
# Tintin: A Unified Hardware Performance Profiling Infrastructure to Uncover and Manage Uncertainty

Ao Li Marion Sudvarg Zihan Li Sanjoy Baruah Chris Gill Ning Zhang
*Washington University in St. Louis*

## Abstract

Hardware performance counters (HPCs) enable the measurement of microarchitectural events, which are crucial for tracking and predicting program behavior. High-fidelity measurement and precise attribution are essential for accurate profiling. However, existing profiling tools have fundamental challenges in both aspects. In measurement, numerous events compete for limited hardware monitoring resources; while for attribution, applications have diverse requirements, but systems provide limited support. Existing tools mitigate the former limitation through event multiplexing, but this approach introduces non-trivial errors. The latter limitation, however, remains largely unaddressed.

This paper introduces *Tintin*, an HPC profiling infrastructure with a modular three-component design that addresses both challenges. Tintin introduces mechanisms to mitigate multiplexing errors by characterizing uncertainty at runtime, scheduling events to minimize it, and reporting uncertainty to applications. It also proposes the Event Profiling Context (*ePX*) as a new OS primitive to unify diverse profiling requirements. Tintin is evaluated using benchmarks as well as real-world resource orchestration, performance debugging, and intrusion detection systems, to demonstrate its ability to improve hardware profiling with low runtime overhead.

## 1 Introduction

Performance profiling involves *measuring* runtime cost metrics and *attributing* these measurements to a portion of one or more programs' execution. Modern processors provide hardware performance counters (HPCs) that can profile microarchitectural events (e.g., cache loads and misses). The hardware events collected from HPCs, referred to as HPC data, are valuable across a wide range of application domains, including debugging [5,23,49,50], workload optimization [8,15,35], power analysis [58,60,76], diagnostics [8], online resource provisioning [25,27,37,54,72], and intrusion detection [19,20]. However, existing profiling infrastructures for hardware events have fundamental limitations in both measurement and attribution.

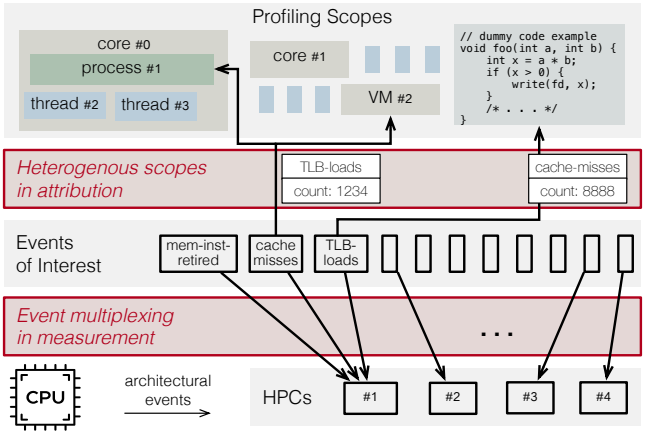



**Figure 1:** An event profiling infrastructure must accurately measure and attribute microarchitectural events to a heterogeneous set of profiling scopes, while managing a limited number of HPCs.

**Problems.** When measuring HPC data, each individual HPC can be programmed to record only a single type of event at a time, but the number of events of interest (typically tens to hundreds) often exceeds the available HPCs (usually 2–6), as shown in Figure 1. Therefore, many existing solutions [44,46] adopt *event multiplexing*, scheduling events in a time-shared manner (where each event typically receives an equal portion of time). Observed counts are interpolated to estimate the total; however, this multiplexing inevitably introduces errors, as events may remain unmonitored for extended intervals. Most approaches to avoiding multiplexing involve running the target application multiple times, each time monitoring a different subset of events, and then merging the results into a single trace [31,41,48]. However, these offline techniques are impractical for emerging applications that use HPC data as real-time input for online decision-making [25,35,54,72], such as dynamic resource orchestration.

Furthermore, hardware event attribution remains challeng-

ing. Unlike software events [6] (e.g., memory usage and page faults) that are directly tied to profiled program execution, hardware events are counted independently by standalone HPCs. Attribution requires system support to align these events with program execution. However, there is a fundamental mismatch between the abstractions provided by existing tools [10, 33, 53, 55, 73] and application requirements. Existing tools only allow profiling events for specific tasks (i.e., threads/processes) or cores. For task-level profiling, HPCs start counting when the target task is scheduled in, then stop when it is switched out. While this design is straightforward to implement – in Linux, events are bound to the task_struct and logic is added to the CPU rescheduling routine – it means the profiling scope[1] is bound to schedulable entities (e.g., task/core switches). However, there is a significant need to define more flexible profiling scopes in practice. For example, developers may want to profile events for specific code regions [1, 4, 35, 62, 64] as shown in Figure 1. Current abstractions are insufficient for these heterogeneous profiling requirements, leading to event misattribution. This heterogeneity might also cause conflicts between overlapping scopes that result in some events remaining unmeasured.

**Our Solution – Tintin.** This paper proposes Tintin, a new hardware event profiling infrastructure that harnesses two key ideas to address the aforementioned challenges. First, Tintin characterizes multiplexing errors as uncertainty at runtime. Leveraging this, Tintin implements an event scheduling algorithm to minimize the overall uncertainty. The algorithm is grounded in real-time scheduling theory and is proven to be optimal. Since uncertainty cannot be fully eliminated, Tintin also reports it to the application via user-space interfaces; to the best of our knowledge, this is the first profiling tool to do so. We present case studies that demonstrate how these reported values may improve application-level decision making. Second, to address the inflexibility of attribution, Tintin proposes to elevate profiling scopes via a first-class object, the Event Profiling Context (*ePX*). Heterogeneous *ePX*s are uniformly managed across the entire system, allowing for flexible scope definition while resolving conflicts between overlapping scopes by jointly scheduling their events.

The system design of Tintin consists of three modular kernel components, each introducing techniques to improve accuracy and reduce overhead in realizing the basic design. *Tintin-Monitor* simultaneously characterizes uncertainty while measuring event counts. As ground truth is not available at runtime, it estimates uncertainty using variance as a proxy. To reduce overhead, it adopts incremental variance updates. *Tintin-Scheduler* represents event multiplexing as an elastic real-time scheduling problem [12, 13] to minimize total expected uncertainty. It presents a novel extension of elastic scheduling to multiple HPCs, assigning shares of HPC time and constructing a schedule in quasilinear time. *Tintin-Manager* serves as another level of indirection to uniformly handle and manage the heterogeneous attribution requirements originating from user space. It extends Linux's familiar perf_event API to specify explicit profiling scopes with flexible granularity and report uncertainty back to user space.

**Summary of Results.** Tintin is evaluated on three real-world applications: Pond [37] for cloud resource provisioning, DMon [35] for performance debugging, and the Diamorphine rootkit [43] for intrusion detection. In these case studies, we demonstrate that Tintin can be easily integrated into existing applications in place of other profiling tools with minimal adaptation. By tracking, minimizing, and reporting uncertainty, Tintin allows Pond to predict latency sensitivity with 64% greater accuracy, and the AUC for Diamorphine rootkit classification increases by 22.8%. Additionally, Tintin's operability to adjust targeted profiling scope allows DMon to identify performance issues in a push-button fashion. Furthermore, we conduct a comprehensive evaluation to assess accuracy improvements, runtime overhead, and scalability on SPEC 2017 [11] and PARSEC [9] benchmarks. Results indicate that Tintin incurs low overhead (2.4%) while interpolating multiplexed event counts 3.09× more accurately than the state of the art.

**Contributions.** This paper proposes Tintin, a kernel infrastructure for hardware event profiling. It includes:

- An uncertainty-driven hardware performance measurement mechanism that quantifies, reports, and schedules to minimize estimated errors.
- A new OS primitive, the Event Profiling Context, to enable flexible and accurate attribution while resolving compatibility issues between conflicting profiling scopes.
- The design, implementation, and evaluation of Tintin[2], a kernel hardware event profiling infrastructure with a three-component modular structure.

## 2 Background

**Performance Monitoring 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 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

[1] Here, we adopt definitions from Saltzer and Schroeder's subject-object model [56, 57]. Hardware events are the *objects* of profiling, while the corresponding sources of those events are the *subjects*. A "scope" therefore defines a subject to profile. This could be execution on CPU core(s), across a virtual machine, a single thread, or originating from a specific code segment.

[2] Source code is available at: https://github.com/WUSTL-CSPL/tintin-kernel and https://github.com/WUSTL-CSPL/tintin-user.

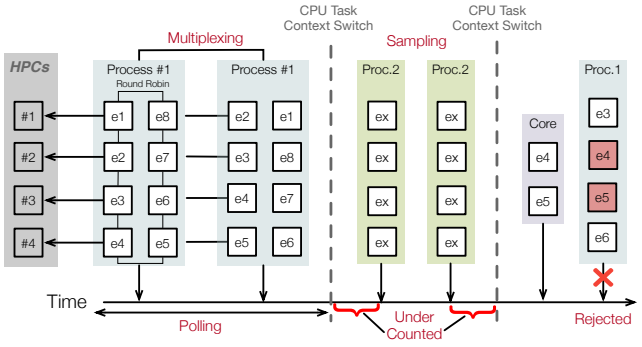



**Figure 2:** The Linux perf_event subsystem multiplexes events on limited HPCs. It is not aware of scope overlap; joint scheduling and common attribution remain unsupported. If a per-core event is pinned to an HPC, per-task events may be rejected, leading to measurement starvation. perf_event supports both instrumented polling and time- or event-triggered sampling. When sampling, events may be undercounted when the sampling intervals do not align with task context switches.

dozen on ARM processors (e.g., 58 on the Cortex-A53 [38] and 151 on the Cortex-A78 [39]) and over one thousand on Intel processors (e.g., 1,623 on HaswellX [75]).

### 2.1 Linux perf_event Subsystem

The Linux perf_event subsystem [65] 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 [10], Intel EMON [33], VTune [55], pmu-tools [73], and the Linux Perf utility [70]. It provides access to both hardware-level HPC data and software-level data (e.g., memory footprint and tracepoints). This section discusses its hardware aspects, which are the exclusive focus of Tintin.

Polling vs. Sampling. Events on HPCs can be monitored through either *polling* [44,62] or *sampling* [21,74]. In polling, HPCs are configured and enabled at explicit points during program execution (via instrumented syscalls) and then read at later points. With sampling, HPCs are read periodically, either in time-triggered fashion at periodic intervals (e.g., by explicit timer handlers or in the system timer interrupt routine) or in event-triggered fashion every time a pre-defined threshold is reached (e.g., using Intel's Precise Event-Based Sampling (PEBS) hardware or ARM's PMU interrupt). It is worth noting that sampling does not necessarily incur less overhead than polling. They in fact represent a trade-off between intrusiveness and precision. Polling typically requires instrumentation but offers higher accuracy by placing calipers at the boundaries of profiling scopes. In contrast, sampling does not require instrumentation but suffers from misattribution, as it does not align events precisely with the target scope, such as task context switches as shown in Figure 2.

Event Scheduling. 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, being multiplexed in a round-robin fashion. Figure 2 presents an illustrative example in which the available HPCs on the CPU are limited to 4, while Process #1 requires monitoring for 8 events. In this scenario, the events {e1, e2, e3, e4} are allocated to the initial time slice, followed by the scheduling of events {e2, e3, e4, e5} in the subsequent time slice.

Profiling Scope. The perf_event subsystem enables specification of hardware event monitoring for either individual tasks (processes/threads) or CPU cores. A key advantage of this design is its simplicity as the kernel only needs to track specified events and their counts within existing data structures for the profiled subject, e.g., the task_struct. It additionally requires minimal additional logic in the task scheduler to switch between the events associated with different tasks or cores.

Upon CPU task scheduling, perf_event 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 Figure 2 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 unnecessary starvation.

### 2.2 Sources of Uncertainty in HPC Profiling

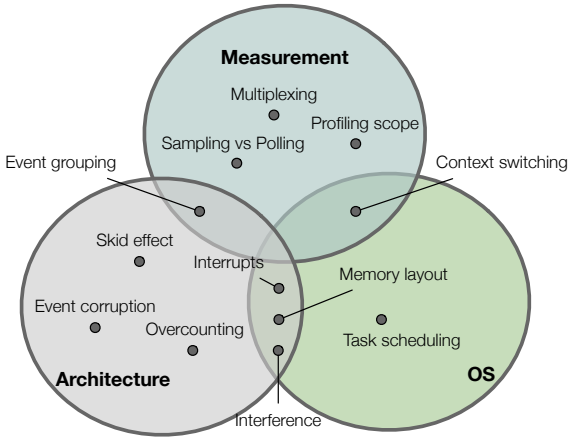



**Figure 3:** Uncertainty sources.

Sources of HPC measurement uncertainty can originate from a combination of three factors: architectural uncertainty, the operating system, and measurement method limitations, as illustrated in Figure 3.

**Architectural Uncertainty.** This type of uncertainty is caused by flaws in the microarchitecture. Several known sources of such uncertainty have been studied in prior work.

- Skid effect [28, 74]: To support event-triggered sampling, many PMUs can be configured to deliver an interrupt when an HPC overflows. On some processors, additional instructions execute during the time delay between interrupt initiation and signal delivery to the CPU; associated events may be incorrectly counted or attributed.
- Event corruption [21]: On some Intel processors with Hyper-Threading, certain event counts will be corrupted if monitored by matching counters on the same physical core.
- Overcounting [67, 68]: Certain events will cause an unrelated counter to increment on several older x86 processors from both Intel and AMD.

**OS Uncertainty.** Profiling might not yield identical results across different runs of the same program, primarily because the runtime environment can vary between measurements. For example, factors such as OS activity, program scheduling in multitasking environments, memory layout and pressure, and multi-processor interactions may differ from run to run.

**Measurement Method Uncertainty.** Many tools have been developed to facilitate the collection of HPC data through various measurement methods. However, these methods often introduce uncertainty. For example, as discussed in the previous section, the Linux perf_event subsystem provides profiling support through several mechanisms, each contributing to potential measurement error. These include the data collection technique (e.g., polling or sampling), the profiling scope (e.g., thread, process, core, or multiple cores), and the use of multiplexing. As polling and sampling each represent a different trade-off between accuracy and intrusiveness, removing their uncertainty is out of Tintin's scope. In fact, Tintin works seamlessly with either technique.

## 3 Challenges and Pitfalls

**Limited HPCs Compared to Events of Interest.** Modern CPUs support dozens to hundreds of types of events that can be monitored. However, the number of available HPCs is very limited; most processors provide only 2–6 HPCs per physical core [70, 75], and this number is effectively halved per logical core when Simultaneous Multithreading (SMT) is enabled. Additionally, Linux reserves one HPC by default for the watchdog timer, further reducing the counters available. This creates a fundamental tension between the number of events of interest and the available HPCs.

Can We Profile Fewer Events? Existing work has sought to select relevant events carefully for a given application. However, the number of events of interest remains large. For example, in [45], the 120 events available on an ARMv7-A CPU were narrowed to just 34 of importance for predictive DVFS. CounterMiner [41], an offline analysis tool for predictive modeling with event counts, achieves the most accurate IPC predictions for HiBench [32] workloads using ~150 events. Furthermore, some applications profile derived metrics, such as

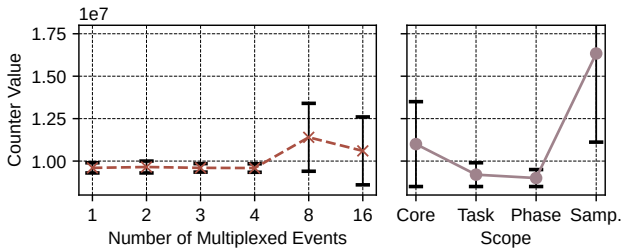



**Figure 4:** Preliminary examination of multiplexing and scope-based errors. (Left) When the number of events exceeds the number of HPCs, variation in reported event counts increases. (Right) A precise definition of profiling scopes is also essential for accurate results.

Memory_Bound [35]. These combine as many as 16 individual events [18]. Pond [37], which we evaluate as a case study in §8.1.1, provides a memory pooling model for cloud infrastructure. Its model takes 7 derived metrics as inputs, spanning 20 events on Intel Skylake.

Event Multiplexing Introduces Errors. The current approach to managing this limitation is through event *multiplexing*. In the Linux perf_event subsystem, events are scheduled for monitoring in a time-shared fashion on the available HPCs [2, 21, 47, 75]. HPC measurements for each event can then be interpolated to estimate the total count. However, this method unavoidably introduces errors, as the reported values are based on interpolation, and variance in event arrival further contributes to inaccuracies in the results.

Examination of Multiplexing Error. To illustrate this effect, we use the Linux Perf utility to profile the 541.leela_r Go engine in the SPEC CPU®2017 benchmark suite [11] on an Intel Xeon Gold 6130 Skylake CPU. The target PMU has 4 generic counters, so we profiled 1–4 (non-multiplexing), 8, and 16 selected events. The multiplexing frequency in the experiments is set to the default value (4 ms) in Linux perf_event, which aligns with the CPU task scheduling frequency. The mean and standard deviation of reported LLC-load-misses over 10 runs are plotted in Figure 4. Without multiplexing, reported event counts remain stable; they do not vary significantly with the number of events profiled. However, when the number of events exceeds the available counters, the reported metrics vary significantly, as multiplexing fails to capture the true characteristics of the workload [59, 75].

**Heterogeneous Profiling Scopes.** HPCs are simply counters and are decoupled from the profiling scope. Therefore, profiling infrastructure must align recorded events with program execution on the CPU. Developing such mechanisms is challenging due to the heterogeneity of profiling requirements.

Incomplete Handling of Diverse Scopes. Applications require event attribution to diverse scopes, such as execution instances [25, 37, 54] (e.g., a process, a core, a VM) or code segments [1, 35, 62, 64] (e.g., functions, basic blocks). Existing abstractions only explicitly bind events to per-process or

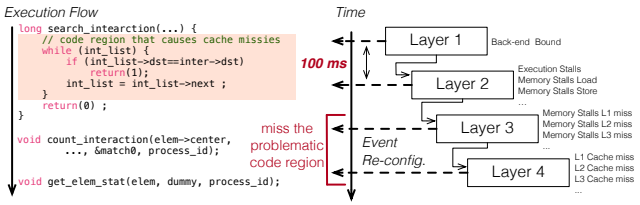



**Figure 5:** In DMon [35], inflexible profiling scopes cause event misattribution and thus misidentification of data locality issues.

per-core scopes. While the sampling-based method can record program counters (i.e., code addresses) and map events back to source code using debugging tables, this approach cannot filter events for a specific scope and suffers from sampling bias, favoring frequently executed code [16].

Ignorance about Overlapping Scopes. Different scopes may overlap both temporally (e.g., when a process and the core it runs on both have active profiling scopes) and in their profiled event types. If these scopes are unaware of each other, the direct result is that, due to the limited availability of HPCs, events assigned earlier are scheduled, while subsequent assignments are rejected, leading to unfair HPC sharing.

Inflexibility Leads to Misattribution. Without flexible abstractions for defining profiling scopes, users risk mismatches among them; this may lead to misattribution of unrelated event counts. For example, the survey study in [19] examines various applications that aim to characterize program workloads. These often profile the entire task, starting immediately after the process is created, which incidentally captures events from process initialization that are unrelated to the actual profiling scope of interest. To profile events across multiple tasks, dCat [72] pins target tasks to specific cores and then profiles these cores. This approach inevitably includes event counts from context switches on these cores and may also capture events from other tasks running on the same cores.

Misattribution Example. We use Dmon [35] as an illustrative example of the necessity of flexible profiling scopes. DMon [35] is a performance diagnostic tool that detects software data locality issues using memory- and cache-based event profiling. It employs Intel's Top-Down methodology [73] to categorize events into a four-layer hierarchy, monitoring events at subsequent layers based on profiling results from the current layer. As shown in Figure 5, DMon starts by profiling the derived Back-end Bound metric (revealing how often micro-ops are not delivered due to back-end resource shortages) for 100ms. If the program is identified to be back-end bound, DMon then profiles the metrics in the next layer (e.g., execution and memory stall load).

Problematic code segments with data locality issues are often loops that repeatedly perform data operations, e.g., recursive pointer chasing [35]. To identify these operations, DMon must attribute events to the corresponding regions of code. However, DMon modifies events on the fly as it moves through the top-down hierarchy, during which time the target program's execution may move away from the problematic code segments; events from subsequent layers of the hierarchy might therefore be attributed to other segments, with problematic ones remaining undetected. [35] suggests mitigating this by switching to the next layer more swiftly. However, this requires manual tuning of the time window, which varies across different workloads and platforms.

Examination of Mismatched Scope Errors. We extend our preliminary study to quantify the effects of misattribution due to mismatched profiling scopes. We again profile the 541.leela_r benchmark, pinning the workload to a single core and using Linux Perf to count all occurrences of LLC-load-misses in four different profiling scopes: (i) on that core, (ii) by the process, (iii) between explicit polling points instrumented at the beginning and end of program execution, and (iv) using time-triggered sampling. Results are measured across 10 runs and plotted in Figure 4. The instrumented scope produces the most stable results, acting as calipers to filter just those events related to the program.

## 4 Tintin Design Overview

Tintin is a hardware event monitoring infrastructure that aims to solve these limitations, harnessing the following insights.

### 4.1 Key Insights

**I1. Multiplexing errors can be characterized at runtime and reported to applications.**

While multiplexing errors cannot be entirely eliminated, the profiling infrastructure can quantify the expected magnitude of these errors based on observed variance in event rates. These can be reported back to user-space applications alongside the measured counts. These errors serve as indicators of confidence in the measurements and can inform the application's decision-making accordingly.

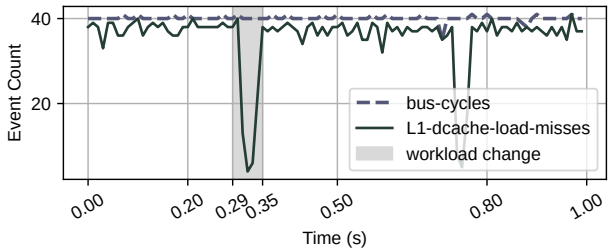



**Figure 6:** Event count changes over time.

**I2. Events exhibit dynamic and distinct errors at runtime, presenting opportunities for scheduling.**

By allocating more HPC time to events with larger variance, the overall error can be mitigated. Figure 6 plots counts of the

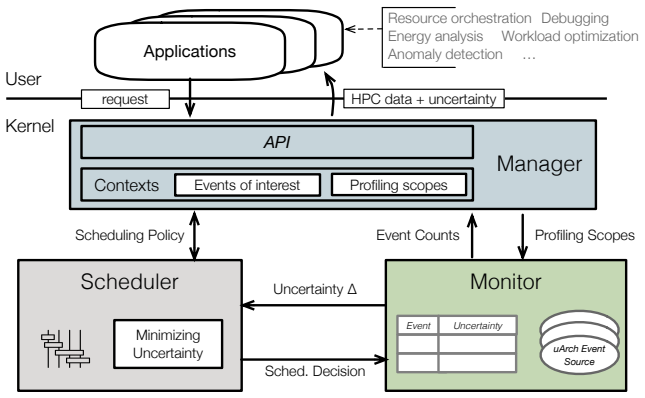



**Figure 7:** Tintin design overview.

bus-cycles and L1-dcache-load-misses events when profiling the 541.leela_r benchmark for 1 second. This illustrates that variance may depend on the specific execution phase. In this period, L1-dcache-load-misses fluctuates significantly, while bus-cycles remains stable. Allocating more HPC time to the former could reduce measurement errors.

It is worth noting that Tintin does *not* require offline analysis to characterize execution phases of the target workload. Instead, it detects sharp changes online, which is made possible because these typically persist for at least tens of milliseconds [59], as shown by the gray area in Figure 6. Sharp changes at the millisecond level tend to be imperceptible to system performance [25, 45, 54, 72]. Nonetheless, Tintin supports programmable intervals of 1ms or less with negligible overhead, allowing it to capture even these transient effects.

**I3. A level of indirection can provide a uniform mechanism to handle the heterogeneity of profiling requirements.**

> *"We can solve any problem by introducing an extra level of indirection."* – often attributed to Butler Lampson, who attributes it to David Wheeler.

The two issues related to profiling scope – inflexibility in definition and conflicts due to overlap – both point to the same solution: a new layer to standardize and manage heterogeneous profiling requirements. With such a layer in the kernel, the system can provide a rich programming API to define diverse profiling scopes without adding management complexity. The heterogeneous requirements are translated into a uniform abstraction within the indirection layer. This allows for collaborative management of all profiling requirements across the system, resolving scope conflicts.

### 4.2 Tintin Structure

Figure 7 depicts the structure of Tintin, which consists of three internal modular components.

**Tintin-Monitor (§5)** measures HPC data and characterizes errors simultaneously at runtime. It tracks event counts and unmonitored time, and based on this data, employs a lightweight calculation method to estimate errors caused by multiplexing. These error estimates are maintained alongside the HPC data and reported to user-space applications upon request.

**Tintin-Scheduler (§6)** 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 in real-time scheduling theory [12, 13]. We present a quasilinear algorithm to solve this optimally for events on multiple HPCs, maintaining low overheads.

**Tintin-Manager (§7)** provides another level of indirection in the kernel. It proposes a new first-class OS abstraction object – the *ePX*– translating heterogeneous profiling requirements from user space into a uniform representation. It then manages these *ePX*s collectively for all applications, avoiding inefficiency caused by conflicting profiling scopes. Additionally, most of Tintin-Manager's APIs have counterparts in existing profiling tools, ensuring minimal effort to migrate existing applications to use Tintin.

## 5 Tintin-Monitor

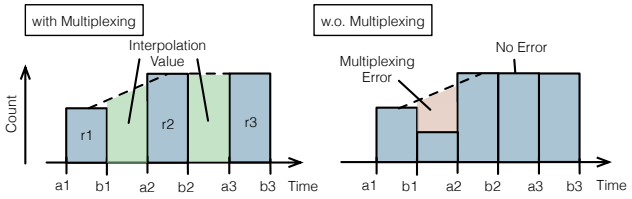



**Figure 8:** Tintin leverages variance as a proxy to model the uncertainty in the context of multiplexing.

Tintin-Monitor interpolates multiplexed event counts and aims to characterize the resulting estimate errors. These errors are then used for event scheduling and are reported to user space to inform the measurement confidence.

**Modeling Runtime Uncertainty.** A challenge in calculating runtime errors in estimated event counts is the lack of ground truth measurements during runtime. To address this, Tintin-Monitor leverages measurement variance as a proxy for error. The underlying idea is that existing interpolation mechanisms [47] implicitly assume linear changes in event rates [22] between subsequent counts. Figure 8 shows an example of interpolation, where the rectangles represent the time slices during which events are measured. If the rate of change in interstitial time slices remains strictly linear, there would be no errors in the reported counts. Prior work [22] therefore links interpolation uncertainty to non-linearity in the unmeasured quantities.

Interpolation. Formally, an event $e_i \sim (x_i, \sigma_i)$ has an estimated count $x_i$ and expected uncertainty $\sigma_i$ due to interpola-

tion over multiplexed observations. As shown in Figure 8, the 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 $\delta_i^j$. Then its average rate $r_i^j$ over this interval is $x_i^j/\delta_i^j$. To interpolate over unmonitored intervals, Tintin-Monitor uses a custom implementation of the trapezoid area method (TAM), which was shown to be the most accurate methodology for online interpolation of event counts [47]. Tintin-Monitor assumes a linear rate of change in the event rate, constructing a 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.

Uncertainty. For an event $e_i$ with interpolated count $x_i$, Tintin-Monitor defines uncertainty as the expected error $\sigma_i$ 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 instantaneous rates cannot be measured at every instant, Tintin-Monitor instead characterizes expected variance $V(\mathbf{r}_i)$ by taking the variance of the sample means, weighted by duration. In many stochastic processes, variance scales linearly with time [26]; 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 $V(x_i, t) = V(\mathbf{r}_i) \cdot t^2$. Using variance as a proxy for uncertainty, the expected error $\sigma_i$ in the estimated event count is $\sqrt{V(x_i, t)}$.

Efficient Incremental Update of Uncertainty. Tintin-Monitor needs to update variance at each monitoring interval. The standard computation of variance requires revisiting all past data points, which is time- and space-consuming. Tintin-Monitor instead implements a weighted version of Welford's method [69] for incremental variance updates. At the end of monitoring intervals $I_i^j$ $(j>1)$, variance is updated as $V_i \leftarrow V_{i-1} \cdot (r_i - \mu_i) \cdot (r - \mu_{i-1})$, where $\mu$ is the time-weighted mean over collected counts.

**Uncertainty Usage Models.** To the best of our knowledge, Tintin is the first hardware event profiling system to report measurement uncertainty to user space. By serving as an indicator of measurement confidence, uncertainty can be explicitly incorporated into application decision-making. While the use case depends on the specific application, our investigation shows that profiling-informed decision making generally follows two broad patterns: rule-based processes [8, 15, 25, 27, 35, 72], which rely on procedural logic derived from human domain expertise; and model-based decision making [20, 37, 54] based on data-driven learning.

For rule-based approaches, conditional logic may be predicated on observed HPC data, e.g., an event count must exceed a certain threshold for an action to be taken. In this context, uncertainty can be incorporated to ensure actions are taken only when the observed data reflect with sufficient confidence that the target condition has been met (i.e., uncertainty is sufficiently low), thereby increasing the reliability of the decision. For example, Caladan [25], a global task scheduler for multicore and multiprocessor systems, attributes memory bandwidth usage to scheduled tasks based on LLC misses and revokes a core from the task with the highest count. However, uncertainty in the profiling data can lead to false identification of high bandwidth usage. By incorporating an uncertainty threshold as an additional condition, the system can avoid mistakenly penalizing tasks.

Model-based approaches can incorporate reported uncertainty values directly into the model as additional input dimensions, allowing the system to account for the confidence of each measurement during prediction. For instance, in our case studies (§8.1), we included error estimates as input features to the predictive models used by Pond to improve the accuracy and robustness of resource orchestration.

**Implementation.** Tintin-Monitor is activated using the Linux kernel's hrtimer. It leverages the existing hardware interfaces in perf_event for reading and configuring PMUs across architectures, intercepting functions to read raw HPC data to perform TAM-based interpolation and incorporate uncertainty characterization. Variance is initialized by a warm-up period where events are scheduled in a round robin fashion, then updated with each HPC read using Welford's method. 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.

## 6 Tintin-Scheduler

Tintin-Scheduler draws from a branch of real-time systems theory that addresses scheduling tasks on limited resources. More specifically, elastic scheduling theory [12, 13] adapts the allocation of processor resources to tasks on overloaded systems to optimize some objective (e.g., result quality). Tintin-Scheduler's goal is similar: it aims to adjust allocations of time for events on limited monitoring resources (HPCs) to minimize overall uncertainty. In this section, we formulate the event scheduling problem and translate it into a form that is semantically equivalent to elastic scheduling.

**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 $\mathcal{S}(t) : \mathbf{C} \rightarrow \{\mathbf{E}, \phi\}$ 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. A function $\mathcal{L}_i(U_i)$ characterizes the loss associated with multiplexing event $e_i$. $\mathcal{L}_i(1)$ is assumed to be 0 (if the event is always counted, there is no multiplexing error), and the function is assumed to be monotonically non-increasing.

**Scheduling Policy.** From §5, the expected error $\sigma_i$ in the

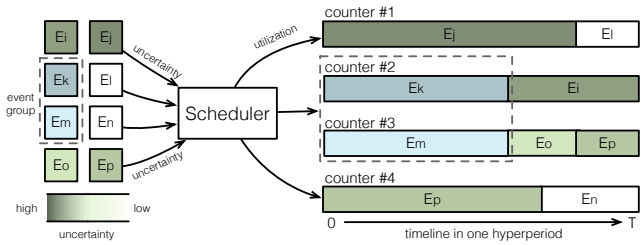



**Figure 9:** Tintin-Scheduler workflow.

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, normalized to the count, as $\sigma_i = \left(\sqrt{V(\mathbf{r}_k)}/x\right) \cdot (1 - U_i)$. To minimize the sum of squared errors, the loss function $\mathcal{L}_i(U_i)$ is formulated as $\sigma_i^2 = \left(V(\mathbf{r}_i)/x_i^2\right) \cdot (1 - U_i)^2$. The scheduling problem is therefore stated as the following constrained optimization problem:

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

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

where $w_i$ is a weight assigned to event $e_i$, denoting its relative importance. By default, $w_i$ is set to 1, but users can adjust it manually to specify the importance of events.

**Online Solver.** The above optimization problem is the same quadratic program as that presented in [14] to represent real-time elastic scheduling [12, 13]. The existing solver is tailored for single-core scenarios, whereas the problem under consideration involves multiple HPCs, which is conceptually equivalent to a multi-core scenario. Tintin-Scheduler adapts the solver by concatenating multiple HPC resources into a single virtual resource and scheduling events sequentially. This prevents any event from being scheduled concurrently on multiple counters, which would yield identical counts and not reflect the effective utilization assignment.

Formally, the "hyperperiod" of the schedule, denoted as $H$, represents the time after which the schedule repeats. For simplicity, it is normalized to $H = 1$. Events and counters are considered in order: event $e_1$ is assigned the interval $[0, U_1]$ on counter $c_1$. Events are placed sequentially on a counter until the total utilization on that counter would exceed 1, whereupon a portion of the event's utilization is assigned to the end of the hyperperiod interval on the current counter, and the remaining utilization is placed at the beginning of the interval on the next open counter. Since $U_i \leq 1$, these intervals do not overlap in time. The schedule construction is illustrated in Figure 9, with the detailed procedure and an optimality sketch provided in Appendix A.

Event Groups. The Linux perf_event subsystem supports user-specified event groups, ensuring that even as events are multiplexed, those in the same group are always scheduled simultaneously. This is useful for identifying correlations among related events. Tintin-Scheduler extends this functionality, treating the group as a single schedulable entity. The total time allocated among all events in a group by the elastic scheduling policy is split evenly between them so all events are allocated to the same time slices across different counters.

**Efficiency and Robustness Optimization.** The solver retains the same runtime complexity as the original in [12], which is quadratic – $O(n^2)$ – in the number of events $n$. This is not negligible given the high scheduling frequency. Briefly, the solver iterates through all events to calculate their utilization. If any event produces a negative utilization, it must be set to zero, indicating that the event will not be scheduled. This requires backtracking and re-iterating through all the events. Consequently, the worst-case complexity is $O(n \times (n-1))$. With the insight that only low-uncertainty events produce negative utilization, the solver can prioritize high-uncertainty events, avoiding re-iterations. Leveraging this invariant, the solver first sorts all events by their uncertainty and then calculates utilization in a single pass. The overall complexity is quasilinear, only dominated by the sorting step, $O(n \log n) + O(n) = O(n \log n)$.

Furthermore, the calculated utilization could be extremely low in some cases. Besides the performance impact caused by closely triggered timer interrupts, such low utilization results in unstable estimations, as measured errors may be amplified when scaled to the full range. As such, Tintin-Scheduler imposes a lower bound $U^{\min}$ on the scheduling quantum (minimum schedulable utilization time slice), which is set to one-tenth of the hyperperiod in our implementation.

**Event Starvation.** A potential issue is that events with low uncertainty might not be scheduled, resulting in a persistently low uncertainty value. However, the uncertainty metric addresses this by increasing an event's uncertainty as the interval since its last scheduling grows, thereby ensuring that all events are eventually allocated on the HPC.

**Implementation.** Tintin-Scheduler is activated when Tintin-Monitor detects the end of a hyperperiod $H$, which is set by default to the CPU task scheduling interval. In addition to elastic scheduling, Tintin-Scheduler can be configured for simple round-robin multiplexing. A simpler priority-driven scheduling policy, *Uncertainty-First*, does not compute utilization, but instead prioritizes the events with the highest uncertainty at each hyperperiod. We note that support for PMU architectures that restrict some events from being assigned to certain HPCs is still limited, as existing algorithms to solve the restricted assignment problem with elastic scheduling are too complex for an efficient in-kernel implementation.

## 7 Tintin-Manager

Internally, Tintin-Manager is composed of two parts: a component for managing profiling scopes and HPC data, and an interface for handling user requests.

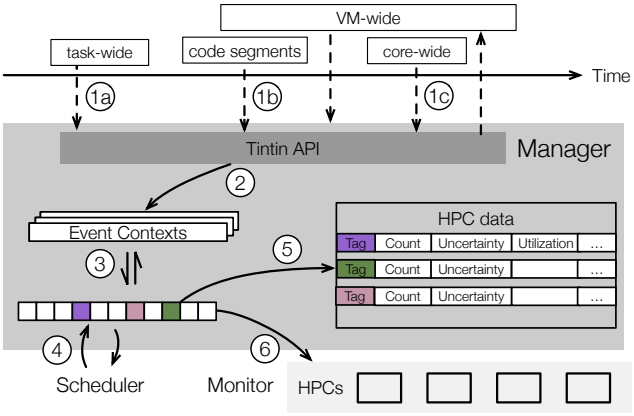



**Figure 10:** Tintin-Manager workflow. (1a) An *ePX* is created for a task, (1b) associated to code segments, or bound to a broader scope such as (1c) core-wide or VM-wide profiling. (2) Upon switching *ePX*s, Tintin-Manager updates outgoing event statistics and (3) fetches events for any incoming *ePX*s. It then (4) invokes Tintin-Scheduler to assign event priorities and time shares. It also (5) updates the HPC data if any events need to be scheduled out. Finally, (6) it places new events on the HPCs.

**Event Profiling Context.** To abstract heterogeneous profiling requirements, Tintin elevates profiling scope via a new first-class kernel object, the Event Profiling Context (*ePX*) abstraction. The *ePX* encapsulates the monitoring of all events associated with a given profiling scope, $ePX \sim \{e_i,..,e_j\}$. An *ePX* can be defined from the perspective of execution instances (e.g., a thread, a process, or all threads on a single core) and from the perspective of code segments (e.g., a function). Multiple contexts, $ePX_m$ and $ePX_n$, may share common events $e_k$, but the HPC data (including counts and uncertainty) are maintained separately, with $e_k^m \rightarrow ePX_m$ and $e_k^n \rightarrow ePX_n$.

Contexts have been successfully used as first-class objects in prior work [40, 42, 51, 61], primarily for flexible privilege separation [17, 40, 51, 61] and time management [42]. Here, the *ePX* concept enriches hardware event profiling as follows.

**Unified Management of *ePX*s.** Figure 10 depicts the workflow and unified management approach of Tintin-Manager.

Flexible Definition of Profiling Scopes. Tintin-Manager translates the heterogeneous profiling requests from program segments and execution instances into a unified *ePX* format. Additionally, it supports *ePX* binding, enabling explicit scoping to arbitrary profiling scopes, such as multiple code segments or threads. The bound *ePX*s form a new *ePX*. This allows *ePX*s to be managed uniformly and collectively.

Correct Attribution of Event Counts. Tintin-Manager places calipers at an *ePX*'s entry and exit points to filter unrelated event counts, activating counting upon entry and deactivating it upon exit. To remain aware of *ePX* switches, Tintin-Manager listens for two types of signals. For *ePX*s associated with code regions, it utilizes instrumentation to insert specific syscalls at relevant code locations, triggering *ePX* switches.[3] For *ePX*s associated with execution instances, Tintin-Manager listens for CPU scheduling events to detect *ePX* switches.

Scope Conflict Resolution. To support overlapping profiling scopes, Tintin-Manager supports the simultaneous activation of multiple *ePX*s. Specifically, when more than one *ePX* is active, it groups all events and schedules them collectively, avoiding contention from overlapping events. An *ePX* may be assigned a weight, e.g., to avoid unfairness in resource allocation when it profiles numerous types of events. Tintin-Scheduler applies these weights as multipliers to the individual event weights within the *ePX*. Counts, variance, and uncertainty for each event are still maintained separately within each individual *ePX*. When an HPC is read, Tintin-Monitor attributes the count to all active *ePX*s monitoring that event, but their interpolated counts and uncertainty are updated individually. Upon *ePX* exit, Tintin-Manager removes the relevant events and reschedules.

On-the-fly Event Reconfiguration. Since *ePX*s are elevated to first-class objects, they can be shared and used by other components to reconfigure the events of interest dynamically within a profiling scope. This capability is crucial for applications that require adaptive profiling [35].

**Programming API and Generality.** In general, existing applications using perf_events may use Tintin instead without modification.

The perf_event Usage Model. Section §2.1 describes the Linux perf_event subsystem. In general, applications use it either via explicit instrumentation of syscalls, or by launching processes with the Perf command-line utility. In keeping with the Linux philosophy that "everything is a file," the driving syscall is *perf_event_open*, which takes several arguments indicating the profiling settings, and returns a file descriptor from which, among other data, HPC counts can be read.

Using Tintin Instead. A Linux kernel patched with Tintin switches between Tintin and the classical perf_event via an interface in procfs. When Tintin is selected, Tintin-Monitor and Tintin-Scheduler become active, but both the syscall API and Perf remain the same, allowing compatibility with existing applications. To access extended functionality, e.g., *ePX* creation and reported uncertainty values, Tintin-Manager provides the extended API outlined in Table 1. An *ePX* can be created for a given scope with a call to *tintin_create_context*. The API allows any *ePX* to be disabled on demand using *tintin_disable_context* and allows on-the-fly event modifica-

[3] A less intrusive alternative [62] is to leverage CPU debugging registers to insert interrupts at specific binary addresses without modifying the target executable; extending this functionality is left for future work.

tion with *tintin_add/remove_event*. Since an *ePX* identifier can be referenced globally, it can also be used to associate different *ePX*s using *tintin_associate_contexts* across different threads or processes, merging their events.

**Table 1:** Tintin-Manager API Functions

| System calls | Descriptions |
|---|---|
| *tintin_create_context(s)* | Create an *ePX* at the specified scope *s* |
| *tintin_enable_context(c)* | Begin monitoring of the given *ePX c* |
| *tintin_disable_context(c)* | End monitoring of the given *ePX c* |
| *tintin_add_event(e, c)* | Begin monitoring event *e* in given *ePX c* |
| *tintin_remove_event(e, c)* | Stop monitoring event *e* in given *ePX c* |
| *tintin_set_event_weight(e, w)* | Set or modify event *e* scheduling weight *w* |
| *tintin_associate_contexts($c_1$, $c_2$)* | Aggregate the counts of two *ePX*s |
| *tintin_read_with_uncertainty(e)* | Read both event count and uncertainty |

**Implementation.** All APIs are implemented as system calls, wrapped in C. To reduce the effort required for instrumentation, we implemented a set of LLVM [36] compiler passes that automatically instrument the system calls during compilation. By utilizing different levels of passes (e.g., function-level, basic block-level), users can achieve various levels of granularity by adding a compilation flag. The current Tintin implementation defines profiling scopes in source code, but binary instrumentation [62] could extend this to binaries.

# 8 Evaluation

This section seeks to answer the following questions:

- How can Tintin be used in real-world applications? (§8.1)
- How effective is Tintin in improving event measurement accuracy? (§8.2)
- What is Tintin's overhead? (§8.3)
- How scalable is Tintin? (§8.4)

**Experimental Setup.** All experiments were conducted on a server with two Intel Xeon Gold 6130 (Skylake) CPUs and 32GB of RAM. Each CPU has 16 physical cores running at 2.10GHz with Hyper-Threading disabled. By default, the hyperperiod (scheduling cycle) in Tintin-Scheduler is set to 4 ms (250 Hz), aligning with the default scheduling interval of the Linux perf_event subsystem [65] on the tested machine. The scheduling quantum is set to $^1/_{10}$ the hyperperiod (0.4 ms). As part of our scalability analysis in §8.4, we explore the effects of these parameters with different values.

Ground Truth. Due to the non-determinism of modern CPUs, obtaining ground truth data outside of simulation is nearly impossible. Similarly to CounterMiner [41] and BayesPerf [2], we measure ground truth by pinning an event to an HPC, then reading its value on completion of a benchmark.

## 8.1 Case Studies

We use three case studies to demonstrate the effectiveness and usability of Tintin.

### 8.1.1 Dynamic Resource Provisioning with Pond

**Target Application.** Pond [37] is a resource orchestration subsystem developed by the Microsoft Azure team. In cloud environments, memory pooling may improve DRAM utilization and reduce cost, but using pooled instead of local memory might substantially increase the latency of some virtual machine workloads. To allocate the limited local memory appropriately, Pond predicts whether a VM workload's latency will be sensitive to the use of pooled memory.

**Evaluation Methodology.** Pond is a proprietary, closed-source system, so its authors do not provide the data or models used in [37]. Instead, we use Pond's open-source emulation layer [66] to reproduce its latency insensitive model [37, Figure 12] according to the paper's description. The predictive model is a random forest regressor implemented using Scikit-learn [52]. The model's inputs consist of 20 hardware events, sampled at 100 ms intervals via Intel's EMON tool [33]. To generate the training data, Pond runs the workload entirely on a local NUMA node, then again with memory from a remote node, reporting the execution time increase. We follow the experimental setup in [37], generating data using 12 workloads from SPEC CPU®2017 [11] and 30 from the GAP Benchmark Suite [3]. We randomly select 30 workloads for training and 12 for testing.

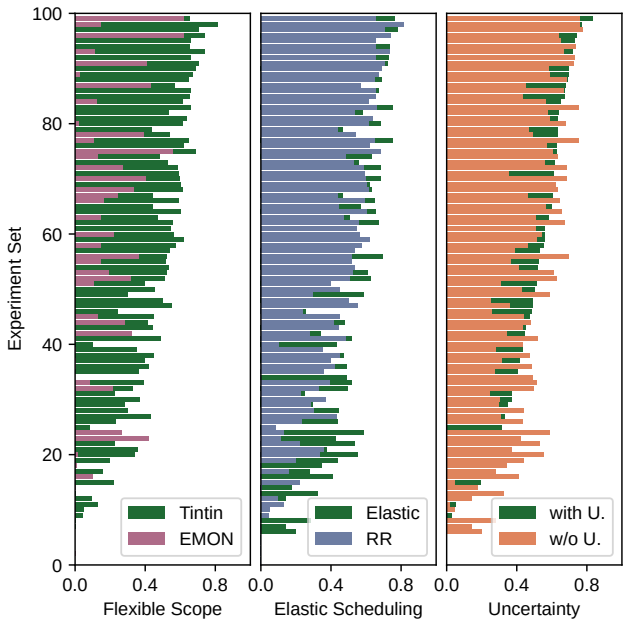



**Figure 11:** Results on Pond [37]. Each bar represents the score of the latency sensitivity prediction.

**Results.** A comparison of model prediction accuracy is shown in Figure 11; values closer to 1 indicate better performance.

Profiling Scopes. As Intel EMON only supports CPU-level event profiling, Pond's emulation layer pins VM threads to specified cores, then attributes all event counts from those cores to the VM. We modify Pond to instead use Tintin to associate an *ePX* with all VM threads. In the plot on the left, Tintin outperforms the baseline, measured by EMON [33], in 95 out of 100 experiments. On average, Pond's accuracy with Tintin is 0.51 higher than the baseline. This improvement is primarily due to Tintin's ability to attribute events directly via correctly defined profiling scopes.

Elastic Scheduling. To demonstrate the advantages of Tintin-Scheduler's elastic scheduling approach in reducing total uncertainty, we compare it to a round-robin scheduling strategy across another 100 experiment sets. The results are shown in the middle plot of Figure 11. Elastic scheduling outperformed the round-robin strategy in 64 out of 100 experiments, improving model prediction scores by an average of 0.15.

With Uncertainty. To demonstrate how user-space applications might use the uncertainty reported by Tintin, we modify our implementation of Pond's latency prediction model to include the uncertainty values reported by Tintin-Monitor as inputs. Results are illustrated in the right plot in Figure 11. With this additional information, scores are higher in 55 out of 100 cases compared to the model without uncertainty, and the average score increases by more than 0.02. The minor improvement is attributed to the fact that, in these tests, Tintin's targeted scope and scheduling policy succeed in minimizing uncertainty, so the small error values given to the model provide only incremental improvements.

Resolving Scope Conflicts. To evaluate Tintin's ability to resolve profiling scope conflicts, we modify the workloads managed by Pond to also require their own event monitoring, leading to potential conflicts with Pond's profiling scope. We instrument each workload to profile 16 events and then measure both the counting accuracy of the target workload and the prediction accuracy of Pond's latency model. The counting accuracy is measured by pinning a specific event (e.g., cache misses) to a dedicated hardware counter, which serves as the ground truth for comparison.

By default, Pond profiles events at the CPU core scope, and the Linux perf_event subsystem prioritizes these events over any other event bound to tasks. As a result, if other workloads attempt to profile events on the same core, their events may never be scheduled, leading to starvation. Tintin avoids this issue by collectively scheduling all events. Under this collective scheduling approach, measurement accuracy does not degrade significantly compared to separate runs that just profile each scope in isolation. From the workloads' perspective, the counting error increases slightly from 3.11% to 3.56%. From Pond's perspective, the prediction accuracy remains comparable, with an average score decrease of only 0.01 across 100 test cases. Moreover, the overhead of Tintin remains below 2.4% in both conflicting and non-conflicting scenarios, demonstrating that resolving conflicting scopes does not introduce significant overhead.

| Cores | Metrics | | % Slots |
|---|---|---|---|
| C0 | FE | Frontend_Bound | 39.0 <== |
| C0 | BE | Backend_Bound | 33.3 |
| C0 | RET | Retiring | 19.5 |
| C0 | FE | Frontend_Bound.Fetch_Latency | 28.7 |
| C0 | BE/Mem | Backend_Bound.Memory_Bound | 23.8 |
| C0 | RET | Retiring.Heavy_Operations | 12.5 |

(a) Results by Linux perf_event subsystem.

| - | Metrics | | % Slots |
|---|---|---|---|
| S1 | BE | Backend_Bound | 96.8 <== |
| S1 | BE | Backend_Bound.Memory_Bound | 90.9 |
| S1 | FE | Fetch_Latency.MS_Switches % Clocks_est | 9.8 |
| S1 | BE/Mem | Backend_Bound.Memory_Bound | 80.5 <== |

(b) Results by Tintin.

**Figure 12:** Results on DMon.

### 8.1.2 Data Locality Problem Detection with DMon

A description of the target application, DMon [35], and its limitations has already been introduced in §3.

**Evaluation Methodology.** We use DMon's artifact [24] as a baseline for comparison and target the workload from its repository, which contains one data locality issue. Rather than using DMon's built-in time window, Tintin binds *ePX*s at the individual loop level via syscalls automatically instrumented at the beginning and end of each function in a compiler pass. This enables profiling different layers in Intel's Top-Down event hierarchy [73] every time a function is entered, eliminating the need to switch layers during function or loop execution. Since some functions may be invoked only once in a given run, a user may execute the target workload multiple times to ensure each layer is adequately profiled. By providing more targeted profiling scopes via its *ePX*s, Tintin ensures that events are correctly filtered for each function so as to correctly attribute the locality issue. In our case study, results are evaluated based on the accuracy with which they identify the code region experiencing Back-end Bound behavior.

**Results.** Visualization of DMon's output is shown in Figure 12. Among 10 experiment runs, DMon often results in an ambiguous mixture of Front-end and Back-end Bound instances, failing to identify data locality issues when using Linux perf_event. In Figure 12(a), the results indicate that most of the time is spent on the frontend. However, the program is actually Back-end Bound. Since DMon aggregates the counts over the program's execution, the events originating from Back-end Bound code segments are overshadowed by other normal code. In contrast, Tintin offers more precise profiling via loop-level scope. Notably, in Figure 12(b), the target function is identified as spending 96.8% of its time on the backend, directly indicating a data locality issue. Among

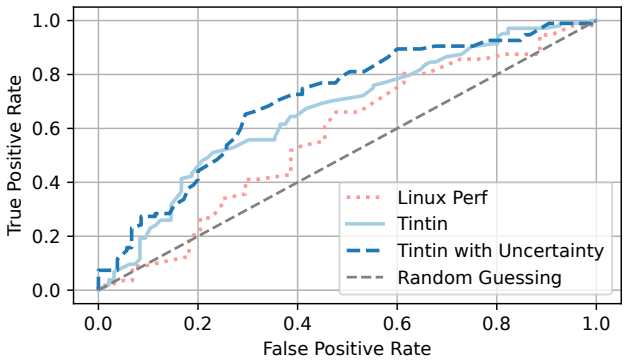



**Figure 13:** Accuracy of the random forest classifier in detecting the Linux rootkit.

10 repeated experiment runs, DMon failed in 9 out of 10 runs with perf_event. However, with Tintin, DMon reliably detects Back-end Bound issues; the reported backend time for the target consistently exceeds 91.1%.

#### 8.1.3 Intrusion Detection System

HPC data has also been used for security purposes in the last two decades. Previous works [20, 34] leverage the measured HPC values to classify an unknown program as either benign or malicious. This case study investigates whether Tintin enhances the efficacy of intrusion detection.

**Evaluation Methodology.** We adopt the experimental setup from the seminal work of Demme et al. on detecting Linux rootkits with hardware event data [20]. As our malware, we use the open-source Linux rootkit Diamorphine [43], which is implemented as a loadable kernel module. To activate the rootkit's functionality, we execute common Linux commands, including ps, ls, whoami, printenv, and pwd, while monitoring their behavior. The rootkit is configured to hide processes, escalate user privileges to root, and conceal files. Since Demme et al. do not specify the hardware events used in their experiments, we select the 10 commonly used events predefined under PERF_TYPE_HARDWARE as described in [44].

We conducted three sets of experiments. The first set evaluates perf_event at the granularity of the target victim tasks. The second set evaluates Tintin's elastic scheduling. In this setup, we replaced the event scheduling logic with Tintin's scheduler but kept the rest unchanged. The third set extends the previous configuration by collecting uncertainty information at each measurement point. For each set, we collect 800 samples for training and 200 samples for testing; these are evenly split between execution scenarios with and without the rootkit. For the detection model, we adopt a random forest classifier, which is also used in [20, 77].

**Results.** We observed that using HPC data to detect malware within a single, fixed target task (e.g., ls) yields excellent classification results. However, this approach requires constructing a separate HPC-based malware signature for each individual task, which is impractical and limits generalizability. In our more realistic experimental setting, where the five target tasks vary, the detection performance is less compelling. Figure 13 presents the ROC curves for Linux perf_event, Tintin without uncertainty information, and Tintin with uncertainty information. The AUC (Area Under the Curve) for perf_event is 0.57. Tintin improves the AUC to 0.66 through its event multiplexing-aware scheduling. When uncertainty information is incorporated, the AUC further increases slightly to 0.70. Although the improvement is modest, these results indicate that Tintin provides benefits in this scenario.

Further Discussion. There are inherent limitations of relying solely on HPC data for intrusion detection because malicious software behavior may not always manifest in the specific microarchitectural events captured by HPCs [77]. Although our experiments showcase that Tintin raises the bar for attackers, sophisticated attacks can still evade detection.

### 8.2 Effectiveness of Tintin

For a more comprehensive evaluation, we use the SPEC CPU®2017 [11] and PARSEC 3.0 [9] benchmark suites to assess the accuracy of the event counts collected online by Tintin, comparing it to Linux perf_event and CounterMiner [41]. While CounterMiner is primarily intended for offline predictive modeling and analysis of events, it does attempt to reduce event count errors by smoothing outliers. For a reasonable runtime comparison, we implement its data cleaning phase in Linux perf_event. Another related infrastructure, BayesPerf, is not included in the comparison because it requires algebraic relations between events, which are absent from most event primitives in practice.

To evaluate the effectiveness of elastic scheduling, we compare it to Tintin-Scheduler's "Uncertainty-First" policy. In this configuration, events with higher uncertainties are prioritized, and the scheduled event occupies the HPC until the next hyperperiod. We select the 24 default predefined events in Linux perf_event to profile simultaneously [44]. To obtain ground truth, we pin one event to an HPC in each run. We repeat the experiments for the first 5 pre-defined events in PERF_TYPE_HARDWARE in [44]. The average errors are reported in Figure 14.

Event counts obtained with Linux perf_event were, on average, 9.01% off from the ground truth, with a maximum error of 53.27%. CounterMiner performed similarly with an average error of 8.80% and a maximum of 56.21%. In comparison, Tintin's errors remained well under 5% in most cases. Furthermore, while "Uncertainty-First" also reduces errors, it does not perform as well as elastic scheduling, achieving an average error of 6.51%, compared to 2.91%. This is due to two factors: first, its scheduling is heuristic rather than optimal; second, it cannot allocate finer-grained timeslices for

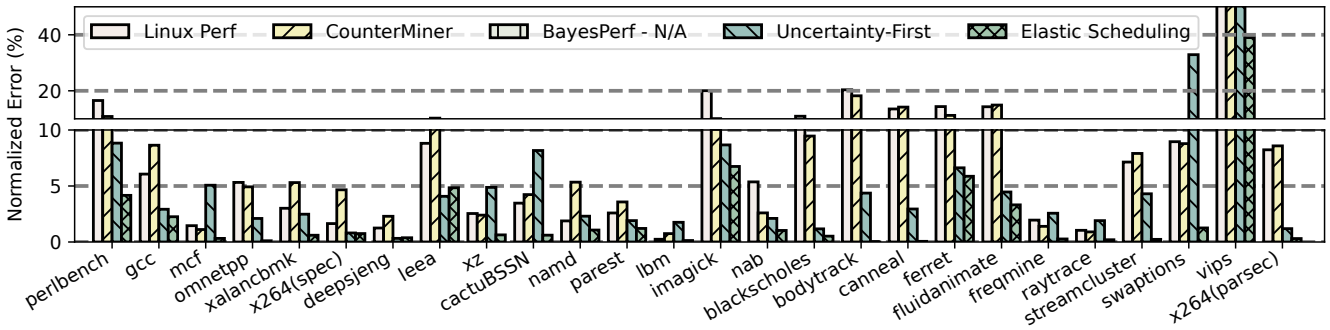



**Figure 14:** Measurement errors.

events, causing each event to occupy the HPC for an entire hyperperiod, which may lead to event starvation.

### 8.3 Runtime Overhead of Tintin

We also evaluate the runtime overhead of Tintin in comparison to Linux perf_event and BayesPerf [2]. We exclude CounterMiner [41] from these experiments, as it is a purely offline analysis tool. We instrument a CPU implementation of BayesPerf by modifying the Linux perf_event timer callback to run BayesPerf's Expectation Propagation algorithm [2, Alg. 1]. Figure 15 presents execution time means and standard deviations across 10 runs of each considered benchmark (normalized to the execution time without profiling). Without the dedicated hardware accelerator in [2], BayesPerf incurs excessive overhead, with up to 31.3% slowdown in the profiled benchmarks. This limits portability, as it cannot achieve acceptable runtime performance on standard hardware.

On the other hand, Tintin exhibits an average overhead of only 2.4%, only slightly higher than perf_event's 1.9%. This difference was found to be statistically insignificant. In the worst-case scenarios, we observed Tintin's overhead to reach up to 7.6% while perf_event's could be up to 12.7%. In some scenarios, Tintin achieves even better execution time performance than Linux perf_event due to its elastic scheduling method, which can trigger fewer interrupts.

### 8.4 Scalability Analysis

There are three factors impacting the accuracy and overhead of Tintin: scheduling hyperperiod, scheduling quantum, and number of monitored events. In this section, we examine how Tintin's performance scales with these factors. While the number of scopes might also impact scalability, these effects are reflected by the number of events.

**Hyperperiod.** We target the 505.mcf workload from the SPEC 2017 benchmark, varying the hyperperiod from 15 ms to 1 ms. The resulting changes in accuracy and overhead are recorded in Figure 16. We observe in Figure 16(a) that the accuracy improves with finer-grained hyperperiods, achieving an error rate of only 0.2% at 1 ms. In contrast, the error rate increases to 1.01% at a 15 ms hyperperiod. Meanwhile, perf_event exhibits significantly higher error rates across all hyperperiod settings, reaching up to 7.8% at 15 ms. Nonetheless, Figure 16(b) shows that while overhead increases with smaller intervals, it remains below 5%.

**Scheduling Quantum.** Since Linux perf_event schedules events at the hyperperiod, it does not support tuning the scheduling quantum. Consequently, we exclude experiments involving it. The accuracy and overhead results are presented in Figure 17(a). Accuracy stabilizes at 0.5 ms, with an error rate of 0.6%. Reducing the scheduling quantum further does not significantly enhance accuracy but noticeably increases overhead. For instance, at a quantum of 0.05 ms, the overhead surpasses 7.5%.

**Number of Events.** We evaluate the scalability of elastic scheduling in handling varying numbers of events, as shown in Figure 17(b). The error remains below 5.4% with 512 event types, but when the number reaches 1024, Tintin can render the kernel unresponsive. This issue occurs when TintinScheduler is triggered during a CPU context switch. To generate a new schedule, it sorts all events by uncertainty, which takes too long with 1024 events. As a result, it may not finish before the next jiffy, causing the CPU scheduler to miss the deadline and be deemed unresponsive. In practice, the number of profiled events is usually less than 256, so Tintin remains usable. In contrast, the errors in perf_event increase more significantly with the number of events, exceeding 20% when monitoring 256 events.

## 9 Related Work

**Measurement Error.** Most existing work that avoids multiplexing does so by running a target application multiple times, each time monitoring a different subset of events. Results are aligned into a single trace [31, 41, 48] using techniques such as dynamic time warping [7] or uniform scaling [47]. However, these offline techniques are unsuitable for online monitoring, which is required in many applications such as resource orchestration [37, 54] and performance interference

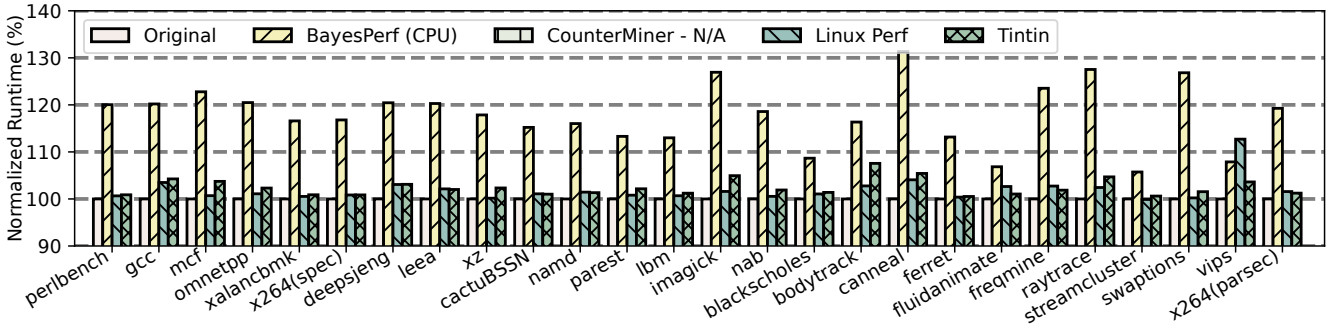
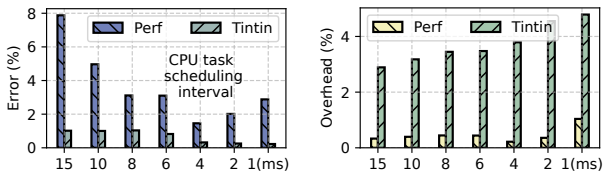
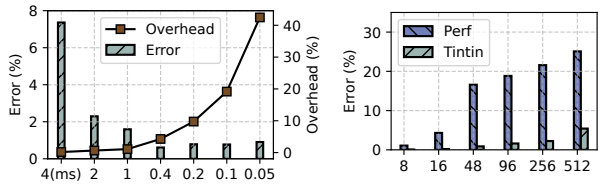



**Figure 15:** Runtime overhead.




(a) Accuracy. (b) Overhead.

**Figure 16:** Scalability analysis of event scheduling hyperperiod.




(a) Scheduling quantum. (b) Number of events.

**Figure 17:** Scalability analysis scheduling quantum and event count.

mitigation [25, 54, 72]. Additionally, merging runs may fail to identify some performance anomalies, as performance can vary widely across runs, with anomalies only appearing in some.

BayesPerf [2] schedules events online, constructing event groups to guarantee statistical dependencies between events in adjacent groups, thus inferring the counts of unscheduled events. However, it has three main drawbacks. First, it depends on algebraic relationships among events, which are unavailable for most event primitives on platforms like Intel [18], IBM [30], and ARM. Second, it incurs prohibitive overhead, necessitating customized hardware for acceleration. Third, its scheduling algorithm is designed offline, preventing adaptation for dynamic workloads. With elastic scheduling, Tintin overcomes these limitations and achieves accuracy improvements comparable to those reported in prior work [2].

**Attribution Error.** Existing work on attribution falls into two main categories. The first focuses on precise event filtering for target profiling scopes, achieved either through efficient instrumentation during compilation [1, 62] or by using techniques for post-mortem analysis [64]. The second addresses conflicts within overlapping profiling scopes. For example, Metis [71] virtualizes HPCs using time division, allowing multiple users in different virtual machines to monitor various CPU events simultaneously. Tintin unifies these functionalities across various use cases through the *ePX* and system support.

Another line of research focuses on architectural misattribution caused by skid effects, which is discussed in §2.2. Addressing this issue often requires hardware redesign [28, 29]. Software-based methods, such as padding with dummy instructions, are also an alternative [74] but incur significant overhead. These are out of scope for this paper.

## 10 Conclusion

This paper introduces Tintin, a hardware event profiling infrastructure designed to address event multiplexing errors and the inflexibility of profiling scope definitions. Designed with a three-component structure, each component of Tintin individually mitigates these tensions to some degree, while also collaboratively working together for enhanced overall performance. The evaluation results showcase that Tintin provides flexibility in profiling and improves profiling accuracy with minimal overhead.

## Acknowledgment

We thank our shepherd, Pedro Fonseca, and the anonymous reviewers for their valuable feedback. We also thank Tobias Pristupin and Lien Zhu for their contributions to the implementation. 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] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[2] Subho S. Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar 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*, ASPLOS '21, page 832–844, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[4] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. Profiling dataflow systems on multiple abstraction levels. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 474–489, 2021.

[5] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 177–191, USA, 2010. USENIX Association.

[6] Emery D Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating python performance issues with {SCALENE}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 51–64, 2023.

[7] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, AAAIWS'94, page 359–370, Seattle, WA, 1994. AAAI Press.

[8] Sapan Bhatia, Abhishek Kumar, Marc E. Fiuczynski, and Larry Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 103–116, USA, 2008. USENIX Association.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[10] 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.*, 14(3):189–204, aug 2000.

[11] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

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

[13] Giorgio C. Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, March 2002.

[14] Thidapat Chantem, Xiaobo Sharon Hu, and M. D. Lemmon. Generalized elastic scheduling. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, page 236–245, USA, 2006. IEEE Computer Society.

[15] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 12–23, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for fdo compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, page 42–52, New York, NY, USA, 2010. Association for Computing Machinery.

[17] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016.

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

[19] Sanjeev Das. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Symposium on Security and Privacy (SP)*, pages 20–38, Oakland,CA,USA, 2019. IEEE, IEEE.

[20] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 559–570, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. Reliable and efficient performance monitoring in linux. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, Salt Lake City, Utah, 2016. IEEE Press.

[22] Manfred Drosg. *Dealing with uncertainties*. Springer, 2007.

[23] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[24] EfesLab. Dmon-ae. https://github.com/efeslab/DMon-AE, 2023. Accessed: 2023-11-26.

[25] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[26] Crispin W. Gardiner. *Stochastic Methods: A Handbook for the Natural and Social Sciences*. Springer, 4th edition, 2009.

[27] Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. Dna: Dynamic resource allocation for soft real-time multicore systems. In *27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 196–209, Nashville, TN, USA, 2021. IEEE.

[28] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. Tip: Time-proportional instruction profiling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 15–27, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. Tea: Time-proportional event analysis. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Brian Hall, Peter Bergner, Alon Shalev Housfater, Madhusudanan Kandasamy, Tulio Magno, Alex Mericas, Steve Munroe, Mauricio Oliveira, Bill Schmidt, Will Schmidt, et al. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2017.

[31] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 281–296, New York, NY, USA, 2005. Association for Computing Machinery.

[32] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[33] 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.

[34] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[35] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris 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)*, pages 163–181, Virtual, July 2021. USENIX Association.

[36] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[37] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo 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*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Arm 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.

[39] Arm Limited. Arm cortex-a78 technical reference manual. https://developer.arm.com/documentation/101430/0102/Debug-descriptions/Performance-Monitoring-Unit/PMU-events. Accessed 2023-11-16.

[40] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. {Light-Weight} contexts: An {OS} abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, 2016.

[41] Yirong Lv, Bin Sun, Qinyi Luo, Jing Wang, Zhibin Yu, and Xuehai Qian. Counterminer: Mining big performance data from hardware counters. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 613–626, Fukuoka, Japan, 2018. IEEE Press.

[42] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

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

[44] man7.org Linux Man-pages project. *perf_event_open(2) - Linux Manual Page*. Accessed: 2024-11-02.

[45] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 409–425, USA, 2018. USENIX Association.

[46] John M May. Mpx: Software for multiplexing hardware performance counters in multithreaded programs. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 8–pp. IEEE, 2001.

[47] Todd Mytkowicz, Peter F. Sweeney, Matthias Hauswirth, and Amer Diwan. Time interpolation: So many metrics, so few registers. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, page 286–300, USA, 2007. IEEE Computer Society.

[48] Richard Neill, Andi Drebes, and Antoniu Pop. Fuse: Accurate multiplexing of hardware performance counters across executions. *ACM Trans. Archit. Code Optim.*, 14(4), dec 2017.

[49] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 377–389, USA, 2017. USENIX Association.

[50] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 97–108, New York, NY, USA, 2009. Association for Computing Machinery.

[51] Gabriel Parmer. The case for thread migration: Predictable ipc in a customizable and reliable os. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2010)*, page 91, 2010.

[52] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[53] perf event. https://www.brendangregg.com/linuxperf.html. Accessed: 2022-11-13.

[54] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar 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*, OSDI'20, USA, 2020. USENIX Association.

[55] James Reinders. *VTune performance analyzer essentials*, volume 9. Intel Press, Santa Clara, 2005.

[56] Jerome H. Saltzer. Protection and control of information sharing in multics. In *Proceedings of the Fourth ACM Symposium on Operating System Principles*, SOSP '73, page 119, New York, NY, USA, 1973. Association for Computing Machinery.

[57] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[58] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. *ACM SIGARCH Computer Architecture News*, 41(1):65–76, 2013.

[59] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGPLAN Not.*, 37(10):45–57, oct 2002.

[60] Karan Singh, Major Bhadauria, and Sally A McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[61] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222, 2010.

[62] Pengfei Su, Shuyin Jiao, Milind Chabbi, and Xu Liu. Pinpointing performance inefficiencies via lightweight variance profiling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[63] Marion Sudvarg, Chris Gill, and Sanjoy Baruah. Linear-time admission control for elastic scheduling. *Real-Time Systems*, 57(4):485–490, Oct 2021.

[64] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 441–452, New York, NY, USA, 2009. Association for Computing Machinery.

[65] Linus Torvalds and Contributors. Linux kernel perf events directory. https://github.com/torvalds/linux/tree/master/kernel/events, 2024. Accessed: 2024-04-27.

[66] Vtess. Pond emulation. https://github.com/vtess/Pond. Accessed: December 5, 2023.

[67] Vincent M Weaver and Sally A McKee. Can hardware performance counters be trusted? In *2008 IEEE International Symposium on Workload Characterization*, pages 141–150, Seattle, USA, 2008. IEEE.

[68] Vincent M Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224, Austin, TX, 2013. IEEE.

[69] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[70] Perf Wiki. Tutorial: Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial, 2023. Accessed on: 2023-11-16.

[71] Xia Xie, Haiou Jiang, Hai Jin, Wenzhi Cao, Pingpeng Yuan, and Laurence Tianruo Yang. Metis: a profiling toolkit based on the virtualization of hardware performance counters. *Human-centric Computing and Information Sciences*, 2:1–15, 2012.

[72] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. Dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

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

[74] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 98–105, New York, NY, USA, 2020. Association for Computing Machinery.

[75] Gerd Zellweger, Denny Lin, and Timothy Roscoe. So many performance events, so little time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[76] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. Happy: Hyperthread-aware power profiling dynamically. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 211–218, USA, 2014. USENIX Association.

[77] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 457–468, New York, NY, USA, 2018. Association for Computing Machinery.

**Algorithm 1:** Elastic_Scheduling

**Input:** Lists $\{U_i^{\min}\}, \{U_i^{\max}\}, \{E_i\}$ sorted in non-decreasing order of $\frac{U_i^{\max}-U_i^{\min}}{E_i}$ and a desired utilization $U_D$

**Output:** The list of assigned $U_i$ values

$U_{\text{SUM}} \leftarrow 0$; $E_{\text{SUM}} \leftarrow 0$; $\Delta \leftarrow 0$
**forall** $\tau_i \in \Gamma$ **do**
  $U_{\text{SUM}} = U_{\text{SUM}} + U_i^{\max}$
  $E_{\text{SUM}} = E_{\text{SUM}} + E_i$
**end**
**forall** $\tau_i \in \Gamma$ **do**
  **if** $\left(U_i^{\max} - \frac{U_{\text{SUM}}-(U_D-\Delta)}{E_{\text{SUM}}} \times E_i \le U_i^{\min}\right)$ **then**
    $U_i \leftarrow U_i^{\min}$
    $\Delta \leftarrow \Delta + U_i^{\min}$
    **if** $(\Delta > U_D)$ **then return** INFEASIBLE
    $U_{\text{SUM}} \leftarrow U_{\text{SUM}} - U_i^{\max}$
    $E_{\text{SUM}} = E_{\text{SUM}} - E_i$
  **else**
    $U_i \leftarrow U_i^{\max} - \frac{U_{\text{SUM}}-(U_D-\Delta)}{E_{\text{SUM}}} \times E_i$
  **end**
**end**
**return** FEASIBLE

# A Solving the Scheduling Problem

The scheduling problem in Section 6 assigns a utilization to each event of interest to minimize total expected normalized error, with additional weights assignable by the user. This is expressed as the following constrained optimization problem:

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

$$\text{s.t.} \quad \sum_{i=1}^{n} U_i \le m \tag{2b}$$

$$\forall_i \quad U^{\min} \le U_i \le 1 \tag{2c}$$

where $U_i$ is the utilization assigned to event $e_i$, $w_i$ is a user-defined weight, $V(\mathbf{r}_i)$ is the observed variance, and $x_i$ is the estimated count. $m$ denotes the number of available HPCs.

This reduces in polynomial time to the formulation in [14] of elastic scheduling as a constrained optimization problem,

$$\min_{U_i} \quad \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{\max} - U_i)^2 \tag{3a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} U_i \le U_D \tag{3b}$$

$$\forall_i, \ U_i^{\min} \le U_i \le U_i^{\max}, \tag{3c}$$

by setting $U_i^{\max} = 1$, $E_i = \frac{x_i^2}{w_i V(\mathbf{r}_i)}$, and $U_D = m$.

## A.1 Optimal Solution Algorithm

From [63, Algorithm 1], this can be solved in quasilinear time by the procedure listed in Algorithm 1.

## A.2 Constructing a Schedule

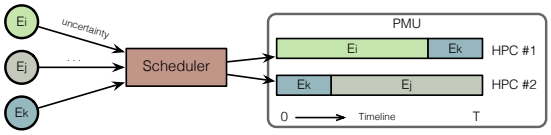



**Figure 18:** Constructing a schedule of 3 events on 2 HPCs from assigned utilizations.

Although the adopted elastic scheduling problem targets uniprocessor scheduling, we now show that our problem admits a valid schedule for events on *multiple* HPCs. We denote by $H$ the hyperperiod of the schedule, after which it repeats itself; for simplicity of explication we normalize to $H = 1$. Events and counters are considered in order: event $e_1$ is assigned the interval $[0, U_1]$ on counter $c_1$. Events are placed sequentially on a counter until total counter utilization would exceed 1, whereupon an event's utilization is split between the end of the hyperperiod interval on the current counter and the beginning of the interval on the next open counter. This continues until all events have been scheduled. This method guarantees that no event is scheduled concurrently on two counters at the same time: if an event is assigned to two counters, the assignments are respectively at the end and beginning of the hyperperiod. Since $U_i \le 1$, these intervals do not overlap, as illustrated in Figure 18.