Linux Kernel Scheduling and Split-LLC Architectures

Overview, Challenges, and Opportunities

Gautham R. Shenoy
K. Prateek Nayak

Challenges

Issue that we have observed with scheduling on Split-LLC Architecture

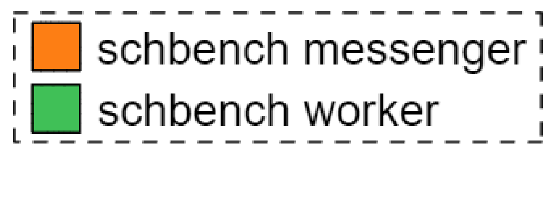
[AMD Official Use Only - General]

Challenges of Split-LLC

Cross LLC communication overhead in schbench

Challenges with schbench

- **schbench** at **lower worker count shows a large amount of run-to-run variance** based on how the tasks are spread around in the system.
- **schbench prefers messenger and worker to be co-located** on the same LLC with 99th %ile latencies degrading as the distance between the worker and messenger increases.
- **schbench** uses **Futex** to signal the waiting worker to wakeup, recording the time elapsed between signaling and the worker waking up. Wakeup from Futex **doesn't have WF_SYNC** flag set hence, the scheduler **will not** be aggressive at **consolidating** the messenger and worker on the same LLC.



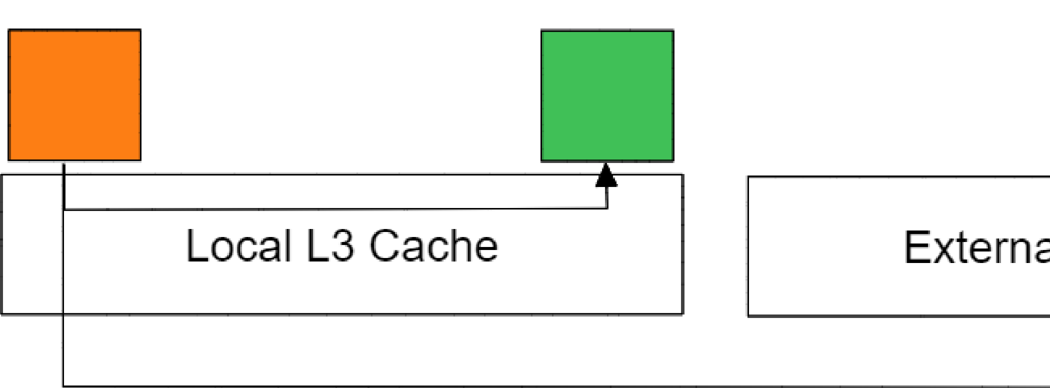


Figure: Variation in the tail latency reported by schbench based on placement of messenger and worker.

Variance in Tail Latencies Reported

Surprising tale of consolidation

- We observe that the schbench **messenger and worker are eventually co-located on the same LLC**, not as a result of wakeup migration, but **because of new-idle balance**.
- During the runtime, there arises a scenario where a **kworker** thread triggered by schbench and the **schbench thread are placed on the same run queue**.
- The **messenger**, having just **went to sleep**, will **trigger a new-idle balance** on the CPU where it was running, and this will **pull the schbench worker thread towards messenger's LLC** as the **kworker cannot be migrated**.
- When the **messenger wakes up** later, it'll find the CPU it previously ran on busy and **will find an idle CPU on same LLC**, thus leading to **colocation of worker and messenger**.

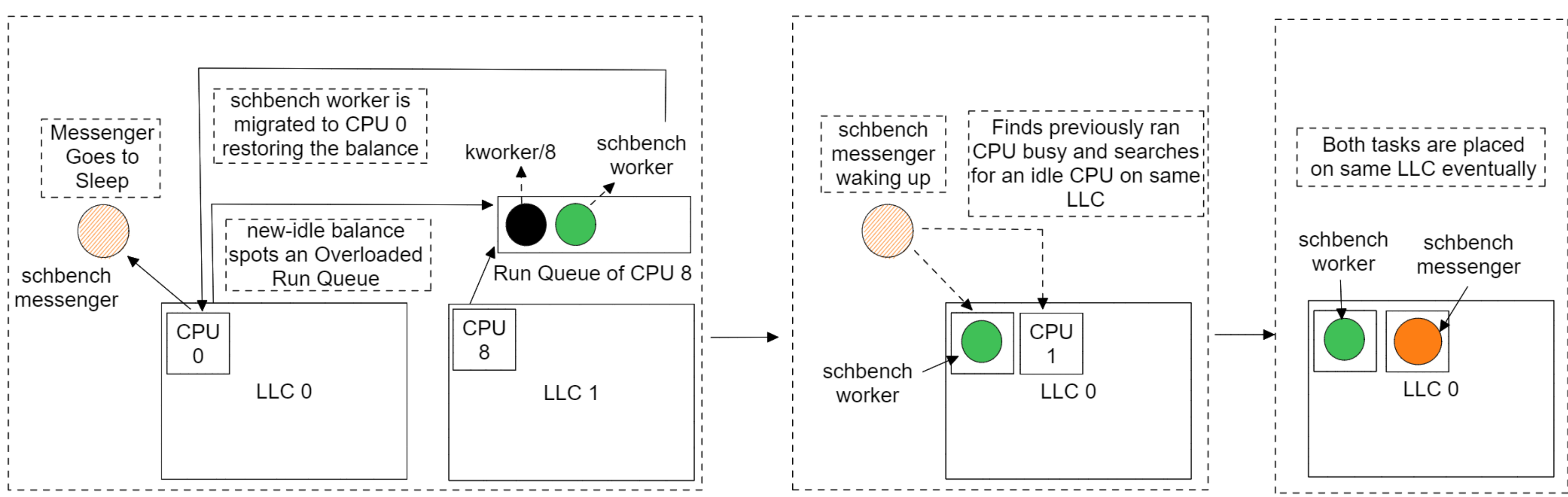
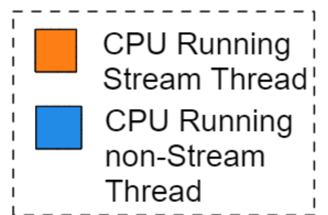


Figure: Series of events resulting in schebench messenger-worker consolidation

[AMD Official Use Only - General]

Challenges with Split-LLC

Inconsistency in Stream after NUMA Imbalance Rework



Challenges with Stream and Initial Placement

- With the **NUMA Imbalance rework by Mel Gorman**, we start **exploring LLCs in the non-local NUMA node when we have equivalent of one task per LLC in the local NUMA node**.
- With the current logic, we can **optimally place the Stream threads out of the box** and get **same performance as case with pinning**.
- However, when there is an **external task** running in the system, the **current logic will default to local LLC when there is a tie in number of idle CPUs between the local LLC and the idlest LLC** thus **leading to a pileup and performance degradation of 8-12% is observed** between good runs and bad runs.
- **Before** the NUMA Imbalance rework, the **Stream results were consistently poor** as a result of the fact that we were **never exploring the LLCs from the non-local NUMA node,** and we always had **at least one LLC with more than one Stream thread**.

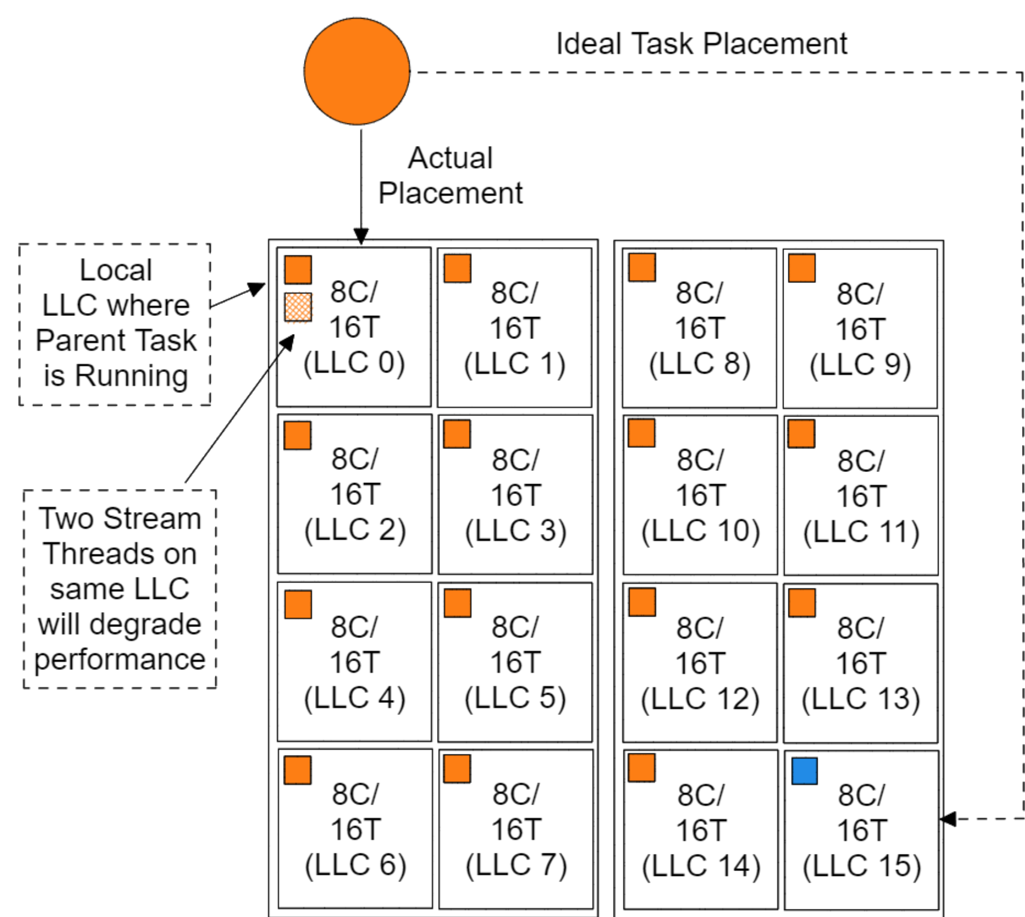


Figure: Two Stream treads being placed on the same LLC as a result of bias towards local LLC when there is a tie in number of idle CPUs in the local LLC and the idlest LLC

AMD together we advance_

How to detect if the Workloads are Bandwidth Intensive? (like Stream)

Lack of cached metrics to spot bandwidth-oriented task

- The **bias toward local group** when both the local group and the idlest group have **same number of idle CPUs** is **unfavorable for workloads such as Stream**.
- Going by the **utilization to break the tie** helped to an extent but was not foolproof because:
  - Kernel threads such as **kcompactd** bumped up the utilization, there by biasing towards the local group yet again.
  - Workloads such as **hackbench** which **preferred consolidation** regressed.
- Stream has a **large memory footprint** right from the moment the Stream threads are forked. Can metrics such as the memory footprint be **used as a proxy for the cache busyness** of the local group thereby facilitating better spread?

Hackbench – runtime (less is better)

# of groups	Prefer Local Group	Choose Based on group_util	Difference (%)
1	1.00	1.02	-2%
2	1.00	0.99	+1%
4	1.00	1.00	0%
8	1.00	1.02	-2%
16	1.00	1.06	-6%

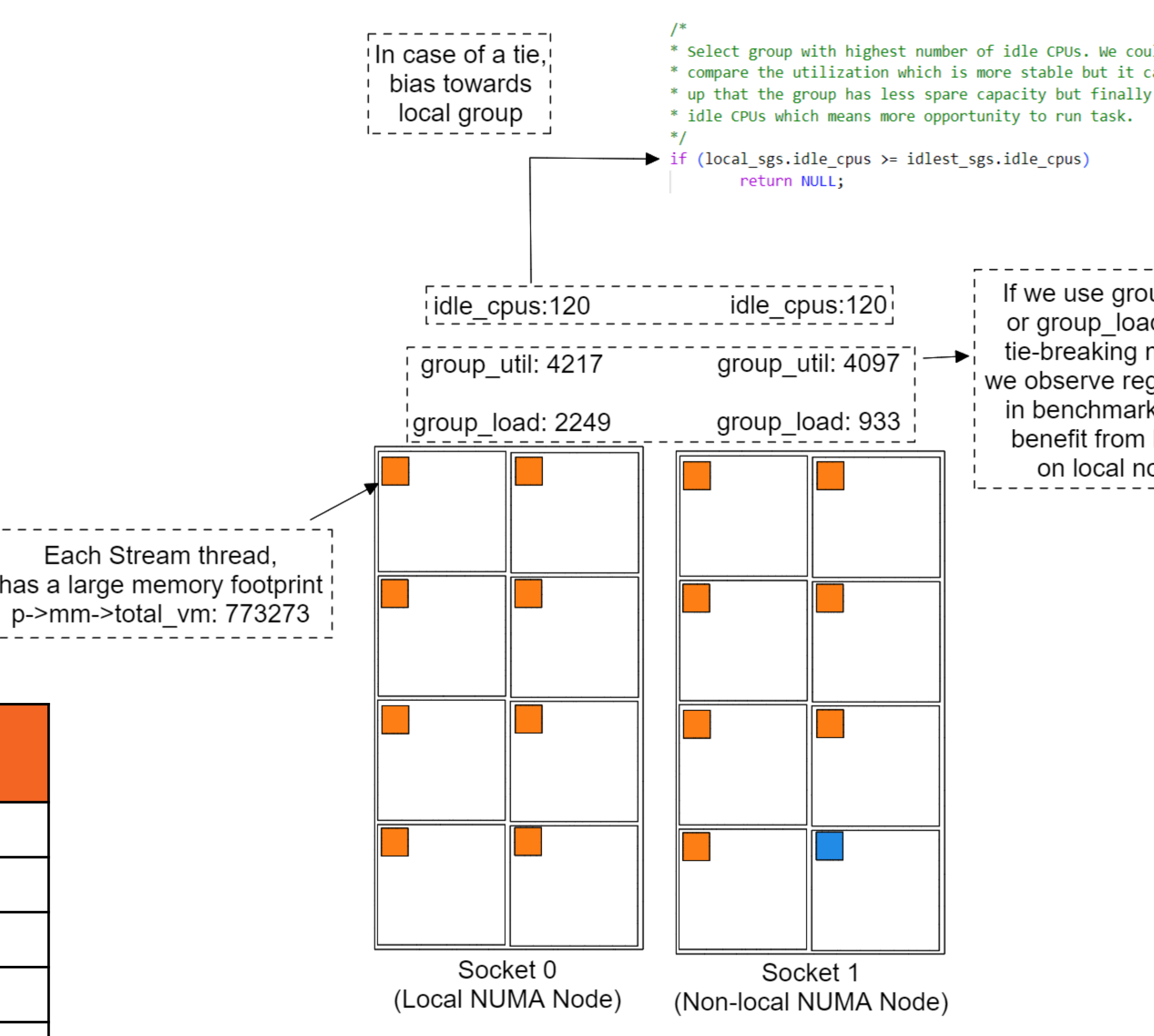


Figure: Potential tie breaking metrics

AMD together we advance_

Challenges with Split-LLC

tbench: Thundering Herd Scenario

Thundering herd in tbench

- tbench tasks have a peculiar initial wakeup behavior where the tasks, **once placed** on the CPU **will wake up and soon go to sleep**.
- With the **initial wakeup path** depending on the **number of idle CPUs**, this pattern is not favorable as the task that goes to sleep soon after waking up will not change the number of idle CPUs for long.
- With **only few tasks running in the system**, the **NUMA imbalance threshold** is rarely crosses and most **tasks will be placed on the local NUMA node**.
- When the **tasks wakeup** later, **they storm the small LLC** and thus end up **overloading** it, later **depending on the load balancer** to reach an optimal state later.
- We've observed that **with a more balanced initial placement**, we can not only reduce the number of migrations required later to reach an optimal stable state, but also improve tbench throughput in several different cases.

Go to sleep waiting for a signal

NUMA Imbalance threshold: 8

tbench tasks forked

After receiving signal, all task wakeup at once overwhelming the local NUMA node and LLCs on it

8C/ 16T (LLC 0)	8C/ 16T (LLC 1)
8C/ 16T (LLC 2)	8C/ 16T (LLC 3)
8C/ 16T (LLC 4)	8C/ 16T (LLC 5)
8C/ 16T (LLC 6)	8C/ 16T (LLC 7)

Socket 0 (Local NUMA Node)

8C/ 16T (LLC 8)	8C/ 16T (LLC 9)
8C/ 16T (LLC 10)	8C/ 16T (LLC 11)
8C/ 16T (LLC 12)	8C/ 16T (LLC 13)
8C/ 16T (LLC 14)	8C/ 16T (LLC 15)

Socket 1 (Non-local NUMA Node)

Figure: Illustration of tbench initial wakeup behavior leading to thundering herd scenario

tbench Initial Placement Imbalance

Current Situation and Consequences

Initial LLC Distribution

LLC	Task Spawn Count	% of total task	Overloaded?
0	4	3.12	
1	1	0.78	
2	2	1.56	
3	1	0.78	
4	3	2.34	
5	3	2.34	
6	2	1.56	
7	7	5.46	
8	15	**11.71**	
9	12	**9.37**	
10	13	**10.15**	
11	11	**8.59**	
12	2	1.56	
14	**32**	**25.00**	Overloaded
15	**20**	**15.62**	Overloaded

Initial NUMA Distribution

NUMA Node	Task Spawn Count	% of total task
0	23	17.96
1	105	**82.03**

|
v

NUMA Distribution after 10000 Migrations

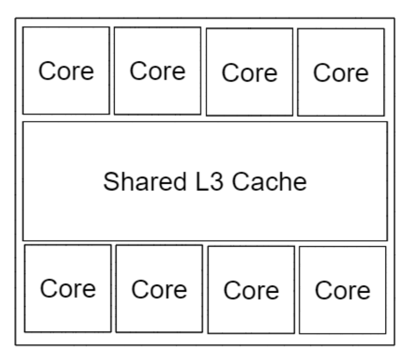
NUMA Node	Task Spawn Count	% of total task
0	67	52.34
1	61	47.65

Note: We've observed the number of **migrations** come **down by 85%** with a **more optimal initial placement** and an improvement of **20% in the reported bandwidth** for **64 client** case and an improvement of **12% in reported bandwidth** for **128 client** case on a **dual socket system featuring 3rd Generation EPYC processors**

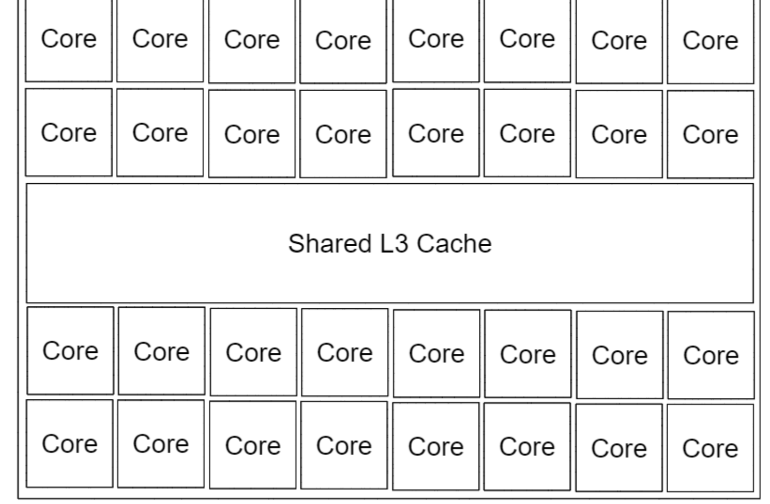
[AMD Official Use Only - General]

Why do we face these challenges only on Split-LLC Architectures?

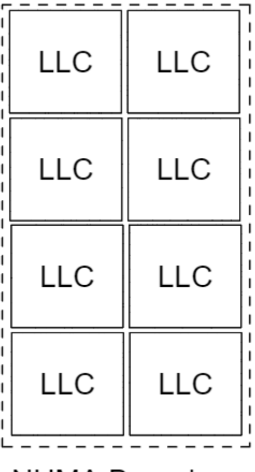
- Often, a **unified LLC architecture** will have a **greater number of CPUs attached to same L3** cache compared to the Split-LLC offerings. With the current scheduler heuristics in the wakeup path, there is **a higher probability that communicating tasks get consolidated onto CPUs belonging to same LLC**.
- **Crossing an LLC boundary** in a **unified LLC** architecture almost always results in **crossing the NUMA boundaries**. With optimizations such as **Auto NUMA** in place, there is a greater chance for the task to be placed on the **most optimal NUMA node** and hence on the **most optimal LLC**.
- Thus, with the current scheduler heuristics, the ***probability of getting the placement decision incorrect is lower on unified LLC architectures*** as opposed to on the split LLC architectures.



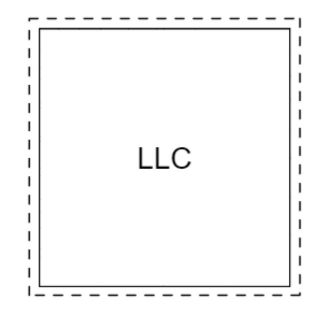
Less opportunity to consolidate tasks on same LLC



More opportunity to consolidate tasks on same LLC



NUMA Boundary (Multiple LLC on same NUMA Node)



NUMA Boundary (Each LLC is itself is on a separate NUMA Node)

Figure: Limitations of Split-LLC architectures when comparing with a unified LLC design.

Potential solution

What we have tried out

Userspace Hinting for Scheduler

Defining expected scheduler behavior for the workload

Peter Zijlstra's case for hints based on workload characteristics
(https://lore.kernel.org/lkml/YVwnsrZWrnWHaoqN@hirez.programming.kicks-ass.net)

```
I'm also thinking that adding more heuristics isn't going to improve the
situation lots.

For the past # of years people have been talking about extendng the task
model for SCHED_NORMAL, latency_nice was one such proposal.

If we really want to fix this proper, and not make a bigger mess of
things, we should look at all these various workloads and identify
*what* specifically they want and *why*.

Once we have this enumerated, we can look at what exactly we can provide
and how to structure the interface.

The extention must be hint only, we should be free to completely ignore
it.

The things I can think of off the top of my head are:

- tail latency; prepared to waste time to increase the odds of running
  sooner. Possible effect: have this task always do a full
  select_idle_sibling() scan.

  (there's also the anti case, which I'm not sure how to enumerate,
  basically they don't want select_idle_sibling(), just place the task
  wherever)

- non-interactive; doesn't much care about wakeup latency; can suffer
  packing?

- background; (implies non-interactive?) doesn't much care about
  completion time either, just cares about efficiency

- interactive; cares much about wakeup-latency; cares less about
  throughput.

- (energy) efficient; cares more about energy usage than performance
```

Userspace Hinting for Scheduler

Exploration : RFC Patches at https://lore.kernel.org/all/20220910105326.1797-1-kprateek.nayak@amd.com/

- Task placements and movement decisions are taken by the scheduler only at the following points:
  - **During fork() / exec()**
  - **During subsequent task wakeup**.
  - **During load balancing.**
- We can have a **hint for each decision point** to influence task placement in a desired way.
- Hints explored for **initial placement**:
  - **FORK_AFFINE**: Wakeup close to parent
  - **FORK_SPREAD**: Spread regardless of NUMA imbalance threshold restriction. Use utilization as a tie breaking metric when number of idle CPUs in groups are same.
- Hints explored for **subsequent wakeup**:
  - **WAKE_AFFINE**: Wakeup close to waker
  - **WAKE_HOLD**: Wakeup on same LLC where the task previously ran
- **Hints are ignored** if the preferred LLC or the currently running LLCs are **overloaded.** Scheduler is aware if the task is on a preferred LLC and will try to **avoid moving** the elsewhere **during load balancing** if LLC has capacity.
- For a **user consumable API** these hints needs to be further **abstracted** and possibly be **paired with other optimal tunable values** that **favor** the characterized **workload.**

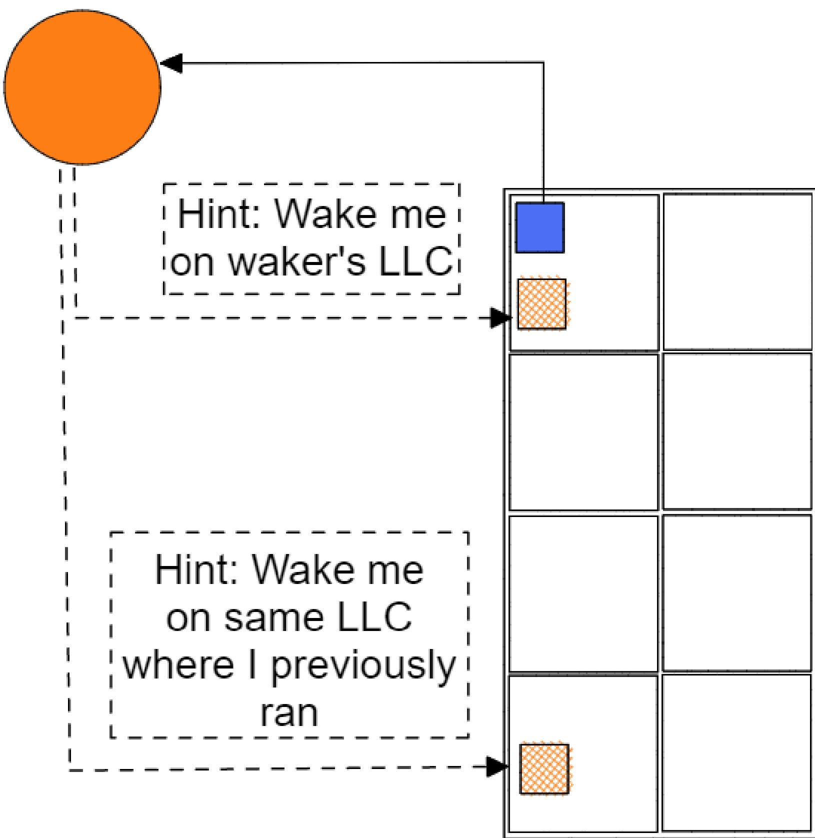


Figure: Low level hints that define task's wakeup behavior

Userspace Hinting : schbench

schbench (correct hints) – tail latency (Less is better)

# of clients	Default (Normalized)	Hint: FORK_AFFINE (Normalized)	Improv. (%)
1	1.00	0.86	+14%
2	1.00	0.91	+8%
4	1.00	0.90	+10%
8	1.00	0.77	+23%
16	1.00	0.86	+14%
32	1.00	0.92	+8%
64	1.00	0.97	+3%
128	1.00	0.96	+4%
256	1.00	1.03	-3%

schbench (incorrect hints) – tail latency (Less is better)

# of clients	Default (Normalized)	Hint: FORK_SPREAD (Normalized)	Improv. (%)
1	1.00	2.04	-104%
2	1.00	1.82	-82%
4	1.00	1.16	-16%
8	1.00	1.06	-6%
16	1.00	0.99	+1%
32	1.00	1.00	+0%
64	1.00	1.00	+0%
128	1.00	0.98	+2%
256	1.00	0.99	+1%

All benchmarks were run on a dual socket (**2 x 64C/128T**) **system featuring 3rd Generation EPYC processors**
running modified kernel based on the baseline **tip:sched/core** at **sched-core-2022-08-01**

Userspace Hinting : Hackbench and tbench

- Hackbench (correct hints) – runtime (less is better)

# of clients	Default (Normalized)	Hint: FORK_AFFINE + WAKE_AFFINE (Normalized)	Difference (%)
1	1.00	0.97	+3%
2	1.00	0.98	+3%
4	1.00	0.98	+2%
8	1.00	0.96	+4%
16	1.00	0.96	+4%

- Hackbench (incorrect hints) – runtime (less is better)

# of clients	Default (Normalized)	Hint: FORK_SPREAD (Normalized)	Difference (%)
1	1.00	1.03	-3%
2	1.00	1.06	-6%
4	1.00	0.98	+2%
8	1.00	0.98	+2%
16	1.00	0.98	+2%

- tbench (Correct Hint) – Bandwidth (More is better)

# of clients	Default (Normalized)	Hint: FORK_SPREAD (Normalized)	Difference (%)
1	1.00	0.98	-2%
2	1.00	0.97	-3%
4	1.00	0.99	-1%
8	1.00	1.03	+3%
16	1.00	1.08	+8%
32	1.00	1.18	+18%
64	1.00	1.24	+24%
128	1.00	1.12	+12%
256	1.00	1.00	0%

- tbench (Wrong Hint) – Bandwidth (More is better)

# of clients	Default (Normalized)	Hint: FORK_AFFINE (Normalized)	Difference (%)
1	1.00	1.00	0%
2	1.00	0.98	-2%
4	1.00	1.00	0%
8	1.00	0.93	-7%
16	1.00	0.95	-5%
32	1.00	0.93	-7%
64	1.00	0.78	-22%
128	1.00	0.64	-36%
256	1.00	0.45	-55%

All benchmarks were run on a dual socket (**2 x 64C/128T) system featuring 3rd Generation EPYC processors**
running modified kernel based on the baseline **tip:sched/core** at **sched-core-2022-08-01**

Discussion

What are your thoughts? What is the way ahead?

AMD

Acknowledgements

We would like to thank the Linux Kernel Scheduler community for its continued efforts at optimizing the Linux Kernel Scheduler for Split-LLC Architectures and testing the changes on various hardware configuration out there.

COPYRIGHT AND DISCLAIMER

©2022 Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD Arrow logo, EPYC and combinations thereof are trademarks of Advanced Micro Devices, Inc. Linux is a registered trademark of Linus Torvalds. Other company, product, and service names used in this publication are for identification purposes only and may be trademarks of their respective companies.

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD

Backup

Data and accounts to aid the discussion

Challenges from the Past

Modelling the MC Domain

Why does it matter?

- Without a properly modelled MC domain, tasks that are waking up, would have been placed on an idle CPU **without considering** if the CPU is from a **cache-hot LLC**.

Solution

- With a **MC domain** modelled to **represent a group of CPUs sharing the same L3**, we can target the CPUs from a **cache hot LLC** during the task wakeup.
- The **CPU search space** for subsequent task wakeup has been **narrowed down** to relevant cache-hot CPUs.

Side Effects

- If a **latency sensitive task**, which doesn't benefit from cache-hotness, targets an **LLC with no idle CPUs**, it'll be **queued on a busy run queue** until the load balancer is triggered and the task is migrated to an idle CPU. Without the restriction of an MC Domain, the task could have found an idle CPU on the system and wouldn't have been **dependent on the load balancer to find an idle CPU** to run on.

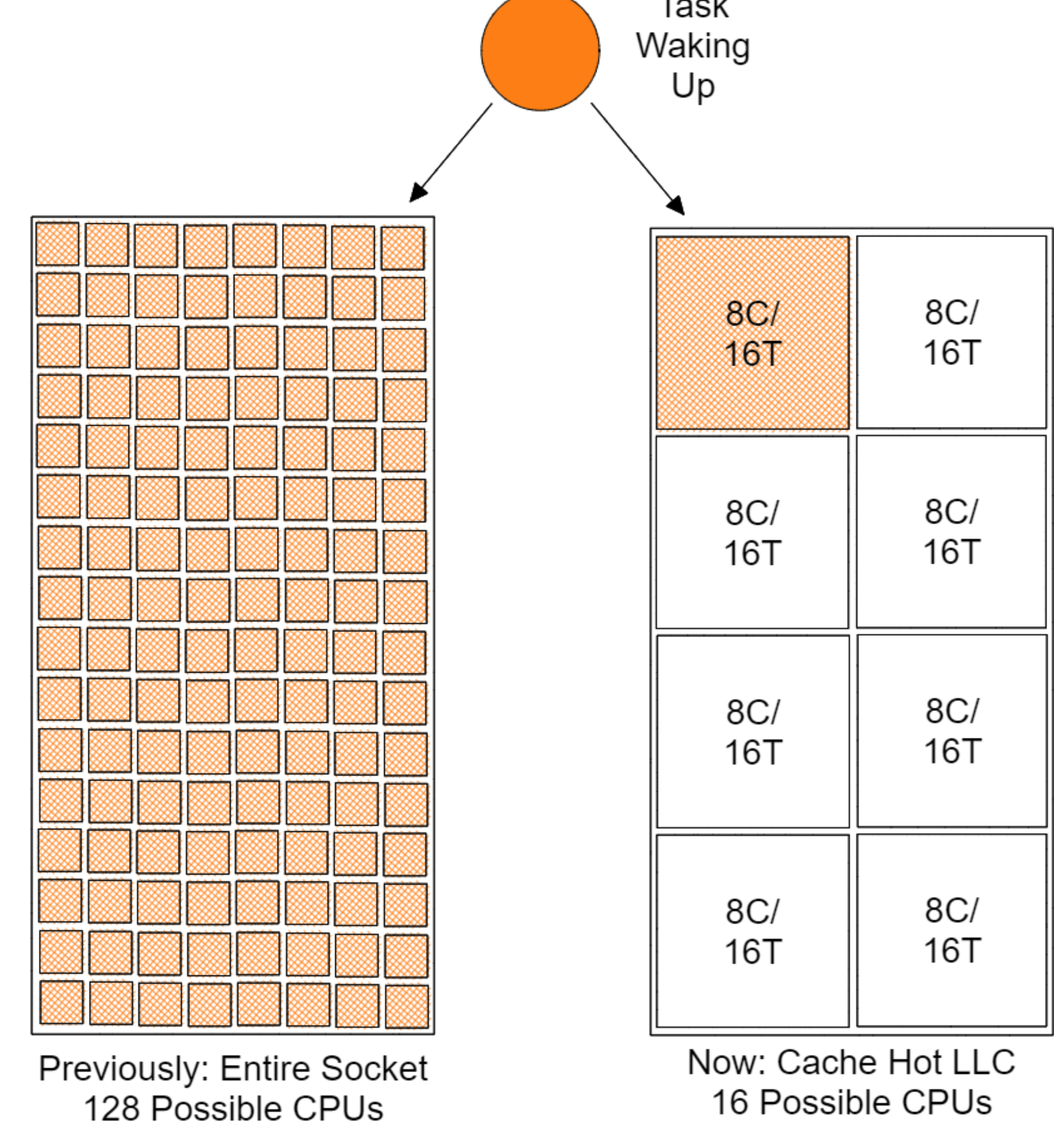


Figure: MC Modelling in EPYC Processors

AMD
together we advance_

Challenges from the Past

Consequence of not having an MC Domain

hackbench – runtime (less is better)

# of clients	tip (Normalized)	tip + CONFIG_SCHED_MC = n (Normalized)	Improvement (%)
1	1.00	1.78	-78%
2	1.00	1.57	-57%
4	1.00	1.45	-45%
8	1.00	1.46	-46%
16	1.00	1.70	-70%

Stream – Bandwidth (more is better)

# of clients	tip (Normalized)	tip + CONFIG_SCHED_MC = n (Normalized)	Improvement (%)
1	1.00	0.57	-43%
2	1.00	0.55	-45%
4	1.00	0.50	-50%
8	1.00	0.51	-49%

tbench – Bandwidth (More is better)

# of clients	tip (Normalized)	tip + CONFIG_SCHED_MC = n (Normalized)	Improvement (%)
1	1.00	0.77	-23%
2	1.00	0.82	-17%
4	1.00	0.91	-9%
8	1.00	0.90	-10%
16	1.00	0.93	-7%
32	1.00	0.94	-6%
64	1.00	1.02	+2%
128	1.00	1.25	+25%
256	1.00	1.11	+11%

All benchmarks were run on a dual socket (**2 x 64C/128T**) **system featuring 3rd Generation EPYC processors** running modified kernel based on the baseline **tip:sched/core** at **sched-core-2022-08-01**

[AMD Official Use Only - General]

Side Effects

Consequence of limiting search space

tbench – Bandwidth (More is better)

# of clients	tip (Normalized)	tip + CONFIG_SCHED_MC = n (Normalized)	Difference (%)
1	1.00	0.77	-23%
2	1.00	0.82	-17%
4	1.00	0.91	-9%
8	1.00	0.90	-10%
16	1.00	0.93	-7%
32	1.00	0.94	-6%
64	1.00	1.02	+2%
128	1.00	1.25	+25%
256	1.00	1.11	+11%

schbench – tail latency (Less is better)

# of workers	tip (Normalized)	tip + CONFIG_SCHED_MC = n (Normalized)	Difference (%)
1	1.00	0.71	+29%
2	1.00	0.64	+36%
4	1.00	0.80	+20%
8	1.00	0.89	+11%
16	1.00	0.84	+16%
32	1.00	0.99	+1%
64	1.00	1.03	-3%
128	1.00	0.99	+1%
256	1.00	0.99	+1%

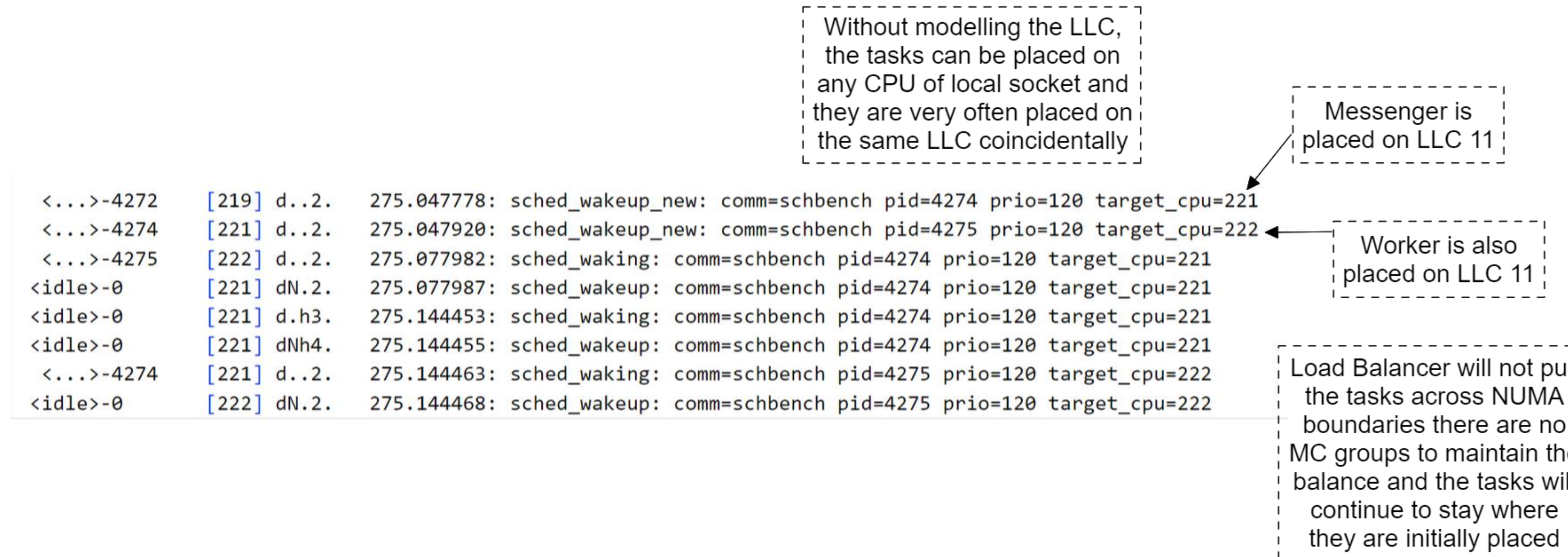
All benchmarks were run on a dual socket (**2 x 64C/128T) system featuring 3rd Generation EPYC processors**
running modified kernel based on the baseline **tip:sched/core** at **sched-core-2022-08-01**

AMD
together we advance_

[AMD Official Use Only - General]

Why schbench improves after disabling CONFIG_SCHED_MC?

- Following sched tracepoints can be enabled to observe the reason for the improvements:
  - **sched_wakeup_new**: To verify the LLCs where the tasks are initially places
  - **sched_waking**: To verify if a migration is a wakeup migration or a load balancer migration
  - **sched_wakeup**: To verify if a migration is a wakeup migration or a load balancer migration
  - **sched_migrate_task**: To track task movement through out the system
- Without the MC Domains to detect the split-LLC design, the NUMA imbalance value is now 16 for the dual socket system.
- More often than not, most schbench threads are placed on the same LLC



AMD together we advance_

[AMD Official Use Only - General]

Challenges from the Past

NUMA Imbalance

Why does it matter?

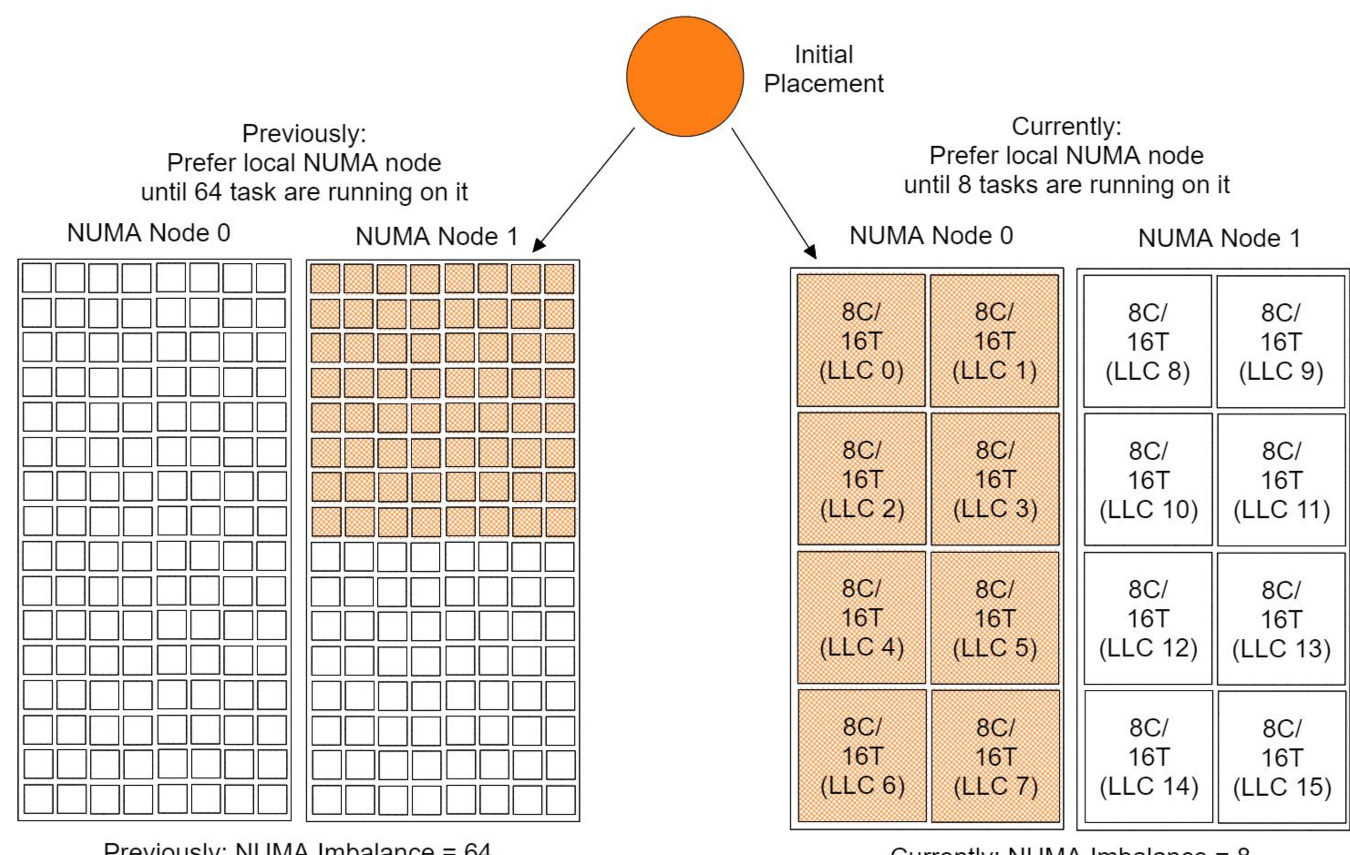
- With a generous threshold of **25% of total CPUs** in the sched domain with **SD_NUMA** flag set, a dual socket offering wouldn't have placed any task initially on an external NUMA node until the threshold is crossed.
- Bandwidth oriented workloads, such as **Stream**, took a huge **toll on performance** due to the inevitable **piling up of tasks on same LLC** leading to cache contention.

Solution

- With Mel's **NUMA imbalance** rework, the threshold for split-LLC architectures is set to **number of LLCs in NUMA nodes**, the Stream threads are placed ideally right from the start.

Side Effects

- **Communicating tasks are now spread across NUMA boundaries early** on thus relying on subsequent wakeups for task consolidation.
- With lower imbalance threshold, the initial placement of Stream threads is **more sensitive to external tasks** running in system and can cause run-to-run variance.



We would like to thank Mel Gorman, and everyone involved in development, discussion, and testing of the NUMA Imbalance rework.

AMD
together we advance_

Challenges from the Past

NUMA Imbalance Rework

Following are the results comparing the results of Stream on dual socket **2 x 64C/128T system featuring 3rd Generation AMD EPYC processors** before and after Mel's rework:

Stream (10 runs) (NPS1) – Bandwidth (more is better)

Stream Kernel	Before Rework (Normalized)	After Rework (Normalized)	Difference (%)
Copy	1.00	1.33	+33%
Scale	1.00	1.70	+70%
Add	1.00	1.70	+70%
Triad	1.00	1.70	+70%

Stream (100 runs) (NPS1) – Bandwidth (more is better)

Stream Kernel	Before Rework (Normalized)	After Rework (Normalized)	Difference (%)
Copy	1.00	1.70	+70%
Scale	1.00	1.69	+69%
Add	1.00	1.79	+79%
Triad	1.00	1.74	+74%

Stream (10 runs) (NPS2) – Bandwidth (more is better)

Stream Kernel	Before Rework (Normalized)	After Rework (Normalized)	Difference (%)
Copy	1.00	2.67	+167%
Scale	1.00	3.46	+246%
Add	1.00	3.35	+235%
Triad	1.00	3.37	+237%

Stream (100 runs) (NPS2) – Bandwidth (more is better)

Stream Kernel	Before Rework (Normalized)	After Rework (Normalized)	Difference (%)
Copy	1.00	2.71	171%
Scale	1.00	2.47	147%
Add	1.00	2.67	167%
Triad	1.00	2.59	159%

AMD
together we advance_

[AMD Official Use Only - General]

Challenges from the Past

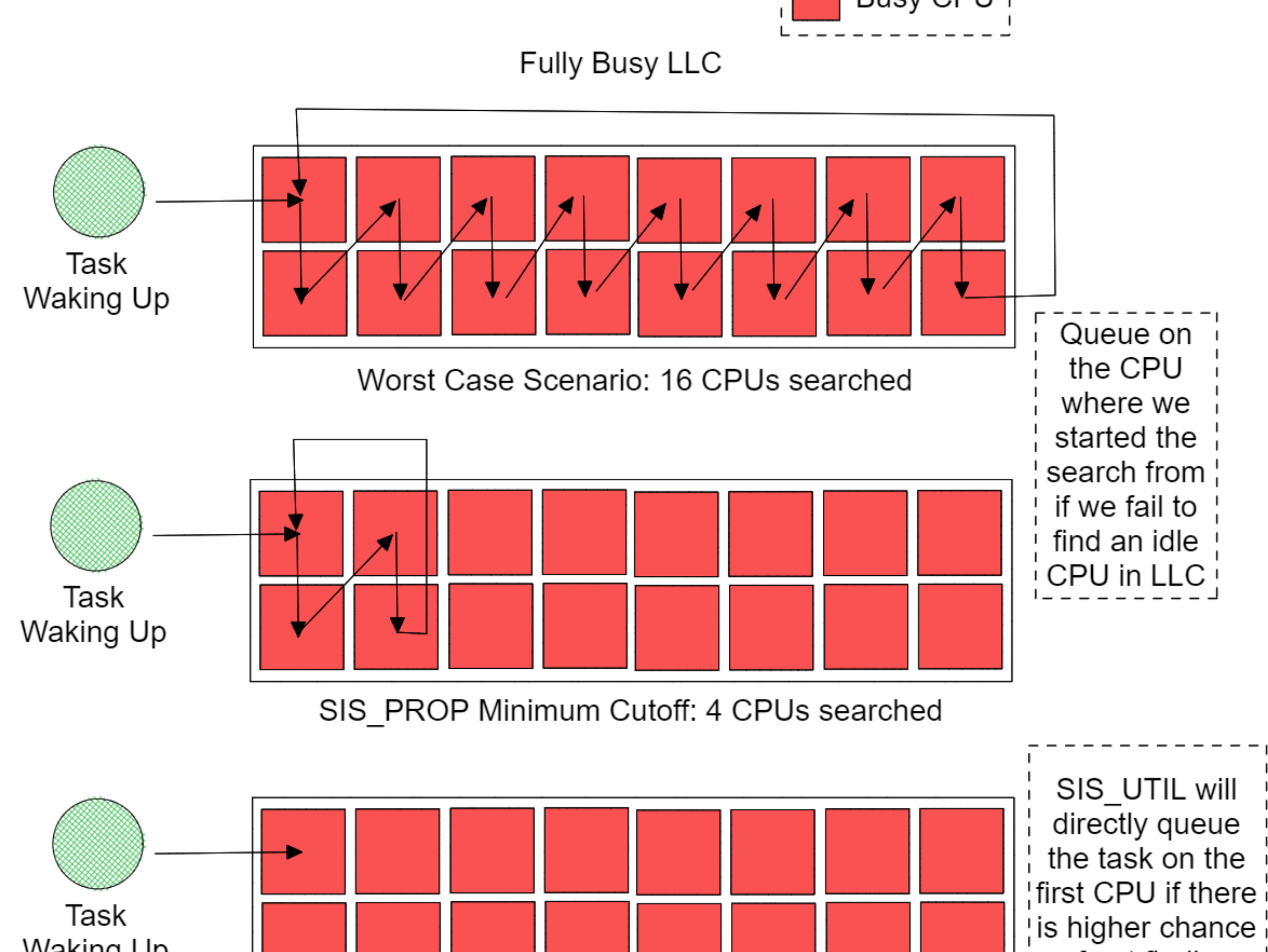
Search latency in a busy LLC (**Thank you Chen Yu!**)

Why does it matter?

- In the absence of an idle core in an LLC, the **Stop Idle Search** algorithm previously used the **average LLC scan cost** and amount of time a CPU was idle to limit the search space for an idle CPU with a lower limit set to 4.
- The metric was not only inaccurate but also led to wasted search effort when LLC is fully loaded.

Solution

- With the **SIS_UTIL** algorithm, we can better estimate how idle an LLC is based on its utilization and limit the search more accurately.
- The **initially proposed linear function was not optimal for split-LLCs**.
- Based on the feedback in the community, a **quadratic function** was adopted, which allowed for a **larger search of the LLC space when LLC was less utilized** and **cut off the idle CPU search when LLC was overloaded**.
- With **SIS_UTIL** algorithm, the **run-to-run variance** observed in tbench when system is fully loaded **disappeared and a stable 79% improvement was observed** for the same.



We would like to thank Chen Yu, and everyone involved in development, discussion, and testing of the SIS_UTIL algorithm.

AMD together we advance_

schbench: Tracing events leading to variation in tail latencies

Verifying the timeline

The described scenario can be confirmed by enabling the following sched tracepoints:

- **sched_wakeup_new**: To verify the LLCs where the tasks are initially places
- **sched_waking**: To verify if a migration is a wakeup migration or a load balancer migration
- **sched_wakeup**: To verify if a migration is a wakeup migration or a load balancer migration
- **sched_migrate_task**: To track task movement through out the system

Depending on how long it takes for new-idle balance to consolidate the two tasks, the 99th %ile latencies tend to vary a lot

schbench messenger is placed on LLC 6

```
   <...>-4521     [039] d..2.  4199.742324: sched_wakeup_new: comm=schbench pid=4523 prio=120 target_cpu=176
   <...>-4523     [176] d..2.  4199.742530: sched_wakeup_new: comm=schbench pid=4524 prio=120 target_cpu=063
schbench-4524     [063] d..2.  4199.772654: sched_waking: comm=schbench pid=4523 prio=120 target_cpu=176
  <idle>-0        [176] dNh2.  4199.772685: sched_wakeup: comm=schbench pid=4523 prio=120 target_cpu=176
  ...
schbench-4524     [063] d..2.  4200.432480: sched_waking: comm=schbench pid=4523 prio=120 target_cpu=176
  <idle>-0        [176] dNh2.  4200.432490: sched_wakeup: comm=schbench pid=4523 prio=120 target_cpu=176
  <idle>-0        [176] dNh5.  4200.432493: sched_wakeup: comm=kworker/176:2 pid=4494 prio=120 target_cpu=176
schbench-4524     [063] d..2.  4200.432499: sched_migrate_task: comm=schbench pid=4523 prio=120 orig_cpu=176 dest_cpu=63
  ...
  <idle>-0        [063] d.h3.  4200.497745: sched_waking: comm=schbench pid=4523 prio=120 target_cpu=063
  <idle>-0        [063] dNh4.  4200.497752: sched_wakeup: comm=schbench pid=4523 prio=120 target_cpu=063
   <...>-4523     [063] d..2.  4200.497763: sched_waking: comm=schbench pid=4524 prio=120 target_cpu=063
   <...>-4523     [063] d..2.  4200.497764: sched_migrate_task: comm=schbench pid=4524 prio=120 orig_cpu=63 dest_cpu=184
  <idle>-0        [184] dNh2.  4200.497797: sched_wakeup: comm=schbench pid=4524 prio=120 target_cpu=184
```

schbench worker is placed on LLC 7

kworker and schbench task wake up on the same CPU

new-idle balance will pull task to the CPU where other schbench was previously running

When the worker wakes up, it finds the CPU where it previously ran busy and finds an idle CPU in the same LLC

AMD
together we advance_

MM Statistics to Influence Initial Task Placement

Predicting Task Behavior from Memory Footprint

Initial Task Placement Strategy

- Use **total number of pages allocated by a task as a proxy for the LLC utilization** by the task.
- Mark **an LLC as overloaded if it has no more idle CPUs** or if the **sum of memory footprint of the tasks running is the LLC is 4 times the size of the LLC**.
- Use the **best-fit algorithm** to bias the task placement of incoming task towards an LLC with **the smallest memory-hole** that can **fulfil the memory requirement of the incoming task without overloading**.
- In case all **LLCs are overloaded** or cannot accommodate the memory footprint of the incoming task, **use the current logic based on number of idle CPUs**. In case of a **tie between the number of idle CPUs**, the **total memory footprint** of tasks running in the LLC is used as a **tie-breaking metric**.

Memory Footprint
Incoming Task
Possible LLC Choices
Has Capacity
Has Capacity
Overloaded
Look at LLCs with spare capacity
Can Accommodate
Will lead to Overloading
Select LLC that can accommodate the task

Memory Footprint
Incoming Task
Possible LLC Choices
Overloaded (idle_cpus: 5)
Overloaded (idle_cpus: 5)
Overloaded (idle_cpus: 3)
Select LLC with most number of idle CPUs
idle_cpus: 5
Heavily Overloaded
Slightly Overloaded
Select slightly overloaded LLC

Figure: Illustration of task placement strategy based on memory footprint of task running in an LLC

[AMD Official Use Only - General]

MM Statistics for Initial Task Placement

hackbench – runtime (less is better)

# of groups	Default Logic (Normalized)	MM Statistics Rework (Normalized)	Difference (%)
1	1.00	0.93	+6%
2	1.00	0.97	+3%
4	1.00	0.97	+3%
8	1.00	0.94	+6%
16	1.00	0.99	+1%

Stream – Run to Run Variance (less is better)

Stream Kernel	Default Logic (Normalized)	MM Statistics Rework (Normalized)	Difference (%)
Copy	1.00	0.38	-63%
Scale	1.00	0.40	-60%
Add	1.00	0.40	-60%
Triad	1.00	0.39	-61%

tbench – Bandwidth (More is better)

# of clients	Default Logic (Normalized)	MM Statistics Rework (Normalized)	Difference (%)
1	1.00	1.01	+1%
2	1.00	1.01	+1%
4	1.00	0.96	-4%
8	1.00	1.02	+2%
16	1.00	1.04	+4%
32	1.00	1.01	+1%
64	1.00	1.02	+2%
128	1.00	1.04	+4%
256	1.00	0.76	-24% *

* **Note**: tbench regresses at higher worker count as all tasks wake up on parent's LLC since we only account the memory footprint of running tasks and the tbench tasks sleep soon after the initial wakeup thus leading to overloading.

schbench – tail latency (Less is better)

# of workers	Default Logic (Normalized)	MM Statistics Rework (Normalized)	Difference (%)
1	1.00	0.69	+31%
2	1.00	0.57	+43%
4	1.00	0.66	+33%
8	1.00	0.70	+30%
16	1.00	0.78	+22%
32	1.00	0.88	+12%
64	1.00	0.94	+6%
128	1.00	0.99	+1%
256	1.00	0.97	+3%

All benchmarks were run on a dual socket (**2 x 64C/128T**) **system featuring 3rd Generation EPYC processors** running modified kernel based on the baseline **tip:sched/core of 5.17.0-rc5**

[AMD Official Use Only - General]

Userspace Hinting : schbench with Two Level Wakeup

- **schbench (correct hints) – tail latency (Less is better)**

# of clients	Default (Normalized)	Hint: FORK_AFFINE + WAKE_HOLD + WAKE_WIDE (Normalized)	Difference (%)
1	1.00	0.81	**+19%**
2	1.00	0.96	**+4%**
4	1.00	1.00	**+0%**
8	1.00	0.91	**+9%**
16	1.00	0.95	**+5%**
32	1.00	1.00	**+0%**
64	1.00	0.93	**+7%**
128	1.00	0.97	**+3%**
256	1.00	0.96	**+4%**

All benchmarks were run on a dual socket (**2 x 64C/128T) system featuring 3rd Generation EPYC processors**
running modified kernel based on the baseline **tip:sched/core** at **sched-core-2022-08-01**

AMD