Abstract ID : **452**

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

Content

Abstract

Modern CPUs fetch data in fixed-size cache lines (typically 64 B or 128B) expecting spatial locality: data placed close in memory are often accessed together. However, large and complex kernel structures are often organized in ways that do not reflect runtime access patterns, leading to unnecessary cache-line fetches and reduced efficiency. Prior work on data-structure splicing (DSS) shows that reorganizing structures (e.g: reordering fields, splitting or merging structures) can improve spatial locality by collocating frequently co-accessed fields. This has been demonstrated in earlier research such as "A Unifying Abstraction for Data Structure Splicing (MEMSYS'19)" [1] . In the Linux kernel context, we believe a similar approach can benefit scheduler structures such as struct rq, as well as other widely used structures.

Prior Work

Recent proposals in the kernel community have focused on reordering fields based primarily on hot vs cold access frequency. These methods group frequently accessed ("hot") fields together and separate less frequently accessed ("cold") fields. While this approach is useful, it does not necessarily capture spatial locality: whether fields are actually accessed together within short time windows. Our methodology differs by explicitly considering co-access affinity, not just raw access counts, enabling more locality-aware placement.

Some similar work have already been proposed upstream for evaluation - for example, recent patches have explored structure layout tuning by reordering fields based on access frequency [2] [3]. One such patch in the networking subsystem [4], which has already been merged and demonstrated significant TCP performance gains through field reordering. These efforts highlight the performance gains achievable through structure layout optimization. Our work extends this approach by analyzing co-temporal access patterns from execution traces and aims to deliver a generic tool [5] that automates such optimizations across data structures, achieving more locality-aware layouts beyond frequency-based heuristics.

Access-Affinity Profiling Methodology

Our approach builds on the idea of "access affinity" the observation that fields accessed close together in time should ideally be placed close together in memory. To capture this, we analyse memory access traces of selected structures and measure the "closeness" of field accesses. If two fields are accessed within a short window of intervening operations, we record an affinity event between them. Over the course of a trace, these events accumulate into a measure of how strongly pairs of fields tend to be accessed together.

From this data we construct an access-affinity graph:

- Each node represents a field of the structure.
- Each edge between two fields carries a weight proportional to the number of affinity events observed for that pair.

Clusters of fields connected by strong affinities represent natural groups that should be laid out contiguously in memory. By arranging these clusters within cache-line boundaries, we improve spatial locality: fetching

one field into the cache also brings in its frequently co-accessed neighbours, reducing cache misses. Fields with weaker or rare relationships can remain separated, or be grouped into cold sections.

This abstraction provides a systematic way to translate runtime access patterns into layout recommendations, ensuring that the structure better reflects its real-world usage.

Prototype Implementation and Early Results

In the current implementation, static tracepoints are manually inserted for each load and store of rq structure fields within the scheduler subsystem to collect field-level access traces during representative workloads. These traces are then analyzed to compute co-access frequencies, and hierarchical clustering is applied to derive potential reorderings of structure fields for improved spatial locality.

We evaluated the prototype using the wait stressor benchmark [6] a microbenchmark designed to exercise the wait() and waitpid() system calls under intense signaling and process state transitions. The benchmark launches multiple worker processes, each continuously spawning two subprocesses:

- A runner process that repeatedly executes pause() and remains idle until signaled.
- A killer process that rapidly alternates the runner's state by issuing SIGSTOP and SIGCONT, forcing the kernel scheduler to update task states and trigger frequent waitpid() wakeups.
- The parent repeatedly calls waitpid() and wait() to catch these transitions, effectively generating a steady stream of wait-related events that stress scheduling and task management paths in the kernel.

This workload is particularly suitable because it involves frequent transitions between runnable and stopped states, which exercise per-CPU runqueue management a structure rich in cross-field accesses.

Performance Results

We compared the baseline kernel layout with a reordered version derived from the access-affinity clustering.

Baseline (before reordering):

36,200,729,522 cache-misses \# 13.878% of all cache refs
260,843,781,303 cache-references

cache-misses events (top symbols):

22.89% 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 field reordering (affinity-based layout):

25,141,476,949 cache-misses # 9.442% of all cache refs
266,268,846,412 cache-references

cache-misses events (top symbols):

24.99% wait update_sd_lb_stats.constprop.0
3.11% wait idle_cpu
2.05% swapper enqueue_task_fair
1.92% wait __schedule

Across multiple runs, the cache-miss ratio dropped from roughly 13-14% to 9-11%, representing a 15-20% reduction in misses. This improvement correlates with the reorganization of frequently co-accessed fields especially those used in idle accounting and balance decision logic being brought into closer proximity.

Access-Affinity Clustering

The access graph analysis identified several strongly connected components, corresponding to groups of fields that are frequently accessed together within short temporal windows. Reordering the structure fields based on these affinity groups for instance, placing fields related to idle CPU checks (such as idle and curr) next to each other improved spatial locality within cache lines.

These findings demonstrate the effectiveness of access-affinity-driven structure reordering. While initial results show measurable improvements in cache efficiency along critical scheduling paths, maintaining such

optimized layouts requires continuous adaptation. When new fields are added to performance-critical structures, it becomes necessary to re-run the tool to re-evaluate and reorder the layout, ensuring that spatial locality remains optimal over time.

The proposed methodology and tooling can aid developers in making data-driven layout decisions—guiding where to place new fields, detecting potential false-sharing scenarios, and quantifying the performance implications of structural modifications prior to merging patches.

Discussion and Future Work

- Profiling methodology: Currently, we manually insert static tracepoints at every load and store instruction to collect memory access traces. However, this approach is not scalable. A more generic and automated framework for capturing structure-level access patterns is needed. Alternatively, instruction tracing could be used to derive load/store access patterns without manual instrumentation. Sampling-based tools such as perf mem record offer only partial visibility into the true access order, limiting their usefulness for detailed profiling.
- Workload selection: Selecting the appropriate benchmarks or workloads for each structure is crucial to capture meaningful access patterns and ensure that the resulting layout optimizations deliver performance benefits under realistic system conditions.
- Cache-line layout optimization: Different architectures have varying cache-line sizes (e.g: 64 B vs. 128 B). Discussions are needed on how to reorder fields and cluster data optimally based on the target cache-line size to maximize spatial locality while avoiding false sharing.
- Target structures: While our prototype focuses on the per-CPU runqueue, this methodology could be applied to other performance-critical kernel structures. Discussion is needed to identify which structures are most suitable for access-affinity-based reordering and where the potential gains are highest.
- Static vs. dynamic analysis: While we are currently exploring dynamic profiling, static analysis could complement this by predicting likely access patterns without runtime overhead. A hybrid approach combining both techniques may provide better guidance.
- Handling Structure Updates: Whenever a new field is added to a structure, the tool needs to be re-run and the structure needs to be reordered to maintain optimal layout.

Conclusion

Access-affinity-based struct reordering can improve spatial locality by collocating fields that are frequently accessed together, as demonstrated in prototype experiments on the per-CPU runqueue. Compared to prior hot/cold frequency-based reordering, this method captures temporal co-access patterns more effectively. If there is a potential generic tool that can be developed along these lines, it could be used for optimizing frequently accessed large kernel structures. We invite discussion on generalizing the tooling, refining the methodology, and identifying the right structures and workloads where this approach can bring real benefits.

References

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

2 https://lore.kernel.org/all/20250730205644.2595052-1-blakejones@google.com/

3 https://lore.kernel.org/all/20250402212904.8866-1-zecheng@google.com/

4 https://lore.kernel.org/netdev/20230916010625.2771731-1-lixiaoyan@google.com/

5 https://github.com/AboorvaDevarajan/kstruct-tuner/

6 https://github.com/AboorvaDevarajan/kstruct-tuner/blob/main/workloads/sched/wait_stressor/wait_stressor.c

Primary authors: DEVARAJAN, Aboorva; K A, Nysal Jan

Presenter: DEVARAJAN, Aboorva

Track Classification: Linux System Monitoring and Observability MC

Submitted by **DEVARAJAN, Aboorva** on **Friday 10 October 2025**