Tintin: PMU Scheduling to Minimize Uncertainty

Marion Sudvarg*†, Ao Li*, Zihan Li*, Sanjoy Baruah*, Chris Gill*, Ning Zhang*
*Department of Computer Science and Engineering, †Department of Physics
Washington University in St. Louis
(msudvarg, ao, tomson.li, baruah, cdgill, zhang.ning)@wustl.edu

***Abstract*—Hardware performance counters (HPCs) on CPUs are essential for tracking program and system behavior by recording microarchitectural events. A major challenge is that, even on modern platforms, the number of available HPCs is limited compared to the events of interest. Existing software tools mitigate this by multiplexing events in a round-robin fashion; however, this approach may introduce significant errors. In a recently-published paper at OSDI, we introduced Tintin, a new HPC profiling infrastructure that mitigates multiplexing errors by characterizing event count uncertainty at runtime and scheduling events on counters to minimize it. This paper provides details on Tintin's elastic model for HPC scheduling that were omitted from the original paper, including a new method for interpolating multiplexed event counts, evaluation of Welford's method for online variance updates, an elastic scheduling policy to minimize predictive uncertainty, and handling of overlapping profiling scopes (e.g., core-level and process-level event monitoring). We also provide a roadmap for using Tintin with the RT-Bench framework for real-time systems benchmarking.**

***Index Terms*—hardware performance counters, measurement uncertainty, elastic scheduling**

I. Introduction

Hardware performance counters (HPCs), which are widely available on modern server, desktop, and embedded CPUs, provide essential tracking of program and system behavior. HPC data are valuable across a wide range of application domains, including debugging [1]–[4], workload optimization [5]–[7], power analysis [8]–[10], diagnostics [7], online resource provisioning [11]–[15], and intrusion detection [16], [17]. Often, HPC data are used to make predictions about program performance to inform decisions. For each use case, it is important to model how target metrics or behaviors of interest depend on hardware event counts.

Brokering HPC Access. HPCs are typically made available to system software via a per-core performance monitoring unit (PMU). The PMU provides an interface to program each HPC to monitor a particular microarchitectural event (e.g., cache loads or misses). Effective PMU usage presents a major challenge even on modern platforms: the number of available HPCs is limited (often 2–6 per PMU) versus how many types of events they can monitor (dozens to several hundred). Towards managing this tension, event profiling tools such as Linux Perf [18] broker access to HPCs. In much the same way that threads and address spaces virtualize the limited physical CPU and memory resources on a system, these tools provide an abstraction layer over the PMU and its limited HPCs.

However, problems arise when using existing event profiling tools to broker HPC access. First, those that mitigate the challenge of limited HPC availability do so via *event multiplexing*, monitoring events in a time-shared round-robin manner, where each event typically receives an equal portion of time. This implies that events will remain unmonitored for some time; observed counts are typically interpolated over these intervals, which may introduce significant errors. Second, event profiling may be requested from multiple *overlapping scopes*. For example, a running process could profile its own hardware events while event counts are simultaneously collected for the processor core it's on. Profiling tools typically treat these scopes independently, which may exacerbate the effects of multiplexing, especially when they share events in common. Moreover, in some cases, scope conflicts can give rise to starvation scenarios where an event is never monitored.

Tintin. In a recent OSDI paper [19], we presented Tintin, a new hardware event profiling infrastructure that addresses the aforementioned challenges while providing flexible specification of event profiling scopes with fine granularity. Of primary relevance to this paper, Tintin characterizes multiplexing errors as uncertainty, which it dynamically tracks during runtime. We demonstrated that the problem of scheduling events on HPCs to minimize overall uncertainty reduces to elastic scheduling, then implemented our algorithm from [20] to allocate HPC time in the Linux kernel. Moreover, since multiplexing-based errors cannot be fully eliminated, Tintin also reports uncertainty via user-space interfaces; applications may use these to enhance their predictive models or rule-based decision logic.

Contributions. A few details were omitted from [19] due to space constraints. First, Tintin implements a custom method for interpolation over multiplexed event counts to address a problem with the original Trapzeoid Area Method proposed in [21]. Tintin's approach is derived and shown to be correct in §III-A. Second, Tintin uses Welford's method [22] to track observed variance in event rates without storing past observations, which improves efficiency in both time and memory. In §III-B, we provide additional implementation details of Tintin's weighted version of Welford's method; this is evaluated in the Linux kernel against a traditional variance computation in §VI-B. Third, although we've already presented Tintin's elastic hardware event scheduler in [19], supplementary details relating input uncertainty to predictive uncertainty are added in §IV-A. Finally, Tintin handles joint scheduling of events from multiple profiling scopes; a previously omitted derivation of its policy is found in §IV-B.

Of particular relevance to the real-time operating systems

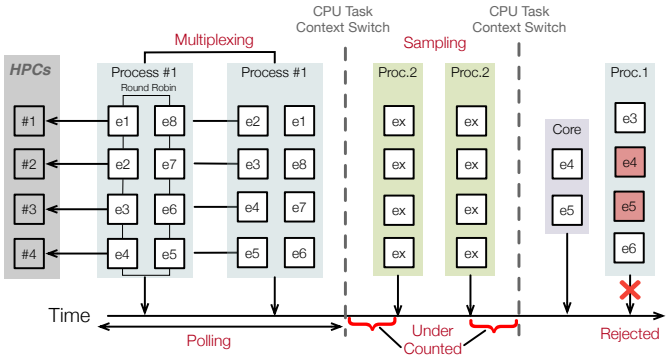

Fig. 1. The Linux *perf_event* subsystem multiplexes events on limited HPCs. It is not aware of scope overlap; joint scheduling and common attribution remain unsupported. If a per-core event is pinned to an HPC, per-task events may be rejected, leading to measurement starvation.

community in general, and this year's OSPERT workshop in particular, we provide a roadmap for using Tintin with the RT-Bench [23] open-source framework which integrates existing benchmarks into a recurrent real-time task model. A presentation of its latest updates also appears at this year's OSPERT [24]. RT-Bench uses Linux *perf_event* to sample hardware events during benchmark execution. in §V, we illustrate the minor code changes necessary to allow RT-Bench to use Tintin's elastic scheduling and uncertainty reporting to enable monitoring of more events than available HPCs with minimal losses in accuracy.

For completeness, §VI-A summarizes experimental results from [19] comparing Tintin to the Linux *perf_event* subsystem.

II. Background

A. Hardware Performance Monitoring

Hardware Performance Counters. Modern processors make hardware event profiling available to system software via a per-core performance monitoring unit (PMU), which provides a set of programmable hardware performance counters (HPCs) that can be configured individually to measure a specific type of microarchitectural event, e.g., cache or bus accesses, cache writebacks or refills, branch misprediction, etc. When enabled, an HPC increments whenever its programmed event occurs. The number of measurable events is substantial, typically exceeding several dozen on ARM processors (e.g., 58 on the Cortex-A53 [25] and 151 on the Cortex-A78 [26]) and over one thousand on Intel processors (e.g., 1,623 on HaswellX [27]). However, the number of available HPCs is very limited; most processors provide only 2-6 per physical core [27], [28], and this number is effectively halved per logical core when enabling Simultaneous Multithreading (SMT).

The Linux *perf_event* Subsystem. This is Linux's kernel-level abstraction layer for interacting with HPCs. It serves as the de facto infrastructure widely utilized by many tools such as PAPI [29], Intel EMON [30], VTune [31], pmu-tools [32], and the Linux Perf utility [28]. It provides access to hardware-level HPC data and software-level data (e.g., memory footprint and tracepoints). The former is the exclusive focus of Tintin.

Scheduling Events on Limited HPCs. Existing work has sought to select relevant events carefully for a given application. However, the number of events of interest often remains larger than the available HPCs. For example, in [33], the 120 events available on an ARMv7-A CPU were narrowed to only 34 of importance for predictive DVFS. CounterMiner [34], an offline analysis tool for predictive modeling with event counts, achieves the most accurate IPC predictions for HiBench [35] workloads using $\sim$150 events. Furthermore, some applications profile derived metrics, such as *Memory_Bound* [5]. These combine as many as 16 individual events [36]. Pond [14], which we evaluated as a case study for Tintin in [19], provides a memory pooling model for cloud infrastructure. Its latency-prediction model takes 7 derived metrics as inputs, spanning 20 events on Intel Skylake.

The current approach to managing this limitation is through event *multiplexing*. When an HPC is configured to count a particular event, we say that the event is *scheduled* on the counter. If the number of events to be monitored exceeds the number of available HPCs, they must time-share the counters; *perf_event* schedules events for monitoring in round-robin fashion [21], [27], [37], [38]. Fig. 1 presents an illustrative example in which there are 4 available HPCs, while Process #1 requires monitoring for 8 events. In this scenario, the events {e1, e2, e3, e4} are scheduled in the initial time slice, followed by events {e2, e3, e4, e5}. HPC measurements for each event can then be interpolated to estimate the total count, but this unavoidably introduces errors. For example, we found that the standard deviation for hardware event counts generated by the *541.leela_r* Go engine in the SPEC CPU®2017 benchmark suite [39] as reported by Linux Perf increased by over 6× when profiling 8 events compared to 4 [19].

Profiling Scope. The *perf_event* subsystem enables specification of hardware event monitoring for either individual tasks (processes/threads) or CPU cores. Upon CPU task scheduling, it first schedules events bound to the current core before adding those associated with the active process. Since events are bound to per-core and per-task data structures, they are managed independently, even if they share common events of interest. The right side of Fig. 1 illustrates this scenario: a user assigns two events, {e4, e5}, to the current core; these are placed on HPCs #2 and #3. Consequently, events {e4, e5} monitored for Process #1 are not scheduled, even though they represent the same event types, resulting in starvation.

B. Tintin

Tintin, a hardware event monitoring infrastructure introduced in our recently-published paper at OSDI [19], aims to solve these limitations via the three components in Fig. 2.

Quantification of Multiplexing Errors. Multiplexing errors can be quantified at runtime based on *variance* in observed event rates. The Tintin-Monitor component tracks variance, then reports *expected error* back to user-space applications alongside the measured counts, helping to better inform profiling-based decision-making. Details on Tintin-Monitor's

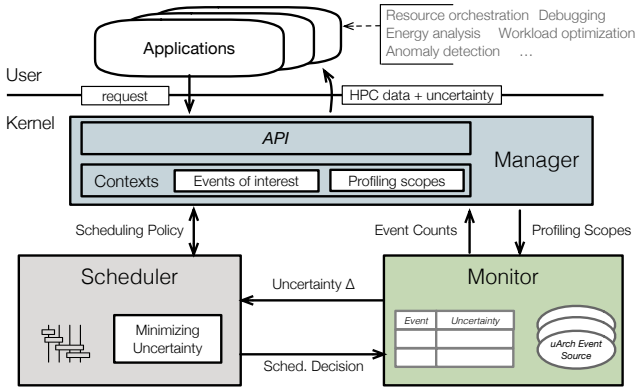

Fig. 2. Tintin design overview.

rate-based count interpolation and variance tracking which were omitted from [19] are provided in §III.

Scheduling to Minimize Uncertainty. By allocating more HPC time to events with larger variance, overall error is minimized. The Tintin-Scheduler component uses the errors reported by Tintin-Monitor to schedule events on HPCs, assigning each event a unique share of time with the objective of minimizing overall error. This problem is shown to be semantically equivalent to elastic scheduling. §IV-A provides deeper insights into this connection than were given in [19].

Indirection to Handle Profiling Scopes. An additional level of indirection can provide a uniform mechanism to handle the heterogeneity of profiling requirements. Tintin provides abstractions to handle issues related to profiling scope, including a uniform API that enables greater flexibility and granularity in specification via the Tintin-Manager component. It elevates a profiling scope to a first-class object, an Event Profiling Context (*ePX*). Tintin-Manager manages *ePX*s collectively for all applications, enabling overlapping events from concurrently-active scopes to be scheduled jointly. Details on how joint scheduling is handled within the elastic model, which were only mentioned cursorily in [19], are provided in §IV-B.

III. Count and Uncertainty Estimation

Formally, an event $e_i \sim (x_i, \sigma_i)$ has an estimated count x_i and uncertainty σ_i due to interpolation over multiplexed observations. In this section, we describe the design rationale behind Tintin's interpolation and uncertainty tracking mechanisms.

A. Interpolation with the Trapezoid Area Method

To derive the estimated total count x_i from a measured count x'_i, many tools (e.g., Linux Perf) use count-based interpolation: $x_i = x'_i/U_i$, where U_i is the fraction of time that event type e_i was scheduled on a counter. Several alternative methodologies are explored in [21], [40], [41]. Besides those that avoid multiplexing via multiple program runs and offline analysis, the trapezoid area method (TAM) —which uses rate-based interpolation— is suggested to be the most accurate [21].

Formally, an event e_i is measured with count x_i^j over some continuous time interval $I_i^j = [a_i^j, b_i^j]$ of duration δ_i^j. Then its average rate r_i^j over this interval is x_i^j/δ_i^j. Over consecutive intervals, TAM's original implementation [40] assumes that

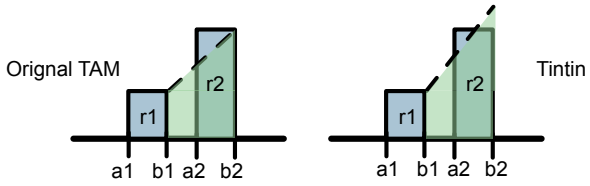

Fig. 3. Tintin's TAM implementation.

the arrival rate follows the straight line connecting the points (b_i^j, r_i^j),(b_i^{j+1}, r_i^{j+1}), and interpolates the count using the area of the resulting trapezoid, as illustrated in Fig. 3. However, this violates the following proposed invariant:

If an event e_i is always monitored, the produced event count estimate x'_i should equal the observed count x_i.

Tintin-Monitor instead constructs the trapezoid so that its top passes through the midpoint $((b_i^{j+1} + a_i^{j+1})/2, r_i^{j+1})$ of the second measured interval. This is derived as follows.

For event e_i, we assume a constant change in the rate of event arrival from interval I_1 to I_2. This is the line $r(t) = mt + c$ connecting the center of each interval at points:

$$\left(\frac{a_1 + b_1}{2}, r_1\right) \text{ and } \left(\frac{a_2 + b_2}{2}, r_2\right)$$

Solving for the slope:

$$m = \frac{r_2 - r_1}{\frac{a_2+b_2}{2} - \frac{a_1+b_1}{2}} = \frac{2(r_2 - r_1)}{a_2 + b_2 - a_1 - b_1}$$

Then solving for the intercept c:

$$r_1 = \frac{(r_2 - r_1)(a_1 + b_1)}{a_2 + b_2 - a_1 - b_1} + c$$

The formula can then be expressed as:

$$r(t) = \frac{2(r_2 - r_1)}{a_2 + b_2 - a_1 - b_1}t + r_1 - \frac{(r_2 - r_1)(a_1 + b_1)}{a_2 + b_2 - a_1 - b_1}$$

After measuring event e_i during interval I_2, we interpolate over the unmonitored time to estimate the total event arrival count Δx_i since the end of I_1 by integrating from b_1 to b_2:

$$\Delta x_i = 0.5 \cdot (r(b_1) + r(b_2)) \cdot (b_2 - b_1)$$

This equals:

$$\begin{aligned}\Delta x_i &= \left(\frac{(r_2 - r_1)(b_1 + b_2)}{a_2 + b_2 - a_1 - b_1} + r_1 - \frac{(r_2 - r_1)(a_1 + b_1)}{a_2 + b_2 - a_1 - b_1}\right) \cdot (b_2 - b_1) \\ &= \frac{(b_1 - b_2)(r_1(a_2 - b_1) + r_2(b_2 - a_1))}{a_1 - a_2 + b_1 - b_2}\end{aligned}$$

Invariance can be proven because, if $b_1 = a_2$ then we have:

$$\Delta x_i = \frac{(a_2 - b_2)(r_1(a_2 - a_2) + r_2(b_2 - a_1))}{a_1 - a_2 + a_2 - b_2}$$

$$\Delta x_i = \frac{(a_2 - b_2)(r_2)(b_2 - a_1)}{a_1 - b_2} = r_2(b_2 - a_2)$$

Which is precisely the count in interval I_2.

B. Tracking Variance with Welford's Method

As described in [19], for an event e_i with interpolated count x_i, we define uncertainty as the expected error σ_i in the count. The instantaneous rate of event arrival is a random variable $\mathbf{r}_i$, with a mean rate r_i^j obtained during each measurement interval I_i^j. Since it is infeasible to measure instantaneous rates, Tintin-Monitor instead characterizes expected variance $E(V(\mathbf{r}_i))$ by taking the variance of the sample means, weighted by duration. In many stochastic processes, variance scales linearly with time [42]; since we are measuring variance in *rate*, it therefore follows that the expected variance $V(x_i, t)$ in event *count* x_i over the time t that the event is not monitored can be expressed as $V(x_i, t) = V(\mathbf{r}_i) \cdot t^2$. Then the expected error σ_i in the estimated event count is the standard deviation $\sqrt{V(x_i, t)}$.

As noted in [19], to update the observed variance in sample means efficiently during online profiling, Tintin-Monitor uses a weighted version of Welford's method [22]. Unlike standard variance calculations, this only requires one pass through the data, obviating the need for Tintin to store all sampled data. At the end of monitoring intervals I_i^j, variance is updated as

$$V_i = V_{i-1} \times (r - \mu_i) \times (r - \mu_{i-1}) \tag{1}$$

where μ is the time-weighted mean over collected counts. To avoid involving floating-point operations in the involved division steps, we apply a scaling factor, where necessary, to the numerator then perform integer division to achieve a fixed-precision result. To prevent overflow, we use 64-bit integers and hand-tune the order of operations to avoid large values.

Because Welford's method is a one-pass technique, as new measurements are obtained, variance is updated in constant time. §VI-B shows an empirical comparison of the kernel overhead imposed by Welford's method, versus a traditional two-pass variance calculation.

IV. Elastic Scheduling to Minimize Uncertainty

A. Hardware Event Scheduling

Tintin-Scheduler aims to adjust allocations of time for events on limited HPCs to minimize overall uncertainty.

Scheduling Model. Formally, for a profiling scope, we define $\mathbf{E} = \{e_i\}$, the set of events to be monitored, where $|\mathbf{E}| = n$. Similarly, $\mathbf{C} = \{c_j\}$ denotes the set of HPCs available to monitor them, where $|\mathbf{C}| = m$. The problem is to determine a schedule $\mathcal{S}(t) : \mathbf{C} \to \{\mathbf{E}, \phi\}$ that defines, at each time t, the event assigned to each counter. We define an event's *utilization* U_i as the fraction of time it occupies a counter.

Scheduling Policy. Many applications use HPC data as inputs to statistical or learning-based models that predict program performance to inform decisions. For example, Pond [14] is a Microsoft Azure resource orchestration system. In cloud environments, memory pooling may improve DRAM utilization and reduce cost, but using pooled instead of local memory substantially increases the latency of some virtual machine workloads. To allocate limited local memory appropriately, Pond uses hardware event data to predict whether a VM workload's latency will be sensitive to pooled memory use.

Uncertainty in input values induces uncertainty in output predictions. Tintin-Scheduler is designed primarily to schedule events on HPCs so as to minimize output uncertainty in prediction, although it works in general to minimize overall event count uncertainty when no single predicted metric is targeted. Tintin-Miner, an offline analysis tool for which development is currently a work in progress, uses model-agnostic variance-based sensitivity analysis to gauge the impact of each input event on output fidelity. Specifically, Tintin-Miner employs Sobol's indices [43], which measure the contribution of each input variable to the output variance. By changing the counts of an individual event e_i and then measuring the variance of output, it calculates the event's first-order sensitivity index S_i^1, which represents the percentage of model output variance contributed individually by that event as an input to the model (i.e., the effect of changing the count x_i only).

To minimize predictive uncertainty, we therefore want to minimize the total variance, weighted by sensitivity, of the inputs. From §III-B, the expected error σ_i in the estimated count of event e_i is $\sqrt{V(\mathbf{r}_i)} \cdot t$, where t is the amount of unmonitored time. Future unmonitored time for event e_i is thus proportional to $1 - U_i$, where U_i is its utilization; thus, we express the expected error as $\sigma_i = \left(\sqrt{V(\mathbf{r}_k)}\right) \cdot (1 - U_i)$. Variance is thus $\sigma_i^2 = V(\mathbf{r}_i) \cdot (1 - U_i)^2$. The scheduling problem is therefore stated as the following constrained optimization:

$$\min_{U_i} \quad \sum_{i=1}^{n} w_i V(\mathbf{r}_i) \cdot (1 - U_i)^2 \tag{2a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} U_i \leq m \quad \text{and} \quad \forall_i, \quad U^{\min} \leq U_i \leq 1 \tag{2b}$$

Where $U^{\min}$ is a lower-bound on the scheduling quantum (minimum schedulable utilization time slice) to avoid event starvation or floods of timer interrupts. For generality, we use w_i to denote a weight assigned to event e_i; from our sensitivity model described above, this may be set equal to Sobol's first-order sensitivity index S_i^1. In general, if lacking sensitivity analysis, Tintin assigns $w_i = \frac{w_i^*}{x_i^2}$, where w_i^* is a user-specified weight (set via Tintin's syscall API) and x_i normalizes the standard deviation by the estimated event count.

Eqn. 2 is equivalent to the formulation in [44] of elastic scheduling as a constrained optimization problem. The elastic real-time model, proposed by Buttazzo et al. [45], [46], adapts task utilizations to avoid overload on limited processor resources. Here, we solve the similar problem of adapting the scheduling utilizations of hardware events on limited HPCs. Tintin-Scheduler uses our quasilinear-time elastic scheduling algorithm from [20], [47] to assign utilizations. In §VI-A, we show that Tintin's dynamic variance-weighted elastic scheduling policy improves the accuracy of its event count estimates, even in the absence of sensitivity data.

B. Scheduling Multiple Scopes

As mentioned in §II-B, Tintin collectively schedules all events from any active *EPX*s. The details were omitted from our accepted OSDI paper [19]; we outline the approach here.

Scheduling Model. The problem remains to determine a schedule $\mathcal{S}(t) : \mathbf{C} \to \{\mathbf{E}, \phi\}$ that defines, at each time t, the event assigned to each counter. The complication now is that multiple active *EPX*s may share events in common. For example, the system might be monitoring events $\{e1, e2, e3, e4\}$ on core 0, while a task running on that core monitors events $\{e1, e2, e5, e6\}$. It does not make sense to treat $e1$ and $e2$ separately for each profiling scope for purposes of attribution: if counts are read for either event, they should be attributed to both *EPX*s. Otherwise, unmonitored time becomes unnecessarily inflated, contributing additional uncertainty. However, each *EPX* independently tracks counts and variance for its events, which raises questions about how these should be used as inputs to the scheduling problem.

We denote event $e^*_{i,j} \sim (x_{i,j}, \sigma_{i,j}, w_{i,j}, e_{i,j})$ as the i^{th} event associated with the j^{th} *EPX*. As usual, x and σ are the estimated counts and error; w denotes its weight (see §IV-A) and e denotes the identifier of the hardware event, implying that $e_{a,j}$, $e_{b,j}$ should be unique within a single *EPX*, but $e_{a,j}$ and $e_{b,k}$ might be the same, indicating common events among *EPX*s. Each *EPX* is also assigned a weight z_j. This defaults to 1, but is user-settable via Tintin-Manager's syscall API.

Scheduling Policy. Our objective now is to minimize total weighted uncertainty among the predictive models underpinning each *EPX*. By extension of Eqn. 2, this can be stated as:

$$\min_{U_k} \quad \sum_{i=1}^{n}\sum_{j=1}^{m} z_{i,j} w_{i,j} \sigma^2_{i,j} \cdot (1 - U_k)^2 \tag{3a}$$

$$\text{s.t.} \quad \sum_{U_k} \leq m \quad \text{and} \quad \forall_k, \quad U^{\min} \leq U_k \leq 1 \tag{3b}$$

where we assume there are m active *EPX*s, and we denote the utilization of event $e_{i,j}$ as U_k to reflect that there may be events in common. We may instead denote the collection of monitored *events* $\{e_k\}$, where each event is assigned a set of values $\{z_{k,j}\}$, $\{w_{k,j}\}$, $\{\sigma_{k,j}\}$ according to the *EPX*s that track it. If an event is not tracked by some *EPX*, these values may be set to 0. We can then re-state the optimization problem:

$$\min_{U_k} \quad \sum_{e_k} \left(\sum_{j=1}^{m} z_{k,j} w_{k,j} \sigma^2_{k,j} \right) \cdot (1 - U_k)^2 \tag{4a}$$

$$\text{s.t.} \quad \sum_{U_k} \leq m \quad \text{and} \quad \forall_k, \quad U^{\min} \leq U_k \leq 1 \tag{4b}$$

Notice that this is exactly the quadratic optimization problem in Eqn. 2, but with coefficients $\left(\sum_{j=1}^{m} z_{k,j} w_{k,j} \sigma^2_{k,j}\right)$ for each event e_k. It therefore can be solved using the same algorithm with transformed inputs.

V. RT-Bench Integration

RT-Bench [23] is an open-source tool to address the benchmarking needs of real-time systems. Rather than providing its own computational workloads, RT-Bench serves as a framework in which existing benchmarks, such as Isolbench [48] or SD-VBS [49], can be integrated. RT-Bench provides a wrapper

```
-   int fd = syscall(__NR_perf_event_open, &attr, …);
+   int fd = syscall(__NR_tintin_event_open, &attr, …);
```

```
-     read(fd, &measurement, sizeof(struct read_format));
+     read(fd, &measurement, sizeof(struct tintin_read_format));
      long unsigned value = measurement.value;
+     long unsigned uncertainty = measurement.uncertainty;
```

Fig. 4. Simplified code snippets for using Tintin instead of Linux *perf_event* in RT-Bench. Both changes are made to files in `generator/src`. **(Top)**: Set up monitoring in `performance_counters.c`. **(Bottom)**: Retrieve count and uncertainty measurements in `performance_sampler.c`.

to run benchmark workloads as periodic tasks using a specified period and deadline under Linux's real-time schedulers. It supports processor pinning and constraints on memory allocation.

RT-Bench reports performance statistics from the running benchmarks, including several hardware event counts. When initializing a benchmark workload, it establishes monitoring via Linux *perf_event*. It then launches a dedicated performance monitoring thread, PMThread, to perform high-frequency HPC sampling via returned file descriptors. In its original release, RT-Bench only monitored L2 cache refills [23], but it has since been updated to monitor L1 and L2 references and refills, retired instructions, and CPU cycles. These, and other updates, are also presented at this year's OSPERT [24].

In its current version, RT-Bench does not monitor more events than the number of HPCs, and it assigns all events to the same group, guaranteeing simultaneous monitoring. However, as stated in [23], RT-Bench may benefit from adding more performance counters. Indeed, access patterns to other shared hardware resources, such as the L3 cache, memory bus [50], and TLB [51], may cause significant contention and delays in real-time task execution [52]–[55].

With its ability to significantly reduce measurement errors due to HPC multiplexing, Tintin is a suitable candidate for use in RT-Bench instead of *perf_event*. Moreover, Tintin can report both event counts *and* variance-based uncertainty in those counts. If greater confidence is desired, an RT-Bench user can make an informed decision to rerun a benchmark, either assigning greater weight to selected events for elastic scheduling (the w_i values in Eqn. 2a), or running the benchmark multiple times, each time profiling different subsets of the events of interest. Integrating Tintin into RT-Bench is straightforward, requiring the minimal code changes shown in Fig. 4.

VI. Evaluation

A. Overview of Prior Results

Here, we summarize a few of the key results from our OSDI paper on Tintin's performance [19].

Accuracy and Overhead. We used the SPEC CPU®2017 [39] and PARSEC 3.0 [56] benchmark suites to assess the accuracy of the event counts collected online by Tintin, as well as its runtime overhead, in comparison to Linux *perf_event*. We select the 24 default predefined events in Linux *perf_event* to profile simultaneously [57]. To measure event count accuracy, we obtain ground truth by pinning one event to an HPC in each run. We repeat the experiments for the first 5 predefined

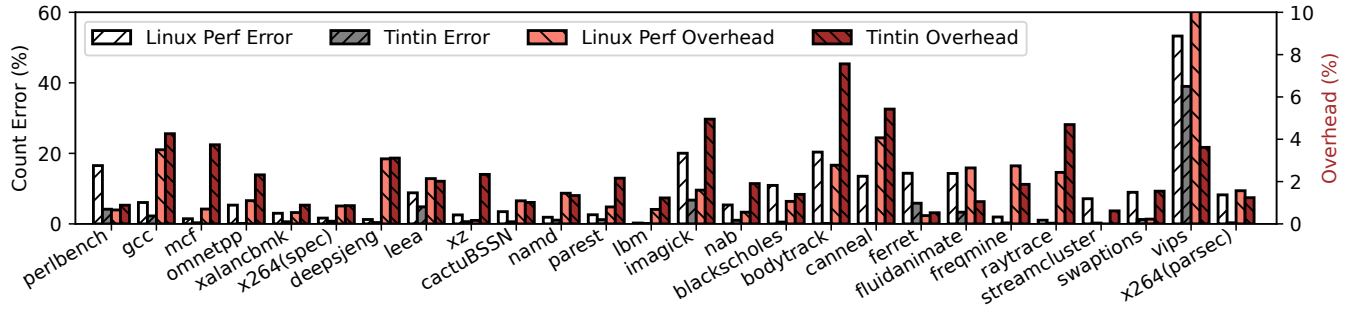

Fig. 5. Benchmark accuracy and overhead results from [19].

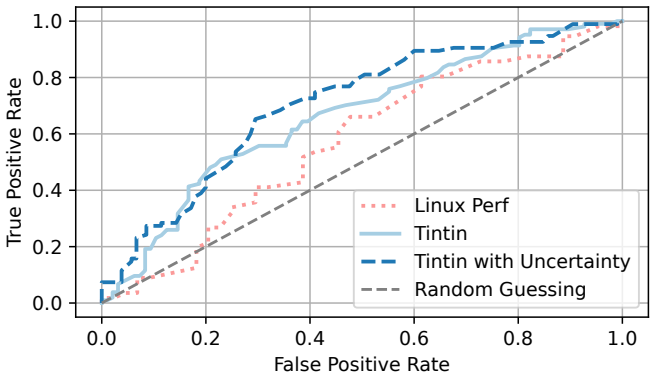

Fig. 6. HPC-based rootkit detection accuracy.

events in `PERF_TYPE_HARDWARE` in [57]. Runtime overheads are obtained by measuring mean benchmark execution times across 10 runs; these are normalized to the execution times without profiling. Results are shown in Fig. 5.

Event counts obtained with Linux *perf_event* were, on average, 9.01% off from the ground truth, with a maximum error of 53.27%. In comparison, Tintin's errors remained well under 5% in most cases, with an average of 2.91%. Tintin exhibits an average overhead of only 2.4%, only slightly higher than *perf_event*'s 1.9%. In the worst-case, we observed Tintin's overhead reached up to 7.6% while *perf_event*'s reached up to 12.7%. Tintin's achieves better execution time performance in scenarios where elastic scheduling allows fewer interrupts.

Impact of Quantifying Uncertainty. Many applications use HPC data to make predictions, e.g., to identify malicious behavior in intrusion detection systems. To evaluate Tintin's ability to enhance such systems, we adopted the experimental setup from [17] on detecting Linux rootkits. As malware, we use the open-source rootkit Diamorphine [58]. We train a random forest classifier (as in [17], [59]) on the events defined under `PERF_TYPE_HARDWARE`. We compare its accuracy when collecting event counts using *(i) perf_event*; *(ii)* Tintin with just elastic scheduling, weighting each event equally; and *(iii)* additionally using the measurement uncertainty reported by Tintin as additional inputs to the classifier. Fig. 6 presents the resulting ROC curves. The area under the curve (AUC) for *perf_event* is 0.57. Tintin improves the AUC to 0.66 with elastic scheduling, and to 0.70 with reported uncertainty.

B. Benefits of Welford's Method

To highlight the benefit of using Welford's method over a traditional two-pass variance calculation, we profile the in-

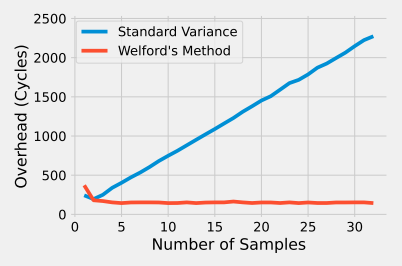

Fig. 7. Comparison of overheads induced by variance computation.

kernel execution time of updating the running variance for an event. For direct comparison, we implement a Linux kernel module that simulates HPC reads and run it on a Raspberry Pi 3 Model B+. Counts and monitoring interval durations are written into an array for use by traditional weighted variance, and we use Tintin-Monitor's *update_variance* function to update the running variance attached to an *ePX* with Welford's method. Overheads, in processor cycles, are measured for each approach every time a sample is obtained. Results are shown in Fig. 7; unsurprisingly, the standard two-pass variance computation times grow linearly with the number of number of samples (about 69 cycles per sample), while Welford's method remains relatively constant (150 cycles on average). We note that the initial larger overheads for the first few samples are likely due to cache effects.

VII. Conclusions and Future Work

This paper presents Tintin [19], a new hardware event profiling infrastructure originally published at OSDI, to the real-time systems community. Tintin aims to schedule events to reduce errors due to multiplexing atop limited HPCs. Tintin is a general-purpose, Linux-based tool. Although it does not specifically target real-time systems, hardware profiling is nonetheless important to our community, as evidenced by the PMThread functionality in RT-Bench [23], [24]. Moreover, Tintin leverages the elastic scheduling model of Buttazzo et al. [45], [46] to minimize measurement uncertainty.

As future work, we intend to release Tintin-Miner, an offline analysis tool that will identify relationships among events so as to identify and remove redundant events from the monitoring pool. It will also perform the sensitivity analysis described in §IV-A to inform weights for Tintin-Scheduler. We will also extend the elastic scheduling policy with principled support for event groups and CPU platforms where certain HPC registers cannot accommodate certain event types.

Acknowledgment

This work was supported by the NSF (CNS-2154930, CNS-2229427, CNS-2141256, CNS-2403758, CNS-2229290), the ARO (W911NF-24-1-0155), the ONR (N00014-24-1-2663), a WashU OVCR seed grant, and Intel.

References

[1] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, "Deterministic process groups in dos," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USA: USENIX Association, 2010, p. 177–191.

[2] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USA: USENIX Association, 2017, p. 377–389.

[3] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient deterministic multithreading in software," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, p. 97–108. [Online]. Available: https://doi.org/10.1145/1508244.1508256

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[5] T. A. Khan, I. Neal, G. Pokam, B. Mozafari, and B. Kasikci, "Dmon: Efficient detection and correction of data locality problems using selective profiling," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. Virtual: USENIX Association, Jul. 2021, pp. 163–181. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/khan

[6] D. Chen, D. X. Li, and T. Moseley, "Autofdo: automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 12–23. [Online]. Available: https://doi.org/10.1145/2854038.2854044

[7] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson, "Lightweight, high-resolution monitoring for troubleshooting production systems," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 103–116.

[8] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 46–55, 2009.

[9] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen, "Power containers: An os facility for fine-grained power and energy management on multicore servers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 65–76, 2013.

[10] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars, "Happy: Hyperthread-aware power profiling dynamically," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 211–218.

[11] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: An intelligent fine-grained resource management framework for slo-oriented microservices," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[12] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[13] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen, "Dna: Dynamic resource allocation for soft real-time multicore systems," in *27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Nashville, TN, USA: IEEE, 2021, pp. 196–209.

[14] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 574–587. [Online]. Available: https://doi.org/10.1145/3575693.3578835

[15] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "Dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190555

[16] S. Das, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *Symposium on Security and Privacy (SP)*, IEEE. Oakland,CA,USA: IEEE, 2019, pp. 20–38.

[17] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 559–570. [Online]. Available: https://doi.org/10.1145/2485922.2485970

[18] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd international workshop on performance analysis of workload optimized systems, FastPath*, vol. 13. Austin, TX: -, 2013, p. 5.

[19] A. Li, M. Sudvarg, Z. Li, S. Baruah, C. Gill, and N. Zhang, "Tintin: A unified hardware performance profiling infrastructure to uncover and manage uncertainty," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025. [Online]. Available: https://sudvarg.com/publications/OSDI25_Tintin.pdf

[20] M. Sudvarg, C. Gill, and S. Baruah, "Linear-time admission control for elastic scheduling," *Real-Time Systems*, vol. 57, no. 4, pp. 485–490, Oct 2021. [Online]. Available: https://doi.org/10.1007/s11241-021-09373-4

[21] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan, "Time interpolation: So many metrics, so few registers," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. USA: IEEE Computer Society, 2007, p. 286–300. [Online]. Available: https://doi.org/10.1109/MICRO.2007.42

[22] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[23] M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso, "RT-Bench: An extensible benchmark framework for the analysis and management of real-time applications," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, 2022, pp. 184–195.

[24] M. Nicolella, D. Hoornaert, and R. Mancuso, "RT-Bench: A long overdue update," in *Proc. of 19th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2025.

[25] A. Limited, "Arm cortex-a53 technical reference manual r0p4," https://developer.arm.com/documentation/ddi0500/j/Performance-Monitor-Unit/Events?lang=en, accessed 2023-11-16.

[26] ——, "Arm cortex-a78 technical reference manual," https://developer.arm.com/documentation/101430/0102/Debug-descriptions/Performance-Monitoring-Unit/PMU-events, accessed 2023-11-16.

[27] G. Zellweger, D. Lin, and T. Roscoe, "So many performance events, so little time," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2967360.2967375

[28] P. Wiki, "Tutorial: Linux kernel profiling with perf," https://perf.wiki.kernel.org/index.php/Tutorial, 2023, accessed on: 2023-11-16.

[29] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, p. 189–204, aug 2000. [Online]. Available: https://doi.org/10.1177/109434200001400303

[30] Intel Corporation, "Emon user's guide," https://www.intel.com/content/www/us/en/content-details/686077/emon-user-s-guide.html, 2023, accessed: 2023-12-01.

[31] J. Reinders, *VTune performance analyzer essentials*. Santa Clara: Intel Press, 2005, vol. 9.

[32] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[33] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 409–425.

[34] Y. Lv, B. Sun, Q. Luo, J. Wang, Z. Yu, and X. Qian, "Counterminer: Mining big performance data from hardware counters," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Fukuoka, Japan: IEEE Press, 2018, p. 613–626. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00056

[35] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*. IEEE, 2010, pp. 41–51.

[36] I. Corp., "Intel 64 and ia-32 architectures software developer manuals," https://software.intel.com/en-us/articles/intel-sdm, 2016, accessed 2019-03-05.

[37] M. Dimakopoulou, S. Eranian, N. Koziris, and N. Bambos, "Reliable and efficient performance monitoring in linux," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Salt Lake City, Utah: IEEE Press, 2016.

[38] S. S. Banerjee, S. Jha, Z. Kalbarczyk, and R. K. Iyer, "Bayesperf: Minimizing performance monitoring errors using bayesian statistics," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 832–844. [Online]. Available: https://doi.org/10.1145/3445814.3446739

[39] J. Bucek, K.-D. Lange, and J. v. Kistowski, "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, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[40] W. Mathur and J. Cook, "Toward accurate performance evaluation using hardware counters," pp. 23–32, 2003.

[41] ——, "Improved estimation for software multiplexing of performance counters," in *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '05. USA: IEEE Computer Society, 2005, p. 23–34.

[42] C. W. Gardiner, *Stochastic Methods: A Handbook for the Natural and Social Sciences*, 4th ed. Springer, 2009. [Online]. Available: https://link.springer.com/book/10.1007/978-3-540-70713-4

[43] I. M. Sobol, "Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates," *Mathematics and computers in simulation*, vol. 55, no. 1-3, pp. 271–280, 2001.

[44] T. Chantem, X. S. Hu, and M. D. Lemmon, "Generalized elastic scheduling," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, ser. RTSS '06. USA: IEEE Computer Society, 2006, p. 236–245. [Online]. Available: https://doi.org/10.1109/RTSS.2006.24

[45] G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," in *Proceedings of the IEEE Real-Time Systems Symposium*, ser. RTSS '98. USA: IEEE Computer Society, 1998, p. 286.

[46] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Transactions on Computers*, vol. 51, no. 3, pp. 289–302, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/12.990127

[47] M. Sudvarg, C. Gill, and S. Baruah, "Improved implicit-deadline elastic scheduling," in *2024 IEEE 14th International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2024, pp. 50–57.

[48] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–12.

[49] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 55–64.

[50] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in Multi-Core systems," in *16th USENIX Security Symposium (USENIX Security 07)*. Boston, MA: USENIX Association, Aug. 2007. [Online]. Available: https://www.usenix.org/conference/16th-usenix-security-symposium/memory-performance-attacks-denial-memory-service-multi

[51] S. A. Panchamukhi and F. Mueller, "Providing task isolation via tlb coloring," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 3–13.

[52] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 357–367.

[53] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 200–212.

[54] M. Bechtel and H. Yun, "Memory-aware denial-of-service attacks on shared cache in multicore real-time systems," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[55] A. Li, M. Sudvarg, H. Liu, Z. Yu, C. Gill, and N. Zhang, "Polyrhythm: Adaptive tuning of a multi-channel attack template for timing interference," in *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 225–239.

[56] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 72–81. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[57] man7.org Linux Man-pages project, *perf_event_open(2) - Linux Manual Page*, 2023, accessed: 2024-11-02. [Online]. Available: https://man7.org/linux/man-pages/man2/perf_event_open.2.html

[58] m0nad, "Diamorphine - lkm rootkit for linux kernels," https://github.com/m0nad/Diamorphine, 2023, accessed: 2025-04-06.

[59] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 457–468. [Online]. Available: https://doi.org/10.1145/3196494.3196515