Appears in: Proceedings of the 23rd International Symposium on Real-Time Distributed Computing (ISORC 2020)
virtual, May 2020

Precisely Timed Task Execution

Stefan Reif and Wolfgang Schröder-Preikschat
Friedrich-Alexander University Erlangen-Nürnberg
{reif,wosch}@cs.fau.de

***Abstract*—The Internet of Things (IoT) requires that all components operate on "fresh" data. However, ensuring a low Age of Information (AoI) is challenging, in particular when communicating across the Internet. Achieving a low AoI requires cooperation of networking strategies as well as node-local strategies so that the entire system performs the right operation at the right moment in time. However, most modern operating systems are not designed to provide accurate timing—various low-level overheads cause delays and jitter that affect communication significantly.**

This paper presents CLOCKFIX, a run-time system for the Linux user-space that executes jobs accurately and precisely at their intended execution times. The evaluation shows that CLOCKFIX reduces undesired delays by more than 95 %, and it improves the AoI in a real-world network protocol by over 16 %.

***Index Terms*—Latency, Jitter, OS Noise, Predictability, Interference, Timed Task Execution**

I. INTRODUCTION

The Internet of Things (IoT) connects billions of devices with each other, enabling communication and cooperation [1]. Applications using internet-connected devices require that all components operate on recent data. Achieving a low Age of Information (AoI) is challenging, as it requires cooperation of network-related schedules (i.e., when to transmit a data packet) and node-local schedules (i.e., when to execute specific functions) [2]–[4]. Implementing real-time network communication hence requires the execution of specific functions with accurate and precise timing. To minimise the AoI, the local schedules adapt to channel properties, particularly in highly dynamic networks such as the Internet or wireless connections [5]. In such systems, the network protocol detects the optimal speed of operation, and both sender and receiver have to adapt their timing to ensure smooth operation. A key challenge is "buffer bloat" [6]—storing data packets temporarily in application-level, system-level or network-level buffers. Buffering causes significant delays and jitter for the stored packet itself, and subsequent packets also have to wait until they can be processed further. Providing a low and predictable AoI therefore requires traffic shaping, and also that packets are processed at the right moments in time [7]. This paper focusses on node-local timing aspects.

Traditionally, operating systems offer multiple approaches for timed task execution. First, hardware-timers ("time-triggered execution") are intended to fire at specific moments in time. However, its timer-interrupt handling mechanism is affected by delays due to the *hardware* (e.g., signal propagation delays and latencies of programmable interrupt controllers), *software* (e.g., interrupt handling overhead, context switches) and also *interference* (e.g., synchronisation with locks and interrupt suppression or scheduling). Even though these delays are relatively small, they harm the performance significantly [8]. Second, interrupt-driven task execution ("event-triggered execution") waits for a hardware signal (e.g., packet arrival). This mechanism suffers from the same latency issues as time-triggered task execution. Furthermore, event-driven task execution does not react on the *absence* of events (e.g., packet loss) and it is not suited for *proactive* measures (e.g., pre-allocation of buffers). Third, polling can eagerly check hardware clocks or event sources for updates, avoiding the interrupt handling mechanism and its overhead. However, it is inefficient with respect to processing time and energy demand. Since many embedded systems are power- or temperature-constrained, minimisation of polling times is desirable.

This paper focusses on soft real-time systems, running Linux or similar operating systems [9], which is typical for the Internet of Things and Edge [10], [11] computers. In such systems, node-local deadlines are considered *soft* because the timing behaviour of Internet-traversing communication is erratic anyway, and applications have to tolerate situations where packets do not arrive in time.

We argue that being *aware* of latencies is a prerequisite for solutions that reduce them. Our key observation is that every event happens later than intended (in terms of physical time), due to various delays. We integrate this observation in CLOCKFIX, a run-time system for precisely timed task execution, running in the Linux user-space. On the one hand, we recognise that hardware- and software-related delays are relatively constant, which means they are *predictable*. We use on-line machine learning to anticipate these unintended delays. We can thus purposefully reduce sleep times, correcting the moment of wake-up. On the other hand, interference-related delays tend to be chaotic and unpredictable. We therefore handle such delays by a specific system architecture.

The contribution of this paper is as follows. First, we present CLOCKFIX, a run-time system for precisely and accurately timed task execution. It combines two approaches—a special-purpose concurrent data structure that manages jobs without blocking or suppressing synchronisation, and a low-overhead on-line machine-learning approach ensures that the task execution starts as accurately as possible at the specified execution time. Second, this paper presents an empirical evaluation using micro-benchmarks that demonstrate a reduction of unwanted latencies by more than 95 %, as well as an evaluation of a state-of-the-art real-time transport protocol demonstrating a 16 % improvement of the age of information.

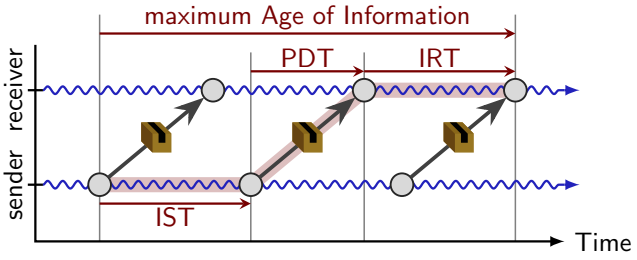

Fig. 1. The maximum age of information is the sum of the Inter-Send Time (IST), the Packet Delivery Time (PDT), and the Inter-Receive-Time (IRT).

The rest of the paper is structured as follows. Section II briefly presents the necessary background information. The architecture and the precise timing mechanism of ClockFix are presented in Sections III and IV, respectively. Sections V and VI empirically evaluate ClockFix with micro-benchmarks as well as a network protocol. Section VII summarises related work and Section VIII concludes the paper.

II. Background

In order to keep the age of information low, both network-related timings as well as node-local timings have to be considered [2]. For dynamic channels, such as the Internet or wireless networks, both the sender and the receiver have to adapt their timings to the current conditions [12]. In communicating systems, the *minimum* age of information on the receiver side is the Packet Delivery Time (PDT). It is effective if the sender produces a data item exactly ahead of a packet transmission, and the receiver processes this information immediately. However, achieving this minimum AoI is not always feasible. For example, receivers can gather information from multiple connections, but the packets typically do not arrive all at exactly the same moment. In consequence, the receiver has to store values temporarily and use them later. Similarly, storing data temporarily can also be required in the sender node, for example, when a sensor operates with a fixed sampling rate. This paper therefore considers the *maximum* AoI. As Figure 1 shows, the maximum AoI is the sum of the Inter-Send Time (IST), the PDT, and the Inter-Receive Time (IRT). The AoI reaches its maximum if an information item is created right *after* a packet transmission, and the receiver keeps the information until the subsequent packet replaces it. Since both node-local as well as network-related timings contribute to the AoI, its minimisation requires cooperation of node-local and network-level schedules.

Systems that communicate through the Internet with predictably low age of information have been enabled by BBR ("Bottleneck Bandwidth and Round-trip propagation time") [5]. This latency-avoiding [13] congestion control algorithm achieves high throughput while avoiding buffer bloat [6]. BBR oscillates between bandwidth-sensing and round-trip-time-sensing phases, to derive the optimal amount of data in-flight. This information enables BBR to keep buffers mostly empty, which minimises buffering-related delays and jitter on the network level.

Keeping *local* buffers empty as well requires that the schedules of each local data processing are controlled by precisely timed task execution. For example, if a *consumer* operates too early, the most-recent data item is not yet available. Hence it has to operate on previous data items, prolonging the lifetime of that piece of information unnecessarily. In contrast, if the consumer is late, data is stored in buffers. Even if buffering times are small, the delays accumulate and, in summary, cause significant latency and jitter [8]. For *producer* tasks, timing is equally important—producing early causes buffering, while producing late forces other tasks to operate on out-dated information or to wait. To keep the AoI low, it is therefore mandatory that all data producers and consumers, on both the sender and the receiver node, operate with the correct timing. Ideally, the entire data processing pipeline operates at a steady rate, where each component adapts to the current bottleneck performance. Then, buffers are kept empty, which minimises buffering-related latency and jitter.

However, operating systems typically introduce *OS noise* [14]–[16], which summarises jitter that disturbs the execution of applications. Sources of OS noise are, for example, hardware interruptions or system-level tasks. Even though OS noise has a low probability and typically causes only minor overhead, its influence at application level can be significant.

III. Timer Architecture

The main goal of the ClockFix architecture is to enable timed task execution with minimal *interference*-related delays. To this end, it avoids locks, interrupt suppression, and similar techniques where threads have to wait for others to proceed. Nevertheless, ClockFix takes further measures to isolate job execution from the remaining sources of interference.

a) Delegated Task Execution: The general architecture is sketched in Listing 1. To minimise interferences, ClockFix uses a dedicated worker thread that executes all submitted jobs sequentially. This worker thread ideally runs as *isolated* as possible. Pinning and isolating the worker thread on a core helps to minimise system-related interference. However, the communication with this thread introduces another source of interference. In ClockFix, jobs are therefore managed in a special-purpose data structure, which is designed to minimise communication-related interferences.

b) Job Management: Listing 2 shows the lock-free linked list that stores submitted timer jobs. Importantly, job submission can be executed concurrently to the worker, ensuring a consistent state even if executions overlap. In consequence, the worker thread can always search, find, and execute jobs, and never has to wait for the completion of concurrent operations.

The list is based on three invariants. First, the list is always valid and readable by the worker thread. In consequence, the worker thread is always able to search and execute pending jobs. Second, the jobs in the list are sorted by the intended execution start time. This enables the worker thread to immediately find the next pending job, without list walks. Third, the

Listing 1. The CLOCKFIX timer employs a worker thread for job execution.

```
timer_t *timer_new(int core) {
  timer_t *self = new(timer_t);
  self->list = list_new(core);
  self->worker = thrd_new(core, timer_worker, self);
  return self;
}

void timer_run(timer_t *self, work_t what, date_t when) {
  if (list_add(self->list, what, when))
    thrd_wake(self->worker);
}

void timer_end(timer_t *self) {
  self->dead = true;
  thrd_join(self->worker);

  list_del(self->list);
  del(self);
}

void timer_worker(timer_t *self) {
  list_t *list = self->list;
  while (true) {
    node_t *item = list_get(list);

    // check for termination
    if (item->date == ∞
        && self->dead && item == list_get(list))
      break;

    // mark item as used
    list_use(item);

    // await execution date (simplified)
    thrd_sleep_until(item->date);
    if (item != list_get(list) || item->date > now())
      continue;

    // execute item
    if (!item->done) {
      item->what();
      item->done = true;
    }

    // retire item
    list_out(list, item);
  }
}
```

list is never empty. To avoid an empty list, a dummy element containing a job infinitely far in the future terminates the job list. Thanks to the dummy element, fewer corner cases exist, which simplifies synchronisation. Needing fewer corner cases has also *timing* implications—there are no rarely-executed slow paths in the list procedures that could cause tail latencies.

c) Data Structures: The list maintains three pointers into the sorted job list. First, `head` always points to the oldest job in the list. Second, *work* points to the next job to execute most of the time. However, there are minor windows of inconsistency when the job submission function has modified the linked list but *work* is not yet updated. This inconsistency window only matters if the submitted job has an intended execution time *before* any other pending job. In this case, the job submission function has to set back the *work* pointer to finish the job submission, otherwise, the worker thread would not notice the new work item. Third, *hold* is used by the worker thread to avoid garbage collection of jobs that are still needed to maintain list consistency. This is crucial when a job is submitted with an execution time in the past—CLOCKFIX allows job submission after its intended execution time. It executes such requests as soon as possible.

All list functions assume that only a single thread submits timer jobs. This makes the list effectively a single-writer data structure, which again eliminates corner cases. If applications have to submit timer jobs concurrently, they need further measures, such as locks. However, such additional synchronisation only affects job submission, not the worker thread.

d) Garbage Collection: The worker thread performs no house-keeping of the job list. Instead, it leaves finished jobs in the linked list until any other thread submits further jobs. That thread will then remove finished jobs from the list. Since jobs are sorted by the execution time, the garbage collector can start at the `head` of the list and follow `next` pointers until it reaches a job that has not been executed yet. The usage of garbage collection has two major benefits. First, it eliminates situations where the worker thread modifies `next` pointers, maintaining the single-writer structure. Second, it isolates the worker thread from interferences from memory deallocation. Typical allocators use locks and memory-related system calls internally, which have the potential to stall the worker.

e) Job Submission: To submit jobs, a client thread has to insert its job description into the linked list while maintaining its invariants. The thread iterates over the list, searching for the right spot to add the new job into the list, considering that the list of jobs is sorted by their intended execution times. To insert the job, the submission function first updates the `next` pointers of the current and the previous list element. Then, it updates the `work` pointer of the list conditionally. If `work` points to a job that runs *after* the new one, `work` is updated to enable the worker thread to find the new job. Otherwise, no update is required, as `work` points to a job *ahead* of the new one, and the worker thread will find the new one later.

f) Job Execution: The worker thread finds pending jobs using the `work` pointer. It executes this request at the intended moment, and then marks it as done. Since it is possible that jobs in the list are already executed, the worker thread uses the `hold` pointer to prevent simultaneous deallocation of the currently used job descriptor.

IV. ACCURATE TIMING

The system architecture described in Section III eliminates unpredictable jitter. In addition, CLOCKFIX compensates for predictable delays, to achieve accurate timing. While it conceptually uses machine learning techniques to anticipate delays, relatively simple and light-weight algorithms are sufficient in practice. CLOCKFIX measures the wake-up delays and collects these measurement values. We use the term *oversleep* for the *actual* (i.e., measured) delay between the intended wake-up time and an actual wake-up event. Using the collected measurements, CLOCKFIX applies multiple learning functions that predict the oversleep of future sleeping operations. Then, the sleep time is reduced by the anticipated delay. If this prediction is correct, the resulting oversleep is zero, and the task execution starts exactly at the intended moment in time.

Listing 2. The ordered job list enables finding jobs with little interference.

```
list_t *list_new(int core) {
  // create dummy node
  node_t *node = new(node_t);
  node->what = ∅;      // do nothing ...
  node->when = ∞;      // ... infinitely far in the future
  node->done = false;
  node->next = NULL;

  // create list
  list_t *self = new(list_t);
  self->head = self->work = self->hold = node;

  return self;
}

node_t *list_get(list_t *self) {
  return self->work;
}

void list_use(list_t *self, node_t *node) {
  self->hold = node;
}

bool list_add(list_t *self, task_t what, date_t when) {
  // create new node
  node_t node = new(node_t);
  node->what = what;
  node->when = when;
  node->done = false;

  // clean up old tasks
  node_t *iter = self->head;
  node_t *stop = self->work;
  node_t *hold = self->hold;
  while (iter != stop && iter != hold) {
    node_t *next = iter->next;
    del(iter);
    iter = next;
  }
  self->head = iter;

  // find position in ordered job list
  node_t **addr = &self->head;
  while (true) {
    iter = *addr;
    if (node->date < iter->date)
      break;
    addr = &addr->next;
  }

  // insert job
  node->next = iter;
  *addr = node;

  // fix list if needed
  node_t *work = self->work;
  if (node->date < work->date
      || (addr == &work->next && work->done)) {
    self->work = node;
    return true;
  }
  return false;
}

void list_out(list_t *self, node_t *item) {
  // advance self->work carefully
  node_t *next = item->next;
  CAS(&self->work, item, next);
}

void list_del(list_t *self) {
  node_t *iter = self->head;
  while (iter) {
    node_t *next = iter->next;
    del(iter);
    iter = next;
  }
  del(self);
}
```

a) On-Line Learning: To predict the delays of future sleep operations, ClockFix uses timing measurements from past sleep operations. Conceptually, this technique requires an on-line machine learning approach. In practice, ClockFix utilises a maximum filter with a fixed-size sliding window to predict oversleep times. While this is only a simple estimator, the evaluation in Section V demonstrates that it achieves significant latency reductions. In fact, the maximum filter has two benefits, compared to other typical estimators. First, it tends to over-estimate wake-up delays, causing ClockFix to wake up early. However, early wake-ups are not harmful because they are compensated internally, while late wake-ups cause application-noticeable jitter. Second, execution of this learning algorithm itself has a relatively predictable timing, compared to more elaborate learning methods. In consequence, it does not introduce unnecessary jitter.

In addition to the maximum filter, ClockFix uses a *decay* mechanism. If the worker thread did not sleep between the execution of two jobs, it reduces all observed oversleep values in the filter window by 25 %. The decay is required to retain *curiosity* for the learning algorithm. Without decay, it would be possible that (due to, for example, an extraordinarily large scheduling latency caused by another process), a single sleep function wakes up extremely late. The consequence would be that a wake-up delay outlier is observed, and the maximum filter stores this value internally. For all subsequent sleeping calls, ClockFix would reduce the sleep time by this outlier value, and then decide to never again go to sleep, as the remaining sleep duration is negative. However, if the system performs no further sleep calls, the outlier would never be evicted from the filter window. Thanks to decay, outliers diminish over time, until the oversleep estimation is low enough to re-allow sleeping. If, however, ClockFix does sleep, a new oversleep measurement value is added to the filter. In consequence, decay is not needed in these situations.

b) Second-Level Learning: Even though the on-line learning function itself is relatively simple, it adds further latency and jitter to ClockFix. Therefore, ClockFix applies a second-level learning function to monitor the execution time of the actual oversleep learning function, and to predict its execution time. However, this causes a chicken-and-egg problem—the second-level learning function also has a non-zero execution time. In theory, this calls for an infinite hierarchy of learning functions—the latency and jitter of the n-th level learning function requires a $(n+1)$-th level learning function that predicts its execution time. In practice, however, ClockFix uses an appropriate second-level learning function with low latency and jitter, so that no further learning function hierarchy is required—an Exponential Moving Average (EMA) function predicts the execution time of the learning function. This function can be implemented with near-constant execution time thanks to a branch-free control flow.

The expected costs of learning are treated like other delays—the sleep time is reduced accordingly. In consequence, ClockFix can execute the learning function between the wake-up time and the job execution time.

c) Deferred Learning: The two-level learning approach enables ClockFix to compensate for most oversleep situations, since the learning functions tends to over-estimate wake-up delays. In addition, deferred learning tackles situations of unusually large oversleep values.

Since the sleep time is reduced by the expected learning function execution time ($\Delta T_{needed}^{(EMA)}$), the learning function can typically be executed between wake-up and job execution. However, if the oversleep happens to be unexpectedly large, not enough time is available (ΔT_{avail}) to execute the learning function before the job is supposed to run (at T_{exec}). In this case, ClockFix defers the execution of the learning function until the job has completed. Since the second-level learning function computes the *average* execution time of the learning function, a safety margin is added—the learning operation is deferred if $\Delta T_{avail} := T_{exec} - T_{now} \overset{?}{<} 2 \times \Delta T_{needed}^{(EMA)}$.

d) Clock Polling: If ClockFix over-estimates the wake-up latency, the worker thread wakes up too early. In the remaining time, it can execute the learning functions, but nevertheless, it is possible that task execution time is not yet due. In this case, ClockFix polls the clock until the intended task execution time is reached. In consequence, the behavior of ClockFix is aligned with typical delay functions that forbid premature wake-ups. Indeed, this behavior is required for many applications. In real-time network protocols, for example, premature task execution can cause problems if data items are not yet available, causing the task to operate on the wrong data set.

In summary, ClockFix uses a *hybrid* approach for precise and accurate timing. It combines time-triggered execution that enables power-efficient sleep states to the extent possible. In addition, it utilises clock polling when needed, to improve the timing precision. The estimated oversleep constitutes an upper bound for the polling duration.

V. Micro-Benchmark Evaluation

Our evaluation uses three different hardware platforms. First, an Intel Xeon E3-1275 v5 Processor ("Xeon") with 4 cores plus hyper-threading, running at 3.6 GHz fixed, with 16 GiB RAM is the most powerful evaluation platform. It is representative for *edge*-located systems that have relatively powerful hardware and constitute one end-point of the real-time network protocol for the communication with embedded nodes. It runs Ubuntu 16.04 LTS with the official 4.4.0-146-lowlatency kernel. Measurements with the standard non-lowlatency kernel show slightly higher latencies, but the key observations are similar. Second, the Raspberry Pi 3B+ is a off-the-shelf single-board computer. We run Raspbian 10 with a Linux 4.19.66-v7+ kernel. The processor frequency is fixed at 1.4 GHz. Third, the Odroid XU4 is a single-board computer with higher performance than the Raspberry Pi. It runs Ubuntu 18.04-1 LTS with Linux 4.14.85-152 kernel. For our evaluation, we disable the "little" cores and only use the "big" ones to obtain repeatable results. The processor frequency is set to 2.0 GHz.

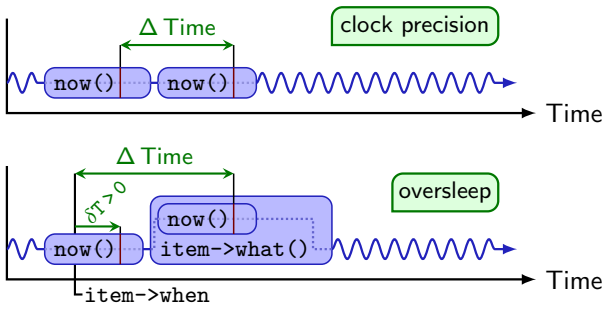

Fig. 2. Time uncertainty (Δ Time) measurement routines to compare the oversleep time with the clock precision.

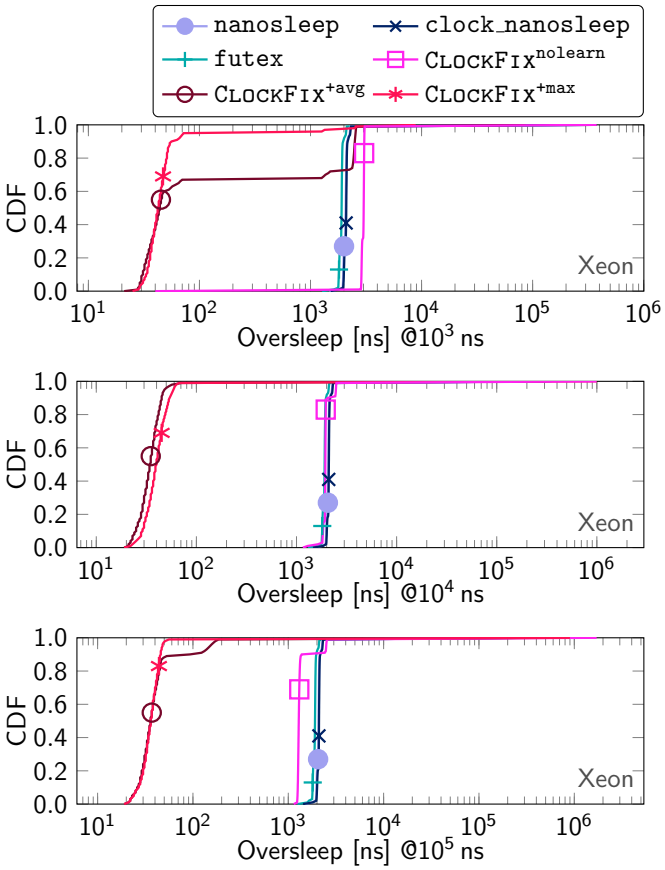

Fig. 3. ClockFix reduces the oversleep significantly, compared to standard library functions and system calls. The plots show intended sleeping durations between 10^3 ns and 10^5 ns. For comparison, a `nolearn` ClockFix variant shows the effect of the timer architecture with oversleep prediction disabled.

Linux initialises the *timer slack* to 50 μs by default to group timer expiration events, for efficiency reasons. Internally, this default configuration implicitly increases all sleeping times. Grouping multiple timer expirations in order to handle them together reduces overhead, such as the number of context switches, but it increases response times for individual timer expirations. We therefore set this configuration to its minimum value, 1 ns, in all experiments using `prctl(PR_SET_TIMERSLACK)` [17].

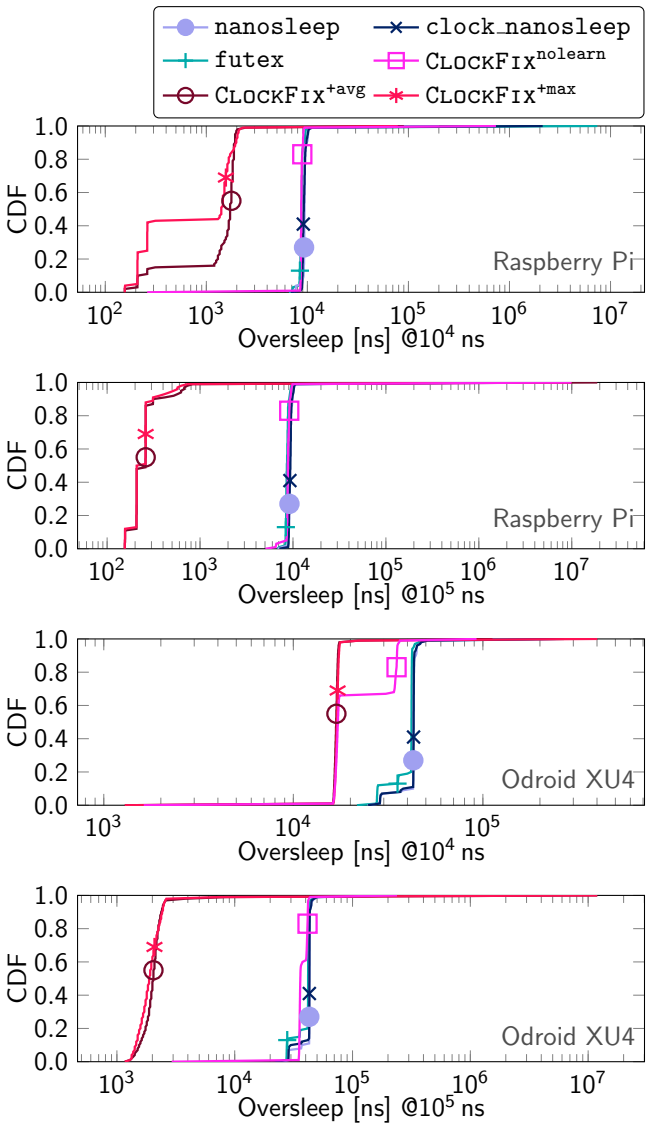

Fig. 4. Standard library functions and system calls can double the waiting time on a Raspberry Pi 3B. ClockFix improves the oversleep time significantly. On the Odroid XU4, all delays are higher.

A. *Oversleep Evaluation*

We compare the application-observed oversleep of ClockFix with standard approaches for timed task execution. We include the raw `futex` Linux system call and the `nanosleep` and `clock_nanosleep` library functions for comparison. These functions take sleep-time parameters that support nanosecond precision. The evaluation further includes a ClockFix variation with a sliding window *average* filter, instead of the maximum filter, as primary learning function. The motivation of the average filter is that it makes "reasonable" oversleep estimations, but it is inherently prone to underestimations, which could cause ClockFix to miss the target job execution time. Furthermore, a ClockFix version without any learning function at all is included to evaluate the overhead of the dedicated worker thread and its communication.

The oversleep measurement routine is depicted in the bottom half of Figure 2. For the oversleep evaluation, we use a special-purpose micro-benchmark that submits a series of timer jobs. First, ClockFix waits until a job is intended to run. Then, as soon as a jobs start executing, it reads the clock and compares it to its intended execution time. For each evaluation scenario, the micro-benchmark first submits 100 jobs where it ignores the results for warm-up, and then it executes 10^6 measurement iterations that are further analysed. We do not perform outlier elimination for the micro-benchmarks. In consequence, a small number of packets ($\ll 1\,\%$) is still affected by system noise in all evaluation scenarios, and shows extremely large values.

Figure 3 summarises the application-level oversleep time on the Xeon machine, as Cumulative Density Functions (CDFs), for sleep durations between 10^3 ns and 10^5 ns. The evaluation shows that all standard library functions perform similarly. Besides, for the ClockFix version without learning, the oversleep time is similar to the standard library functions, indicating that the architecture with a dedicated worker thread causes only a minor overhead, if any. The learning variants, in contrast, show significantly lower delays. For example, the median oversleep time decreases by 98.2 %, compared to `nanosleep`, for 10^5 ns waiting time. ClockFix also reduces jitter—the 1st and the 99th percentile differ by 35 ns for ClockFix$^{+\texttt{max}}$, and by 527 ns for `nanosleep`.

Comparing the learning functions, the average filter is more susceptible to tail latencies than the maximum filter. For very short sleeping times, the oversleep duration is sometimes even as large as it is for the standard library functions. In contrast, the maximum filter reduces the probability of latency tails, compared to the average filter. In many cases, however, both learning functions perform similar. This result indicates that more sophisticated learning functions would not lead to more accurate task execution times.

Figure 4 summarises the oversleep evaluation on the Raspberry Pi and the Odroid. The results are higher than on the Xeon, since the processor cores run at a lower frequency. Again, the delays with ClockFix are significantly lower than the standard library functions—the median latency is reduced by 97.7 %, compared to `nanosleep`. The difference between the 1st and the 99th percentile is 3124 ns for `nanosleep`, but only 728 ns for ClockFix$^{+\texttt{max}}$. The Odroid platform has higher oversleep times than the Raspberry Pi, even though it has a higher clock frequency. Further measurements that are discussed below show that the reason for the large delay is that, on this platform, the clock precision is relatively coarse, which limits the precision of polling. Again, ClockFix reduces the median oversleep time by 95.6 % at 10^5 ns waiting time.

In summary, ClockFix$^{+\texttt{max}}$ improves oversleep times significantly. Similar improvements (i.e., a $> 95\,\%$ reduction of latencies) can be observed on all three evaluation platforms. Thereby, the main contributor to latency reduction is the learning approach of ClockFix. It effectively anticipates delays and thus reduces oversleep durations.

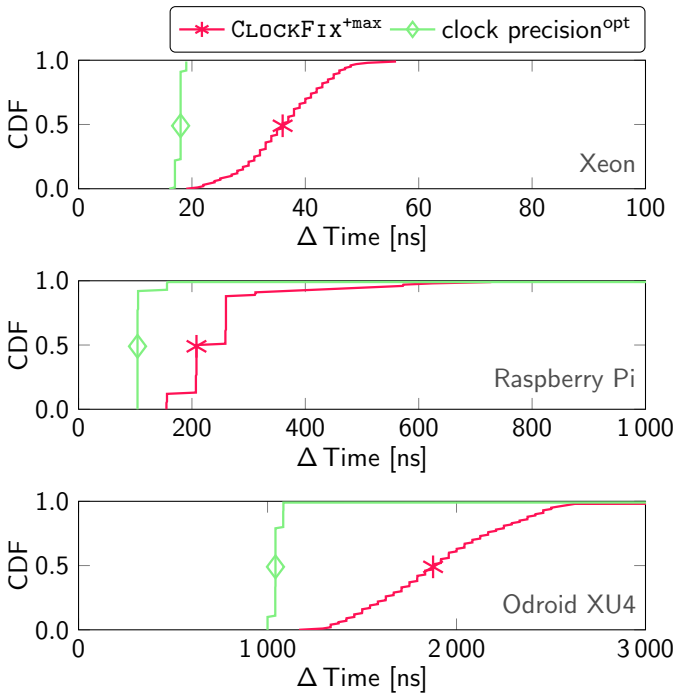

Fig. 5. The remaining oversleep time of ClockFix is close to the clock precision on all evaluation platforms.

B. Time Uncertainty Evaluation

Since the timing precision of ClockFix depends on clock polling, we evaluate the clocks of all three hardware platforms. We measure the clock precision by reading the clock in a tight loop. The timing difference between two consecutive calls is, approximately, the execution time of the function that reads the clock. Even though the Linux system calls return values with nanoseconds precision, the execution time of the corresponding system call represents the timing uncertainty because it is not clear at which moment in time the physical hardware clock is accessed. The measurement method with a tight loop is *optimistic* because it warms up caches. Non-repeated clock-read operations can therefore need more time.

Figure 2 explains the comparison between the timing uncertainty of the clock-read operation and ClockFix. Thereby, ClockFix internally polls the clock until the current time is after the desired job execution time ($\delta T > 0$). Afterwards, it executes the submitted job. To measure the oversleep times, we use a micro-benchmark which, as soon as its execution starts, compares the (then slightly increased) time with its desired execution time. Since $\delta T > 0$, the measured oversleep is expected to be above the clock precision. Indeed, the optimistic clock precision is a *lower bound* for the task execution delay. Thus, the time uncertainty describes the potential behaviour of an optimal learning algorithm that perfectly predicts oversleep times, but would still be limited by the clock precision.

Figure 5 shows that the oversleep time of ClockFix is close to the optimistic clock precision. On all architectures, the time uncertainty of ClockFix is below 3× the time uncertainty of reading the clock, except for a small latency tail on the Raspberry Pi. On the Odroid platform, the clock is particularly imprecise—the median difference between two consecutive clock-read operations is 1042 ns. This result explains why the oversleep is higher on this platform, compared to the Raspberry Pi, despite the higher processor frequency.

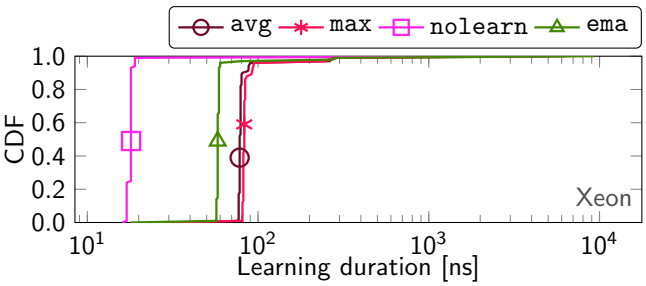

Fig. 6. All learning algorithms need similar amounts of time, with `EMA` being slightly faster. The `nolearn` entry shows the overhead of the time measurement routine, which equals the clock precision.

C. Learning Duration Comparison

Figure 6 shows the execution time of learning functions, on the Xeon platform. Both the average and the maximum window filters have a similar execution time (approx. 80 ns) with little jitter. The EMA function is slightly faster, and it also shows only minimal jitter. As expected, a baseline measurement of not executing any learning function equals the clock precision (i.e., 18 ns). These results strengthen our point that the relatively simple learning algorithms in ClockFix cause little noise.

VI. Real-Time Network Protocol Evaluation

We also evaluate ClockFix with PRRT, a state-of-the-art real-time network protocol tailored for IoT systems [12]. PRRT with ClockFix is available online [18] under an open-source license. This protocol provides a partially reliable low-latency packet stream with configurable packet deadlines. It further combines a BBR variant with node-local timing measurements to detect the bottleneck component in the system and to adapt the node-local schedules accordingly.

We connect two Raspberry Pi 3B+, with the same configuration as described in Section V, via Gigabit Ethernet. Figure 7 visualises the experiment setup. Each node supports two protocol versions—one using ClockFix and one using `nanosleep` to adapt the node-local timing[1]. We evaluate each protocol version individually.

Before the experiment, we synchronise the system clocks with a local NTP server, and disable NTP during the experiment execution to avoid cross-traffic and erratic clock jumps. Despite the NTP synchronisation, the clocks diverge slightly but noticeably during the experiment execution, which affects the packet delivery time (PDT) measurements. As shown in Figure 1, the first time-stamp of the PDT is taken on the

[1]Both versions also occasionally use other standard timing-functions internally, such as the `futex` system call, where appropriate.

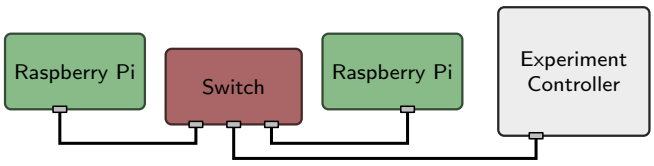

Fig. 7. The real-time network protocol evaluation uses two Raspberry Pi computers that communicate via Ethernet.

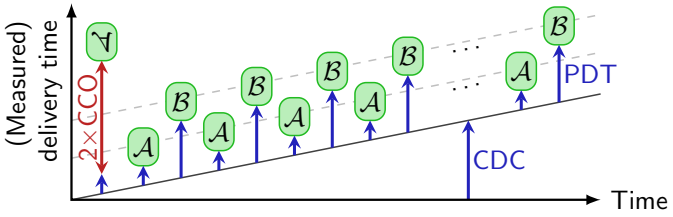

Fig. 8. The Packet Delivery Time (PDT) is adjusted by the Clock Drift Correction (CDC) to compensate for slightly different clock speeds. An inverse iteration reveals the remaining Constant Clock Offse (CCO).

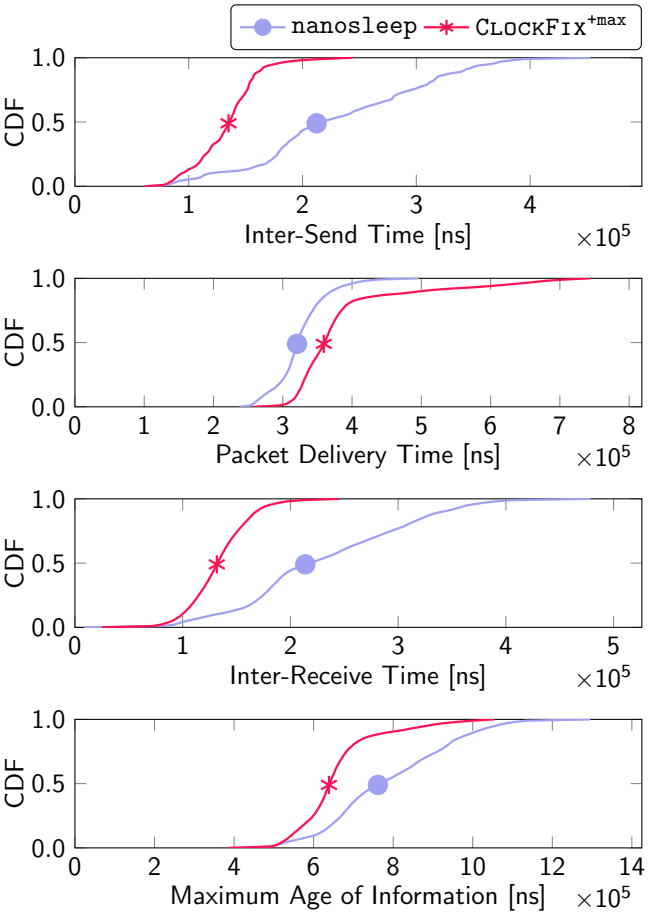

Fig. 9. ClockFix improves the application-level AoI in a real-world real-time transport protocol by reducing the node-local task execution delays.

sender node, and the second time-stamp is taken on the receiver node. The difference between the two time-stamps (i.e., the PDT) is thus distorted by the clock drift. We therefore design the experiment specifically in a way that enables for a compensation of minor clock synchronisation imprecisions.

A. Clock Drift Correction

The packet delivery time (PDT) measurement is affected by clock speed differences—if the clock on the sender side runs slower than its receiver-side counterpart, the measured PDTs increases over time. Vice versa, if the receiver-side clock is faster, the measured delivery times decrease. In consequence, packets at the beginning of the experiment automatically show a different *measured* delivery time, compared to packets at the end of the experiment. Since it is impractical to synchronise clocks during the experiment (it adjusts the local clocks unpredictably), we synchronise the clocks retroactively, by analytical measures.

To compensate for slightly different clock speeds, we *exploit* the fact that the measured delivery times diverge. We execute multiple experiment iterations, and evaluate the two protocol versions alternatingly. If the clocks really diverge, the measured delivery time either increases or decreases as the experiment progresses. This divergence enables an *analytical* Clock Drift Correction (CDC), as depicted in Figure 8. We fit two parallel lines into the two data sets ($\mathcal{A}$, $\mathcal{B}$) of the two protocol versions, using the experiment dates and the measured PDTs, with a least-squares approximation [19]. The (identical) slope of the fitted lines constitutes the clock drift, which we subtract from the delivery time measurement. In consequence, the corrected clocks on the two nodes run at the same speed.

B. Constant Clock Offset Correction

The CDC computes virtual clocks that have the same speed on both the sender and receiver node. However, the two clocks still have an offset which, as both have the same speed, remains constant during the experiment execution. Our method to derive and correct this Constant Clock Offset (CCO) works similarly to NTP. We execute an additional reverse iteration where the sender and receiver node swap roles. The CCO affects the communication in both directions, but in one direction, it increases the measured PDTs, and it decreases the values in the other direction. For the CCO correction, we consider the minimum packet delivery time (i.e., the fastest packet) of each direction and assume that these packets faced no obstructions during delivery, which implies that the corrected delivery time is symmetric. We hence compute the CCO as $\frac{1}{2}\left(min\{\texttt{pdt}^{\downarrow}\} - min\{\texttt{pdt}^{\uparrow}\}\right)$. If, however, the actual delivery time of the two packets was not perfectly symmetric, a small (but still constant) offset remains, and this offset is added to *all* PDT and AoI values. Hence, the clock-corrected results are nevertheless comparable.

C. Experiment Results

We run 10 iterations for each protocol version, plus an inverse iteration, with 1000 packets each. Then we correct the clock drift and offset as described above. Figure 9 visualises the results for each component of the AoI—on the sender node (IST), in the network (PDT), and on the receiver

node (IRT), as described in Figure 1. We remove outliers from the data set because the network protocols can handle unexpected delays—it signals packet loss to the application.

The results show that both the sender-side and receiver-side timings improve, thanks to CLOCKFIX—the durations between packet transmissions and receptions decrease, and also become more predictable since the range of inter-packet time values narrows down. In consequence, the packet rate is much more stable, thanks to CLOCKFIX. However, the packet delivery time increases slightly with CLOCKFIX. This adverse effect is caused by BBR—the bandwidth probing mechanism acts more greedily if the network is faster, causing in-network buffers to be filled temporarily. This is a well-known issue of BBR [20], and a recent topic of communication systems research. In summary, the node-local timing improvements of CLOCKFIX outweigh the delivery time increases. The median AoI decreases by more than 16 %.

VII. RELATED WORK

Several papers have analysed latencies in Linux. Reginer et al. [21] provide a high-level analysis. Other papers identify the timer resolution and non-preemptable sections [22], scheduling latency [23], interrupt bottom handlers [24], and the hardware [25] as latency sources.

Approaches to *reduce* latencies typically remove communication indirections, for example by moving network controllers closer to the processor [26] or kernel bypass networking [27], [28]. Latencies generally become a problem in network-dependent applications because network speed grows [29]–[31] while the single-core processor performance has been stagnating [32]. Thereby, even seemingly small latencies can accumulate [8] and severely harm the system's performance [15], [16], [33]. In consequence, delays that were considered negligible in the past (most prominently, the system-call overhead) are becoming problematic in networked systems. Our evaluation demonstrates that the interrupt handling overhead should be avoided, but can be compensated.

A notable approach to make interrupt latencies in Linux more *predictable* is the Preempt-RT patch set [21], [23], [34]. CLOCKFIX minimises such interferences by thread isolation. Patel et al. propose TimerShield [35] to avoid noise by disabling low-priority timer interrupts while high-priority tasks execute. This approach reduces the amount of possible interruptions, and thus the worst-case execution time, for high-priority tasks. In comparison, CLOCKFIX assumes that the network-defined task execution times do not conflict[2], which implies that no task priorities are needed internally.

Non-deterministic execution times have also become a *consistency* problem [36]. If threads access shared data concurrently, but the execution time is not deterministic, it is unclear which thread observes which version of the shared state. Thus, a system-level non-functional property (i.e., timing) propagates to application-level functional properties (data consistency, age of information). An approach to make data flow deterministic is Logical Execution Time (LET) [37], where threads copy all input data into local buffers ahead of execution, and write back modifications afterwards. Thus, LET relies on precise timing for the copy and write-back operations. However, the write-back operation is scheduled pessimistically, after the worst-case execution time, to make data-flow chains deterministic. In consequence, the pessimistic execution time of the write-back operation causes relatively long data processing durations, which increase the AoI.

[2]If the execution times overlap, CLOCKFIX executes requests sequentially.

VIII. CONCLUSION

This paper has presented CLOCKFIX, a run-time system that executes jobs as precisely and accurately as possible at the intended moment in time. CLOCKFIX combines multiple techniques to eliminate unwanted delays and jitter. It tackles system-related noise by core pinning and isolation, communication-related noise by a dedicated data structure, and the remaining delays by an on-line machine learning approach. Our empirical evaluation results show that CLOCKFIX reduces unwanted latencies by more than 95 %, compared to the state of the art. The remaining delays are close to the clock precision on all evaluated platforms ($< 3\times$). In a real-world real-time network protocol, the timing precision improvements of CLOCKFIX cause the AoI to decrease by over 16 %.

IX. ACKNOWLEDGMENT

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants no. SCHR 603/15-2 ("e.LARN") and SCHR 603/8-2 ("LAOS"). We would like to thank Andreas Schmidt and Pablo Gil Pereira for their insightful feedback.

REFERENCES

[1] "Gartner says 5.8 billion enterprise and automotive IoT endpoints will be in use in 2020," https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io, 2019, acc. 2020-01-23.

[2] S. Kaul, M. Gruteser, V. Rai, and J. Kenney, "Minimizing age of information in vehicular networks," in *Proceedings of the 8th Annual Conference on Sensor, Mesh and Ad Hoc Communications and Networks*. IEEE, 2011, pp. 350–358.

[3] L. Huang and E. Modiano, "Optimizing age-of-information in a multi-class queueing system," in *Proceedings of the 2015 International Symposium on Information Theory (ISIT 2015)*. IEEE, 2015, pp. 1681–1685.

[4] S. Kaul, R. Yates, and M. Gruteser, "Real-time status: How often should one update?" in *Proceedings of the 31st Annual International Conference on Computer Communications (INFOCOM 2012)*. IEEE, 2012, pp. 2731–2735.

[5] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *ACM Queue*, vol. 14, no. 5, pp. 50:20–50:53, Dec. 2016.

[6] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the Internet," *ACM Queue*, vol. 9, no. 11, pp. 40:40–40:54, Nov. 2011.

[7] A. Ousterhout, A. Belay, and I. Zhang, "Just in time delivery: Leveraging operating systems knowledge for better datacenter congestion control," in *Proceedings of the 11th Workshop on Hot Topics in Cloud Computing (HotCloud 2019)*. USENIX, 2019, pp. 1–7.

[8] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[9] "IoT developer survey 2019 results," https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf, 2019, acc. 2020-01-23.

[10] W. Shi and S. Dustdar, "The promise of edge computing," *IEEE Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[11] M. Satyanarayanan, "The emergence of edge computing," *IEEE Computer*, vol. 50, no. 1, pp. 30–39, Jan. 2017.

[12] A. Schmidt, S. Reif, P. Gil Pereira, T. Hönig, T. Herfet, and W. Schröder-Preikschat, "Cross-layer pacing for predictably low latency," in *Proceedings of the 6th International IEEE Workshop on Ultra-Low Latency in Wireless Networks (Infocom ULLWN'19)*. Paris, France: IEEE, Apr. 2019.

[13] D. Zarchy, R. Mittal, M. Schapira, and S. Shenker, "An axiomatic approach to congestion control," in *Proceedings of the 16th Workshop on Hot Topics in Networks (HotNets-XVI)*. ACM, 2017, pp. 115–121.

[14] R. Mraz, "Reducing the variance of point to point transfers in the ibm 9076 parallel computer," in *Proceedings of the 7th Annual International Conference on Supercomputing (SC 1994)*. IEEE, 1994, pp. 620–629.

[15] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *Proceedings of the 8th Annual International Conference on Cluster Computing*, 2006, pp. 1–12.

[16] D. Tsafrir, Y. Etsion, D. Feitelson, and S. Kirkpatrick, "System noise, os clock ticks, and fine-grained parallel applications," in *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, 2005, pp. 303–312.

[17] M. Kerrisk *et al.*, "The Linux *man-pages* project," https://www.kernel.org/doc/man-pages, 2019, version 5.00.

[18] A. Schmidt and contributors, "Predictably reliable real-time transport (prrt)," http://prrt.larn.systems, 2020.

[19] P. Arbenz, "8.4.1 fitting two parallel lines," http://people.inf.ethz.ch/arbenz/MatlabKurs/node86.html, 2008, acc. 2020-01-23.

[20] M. Hock, R. Bless, and M. Zitterbart, "Experimental evaluation of bbr congestion control," in *Proceedings of the 25th International Conference on Network Protocols (ICNP 2017)*. IEEE, 2017, pp. 1–10.

[21] P. Regnier, G. Lima, and L. Barreto, "Evaluation of interrupt handling timeliness in real-time Linux operating systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 52–63, 2008.

[22] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of Linux," in *Proceedings of the 8th Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*. IEEE, 2002, pp. 133–142.

[23] F. Cerqueria and B. Brandenburg, "A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS^{RT}," in *Proceedings of the 9th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'13)*, 2013, pp. 20–30.

[24] M. Wilcox, "I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers," in *Proceedings of the 3rd linux.conf.au Conference*, 2003, pp. 1–6.

[25] M. Samadzadeh and L. Garalnabi, "Hardware/software cost analysis of interrupt processing strategies," *IEEE Micro*, vol. 21, no. 3, pp. 69–76, May 2001.

[26] S. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. Ousterhout, "It's time for low latency," in *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS'11)*. USENIX, 2011, pp. 1–5.

[27] S. Peter, J. Li, I. Zhang, D. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI 2014)*. USENIX, 2014, pp. 1–16.

[28] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI 2014)*. USENIX, 2014, pp. 49–65.

[29] B. H. Leitao, "Tuning 10Gb network cards on Linux," in *Proceedings of the Linux Symposium (OLS'09)*, 2009, pp. 169–184.

[30] J. D'Ambrosia, "100 gigabit ethernet and beyond," *IEEE Communications Magazine*, vol. 48, no. 3, pp. 6–13, 2010.

[31] N. Hanford, V. Ahuja, M. Farrens, D. Ghosal, M. Balman, E. Pouyoul, and B. Tierney, "Improving network performance on multicore systems: Impact of core affinities on high throughput flows," *Future Generation Computer Systems*, vol. 56, pp. 277–283, 2016.

[32] A. Kaufmann, T. Stamler, S. Peter, N. Sharma, A. Krishnamurthy, and T. Anderson, "TAS: TCP acceleration as an OS service," in *Proceedings of the 14th EuroSys Conference (EuroSys 2019)*. ACM, 2019, pp. 24:1–24:16.

[33] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[34] D. B. de Oliveira and R. S. de Oliveira, "Timing analysis of the PREEMPT RT linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 789–819, Apr. 2016.

[35] P. Patel, M. Vanga, and B. Brandenburg, "TimerShield: Protecting high-priority tasks from low-priority timer interference (outstanding paper)," in *Proceedings of the 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS'17)*. IEEE, 2017, pp. 3–12.

[36] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *Journal of Systems Architecture*, vol. 80, pp. 104–113, Oct. 2017.

[37] C. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.