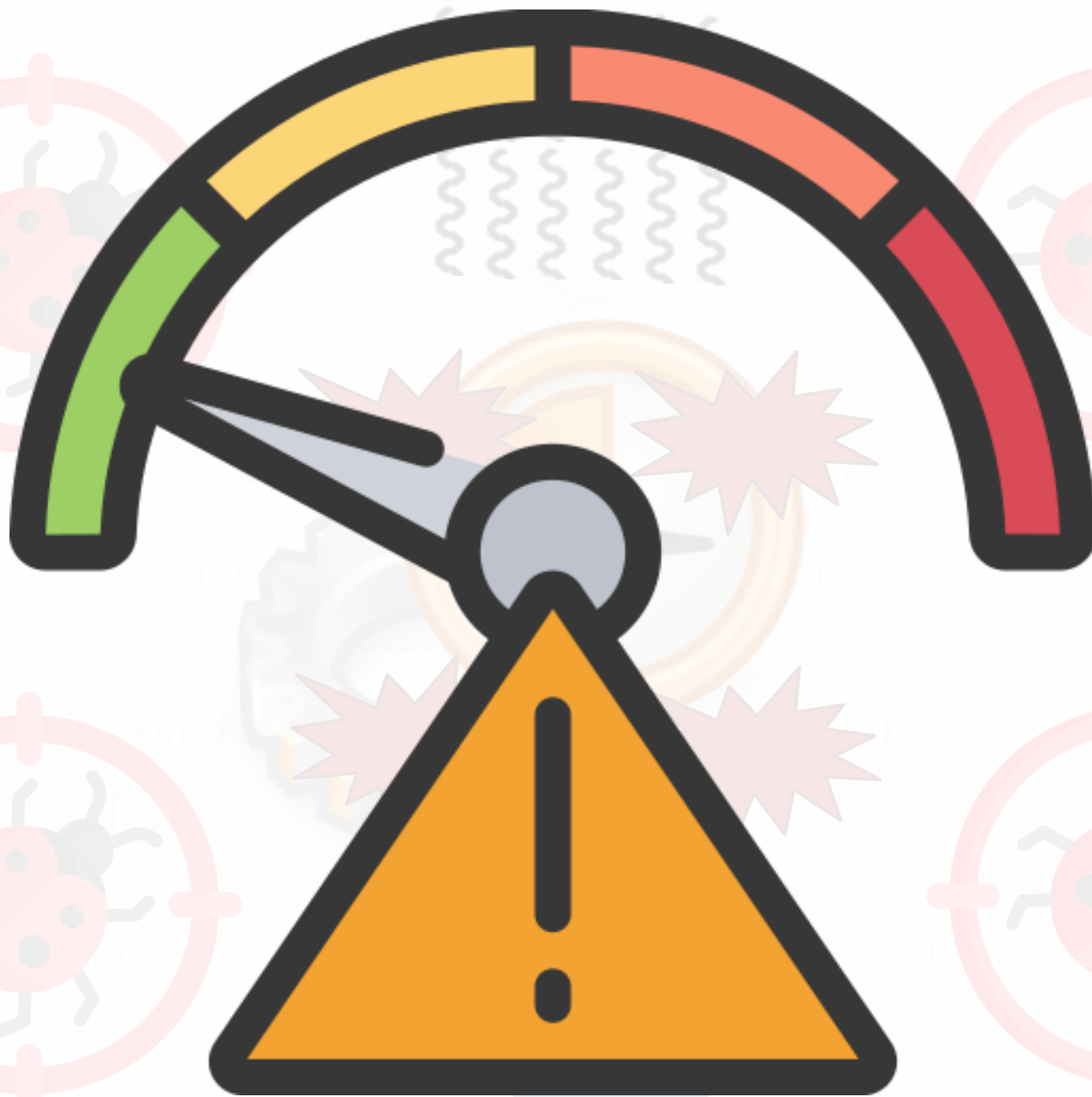
2004

Fin de la era del microprocesador (FET)

Mejor en procesadores
núcleo.

Importancia de la eficiencia
energética.

- NUMA
- Coherencia en caché
- Velocidad memoria vs CPU



Introducción

La detección es compleja



Crasheos o reinicios.

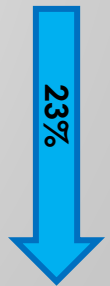
Congelamiento o cuelgues.

```
marko@pnep:~/linux-6.0.7$ make
SYNC      include/config/auto.conf.cmd
HOSTCC    scripts/kconfig/conf.o
HOSTLD    scripts/kconfig/conf
SYSHDR    arch/x86/include/generated/uapi/asm/unistd_32.h
SYSHDR    arch/x86/include/generated/uapi/asm/unistd_64.h
SYSHDR    arch/x86/include/generated/uapi/asm/unistd_x32.h
SYSTBL    arch/x86/include/generated/asm/syscalls_32.h
SYSHDR    arch/x86/include/generated/asm/unistd_32_ia32.h
SYSHDR    arch/x86/include/generated/asm/unistd_64_x32.h
SYSTBL    arch/x86/include/generated/asm/syscalls_64.h
HYPERCALLS arch/x86/include/generated/asm/xen-hypercalls.h
HOSTCC    arch/x86/tools/relocs_32.o
HOSTCC    arch/x86/tools/relocs_64.o
HOSTCC    arch/x86/tools/relocs_common.o
HOSTLD    arch/x86/tools/relocs
HOSTCC    scripts/genksyms/genksyms.o
YACC      scripts/genksyms/parse.tab.[ch]
```



Compilación de *kernel*

TPC[®]

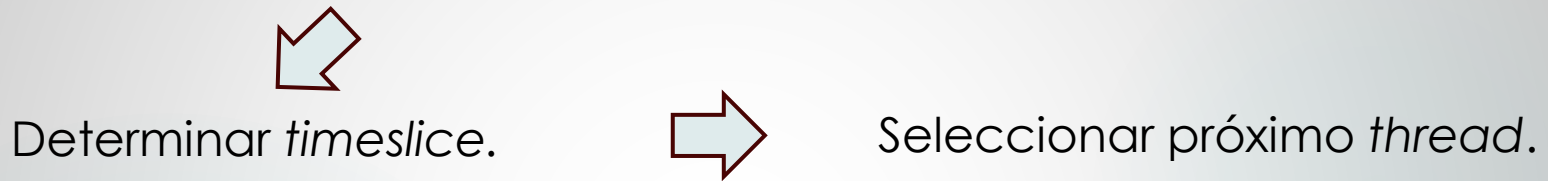


DB Queries (TPC-H)

El planificador de Linux

CFS

- Implementación de WFQ (Weighted Fair Queueing).
- Ciclos de CPU repartidos a los *threads* en base a sus pesos.



Mayor o menor en función del peso.

Peso \approx Prioridad

Todos lo ejecutan 1 vez al menos.

Aquel con menor *vruntime*.

$$vruntime = \frac{runtime}{peso}$$

El planificador de Linux

CFS

- Implementación de WFQ (Weighted Fair Queueing).
- Ciclos de CPU repartidos a los *threads* en base a sus pesos.

Determinar *timeslice*.



Seleccionar próximo *thread*.

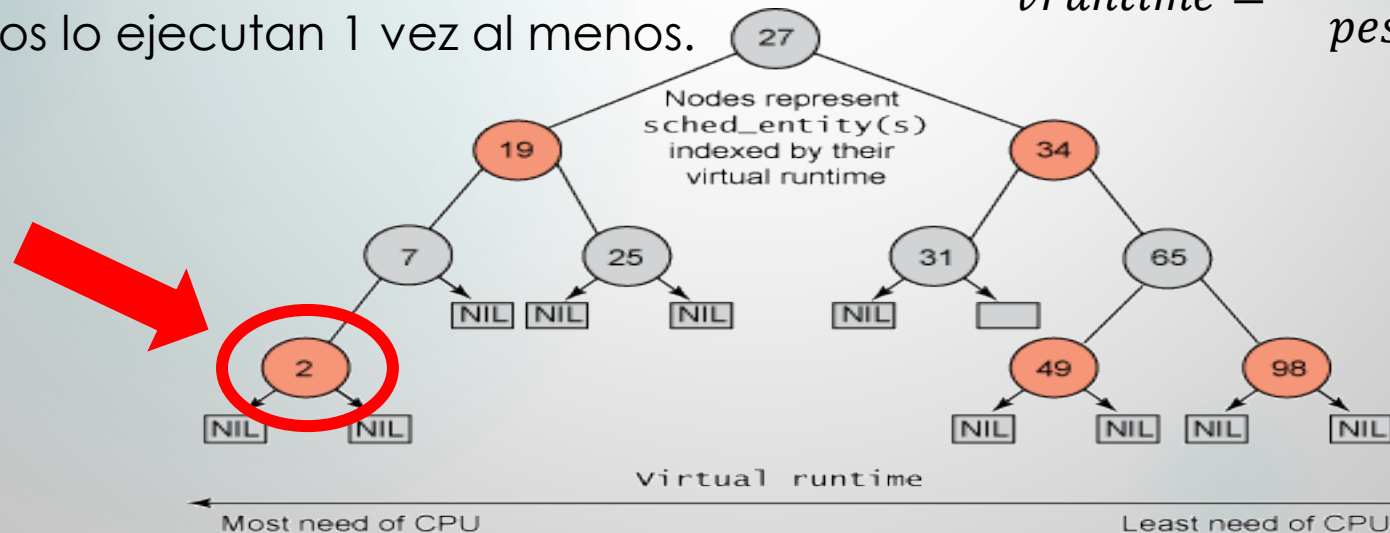
Mayor o menor en función del peso.

Aquel con menor *vruntime*.

Peso \approx Prioridad

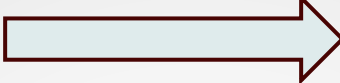
Todos lo ejecutan 1 vez al menos.

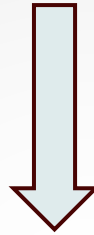
$$vruntime = \frac{runtime}{peso}$$



El planificador de Linux

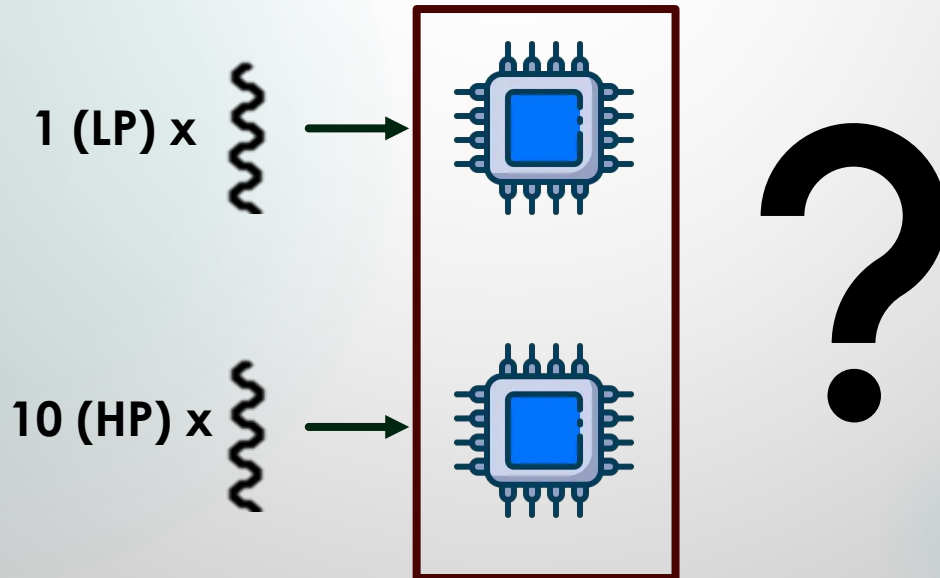
CFS (Multi-core)

- No existe una cola única.  Una cola para cada core.



Minimiza el coste de los cambios de contexto.

- Las colas de cada core deben encontrarse balanceadas.



El planificador de Linux

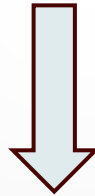
CFS (Multi-core)

“I suspect that making the scheduler use per-CPU queues together with some inter-CPU load balancing logic is probably trivial.”

El planificador de Linux

CFS (Multi-core)

“I suspect that making the scheduler use per-CPU queues together with some inter-CPU load balancing logic is probably trivial.”

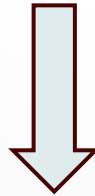


“Patches already exist, and I don’t feel that people can screw up the few hundred lines too badly.”

El planificador de Linux

CFS (Multi-core)

“I suspect that making the scheduler use per-CPU queues together with some inter-CPU load balancing logic is probably trivial.”



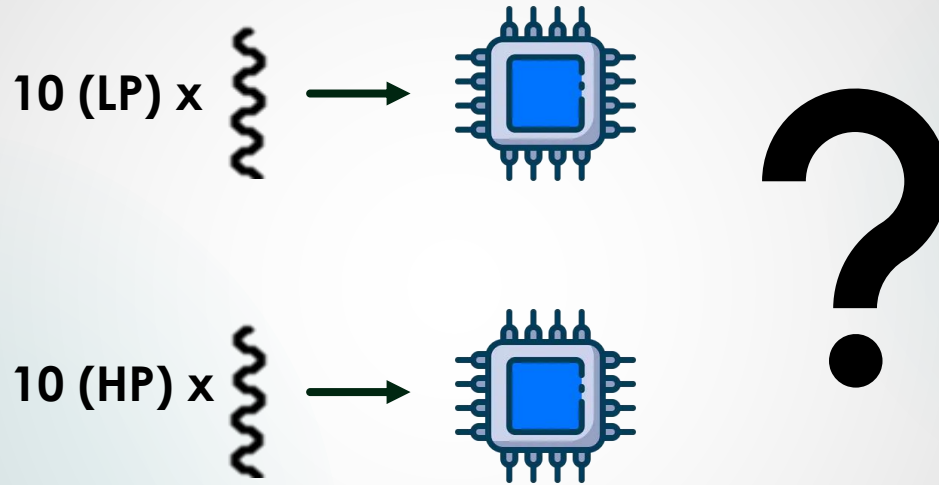
“Patches already exist, and I don’t feel that people can screw up the few hundred lines too badly.”

Linus Torvalds, 2001

El planificador de Linux

El algoritmo de balance

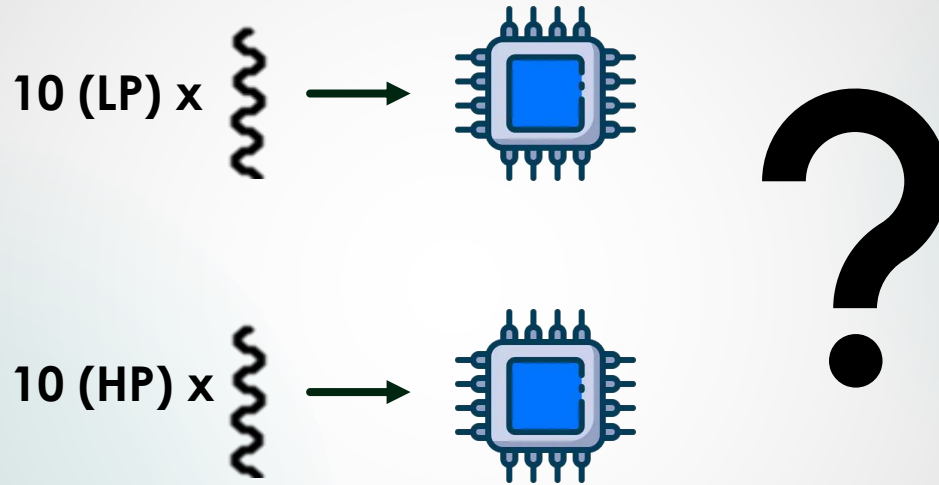
¿Es suficiente con tener en cuenta el número de *threads*?



El planificador de Linux

El algoritmo de balance

¿Es suficiente con tener en cuenta el número de *threads*?



Este planteamiento otorgaría el mismo tiempo de CPU
tanto a ambos tipos de *threads*.

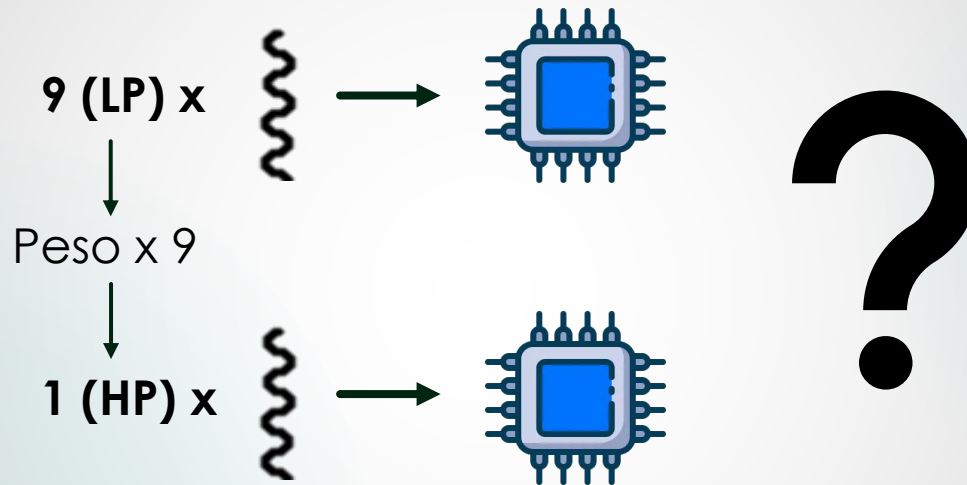


Es necesario tomar en cuenta el peso.

El planificador de Linux

El algoritmo de balance

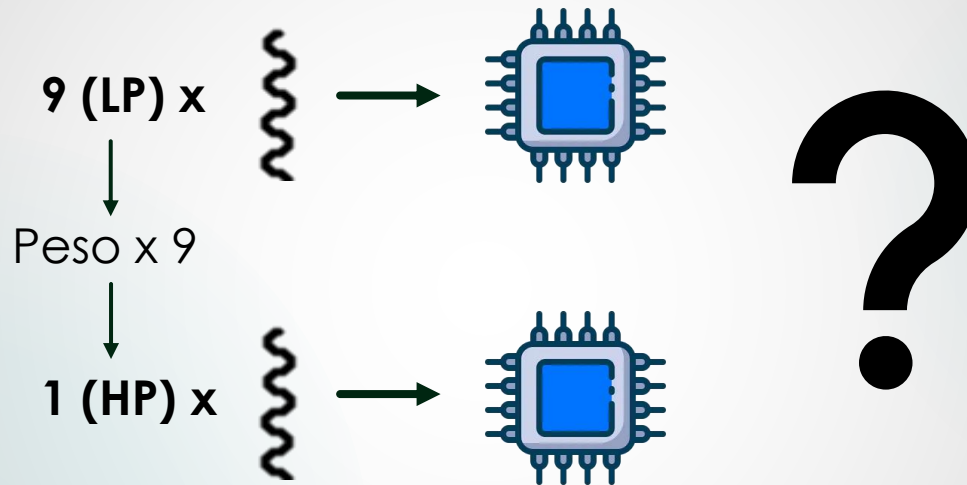
¿Entonces el peso como criterio es la solución?



El planificador de Linux

El algoritmo de balance

¿Entonces el peso como criterio es la solución?



Se determinaría que las colas están balanceadas, pero un solo *thread* consume 9 veces más tiempo de CPU.

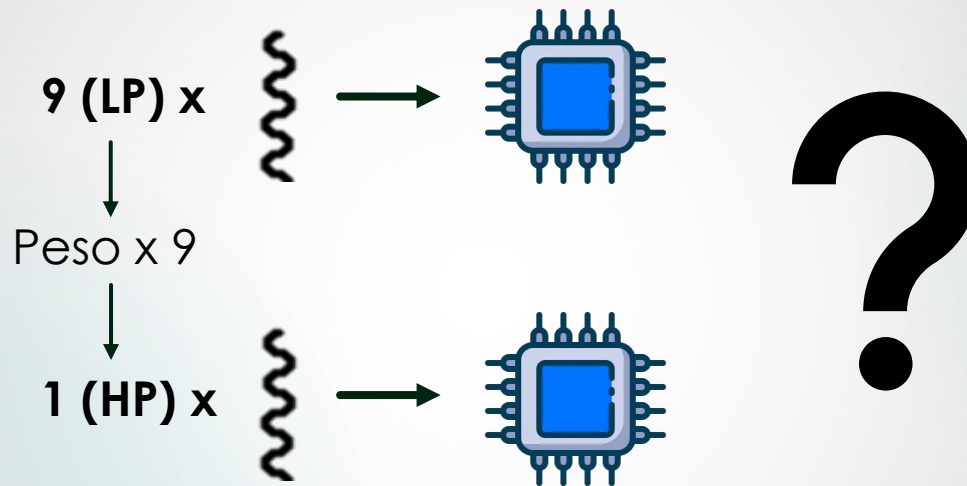


¿Y si el *thread* prioritario duerme periódicamente pudiendo así robar *threads* de la otra cola?

El planificador de Linux

El algoritmo de balance

¿Entonces el peso como criterio es la solución?



Se determinaría que las colas están balanceadas, pero un solo *thread* consume 9 veces más tiempo de CPU.



¿Y si el *thread* prioritario duerme periódicamente pudiendo así robar



threads de la otra cola?



El planificador de Linux

El algoritmo de balance

Métrica Load



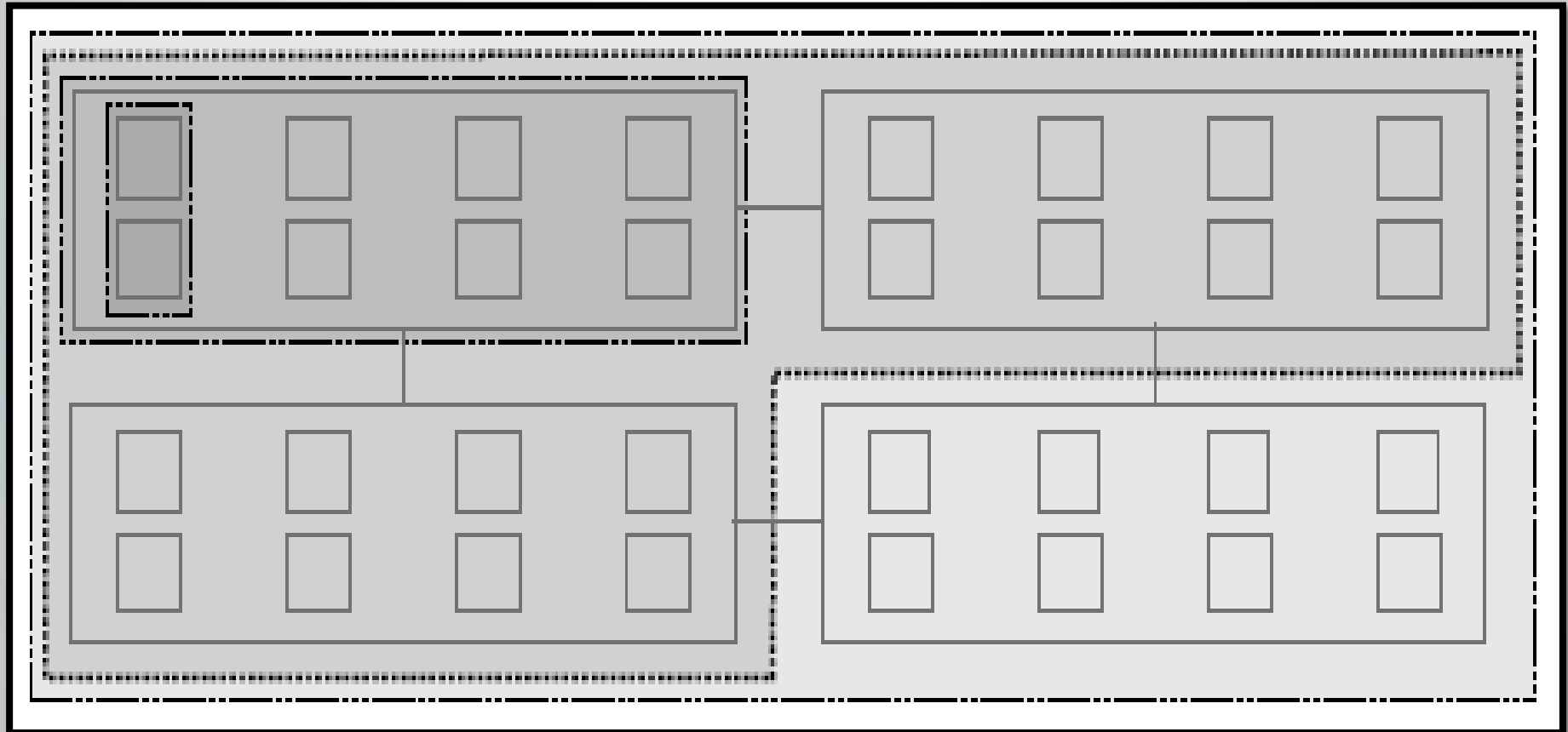
!

CGROUP

El planificador de Linux

El algoritmo de balance

NUMA y la necesidad de establecer dominios



El planificador de Linux

El algoritmo de balance

Algorithm 1 Simplified load balancing algorithm.

{Function running on each cpu *cur_cpu*:}

```
1: for all sd in sched_domains of cur_cpu do  
2:   if sd has idle cores then  
3:     first_cpu = 1st idle CPU of sd  
4:   else  
5:     first_cpu = 1st CPU of sd  
6:   end if  
7:   if cur_cpu  $\neq$  first_cpu then  
8:     continue  
9:   end if  
10:  for all sched_group sg in sd do  
11:    sg.load = average loads of CPUs in sg  
12:  end for
```

```
13:    busiest = overloaded sg with the highest load  
        (or, if nonexistent) imbalanced sg with highest load  
        (or, if nonexistent) sg with highest load  
14:    local = sg containing cur_cpu  
15:    if busiest.load  $\leq$  local.load then  
16:      continue  
17:    end if  
18:    busiest_cpu = pick busiest cpu of sg  
19:    try to balance load between busiest_cpu and cur_cpu  
20:    if load cannot be balanced due to tasksets then  
21:      exclude busiest_cpu, goto line 18  
22:    end if  
23:  end for
```

El planificador de Linux

Optimizaciones

❑ Evitar el trabajo duplicado:

1. Los cores reciben un *clock tick* y ejecutan el algoritmo de balance.
2. Se comprueba si existe algún core en idle y se asigna a aquel con menor número.
3. Si no existe ninguno en idle, se asigna de la misma forma a alguno de los que ya tienen carga.

❑ Ahorrar energía (> Linux 2.26.21):

1. Los cores pueden entrar en reposo sin despertar con un *clock tick*.
2. Si el resto de cores activos están sobrecargados, pueden despertar a uno de los cores ociosos, al que se le asigna la ejecución del algoritmo.

❑ Ejecución tras despertar un *thread*:

- 1a. Se intenta asignar a un core en idle.
- 1b. Si lo despertó otro *thread*, a un core con el que comparta caché.

Bugs investigados

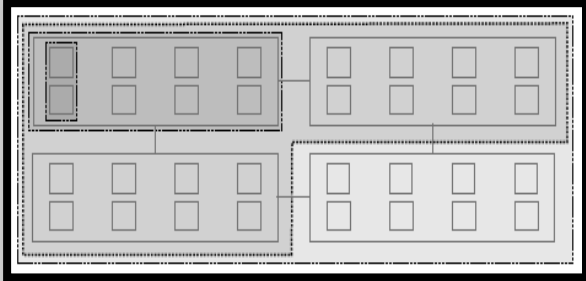
¿Un código perfecto a la primera?

**“Nobody actually creates perfect code the first time around, except me.
But there’s only one of me.”**

Linus Torvalds, 2007

Bugs investigados

Group Imbalance - Entorno



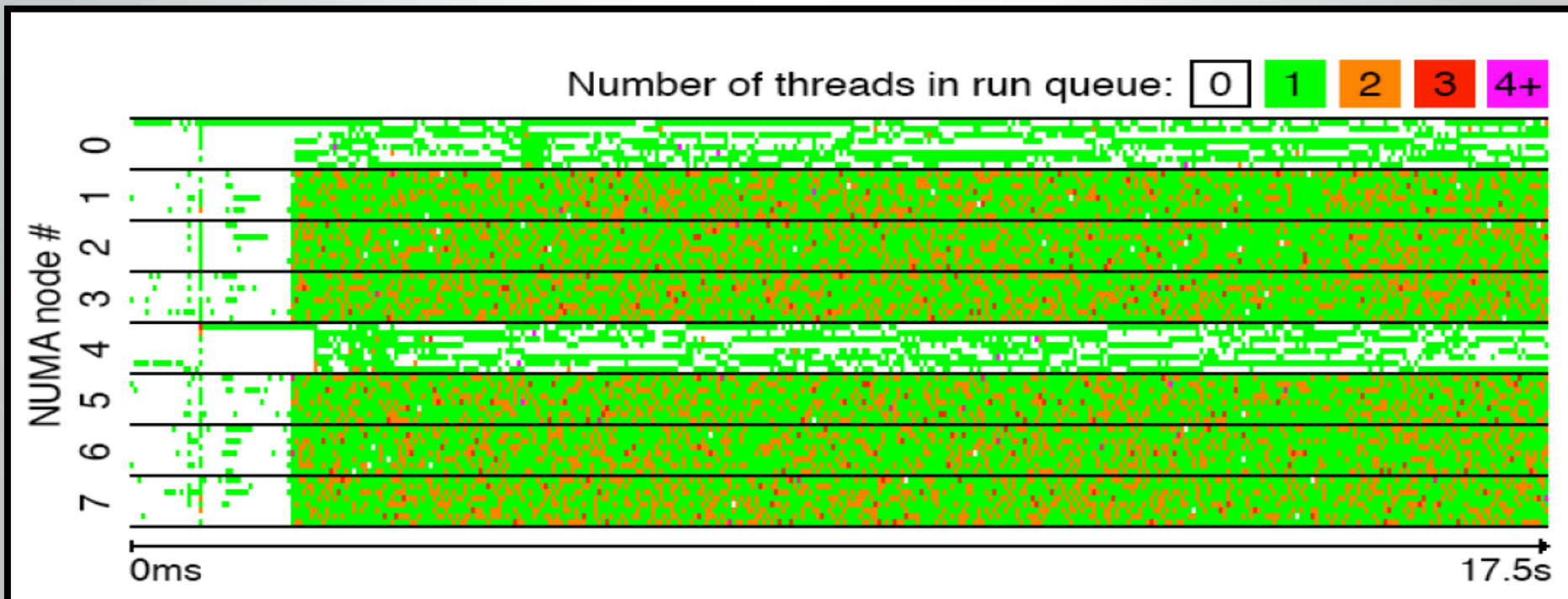
64 cores – 8 nodos NUMA



2 procesos x 1 thread



64 threads



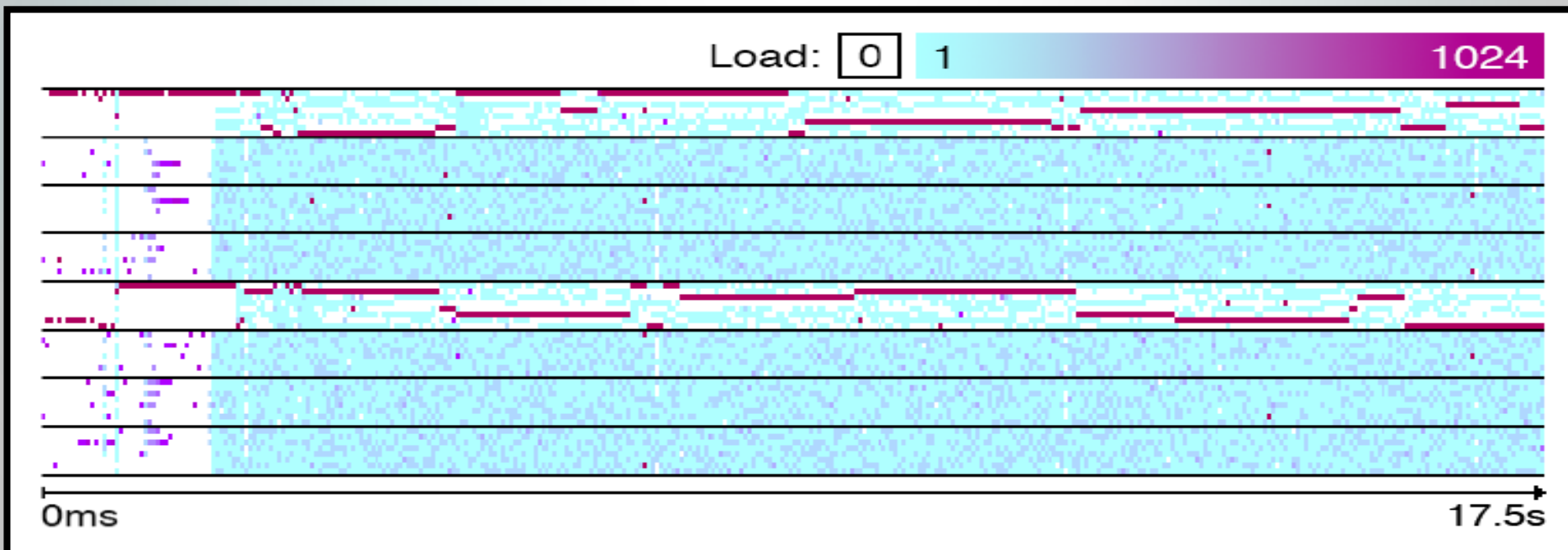
Bugs investigados

Group Imbalance - Causas

❑ Complejidad de la métrica “load”:

- *Load* = Combinación de peso y CPU requerida.
- Los *threads* dedicados a compilar el *kernel*, se encuentran agrupados para una misma tarea (*autogroup*) → La carga se reparte:

$64 \times \text{Load de 1 thread en make} \approx \text{Load de 1 thread en R}$

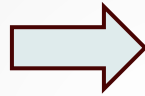


Bugs investigados

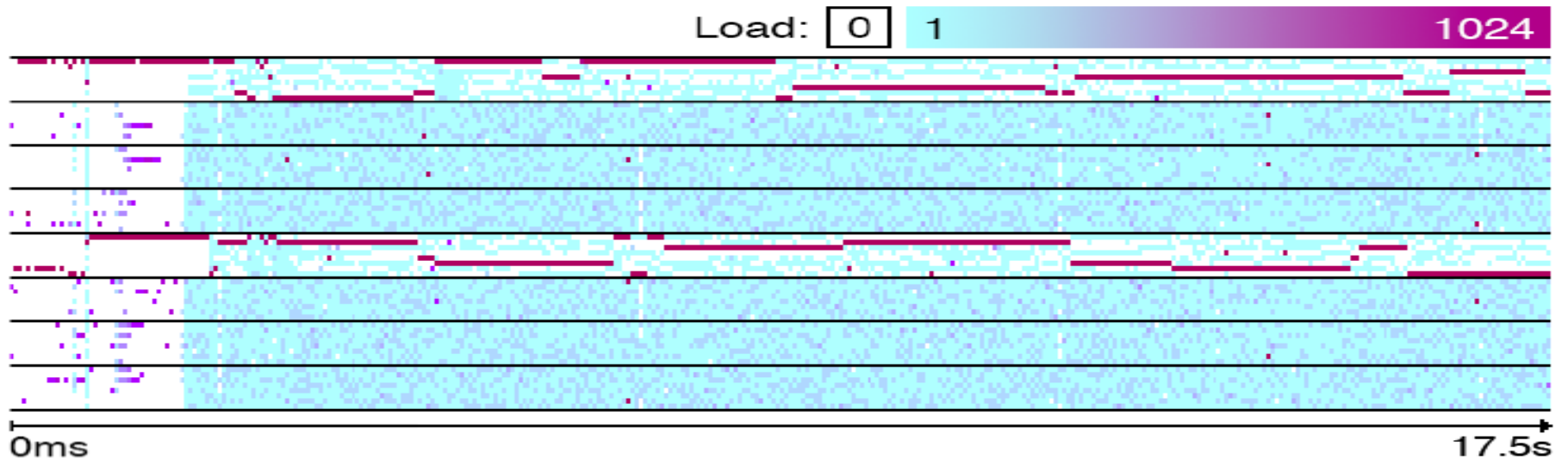
Group Imbalance - Causas

❑ Diseño jerárquico:

```
10: for all sched_group sg in sd do
11:   sg.load = average loads of CPUs in sg
12: end for
```



A los ojos del planificador, la media de carga para cada nodo es la misma aproximadamente.

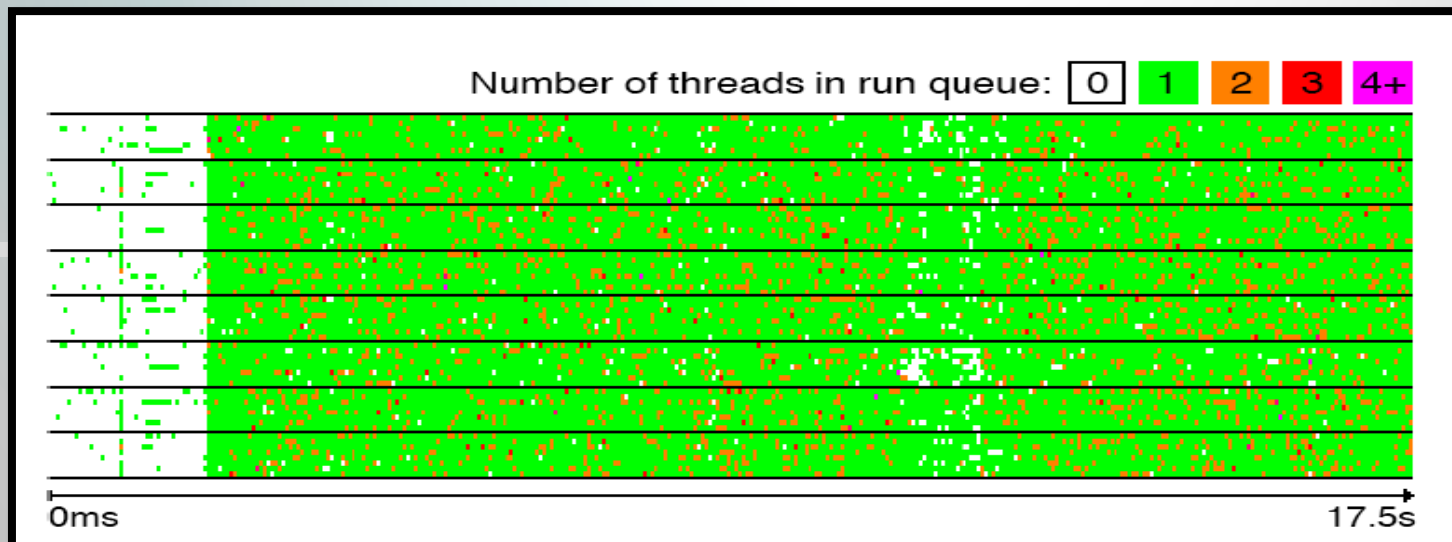


Bugs investigados

Group Imbalance - Solución

Reemplazar el uso de la media por la mínima.

- Mínima → *Load* del core menos cargado del grupo.
- No requiere una complejidad computacional mayor.
- Se siguen manteniendo las excepciones ante el uso de *tasksets*.
- No se observa un incremento de migraciones de un grupo a otro de vuelta, que pueda ocasionar problemas de rendimiento.



• **Make:**

-15%

• **R:**

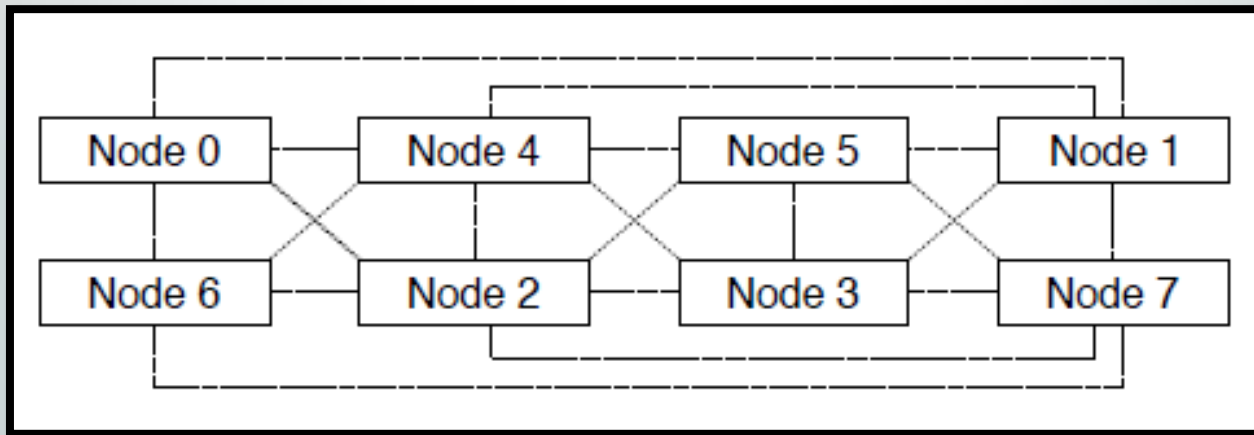
Se mantiene.

Bugs investigados

Scheduling Group Construction

❑ Uso del comando **taskset**:

- Permite correr una aplicación en unos cores determinados.
- Puede causar problemas de rendimiento en máquinas NUMA si los cores distan más de dos saltos entre sí (ej. : Nodos 1 y 2):



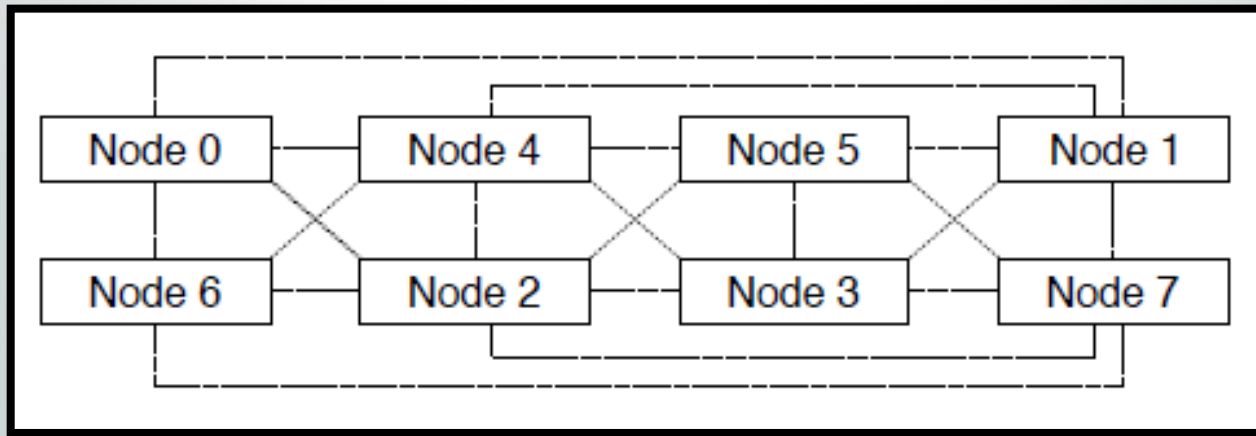
- Por defecto, los *threads* intentarán ejecutarse en el nodo del mismo *thread* que los creó. Pero ya no migrarán de ahí, independientemente de la saturación.

Bugs investigados

Scheduling Group Construction - Causa

❑ Falta de adaptación a máquinas NUMA:

- Los grupos de ejecución se construyen desde la perspectiva de un core concreto (el 0 en esta ocasión), en lugar del core planificador.



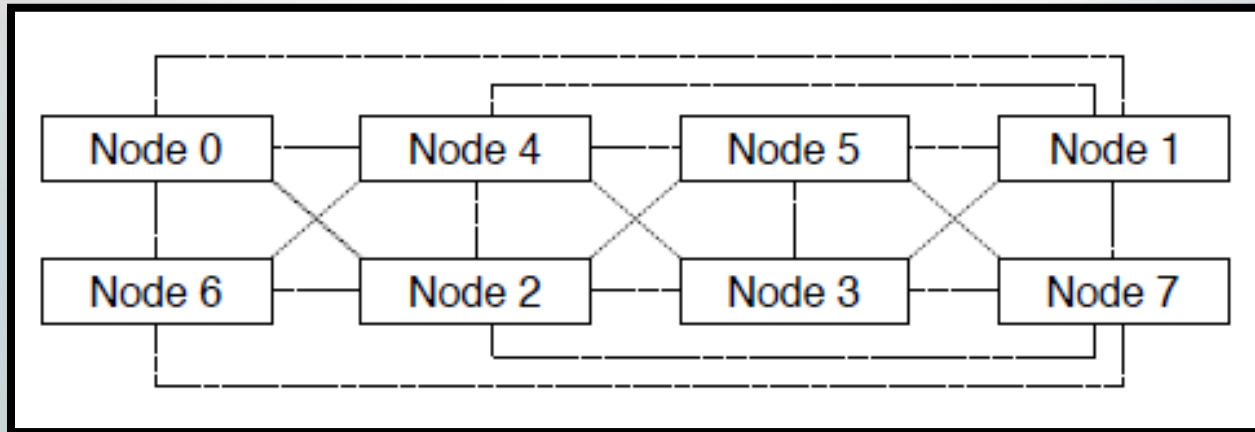
- Grupos generados, situándose el core en el Nodo 0, y el siguiente en el 3:
 - Grupo 1: Nodos 0, 1, 2, 4, 6.
 - Grupo 2: Nodos 1, 2, 3, 4, 5, 7.

Bugs investigados

Scheduling Group Construction - Causa

❑ Falta de adaptación a máquinas NUMA:

- Grupo 1: Nodos 0, 1, 2, 4, 6.
- Grupo 2: Nodos 1, 2, 3, 4, 5, 7.



- Los nodos 1 y 2 aparecen en ambos grupos, aunque los separan 2 saltos.
- ¿Y si una aplicación se designa a ambos nodos?:
 1. Se crean los *threads* en el mismo core del padre, por ejemplo, en Nodo 1.
 2. Cuando un core en Nodo 2 ejecute el balanceo, dado que el Nodo 1 está en ambos *scheduling groups*, no observará diferencias de carga.

Bugs investigados

Scheduling Group Construction - Solución

❑ **Modificar el criterio para construir los scheduling groups:**

- Cada core emplea grupos contruidos desde su perspectiva.

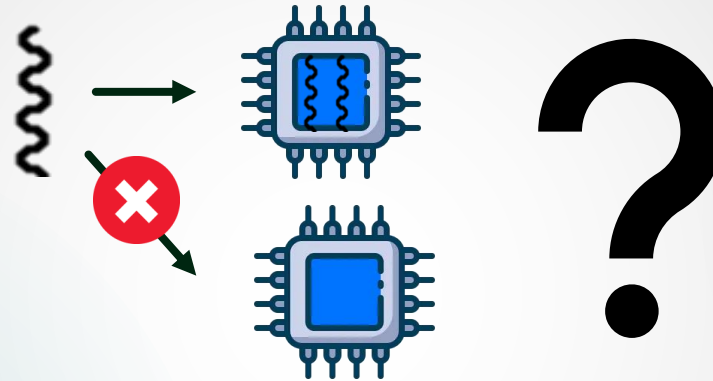
| Application | Time w/ bug (sec) | Time w/o bug (sec) | Speedup factor (×) |
|-------------|----------------------|-----------------------|-----------------------|
| bt | 99 | 56 | 1.75 |
| cg | 42 | 15 | 2.73 |
| ep | 73 | 36 | 2 |
| ft | 96 | 50 | 1.92 |
| is | 271 | 202 | 1.33 |
| lu | 1040 | 38 | 27 |
| mg | 49 | 24 | 2.03 |
| sp | 31 | 14 | 2.23 |
| ua | 206 | 56 | 3.63 |

Table 1: Execution time of NAS applications with the Scheduling Group Construction bug and without the bug. All applications are launched using `numactl --cpunodebind=1,2 <app>`.

Bugs investigados

Overload-on-Wakeup

❑ Introducido en el código `select_task_rq_fair` para despertar threads:



Si el *thread* es despertado por otro *thread* del mismo nodo en el que se durmió, no se consideran otros nodos en la planificación.



Se busca optimizar la caché y la reutilización de sus datos.



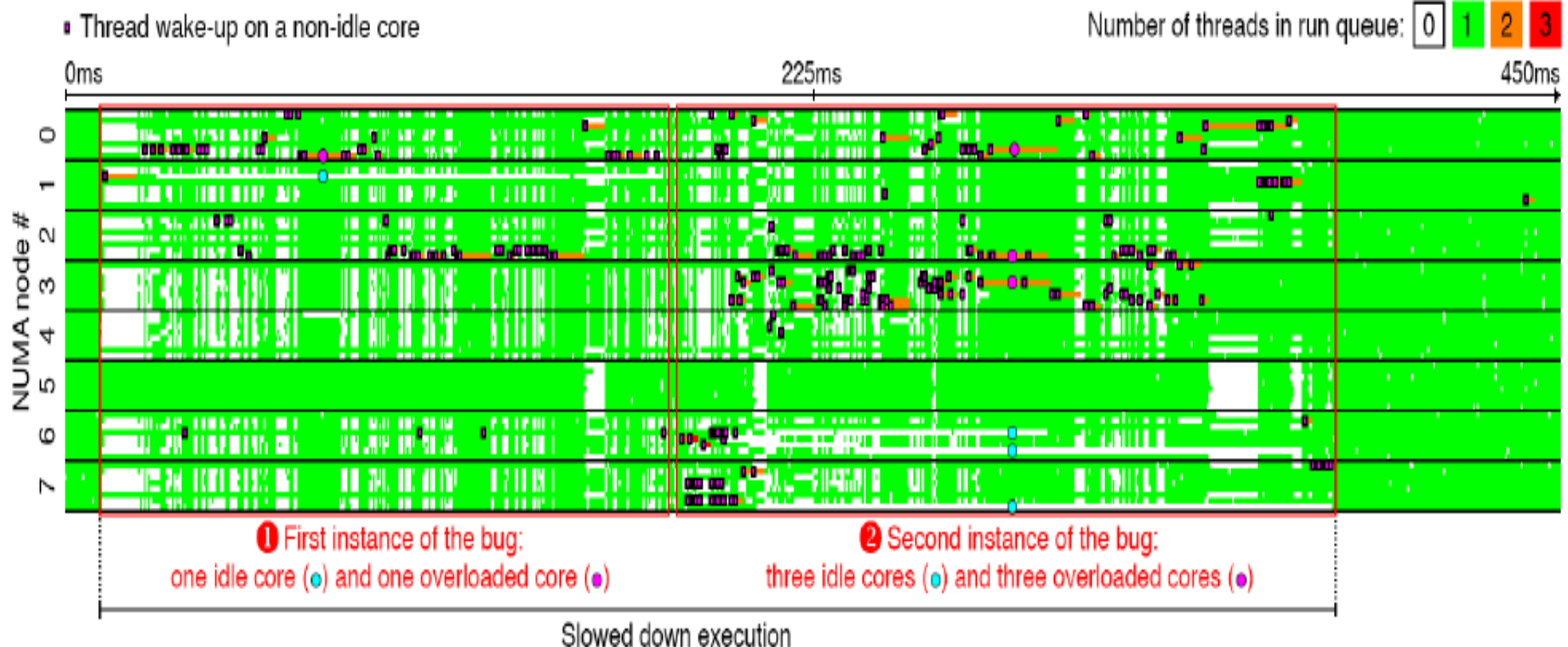
Pero puede ser un problema si todos los cores del nodo están ocupados.

Bugs investigados

Overload-on-Wakeup – Reproducción

- ❑ Usar 64 threads en TPC-H junto con *threads* efímeros para tareas del sistema:

Para reproducir el *bug*, se deshabilitan los *autogroups*, para evitar la influencia del *bug* anterior.



Bugs investigados

Overload-on-Wakeup – Paso por paso

1. Uno de los *threads* temporales es planificado en un core que ejecuta *threads* de la base de datos.
2. El planificador detecta el Nodo A como uno más cargado, migrando uno de los *threads* a otro (Nodo B).
3. Si el *thread* migrado es el asignado a tareas de la base de datos, aparece el *bug*.
4. Ahora el Nodo B, cuenta con más de un *thread* pesado. Además, estos siempre duermen y vuelven a despertar en el mismo nodo.
5. El Nodo A finaliza el *thread* temporal y queda ocioso, pasando varios milisegundos hasta que el planificador es capaz de detectar la situación y recuperarse.

Bugs investigados

Overload-on-Wakeup - Solución

❑ **Modificar el criterio que determina en qué core despiertan los *threads*:**

1. Asignar en el mismo core donde despertó la última vez si está en idle.
2. Si está ocupado, despertar el *thread* en el core que lleve más tiempo en idle.



| Bug fixes | TPC-H request #18 | Full TPC-H benchmark |
|---------------------------|-------------------|----------------------|
| None | 55.9s | 542.9s |
| <i>Group Imbalance</i> | 48.6s (−13.1%) | 513.8s (−5.4%) |
| <i>Overload-on-Wakeup</i> | 43.5s (−22.2%) | 471.1s (−13.2%) |
| Both | 43.3s (−22.6%) | 465.6s (−14.2%) |

Bugs investigados

Missing Scheduling Domains

- ❑ **Aparece al desactivar alguno de los cores a través de la interfaz /proc:**

Una variable global que representa el número de dominios de planificación se actualiza incorrectamente, bloqueando el balance entre nodos NUMA.



La variable toma como valor el número de dominios dentro de un nodo NUMA.



El algoritmo de balance deja de funcionar correctamente.



Los *threads* se ejecutan únicamente en el nodo previo a la desactivación, o en aquel al que pertenece el padre, si son creados posteriormente.

Bugs investigados

Overload-on-Wakeup - Solución

- ❑ **Permitir la regeneración de dominios de planificación a través de los nodos:**

Es posible recuperando una función especializada en ello, pero eliminada por los desarrolladores en procesos de refactorización del código.

| Application | Time w/ bug (sec) | Time w/o bug (sec) | Speedup factor (×) |
|-------------|----------------------|-----------------------|-----------------------|
| bt | 122 | 23 | 5.24 |
| cg | 134 | 5.4 | 24.90 |
| ep | 72 | 18 | 4.0 |
| ft | 110 | 14 | 7.69 |
| is | 283 | 53 | 5.36 |
| lu | 2196 | 16 | 137.59 |
| mg | 81 | 9 | 9.03 |
| sp | 109 | 12 | 9.06 |
| ua | 906 | 14 | 64.27 |

Bugs investigados

Resumen

| Name | Description | Kernel version | Impacted applications | Maximum measured performance impact |
|--------------------------------------|--|----------------|---------------------------------|-------------------------------------|
| <i>Group Imbalance</i> | When launching multiple applications with different thread counts, some CPUs are idle while other CPUs are overloaded. | 2.6.38+ | All | 13× |
| <i>Scheduling Group Construction</i> | No load balancing between nodes that are 2-hops apart | 3.9+ | All | 27× |
| <i>Overload-on-Wakeup</i> | Threads wake up on overloaded cores while some other cores are idle. | 2.6.32+ | Applications that sleep or wait | 22% |
| <i>Missing Scheduling Domains</i> | The load is not balanced between NUMA nodes | 3.19+ | All | 138× |

Table 4: Bugs found in the scheduler using our tools.

Bugs investigados

Conclusiones



Bugs investigados

Conclusiones



No es la mejor alternativa a largo plazo: Cambios constantes y gran número de desarrolladores. ¿Y una remodelación o actualización?

Bugs investigados

Conclusiones



No es la mejor alternativa a largo plazo: Cambios constantes y gran número de desarrolladores. ¿Y una remodelación o actualización?



Herramientas

Online Sanity Checker

Algorithm 2 “No core remains idle while another core is overloaded”

```
1: for all CPU1 in CPUs do
2:   if CPU1.nr_running  $\geq 1$  {CPU1 is not idle} then
3:     continue
4:   end if
5:   for all CPU2 in CPUs do
6:     if CPU2.nr_running  $\geq 2$  and can_steal(CPU1, CPU2) then
7:       Start monitoring thread operations
8:     end if
9:   end for
10: end for
```

S → Periodo fijado para realizar las
comprobaciones (1s aprox.).

M → Periodo fijado para determinar si es
un incumplimiento preocupante (100ms)

Si el bug se detecta, comienza a realizar el muestreo con *systemtap* unos 20ms, para evitar un *overhead* excesivo.

Herramientas

[...]



| PID | UID | DEV | XMIT_PK | RECV_PK | XMIT_KB | RECV_KB | COMMAND |
|-------|-----|------|---------|---------|---------|---------|-----------------|
| 0 | 0 | eth0 | 0 | 5 | 0 | 0 | swapper |
| 11178 | 0 | eth0 | 2 | 0 | 0 | 0 | synergyc |
| PID | UID | DEV | XMIT_PK | RECV_PK | XMIT_KB | RECV_KB | COMMAND |
| 2886 | 4 | eth0 | 79 | 0 | 5 | 0 | cups-polld |
| 11362 | 0 | eth0 | 0 | 61 | 0 | 5 | firefox |
| 0 | 0 | eth0 | 3 | 32 | 0 | 3 | swapper |
| 2886 | 4 | lo | 4 | 4 | 0 | 0 | cups-polld |
| 11178 | 0 | eth0 | 3 | 0 | 0 | 0 | synergyc |
| PID | UID | DEV | XMIT_PK | RECV_PK | XMIT_KB | RECV_KB | COMMAND |
| 0 | 0 | eth0 | 0 | 6 | 0 | 0 | swapper |
| 2886 | 4 | lo | 2 | 2 | 0 | 0 | cups-polld |
| 11178 | 0 | eth0 | 3 | 0 | 0 | 0 | synergyc |
| 3611 | 0 | eth0 | 0 | 1 | 0 | 0 | Xorg |
| PID | UID | DEV | XMIT_PK | RECV_PK | XMIT_KB | RECV_KB | COMMAND |
| 0 | 0 | eth0 | 3 | 42 | 0 | 2 | swapper |
| 11178 | 0 | eth0 | 43 | 1 | 3 | 0 | synergyc |
| 11362 | 0 | eth0 | 0 | 7 | 0 | 0 | firefox |
| 3897 | 0 | eth0 | 0 | 1 | 0 | 0 | multiload-apple |

[...]

Herramientas

Scheduler Visualization Tool

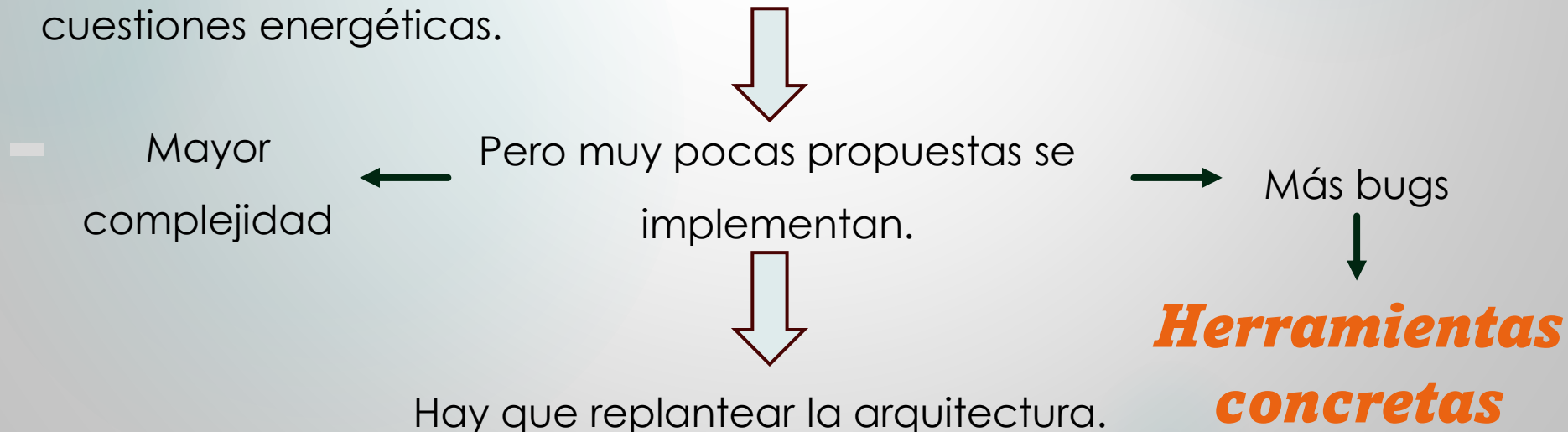


- Permite mostrar la actividad realizada relacionada con la planificación.
- Perfila y muestra concretamente:
 - El tamaño de las colas.
 - La carga (load) total de cada cola.
 - Cores considerados en cada balanceo y wakes.
- No realiza un muestreo continuo y se limita a la detección de cambios.
- Toda la información se almacena en un *array* global para reducir el *overhead*.
- Esto implica modificar algunas funciones del kernel para registrar los cambios:
 - *add_nr_running* y *sub_nr_running*: Tamaño de cada cola.
 - *account_entity_enqueue...* : Load de cada cola.
 - *select_idle_sibling...* : Registrar si el core se empleó (1) o no (0).

Lecciones

El planteamiento del scheduler

- El *scheduler*, hoy en día, debe de tener en cuenta aspectos relacionados con la gestión de una memoria más compleja y el consumo energético.
- Resulta complejo seguir entendiendo el *scheduler* como una pieza simple y aislada del *kernel*. → Complica el código.
- Existen varios *papers* abordando problemáticas similares, como el hecho de tener en cuenta la separación en nodos NUMA o el apagado de cores por cuestiones energéticas.

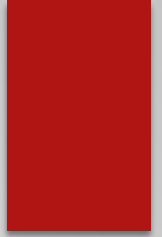


Evaluación

Experiencia del revisor y comentarios

- **Experiencia previa:** Media
- **Comentarios positivos:**
 - ☐ Simplificación de algunos algoritmos para gente no muy experimentada.
 - ☐ Desarrollo de herramientas portables y descripción clara de su funcionamiento.
 - ☐ Aporte de ejemplos sencillos de comprender sobre los diferentes *bugs*.
 - ☐ Aporte de métricas detalladas.
- **Comentarios negativos:**
 - ☐ En algunas ocasiones, si bien no son muchas, aparecen conceptos propios del planificador difíciles de entender (*nr_running*).
 - ☐ No se contemplan avances más recientes, como los cores de alto y bajo consumo.
 - ☐ No se aclara qué versiones del *kernel* se emplean.
 - ☐ Casos ideales.

Evaluación



¿Decisión final?

Evaluación

¿Decisión final?

ACEPTADO

Evaluación

¿Decisión final?

ACEPTADO

¡Con matices!

The Linux Scheduler: *A Decade of Wasted Cores*

