Profiling Cache and Thread Migration Patterns in Modern Linux Schedulers

Prakhar Gupta
Ingi Helgason
Pradyun Narkadamilli

I. Introduction

Across production and internal servers, Linux is one of the most ubiquitous operating systems on the planet. In many of these environments, there is significant effort put forth in engineering hardware-level, OS-level, and application-level support to improve throughput and performance. At the OS-level, a topic that has seen renewed interest in recent years is the scheduler.

While one important responsibility of the scheduler is to fairly share a single processor's hardware resources between resident workloads, another increasingly important problem is how threads are (re)assigned to the many logical CPUs present in the system. Though extensive work has been done to ensure that the amount of work each processor is doing is fair, even accounting for heterogeneity in processor performance (i.e big.LITTLE), it does not seem like optimizing for cache affinity placement is an express development goal in the mainline Linux scheduler.

Recently, a team at Intel has begun working with other Linux maintainers on an extension to the Linux Scheduler called "Cache-Aware Scheduling". They have posted promising results on the Linux mailing list [1]. Similarly, SCX_LAVD [2] is a BPF scheduler which is designed to have improved handling of interactive and latency-critical workloads. Extended work by Meta [3] applies this scheduler work to server architectures, showing promising results when handling machines with multiple sockets and LLC domains.

However, despite these visible initiatives, little data is available in the way of scheduling patterns, performance counting, or bottleneck analysis of scheduler behavior. As seen in [1], latency or throughput figures can improve on various 2-socket systems. Similarly, in [3], the cache miss rate is shown in addition to latency and throughput. However, figures beyond these are not provided by either project.

This project aims to bridge that gap by investigating performance counter data and scheduler records on a representative benchmark suite running on a Sapphire Rapids server. We study the cache-level and scheduler-level behavior of benchmarks under two schedulers: Linux's default scheduler, and the SCX_LAVD scheduler using a virtual LLC configuration similar to Meta's reported usage.

The paper is structured as follows. Section 2 (Background) presents analysis of the Linux scheduler, Intel's "Cache Aware Scheduling" patch, and the SCX_LAVD scheduler. Section 3 introduces the methodology we apply to profile cache and scheduler behavior, which we use to analyze characteristics of both the default Linux scheduler and the SCX_LAVD scheduler. Section 4 then discusses experimental results and their interpretation. Finally, in Section 5, we summarize our contributions, discuss limitations in our methodology, and describe what we believe future work should aim to address, based on our findings at this time.

II. Background

This section describes the architecture of the Linux scheduler, its properties and issues in many processor systems, and existing efforts to remedy these issues. To emphasize the history of thread migration phenomena and cache utilization, we begin with general discussion more relevant to uniprocessor systems. Later, multiprocessor considerations are reintroduced, and a number of recent solutions and areas of focus are introduced.

A. The Linux Scheduler

The Completely Fair Scheduler (CFS) [4], which began its tenure as Linux's default scheduler in 2007, has the job of assigning CPU time to different threads on the processor's *run queue*. More specifically, it divides the scheduler interval (a fixed quantity) into *timeslices* which are sub-intervals with lengths determined based on a *weight*, which estimates the need to schedule each thread.

It is important to note that the default scheduler is only responsible for tasks annotated with normal scheduling priorities (`SCHED_NORMAL|SCHED_IDLE|SCHED_BATCH`), where a real-time policy is not required and time is supposed to be shared fairly. The vast majority of tasks in a Linux system receive one of these scheduling priorities (typically, `SCHED_NORMAL`), and as such [5], fall under the purview of the default fair scheduler. The real-time and deadline based scheduling strategies are beyond the scope of this evaluation, as they are not seen in typical, unconfigured use cases. Further, in the case of real-time policy, preclude the possibility of migrations.

It is well-documented that Linux maintainers believe that the "CFS algorithm", weighted fair queueing, has some shortcomings, and the amount of time given to each thread is the main invariant it actually defines [6]. Weighted fair queueing has been considered fragile because of its dependence on fragile heuristics, and has had various bugs in its ability to deliver its

expected behavior [7]. Its temporal behavior (pattern of time slice assignments or consistent under-provisioning of CPU time) can be difficult to control. Per our understanding, this was the primary motivation for Linux's recent move to the Earliest Eligible Virtual Deadline First (EEVDF) scheduler.

The new policy EEVDF is similar in its goals and in that it divides a time interval fairly across processes [8]. Only permitting the scheduling of threads "owed" runtime by the scheduler guarantees distribution of the bandwidth over longer, temporal intervals. Most importantly, EEVDF allows Linux kernel developers to reduce the number of required heuristics [9], and move towards an algorithm with more proven and understood mathematical properties.

B. EEVDF

Fairness between different tasks is handled by maintaining a "virtual lag" value for each scheduler entity, which indicates if a task has received more or less service time than it should receive in a "fair" environment. The Linux scheduler then only makes an entity eligible for scheduling only if it has received less service time than it is supposed to have (i.e., if the virtual lag is positive).

Latency is handled by maintaining virtual deadlines for each task. For each task, a value δ is added to the first eligible time for a task, where δ is a function of the fixed timeslice and that task's weight (based on its "nice" value). The eligible task (by the aforementioned rule) with the nearest virtual deadline is scheduled first.

The EEVDF policy is designed to fairly order tasks within a single runqueue (which corresponds to a single logical CPU). However in today's computing landscape, the operating system not only has a large number of tasks to choose from to execute, but also many physical (and logical) processors that it can dispatch them to.

C. Multiprocessor Systems

As the scheduler above is responsible for fair scheduling on a single core, extending the Linux scheduler for multicore systems led to the introduction of a *load balancer* module. This component is critical for ensuring that the system can be throughput efficient. To observe this effect more clearly, consider a pathological worst-case example, wherein 10 largely independent threads are all present in one CPU's run queue, but few exist on other processors. If we assume the scheduler is without bugs, then the fairness guarantees of the scheduler will lead to steady progress on all threads. However, throughput will be higher if idle threads can be moved to processors with run queue availability.

Linux schedulers generally maintain per-CPU run queues. We identify two salient reasons for this. One is that dispatching from the run queue requires exclusive access. In Linux, this structure is protected by a spinlock to guarantee mutual exclusion. With large processor counts, contention for a single global run queue could become prohibitively large. Since context switches are frequent, unpredictable dispatch delays can create a large amount of waste. Additionally, keeping run queues processor-local helps them stay valid in the private cache hierarchy. Scalability factors will reappear in the discussion of newer proposed solutions.

Because of the run queue architecture, the current Linux load balancer module functions by periodically activating via interrupt (softirq), "pulling" work from busy cores to free cores. During load balancing, the kernel iterates by starting with scheduling domains the idle, stealer CPU belongs to, determining which domains (starting from the bottom up) are eligible for load balancing. Here, the core which will have tasks taken from its runqueue is called the *stealee*, as it is having its tasks taken. A stealee is identified by selecting the busiest core in the busiest core-group, which acts as a proxy for the most overloaded core. After a stealee is selected, the `detach_tasks()` function will be called to attempt to detach entries from its run queue. The exact metric that the load balancer focuses on is controlled by `enum migration_type`.

While iterating through the stealee's runqueue, the `can_migrate_task()` function is called to determine if a thread can be migrated, and if it should be migrated. A set of heuristic functions is called to override if a task can be migrated, to try to prevent harmful migrations. The main goal of this check is, perhaps obviously, to prevent impossible migrations from being attempted (such as the migration of currently-running or CPU-locked tasks). However, the function will also assess other heuristics and conditions to block migration; for instance, it will tend to not migrate a task if an excessive number of migrations is already occurring. The current set of behaviors which block migration is fairly modest, and can be found in `kernel/sched/fair.c` [10].

Notably, the current development branch of Linux ostensibly makes some account of cache locality, by preferring to migrate cache-cold tasks, and checking if the destination lies on the same NUMA node (again, see the above reference).

D. Intel: Cache-Aware Scheduling

Cache Aware Scheduling is a set of proposed changes to Linux's fair scheduler. As of the writing of this document, four revisions of this patch set have been shared for review. This analysis focuses on the contents of the v3 and v4 patches, which have a great deal in common.

The key modification made in the Cache Aware Scheduling patch (henceforth, CAS) is to assign to each thread a preferred LLC. Preferred LLCs are assigned based on which threads share virtual address spaces (`mm_struct`). It takes this approach since threads in the same process are likely to share data. With this assignment, two behaviors are prioritized [11].

Namely,

1) Tasks are prevented from migrating from their preferred LLC if not necessary.
2) Tasks are migrated to their preferred LLC if necessary.

The exact details of how pulling is discouraged will not be discussed here, as this patch set is not part of our study, due to technical limitations (discussed in Section 5).

Testing of the patch identified performance degradation when the number of active threads belonging to a process exceeds the number of true cores (disregarding SMT) in the LLC domain. We think this makes sense, because in these cases, the load balancer would be otherwise impeded (by the patch's modifications) from performing a necessary behavior to spread tasks across processors. In these cases, cache aware scheduling behavior is disabled.

Ultimately, in the current patch, balancing is implemented entirely by guiding the load balancer to perform cache-aware aggregation during periodic balancing. The authors comment that other strategies may be explored in the future.

Their evaluation centers on the hackbench and schbench benchmarks, generally run with few threads per process. One notable benchmark featured across patchset versions is an 8 threaded Verilator simulation of the XiangShan processor running ChaCha20, where a major improvement to throughput (24%) is noted.

More generally, their evaluation consistently shows no degradation or latency/throughput improvements. In the v4 patch [12], the authors specifically note that significant improvements are noticed when the system is underloaded. Beyond these time metrics, however, we cannot find more detailed information or benchmarking online.

E. Meta: SCX LAVD

Another parallel branch of work, performed by Meta, explores the usage of the Latency-criticality Aware Virtual Deadline (LAVD) scheduler. LAVD is an open source scheduler based on the Linux kernel feature sched_ext, which allows kernel thread schedulers to be implemented in userspace using eBPF, and loaded into the kernel at runtime. This scheduler was originally developed by Igalia, under contract with Valve, to improve the performance of the Steam Deck gaming platform [13]. Beyond this use case, Meta announced at the Linux Plumbers Conference that they use this as a default scheduler for their servers [14], whenever a specialized scheduler is not used.

At an algorithmic level, LAVD is quite comparable to the EEVDF scheduler policy. Meta contributed a series of changes to the scheduler, with the intention of making it more suitable for various server CPU architectures and workloads, which they detail in their talk. In public discussion, they describe running various targeted tests to evaluate scheduler performance. The main workload types they discuss running are large, user-facing services with L2 usage-correlated throughput, and workloads with processor-pinned, latency-sensitive Erlang `erts_sched` tasks.

The key changes which Meta contributes is the usage of `virtual LLCs` [15], with private dispatch queues, or DSQs. DSQs are analogous to the role of run queues in a typical kernel-mode scheduler. Tasks which wake up or yield are delivered to the sched_ext class, if one is enabled. The eBPF program can then suggest which dispatch queue to place this task in. Whenever a processor finishes a task, it receives the next task from a DSQ that is local to that particular CPU.

The granularity of a DSQ is slightly different from that of a run queue, because they are not inherently per-CPU. For instance, prior to their contribution by Meta [16], the LAVD scheduler actually did not assign per-CPU queues, instead providing one DSQ per LLC domain. It is worth noting that Meta motivates this addition primarily for the case of pinned tasks, which led to significant contention issues when stored in shared DSQs. The significance of this overhead is worth studying.

Meta notes performance degradation when a single large LLC domain (in their example, over 50 CPUs) shares a single DSQ, due to lock contention and frequent migration between CPUs in the large LLC domain. The contribution of virtual LLCs, per Meta's analysis, introduces some tradeoffs but generally, among other things, aims to improve the cache locality ("cache awareness") of scheduling. Sweeping virtual domain sizes, they find that modestly sized domains strike the best balance of work conservation, contention for shared DSQs, and cache locality (due to the smaller set of CPU migration candidates).

Their results are promising, showing good throughput, low miss rates, and a sweet spot for average latency with medium-sized CPUs per domain. However, their benchmark set and evaluation framework are not shared. Further, beyond throughput, latency, and aggregate miss rates (admittedly, useful metrics), little other data exists justifying the quality or impact of these changes.

As far as we can tell, LAVD as applied by Meta is a major work which is understood to improve cache behavior. Based on our literature review and code analysis of this scheduling algorithm, LAVD's virtual LLC partitioning should approach or approximate cache-aware scheduling behavior (as defined above).

Importantly, since it is an eBPF scheduler, LAVD can be dynamically loaded at runtime without the need to compile a custom operating system. As we were unable to gain private access to a suitable server to boot a custom-built operating system, this led us to study and characterize LAVD in comparison to the default Linux scheduler for our study, as a proxy for assessing the general impact of cache-aware scheduling mechanisms as they present themselves in CAS.

III. Methodology

One inconvenient point is that schedulers can be difficult to profile. Our thought process on workload selection evolved throughout the project. The specific workloads selected in our early evaluation and later experiments will be discussed separately alongside our results, in Section 4.

In both cases, the key qualitative data we aim to observe are the following:

1) Quality of processor selection under scheduling schemes
2) Cache implications of scheduler policy
3) Migration patterns
4) Throughput and speedup (or lack thereof)

Typical scheduler benchmarks will attempt to pressure the scheduler to make as many decisions as possible from a fair-

ness perspective. This paradigm inherently encourages benchmarks with high thread counts and frequent oversubscription as a way to benchmark fairness.

However, as our project focuses on the ability of the scheduler to make efficient core placements, we instead wish to increase the number of decisions that the load balancer has to make, as well as the number of options that the load balancer has to choose from. To this end, we consistently try to choose benchmarks that undersubscribe the system's logical core count.

The experiments in this paper were conducted on Dual-Socket Xeon (Sapphire Rapids) servers. The machine prior to the midterm report had 40 physical processor cores per die, while we later transitioned to a machine with 32 processor cores per die. Existing results were all re-benchmarked on the new system to ensure inter-comparability with future results. All processor cores support hyperthreading, but this is selectively turned on/off as discussed in our Results.

To match the LAVD scheduler parameters used by Meta, we configure it to group CPU cores into virtual LLC domains of 4 adjacent cores (e.g each true LLC domain consists of 8 contiguous vLLC domains), and utilize private per-core DSQs rather than a per-vLLC DSQ. These settings both serve to reduce the amount of contention and write-invalidations encountered by the shared scheduler metadata.

While the majority of this section will focus on the evaluation methodology and tooling used in the final, post-midterm experiment, discussion of early motivating investigation will also be discussed in Results.

A. Overview of Testing Harness

Before running experiments, the CPU frequency governor is set to `performance` on all cores. The default governor, `schedutil`, typically varies CPU frequency based on scheduler activity, leading to higher operating frequencies on CPUs with busier run queues.

However, when using an SCX scheduler, the frequency governor seems to continue to monitor state associated with the native Linux scheduler. This leads to significantly lower operating frequencies when benchmarking LAVD naively, and almost 30% increase in runtime across the board in our early testing.

All benchmarks are launched through a common wrapper script that standardizes scheduler selection, socket configuration, performance-counter collection, and runtime logging. Each workload is evaluated in four configurations: default dual-socket execution, default single-socket execution, LAVD dual-socket execution, and LAVD single-socket execution. Single-socket runs are created by temporarily offlining all CPUs on socket 1 (using sysfs) before the workload starts and restoring them after completion.

A performance-collection subscript is invoked for the workload instrumenting `perf stat` using a fixed counter list and records interval-based, per-core PMU data.

For each run, the harness stores the Intel Performance Monitoring Unit (PMU) output in a job-specific directory and records wall-clock runtime separately. After the workload is completed, the wrapper restores any modified system state, including the scheduler process and CPU online state.

TABLE I
PERFORMANCE COUNTERS

Event	Purpose
`task-clock`	CPU time
`duration_time`	wall time
`cycles`	core cycles
`instructions`	retired instr.
`ref-cycles`	ref. cycles
`l1d.replacement`	L1D churn
`L1-dcache-loads`	L1D loads
`L1-dcache-load-misses`	L1D misses
`L1-icache-load-misses`	L1I misses
`icache_data.stalls`	I-cache stalls
`icache_tag.stalls`	I-cache stalls
`memory_activity.stalls_l1d_miss`	L1D miss stalls
`l2_rqsts.references`	L2 refs
`l2_rqsts.miss`	L2 misses
`l2_rqsts.rfo_hit`	L2 RFO hits
`l2_rqsts.rfo_miss`	L2 RFO misses
`longest_lat_cache.reference`	LLC refs
`longest_lat_cache.miss`	LLC misses
`ocr.demand_rfo.l3_hit`	L3 RFO hits
`ocr.demand_rfo.l3_miss`	L3 RFO misses
`ocr.demand_rfo.dram`	DRAM RFO
`ocr.demand_rfo.local_dram`	local DRAM RFO

B. Counters and Performance Metrics

We collect hardware performance data using `perf stat`. A limited number of hardware performance counters can be tracked at any moment in time, but the PMUs will rotate through measuring different samples, and interpolate data for missed samples. The counter set covers runtime, core throughput, private-cache behavior, last-level-cache behavior, and RFO/coherence traffic. Table I summarizes the collected events.

In practice, since the logging interval is not tied to the timeslice of a single process, we are unable to correlate PMU data directly with the processes we are observing. We instead compare full-system PMU behavior across benchmark runs. Since active processes and the benchmarking environment is consistent across testing configurations (barring swept parameters), we consider our PMU data sufficient for relative analysis in our study (though they are likely incomparable to results from other systems).

As a result of considering full-system PMU values, data presented in this study may be subject to a small amount of background system noise, or may result in artificially decreased numbers for the dual-socket system due to a higher number of aggregate background instructions. Despite this, our study consistently shows significant changes in key performance counter values, implying that the true perceived microarchitectural impacts may be more pronounced than immediately apparent from our investigation.

IV. RESULTS

A. Initial Investigation (Midterm Progress)

To profile decision-making and performance baselines for the default Linux scheduler, we run Intel's headline ChaCha20

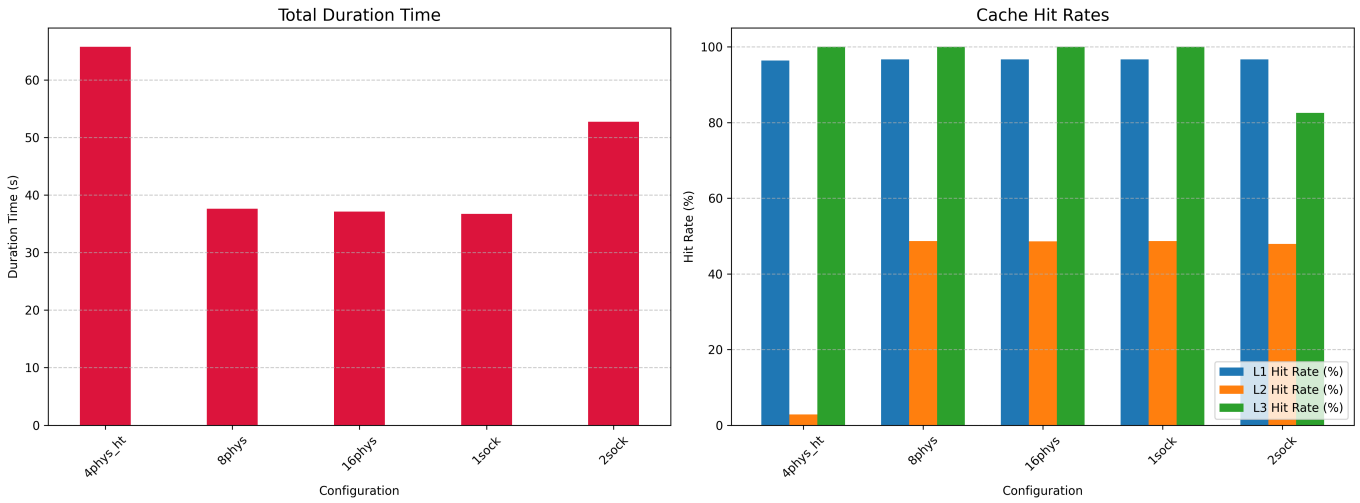

Fig. 1. Basic Metrics for ChaCha20

benchmark across a number of core configurations. The goal at this phase of testing is to determine if there was an apparent "performance ceiling" for the scheduler to chase down, and if so, what level of decision-making freedom had to be removed from the scheduler for that ceiling to be reached by default.

The following settings were tested by restricting scheduling candidates for a process via `numactl` and manually enabling/disabling SMT pairs via `procfs`.

- **4phys_ht**: 4 physical cores + 4 SMT siblings
- **8phys**: 8 physical cores, single socket, no SMT.
- **16phys**: 16 physical cores, single socket, no SMT.
- **1sock**: Full socket (80 logical CPUs), memory-bound.
- **2sock**: Both sockets (160 logical CPUs), no constraints.

We first observed the runtime and basic cache statistics of this benchmark as shown in Figure 1. It is immediately apparent that co-location with SMT siblings is not feasible for this benchmark, as the working set is too large to accommodate two threads' worth of traffic on a single L2 cache (per the L2 hitrate). Our working theory is that this is due to set and capacity conflicts between the two processes. Unfortunately, this cannot be empirically confirmed on real systems via performance counters at runtime, but we felt the performance impact severe enough to warrant disabling SMT for all further testing.

Surprisingly, we see even runtimes and cache statistics between all three of our single-socket non-SMT configurations. This implies that the specific placement of cores within a single LLC domain is not significantly impeding performance on this benchmark. In our testing, we saw runtimes and cache statistics fluctuate in these 3 configurations and, within a reasonable margin of error, are inclined to say that the three configurations exhibit equivalent performance. However, using the entire system, i.e not using `numactl` to restrain scheduling in the `2sock` test, we see a significant increase in duration, as well as a drop in the L3 hitrate.

To better understand this behavior, we reran the baseline

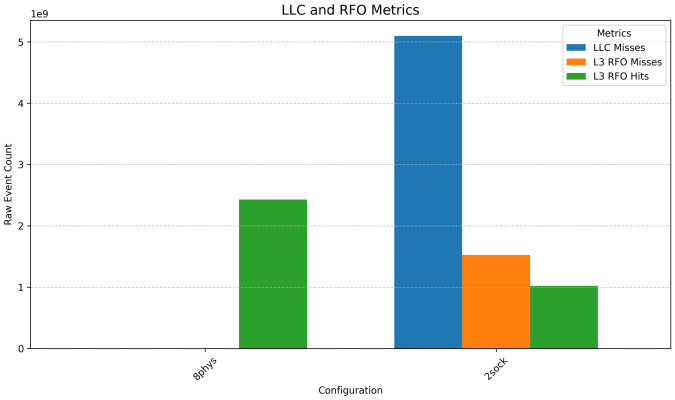

Fig. 2. Read-for-Ownership Metrics for ChaCha20

8-core configuration and the full-system configurations with additional memory performance counters from the PMU, monitoring L3 read-for-ownerships, as seen in Figure 2.

Evidently, there is an extreme rise in the L3 miss rate and RFO misses between the two sockets, to such a degree that the LLC cold start misses for the 8-core configuration isn't visible (though it is nonzero). This is a pattern we see going forwards in that scheduling different threads of the strong data-sharing processes across two sockets causes extreme performance regression due to increased cacheline bouncing.

Since the L3 is shared across all cores of a die (and is the only shared cache in Sapphire Rapids), this is likely also the reason we see homogeneous performance across the different single-socket configurations (as coherency misses between the cores dominates runtime). As a result, at this stage of our testing we inferred that the impact of varying coherence traffic latency due to core placement within an LLC (as a result of network topology) was dwarfed by LLC selection for strong data-sharing workloads.

In our analysis, we noticed that EEVDF does not per-

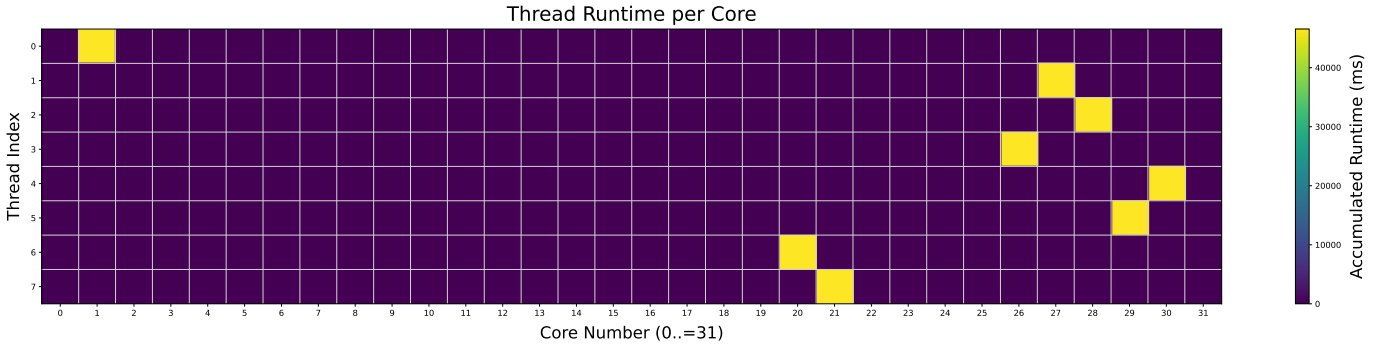

Fig. 3. Default scheduler migrations (ChaCha20 program) on one socket.

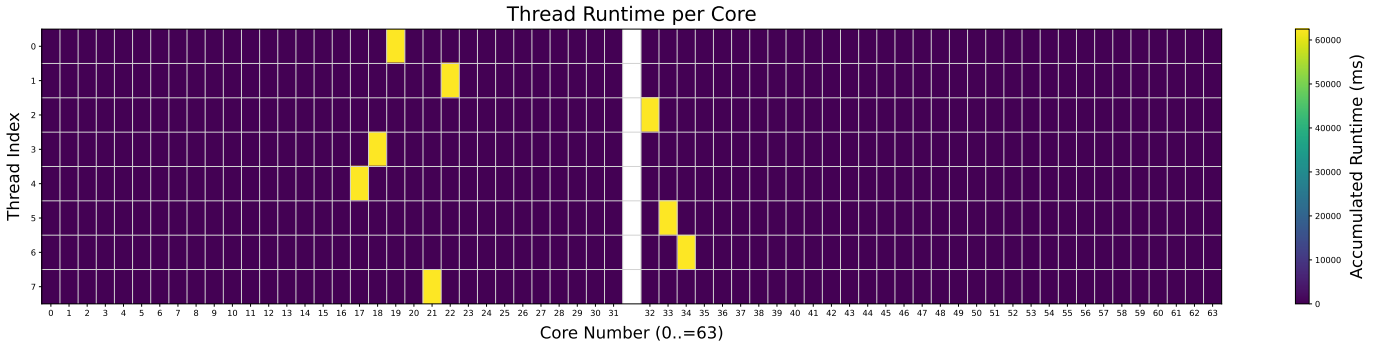

Fig. 4. Default scheduler migrations (ChaCha20 program) on two sockets.

form thread migrations whatsoever during execution of the ChaCha20 benchmark. A graphical representation of limited migration is shown in the heatmap figures, Figures 3 and 4. Even when presented with a large number of load balancing destinations, the default Linux scheduler does not aggressively migrate threads when it is undersubscribed and underloaded. This led us to reject an early theory that underloaded systems (with many load balancing destinations) could trigger excessive thread migrations, leading to performance degradation.

Empirically, the pattern of minimal migrations in undersubscription held true for every benchmark we considered at this time (e.g. hackbench, schbench with low thread counts). Hackbench and schbench results are not included because these synthetic benchmarks show minimal data sharing, and thus were not relevant to our study.

Overall, this realization led to us removing sub-socket core sets from our scheduler evaluation. This further broadens our options for benchmark selection, as it allows us to test with modest thread counts without oversubscribing to system resources.

B. Post-Midterm Results

As mentioned in our original proposal, our post-midterm investigation was oriented as an extended data-gathering and interpretation phase. At this point we began evaluating LAVD in addition to the default Linux scheduler, and ran an extended benchmark selection with a number of performance counter configurations. Due to limitations on the number of PMU counters that `perf` and `PCM` can track during a single run, multiple runs were required per benchmark to aggregate all the scheduler and performance counter data seen below. To maintain accuracy, any directly compared fields (e.g calculating hit/miss ratios or local/remote DRAM ratios) are based on values collected together on a single run.

It should be noted that we observed some level of variance in individual metrics between runs (runtime, most obviously). However, any such variance was qualitatively understood to be within margin of error. Unfortunately our current infrastructure replaces our results in-place between collection runs so we cannot show specific variation estimates, but it is our strong belief that no significant changes in runtime behaviors was encountered between different runs of the benchmarks.

At this stage of our testing, we wanted to pivot our evaluation suite towards benchmarks that were more representative of real-world benchmarking workloads, rather than scheduler-specific benchmarks. By doing so, we can estimate more realistic impacts on performance-sensitive applications, as opposed to classical scheduler benchmarks like `hackbench` or `schbench` which can have contrived forking and data movement patterns.

We continue to benchmark XiangShan's simulator binary in this phase, as its strong data sharing patterns are tightly aligned with the motivation for LLC-aware scheduling. Additionally, we evaluate the VideoTranscodeBench and Mediawiki benchmarks from DCPERF [17], as well as the CactuBSSN and

Workload	Data sharing	Cache bound	Thread Lifetime
ChaCha20	High	High	Long
MediaWiki	Low	High	Long
Video SVT	Medium	High	Short
cactuBSSN	High	High	Long
perlbench	Low	Low	Long

TABLE II
WORKLOAD TAXONOMY: POST-MIDTERM SCHEDULER EVALUATION

Benchmark	EEVDF 2S	EEVDF 1S	SCX_LAVD 2S	SCX_LAVD 1S
video	3:17	2:28	3:25	2:28
mediawiki	16:24	15:39	16:18	15:38
cactus	4:30	2:29	4:37	2:32
perlbench	17:52	05:22	17:27	5:24
xiangshan	1:05	0:47	1:05	0:47

TABLE III
RUNTIMES ACROSS TESTED BENCHMARK CONFIGURATIONS

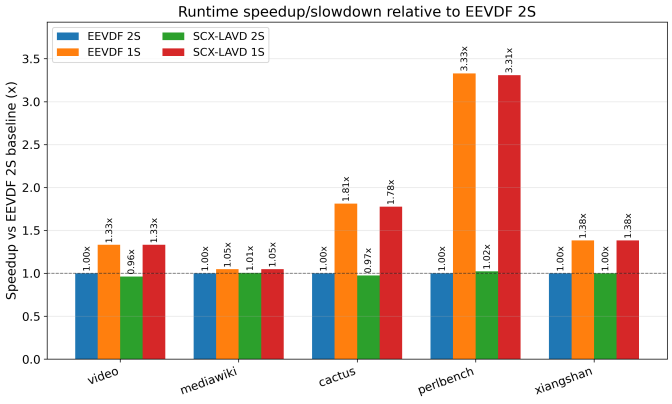

Fig. 5. Normalized Speedup of Benchmark Runtime

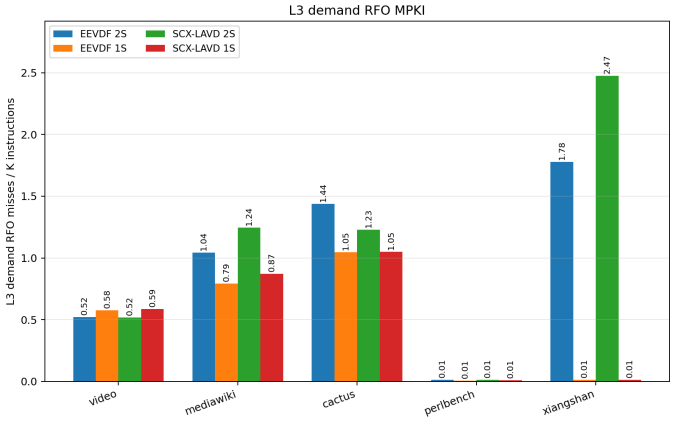

Fig. 6. L3 Cache Read-for-Ownership Misses

Perlbench benchmarks from SPEC2017 [18]. This selection gives us an array of benchmarks with varying cache reuse rates, data sharing, all of which have fairly realistic computational characteristics. We have compiled our understood profiles of these workloads in Table II.

In line with our evaluation motivation from Section 3, we compile and run each benchmark to utilize no more than 32 active threads.

1) Latency, Throughput, and Performance: As mentioned earlier, we restrict our future testing to single-socket and dual-socket (1S/2S) tests of EEVDF and SCX_LAVD with SMT turned off. As seen in Table III and Figure 5, we see dramatic improvements on most benchmarks by disabling the second socket. Despite LAVD's goal of being designed for latency-optimal scheduling, we found that even in undersubscribed workloads its runtime was almost identical to that of the default Linux scheduler.

Looking at L3 RFO misses again in Figure 6, this time normalized to per-kilo-instruction, we see a fairly marked increase going from 1 socket to 2 socket in our high-data-sharing workloads (Cactus, Video, Xiangshan). Especially in the case of Xiangshan, this difference is so dramatic that it can single-handedly account for the speedup we see. This is despite, as seen in Figures 7, L2 access frequency and miss frequency being fairly invariant of scheduler or socket configuration. This comes with the sole exception of Perlbench, which also happens to be the only benchmark with near-zero RFO MPKI in Figure 6.

Rerunning the benchmark suite with the L3 counters mentioned earlier, we find that Perlbench also happens to be the only benchmark where the workload comfortably fits within the L3 cache, per PMU counters shown in figures 9 and 10.

By investigating scheduling records, we found that even on the 32-thread configuration of SPEC2017's INTSPEED suite, Perlbench still maintains only a single active thread. Though it's not clear to us *why* this is the case, the L1I miss rate of Perlbench inflates significantly when run with both sockets enabled, as shown in Figure 11. This metric should be invariant of memory residency (we would expect L1I stall cycles to go up, but the miss count to be identical), nor do we see any ostensible cross-socket migration of the benchmark thread based on the scheduling records we analyzed.

Why Perlbench behaves the way it does is an open question for our group, and we hope to further investigate it over the

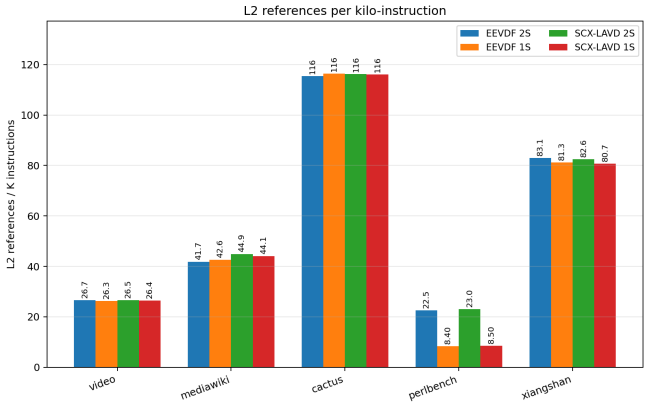

Fig. 7. L2 reference count per core across benchmark configurations

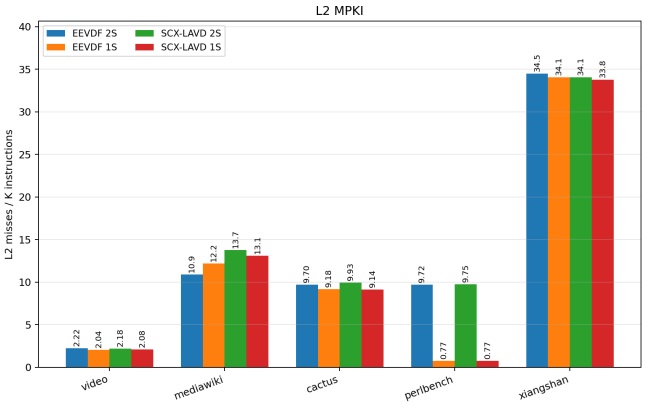

Fig. 8. L2 MPKI across benchmark configurations

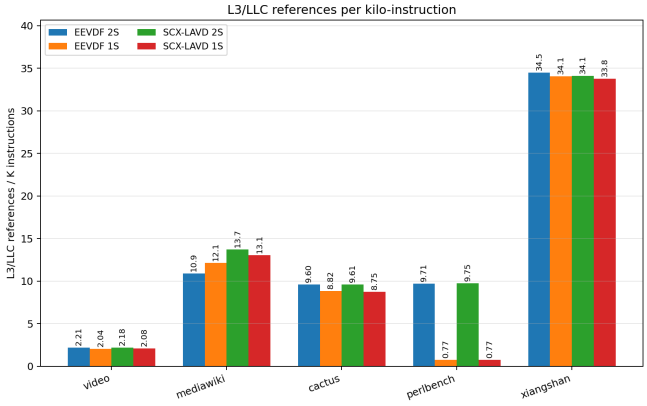

Fig. 9. L3 reference count per core across benchmark configurations

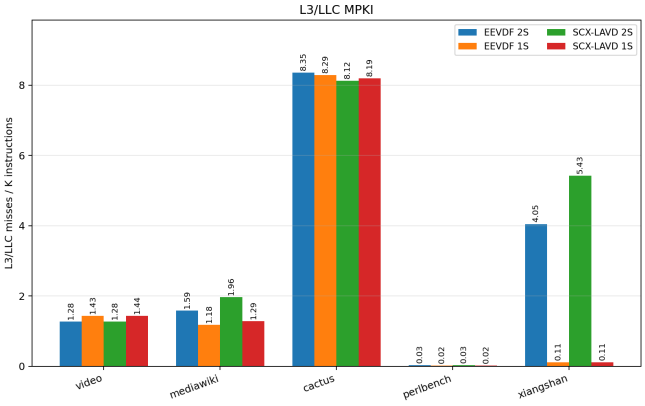

Fig. 10. L3 MPKI across benchmark configurations

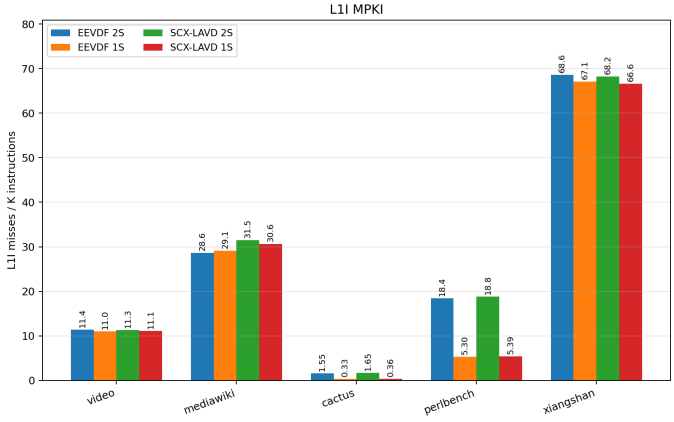

Fig. 11. L1 ICache Misses per Kilo-Instruction

summer, as the underlying cause seems to have massive performance implications, even in single-threaded workloads. We have verified that this behavior persists between different runs and multiple system boots on our sapphire rapids machine, and intend to test it on other dual-socket servers as availability allows.

Looking at the other benchmarks that are faster on single-socket execution, particularly CactuBSSN (cactus) and Video, their speedup cannot fully be explained by the reduction in L3 RFO misses. As these benchmarks have larger working sets, their L3 miss counts also largely consist of increased cold misses from fetching new data – this reflects as a minimally variant L3 MPKI, shown in Figure 10.

After our final presentation, we went back and re-gathered data for these benchmarks across a number of additional performance counters to assess potential reasons for improvement. While there is no inherent counter to show the amount of latency incurred by L3 misses directly, we found that Cactus and Video in particular encountered a higher rate of remote DRAM requests when executed in dual-socket mode, as seen in Figure 12.

Circling back towards the initial speedup graph, something that has likely stood out by now is the uniformity of results between LAVD and EEVDF. Though, as we will see in the Scheduler Topology Analysis section, these two schedulers display vastly different behavior in the way they move threads and assign their positions, we find that LAVD has nearly identical micro-scale behavior from a performance perspective.

This is most likely due to the fact that LAVD's first-order

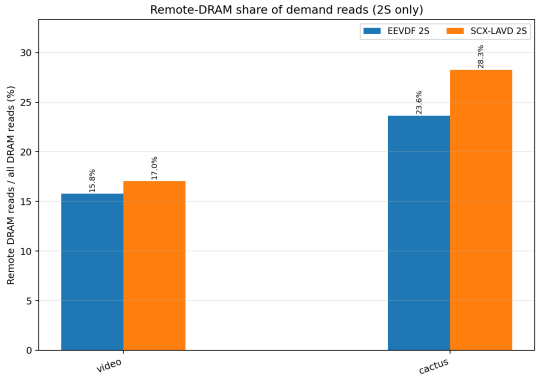

Fig. 12. Percentage of Requests Serviced by Remote DRAM

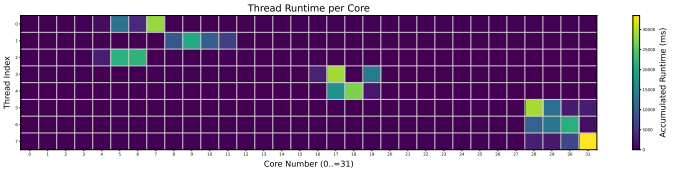

Fig. 13. LAVD scheduler migrations (ChaCha20 program) on one socket.

design goal was to improve program latency on the Steam Deck, a platform with limited compute (a small Zen 2 core). This was primarily done by promoting better L1/L2 cache locality by limiting the maximum average distance a thread can travel when migrating to another core.

By contrast, our testing is on Xeon processors with massive L1/L2/L3 caches. The L1 and L2 are private at 80KiB (I/D aggregate) and 2MiB respectively, while the L3 is a shared 60MiB cache per die. We are significantly less likely to encounter capacity or conflict misses when compared to the Steam Deck. As a result, the throughput benefits of LAVD's strict locality enforcement are largely masked by the sheer capacity of the Xeon's private cache hierarchy.

2) Scheduler Topology Analysis: In addition to PMU data, we also reconstructed core migration records from kernel scheduler events aggregated via `perf sched` and post-processed in Python. This was used to analyze migration patterns and scheduling topology in our benchmark suite.

Through this data, we are able to identify two major themes which are generally applicable across the set of benchmarks:

1) Thread migrations follow consistent patterns.
2) Both scheduling policies encounter similar issues with cross-socket thread distribution.

The latter point is not discussed extensively, as it is a theme found across all data collected, rather than an insight which requires specific case study.

Thread migrations follow consistent patterns. Across all evaluated benchmarks, threads typically migrate only within a small cluster of processors when the LAVD scheduler is used. This corresponds to tasks being repeatedly assigned within a given vLLC, rather than to a specific core. As an example of this behavior in the ChaCha20 benchmark, refer to Figures 3 and 4 for the EEVDF scheduler, and Figures 13 and 14 for the LAVD scheduler. This pattern, where threads are confined to a small group of cores, is also seen when the LAVD scheduler is used with the Perlbench benchmark. The single-socket and dual-socket configurations are shown in Figures 15 and 16, showing a similar pattern to the XiangShan benchmark.

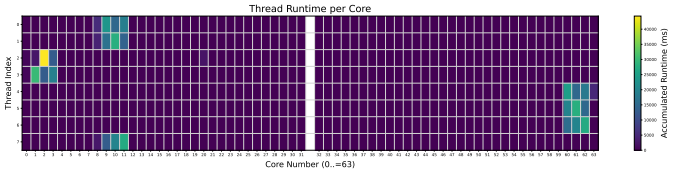

Fig. 14. LAVD scheduler migrations (ChaCha20 program) on two sockets.

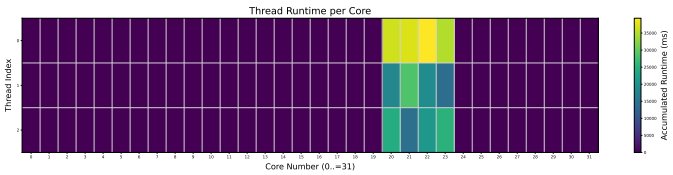

Fig. 15. LAVD scheduler migrations (perlbench) on one socket.

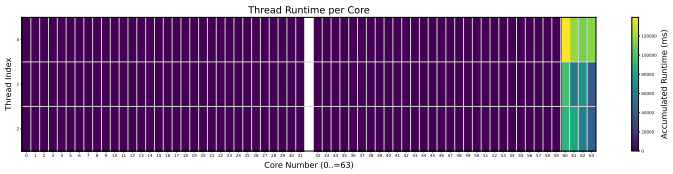

Fig. 16. LAVD scheduler migrations (perlbench) on two sockets.

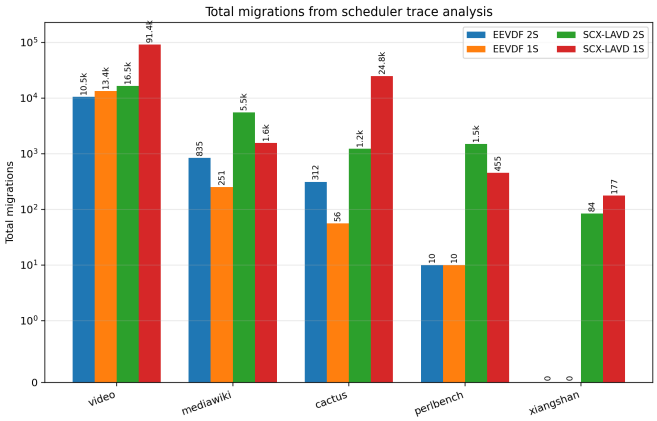

Fig. 17. Total number of scheduler thread migrations.

Interestingly, we find that SCX_LAVD is more aggressive with core migrations than the default scheduler. Figure 17 shows that the number of overall core migrations experienced goes up across the board using the LAVD scheduler.

This leads to a rather unexpected insight. Typically, threads that migrate between cores lose warm cache state. Hence, L1/L2 "losses" would naively be expected to be higher for LAVD.

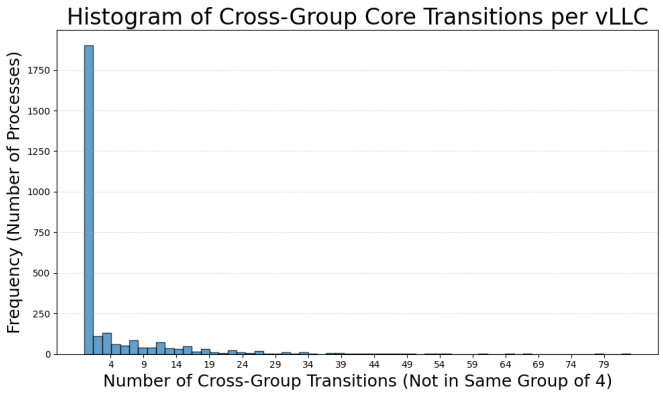

Fig. 18. Histogram of Video migrations, under LAVD scheduler.

Despite this, the overall runtimes, per Figure III, are generally quite similar to their EEVDF counterparts across the benchmark set. This, combined with previous analyses, suggests that LAVD does a comparable job of satisfying the cache needs of benchmarks, despite its tendency to be more "skittish" with thread migrations. This holds true even in cache-bound workloads like Perlbench.

Together, these graphs also provide another "visual suggestion". Namely, the LAVD scheduler leads to a thread's work being performed primarily on a resident vLLC, at least in the situations studied (underutilization). To confirm that the behavior holds in less trivial circumstances with low thread counts, we offer a related visualization for the `video` benchmark, for which we identify 2824 (largely short-lived) threads belonging to the parent process. With this many threads, the previous graphical method is not a viable way to illustrate this pattern.

Empirically, the finding is that cross-vLLC transitions are fairly rare, across the benchmark set. For the `video` benchmark, a long-running benchmark which sees the most times where threads depart from their preferred vLLCs, we see that the vast majority of tasks are occasionally placed in dispatch queues which lead to their execution by processors not in the same virtual LLC domain. However, the overall distribution shows that these events, while non-zero, are still relatively rare. The distribution of how many cross-vLLC transitions a thread appears for each of these threads is provided in Figure 18.

All in all, this illustrates that the LAVD scheduler is able to deliver on an important invariant prioritized by the Cache Aware Scheduling patch: namely, threads which are contained in preferred vLLCs are not frequently evicted from them. Granted, there is no analogous phenomenon to the new load balancer behavior type which Cache Aware Scheduling plans to add (namely, active correction to place threads back in their preferred LLCs).

V. Conclusion

A. Main Findings

Our main findings are as follows. First, cross-socket placement is the dominant source of performance loss for data-sharing workloads in our benchmark suite. As a result, frequent migration is not necessary for scheduler-induced performance degradation; even stable placements with no migration will perform poorly if communicating threads are spread across sockets.

Second, while LAVD changes migration and which processors threads reside in, its runtime is nearly identical to EEVDF across the tested workloads. Finally, L1 and L2 cache metrics are not consistently predictive of runtime, while L3 RFO behavior better captures the cost of poor placement for shared-data workloads.

B. Limitations

This study has several limitations. Our PMU measurements are collected at the system level rather than being perfectly attributed to the benchmark process. We minimize background activity and compare configurations under consistent conditions, but the counters may still include some unrelated system noise.

Second, the number of hardware performance counters available on the system is limited. Because our counter set is larger than the number of events that can be measured simultaneously, `perf` multiplexes events across time and interpolates counts for periods in which an event is not actively being sampled. This means that some PMU values are estimates rather than direct measurements. We therefore treat the PMU data primarily as relative evidence across controlled configurations, rather than as exact absolute counts.

We also evaluate SCX_LAVD as a practical proxy for cache-aware scheduling behavior, but we do not directly evaluate Intel's Cache-Aware Scheduling patch because it requires a custom kernel configuration that was not available in our testing environment. Our benchmark suite is intentionally undersubscribed and may not capture behavior under heavy oversubscription, where fairness and load-balancing pressure would be higher.

C. Future Work

Future work should evaluate these schedulers under a wider range of systems, workloads, and thread-count regimes. In particular, the unexplained Perlbench behavior warrants further investigation, since it shows a large single-threaded runtime difference between single-socket and dual-socket configurations without an obvious migration explanation. Additional experiments on other dual-socket machines would help determine whether this effect is specific to our Sapphire Rapids platform or reflects a broader scheduler or memory-placement interaction.

Future work should also collect per-process or cgroup-filtered performance counters where possible, retain repeated trials for statistical analysis, and directly test the Cache-Aware Scheduling patch. More generally, future scheduler designs should prioritize mechanisms that keep strongly communicating threads within the same LLC or socket while still preserving enough work conservation for oversubscribed workloads. It is our understanding that eBPF has added support for PMU sampling – this functionality could be used to instrument

much more accurate per-thread profiling, potentially baked into LAVD itself.

Finally, we remark that both schedulers exhibit the negative behavior that we set out to study (steady-state placement of sharing threads on different sockets), even though threads are encouraged to cluster in LAVD. As a result, we highly recommend future work to focus on restructuring scheduler topology to correct for inter-socket sharing at runtime. It remains to be seen whether a greedy solution (like the one proposed in Intel CAS) or reactive approaches will perform best.

REFERENCES

[1] T. C. Chen. (2026) [patch v4 00/22] cache aware scheduling. Linux Kernel Mailing List. [Online]. Available: https://lore.kernel.org/lkml/cover.1775065312.git.tim.c.chen@linux.intel.com/

[2] C. Min. (2026) scx_lavd. Intial release 2024. [Online]. Available: https://crates.io/crates/scx_lavd

[3] D. Dai and R. Newton, "LAVD: Meta's New Default Scheduler," Presented at the 2025 Linux Plumbers Conference, Tokyo, Japan, Dec 2025.

[4] The kernel development community, "CFS Scheduler — The Linux Kernel documentation," https://docs.kernel.org/scheduler/sched-design-CFS.html.

[5] "sched(7) — Linux manual page," https://man7.org/linux/man-pages/man7/sched.7.html.

[6] Peter Zijlstra, "Re: [PATCH 00/10] sched: EEVDF using latency-nice," https://lore.kernel.org/lkml/20230307130800.GD2017917@hirez.programming.kicks-ass.net/, Mar 2023.

[7] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The Linux scheduler," in *EuroSys '16: Eleventh EuroSys Conference 2016*, 2016.

[8] The kernel development community, "EEVDF Scheduler — The Linux Kernel documentation," https://docs.kernel.org/scheduler/sched-eevdf.html.

[9] P. Zijlstra. (2023, Mar) [patch 00/10] sched: Eevdf using latency-nice. Linux Kernel Mailing List. [Online]. Available: https://lwn.net/ml/linux-kernel/20230306132521.968182689@infradead.org/

[10] Linux Kernel Developers, "Linux Completely Fair Scheduler source code comments on task migration exclusions," 2026, comments in `kernel/sched/fair.c`, commit `e1914add2799225a87502051415fc5c32aeb02ae`. [Online]. Available: https://github.com/torvalds/linux/blob/e1914add2799225a87502051415fc5c32aeb02ae/kernel/sched/fair.c#L9748-L9756

[11] T. C. Chen, "[Patch v4 14/22] sched/cache: Handle moving single tasks to/from their preferred LLC," Linux Kernel Mailing List, 2026, lKML patch/email message. [Online]. Available: https://lore.kernel.org/lkml/9b816d8c27fabf2a9c0e1f61a6b90afe8ec4ad52.1775065312.git.tim.c.chen@linux.intel.com/

[12] ——, "[Patch v4 00/22] Cache aware scheduling," Linux Kernel Mailing List, 2026, lKML patch series cover letter. [Online]. Available: https://lore.kernel.org/lkml/cover.1775065312.git.tim.c.chen@linux.intel.com/

[13] C. Min, "Using sched_ext to improve frame rates on the SteamDeck: Ideas behind the LAVD scheduler," Linux Plumbers Conference 2024 presentation slides, 2024, presented at Linux Plumbers Conference 2024. [Online]. Available: https://lpc.events/event/18/contributions/1713/attachments/1425/3058/scx_lavd-lpc-mc-24.pdf

[14] D. Dai and R. Newton, "LAVD: Meta's New Default Scheduler," Linux Plumbers Conference 2025 presentation slides, 2025, presented at Linux Plumbers Conference 2025. [Online]. Available: https://lpc.events/event/19/contributions/2099/attachments/1875/4020/lpc-2025-lavd-meta.pdf

[15] David Dai , "scx_util, scx_lavd: Add virtual LLC support," GitHub pull request, sched-ext/scx #2705, 2025, merged on September 8, 2025. [Online]. Available: https://github.com/sched-ext/scx/pull/2705

[16] David Dai, "scx_lavd: Introduce per cpu dsqs," GitHub pull request, sched-ext/scx #2563, 2025, merged on August 12, 2025. [Online]. Available: https://github.com/sched-ext/scx/pull/2563

[17] W. Su *et al.*, "DCPerf: An Open-Source, Battle-Tested Performance Benchmark Suite for Datacenter Workloads," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 1717–1730. [Online]. Available: https://doi.org/10.1145/3695053.3731411

[18] J. Bucek *et al.*, "SPEC CPU2017: Next-Generation Compute Benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771