# Towards a Tool for Access Affinity Based Structure Reordering in the Linux Kernel

Aboorva Devarajan
Madadi Vineeth Reddy
(Linux Technology Center, IBM Systems Development Labs)

# An idea, a method, and a measurement

# Fields accessed close together in time should live close together in memory.

Everything in this talk is built upon this.

# Observation: Perf profile

```
Samples: 13M of event 'cycles', Event count (approx.): 14722905193674
Overhead   Command          Shared Object                     Symbol
  37.72%   workload         [kernel.vmlinux]                  [k] queued_spin_lock_slowpath
  24.17%   swapper          [kernel.vmlinux]                  [k] __ppc64_runlatch_off
   7.60%   workload         [kernel.vmlinux]                  [k] update_sg_lb_stats
   1.26%   workload         [kernel.vmlinux]                  [k] idle_cpu
   1.11%   workload         [kernel.vmlinux]                  [k] _raw_spin_lock_irq
   1.05%   workload         [kernel.vmlinux]                  [k] _raw_spin_lock
   1.01%   workload         [kernel.vmlinux]                  [k] _raw_read_lock
   0.73%   workload         [kernel.vmlinux]                  [k] __schedule
   0.69%   workload         [kernel.vmlinux]                  [k] update_cfs_group
   0.68%   swapper          [kernel.vmlinux]                  [k] update_cfs_group
   0.67%   swapper          [kernel.vmlinux]                  [k] _raw_spin_lock
   0.64%   workload         [kernel.vmlinux]                  [k] do_dec_rlimitt_put_ucounts
   0.60%   workload         [kernel.vmlinux]                  [k] _find_next_and_bit
   0.59%   workload         [kernel.vmlinux]                  [k] inc_rlimitt_get_ucounts
   0.51%   swapper          [unknown]                         [H] 0x000000000002aeddc
   0.50%   workload         [kernel.vmlinux]                  [k] find_busiest_queue
   0.50%   swapper          [kernel.vmlinux]                  [k] __schedule
   0.49%   workload         [kernel.vmlinux]                  [k] _raw_spin_lock_irqsave
   0.47%   workload         [kernel.vmlinux]                  [k] do_wait
```

# Observation: Perf profile

```
Samples: 13M of event 'cycles', Event count (approx.): 14722905193674
Overhead   Command          Shared Object                    Symbol
  37.72%   workload         [kernel.vmlinux]                 [k] queued_spin_lock_slowpath
  24.17%   swapper          [kernel.vmlinux]                 [k] __ppc64_runlatch_off
   7.60%   workload         [kernel.vmlinux]                 [k] update_sg_lb_stats
   1.26%   workload         [kernel.vmlinux]                 [k] idle_cpu
   1.11%   workload         [kernel.vmlinux]                 [k] _raw_spin_lock_irq
   1.05%   workload         [kernel.vmlinux]                 [k] _raw_spin_lock
   1.01%   workload         [kernel.vmlinux]                 [k] _raw_read_lock
   0.73%   workload         [kernel.vmlinux]                 [k] __schedule
   0.69%   workload         [kernel.vmlinux]                 [k] update_cfs_group
   0.68%   swapper          [kernel.vmlinux]                 [k] update_cfs_group
   0.67%   swapper          [kernel.vmlinux]                 [k] _raw_spin_lock
   0.64%   workload         [kernel.vmlinux]                 [k] do_dec_rlimitt_put_ucounts
   0.60%   workload         [kernel.vmlinux]                 [k] _find_next_and_bit
   0.59%   workload         [kernel.vmlinux]                 [k] inc_rlimitt_get_ucounts
   0.51%   swapper          [unknown]                        [H] 0x000000000002aeddc
   0.50%   workload         [kernel.vmlinux]                 [k] find_busiest_queue
   0.50%   swapper          [kernel.vmlinux]                 [k] __schedule
   0.49%   workload         [kernel.vmlinux]                 [k] _raw_spin_lock_irqsave
   0.47%   workload         [kernel.vmlinux]                 [k] do_wait
```

TOP HOT FUNCTION: update_sg_lb_stats

# Why is update_sg_lb_stats() a Top-3 CPU consumer?

~7.6% of CPU samples - a load-balancing helper, not application code

1. **Timer fires, or a CPU goes idle**
   *the trigger for load balancing*
   A scheduler tick or a CPU about to idle kicks off a balance pass.
2. **`sched_balance_rq()`**
   *load balancer invoked*
   Entry point of the scheduler's per-domain load balancer.
3. **`update_sg_lb_stats()`**
   **~7.6% of samples in perf profile**
   Collects load stats for every sched_group / CPU - runqueue load, capacity, utilization, idle count.
4. **Identify the busiest CPU / group**
   *find the imbalance*
   Analyses the collected stats to locate the busiest group.
5. **Pull tasks if an imbalance exists**
   *rebalance the runqueues*
   Migrates tasks from the busiest CPU to idle / under-utilized CPUs.

### What is a sched_group?

A set of CPUs in one scheduling domain - an SMT core, an LLC / CCX cluster, or a NUMA node, stats are aggregated at the group level.

### What does update_sg_lb_stats() do?

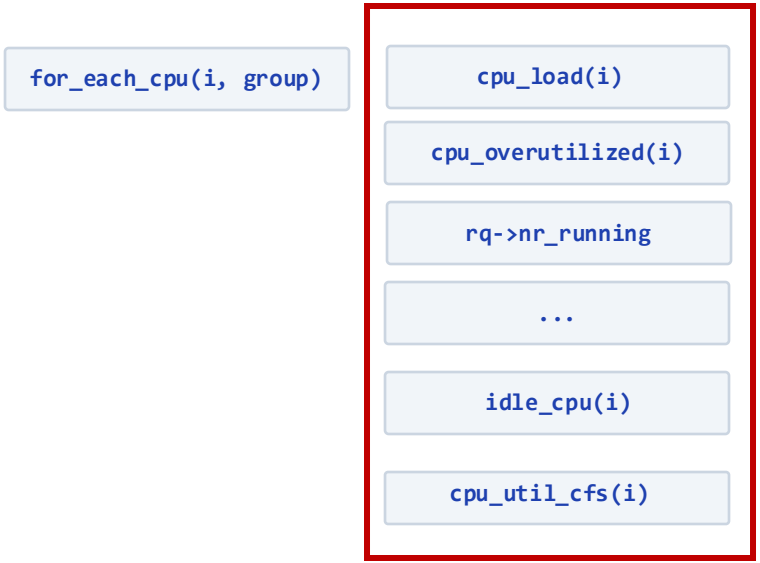
Walks every CPU (runqueue) in a sched_group, gathering load metrics. It does not move tasks it produces the stats that drive the balancer's migration decisions.

**TAKEAWAY** update_sg_lb_stats() tops the profile because the system spends significant time measuring and analysing load to make task-migration decisions. (Note: this is improved in recent kernel: sched/fair: Proportional newidle balance)

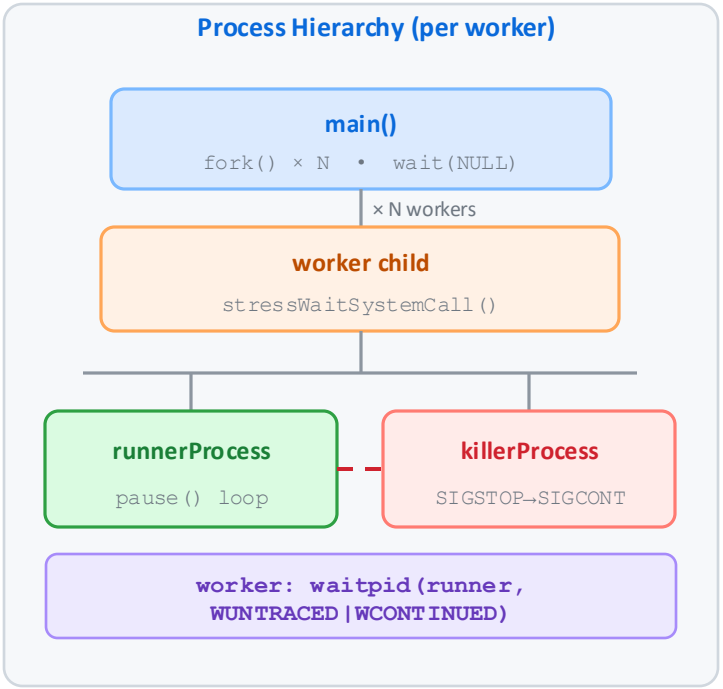
*This is kernel scheduler overhead (load balancing) - not application execution.*

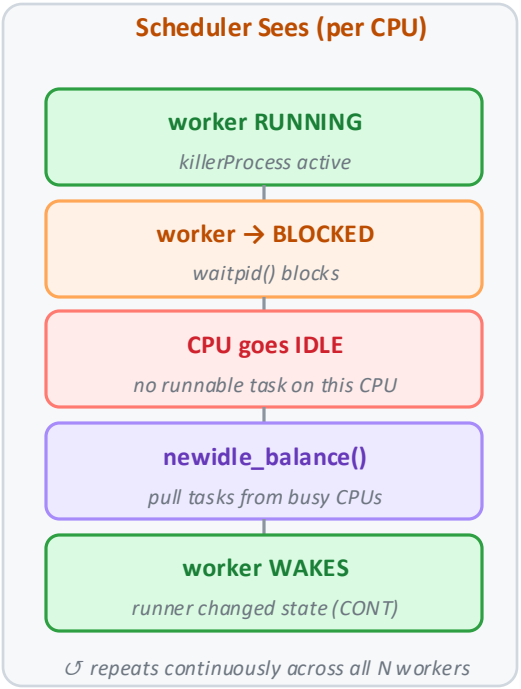
# update_sg_lb_stats: cache-misses

Inside `update_sg_lb_stats()`, each sched_group access involves fields from the large per-CPU runqueue structure, making it susceptible to cache misses.



# wait_stressor: scheduler load balance

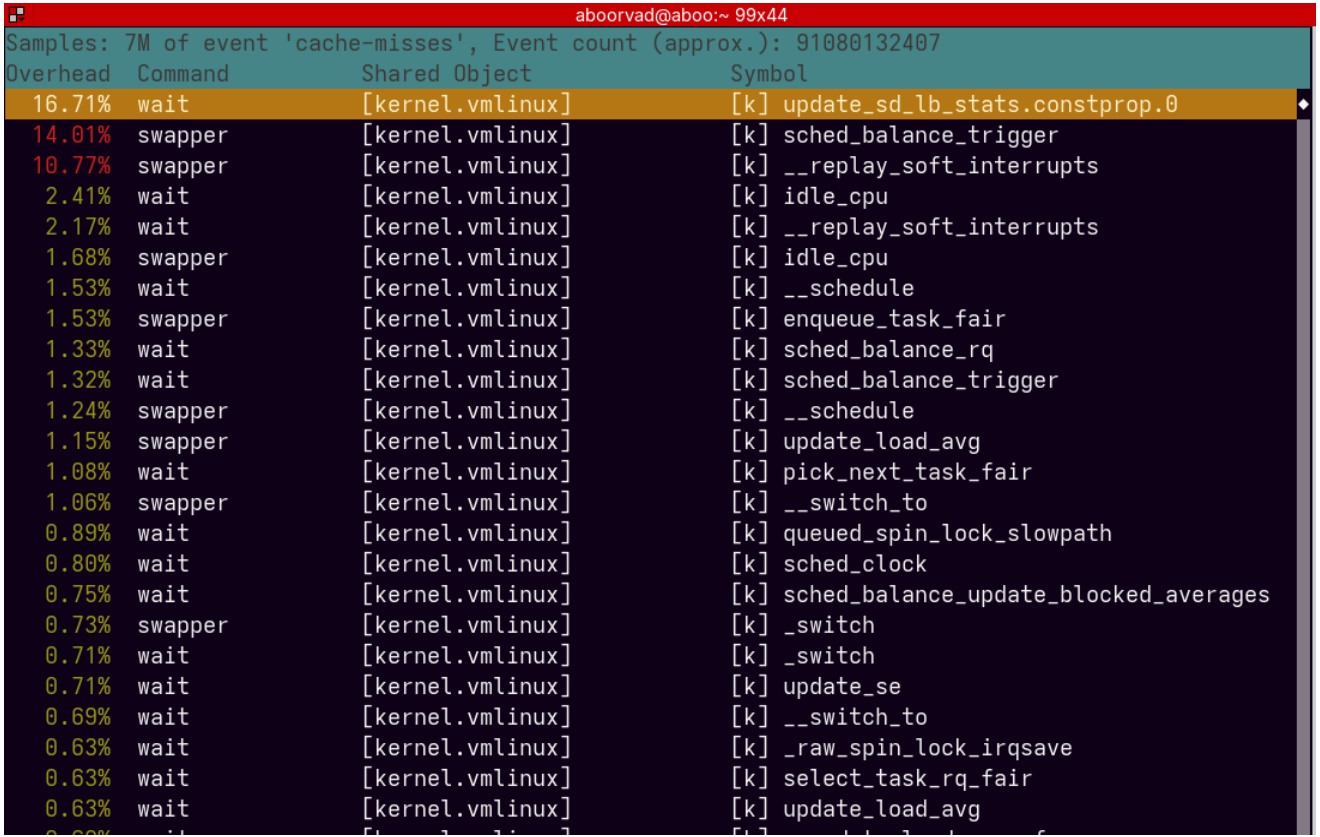



N workers: waitpid() -> BLOCKED -> CPU IDLE -> 128 CPUs, few tasks = frequent idle transitions -> SIGSTOP/SIGCONT storm unbalances per-CPU runqueues -> newidle_balance() / load_balance() fires repeatedly

*perf record -g -e cache-misses -- <workload>*

# update_sg_lb_stats: cache-misses



# idle_cpu cache-misses: instructions

```
aboorvad@aboo:~ 147x44
Samples: 7M of event 'cache-misses', 4000 Hz, Event count (approx.): 91080132407
idle_cpu  /root/.debug/.build-id/59/0fa69886d487979fe823790f39547c3900ed7b/elf [Percent: local peri
Percent |     */
        |     int idle_cpu(int cpu)
        |     {
        |       addis  r2,r12,352
        |       addi   r2,r2,-10660
        |       mflr   r0
        |     → bl     _mcount
        |     struct rq *rq = cpu_rq(cpu);
   0.00 |       addis  r9,r2,253
   0.00 |       sldi   r3,r3,3
   0.01 |       addi   r10,r9,16352
   0.00 |       addis  r9,r2,158
   0.17 |       addi   r9,r9,-5120
   0.00 |       ldx    r10,r10,r3
        |
        |     if (rq->curr != rq->idle)
        |     return 0;
   0.08 |       li     r3,0
        |     struct rq *rq = cpu_rq(cpu);
  25.78 |       add    r9,r9,r10
        |     if (rq->curr != rq->idle)
  53.17 |       ld     r8,3952(r9)
   0.71 |       ld     r10,3968(r9)
   4.38 |       cmpd   r8,r10
   7.20 |     ↓ beq    4c
        |
        |     if (rq->ttwu_pending)
        |     return 0;
        |
        |     return 1;
        |     }
   0.00 | 40:   extsw  r3,r3
   1.71 |     ← blr
```

# Cache lines, briefly.

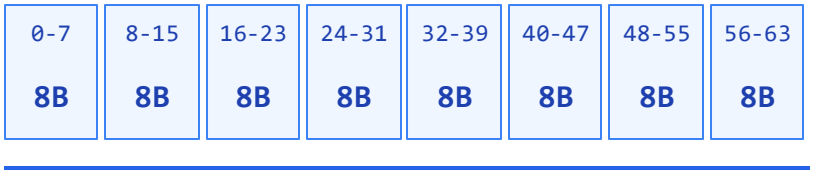
*How the hardware actually moves data.*

**CPU never fetches a single byte.**

It fetches a fixed-size cache line - usually 64 B on x86, 128 B on POWER.

If the data you access next is in that same line, **the fetch is free.** If it lives in a different line, you pay the miss again.

ONE 64-BYTE CACHE LINE (x86)




*fetched together, always (64 bytes..)*

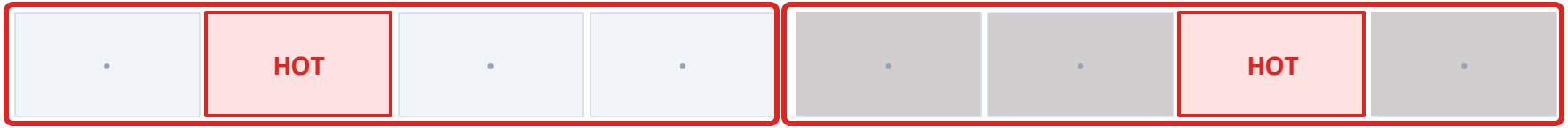
**RULE OF THUMB**

*Fields accessed together -> keep them inside one cache line.*

# Layout decides how many lines you pay for?

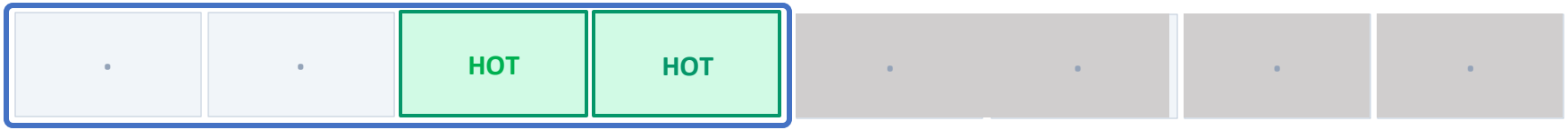
*Same workload, two layouts - one wins.*

LAYOUT A · hot fields scattered across the line




*-> Two distinct cache lines fetched to touch both hot fields*

LAYOUT B · hot fields colocated




*-> One fetch brings both hot fields together - half the work*

## STRUCTURE: struct rq

**struct rq (128 B/line, POWERPC):** *5 KB · 40 cache lines · hot fields scattered (based on the pahole output)*


### Cache line 0 (offset 0-127) · lock + scheduling counters

*read-write lock shares a line with read-mostly scheduling fields*

| | |
|---|---|
| `__lock` | runqueue spinlock – read-write, acquired on every context switch |
| `nr_running` | # tasks currently queued on this CPU |
| `nr_numa_running` | NUMA-local running count |
| `nr_preferred_running` | preferred-node running count |
| `ttwu_pending` | pending try-to-wake-up count |

⚠ *Every lock acquire/release writes this line, invalidating the read-mostly fields that share it.*

### Cache line 30

*offset 3952 · curr*

**`curr/donor`**
currently-running task pointer – read on every scheduling decision

*A single 8-byte pointer – but it sits on the fast path of every reschedule.*

### Cache line 31

*offset 4008 · idle & clock*

**`idle`**
idle-task pointer – checked on every pick_next_task()

**`clock`**
rq clock – updated on every tick & wakeup

### Cache line 32

*offset 4176 · clocks & cpu_capacity*

**`clock_task`**
task-fair clock – ____cacheline_aligned at offset 4096

**`cpu_capacity`**
CPU capacity – read on every load-balance scan

cpu_capacity sits 4176 B from nr_running on POWERPC's 128 B lines, that's **4 separate cache fetches** per load-balance scan.

# Why 4 cache lines per scan?

**4** cache lines per rq scan × **N** CPUs per group × **M** groups per domain × **L** domain levels = **a flood** of line fetches per idle transition

*struct rq (128 B/line, POWERPC) · 5 KB · 40 cache lines*

CL 0: __lock, nr_running  CL 30: curr  CL 31: idle, clock  CL 32: cpu_capacity

*Updating per-CPU load-balance stats touches*

- *__lock & nr_running (Line 0),*
- *curr (Line 30),*
- *idle & clock (Line 31),*
- *cpu_capacity (Line 32) - 4 separate cache-line fetches, every scan, per CPU*

# × N - repeated for every CPU in the group

**4**
cache lines per rq scan

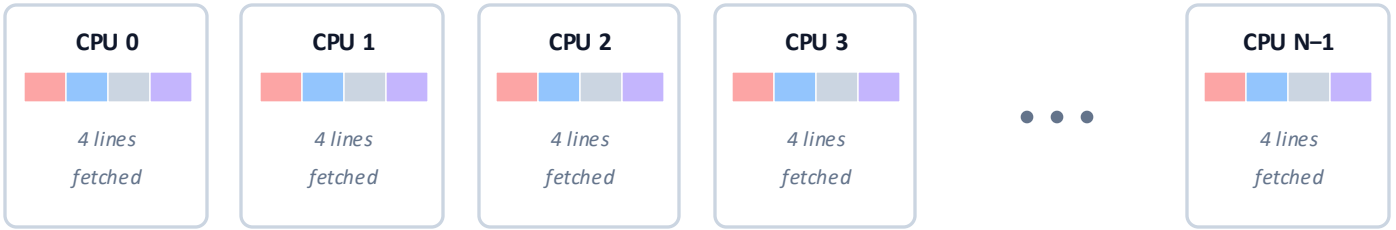
×

**N**
CPUs per group

× **M**
groups per domain

× **L**
domain levels

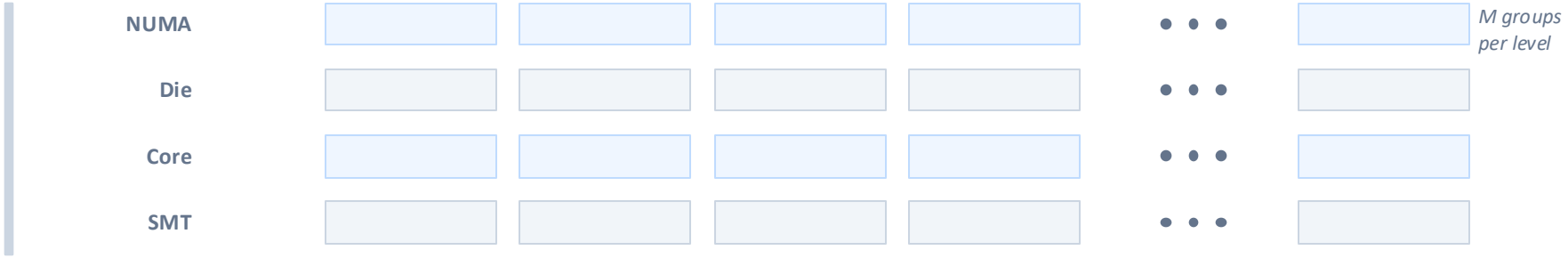
= **a flood**
of line fetches per idle transition




*The load balancer walks every CPU in the scheduling group - each one repeats the same 4-line fetch from the previous step.*

# × M × L - repeated across groups and domain levels

**4** cache lines per rq scan × **N** CPUs per group × **M** groups per domain × **L** domain levels = **a flood** of line fetches per idle transition




…and the whole walk - 4 lines per CPU, N CPUs per group - repeats for every group at every level of the scheduling-domain hierarchy: SMT, Core, Die, NUMA.

# = a flood of line fetches per idle transition

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **4** | × | **N** | × | **M** | × | **L** | = | **a flood** |
| cache lines per rq scan | | CPUs per group | | groups per domain | | domain levels | | of line fetches per idle transition |

**Thousands** of idle transitions per second

*on wait-heavy workloads - and this 4 × N × M × L chain of cache-line fetches runs every single time.*

Each term seemed small on its own - 4 lines, a handful of CPUs, a few groups, a few levels. Multiplied together, and multiplied again by the idle-transition rate, the result is a measurable share of this workload's cache-miss traffic.

# Both CPUs hold a valid copy of Cache Line 0

Cache Line 0 (offset 0–127)

| __lock | nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|---|

**CPU 5** — VALID

*L2 cache*

| __lock | nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|---|

*Holds Line 0, untouched since last sync.*

**CPU 9** — VALID

*L2 cache*

| __lock | nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|---|

*nr_running cached - ready for next read.*

Both CPUs have an identical, valid copy of Line 0 sitting in cache. CPU 9 read nr_running a moment ago - it's ready to use again at cache speed.

# CPU 5 acquires the runqueue lock

Cache Line 0 (offset 0–127)

| __lock | nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|---|

**CPU 5** — WRITE

*L2 cache · writes __lock*

| __lock | nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|---|

*__lock just written - line is now dirty.*

**CPU 9** — VALID

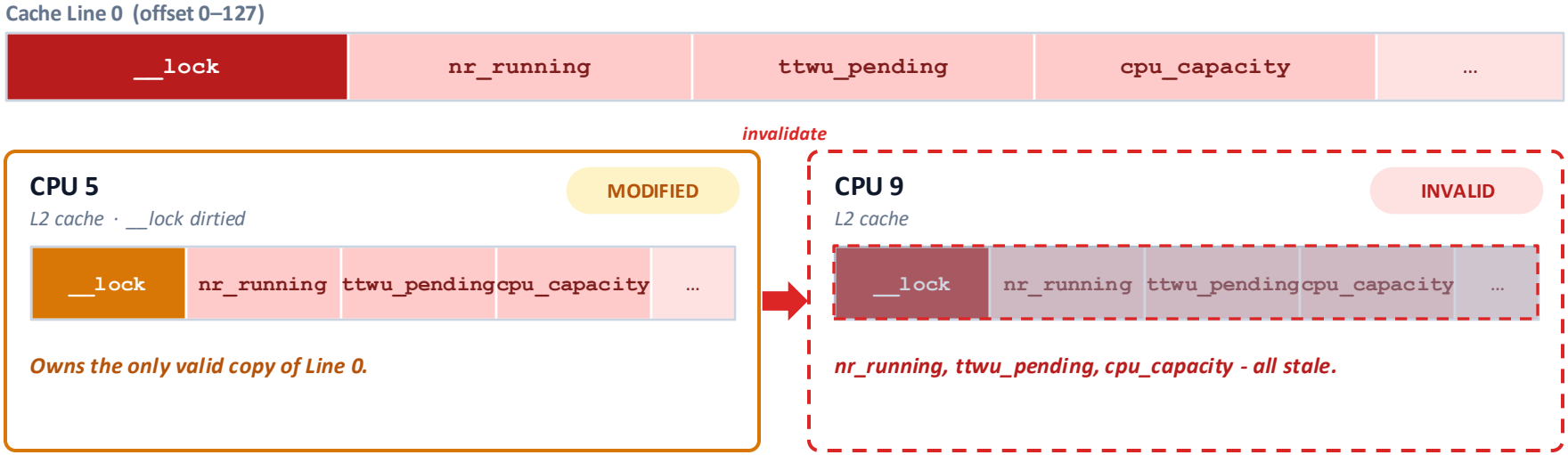
*L2 cache*

| __lock | nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|---|

*Still holding its old copy - for now.*

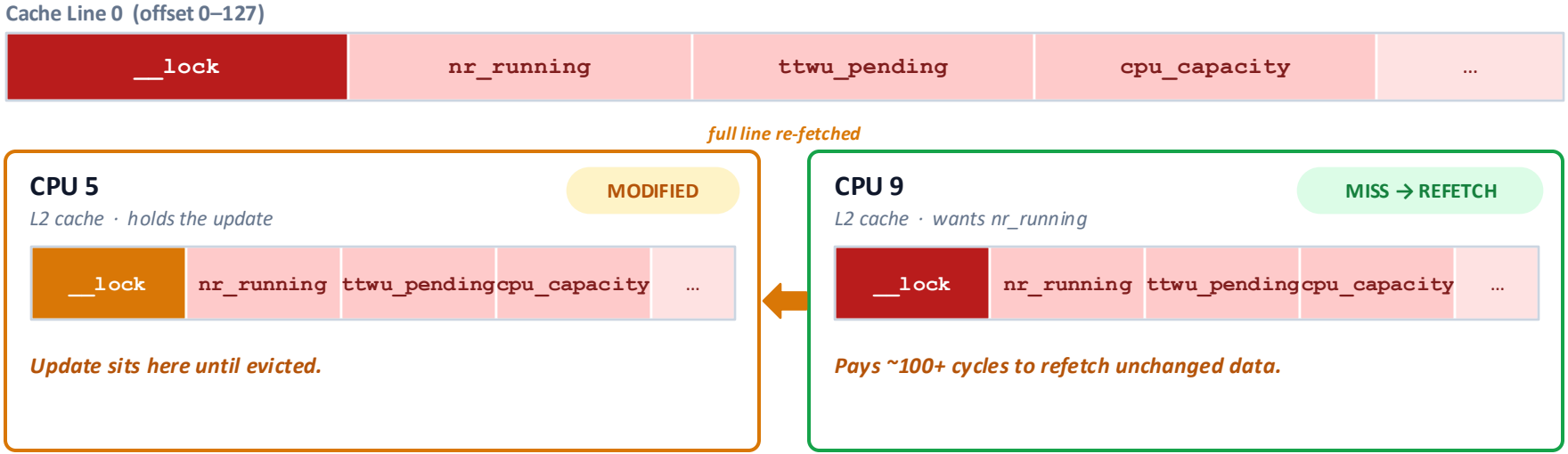
On every context switch, CPU 5 writes to __lock to acquire the runqueue spinlock. nr_running, ttwu_pending, and cpu_capacity on this same line do not change.

# The whole line is invalidated in CPU 9's cache




Cache coherence (MESI) invalidates the entire line in every other cache that holds it - not just the byte that changed. CPU 9's copy of nr_running, ttwu_pending, and cpu_capacity is now stale, even though none of them changed.

# CPU 9 re-fetches the whole line just to read nr_running




nr_running's value is identical to before - CPU 9 pays a full cache-line miss anyway, just to see it. Multiply this by every CPU, on every lock acquisition, thousands of times a second.

# Fix: give the lock its own cache line

Cache Line A · write-hot

| __lock | nr_switches, … (padding) |
|---|---|

Cache Line B · read-mostly

| nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|

**CPU 5** — VALID

*L2 cache*

*Line A*

| __lock | nr_switches, … (padding) |
|---|---|

*Line B*

| nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|

**CPU 9** — VALID

*L2 cache*

*Line B*

| nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|

***Never needs Line A - it only ever reads Line B.***

__lock now lives alone on Line A (padded out to 128 B). nr_running, ttwu_pending, and cpu_capacity move together onto Line B - a line the lock never touches.

# CPU 5 writes __lock - only Line A is invalidated

Cache Line A · write-hot

| __lock | nr_switches, … (padding) |
|---|---|

Cache Line B · read-mostly

| nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|

**CPU 5** — MODIFIED

*L2 cache · writes __lock*

*Line A*

| __lock | nr_switches, … (padding) |
|---|---|

*Line B*

| nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|

**CPU 9** — STILL VALID

*L2 cache · unaffected*

*Line B*

| nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|

***No invalidation reached this line.***

The invalidation is now contained to Line A. CPU 9 never cached Line A - it only holds Line B, and that copy is untouched.

# CPU 9 reads nr_running at cache speed - no re-fetch

Cache Line A · write-hot

| __lock | nr_switches, … (padding) |
|---|---|

Cache Line B · read-mostly

| nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|

**CPU 5**
*L2 cache*

MODIFIED

*Line A*

| __lock | nr_switches, … (padding) |
|---|---|

*Line B*

| nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|

**CPU 9**
*L2 cache · reads nr_running*

HIT

*Line B*

| nr_running | ttwu_pending | cpu_capacity | … |
|---|---|---|---|

***Served straight from cache - no memory trip.***

Same lock activity on CPU 5 as before - but now zero extra cache misses for readers on CPU 9. The read-mostly line simply stays resident.

# The rule: never let frequent writers and readers share a line

**BEFORE**

*Cache Line 0*


Lock and read-mostly fields share one line - every lock write evicts every reader.

**AFTER**

*Cache Line A*


*Cache Line B*


Lock isolated on its own line - readers stay resident; only the lock's line churns.

Write-hot fields and read-hot fields must **never share a cache line**.

*Every lock write evicts the readers - split the line, and the readers stop paying for the writer's traffic.*

# Kernel structs grows...

*What we see ...*

### Fields are added when patches land

Layout reflects merge order some times, not access patterns..

### Subsystems carry old assumptions

### pahole shows gaps everywhere

struct rq alone has multiple "XXX bytes hole, try to pack" annotations.

### Performance debt is invisible

Cache-miss cost on hot paths rarely surfaces in code review.

# Hot / Cold field reordering in the kernel.

*The community has been moving in this direction.*

FREQUENCY-BASED

## Hot / Cold reordering

Group frequently-accessed ("hot") fields together; separate rarely-accessed ("cold") fields.

Active proposals in community targeting multiple subsystems.

A netdev patch merged upstream - reordering produced measurable TCP gains.[1]

THE GAP

## Frequency ≠ Locality

Two fields can both be "frequently accessed" yet never accessed in the same window.

Raw access counts miss the temporal pattern that actually drives cache behaviour.

We want to know which fields are accessed together - not just which are frequent.

[1]. https://lore.kernel.org/netdev/20231129072756.3684495-1-lixiaoyan@google.com/

PRIOR WORK

# Hot / Cold

*The community has be*

FREQUENCY-BA

## Hot / Cold re ity

Group frequently-ac ently accessed" yet never separate rarely-acce .

Active proposals in mporal pattern that actually subsystems.

A netdev patch merg s are accessed together - not measurable TCP gai

[1]. https://lore.kernel.o

## Linux 6.8 Network Optimizations Can Boost TCP Performance For Many Concurrent Connections By ~40%

Written by Michael Larabel in Linux Networking on 9 January 2024 at 02:23 PM EST. 79 Comments

Beyond the usual new wired/wireless network hardware support and the other routine churn in the big Linux networking subsystem, the Linux 6.8 kernel is bringing some key improvements to the core networking code that can yield up to a ~40% improvement for TCP performance when encountering many concurrent network connections.

First up, there's been an analysis and reorganization of core networking structures. This effort has been around optimizing cacheline consumption and adding safeguards to ensure future changes don't regress. In turn this optimizing of core networking structures is causing TCP performance with many concurrent connections to increase by as much as 40% or more!

Coco Li of Google explained of their cachline optimization effort to the networking code:

> ***"Currently, variable-heavy structs in the networking stack is organized chronologically, logically and sometimes by cache line access.***
>
> ***This patch series attempts to reorganize the core networking stack variables to minimize cacheline consumption during the phase of data transfer. Specifically, we looked at the TCP/IP stack and the fast path definition in TCP."***

Their results are mighty impressive for being done to the core networking code:

```
On AMD platforms with 100Gb/s NIC and 256Mb L3 cache:
IPv4
Flows   with patches    clean kernel      Percent reduction
30k     0.0001736538065 0.0002741191042 -36.65%
20k     0.0001583661752 0.0002712559158 -41.62%
10k     0.0001639148817 0.0002951800751 -44.47%
5k      0.0001859683866 0.0003320642536 -44.00%
1k      0.0002035190546 0.0003152056382 -35.43%
```

# Hot / Cold field reordering in the kernel.

*The community has been moving in this direction.*

FREQUENCY-BASED

## Hot / Cold reordering

Group frequently-accessed ("hot") fields together; separate rarely-accessed ("cold") fields.

Active proposals in community targeting multiple subsystems.

A netdev patch merged upstream - reordering produced measurable TCP gains.[1]

THE GAP

## Frequency ≠ Locality

Two fields can both be "frequently accessed" yet never accessed in the same window.

Raw access counts miss the temporal pattern that actually drives cache behaviour.

***We want to know which fields are accessed together - not just which are frequent.***

[1]. https://lore.kernel.org/netdev/20231129072756.3684495-1-lixiaoyan@google.com/

# Data-Structure Splicing (DSS)

*Reorganize fields to match access - long-known in systems research.*

## Reorder

Move frequently co-accessed fields closer together inside the cache line.

## Split

Separate hot fields from cold ones to keep hot cache lines clean.

## Merge

Bring fields from related structures together when always touched as one.

FOUNDATION

*"A Unifying Abstraction for Data Structure Splicing" · MEMSYS '19*

# Access affinity - not just access frequency.

*The approach in one sentence.*

**Fields accessed close together in time**
***should be placed close together in memory.***

**01**

### Trace

Capture per-field load/store accesses during a workload.

**02**

### Score

When two fields are touched in a short window, log an affinity event.

**03**

### Cluster

Build a graph, cluster, then reorder to honour the clusters.

# The kstruct-tuner pipeline

*Four small Python tools, one shell of glue.*

**1**

**`Trace collection`**

*trace + pahole -> CSV*

**2**

**`generate proximity-graph`**

*co-access counts -> matrix*

**3**

**`gen-heatmap for visualization`**

*matrix -> interactive HTML*

→ `heatmap`

**4**

**hierarchical-clustering**

*matrix -> reorder + JSON*

→ insights.json

**INPUTS** `access pattern trace of runqueue fields and pahole of the structure.`

**Data Collection:** Capture per-field load/store accesses during a workload.

OUR APPROACH: TRACE COLLECTION

# Why we need field-granular access data

*To reorder a struct for cache locality we need three things per field:*

| | |
|---|---|
| **How often accessed** | Hotness - which fields dominate cycles |
| **Which fields co-accessed** | temporal affinity - read together = want same cache line |
| **Read vs write ratio** | read-mostly fields can share a line safely, write-hot must be isolated |

## Why not just use perf mem samples?

- perf samples addresses - misses most accesses (typically 1 in 1000+)
- No field attribution - raw address, not rq->nr_running
- Co-access timing is lost - can't build an affinity graph
- We need every access, labelled by field. Sampling can't give that.

## Our approach

| | |
|---|---|
| EXPLORED | perf tracepoints (field-labelled, dense) |
| CURRENT | QEMU TCG plugin (every access, no sampling) |
| TODO | POWER HTM traces (real hardware, no overhead) |

OUR APPROACH: TRACE COLLECTION

# Sample – perf trace

*One line per runqueue field access event · tracepoints inserted at scheduler paths where fields are accessed..*

```
perf 8472 [111] 316.808112: sched:fair_rq_access: Accessed rq[111]->clock_update_flags in rq_pin_lock (modify)
perf 8472 [111] 316.808113: sched:fair_rq_access: Accessed rq[111]->balance_callback in rq_pin_lock (access)
perf 8472 [111] 316.808114: sched:core_rq_access: Accessed rq[111]->clock_update_flags in update_rq_clock (access)
perf 8472 [111] 316.808115: sched:core_rq_access: Accessed rq[111]->cpu in update_rq_clock (access)
perf 8472 [111] 316.808116: sched:core_rq_access: Accessed rq[111]->clock in update_rq_clock (access)
perf 8472 [111] 316.808117: sched:core_rq_access: Accessed rq[111]->clock_task in update_rq_clock_task (modify)
perf 8472 [111] 316.808118: sched:core_rq_access: Accessed rq[111]->cpu in update_rq_clock_pelt (access)
perf 8472 [111] 316.808118: sched:core_rq_access: Accessed rq[111]->clock_pelt in update_rq_clock_pelt (modify)
```

`process pid [cpu] timestamp: tracepoint: rq[cpu]->field in function (access | modify)`

**Limitation:** tracepoints must be manually inserted at every field access site · dense enough for co-access graphs but requires kernel patching · coverage depends on which paths are instrumented

# Better Approach..

EXPLORING **QEMU TCG kstructmem.c**

*hooks every kernel load/store under full-system emulation*

- No kernel changes - TCG sees every guest memory op
- Ring buffer -> captures workload window
- Offline maps raw vaddr -> field name

**Binary record (24 B per access)**

```
u64 seq;    // global access order
u64 vaddr;  // guest virtual address
u16 cpu;    // vCPU index
u16 size;   // access width (bytes)
u32 flags;  // bit0 = store/load
```

**Trade-off**

Slower than KVM · disables hardware virtualisation · suited for controlled workloads

TODO **POWER HTM traces**

*hardware trace macro – POWER 10/11*

- Hardware ring buffer - zero emulation overhead
- Real timing
- Run production workloads at full speed

Same offline pipeline applies after collection

**Comparison**

| | QEMU | HTM |
|---|---|---|
| every access | ✓ | ✓ |
| no kernel patch | ✓ | ✓ |
| real timing | ✗ | ✓ |
| HW speed | ✗ | ✓ |

# Access affinity Graph generation..

# A worked example

| t1 | t2 | t3 | t4 | t5 | t6 |
|---|---|---|---|---|---|
| Field A | Field B | Field C | Field A | Field B | Field D |

**4 fields, 6 accesses, in timestamp order:**

Field A  Field B  Field C  Field D

## ACCESS AFFINITY GRAPH: WINDOW

### --method window

- Slide a time window of size W over the trace.
- Two fields touched within W -> +1 to their edge.
- Captures temporal locality directly.

**OUTPUT:** adjacency matrix of fields × fields with co-access weights.

# First window -> first edges

*Window 1 - t1, t2, t3*

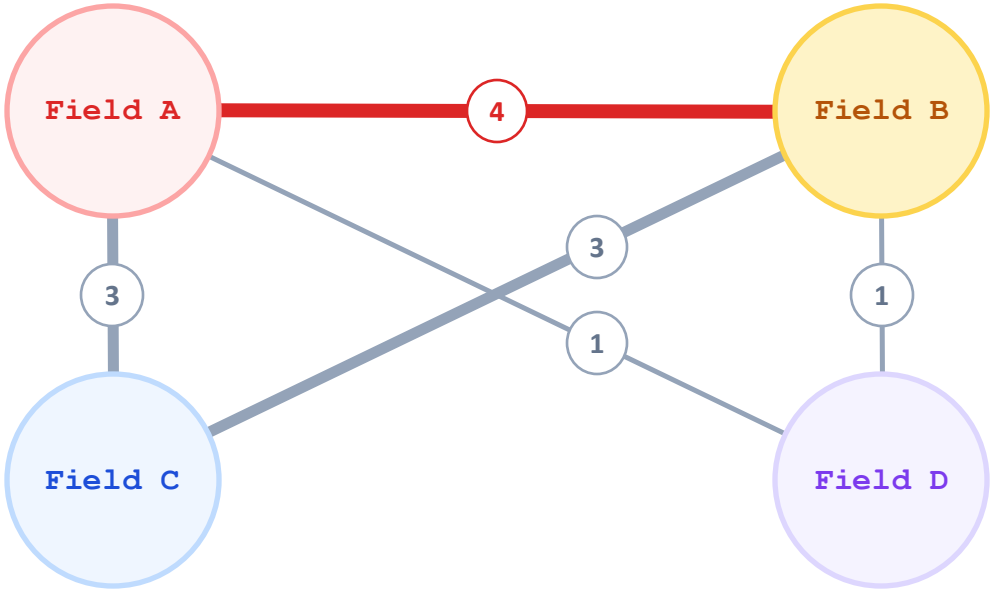
| t1 | t2 | t3 | t4 | t5 | t6 |
|---|---|---|---|---|---|
| Field A | Field B | Field C | Field A | Field B | Field D |

**All 3 fields are distinct -> every pair gets +1:**

| | |
|---|---|
| A ↔ B | 1 |
| A ↔ C | 1 |
| B ↔ C | 1 |

Window 1 covers fields A, B, and C. Each of the three pairs they form scores +1. The window then slides forward one access at a time, repeating this for windows 2-4.

# Sliding through -> the affinity graph




Windows 2 and 3 cover the same trio (A, B, C) as window 1, so those pairs keep climbing. Window 4 swaps in D, adding two lighter edges. After all 4 windows, A ↔ B is the heaviest edge, these two fields are almost always touched together.

# But the affinity graph isn't the whole story

**`Field A` ↔ `Field B`** scores highest

*→ naive read: put them on the same cache line.*

**...but we already saw why that's exactly the trap.**

If A is write-hot (dirtied on every context switch) and B is read-mostly, co-locating them means every write to A invalidates B for every other CPU - the false-sharing scenario from the previous section, reintroduced by the very metric meant to fix layout.

Affinity tells you ***what*** to co-locate. Read/write classification tells you ***whether you're allowed to.***

# hierarchical-clustering

*Adjacency matrix -> reordered layout + insights.*

## INPUTS

- Adjacency matrix

## OUTPUTS

**`insights.json`** - same data, structured
Reordered field list, clusters, top pairs, strengths, cohesion.

## WHAT INSIGHTS.OUT CONTAINS

- Reordered field list (the suggested layout)
- Clusters (which fields go in which group)
- Top field pairs by co-access weight
- Per-field interaction strength (sum of weighted edges)
- Strongest inter-cluster links
- Per-field top-K partners - what each field's closest neighbours are

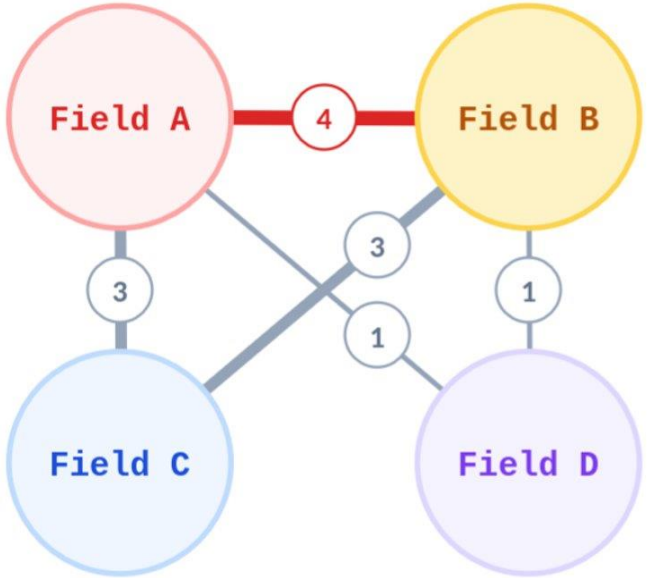
# The access-affinity graph

*How co-temporal access becomes a layout recommendation.*

**Node** a field of the structure under analysis.

**Edge** weight = number of affinity events between two fields.

**STRONGLY-CONNECTED CLUSTERS**
-> natural groups to lay out contiguously inside cache-line boundaries.

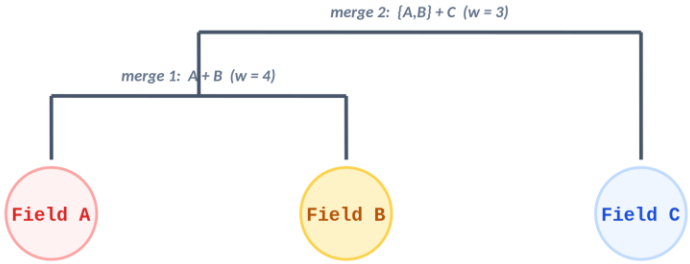


# Hierarchical clustering - how we merge.

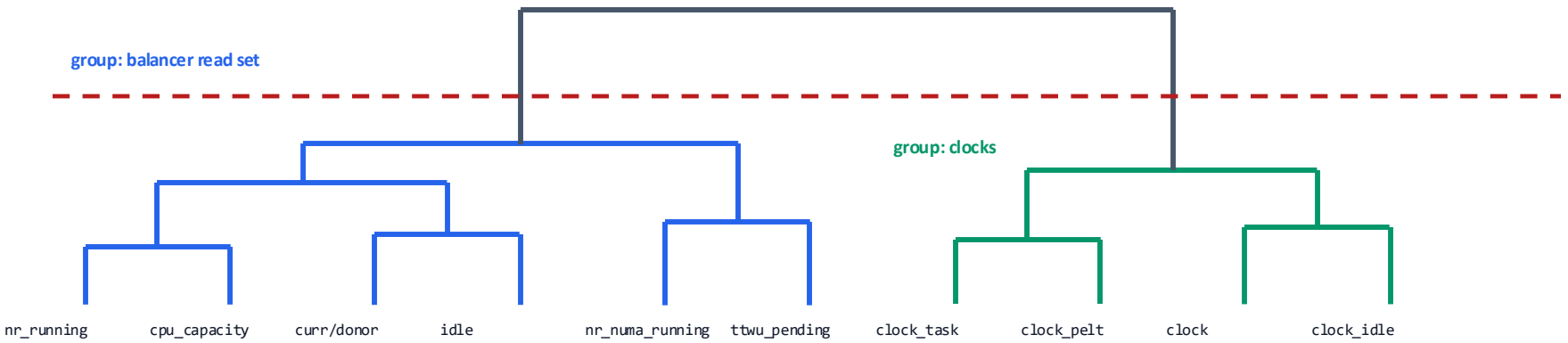
*Greedy merge over strongest edges first.*

### ALGORITHM

1. Start: each field is its own cluster.
2. Sort all edges by weight, descending.
3. For each edge (i, j) in order:
   if i and j are in different clusters, merge their clusters.
4. Stop when no more merges produce useful gains.
5. Result: reordered field list + cluster map.
6. Separate highly contended fields when beneficial (rq->__lock).



# dendrogram – field grouping




low merges = high co-access affinity ·

# Run the whole pipeline.

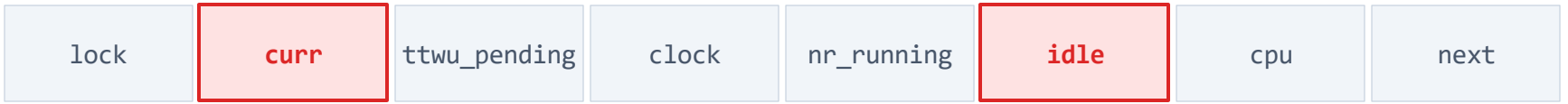
*Four commands, end to end*

```
# 1. parse trace + pahole into a CSV
python3 struct-parser.py --exclude_cross_cpu \
    ../logs/log.min ../logs/pahole ../logs/log.min.csv

# 2. build the co-access adjacency matrix
python3 proximity-graph.py --method window \
    --window_size 5 --n_jobs 10 \
    --input_file ../logs/log.min.csv \
    --output_file ../logs/log.graph.csv

# 3. hierarchical clustering -> reorder + insights
python3 hierarchical-clustering.py ../logs/log.graph.csv \
    -o ../logs/insights.out \
    --json_out ../logs/insights.json \
    --top_n 30 --per_field_top_k 5

# 4. render an interactive HTML heatmap
python3 gen-heatmap.py ../logs/log.graph.csv ../logs/heatmap.html
```

# Before & after - conceptual field layout

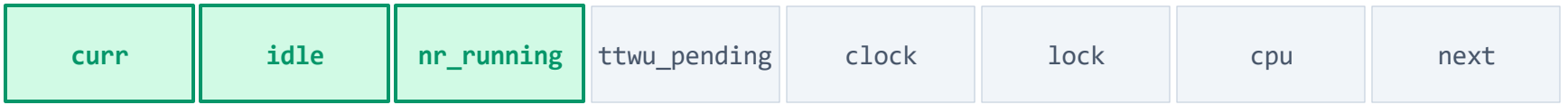
*Idle-accounting fields colocated → single cache line.*

*before - idle & curr separated*




*-> two cache lines touched on idle accounting*

*after - affinity cluster colocated*

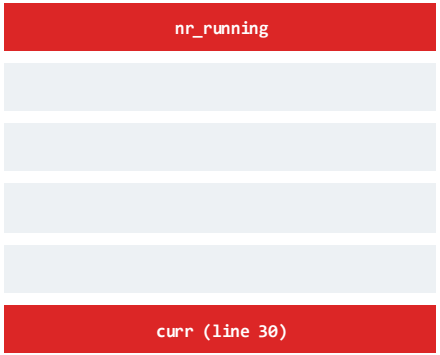



*-> one cache line, all hot fields together*

# Basic cause-and-effect in the study: idle_cpu

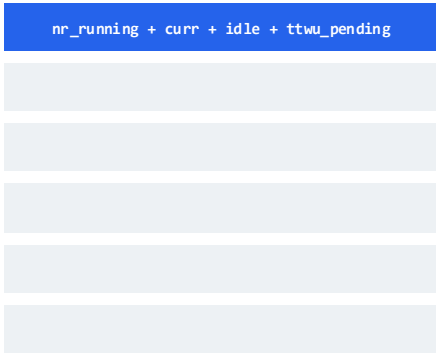
```
int idle_cpu(int cpu) {
  struct rq *rq = cpu_rq(cpu);
  if (rq->curr != rq->idle) return 0;
  if (rq->nr_running)       return 0;
  if (rq->ttwu_pending)     return 0;
  return 1;
}
```

Baseline - 2 line fetches




prediction (from offsets):
3 -> 1 lines ≈ halved misses

Reordered - 1 line fetch




measured (perf record):
6.8% -> 3.1% ✓

# Absolute Cache misses dropped 15-20%.

*Baseline on wait_stressor.*

```
~# /usr/bin/perf stat -a -e cache-misses,cache-references sleep 10
 Performance counter stats for 'system wide':
    36,333,000,900      cache-misses                    #   13.663 % of all cache refs
   265,918,606,358      cache-references

      10.005079575 seconds time elapsed
```

*Affinity-based layout on wait_stressor.*

```
~# /usr/bin/perf stat -a -e cache-misses,cache-references sleep 10
 Performance counter stats for 'system wide':
    30,384,627,728      cache-misses                    #   10.373 % of all cache refs
   292,911,657,780      cache-references

      10.006359270 seconds time elapsed
```

RESULTS

# Absolute Cache misses dropped 15-20%.

*Affinity-based layout vs. baseline on wait_stressor.*

BEFORE · baseline layout

**13.88%**

*cache-miss rate*

AFTER · affinity-based layout

**10.3%**

*cache-miss rate*

**15% to 20%** reduction in absolute cache misses on hot scheduling paths

# Where the misses went

*Top cache-miss symbols - perf record, sorted (wait_stressor)*

| BEFORE (abs: 70726761316) | | |
|---|---|---|
| ... | wait | update_sd_lb_stats.constprop.0 |
| 6.40% | wait | idle_cpu |
| 1.95% | wait | sched_balance_rq |
| 1.93% | swapper | enqueue_task_fair |
| 1.86% | wait | __schedule |

| AFTER (abs: 65317448258) | | |
|---|---|---|
| ... | wait | update_sd_lb_stats.constprop.0 |
| 3.11% | wait | idle_cpu |
| 2.05% | swapper | enqueue_task_fair |
| 1.92% | wait | __schedule |
| - | | *sched_balance_rq dropped out* |

***idle_cpu cache misses fell from 6.4% -> 3.11%, sched_balance_rq dropped out of the top 5***

# pahole, before and after (similar idea)

the layout change in raw offsets · 128 B lines

**BASELINE - scattered**

```
__lock            @    0  line 0  RW
nr_running        @    8  line 0
nr_numa_running   @   12  line 0
ttwu_pending      @  104  line 0
…3.8 KB of other fields…
curr/donor        @ 3952  line 30 <- far
idle              @ 3968  line 31 <- far
cpu_capacity      @ 4176  line 32 <- far


size 5120 · 40 lines · 9 holes
```

**REORDERED - clustered**

```
nr_running        @    0
nr_numa_running   @    4
nr_preferred_run  @    8
ttwu_pending      @   12
cpu_capacity      @   16
curr/donor        @   24
idle              @   32

nr_switches       @  128  aligned
__lock            @  136

clocks consolidated @ 4224+
size 5248 · 41 lines (+1 padding)
```

sched: reorder some fields in struct rq

# Open questions

PROFILING

**Beyond static tracepoints and QEMU**

WORKLOADS

**Which benchmarks really matter**

Picking representative workloads per structure is critical - and not yet generalized.

CACHE LINES

**64 B, 128 B and beyond (architecture dependent), and kernel configuration limits.**

How should we reorder fields across architectures while avoiding false sharing?

TARGETS

**Which structures benefit most**

Beyond struct rq - where else do co-temporal access patterns dominate performance?

STATIC+DYN

**Combining analysis modes**

Can static analysis predict access patterns and complement dynamic tracing?

MAINTAINING

**Layout lifecycle**

Every new field forces a re-run. How do we keep layouts fresh as structures evolve?

# Open Challenge

Existing tools can identify hot data structures, field access frequencies, and performance bottlenecks. However, automatically deriving field-access affinity, generating optimal layouts, and continuously maintaining those layouts as software evolves remains an open challenge.

Locality-aware reordering can improve performance, but the exact benefit is workload, architecture, and configuration-dependent.

VISION

# From prototype to generic tool.

## NOW

### Prototype on struct rq

- Manual and QEMU traces
- Hierarchical clustering
- Validated on wait_stressor

## NEXT

### Generalize the tooling

- Auto-instrumentation / instruction tracing
- Multiple target structures
- Architecture-aware (64 B / 128 B)

## FUTURE

### Possible integration?

- Suggest layout for new fields or gaps in existing structure layouts. (insights).
- Can be part of pahole or perf infrastructure.
- Quantify impact pre-merge

# Paper references

PAPER

## A Unifying Abstraction for Data Structure Splicing

*MEMSYS '19 - the formal model behind reorder/split/merge.*

https://people.ece.ubc.ca/sasha/papers/memsys19.pdf

# References

PATCH

**Struct layout tuning by access frequency**

*proposed approach for kernel-wide layout tuning.*

lore.kernel.org/.../blakejones@google.com

PATCH

**Field reordering proposal**

*exploration of frequency-based reordering.*

lore.kernel.org/.../zecheng@google.com

PATCH

**netdev field reordering**

*already merged; demonstrated TCP gains.*

lore.kernel.org/netdev/.../lixiaoyan@...

# Questions?

# Thank you.

# 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.