NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2026-25

June, 2026

Adaptive Profiling via ML-Driven Hardware Counter Orchestration

Liam Tucker Raab Strand

Abstract

Modern CPUs expose hundreds of hardware performance events but only a handful of physical counters with which to observe them. Profilers reconcile this gap by time-division multiplexing, which trades fidelity for breadth and treats the resulting noise as unavoidable. This thesis reframes the problem as one of state estimation and observation scheduling, and presents SACCADE, a profiling framework that decouples the event source, scheduler, estimator, and sink into independently composable layers. This decoupling makes the design space tractable and admits direct evaluation of any scheduler/estimator pair against ground truth, either on live hardware or in simulation over recorded traces. We instantiate the framework with a family of estimators, including a Kalman Filter formulation of virtual counter reconstruction, and a collection of LLM-driven schedulers that translate natural-language guidance into counter selection decisions. Across NAS Parallel Benchmark and SPEC CPU2017 workloads, we find that closed-loop scheduling can reduce reconstruction error, but that the best scheduler and estimator are coupled and no single configuration consistently dominates passive multiplexing. Live, the stop-the-world cost of swapping counters collapses SACCADE's temporal resolution below kernel-native `perf stat`; in simulation, where that cost vanishes, SACCADE already outperforms `perf stat` on low-noise workloads.

Keywords

hardware performance counters, PMU multiplexing, state estimation, Kalman filtering, LLM-driven scheduling

NORTHWESTERN UNIVERSITY

Adaptive Profiling via ML-Driven Hardware Counter Orchestration

A THESIS

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Computer Science

By

Liam Tucker Raab Strand

EVANSTON, ILLINOIS

June 2026

ABSTRACT

Adaptive Profiling via ML-Driven Hardware Counter Orchestration

Liam Tucker Raab Strand

Modern CPUs expose hundreds of hardware performance events but only a handful of physical counters with which to observe them. Profilers reconcile this gap by time-division multiplexing, which trades fidelity for breadth and treats the resulting noise as unavoidable. This thesis reframes the problem as one of state estimation and observation scheduling, and presents SACCADE, a profiling framework that decouples the event source, scheduler, estimator, and sink into independently composable layers. This decoupling makes the design space tractable and admits direct evaluation of any scheduler/estimator pair against ground truth, either on live hardware or in simulation over recorded traces. We instantiate the framework with a family of estimators, including a Kalman Filter formulation of virtual counter reconstruction, and a collection of LLM-driven schedulers that translate natural-language guidance into counter selection decisions. Across NAS Parallel Benchmark and SPEC CPU2017 workloads, we find that closed-loop scheduling can reduce reconstruction error, but that the best scheduler and estimator are coupled and no single configuration consistently dominates passive multiplexing. Live, the stop-the-world cost of swapping counters collapses SACCADE's temporal resolution below kernel-native `perf stat`; in simulation, where that cost vanishes, SACCADE already outperforms `perf stat` on low-noise workloads.

Acknowledgements

This work would not have been possible without tremendous support. While I could never hope to list every person who lent a hand (or an ear), I shall humbly make my best effort.

My advisor, Peter Dinda, has given thoughtful guidance throughout my time at Northwestern. He always made time to answer any question, big or small, in ways both insightful and surprising. His ability to explain complicated concepts simply, pointed critique of undercooked ideas, and encouragement to fearlessly try new theories has made me a better programmer, engineer, and researcher.

My colleague in the Prescience Lab, Karl Hallsby, has been an invaluable resource for all things hardware. Moreover, they have been a fantastic debugging partner, and their feedback vastly improved this manuscript.

Trevor Sullivan, Avi Shein, and Jacqueline Egidio have supported me outside of the lab in more ways than I can count. From late-night card games to reminding me to eat, they kept my feet on the ground.

Finally, none of my education would have been possible without the unending support of my parents, Bradley and Kristina Strand. I cannot count the number of times that they had just the right advice to address whatever problem I was facing. Their example has inspired me from the day I knew the meaning of the word, and their wisdom has put me on a path that I could never have dreamed.

List of Abbreviations

BBV	Basic Block Vector
BPF	Berkeley Packet Filter
BSD	Berkeley Standard Distribution
DCPI	Digital Continuous Profiling Infrastructure
EKF	Extended Kalman Filter
EMA	Exponential Moving Average
FFT	Fast Fourier Transform
FLOP	Floating-Point Operation
FP	Floating-Point
IPC	Instructions Per Cycle
IQR	Interquartile Range
KF	Kalman Filter
LLM	Large Language Model
MSE	Mean Squared Error
MSR	Model-Specific Register
NPB	NAS Parallel Benchmarks
nRMSE	Normalized Root-Mean-Squared Error
OLS	Ordinary Least Squares
PID	Process ID
PMC	Performance Monitoring Counter
PMU	Performance Monitoring Unit
PSD	Positive Semidefinite
RNG	Random Number Generator
SIMD	Single Instruction, Multiple Data
TID	Thread ID
TLB	Translation Lookaside Buffer
UKF	Unscented Kalman Filter

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Performance counters are among the most useful tools available to a systems engineer, but modern microarchitectures often support only a small handful of physical counters. This leaves the engineer with two unsatisfying paths. They can select a small set of counters to monitor at high-fidelity, or they can rely on kernel-managed time-division multiplexing, which adds noise as the number of events increases.

1.1. Challenges

Hardware performance monitoring units support measuring dozens of critical performance characteristics, ranging from cache behavior across the hierarchy to the utilization of specific functional units. Limiting the performance engineer's visibility to just four or eight of these events can obscure cross-cutting behaviors which would otherwise be plainly visible. For example, if the L1D cache thrashes and starves the core's functional units, but the profiler is only observing the L2 and L3 cache behaviors, the resulting performance degradation will be entirely unattributed.

The natural inclination is then to observe more events concurrently. Although this can result in broader visibility across the system, measurements become noisier and necessarily less granular as the kernel is forced to multiplex the set of events over the smaller set of counters (see §3.2). Without changes at the hardware level to support more concurrent counters, the fundamental tension between the number of observed

counters and the quality of those observations must be intentionally managed rather than passively accepted.

1.2. SACCADE

We present SACCADE[1], a profiling tool that takes a third path, maintaining broad coverage while actively managing noise. It separates hardware counter orchestration into two distinct components: a mechanism to estimate the current state of the system given recent counter observations, and a policy to select which counters should be observed at any given time.

This design enables a proactive approach to managing the noise associated with hardware counter multiplexing. Schedulers can use the inferred current state of the system to make an informed decision about what counters to observe, and those observations flow back into the state estimator for use in the future. The scheduling policy is a natural place to apply an LLM to support natural-language steering and adapting to changing workload behavior.

Furthermore, SACCADE is not only a profiler, but a framework for evaluating profiling strategies. Nearly every facet of the application is pluggable, meaning that schedulers, estimators, and even the event source itself can be reconfigured.

[1]Pronounced sə-KAHD, the name is taken from the rapid intermittent movement of the eye as it focuses on different points in the visual field [8]. Just as the eye can only focus on a small area at a time and the brain must reconstruct the whole image, our tool can only observe a few counters at a time and must infer the state of the whole system.

1.3. Research Questions

This design motivates questions spanning architecture (Q1 and Q2), the performance of LLM-driven schedulers (Q3 and Q4), and practicality (Q5).

(1) Does closed-loop scheduling outperform an open-loop baseline on reconstruction error?
(2) How separable are the scheduler and estimator axes and where do they interact?
(3) How do LLM-driven schedulers perform compared to statistical baselines?
(4) How does natural-language query structure affect an LLM-driven scheduler's ability to reallocate profiling effort towards regions of interest?
(5) What is the overhead of adaptive scheduling?

1.4. Contributions

SACCADE is a flexible framework for prototyping, evaluating, and deploying a wide variety of profiling configurations. We take advantage of this flexibility to present novel estimators and schedulers, and evaluate LLM-driven schedulers' ability to steer profiling effort via natural-language guidance. Our contributions are as follows:

(1) SACCADE, a framework for evaluating performance counter schedulers and estimators, driven by a globally-synchronized orchestrator over simulated counters or real, running applications.
(2) A decomposed architecture that frames schedulers and estimators as independent, mix-and-match components.
(3) A family of estimators, including a formulation of virtual counter reconstruction as a Kalman Filter state-estimation problem.

(4) A collection of LLM-driven schedulers that support natural-language guidance to reallocate counter focus.

(5) A mechanism to reconstruct a global counter trace by stitching multiple anchored profiling passes, and replay it through any scheduler/estimator pairing.

(6) Guidance for the optimal configuration of the SACCADE profiling system to achieve a balance of accuracy, performance, and convenience.

1.5. Thesis

The relative scarcity of hardware counters compared to performance events forces a tradeoff between breadth and fidelity. We reframe this tradeoff as a problem of state estimation and observation scheduling, which we make tractable with a decoupled architecture. Using SACCADE, we find that closed-loop scheduling can reduce reconstruction error, but that the best scheduler and estimator are coupled and no single configuration consistently dominates passive multiplexing. We also find that LLM-driven schedulers allow for natural-language guidance to focus profiling effort and, for workloads that outlast schedule generation latency, support mid-run reconfiguration.

1.6. Roadmap

This thesis is organized as follows: Chapter 2 covers background on PMUs and their multiplexing, eBPF, and Linux scheduling. Chapter 3 presents a survey of the history and state of the art of profilers, PMU multiplexing, eBPF tools, phase-aware simulation and profiling, and autotuning. Chapter 4 describes the design of SACCADE and its motivations. Chapter 5 details the implementation, focusing on the eBPF and hardware subsystems, Kalman Filter configuration, and LLM integration. Chapter 6 contains

our evaluations, both of SACCADE as a framework and the estimators and schedulers it supports. Chapter 7 discusses the limitations of our system and evaluations, and describes opportunities for future work. Chapter 8 summarizes our findings.

We also include two appendices. Appendix A documents our experience using AI tools in the development of a large system for the benefit of future practitioners. Appendix B lists the full text of our LLM prompts and configuration.

CHAPTER 2

Background

2.1. Hardware Performance Monitoring Units

Most modern CPU cores have a performance monitoring unit (PMU). The central purpose of the PMU is to store a set of configurable counters and increment each when its associated event occurs on the main core. The space of events that can be tracked by the PMU is in the hundreds. Our test machine (see §6.1 for details) supports tracking 223 unique events, but other CPUs (especially those from Intel) can support more than twice as many.

These counters are immensely useful for performance engineers as they track down bottlenecks in their applications. The engineer can configure the PMU to observe some events of interest, run the program, and poll those counters periodically during execution. If, for example, the engineer sees that a phase of the workload has a particularly high rate of L1 instruction cache misses, they know that that section could be optimized by reducing code size and inlining. Alternatively, if they see a high frequency of TLB misses, they could use larger pages for contiguous slabs of memory.

PMUs have a wide variety of other features designed to observe the core's execution. These include power/thermal monitoring, interrupt generation[1], and keeping a record

[1]Fire an interrupt after a counter has seen k events, for example.

of the recent branches taken for call-graph reconstruction. In this work, we focus on the event counters themselves for maximum portability.

Innumerable profiling interfaces and tools have been built atop the foundation of the PMU, the most famous of which is the eponymous `perf` (for more detail, see §3.1).

2.2. The Multiplexing Problem

Although it would be ideal to track all of the available counters simultaneously, this is not possible without dedicating a prohibitively large fraction of the core's power and area budget to the PMU. Instead, the PMU supports concurrently tracking only a small subset (usually 4 or 8) of the total events. Profiling tools get around this limitation by multiplexing the desired set of events across the physical counters, and assuming event rates stay constant during the resulting observation gaps. This necessarily introduces noise into the system.

Approaches to minimizing this noise have traditionally been post-hoc analysis passes. The history of PMU counter multiplexing and strategies for minimizing noise are expanded on in much more detail in §3.2.

2.3. eBPF

The event rate for any given counter can fluctuate wildly over the course of a program's execution. To capture this variability, it is ideal to sample counters at a high frequency. Doing this from userspace is challenging because the hardware counters

themselves are protected by the kernel, so we must use a system call to get the raw value[2], which we found to be prohibitively slow.

Instead, SACCADE moves that polling logic into kernelspace using eBPF, eliminating those time-consuming system calls. eBPF allows an application to attach a small program to various tracepoints in the kernel. Then, the eBPF program can communicate with the host application through shared-memory data structures. One especially useful communication channel is a high-performance ringbuffer, which we use to shuttle counter samples from the eBPF program to the host application.

See §3.3 for more on the history and applications of eBPF, and see §4.2 and §5.1 for how eBPF is used in SACCADE. By sampling at a high frequency and scheduling counter observations intelligently, SACCADE is able to reduce multiplexing noise.

2.4. Linux Scheduling

At any time during a user program's execution, it can be preempted by the Linux scheduler, put to sleep for an indeterminate amount of time, and resumed, perhaps on a different core. This introduces many complexities with respect to the PMU and performance counters.

Firstly, the performance counter registers themselves are core-local, so they will not, by default, be transferred with the process to the new core. Furthermore, any notion of event "rate" must account for the time the process was asleep and could not make any forward progress or produce any performance events. These issues can be realized as observing meaningless negative rates, or worse, plausible but skewed positive rates.

[2]We cannot use the `RDPMC` instruction to get the counter values directly because it does not mingle well with thread migration and we need the read to happen in-kernel at specific hooks.

SACCADE goes to great lengths to alleviate these issues by attaching an eBPF program to the `sched_switch` tracepoint, which fires on every thread switch. Details of this mechanism can be found in §4.3.

CHAPTER 3

Related Work

While profiling systems have continuously improved since their inception, the gap between the event measurements desired by users and the number of available hardware counters has been treated as a fixed constraint. §3.1 traces the evolution of production profilers, from `gprof`'s software instrumentation to the hardware-based, transparent, whole-system profiling that established SACCADE's design goals. §3.2 examines the multiplexing problem directly, and the reactive tradition that has treated it as an unavoidable cost. §3.3 surveys the eBPF observability ecosystem, which provides SACCADE's sampling mechanism while leaving the control plane in userspace. §3.4 reviews phase-aware simulation and profiling, which establishes that program behavior is structured, the property that makes proactive scheduling feasible. §3.5 covers autotuning and LLM-driven control, which demonstrates closed-loop, model-driven system tuning while assuming a fixed signal set. Each of these approaches can be categorized by whether it is proactive or corrective about multiplexing noise, and portable or kernel-modifying in its deployment.

3.1. Production Profilers

Many modern profilers can trace their lineage back to `gprof` [17], the first call graph execution profiler. Developed in the 1980s at UC Berkeley, it operates entirely at the software level. The compiler is configured to place a call to a tracing routine in the

prologue of each program routine. A performance engineer can then perform post-hoc analysis on the call counts and execution times of each profiled routine to generate a call graph and accurately aggregate the total time spent executing a given routine, including that routine's children. Unfortunately, `gprof` imposed a 5-30 percent overhead on the targets being profiled.

By the turn of the millennium, hardware-based profiling had become a viable option for performance engineers. A notable example is the Digital Continuous Profiling Infrastructure [3, 2]. DCPI's hardware-focused approach imposed a mere 1-3 percent overhead and has many notable advantages over `gprof`. First, its support for multiprocessing far exceeds that of software-based profilers, which would need to manage thread-safe shared data structures to collect meaningful information. Instead, that work has been offloaded to the hardware's PMC mechanisms. Furthermore, DCPI runs on *unmodified* binaries, meaning that it can be enabled completely transparently to the running program. Finally, DCPI supports whole-system profiling. These three features directly inspired SACCADE's design goals and significant effort was invested in attempting to reach them.

Throughout the early 2000s, the Linux kernel began to stabilize on a userspace interface for hardware performance counters. At that point support had waxed and waned for various performance monitoring subsystems including `perfctr` [32], `oprofile` [13], and `perfmon2` [15]. The more abstracted `perf_event` interface [1] was introduced in 2008 and soon became the standard mechanism for performance engineers to interact with performance counters. The `perf` tool provides userspace access to the hardware

counters and has become a critical tool for performance engineers because of its introspective power and ubiquitous support. SACCADE mirrors `perf`'s architecture, operating as an orchestrator over the `perf_event` interface.

This interface was quickly put to use to develop a top-down strategy for identifying bottlenecks in out-of-order processors [48]. It was a practical method that involved following a hierarchical approach, starting with identifying if the processor is bound by its frontend, speculation, retiring logic, or backend. The performance engineer then follows a decision tree, looking at various performance counters to isolate the root cause of the bottleneck (iTLB misses, for example).

Before long, engineers at Google used continuous profiling techniques and top-down analysis through the modern `perf_event` interface to characterize the observed 30 percent "datacenter" tax associated with cloud machines [33, 22], underscoring the importance of low-overhead, accurate, and continuous profiling tools.

Each generation of profiling infrastructure has improved access to hardware counter data, optimizing interfaces and minimizing overhead. There remains a fundamental tension between the number of counters the user wants to observe and the limited number of slots available. The tools must multiplex naively or force the user to select a subset of their desired counters, treating this limitation as an accepted cost rather than a solvable problem. SACCADE builds on this imperfect ecosystem and attempts to make counter data trustworthy.

3.2. PMU Multiplexing

Although modern CPUs often support tracking hundreds of distinct microarchitectural events, they typically provide only a small number of physical counters (six on our test machine). When profiling tools request to track more events than available counters, the operating system must resort to time-division multiplexing.

Until 2014, the Linux `perf_event` subsystem handled this via round-robin scheduling, and scaling the final event count up by the fraction of time that it was running, implicitly assuming a uniform distribution of events across the target's execution. The error introduced by this multiplexing was substantial. In 2005, Mathur and Cook demonstrated that nearly 42 percent of sampled intervals suffered from errors greater than 10 percent [29].

To mitigate this issue, an improvement to the round-robin scheduling algorithm was introduced in 2014 [26]. This approach monitored the target task's behavior to schedule volatile events more frequently, reducing the MSE of measurements by 22 percent. SACCADE supports this scheduler and we describe the design in §4.6.2.2.

Even with improved scheduling, residual multiplexing error can have concrete consequences for analysis. Xu et al. demonstrated that multiplexing errors can misattribute hotspots, reporting that a function was responsible for 50 percent of execution time when in fact it was only responsible for 34 percent, and proposed a software-based post-hoc rectification framework [46].

Instead of treating events as discrete counters, BayesPerf reframes the problem as a matter of uncertainty quantification [9]. It models the deterministic microarchitectural relationships between different hardware counters using a Bayesian probabilistic

graphical model. By jointly inferring the values of multiplexed events as probability distributions rather than scalar point estimates, BayesPerf reduces the average multiplexing error from over 40 percent down to below 8 percent.[1] SACCADE draws inspiration from representing the counters as probability distributions, specifically in our Kalman Filter §4.5.3 estimator.

CounterPoint [27] takes a different stance on the multiplexing problem altogether. Rather than attempting to recover accurate counts, it relates noisy readings to one another[2] to support or refute user-defined microarchitectural assumptions. The tool therefore treats noisy counters as evidence to be reasoned about rather than a signal to be cleaned, validating a model rather than producing trustworthy values. SACCADE shares the intuition that the relationships between counters carry exploitable information, but applies it proactively at the scheduling layer rather than as a post-hoc check against a hypothesis.

Recent work has attempted to modify the kernel itself to expose the uncertainty generated by multiplexing at runtime to userspace. Tintin [24] introduces the Event Profiling Context (ePX), which operates in parallel to the `perf_event` subsystem to monitor, schedule, and estimate the values of performance counters. SACCADE's architecture is similar to that of Tintin, but does so without requiring new kernel interfaces, depending only on the `perf_event` and eBPF subsystems present in all modern Linux kernels.

[1]The catch is that this model is prohibitively expensive to run on the target CPU as it involves solving Bayesian inference in real-time, so the authors present a new discrete hardware accelerator to handle these computations.
[2]Their algorithm is similar in spirit to the methods we apply in §5.3.1.

The error introduced by PMU multiplexing has been addressed in several ways over the past two decades. All of the post-hoc approaches have conceded that multiplexing is unavoidable, so they focus on managing and mitigating the noise. SACCADE is proactive rather than corrective, manipulating the multiplexing schedule itself to minimize error, rather than accounting for it after the fact. Although the most modern tools do support counter scheduling and exposing uncertainty to userspace, they require extensive modifications to the kernel. SACCADE, by contrast, is portable and runs nearly exclusively in userspace, making broad adoption straightforward.

3.3. eBPF-Based Observability Tools

The BSD Packet Filter [30] was introduced in 1992 to provide improved facilities for user-level networking. At the time, network monitors were required to copy packets from kernel space to userspace before they could be processed. BPF allowed the user-level network monitor to configure a kernel-level "packet filter", which would discard unwanted packets before they could even be copied to userspace, saving memory bandwidth.

eBPF [19] was introduced in 2014 as an extension[3] of BPF to support a much broader range of kernel-level agents. It is a secure, event-driven, Turing-complete execution environment within the kernel. Programs are written in C and compiled to eBPF bytecode. At runtime, the kernel verifies that the program adheres to a variety of security policies (see §5.1.2 for our experience with the verifier) and just-in-time-compiles it to machine code. The machine code is then attached to any number of kernel tracepoints, dynamic

[3]In reality, the name eBPF was chosen to placate nervous kernel maintainers. It is only an "extended" version of BPF in that it is backwards-compatible with classic BPF programs. The underlying architecture and runtime were completely rewritten.

kernel probes, or user-space probes. Critically, all of this happens without requiring an unsafe kernel module or a system reboot.

This functionality has enabled a vast ecosystem of observability tools, beginning with BCC [18] and bpftrace [34], which provided initial high-level tracing abstractions for performance engineers looking to capture granular system metrics. While BCC and bpftrace are effective tools for ad-hoc profiling, other tools, like Parca [39] and Pixie [23] are designed to support continuous profiling. Parca collects high-frequency stack traces across an entire fleet of machines, allowing users to build Prometheus-style dimensional queries of system bottlenecks with minimal overhead. Moreover, since eBPF runs at the system level, the tool profiles applications written in any language without manual instrumentation or even so much as restarting the application. Pixie is tailored for Kubernetes environments and primarily traces network traffic.

While eBPF's operational space has traditionally been limited to the CPU, the recent eGPU project [47, 50] offloads eBPF bytecode to GPUs via dynamic PTX injection. This allows eBPF-based instrumentation to receive real-time GPU telemetry and help performance engineers identify bottlenecks in their GPU-driven workloads. The authors were also able to use this mechanism to automatically tune GPU kernels to improve throughput on their workloads by up to 4.8×. eInfer [12] is a similar tool that offers distributed, vendor-agnostic observability of LLM inference pipelines.

None of these tools address the multiplexing problem inherent to PMU-based profiling. eBPF's role in SACCADE is limited to reliable and low-latency delivery of hardware counter samples to userspace. The decision about which counters to activate is made entirely in userspace, and the eBPF program samples whichever counters are active.

3.4. Phase-Aware Simulation and Profiling

The scale of modern software stacks renders full-fidelity profiling impossible. Execution traces for contemporary enterprise workloads frequently encompass trillions of instructions. Programs, however, are not fully chaotic systems. They exhibit highly structured, repetitive execution phases. By exploiting these phases, profiling systems can drastically reduce data collection demands by sampling a representative fraction of the program's unique behaviors and filling in the rest with statistical inference.

The SimPoint methodology [35] represents the seminal work in establishing the formal mathematics of this domain. Targeting architectural simulation rather than profiling, the authors demonstrated that a program's execution can be characterized by a sequence of Basic Block Vectors (BBV), a one-dimensional array tracking the execution frequency of every basic block in the program over about 100 million executed instructions. Since each BBV captures the control-flow of the program at a point in time, it acts as a fingerprint for each phase of execution. These fingerprints are then grouped into clusters, where each cluster represents a similar phase of execution. By simulating the code associated with a single BBV close to the center of a cluster and extrapolating the information gathered to all BBVs in that cluster, researchers simulated just a small fraction of the program while predicting whole-program behavior with high accuracy (IPC within a few percent).

Following this success, the team behind SimPoint brought their work into the hardware [36]. At each branch instruction, they record the program counter and the number of instructions executed since the previous branch instruction. This tuple identifies the basic block and the number of instructions it executed, the core inputs into their

prior work. Critically, doing this at the hardware level allows phases to be identified *online*. Furthermore, they were able to use this mechanism to predict future phases and dynamically reconfigure resources like the cache layout to support those future phases.

Simultaneously, Dhodapkar and Smith demonstrated that as an application shifts phases, its instruction and data working sets change in predictable ways [14]. They maintain a compact signature for each working set to detect phase changes and trigger re-tuning, and avoid re-tuning when a phase is repeated. Their algorithm, when applied to reconfigurable instruction caches, achieved power savings and performance equivalent to the state of the art, but with far fewer re-tunings.

Taken together, these works show that program behavior is structured and repetitive. SACCADE attempts to take advantage of this predictability to identify the most informative hardware performance counters to observe during the current execution context. Rather than using phase information to reduce observations, we use prior observations to prioritize what to observe next, implicitly responding to phase transitions through the feedback loop.

3.5. Autotuning and LLM-Driven System Control

Modern systems, be they hardware, software, or somewhere in between, have innumerable configuration parameters that can be tuned to achieve the maximum performance in any given deployment. This is necessary because it is difficult for engineers to predict what environments their systems will run in and how those environments will change over time. Unfortunately, these configuration parameters, while powerful

and helpful, are often overwhelming to a system administrator who needs to stand up a program or piece of hardware in short order.

Autotuning systems attempt to alleviate this issue by automatically profiling the execution environment and setting the system's configuration parameters optimally. The work of Harutyunyan Gevorgyan et al. [20] uses hardware performance counters to characterize a GPU's execution environment and predict optimal thread allocation and thread affinity. The counters act as features which are fed into an ensemble ML model, emitting the optimal parameter configurations. They found that this method reached configurations with performance competitive to prior search-based methods in a fraction of the time. SACCADE, in contrast, is a closed loop with the model outputting the observation configuration, and that observation getting fed back into the model.

Classical ML autotuners struggle with semantic gaps and brittle rewards. Liargkovas et al. [25] address this by using an online, LLM-driven agent to tune the parameters Linux Completely Fair Scheduler, finding that their approach outperforms both Bayesian optimization and human experts.

Zheng et al. [49] unleash the LLM to generate custom schedulers and run them at the system level using eBPF and sched_ext. Their LLM agents independently analyze active workloads and generate system scheduling policies that are optimal for those specific workloads. This architecture directly mirrors that of SACCADE (when running with an LLM-driven scheduler, see §4.6.3), with LLMs managing the control plane and their decisions implemented through eBPF.

These works demonstrate that both classical machine learning techniques and LLM-driven tools can effectively autotune complex systems. They all assume, however, that

their observation layer is fixed, consuming whatever signals happen to be available. SACCADE instead focuses on the quality of the observations themselves, intelligently scheduling hardware performance counters to extract as much information as possible from the system.

Across the prior work in this area, tools are often given a fixed observation budget and treat the multiplexing noise as an unavoidable cost. Instead of applying reactive, post-hoc analysis to inherently noisy counter readings, SACCADE proactively schedules the counters that are most likely to have meaningful information. Unlike other tools which require changes to the underlying kernel, SACCADE runs almost entirely in userspace and communicates with the kernel through existing, well-supported interfaces. The next chapter lays out how SACCADE realizes this.

CHAPTER 4

Design

SACCADE decouples the challenge of managing performance event counters and their data into four layers. At the lowest level are the hardware counters themselves; a physical system resource. The second layer is an eBPF data path which manages sample gating and sending high-frequency counter data to userspace. A userspace orchestrator is the third layer. It handles the mechanics of swapping counters via the `perf` subsystem, extracting counter data from the eBPF ringbufs, and enforcing swap atomicity. The final layer is the scheduler and estimator abstractions, which decide which counters to activate at any given time to read the sparse data available from the current observation to estimate the state of *all* of the counters, respectively. Any scheduler can pair with any estimator, which makes the design space tractable and enables interesting evaluation strategies such as a 2D ablation.

This architecture decomposes adaptive profiling into four orthogonal concerns.

(1) The event **source**.

(2) A **schedule** to control the event stream.

(3) A way to estimate the **state** of the system given limited information.

(4) A set of **sink**s that can store and visualize the input data.

Each of these can be mixed and matched to construct a scenario to evaluate. For example, a configuration that mirrors the scheduling and estimation behavior of early versions of `perf` would

(1) use the hardware event source.

(2) the round robin scheduler.

(3) the 'propagate' state estimator.

(4) and the perfetto data sink.

It is easy, however, to imagine other useful configurations mixing together synthetic event sources with experimental schedulers and estimators, and novel output formats. Figure 4.1 shows how these interfaces are composed.

This configurability and adaptability enables a great deal of the evaluations and results presented here, and is a core contribution of this work. SACCADE allows researchers to freely experiment in the event scheduling and imputing space, trusting that the complex underlying work of applying schedules and wiring together high-performance datastreams will be handled correctly.

4.1. Hardware Substrate

This work takes advantage of the performance monitoring unit (PMU) present in modern CPU cores. Unfortunately, most cores only have a handful of PMU event counters (six on our test machine), representing the core constraint in any potential scheduling decision. On x86, the configuration of these counters is managed through issuing a `WRMSR` instruction targeting the beautifully named `IA32_PERFEVTSELx` family of registers. Since the name and behavior of these MSRs are a *micro*architectural concern,

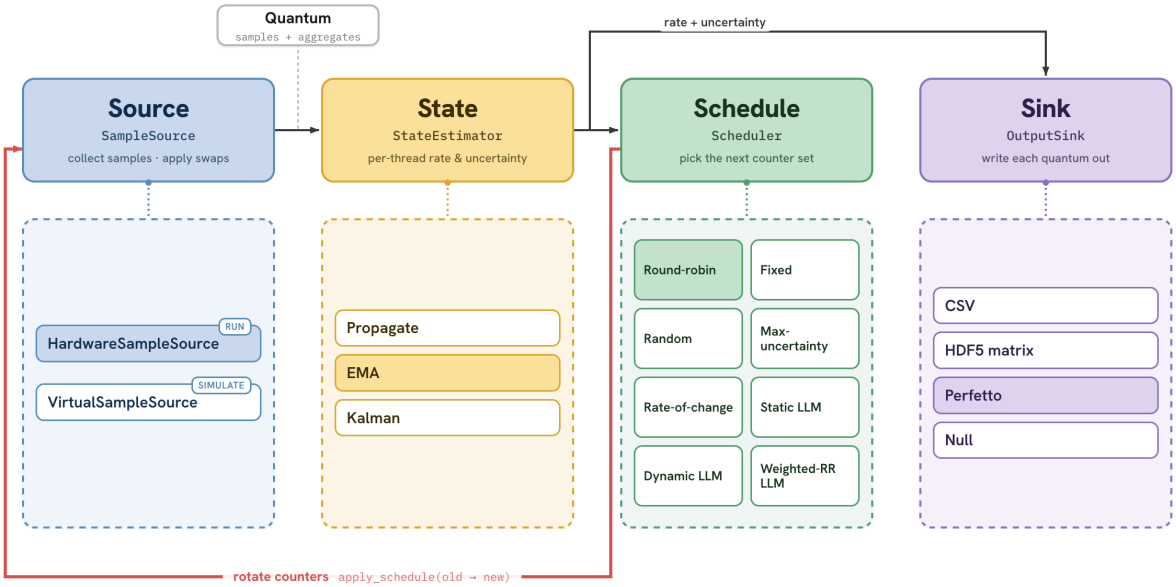


Figure 4.1. The high-level architecture of SACCADE. Each layer can use one of the provided implementations. This allows SACCADE to be used for both online profiling and high-performance simulation of a wide variety of profiler configurations.

the logic to manipulate those MSRs must be customized to the *specific* microarchitecture under test, which would require a vast body of kernel code.

Fortunately, the `perf` subsystem in Linux has already done this work to support the `perf` profiling tool. SACCADE issues requests to manipulate the counters via the `perf_event_open` syscall, and the `perf` subsystem handles translating those requests into the complex MSR writes and potential inter-processor interrupts necessary to configure a performance counter.

Each counter on each CPU gets its own file descriptor, with counters enabled on all processes. This gives SACCADE complete control over the PMU configuration on the entire machine and enables future support for multithreaded applications and system-level profiling. There is, of course, an overhead associated with these syscalls (characterized

in §6.2.3). But, optimization opportunities abound, as described in §7.2.1. Moreover, this overhead is completely eliminated when running under SACCADE's simulation mode (§4.7).

4.2. eBPF Data Path

The counters must be observed if we want to collect any information from them. Since the PMU is a kernel-protected resource, performing these observations is traditionally done by reading from a file descriptor. Since we want to poll at a relatively high frequency (dozens of kHz), this method would introduce far too much overhead and perturbance to gain meaningful insights about the target program. So, the work of reading the counters themselves is moved into the kernel using an eBPF program; the "sampler".

The sampler is loaded and verified as SACCADE initializes, and is invoked when the kernel performs a context switch or by a high frequency software clock. The user program configures the sampler with a minimum sample interval, an optional target program PID, and a global tracking enable switch, all through variables in shared memory. When triggered, the sampler identifies if a sample is unnecessary (returning early if so), reads the counters, packages them with the TID, event IDs, a timestamp, and the sample type, and puts the sample in a high-performance kernel-to-user ringbuffer. A sample is emitted if and only if the user thread is a part of the target program and SACCADE is not actively performing a counter switch. This design decomposes to a logical state machine, shown in Figure 4.2.

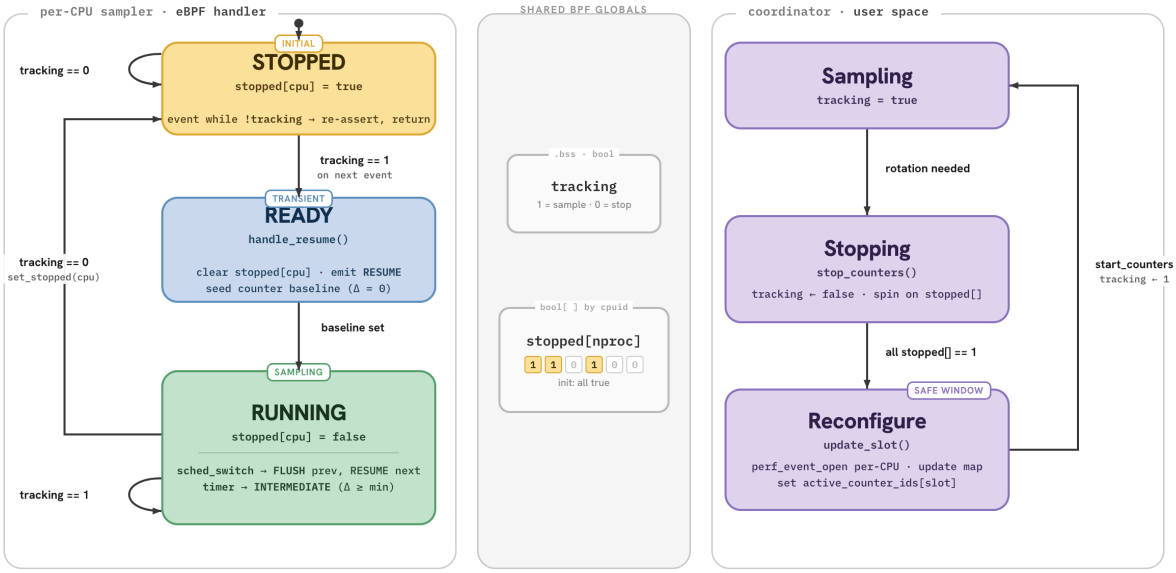


Figure 4.2. The per-CPU eBPF sampler state machine, gated by the global tracking flag, alongside the coordinator's stop-the-world cycle. The coordinator clears `tracking` and spins on the cpuid-indexed `stopped[]` array until all CPUs park, then reconfigures counters and re-enables. Arrows are state transitions; the center cells are the shared globals the two sides coordinate through.

4.2.1. Thread State

SACCADE must know if a particular thread should be sampled and when the most recent sample of that thread occurred. This state is stored in a map where the keys are TIDs and the value is the timestamp of the most recent sample. If a thread ID does not appear in the map, then it should not be sampled.

The map is initialized empty, so every thread is untracked. The first time the sampler sees a thread with the target PID (or any thread, if the target PID is absent), either via a context switch or a timer, it adds that thread's TID to the map along with a timestamp. It then emits a `RESUME` sample to tell the user layer to set its counter

baseline. This is necessary because the event counters do not necessarily start at zero; this avoids artificially large readings at the beginning of the run.

When the sampler is invoked from the timer, it checks the current TID against the map. If the current thread is being tracked, and enough time has passed since the previous sample, the sampler emits a `INTERMEDIATE` sample and updates the timestamp in the map.

When the sampler is invoked from a context switch, it checks the outgoing TID against the map. If the outgoing thread is being tracked, it emits a `FLUSH` sample and removes the TID from the map as it is no longer being tracked. If the incoming thread is being tracked, it emits a `RESUME` sample as above.

4.2.2. CPU State

The state of the counters is inconsistent while SACCADE enacts a scheduling decision. Readings from those counters could pollute the rest of the system, so it is important to avoid collecting any samples during the counter swap. Since the counters are being read on all CPUs, this is a nontrivial requirement. This is accomplished using a stop-the-world synchronization mechanism.

The user code requests that sampling stop by setting a flag in shared memory. When the sampler detects that the flag has been set, it disables itself on that CPU and acknowledges that the CPU has seen the global flag by setting its own, CPU-specific, flag.

When the sampler is stopped and sees that the user is no longer requesting a world stop, it emits a `RESUME` sample so that the user can set its counter baseline so that any

events that occur during the swap (which cannot be attributed to a specific event) are omitted from the results.

4.3. Userspace Orchestrator

The mechanics of implementing a scheduling decision are nontrivial. `perf` file descriptors must be closed and opened, the eBPF sampler must be disabled and enabled, and through it all, the samples themselves must be read from the eBPF ring buffer and fed into the estimator. The orchestrator must also initialize and configure the sampler, and insert it into the kernel.

4.3.1. Implementing Scheduling Decisions

A critical challenge in managing a counter swap is maintaining the correct baseline from which to compare the next event count. Event counts are reported from the PMU as an *absolute* value, but performance engineers are interested in the *rates* of these events.[1] If the counters remain constant throughout a program's execution, converting these absolute values into rates is trivial: simply iterate through the vector of absolute values and compute the difference between each value and its predecessor.

In SACCADE, this conversion must be done online because the estimators consume these rates and schedulers react to those estimates. Moreover, the counters are dynamically reconfigured at runtime, so the simple difference between subsequent samples would produce wild and often negative rates when comparing samples of two different events. To resolve these issues, the orchestrator maintains a vector of "previous values" for each counter, per CPU. On each sample, the difference between the sampled value

[1] "34 L2-cache misses per millisecond" is far more useful than "5,234,895 L2-cache misses".

and the previous value is computed and emitted. This brings the computation online, but it does not help with the issues around dynamic reconfiguration.

To support dynamic reconfiguration, SACCADE does not collect samples while the hardware counters are being configured. Our mechanism (described in §5.2.1) is ripe for optimization, particularly in the common case of a single-threaded workload. We describe these opportunities in §7.2.1.

4.3.2. Sample Handling

The eBPF sampler can emit hundreds of thousands of samples per second. The orchestrator must drink from this proverbial firehose and aggregate the samples so that they can be passed to the estimator at a lower frequency.

Samples are initially read from the eBPF ring buffer and are immediately placed in a large userspace queue for processing. This allows the majority of the buffering to take place in userspace, leaving the kernel-impacting execution paths as free as possible. The samples are eventually read from the userspace queue into an aggregate data structure. The structure encodes the timestamp of the read, the duration over which the samples were collected, the samples themselves, and lazily-computed aggregate statistics. This structure is passed to the estimator.

4.4. Scheduler/Estimator Abstraction

SACCADE decouples the question of how to estimate the current state of all performance counters from how to decide which counters to observe next through the `Scheduler` and `Estimator` traits listed in Figure 4.3. A scheduler is initialized with

```rust
pub trait Estimator {
  fn measurement_update(
    &mut self, tid: u32, event_id: EventId,
    rate: f64, stddev: f64,
    num_samples: u32, timestamp_ns: u64,
  );
  fn time_update(
    &mut self, tid: u32,
    event_id: EventId, elapsed_ns: u64,
  );
  fn all_estimates(&self)
      -> &HashMap<EstimateKey, Estimate>;
}

pub struct Estimate {
  pub rate: f64,
  pub rate_stddev: f64,
  pub uncertainty: f64,
  pub last_updated_ns: u64,
  pub sample_count: u64,
}
```

```rust
pub trait Scheduler {
  fn init(
    &mut self,
    all_events: Vec<EventId>,
    num_slots: usize,
  ) -> Result<(), Box<dyn Error>>;

  fn next_step(
    &mut self,
    quantum: &Quantum,
    estimator: &dyn Estimator,
  ) -> ScheduleDecision;
}

pub struct ScheduleDecision {
  pub active_events: Vec<EventId>,
  pub duration: Option<Duration>,
}
```

Figure 4.3. The `Estimator` (left) and `Scheduler` (right) traits. An `Estimator` ingests new measurements (`measurement_update`), advances its internal model between samples (`time_update`), and exposes a posterior estimate for every event (`all_estimates`); a `Scheduler` is initialized with the event pool and slot count, then on each quantum returns the next set of events to activate. These two traits delimit the design space available to estimators and schedulers.

the total event pool and the number of counter slots supported by the target machine's microarchitecture. On each scheduling quantum boundary, the orchestrator provides a complete `Quantum` (§4.3.2) and a reference to the current `Estimator`. The scheduler then advances the `Estimator` with the new event measurements, and can then query the `Estimator` for a posterior estimate of the rate of all events. This estimate of the microarchitectural state is then used by the `Scheduler` to select which counters to activate

next. The `SchedulingDecision` is finally returned to the orchestrator to be executed. Importantly, the implementations of the `Scheduler` and `Estimator` are fully decoupled.

4.5. Estimators

State estimation has been approached from countless angles in nearly every field where there is a notion of state; from robotics to probability. This work only scratches the surface of what is possible. The estimators presented here are merely a starting point.

4.5.1. Propagate

The propagation estimator is the simplest estimator explored in this work. It assumes that any given rate measurement will persist until the estimator receives a new measurement for that counter. This behavior is quite similar to how early versions of `perf` handle multiplexed counters, so it is a sensible baseline.

Its strength is in its simplicity. It has no configuration parameters and for workloads with consistent event rates, it can be expected to perform well. Unfortunately, if a workload has event rates that are fluctuating, the resulting trace data will show a sharp step whenever the counter is swapped in rather than a smooth line.

4.5.2. Exponential Moving Average

The exponential moving average (EMA) estimator is a simple recursive filter that approximates the true state of the system from noisy measurements. It is extremely lightweight, maintaining a single state variable per counter, and a single global configuration parameter. We can formulate the EMA estimate of the value of a particular

counter i at time k as $\hat{x}_k^i$.

$$\hat{x}_k^i = \alpha \cdot z_k^i + (1 - \alpha) \cdot \hat{x}_{k-1}^i \tag{4.1}$$

where $\alpha \in (0, 1]$ is the smoothing factor (0 is smoother, 1 is noisier). Old measurements receive exponentially decaying weight in the state estimate and this memory is implicit, meaning there is no need to actually store old samples.

We give each counter its own EMA filter because measurements are sparse with respect to the pool of all events and it is not clear what the measurement value should be for a counter that was not actually measured.

4.5.3. Kalman Filter

The Kalman Filter (KF), borrowed here from our friends in probabilistic robotics [21], is a step up in terms of mathematical complexity and modeling power. It is an optimal recursive Bayesian estimator for linear systems with Gaussian noise. PMU noise is not strictly Gaussian; rather it is Poisson at low counts and heavy-tailed during phase transitions. This limitation is discussed further in §7.1.

The state is characterized as a Gaussian belief with mean $\hat{x}$ and covariance P [45, §3.2]. In SACCADE, our state is the vector of all performance counters. Because P is a full covariance matrix, a measurement of counter x_i updates the belief of every other counter, including counters that were not observed this quantum.

We must also define how the state reacts to measurements and the passage of time. First we have the simplified[2] process model, which expresses how the state evolves and becomes less certain over time:

$$x_k = Fx_{k-1} + Bu + w, \qquad w \sim \mathcal{N}(0, Q) \tag{4.2}$$

where x_k is the true state, F is the state transition matrix, B is the control input matrix, u is the control vector, and w is the process noise, which is described by the covariance matrix Q. For our case, F is always the identity, and we have no control input, so Bu can be elided. Q can be configured at startup to reflect learned correlations between counters.

Second, we have the measurement model, which describes how the state estimate updates in response to observations:

$$z_k = H_k x_k + v, \qquad v \sim \mathcal{N}(0, R) \tag{4.3}$$

where z_k is the measurement actually observed, H_k is the observation matrix, mapping state space to measurement space, and v is the measurement noise, characterized by the covariance matrix R. In our formulation, z_k is the scalar counter rate observed, and H_k is a ($1 \times N$ where N is the counter pool size) one-hot row-vector that encodes which counter has been observed. The measurement noise v and its variance R are scalars set at startup.

[2]We would usually define the models in terms of F_k, B_k, Q_k, R_k, u_k, w_k, and v_k, where the k index allows us to vary these terms over the course of program execution. But, for our application, they remain constant, so we omit the subscripts.

With this framing, we can now approach actually updating our estimate of the current state ($\hat{x}_k$ and P_k). To propagate the belief through the process model, we have:

$$\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1} + Bu \qquad \text{and} \qquad P_{k|k-1} = FP_{k-1|k-1}F^\top + Q \tag{4.4}$$

In our case, with $F = I$ and no B or u, this reduces to the extremely simple

$$\hat{x}_{k|k-1} = \hat{x}_{k-1|k-1}$$

Intuitively, this means that in the absence of an observation, we do not adjust our estimate of the mean of any counter. Our uncertainty grows by Q each quantum without an observation.

To incorporate measurement information into our belief, we first compute the Kalman Gain K_k:

$$K_k = P_{k|k-1}H^\top(HP_{k|k-1}H^\top + R)^{-1} \tag{4.5}$$

Equation (4.5) represents how much we trust our measurement. When R is small, we weight the measurement heavily. When R is large, we ignore the measurement. To actually update our belief, we have:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - H_k\hat{x}_{k|k-1}) \qquad \text{and} \qquad P_{k|k} = (I - K_kH_k)P_{k|k-1} \tag{4.6}$$

The cross-correlation information encoded in Q is incorporated into $P_{k|k-1}$ in Equation (4.4), flows into K_k in Equation (4.5), and finally lands in $\hat{x}_{k|k}$ in Equation (4.6). This mechanism makes the Kalman Filter more powerful than EMA and allows it to update the estimate of counters that haven't been directly observed.

When Q is diagonal, the Kalman Filter behaves much like EMA. We offer two non-diagonal Q configurations, both derived from workloads with full counter data, collected using the mechanism described in §4.7.

First, we compute pairwise Pearson cross-correlations [31] between each counter across the concatenation of all workloads' counter data (see §5.3.1). This mechanism revealed 755 pairs of counters with meaningful cross-correlation.

To augment the empirical correlations, we pass the analytical cross-correlation data, and names and descriptions of each of the available events to an LLM and ask it to amend the analytical cross correlation data with any correlations learned from reading the event documentation or its prior training on the AMD Zen architecture. The LLM identified 32 pairs of counters with meaningful correlation. After merging, this resulted in 780 counter pairs with cross-correlation greater than 0.1. These pairs produce a Q that captures both statistically-measured and architecturally-motivated counter relationships.

Off-diagonal noise is scaled by a configurable constant, and all noise is applied to P scaled by the duration since it was last applied, so counters that go unobserved for longer accumulate more uncertainty.

The estimator's `update` method is called by the scheduler once per quantum, advancing the belief through each event measurement in the quantum. The estimator then provides $\hat{x}$ to the scheduler to inform the next decision.

4.6. Schedulers

We present a set of schedulers that span a broad range of complexities. All adhere to the `Scheduler` interface and can be used interchangeably.

4.6.1. Baselines

These first schedulers ignore the counter state entirely and exist primarily as a baseline against which we can compare the more interesting schedulers to follow.

4.6.1.1. Random. The random scheduler makes no use of estimator state or counter history. Every quantum, it selects counters uniformly at random.

4.6.1.2. Round Robin. The round robin scheduler likewise makes no use of estimator state or counter history. It cycles through all events using fixed-size windows. For example, on a system with ten possible events and six physical counters, the first quantum schedules events $[0, 5]$, the second schedules $[6, 9]$ and $[0, 1]$, the third schedules $[2, 7]$, and so on.

4.6.2. Statistical Methods

These next schedulers do some sort of analytical analysis to take advantage of the available counter state to gather more information than the naive baseline schedulers.

4.6.2.1. Maximum-Uncertainty. The maximum-uncertainty scheduler scans through the estimated state, looking for counters that have a high uncertainty. It then selects the counters with the highest uncertainty for scheduling. Without any intelligent state estimation, this scheduler decays to round robin, because the only way for an event's uncertainty to decrease is for it to be directly observed. But, when paired with the

Kalman Filter estimator, the maximum-uncertainty scheduler will schedule correlated events less frequently to maintain uniform uncertainty across all events.

4.6.2.2. Rate-Of-Change. The rate-of-change scheduler is a direct rewrite of the scheduler found in modern implementations of `perf` [26]. It identifies events whose rates are highly non-linear, and schedules those more frequently since linear interpolation of those values (equivalent to propagation of rates as in §4.5.1) will result in the most error for those counters. It remembers the most recent three measurements of each counter, and computes the distance from the middle point to the linear interpolation between the first and last points.

Counters with no observations are scored at `f64::MAX`, and counters with one or two observations are scored at `f64::MAX / 2.0`. Moreover, observations more than 40 quanta old are removed, essentially providing a "floor" on the sampling rate of any particular counter.

Next, the scheduler scales the priorities by multiplying each counter's score by the duration since its last measurement. Finally, the scheduler sorts the counters by these priorities and selects counters in descending order from that list.

Lim et al. praise its efficiency, noting that the score of each event can be calculated in a handful of instructions. While this is a strong argument in favor of this scheduler in the kernel context (it was integrated into `perf`, after all), our architecture runs the scheduler in userspace with a much longer quantum than `perf`, so these strengths, while real, are not overwhelming.

4.6.3. LLM-Driven Schedulers

The LLM-driven schedulers allow users to express profiling intent in natural language, translating high-level guidance such as "I am interested in memory bottlenecks" into concrete event selection decisions. The schedulers that follow are ordered in terms of increasing complexity and coupling to runtime state. They evolve from a statically-weighted round robin, to a fixed schedule, and finally to a dynamically-refreshed schedule that incorporates estimator feedback.

All schedulers share the same system prompt that frames the model as an expert profiler and explains the goal of producing a high-quality schedule for a given number of physical counters. The user prompt varies between schedulers, but they have some key similarities. The LLM receives a list of all available counters and their descriptions, and an optional guidance prompt. This guidance is a free-form natural-language instruction to the LLM, intended to steer its schedule towards counters that the user is likely to be interested in. The full text of all prompts can be found in Appendix B.

If we encounter a transient network/HTTP error, the request is repeated up to four times with exponential backoff. If the LLM emits a malformed response, we append a message to the interaction indicating that the output was not formatted correctly, the error message itself, and a request for a corrected output. The LLM is allowed three attempts before SACCADE accepts defeat and errors out.

LLMs have well documented reproducibility limitations [44]. These are discussed in §7.1.5.1.

In an attempt to mitigate these concerns, we use Google's open source Gemma 4 model [16] with 26 billion parameters in a mixture-of-experts architecture, activating 4

billion parameters at inference. Queries are directed through OpenRouter to a cloud provider with full prompt and completion logging.

4.6.3.1. Weighted Round Robin. In this scheduler, the LLM emits per-counter weights across the entire event pool at startup. These weights are then processed into a weighted round robin schedule as follows:

(1) Weights are normalized to their greatest common divisor.
(2) The counters are sorted in descending order by their normalized weight.
(3) The scheduler repeatedly iterates through the sorted list, emitting event IDs with nonzero weight and decrementing each nonzero weight by one, until all weights are zero.
(4) The resulting IDs are grouped into batches that saturate the hardware counters with each batch representing one quantum in the schedule, which is replayed indefinitely.

This weighting is purely static. The LLM is never called again after startup and is never exposed to the live counter estimates.

4.6.3.2. Static Schedule. Instead of emitting weights, the LLM-guided static scheduler emits a full static schedule. The LLM may choose to use a non-constant scheduling quantum, group events in any fashion, or even omit events entirely from the schedule. The only restrictions on the schedule are that it cannot be empty and each schedule decision must fully utilize the available hardware counters. This additional expressiveness allows the LLM to tailor the composition and cadence of the schedule in ways that a weight-based approach cannot.

4.6.3.3. Dynamic Schedule. The LLM-guided dynamic scheduler initially generates its schedule at startup identically to the static scheduler. Unlike the static scheduler, it evolves the schedule over the course of SACCADE's execution.

Schedule regeneration is triggered every k (configurable) scheduling quanta. When a new schedule is requested, the LLM is provided a list of available counters, the most recent observations of each counter drawn from the estimator's current belief, the current schedule, and the user-provided guidance. The LLM runs in the background and eventually returns an updated schedule, which replaces the existing schedule for future scheduling decisions. If the LLM fails to emit a well-formed schedule, even after retries, SACCADE logs the error and continues running with the existing schedule until a new schedule is requested.

LLM inference is computationally expensive and time-consuming, so we enforce that there can only be one outstanding schedule request at a time. This guards against the case where k is set too small and a schedule is requested before the previous schedule has been prepared.

4.6.3.4. Reasoning Schedulers. Both the static LLM scheduler and dynamic LLM scheduler have counterparts that support reasoning. Before generating a schedule, the model is asked to analyze the counters in free-form prose. Then, that prose is replayed as an assistant turn during the schedule generation so that the structured output is conditioned on the model's own reasoning.

The prompts themselves can be found in §B.2.5.

4.7. Simulation Infrastructure

```rust
pub trait SampleSource {
    /// Collect all raw samples since the last call.
    ///
    /// Returns `(samples, elapsed_ns)` where `elapsed_ns` is the wall-clock
    /// time covered by this collection window.
    fn collect(&mut self) -> (Vec<RawSample>, u64);

    /// Switch which hardware events are being monitored.
    fn apply_schedule(&mut self, old_set: &[EventId], new_set: &[EventId])
        -> Result<(), Box<dyn Error>>;
}
```

Figure 4.4. Sample Source interface.

SACCADE's modular architecture allows for the estimator/scheduler stack to be applied to any stream through the `SampleSource` interface (see Figure 4.4). In normal operation, samples come directly from hardware counters and schedule decisions are applied to those hardware counters. We can also, however, read samples from and apply decisions to a virtual sample source. This allows us to evaluate the estimators and schedulers quickly and in parallel, without interacting directly with hardware counters.

To support this use case, SACCADE provides the `sweep` and `simulate` commands. The former runs a program multiple times, rotating which events are being tracked in every run, to collect the complete counter data for that workload. The latter reads that counter data and replays it through the estimation and scheduling pipeline.

4.7.1. `sweep`

Deterministic programs produce consistent counter rates across runs. We take advantage of this to build a complete picture of a program's counter rates.

We run the target repeatedly with different sets of five counters each run, reserving the sixth slot for `instructions_retired` as a fixed anchor. On our machine, this

requires 45 runs. Then, we normalize all runs against `instructions_retired`, synchronize their timestamps, and aggregate them into a single trace.

This normalization mechanism was introduced to improve repeatability and lower the noise floor of our evaluations. On small, deterministic workloads, introducing normalization reduced run-to-run variance by as much as 20× (as measured following the methodology in §6.2.1).

We selected `instructions_retired` as the anchor counter because it is stable, always increasing[3], and represents the overall speed of the processor itself. Moreover, for any given deterministic workload, we expect a near-identical number of instructions to be executed. In general, most counters scale linearly with `instructions_retired`.

4.7.2. `simulate`

To simulate an event trace, we begin by re-anchoring whatever timestamps are in the provided trace to start at $t = 0$, and calculate the number of scheduler quanta required to fully simulate the trace. We then construct a `VirtualSampleSource` from the provided trace data and use that to feed samples through the estimator/scheduler pipeline as they were observed in the real hardware.

The `VirtualSampleSource` has an active set of counters, just like the real hardware. It also maintains an internal simulation clock which will be used to index into the recorded trace. When its `collect` method is called, for each counter in the active set, it finds the window of the trace associated with that counter and emits samples at a

[3]We do handle the case where a thread migrates to a new core and its counters change out underneath it.

configurable sample rate for the length of that window. If a sample point falls between two records in the trace, its value is linearly interpolated.

Once all samples have been collected for all active counters, the internal clock is advanced by the scheduling quantum. This continues until the trace has been exhausted.

Because `VirtualSampleSource` implements `SampleSource` directly, the scheduler and estimator code runs without modification. The final output, whether it is a trace, a binned matrix, or a raw CSV of counter values, is written by the existing infrastructure.

The `sweep` mechanism forms the foundation of all of the evaluation performed in §6.3 and is the source for the Kalman Filter's training data. `simulate` allows any scheduler/estimator combination to be evaluated against known ground truth at high speed, without raw hardware access or rerunning the target program.

SACCADE is a decoupled system. The sampling mechanism, state estimator, scheduler, and output format can all be modified independently. Moreover, the separation between mechanism and policy allows for schedulers and estimators to be evaluated against ground truth without hardware access, in parallel. The abstraction boundaries between the scheduler and the estimator enable these two critical components to be considered separately in a 2D design space. These features make SACCADE more than just a profiler; it is a framework for *studying* profilers.

CHAPTER 5

Implementation

Extensive code-level documentation of SACCADE is available online [37], along with the codebase itself [38].[1] Instead of providing a thorough tour of the codebase, this chapter will focus on interesting decisions, pain points, and insights that arose as SACCADE was implemented.

5.1. Working With eBPF

Like a kernel module, eBPF code runs in the kernel, but unlike kernel modules, is validated for safety by the kernel at runtime [19]. This has allowed us to develop SACCADE's kernel-side code confidently, since we know that any mistake will not bring down the whole machine.

5.1.1. Rust Skeleton Generation

One challenge with eBPF is that its programs must be compiled to a special bytecode and loaded at runtime, and communication between the userspace program and the eBPF program is handled through specialized data structures in shared memory. In SACCADE, this complexity is handled by the `libbpf-rs` and `libbpf-cargo` crates [41, 40]. When `cargo` is invoked, a special build script runs that invokes `llvm` on the eBPF source code and generates eBPF bytecode. At the same time, a Rust source code file

[1] See the code at `https://github.com/liam-strand/saccade` and documentation at `https://liam-strand.github.io/saccade`.

is generated with convenient interfaces that can be used to interact with the eBPF program.

These interfaces are extremely helpful, but imperfect, as they must translate between Rust's strict, ownership-oriented memory model and C's more relaxed model. This forced us to turn to the Rustonomicon [43], which contains a wide variety of tips and tricks for interacting with unsafe Rust.

Luckily, our exposure is limited. Once the eBPF program is started, the Rust-side manager needs to have a pool of memory that will not be deallocated for the remainder of SACCADE's runtime. We accomplish this by putting a slab of memory on the heap and intentionally "leaking" it, giving it a `'static` lifetime, which is safe as the slab can only be deallocated once SACCADE exits. When the manager opens and loads the eBPF program, the shared memory data structures will end up in that memory, exposing them to the Rust side of the interface.

When we eventually pull raw samples from the eBPF ring buffer, we are forced to perform an unsafe cast into the Rust type. While this is not ideal, we defend ourselves by generating the Rust type at compile time from the associated C type using `bindgen` [42]. This guarantees that both types have identical memory layouts, so the unsafe cast is valid.[2]

[2]A runtime-checked serialization/deserialization routine could also work here, but we would then be required to enforce this contract in both Rust and C, and pay a performance penalty. This mechanism is faster and more resilient because the contract is automatically enforced at compile-time.

5.1.2. The Verifier

When eBPF code is loaded into the kernel, it is checked for a wide range of bugs and vulnerabilities. For example, the verifier rejects *any* array accesses without a bounds check. This forces the developer to program very defensively which is a change from the permissive style used by many experienced C developers. Moreover, verifier errors are often cryptic, and since the verifier only runs at runtime (not compile time), the errors are only surfaced at the last possible moment.

While developing SACCADE, we found that wrapping the bounds checks in an inlined function was a convenient way to separate business logic from the hand-waving required to appease the verifier. By forcing us to formalize separation of concerns, the eBPF verifier was a helpful critic.

Another important limitation imposed by the eBPF verifier is that the program must terminate. It performs conservative analysis on loops and recursive calls, limiting the overall complexity of the code that we can write. This prohibits us from including any heavyweight libraries.

5.2. Hardware PMU Management

Interacting with raw hardware resources is never straightforward. The `perf` subsystem offers abstractions to manage the PMU, but they are not designed to provide the access that is needed for SACCADE. The kernel provides neither the ability to directly measure counter rates nor to swap counters on command.

5.2.1. Enforcing Atomic Swaps

Since we are interacting with the PMU through the `perf` subsystem, the only option available to us is opening a new counter with `perf_event_open` and closing the old one by dropping its file descriptor. This is a time-consuming process involving two system calls; we quantify the latency in §6.2.3. Moreover, the state of the counter itself between issuing the syscall to make the new event counter and returning from the syscall to close the old event counter is unobservable. So, any events that occur between those two times can't be accurately attributed to either the incoming or outgoing counter. This could be a major problem if the outgoing counter is at a high rate (`instructions_retired`, for example) and the incoming counter is at a low rate (like `bp_snp_re_sync`[3]), and events from the former are attributed to the latter. Our only option is to stop collecting samples during the changeover.

To support this, we implemented a world-stop protocol for SACCADE. The user code initiates by setting a global flag to request the world-stop. It then waits for acknowledgments from each core-local eBPF sampler. Then, it closes and opens the necessary file descriptors before unsetting the global flag, allowing the eBPF samplers to continue. When the eBPF samplers resume sampling, their first sample is a `RESUME` sample as described below in §5.2.2.

This procedure prevents SACCADE from attributing events to the wrong counters, keeping the sample stream accurate so that the state estimator and scheduler can rely on it.

[3]This event is uniquely described as "highly unlikely" by `perf`.

5.2.2. Measuring Rates from Counts

PMU event counters are monotonically-increasing absolute values, but rates are the meaningful metric from a performance engineering perspective. The naive technique to convert absolute counts into rates is to measure deltas between successive samples of each counter. This is generally a good strategy and it is what SACCADE uses the majority of the time. We maintain a per-CPU, per-slot "previous values" vector in userspace that is used to compute event deltas/rates.

This mechanism breaks down when we introduce dynamic reconfiguration in two ways. First, the apparent value of a counter slot is no longer monotonically-increasing. When we switch from one event counter to another, it is possible that the second counter's absolute value will be lower than the first. This could result in a nonsensical negative rate if we attribute measurements before and after the swap to the same counter. Second, we need to ignore any events that occurred between when a counter physically becomes active and when we restart the eBPF sampler. If we didn't account for this, we would have an erroneously high rate because we'd be including events that occurred before the sampling window.

We manage this challenge using the `RESUME` samples (see §4.2.1), which are emitted when the target thread is context-switched in and on the first sampling interval when the eBPF sampler returns from a world-stop as described in §5.2.1. When the userspace code receives a `RESUME` sample, it uses it to reset the baseline in the "previous values" vector and does not process the sample as a rate measurement.

These mechanisms allow us to read the reported counter rates with confidence, despite all of the complex machinery.

5.3. Kalman Filter Configuration

The performance of a Kalman Filter depends heavily on its configuration. Without meaningful non-diagonal entries in the process noise covariance matrix, its performance essentially mirrors that of the EMA estimator, but with higher computational complexity. In the field of robotics, where the Kalman Filter originates, the dimensionality of the data is often relatively small, so it is possible to hand-tune the various configuration parameters until the performance is satisfactory. Our system has more than 220 dimensions, meaning hand-tuning is out of the question.

Instead, we analyze trace data to extract sensible configuration parameters. But, since we can only observe six dimensions at a time, collecting and generating insights from this trace data is no small task.

5.3.1. Analytical Correlation

We use SACCADE's `sweep` subcommand (described in §4.7) to collect trace data. Instead of outputting to a perfetto trace for replay with the `simulate` command, we generate a binned sample matrix. This is necessary because perfetto's event-based format is a bad fit for matrix operations, and there is small run-to-run variance in sample rates that we do not want to have to deal with. The binned matrix is populated by subdividing the total execution time of the target program into quanta, and taking the average of all rate samples that occurred within that quantum. We scale the matrix's quantum size such that usually one or two samples occur within a quantum.

We begin by calculating the Pearson correlation r_{ij} between all pairs of event counters i and j in the dataset. Given their values $x_{i,t}$ and $x_{j,t}$ at time t, with means $\bar{x}_i$ and

```
M   = ~np.isnan(rates)
Mf  = M.astype(np.float64)
R   = np.where(M, rates, 0.0).astype(np.float64)
R2  = R * R
n   = Mf @ Mf.T   # [i,j] = co-observed timestep count
si  = R  @ Mf.T   # [i,j] = Σ R[i,t] where both valid
sij = R  @ R.T    # [i,j] = Σ R[i,t]*R[j,t]
sii = R2 @ Mf.T   # [i,j] = Σ R[i,t]² where both valid
sj  = si.T        # symmetric over joint validity
sjj = sii.T       # symmetric over joint validity
```

Figure 5.1. The Pearson accumulators. `rates` is the binned sample matrix. Cells with no samples have a value of `NaN`.

$\bar{x}_j$ over t, we have the standard formula:

$$r_{ij} = \frac{\sum_t (x_{i,t} - \bar{x}_i)(x_{j,t} - \bar{x}_j)}{\sqrt{\sum_t (x_{i,t} - \bar{x}_i)^2 \cdot \sum_t (x_{j,t} - \bar{x}_j)^2}}$$

which can be rearranged to form

$$r_{ij} = \frac{n \cdot \Sigma_{x_i x_j} - \Sigma_{x_i} \cdot \Sigma_{x_j}}{\sqrt{(n\Sigma_{x_i x_i} - \Sigma_{x_i}^2)(n\Sigma_{x_j x_j} - \Sigma_{x_j}^2)}} \tag{5.1}$$

where there are n samples.

This second formulation is especially convenient because it works with raw sums and can be computed in a single pass across four accumulators, all $k \times k$ matrices, where k is the number of event counters that can be measured, indexed by the event IDs i and j. Each entry accumulates only over timestamps t where both counters i and j are valid. The accumulation code can be found in Figure 5.1. We have `n`, the count of jointly valid timestamps; `si`, the sum of $x_{i,t}$; `sij`, the sum of $x_{i,t}x_{j,t}$; and `sii`, the sum of $x_{i,t}^2$. We also need `sj` and `sjj`, but those are trivially computed by transposing `si` and `sii`, respectively, by symmetry of joint validity.

To avoid conditional logic, we compute the matrix `M_f`, which has a row for each event counter and a column for each sample bin, where the value is `1.0` if the bin has samples or `0.0` otherwise. This is used to compute `R`, which takes `rates` (the raw binned samples) and replaces all of the `NaN` values with `0.0`. This way, they will never be included in the accumulations, but we don't need any conditional logic to avoid them. The accumulators themselves are then computed with a series of matrix multiplications.

From there, it is a simple matter of applying Equation (5.1) to calculate the raw Pearson correlation scores. All correlations within $[-0.1, 0.1]$ are clamped to 0.0 to prevent low-confidence correlations from polluting the Kalman Filter.

A valid correlation matrix must be PSD and numerical instability (and the clamping we just did) can potentially break that property. So, our final step is to use eigenvalue clipping to find the nearest PSD matrix, which contains our correlation scores.

5.3.2. "Expert" System

While the analytical method of extracting correlations from trace data is effective, it has a blind spot. Since the traces are collected in batches across multiple runs, the individual traces must be aligned to produce the full trace. This alignment can introduce jitter that attenuates cross-batch temporal correlation, even for architecturally-coupled events. Our normalization against `instructions_retired` fixes level differences but not phase differences across runs.

We can sidestep the jitter problem entirely by collapsing the longitudinal trace data from each benchmark into a scalar mean before running correlation analysis. Now we are correlating benchmark-level responses rather than within-run temporal variation.

The tradeoff is that instead of many thousands of datapoints per event counter, we now have eight. We therefore must raise our significance threshold from 0.1 to 0.9, which is more appropriate for such a small n.

We can also integrate the knowledge of an LLM as a correlator of last resort. The AMD Zen 4 architecture has several alias events, which have different names but observe the same physical paths. If, by chance, these events don't occur in the same batch, they may be missed by the direct Pearson correlation. Also, very rare or sparse events may not accumulate enough datapoints to overcome the noise floor of the system. So, we used an LLM, prompted with the list of event names and their descriptions, to generate a table of events that are likely to be correlated, along with correlation scores. This table was independently verified by the author and correlation scores are intentionally conservative.

These two methods are applied to the trace dataset in a hierarchical order. The highest priority is given to correlations identified by the method described in §5.3.1, with a special emphasis placed on correlations that were identified between event counters that ran in the same batch. Next, the benchmark-level correlations are merged, filling in weakly correlated pairs that were not recorded in the same batch. The expert priors fill in only the gaps that remain. Critically, lower-priority methods can't weaken a pair that the data has already measured confidently.

Finally, we use an iterative PSD projection algorithm to ensure the final correlation matrix is amenable to use in the Kalman Filter. For up to 20 iterations, we first project the matrix to be PSD, then clamp any unspecified pairs (those that came from neither

data, nor expert priors) to 0.0 covariance, make the matrix symmetrical, and set the diagonal to 1.0. This repeats until the fixup steps are negligible.

5.4. LLM Integration

Interacting with an LLM is inherently a high-latency and nondeterministic operation. SACCADE, on the other hand, runs a tight scheduling loop with timing constraints. Integrating these two systems is challenging because the scheduler operates with fixed quanta and cannot block for long periods of time waiting for a network call or transformer inference.

5.4.1. Latency Hiding

Unlike the other LLM-driven schedulers, the dynamic scheduler (§4.6.3.3) is able to execute LLM queries mid-run, not just at startup. This presents a serious challenge from a latency management perspective as the request for a new schedule must traverse the network to a cloud provider, run the inference itself, and wait for the response to come back over the network.

To hide this latency, we manage these mid-run LLM updates from a background thread. The schedule itself lives behind a mutex, so it can be updated asynchronously as the LLM responds. Instead of dealing with making a remote request, the scheduler thread sends a message containing the current counter state to the LLM management thread, which then constructs the prompt, sends it to the LLM, handles retry logic, and places the new schedule in the shared location. In the meantime, the previous schedule remains in place. Additionally, the LLM management thread allows for only

one outstanding schedule request at a time; subsequent requests are dropped until the previous request has been fulfilled. Critically, the scheduling thread itself is never blocked by the network or the LLM itself.

In simulation, we rely on a latency model to determine how long a network-based LLM query should appear to have taken. This latency model is the 8 billion parameter formulation of Gemma 4, which is the largest model that can run stably on our local hardware. We characterize the latency of each kind of query we make (static schedule generation, dynamic updates, reasoning, etc.) separately as a distribution. Then, when it comes time to make the query to a cloud resource, we sample that distribution to determine how far to advance the simulation before applying the new schedule.

The goal is to simulate the latency of the larger model running on more powerful local hardware. We believe this is a reasonable approximation of that performance because both the 8 billion and 26 billion parameter models have very similar architectures and each only activate 4 billion parameters at inference time. Furthermore, we observed that when the larger model did successfully run on our hardware, it had similar latency characteristics to the smaller model.

5.4.2. Structured Output and Retry Logic

Not only are LLMs slow compared to our scheduling loop, they are also nondeterministic. This can bite us in two ways across all LLM-driven schedulers. First, the LLM can produce output that is malformed and cannot be parsed into a usable schedule. Second, results may vary across identical prompts.

The second problem is a limitation of this architecture. We attempt to put some bounds on this nondeterminism with our evaluation methodology (§6.1), but it is impossible to avoid entirely.

The first problem can be addressed with some clever prompting and retry logic. To give the LLM a fighting chance, we restrict its output to a strict JSON schema. This guarantees that we will receive valid JSON, each step holds exactly the right number of counter IDs, and those IDs are restricted to an enumerated set so the model cannot hallucinate IDs (see §B.2.6). This strategy is usually successful.

For the cases where the LLM does not follow our instructions and the emitted schedule is genuinely unusable, we implement retry logic. We read through the parse error to generate a terse but informative message that can be passed along to the LLM. Then, this message is *appended* to the existing conversation and it is asked to regenerate the schedule. We do not include the malformed schedule in our user message because we do not want to pollute the context window with examples of what *not* to do lest the LLM become confused between the likely very similar schedule examples. Moreover, the LLM can see the message in the chat because the context is formed from the concatenation of both sides of the conversation. We would not be providing it with anything it didn't already have.

Implementing SACCADE was a significant engineering challenge. Each layer in the stack, from the PMU hardware, to the eBPF sampler, to the Kalman Filter estimator, to the LLM-driven schedulers, depends on the layer below it to be correct and efficient. Having examined the key implementation challenges in the estimators, schedulers, and

the system connecting them, we now turn to evaluating their performance and effectiveness.

CHAPTER 6

Evaluation

6.1. Methodology

6.1.1. Testbed

Evaluations are performed on a Dell PowerEdge R7615 running an AMD EPYC 9124 with 32GB of memory. The machine is running Linux kernel version 5.15.0 underneath Ubuntu 22.04.5 LTS. SACCADE itself is implemented in the 2024 edition of Rust, version 1.93.1. Local LLM latency collection was performed on a Nvidia A30X.

For the duration of all runs that interact with the hardware, our evaluations had exclusive use of the machine. Although our simulation infrastructure is broadly deterministic, anything that interacts with the LLM introduces enough uncertainty that repeated runs are necessary. Any run that invokes an LLM-driven scheduler is evaluated at least three times and their resulting performance metrics are averaged.

6.1.2. Workloads

Our workloads are selected from the NAS Parallel Benchmarks (NPB) [10] and the SPEC CPU2017 [11] suite. We select 9 of the NPB workloads and 5 of the SPEC workloads for our use. The NPB workloads are sized to be “small but meaningful”, meaning that they should run to completion within about a minute, representing interactive jobs.

Workload	Size	Runtime	Performance characteristics
Evaluation workloads			
NPB IS	Class C	17 s	Integer bucket sort; random memory access, low FLOP, memory-latency bound
NPB LU	Class B	78 s	LU triangular solver; compute-intensive with regular, cache-friendly access
NPB SP	Class B	66 s	Scalar pentadiagonal solver; balanced compute and memory traffic
NPB UA	Class B	45 s	Unstructured adaptive mesh; irregular, dynamically changing access patterns
SPEC deepsjeng_r	ref	246 s	Alpha–beta chess tree search; branch-heavy integer, deep speculation
SPEC imagick_r	ref	370 s	Image processing (ImageMagick); floating-point, SIMD-friendly, compute-bound
KF training workloads			
NPB BT	Class A	35 s	Block tridiagonal solver; compute-intensive with strong cache reuse
NPB CG	Class B	34 s	Conjugate gradient; irregular sparse mat-vec, memory-latency bound
NPB EP	Class A	4 s	Embarrassingly parallel RNG; pure FP compute, minimal memory, branch-light
NPB FT	Class B	18 s	3D FFT; all-to-alh bound
NPB MG	Class C	31 s	Multigrid V-cycle; hierarchical multi-resolution memory access
SPEC mcf_r	ref	276 s	Combinatorial route optimization; heavy pointer chasing, cache-miss / latency bound (integer)
SPEC lbm_r	ref	169 s	Lattice Boltzmann bandwidth bound
SPEC exchange2_r	ref	195 s	Recursive Sudoku solver; branch- and compute-bound integer, fits in cache

Table 6.1. Workloads used in evaluation (top) and KF training (bottom). Runtimes are wall-clock durations measured from the collected traces.

The SPEC workloads are, in contrast, left at their default scales to represent large, long-running applications.

We aim to select workloads that generate as wide a range of performance events as possible; including memory, integer compute, floating-point compute, cache hierarchy,

and microarchitectural behaviors. All workloads are single threaded. The full list of workloads can be found in Table 6.1.

6.2. System Performance

We will begin by evaluating the performance of SACCADE itself.

6.2.1. Noise Floor

Before we can make any pronouncements about the relative performance of different schedulers and estimators, we must first determine how much noise is introduced by SACCADE's simulation infrastructure. We do this by collecting ground-truth data on a workload twice using SWEEP, and then using EVALUATE to compute the difference between them. In a perfect world, this would result in a delta of exactly zero.

Our standard performance metric going forward is going to be normalized root-mean-squared error (nRMSE). This normalization is against the mean rate of the underlying signal. So, we accept more absolute noise on a high-frequency counter than we would a low-frequency counter. This makes sense in terms of performance counters because their frequencies span many orders of magnitude.

When targeting NPB CG, SACCADE has a noise floor of between 0.02 and 0.03 nRMSE. So, any accuracy result below this band can be attributed to measurement noise and is not meaningful. That being said, this result does suggest that SACCADE's simulation infrastructure can record and replay the full set of performance counters of a deterministic workload within 2-3% of their true values.

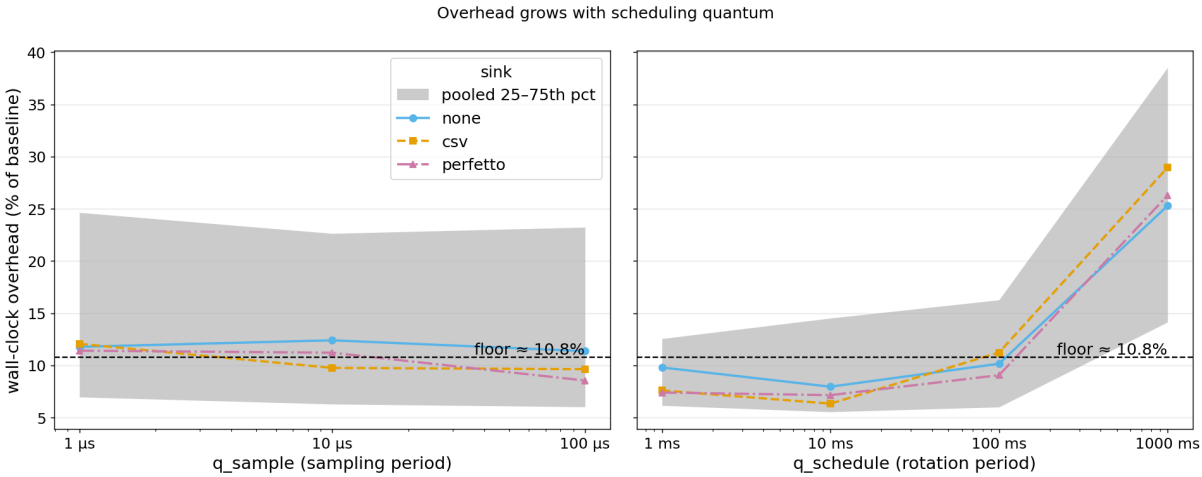


Figure 6.1. Overhead (% of baseline) sweeping the sampling period `q_sample` (left) and the rotation period `q_schedule` (right), each pooled over the other knob. Lines are per-sink medians, the shaded band the pooled 25–75th percentile, and the dashed line the overall ≈10.8% floor. Overhead is flat to within noise ($\approx \pm 1.5$ pp, no monotonic trend) across sampling period. We see a dramatic increase in overhead with a *lower* scheduling frequency.

6.2.2. Runtime Overhead

To understand how much SACCADE imposes on the underlying program we intend to profile, we measure the wall-clock overhead of running a program under SACCADE compared to running the program by itself. We sweep the two timing parameters that govern SACCADE's behavior, the counter-rotation period `q_schedule` and the sampling period `q_sample`, on logarithmic scales. These configurations are crossed with the three available output sinks (none, CSV, and perfetto), for 36 total configurations. Each configuration is run 10 times, and an unprofiled baseline is interleaved with the rest. These 46 configurations are then shuffled to avoid confounding factors like thermal or load drift. All configurations target the NPB UA workload at size A.

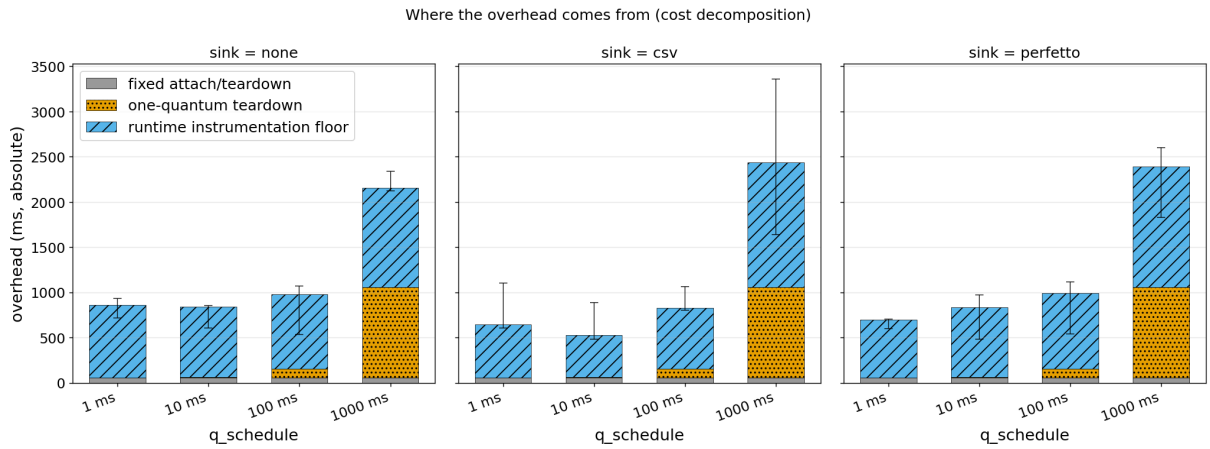


Figure 6.2. Saccade's runtime overhead decomposed into additive layers (absolute ms), per output sink, against the scheduling period `q_schedule`. Each bar splits the total into a fixed attach/teardown cost (≈49 ms), a one-quantum teardown term (≈1×`q_schedule`, most prominent at 1000 ms), and the dominant *runtime instrumentation floor*. Startup fits ≈ 49 ms + 1.00×`q_schedule`; error caps span the min–max across `q_sample`. The floor dominates the fixed startup and per-quantum terms at every setting.

SACCADE's overhead is small and insensitive to configuration, unless a high scheduling quantum is selected. Figure 6.1 sweeps each timing parameter while pooling over the other. With respect to sampling period, overhead remains a flat floor of roughly 10.8% of the baseline, varying no more than a few percent. The choice of sink is likewise negligible. Counterintuitively, we see that very infrequent schedule changes produce a very *high* overhead. This is the opposite of what we would expect, but it does show that our design never blocks the execution of the target process by scheduling or sampling. If the scheduling or sampling were blocking operations, we would see a strong correlation between sample/schedule frequency and overhead. We see no correlation in sampling, and an inverse correlation in scheduling.

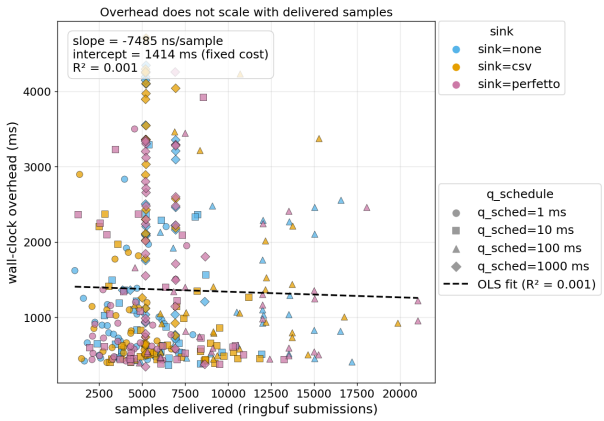


Figure 6.3. Per-run wall-clock overhead versus the number of samples actually delivered (eBPF ring-buffer submissions), coloured by sink and marked by `q_schedule`. Although the delivered count varies 13× across configurations, overhead is uncorrelated with it: the per-sample marginal cost is negligible beside a fixed instrumentation floor.

Figure 6.2 decomposes the overhead into additive components. To separate the fixed startup and teardown costs from the costs that accumulate over the course of SACCADE's execution, we additionally profiled `/bin/true`, which executes instantly. We find that the startup cost is roughly 49 ms of setup plus one unit of `q_schedule`. Everything above those fixed terms is a cost associated with instrumentation at runtime.

The flatness of this floor is surprising. It is reasonable to expect that the profiler's overhead will grow with how much work it does, and SACCADE certainly does more work with a small `q_sample`. To test this directly, we instrumented SACCADE's eBPF program to count the number of samples it actually delivers to userspace and recorded this for every run. Figure 6.3 plots per-run overhead against the delivered count. Although the count varies by a factor of 13 across the configurations, overhead is essentially uncorrelated with it. The marginal cost of an individual sample is lost in the noise; the overhead is not a sampling cost at all.

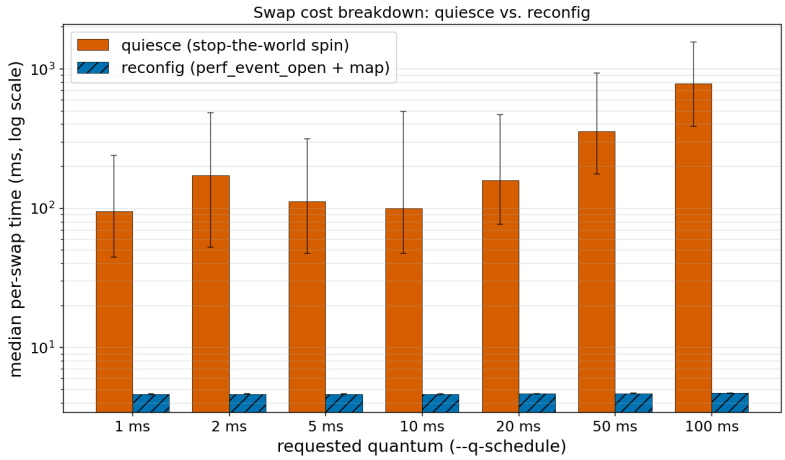


Figure 6.4. Where the slot-swap time goes: the stop-the-world *quiesce* spin versus true counter *reconfiguration* (bars are medians, whiskers the p25–p75 spread). Reconfiguration is flat and tight at $\approx$2.8 ms regardless of quantum ($\approx$ 0.7 ms per slot). Quiesce dominates at 95–99% of the cost, grows at larger quanta, and is heavily right-tailed (IQR exceeds the median at every point).

SACCADE's overhead is largely attributed to an always-on instrumentation cost of roughly 10.8%, with additional latency imposed by long scheduler quanta. The practical consequence is that SACCADE's sample frequency parameter can be chosen purely for analysis quality, without worrying about wall-clock penalties.

6.2.3. Slot-Swap Cost

Since SACCADE pauses sample collection while counters are being reconfigured, it is important to understand where the slot-swap latency comes from. We measured 8,368 real swaps across 7 quanta from 1–100 ms in shuffled order; each swap reconfigures 4 slots.

Figure 6.4 decomposes this cost. The counter reconfiguration work (the calls to `perf_event_open` and updating eBPF maps) is a small, quantum-independent constant

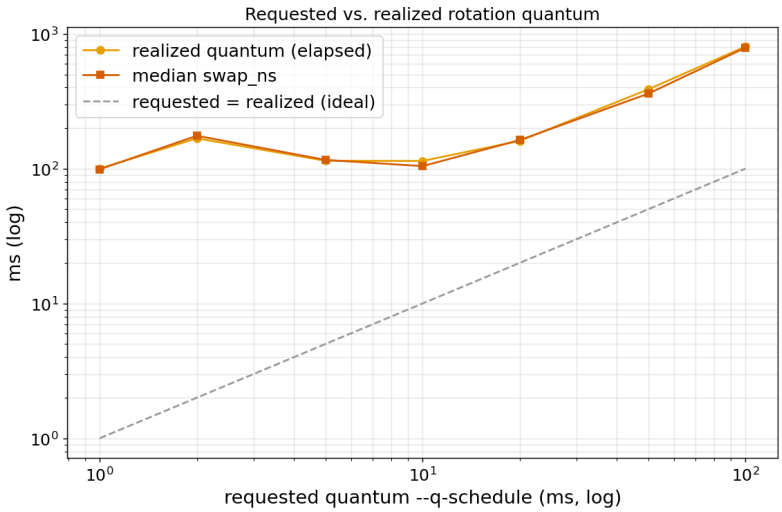


Figure 6.5. Requested versus realized quantum (log–log). Median swap latency tracks the realized quantum almost exactly, and both sit far above the ideal requested = realized line. The gap is the rotation-period floor imposed by the swap: because each rotation must wait roughly one quantum for in-flight counters to update and reset, the counters cannot be rotated faster than roughly once per quantum.

of roughly 2.8 ms. The apparent "swap cost" is actually dominated by the stop-the-world barrier and quiesce. This synchronization cost accounts for 95–99% of the total.

As it stands, SACCADE is unable to deliver a requested scheduling quanta. Figure 6.5 shows that the realized scheduling quantum is consistently far greater (up to 100×) than the requested quantum. But, this quantum can be nearly entirely attributed to the cost of synchronization, not swapping the counters or scheduler business logic.

6.2.4. LLM Latency

Two features stand out. First, every call type costs tens of seconds: even the cheapest, `static_setup`, has a median of roughly 44 seconds, and `wrr_setup` runs to a median of 114 seconds. At this scale the LLM cannot be consulted inline on the sampling hot

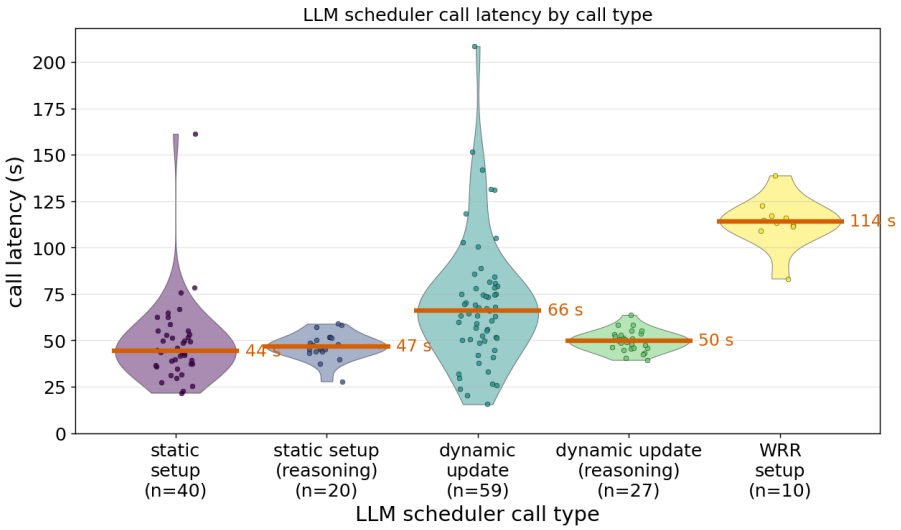


Figure 6.6. Distribution of LLM scheduler call latency by call type. Each violin shows the sample density, individual calls are overlaid as jittered dots, and the red bar marks the median. The `dynamic_update` call exhibits by far the widest spread and a long upper tail, while the reasoning variants and `wrr_setup` are comparatively tight.

path; its decisions must be computed out of band and amortised across many quanta. Second, the spread is strongly call-type dependent. `dynamic_update` is both the most frequent call ($n = 59$) and by far the most variable, with a median near 66 seconds but a long tail reaching past 200 seconds (p95 $\approx$ 108 seconds), a consequence of the growing, variable-length context each periodic reschedule must process. By contrast the free-form reasoning calls (`static_setup_reason`, `dynamic_update_reason`) and the one-shot `wrr_setup` are comparatively tight, their latency set mostly by a fixed-size prompt rather than accumulated state. The practical takeaway is that the dominant cost and the dominant *uncertainty* both come from the periodic dynamic-update path, which is the call worth budgeting for and the one whose tail must be hidden from the profiled workload.

6.3. Space Exploration

Now that we have characterized SACCADE's performance, we will use SACCADE to explore the space formed by the scheduler and estimator axes.

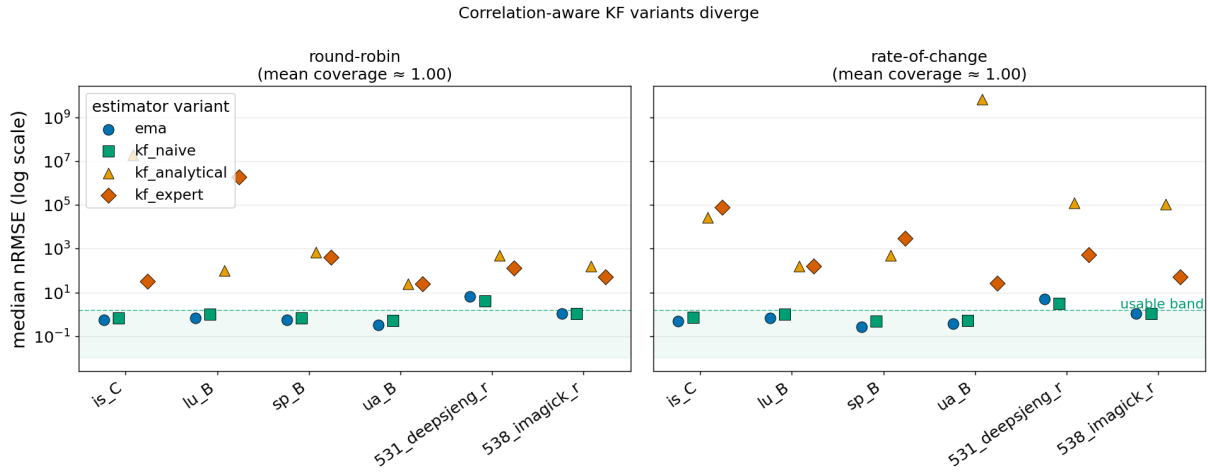
6.3.1. Kalman Filter Configurations

We began by identifying the ideal Kalman Filter configuration; comparing a naive configuration that only sets global hyperparameters with no off-diagonal cross-correlational information, a configuration that additionally incorporated our analytically derived cross-correlation, and finally a configuration that added in the LLM-guided correlations. The Kalman filters were additionally compared against an EMA baseline.

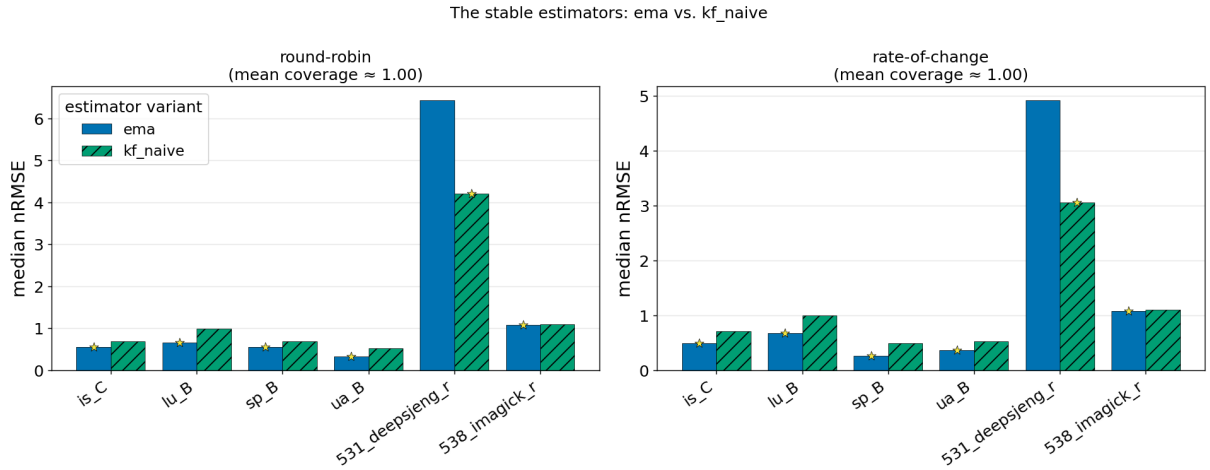
All four of these estimators were paired with the round-robin scheduler and the rate-of-change scheduler to explore how the feedback loop of the rate-of-change scheduler might exacerbate any results seen in the round-robin configuration. We ran each of these 8 setups against the 4 NPB and 2 SPEC evaluation workloads, giving us 48 total runs.

We found that the EMA estimator consistently outperforms the naive Kalman Filter on smaller workloads. A high-variance workload gives a slight edge to the Kalman Filter, as shown in Figure 6.7b.

Unfortunately, the filters with off-diagonal cross-correlation perform exceptionally poorly. Despite repeated attempts to enforce numerical stability and clamp uncertainty, their estimates consistently and quickly diverge from ground truth, resulting in astronomical nRMSE error as shown in Figure 6.7a.



(a) All four variants on a log scale. The correlation-aware variants (`kf_analytical`, `kf_expert`) diverge by orders of magnitude; `ema` and `kf_naive` stay in the usable band.



(b) Zoom on the two stable variants (linear axis). `ema` wins the NPB workloads, `kf_naive` `deepsjeng`, and a tie on `imagick`; neither dominates. ⋆ marks the lower nRMSE.

Figure 6.7. Kalman-filter estimator variants under estimator-independent schedulers (round-robin, rate-of-change), median nRMSE per workload.

A possible explanation for the degenerate behavior is that the Kalman Filter models the counter probability distribution as a linear system of Gaussians. If the true counter

distributions are not approximately Gaussian and approximately linear, the math behind the Kalman Filter starts to break down and we start to see outrageous predictions and extremely high error.

In all future evaluations, the Kalman Filter is used with only its best configuration: global tuning, no a-priori cross-correlation. Unfortunately, even though we don't use the divergent Kalman Filters going forward, the workloads that generated the data that was used to configure them could not be recycled as additional evaluation workloads because of time constraints.

6.3.2. Accuracy of Adaptive Scheduling

Now we put SACCADE to work to evaluate our schedulers and estimators. Unfortunately, we do not see that any one configuration dominates. Instead, the best estimator is coupled to the scheduler.

Figure 6.8 provides a high-level summary of the performance of these configurations. Each row is a scheduler/estimator pair, grouped by scheduler. Absolute nRMSE spans orders of magnitudes, so the color of the heatmap is normalized to the performance of a random scheduler with the basic EMA estimator.

The rows with the EMA estimator tend to be less amber (less error relative to random), performing well compared to the other estimators. This pattern breaks down, however, with the max-uncertainty estimator, which sees an advantage when paired with the Kalman Filter.

In general, we find that while the random scheduler is never the *best* choice, none of the other configurations can consistently outperform it.

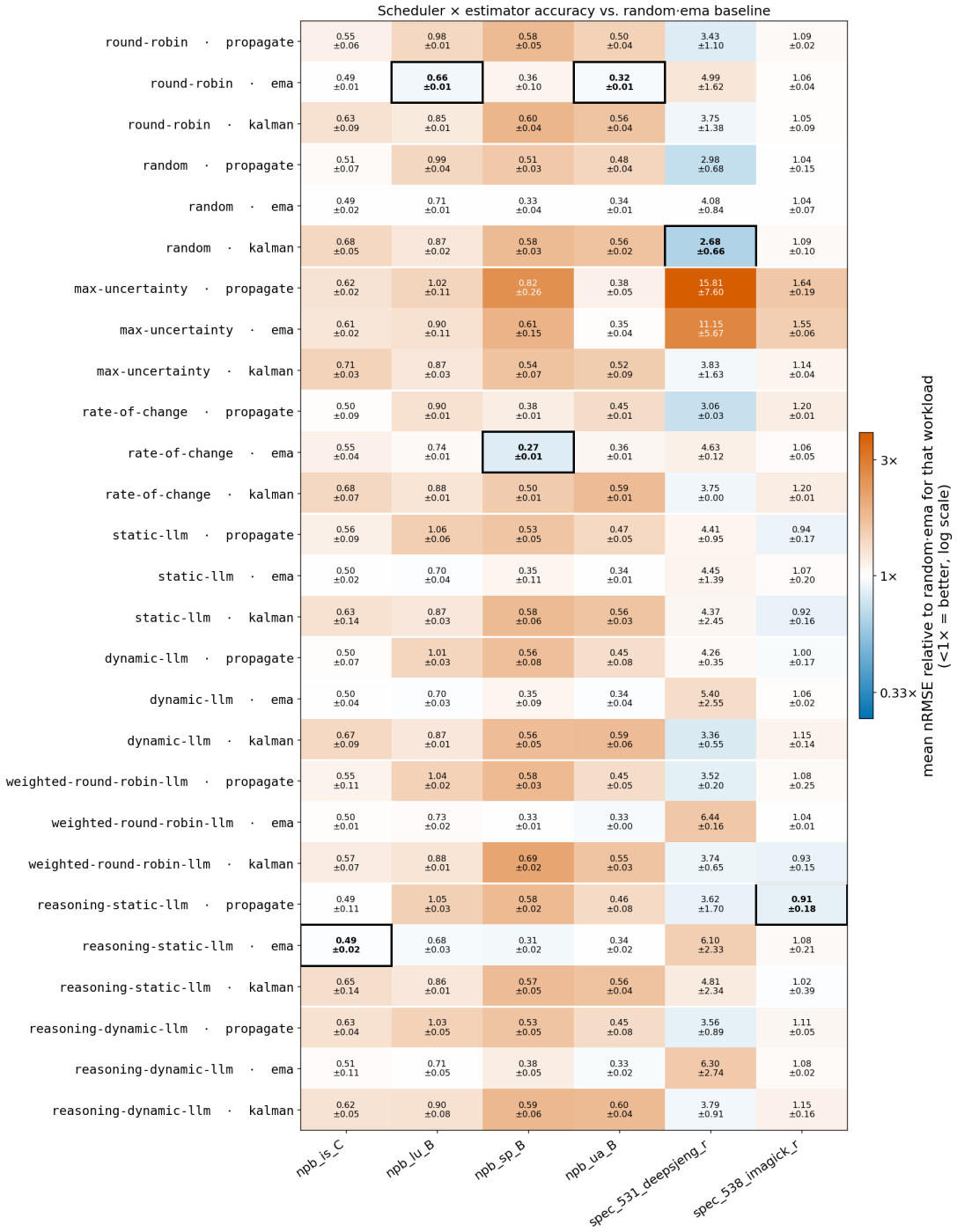


Figure 6.8. Accuracy of all scheduler × estimator combinations across six workloads (cell value = raw median nRMSE; color = performance relative to random scheduler with EMA estimator, log scale; black box = best configuration per workload winner). Rows are grouped by scheduler in blocks of three estimators.

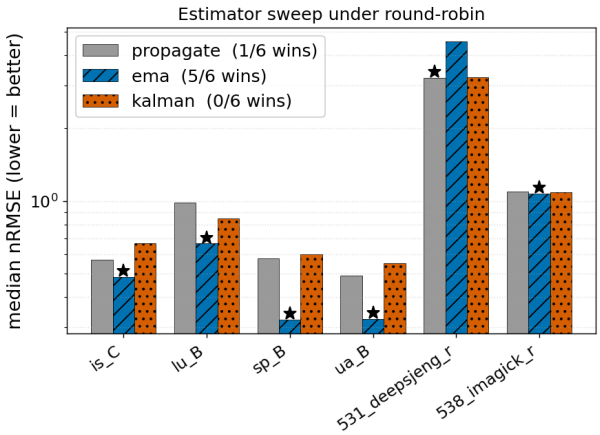


Figure 6.9. Estimator accuracy under the round-robin scheduler across six workloads (median nRMSE, log scale; lower is better; ⋆ marks the best estimator per workload). With an uncertainty-agnostic scheduler, EMA's smoothing wins on 5 of 6 workloads.

We will now take a closer look at three illustrative schedulers: a baseline (uncertainty-agnostic) scheduler, an uncertainty-driven scheduler, and the best overall scheduler.

6.3.2.1. Estimators over Round-Robin Scheduler. The round-robin scheduler samples blindly, never observing the state of the counter estimates. The estimator's only job here is to interpolate between samples, and EMA's smoothing is the best general-purpose interpolator, winning 5/6 workloads. On smooth workloads like `npb_sp_B` or `npb_ua_B`, it shows a 3× reduction in nRMSE (see Figure 6.9). SPEC's `deepsjeng` workload, in contrast, is extremely high-variance, and EMA performs poorly here, smoothing over the spikes.

6.3.2.2. Estimators over Max-Uncertainty Scheduler. If we close the scheduler/estimator loop, the estimator's responsibility grows from merely interpolating samples to steering the scheduler's data collection. Of all of the estimators evaluated here,

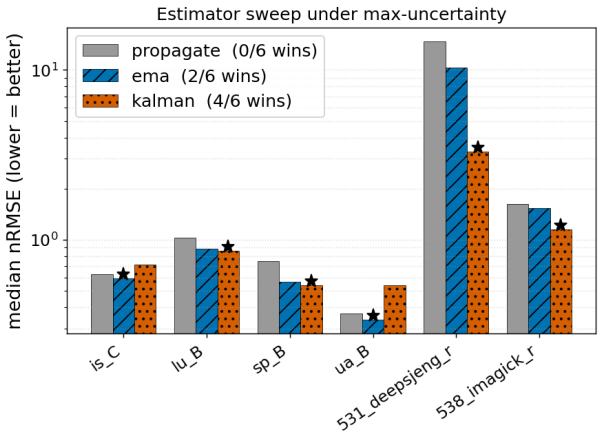


Figure 6.10. Estimator accuracy under the max-uncertainty scheduler (median nRMSE, log scale; lower is better; ⋆ marks the best estimator per workload). Because this scheduler steers sampling on the estimator's uncertainty signal, Kalman (the only estimator whose uncertainty reflects per-event variance, not just staleness) wins on 4 of 6 workloads, most dramatically on `531_deepsjeng_r`.

only the Kalman Filter emits a per-event, variance-aware uncertainty. The propagate and EMA estimators expose only staleness, not variance. Under max-uncertainty, propagate performs poorly as it has no notion of uncertainty and ignores any potential variance or noise as seen in Figure 6.10. Overall, the Kalman Filter performs very well, claiming victory in 4/6 of the workloads, most notably on the challenging `deepsjeng`.

6.3.2.3. Estimators over Dynamic-LLM Scheduler. The Dynamic LLM scheduler does not provide meaningful benefit over a random scheduler. As we might expect at this point, EMA is the best estimator for 5 of 6 workloads, as seen in Figure 6.11.

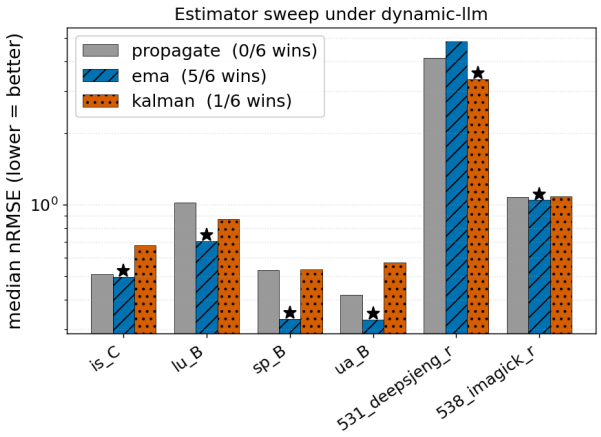


Figure 6.11. Estimator accuracy under the dynamic-LLM scheduler (median nRMSE, log scale; lower is better; ⋆ marks the best estimator per workload). As with most other policies, EMA wins on 5 of 6 workloads.

6.3.3. Simulation and Reality

The preceding sections evaluate every scheduler/estimator pair inside the simulator, replaying a recorded ground-truth sweep and scoring the reconstructed counter trajectories. That methodology isolates estimator quality, but it leaves two questions open. First, does the accuracy ranking obtained in simulation survive when the same configuration drives a *live* profiling run, where counters must be physically reprogrammed on real hardware? Second, how does SACCADE compare against the tool a practitioner would otherwise reach for, namely kernel-native counter multiplexing via `perf stat`?

To answer both, we evaluate five configurations against the same ground-truth sweep traces, scoring each by median nRMSE (lower is better) and by *coverage*, the fraction of ground-truth time bins for which the configuration actually emits an estimate. The two simulated legs (`best`, the per-workload winner from the accuracy grid, and `baseline`, round-robin scheduling with the propagate estimator) are re-evaluated in the simulator;

the two corresponding *real* legs run the identical configuration live through `saccade run` on the real benchmark binary, aggregated over five repetitions; and `perf stat` multiplexes the full event set natively. Results for the two SPEC workloads are shown in Figure 6.12.

We see the coverage metric collapse when transitioning from simulation to real hardware. Simulated runs model counter switches as essentially free and thus see coverage of essentially the entire ground-truth timeline. On real hardware the same configurations cover only 5–10% of the ground-truth time bins. The cause is the per-quantum cost of synchronization analysed in §6.2.3.

This resolution gap is the key caveat for interpreting the upper panel: a live nRMSE is scored over a sparse subset of the timeline, so the difference between a simulated and a live score conflates estimator error with missing temporal resolution. The real and simulated legs are *not* a controlled apples-to-apples comparison of estimator fidelity. Yet, we can still draw interesting conclusions from these data.

First, the move to real hardware has tremendous consequences for all SACCADE configurations. For `deepsjeng` the losses are relatively moderate. But for `imagick`, the degradation is catastrophic and turns what was a useful configuration into little more than noise. Second, comparing like resolution against like, SACCADE cannot yet compete with kernel-native `perf stat`. That being said, for less noisy workloads like `imagick`, the simulated versions of both SACCADE configurations outperform `perf` which suggests that reducing the swap tax could lead to significantly improved performance. Crucially, that swap tax is an artefact of the current implementation rather than a fundamental limit. Opportunities for optimization are discussed in §7.2.

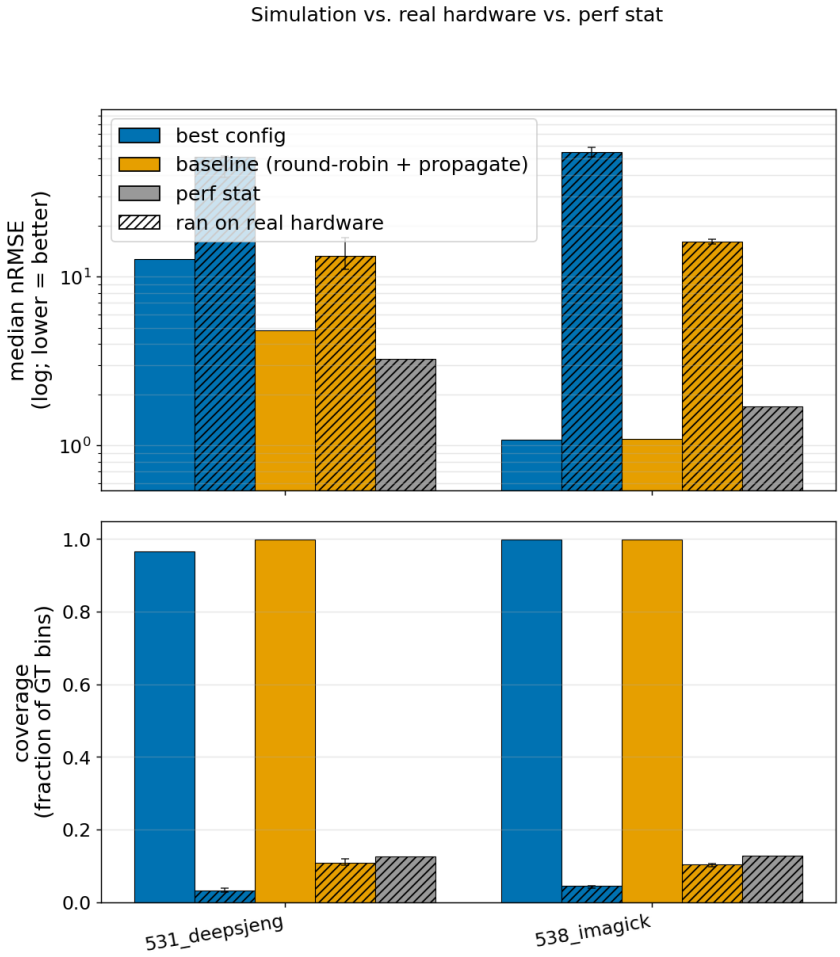


Figure 6.12. Simulated estimator accuracy compared against live `saccade run` and against kernel-native `perf stat`. *Top:* median nRMSE (log scale; lower is better), with p_{25}/p_{75} whiskers on the live legs. *Bottom:* coverage, the fraction of ground-truth time bins carrying an estimate. Bar colour denotes the configuration; the hatched bars ran on real hardware. Every simulated leg covers the timeline almost completely (≈ 1.0), whereas every live and `perf stat` leg collapses to ≈ 0.1. The live nRMSE values must therefore be read together with their coverage rather than against the simulated legs directly.

6.3.4. LLM-Driven Scheduler Guidance

The LLM-driven schedulers (§4.6.3) accept an optional free-text guidance string, intended to steer event selection toward the counters a user actually cares about. A natural question is whether this guidance changes what SACCADE reconstructs, and if so, in which direction. To isolate its effect, we run each of the five LLM-driven schedulers twice on every workload: once with no guidance, and once with a hint that asks the model to "focus on memory hierarchy behavior: cache misses, TLB pressure, and memory bandwidth utilization" and to deprioritize integer-arithmetic counters. Every combination is paired with the exponential-moving-average estimator (§4.5.2) and repeated for three trials, so that the LLM's run-to-run variability is visible rather than hidden. We report the median per-counter nRMSE; lower is better.

Figure 6.13 summarizes the result, and the headline is that guidance is not uniformly beneficial. Its impact is bounded by how much accuracy headroom a workload leaves, and, where headroom exists, its sign depends on the scheduler.

On `538.imagick_r`, every scheduler sits at an nRMSE floor of roughly 1.07, the same floor reached by all estimators in §6.3.2, and guidance moves nothing. The one exception is the reasoning static scheduler, whose unguided run lands at an anomalous 1.70 and whose guided run is pulled back down to the floor; we read this as guidance stabilizing a degenerate schedule rather than as a genuine accuracy gain.

On `531.deepsjeng_r`, which leaves real headroom (unguided nRMSE between three and seven), guidance swings the result in both directions. It is strongly beneficial for the dynamic scheduler (-63%) and the reasoning static scheduler (-59%), both improvements that clear the trial-to-trial noise band. It is strongly harmful for the weighted

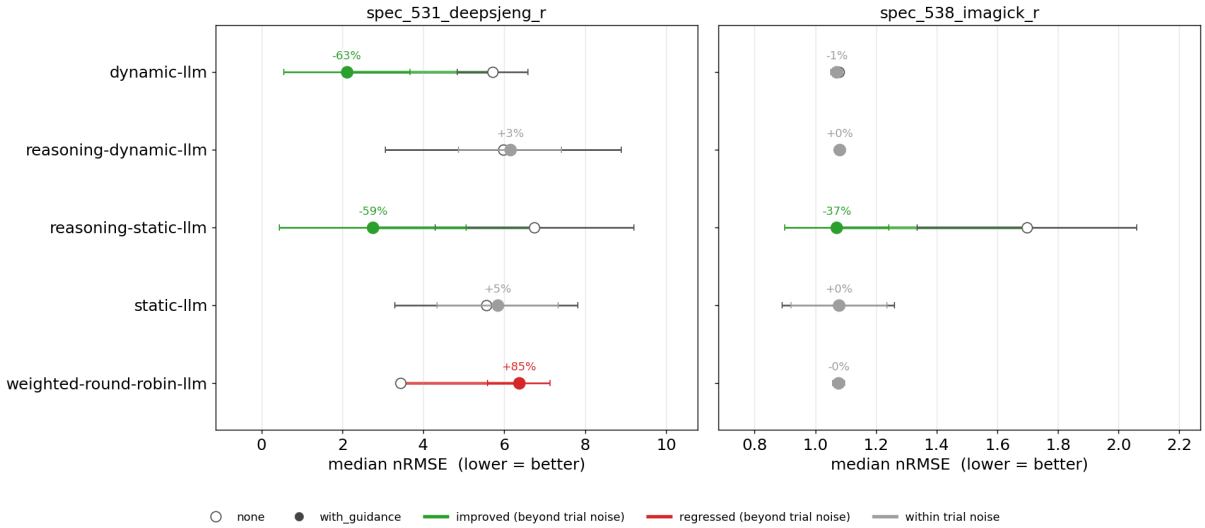


Figure 6.13. Effect of a memory-hierarchy guidance hint on reconstruction accuracy, by scheduler and workload. Each row pairs a scheduler's unguided result (open circle) with its guided result (filled circle); horizontal whiskers span the trial-to-trial standard deviation, and the annotation is the change in median nRMSE. Moves are colored by whether they clear the combined trial noise $\sqrt{\sigma^2_{\text{none}} + \sigma^2_{\text{guided}}}$: green for a real improvement, red for a real regression, grey for within-noise. The two workloads use independent horizontal axes. Guidance helps only where the workload leaves accuracy headroom, and even there it depends on the scheduler.

round-robin scheduler (+85%), and it leaves the static and reasoning-dynamic schedulers unchanged within noise. We attribute the weighted round-robin regression to its purely static weighting (§4.6.3.1): with no estimator feedback to revise an unfortunate weight assignment, a hint that biases the weights toward one counter class is never revisited, and the events it deprioritizes go unmeasured for the entire run.

Across every cell, coverage (∼0.98–1.0) and calibration (∼0.97–0.99) are unchanged by guidance. Guidance therefore reallocates *which* counters SACCADE spends its slots on, rather than *how much* of the workload it observes, which is precisely its intent. We note that median nRMSE, aggregated over all counters, understates this mechanism:

a hint that improves the targeted memory counters at the expense of the deprioritized integer counters can leave the median nearly flat even when the per-counter error has moved substantially. A per-counter-class decomposition of this trade-off is left to future work.

6.4. Configuration Recommendation

SACCADE exposes enough independent knobs that a new user can reasonably ask which combination to reach for first. This section synthesizes the preceding evaluation into a concrete recommendation, organized around the triad of accuracy, performance, and convenience promised in §1.4. We give a default that should work well in the absence of any prior knowledge of the workload, and call out the few cases where another configuration is clearly preferred.

6.4.1. Default Configuration

The safest starting point is the exponential moving average estimator (§4.5.2) paired with the round-robin scheduler (§4.6.1.2). The EMA estimator wins the majority of workloads under nearly every scheduler we evaluated (§6.3.2), and the round-robin scheduler provides performance no worse than random but is deterministic. The pairing is reproducible, has no per-quantum inference latency to amortize, and runs equally well on short-lived and long-lived workloads. For a user with no other information about their target, this is the configuration to beat.

6.4.2. When to Deviate

Two clean cases warrant a different choice.

If the scheduler is the max-uncertainty policy (§4.6.2.1), swap the EMA estimator for the naive Kalman Filter (§4.5.3). The max-uncertainty scheduler steers on a per-event uncertainty signal, and the Kalman Filter is the only estimator whose uncertainty reflects per-event variance rather than mere staleness. This is the one pairing where the Kalman Filter outperformed EMA across the accuracy grid.

If the workload runs long enough for the first dynamic LLM update to complete before the program exits (comfortably longer than the median `dynamic_update` latency of roughly 66 seconds, with headroom for its long upper tail, Figure 6.6), the dynamic LLM scheduler (§4.6.3.3) becomes attractive because of its adaptability to guidance. For shorter workloads, where the dynamic scheduler would degenerate into its static initialization before any feedback could land, the static LLM scheduler (§4.6.3.2) is the appropriate substitute (particularly its reasoning variant).

The correlation-aware Kalman Filter variants (§4.5.3) are not recommended in any configuration. Their off-diagonal process noise interacts poorly with the non-Gaussian, heavy-tailed counter dynamics that appear at phase transitions, and the resulting estimates diverge from ground truth by orders of magnitude (Figure 6.7a).

6.4.3. Timing Parameters

`q_sample` can be chosen almost entirely for analysis quality. The marginal cost of a delivered sample is lost in that floor (Figure 6.3), so there is no overhead penalty for sampling aggressively.

SACCADE's overhead counterintuitively explodes with long scheduling intervals. The stop-the-world quiesce forces the realized scheduling quantum to sit roughly an order

of magnitude above the requested quantum (Figure 6.5), so requesting a value below approximately 10 ms buys nothing but a misleading configuration value. We recommend setting `q_schedule` no smaller than the temporal resolution actually needed by the downstream analysis, with a practical floor in the low tens of milliseconds, but no larger than the low hundreds to avoid a latency explosion. `q_sample` can be set to whatever cadence the estimator deserves.

6.4.4. Natural-Language Guidance

Free-text guidance is worth providing when the user has a specific microarchitectural region of interest and the workload leaves accuracy headroom. On high-variance workloads, guidance can swing reconstruction error by more than a factor of two in either direction (§6.3.4); on noise-floored workloads it is inert. Of the LLM-driven schedulers, the dynamic scheduler and the reasoning static scheduler are the most responsive to guidance.

The one clear contraindication is the weighted round-robin scheduler (§4.6.3.1). Its weights are assigned once at startup and never revised, so a hint that deprioritizes a class of counters causes those counters to go essentially unmeasured for the entire run. Users who want guided scheduling should reach for the dynamic or static LLM schedulers instead.

6.4.5. Summary

The default SACCADE configuration is the EMA estimator paired with the round-robin scheduler: reproducible, robust across our evaluation workloads, and free of LLM-induced latency. Workloads with substantial runtime or a stated region of interest should reach for the dynamic LLM scheduler, optionally with natural-language guidance. The Kalman Filter pairs with the max-uncertainty scheduler when proactive uncertainty steering is desired, and stays out of correlation-aware mode in every case. The sample frequency can be chosen for analysis fidelity rather than overhead, subject to the realized-quantum floor. These recommendations describe the configurations that the present evaluation supports; the next chapter examines the limitations that constrain them and the directions in which they can be sharpened.

CHAPTER 7

Discussion

The evaluations in Chapter 6 reveal a system that largely meets its design goals while exposing concrete limitations and raising new questions. This chapter examines both.

7.1. Limitations

7.1.1. Hardware Constraints

The six-counter ceiling is the hardest constraint in the system. It is a physical property of the PMU and no amount of algorithmic cleverness can increase it. That said, it is not an architectural limitation of SACCADE: the system is parameterized on `num_slots` and will benefit immediately from any future microarchitecture that exposes more physical counters.

The always-on instrumentation overhead of roughly 10.8% (§6.2.2) is similarly grounded in hardware. The dominant cost is the stop-the-world quiesce protocol, which accounts for 95–99% of the slot-swap latency (§6.2.3). The consequence is that the temporal resolution of live profiling runs collapses to roughly 5–10% of the ground-truth timeline (§6.3.3), far below what simulation suggests is achievable. This is a serious practical limitation for short-running workloads.

7.1.2. The Linear-Gaussian Assumption

The Kalman Filter assumes that counter noise is approximately Gaussian and that counter dynamics are approximately linear. Neither holds strictly. At low event rates, PMU counter noise follows a Poisson distribution; during phase transitions, the distribution develops heavy tails. The direct consequence is the divergence seen in the correlation-aware KF variants (Figure 6.7a). Once the off-diagonal process noise Q encodes meaningful cross-counter correlations, a single phase transition can produce an update that inflates the covariance catastrophically. The naive KF avoids this failure by falling back to diagonal process noise, which prevents cross-counter inference from running away, at the cost of making the filter behave largely like a smoothed EMA.

This is a deliberate choice: a simple, stable estimator that provides a clear baseline and an open door for better formulations, rather than a sophisticated one that works only on well-behaved workloads. The path forward is not to tune the linear KF more aggressively, but to replace it with an estimator whose assumptions more closely match reality (§7.2.2).

7.1.3. The Expert-System Prior

The LLM-assisted correlation priors in the KF configuration (§5.3.2) are informally verified and conservatively scored by the author. The analytical correlations from sweep data are more principled, but they too depend on the choice of training workloads. A more systematic study of the prior's sensitivity to workload choice and to LLM-generated correlation quality would strengthen the configuration methodology and sharpen the claims about the Kalman Filter's behavior.

Moreover, the need to collect a separate suite of training workloads (Table 6.1) that cannot be recycled as evaluation workloads adds friction. This separation is necessary to avoid data leakage, but it means that any improvement to the correlation prior requires additional time-consuming sweep runs.

7.1.4. Deployment Requirements

SACCADE requires a modern Linux kernel with `BPF_PROG_TYPE_PERF_EVENT` support and ring-buffer capabilities, root-equivalent permissions to open `perf_event` file descriptors, and an unprivileged-BPF-capable kernel configuration. This rules out many container environments with restricted capabilities, non-Linux platforms, and kernels predating the eBPF features SACCADE depends on. The tradeoff was explicit from the start: depending on existing, well-supported interfaces rather than patching the kernel makes SACCADE broadly deployable *within* the population of machines it supports, at the cost of excluding others.

7.1.5. LLM-Driven Schedulers

Several limitations are specific to the LLM-driven schedulers.

7.1.5.1. Reproducibility. Even with a fixed model and prompt, LLM outputs are nondeterministic. The evaluation methodology mitigates this by averaging over at least three runs per LLM configuration (§6.1.1), but the underlying variance cannot be eliminated. A scheduler that produces a different schedule on two runs with identical inputs is difficult to debug and impossible to fully characterize in a finite evaluation.

7.1.5.2. Model Version Drift. SACCADE uses Google's Gemma 4 `gemma-4-26b-a4b-it` model (§5.4), accessed through OpenRouter. A future model version with different weights would produce different schedules. This is not hypothetical: the literature already documents that LLM reliability varies significantly across versions [44].

7.1.5.3. Latency Budget. The median latency for a dynamic-update call is roughly 66 seconds, with a long tail past 200 seconds (Figure 6.6). The latency-hiding mechanism (§5.4.1) prevents the scheduler from blocking, but it means that for workloads shorter than the first update's latency, the dynamic scheduler is effectively static. The static LLM scheduler is the more appropriate choice for those workloads.

7.1.5.4. Guidance Prompt Sensitivity. The guidance prompt is free-form natural language, and the relationship between phrasing and schedule quality is not fully characterized in §6.3.4. The outer boundary of prompt sensitivity remains open.

7.1.5.5. Prior Knowledge Dependence. The LLM scheduler's quality is bounded by the model's knowledge of the AMD Zen microarchitecture, the specific events available on the test machine, and workload behaviors. Workloads or architectures underrepresented in the model's training data may produce worse schedules, and there is no mechanism to detect this at runtime.

7.2. Future Work

7.2.1. Reducing the Swap Tax

The clearest optimization target is the synchronization cost that dominates slot-swap latency. For the common case of a single-threaded workload, the stop-the-world barrier is unnecessarily conservative: only the CPU running the target thread needs to quiesce.

A targeted quiesce that sends an inter-processor interrupt only to the relevant CPU would reduce the effective swap cost from approximately the realized quantum to near the raw reconfiguration cost of $\sim$2.8 ms (Figure 6.4), recovering temporal resolution on live hardware.

7.2.2. Beyond the Linear-Gaussian Assumption

The Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF) handle mildly nonlinear systems and are the natural first step past the divergence seen in §7.1.2. Particle filters make no distributional assumptions at all and could capture the heavy tails that appear during phase transitions. All three are compatible with the existing `Estimator` trait without any changes to the scheduler or orchestrator, a direct benefit of the decoupled architecture.

A complementary improvement is online calibration of the process noise covariance Q. The current configuration is derived from a static training sweep (§5.3); running the sweep mechanism continuously as a background process and updating Q online would allow the KF to adapt to workloads that diverge from the training distribution.

7.2.3. Learned Schedulers

The most natural successor to the LLM-driven schedulers is a small learned policy trained entirely on SACCADE's own traces. The existing observation flow into the estimator is already what a reinforcement learning agent would consume, and reconstruction error against the ground-truth sweep provides a natural training signal. A policy compact enough to run within the scheduling quantum, on the order of milliseconds rather

than tens of seconds, would eliminate both the latency and the reproducibility problems of the current LLM approach.

A narrower version of the same idea is a distilled guidance interpreter: instead of training a full scheduling policy, train a small model to map natural-language guidance to counter weights, replacing only the LLM weight-assignment step. The LLM itself would generate the training corpus from a collection of annotated profiling sessions.

7.2.4. Multi-Threaded Workloads

All evaluation workloads are single-threaded. The SACCADE architecture supports multi-threaded workloads: each thread is tracked independently in the eBPF thread map. But, the scheduler and estimator have not been evaluated in that setting. The open design question is how the scheduler should allocate the shared counter slots across threads: maintain per-thread state, aggregate across threads, or something in between. This is a natural next step once the swap-tax issues are resolved.

CHAPTER 8

Conclusion

Performance counters are scarce, but the events worth observing are not. Profilers have traditionally treated the resulting multiplexing noise as an unavoidable cost, either restricting the user to a small set of counters at high fidelity or accepting the noise of round-robin multiplexing in exchange for broader coverage. This thesis reframes that tradeoff as a problem of state estimation and observation scheduling, and shows that the resulting design space is both tractable and worth exploring.

SACCADE is the vehicle for that exploration. It separates adaptive profiling into four orthogonal concerns (the event source, the scheduler, the estimator, and the sink) so that any combination can be mixed, replaced, and evaluated against the same ground truth. The orchestrator and eBPF data path handle the mechanics of swapping counters atomically, converting absolute counts to rates, and shuttling samples to userspace at high frequency; the scheduler and estimator are left to focus on policy. The same `sweep` and `simulate` infrastructure that produces the Kalman Filter's training data also makes it possible to replay any scheduler/estimator pair against a recorded trace, which is what enables the two-dimensional evaluation in Chapter 6 without rerunning hardware for every cell.

Our contributions follow from that architecture. We presented a family of estimators, including a formulation of virtual counter reconstruction as a Kalman Filter state-estimation problem with both analytically- and LLM-derived process noise. We

presented a collection of LLM-driven schedulers that translate natural-language guidance into concrete counter selection decisions, ranging from a one-shot weight assignment to a dynamic scheduler that revises itself mid-run while hiding inference latency on a background thread. We presented a mechanism to stitch multiple anchored profiling passes into a global counter trace, and a simulator that replays it through any scheduler/estimator pair. And we presented a configuration recommendation, grounded in the evaluation, for users who would rather not navigate the full design space themselves.

The evaluation supports the thesis but qualifies it. Closed-loop scheduling can reduce reconstruction error, but the best scheduler and estimator are coupled and no single configuration consistently beats a passive round-robin baseline. Live, SACCADE cannot yet match kernel-native `perf stat`, because the stop-the-world swap tax collapses its temporal resolution; in simulation, where that tax vanishes, SACCADE already outperforms `perf stat` on low-noise workloads, which is what makes reducing the swap tax the clearest path forward. LLM-driven schedulers do accept natural-language guidance and, on workloads that outlast schedule-generation latency, can reconfigure mid-run. But no single scheduler/estimator pair dominates: EMA wins under uncertainty-agnostic schedulers, the Kalman Filter wins only when its uncertainty signal is the scheduling signal, and guidance helps only when the workload leaves accuracy headroom. The correlation-aware Kalman Filter variants, which were the most mathematically ambitious of the estimators, diverge in practice because counter dynamics are neither linear nor Gaussian at phase transitions. SACCADE's overhead is dominated by an always-on instrumentation floor and a stop-the-world quiesce that are artifacts of the current

implementation rather than fundamental limits, and the clearest optimization targets follow from that.

The relative scarcity of hardware counters is not going away, and neither is the appetite for broader microarchitectural visibility. By separating mechanism from policy and treating multiplexing as a scheduling problem rather than a noise problem, SACCADE makes it possible to study, compare, and ultimately improve the profilers that close that gap.

References

[1] Performance counters for Linux. http://lwn.net/Articles/310176, 2008.

[2] J. Anderson, L. Berc, G. Chrysos, J. Dean, S. Ghemawat, J. Hicks, S.-T. Leung, M. Lichtenberg, M. Vandevoorde, C. A. Waldspurger, et al. Transparent, low-overhead profiling on modern processors. In *Workshop on Profile and Feedback-directed Compilation*, 1998.

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, Nov. 1997. ISSN 0734-2071. doi: 10.1145/265924.265925. URL https://doi.org/10.1145/265924.265925.

[4] Anthropic. Claude Opus 4.6 System Card. https://anthropic.com/claude-opus-4-6-system-card, February 2026.

[5] Anthropic. Claude Opus 4.7 System Card. https://anthropic.com/claude-opus-4-7-system-card, April 2026.

[6] Anthropic. Claude Opus 4.8 System Card. https://anthropic.com/claude-opus-4-8-system-card, May 2026.

[7] Anthropic. Claude Sonnet 4.6 System Card. https://www.aonthropic.com/system-cards, February 2026.

[8] R. W. Baloh, A. W. Sills, W. E. Kumley, and V. Honrubia. Quantitative measurement of saccade amplitude, duration, and velocity. *Neurology*, 25(11):1065–1065, 1975.

[9] S. S. Banerjee, S. Jha, Z. Kalbarczyk, and R. K. Iyer. BayesPerf: minimizing performance monitoring errors using Bayesian statistics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 832–844, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi:

10.1145/3445814.3446739. URL https://doi.org/10.1145/3445814.3446739.

[10] N. P. Benchmarks. Nas parallel benchmarks. *CG and IS*, 2006.

[11] J. Bucek, K.-D. Lange, and J. v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[12] K. Chu, J. Su, Y. Zhang, C. Zhao, Y. Yang, Y. Zheng, S. Lin, S. Zhao, and W. Zhang. eInfer: Unlocking fine-grained tracing for distributed LLM inference with eBPF. In *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions*, eBPF '25, pages 76–83, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400720840. doi: 10.1145/3748355.3748372. URL https://doi.org/10.1145/3748355.3748372.

[13] W. E. Cohen. Tuning programs with OProfile. *Wide Open Magazine*, 1:53–62, 2004.

[14] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 233–244, USA, 2002. IEEE Computer Society. ISBN 076951605X.

[15] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. of the 2006 Ottawa Linux Symposium*, pages 269–288, 2006.

[16] Google DeepMind. Gemma 4: Our most intelligent open models. https://deepmind.google/models/gemma/gemma-4/, 2026. Accessed: 2026-05-21.

[17] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, Apr. 2004. ISSN 0362-1340. doi: 10.1145/989393.989401. URL https://doi.org/10.1145/989393.989401.

[18] B. Gregg. bcc: Taming Linux 4.3+ tracing superpowers. https://www.brendangregg.com/blog/2015-09-22/bcc-linux-4.3-tracing.html, 2015.

[19] B. Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019. URL https://www.brendangregg.com/bpf-performance-tools-book.html.

[20] S. Harutyunyan Gevorgyan, E. César, A. Sikora, J. Filipovič, and J. Alcaraz. Automatic tuning based on hardware performance counters and machine learning. *Future Generation Computer Systems*, 179:108358, 2026. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2025.108358. URL https://www.sciencedirect.com/

science/article/pii/S0167739X25006521.

[21] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960. URL https://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf.

[22] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334020. doi: 10.1145/2749469.2750392. URL https://doi.org/10.1145/2749469.2750392.

[23] P. Labs. Pixie: Scriptable observability for Kubernetes, 2021. URL https://px.dev.

[24] A. Li, M. Sudvarg, Z. Li, S. Baruah, C. Gill, and N. Zhang. Tintin: a unified hardware performance profiling infrastructure to uncover and manage uncertainty. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-47-2.

[25] G. Liargkovas, V. Jabrayilov, H. Franke, and K. Kaffes. An expert in residence: LLM agents for always-on operating system tuning. In *NeurIPS 2025 Workshop on Machine Learning for Systems*, 2025. URL https://liargkovas.com/assets/pdf/Liargkovas_ML4Sys_NeurIPS25.pdf.

[26] R. V. Lim, D. Carrillo-Cisneros, W. Y. Alkowaileet, and I. D. Scherson. Computationally efficient multiplexing of events on hardware counters. In *Linux Symposium*, 2014. URL https://www.kernel.org/doc/ols/2014/ols2014-lim.pdf.

[27] N. Lindsay, C. Trippel, A. Khandelwal, and A. Bhattacharjee. Counterpoint: Using hardware event counters to refute and refine microarchitectural assumptions. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 459–475, 2026.

[28] J. Liu, X. Zhao, X. Shang, and Z. Shen. Dive into Claude Code: The design space of today's and future ai agent systems, 2026. URL https://arxiv.org/abs/2604.14228.

[29] W. Mathur and J. Cook. Improved estimation for software multiplexing of performance counters. In *13th IEEE International Symposium on Modeling, Analysis,*

and Simulation of Computer and Telecommunication Systems, pages 23–32, 2005. doi: 10.1109/MASCOTS.2005.34.

[30] S. McCanne, V. Jacobson, et al. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[31] K. Pearson. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London Series I*, 58:240–242, Jan. 1895.

[32] M. Pettersson. Perfctr: the Linux performance monitoring counters driver, 2005.

[33] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010. URL `http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68`.

[34] A. Robertson and bpftrace Authors. bpftrace. `https://bpftrace.org`, 2016.

[35] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135742. doi: 10.1145/605397.605403. URL `https://doi.org/10.1145/605397.605403`.

[36] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 336–349, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 0769519458. doi: 10.1145/859618.859657. URL `https://doi.org/10.1145/859618.859657`.

[37] L. Strand. Saccade documentation. `https://liam-strand.github.io/saccade/saccade/index.html`, 2026. Accessed: 2026-05-22.

[38] L. Strand. Saccade: ML-driven hardware counter orchestration. `https://github.com/liam-strand/saccade`, 2026. Accessed: 2026-05-22.

[39] P. D. Team. Parca: Continuous profiling platform, 2021. URL `https://github.com/parca-dev/parca`.

[40] The libbpf-cargo Team. libbpf-cargo. `https://github.com/libbpf/libbpf-cargo`, 2026. Accessed: 2026-05-22.

[41] The libbpf-rs Team. libbpf-rs. https://github.com/libbpf/libbpf-rs, 2026. Accessed: 2026-05-22.

[42] The Rust Team. rust-bindgen. https://github.com/rust-lang/rust-bindgen, 2026. Accessed: 2026-05-22.

[43] The Rust Team. The Rustonomicon. https://doc.rust-lang.org/nomicon, 2026. Accessed: 2026-05-22.

[44] L. D. Thomas, A. K. G. Romasanta, and L. Pujol Priego. Jagged competencies: Measuring the reliability of generative AI in academic research. *Journal of Business Research*, 203:115804, 2026. ISSN 0148-2963. doi: https://doi.org/10.1016/j.jbusres.2025.115804. URL https://www.sciencedirect.com/science/article/pii/S0148296325006277.

[45] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. Intelligent Robotics and Autonomous Agents series. MIT Press, 2005. ISBN 9780262201629. URL https://books.google.com/books?id=2Zn6AQAAQBAJ.

[46] H. Xu, Q. Wang, S. Song, L. K. John, and X. Liu. Can we trust profiling results? understanding and fixing the inaccuracy in modern profilers. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 284–295, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360791. doi: 10.1145/3330345.3330371. URL https://doi.org/10.1145/3330345.3330371.

[47] Y. Yang, T. Yu, Y. Zheng, and A. Quinn. eGPU: Extending eBPF programmability and observability to GPU s. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems*, HCDS '25, pages 73–79, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714702. doi: 10.1145/3723851.3726984. URL https://doi.org/10.1145/3723851.3726984.

[48] A. Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014. doi: 10.1109/ISPASS.2014.6844459.

[49] Y. Zheng, Y. Hu, W. Zhang, and A. Quinn. Towards agentic OS: An LLM agent framework for Linux schedulers, 2025. URL https://arxiv.org/abs/2509.01245.

[50] Y. Zheng, T. Yu, Y. Yang, M. Jiang, X. Gao, J. Su, Y. Hu, W. Mao, W. Zhang, D. Williams, and A. Quinn. gpu_ext: Extensible OS policies for GPUs via eBPF, 2025. URL https://arxiv.org/abs/2512.12615.

APPENDIX A

A Case Study in AI-Assisted Systems Programming

A.1. Introduction and Scope

Recent developments in LLMs and their associated tooling have revolutionized nearly every corner of software development. Although SACCADE's 10,000 lines of Rust pale in comparison to the monstrous codebases that are all too common in systems software, it touches many parts of the runtime stack with complexity sufficient to challenge the programmer, making it a helpful case-study for understanding the performance of modern AI-driven programming tools. This appendix focuses on understanding where these tools helped and where they hurt. Systems software engineers can use these insights to guide their collaboration with this new generation of programming-assistance tools.

The main body of SACCADE was developed between late March and June, 2026. The conventional wisdom at the time was that Anthropic's Claude family of models was the state-of-the-art in systems programming. This space is so volatile that any notion of a "best" model is at the very least shortsighted and perhaps not even the right framing as the most powerful models are also the most expensive. We primarily used Claude Sonnet 4.6 [7] for code generation and Claude Opus 4.6, 4.7, and 4.8 [4, 5, 6] for planning, orchestration, and critique[1]. These models are accessed through Claude Code [28], a terminal-native agentic programming platform.

[1]We quickly adopted the newly released versions of Claude Opus in April and May 2026.

This is, naturally, an $n = 1$ field report, not a controlled study. We nevertheless hope that engineers and researchers find these insights to be a useful starting point for experimentation.

A.2. The Terrain

SACCADE is a complex piece of software that must interact with systems ranging from eBPF to the OpenAI chat API. Several specific subsystems made SACCADE especially challenging to build.

eBPF remains a fairly niche technology. While the core documentation is acceptable, parts of eBPF's broad API surface remain largely undocumented. Furthermore, the verifier runs at *run-time*, not compile-time. This means that it takes longer to discover verification issues and iterate patches. Finally the correctness of the eBPF sampler is hard to evaluate. Any tests that exercise eBPF must be run with root access (or after adding capabilities, which requires root).

The userspace PMU orchestrator, in contrast, interacts with the well-documented `perf_event` interface. The difficulty arose when discovering and resolving synchronization and concurrency bugs, which are inherently nondeterministic.

While Kalman Filters are well-understood and widely-used, our particular application is novel. The interactions between the off-diagonal correlation seeding and the timing model are non-intuitive and very difficult to debug.

Interacting with remote LLM providers for schedule generation was a relatively simple task (the OpenAI chat API is a de facto standard), but integrating the generated schedules into the rest of SACCADE was anything but. Specifically, ensuring that the

latency of schedule generation appears in simulation as it would when running on hardware resulted in significant additional complexity.

Finally, the simulation and evaluation infrastructure represented a tremendous body of engineering work. Dozens of small fiddly systems needed to be developed to provide a transparent sample source interface to the rest of the sampler. Even seemingly unimportant details, like reading in a global trace for replay, became kludgy nightmares[2] when scaled to dozens of simultaneous simulations.

A.3. Where AI Was High-Leverage

AI's leverage was highest where the work was voluminous but well-trodden, and where a cheap oracle (a compiler error, a failing test, or simply the shape of a plot) could confirm or reject its output in seconds. SACCADE was built over roughly three months and nearly ninety development sessions in 2026, and the bulk of that interaction fell into a handful of task shapes that played to these strengths.

The first was scaffolding. SACCADE's simulation and evaluation infrastructure (§4.7) and the Python layer that consumes its output are large, fiddly bodies of code with little conceptual novelty: argument plumbing, progress reporting, resume-and-skip logic, and the scripts that regenerate every figure in this thesis. This is precisely the kind of work that is tedious to write by hand but trivial to verify by running. We routinely handed the assistant a sketch and received a working first draft; on one occasion a single planning pass fanned out into nine parallel subagents that each built one component of the evaluation pipeline. The structural plumbing was high-leverage even when the

[2]For this case, a batch mode was added to `saccade simulate` that read in a global counter trace once and reuse it across multiple parallel simulation runs.

surface polish was not; plot styling conventions, in particular, took several rounds to settle.

The second was learning unfamiliar APIs and idioms quickly. Much of SACCADE's difficulty at the Rust–eBPF boundary, described in §5.1.1, comes from reconciling Rust's ownership model with the relaxed memory model of the generated C interfaces; the deliberately leaked `'static` slab and the `bindgen`-generated cast both fall into territory that is well-documented but unfamiliar to a newcomer. Here the assistant served as a fast index into the `libbpf-rs` ecosystem and the Rustonomicon, surfacing the correct idiom far more quickly than a manual search would have. The documentation existed and was sound; the assistant's value was in retrieval, not invention.

The third was mechanical refactoring and translation. When we migrated SACCADE from four to six hardware counters, the change threaded through the eBPF source, the orchestrator, and the simulation harness as a wide but shallow edit. The same transformation was repeated in many places, but each instance was dependent on local context. Tasks of this shape map naturally onto parallel agents: a single instruction once validated and updated documentation across fifty-one files in one fan-out, and edits of this kind account for the great majority of the roughly eight hundred edit operations performed over the life of the project. None of these changes required insight, only consistency, which is exactly what a tireless and literal collaborator provides.

The fourth, and perhaps least expected, was debugging; not as an authority on the answer, but as a generator and tester of hypotheses. The assistant's most valuable contributions in this mode were the ones that *refuted* our intuitions. In investigating SACCADE's runtime overhead, we asked it to confirm a mechanistic theory of where the

cost originated. It instrumented the relevant path, collected fresh measurements, and confidently reported that the data contradicted the hypothesis. It played the same role in tracking a divergence in the Kalman estimator (§5.3) back to its root cause.

The unifying thread across all four cases is that AI's leverage tracked task novelty inversely and verifiability directly: the more familiar the territory and the cheaper the oracle, the more we could trust and the less we had to check. Where that condition fails (novel, sparsely documented, and correctness-critical work), the picture inverts.

A.4. Where AI Was Low-Leverage or Actively Misleading

The properties that made AI high-leverage, when inverted, mark out where it became a liability. The same fluency that produced a working scaffold in seconds also produced fluent answers in domains we did not yet understand, and a confident wrong answer in unfamiliar territory cost us far more than no answer at all. Unsurprisingly, these failures cluster in the same subsystems that §A.2 flagged as hardest to build.

The eBPF verifier was the most acute case. Because the verifier runs at *run-time* and emits cryptic diagnostics (§5.1.2), the assistant could neither see the real error surface nor reason reliably about it; it proposed confidently-wrong fixes and reached for helper functions that either do not exist or are unavailable in our program type. The feedback loop was doubly throttled. Even *running* the verifier to test a candidate fix requires elevated privileges, either root or the `CAP_BPF` and `CAP_SYS_ADMIN` capabilities, so the assistant could not iterate against the real oracle on its own; every attempt had to round-trip through us. That opaque, privilege-gated loop is precisely the condition under which a plausible-but-wrong suggestion wastes more time than it saves.

Hardware PMU management fared little better. The `perf_event` interface is documented, but the access pattern SACCADE needs is not, and the world-stop protocol that enforces atomic counter swaps (§5.2.1) is both correctness-critical and racy. The concurrency bugs in this code are nondeterministic, as §A.2 notes, so the assistant's suggestions could be neither quickly validated nor safely trusted, and the cost of a misattributed event is silent data corruption rather than an obvious crash. Here AI offered little leverage and some genuine hazard, at one point proposing an environment-variable toggle in place of the explicit synchronization the problem actually demanded.

The most insidious failures were in numerical and statistical work, where wrong answers look exactly like right ones. The clearest example arose while seeding the off-diagonal correlation entries of the Kalman filter's covariance (§5.3.1). The AI-suggested construction was entirely plausible, but it omitted the requirement that the covariance matrix remain positive semi-definite, so the filter quietly diverged; the defect surfaced only as degraded estimates, never as an error or exception. This is the canonical "looks right, is wrong" failure mode, and the most dangerous one, because nothing in the toolchain objects. Tellingly, even after we corrected the math, the elaborate correlation model never beat the naive per-event filter, a reminder that confident, complex AI output can be not only subtly wrong but *unnecessary*.

A final, lower-stakes category was hallucinated APIs and outdated patterns. While integrating `bindgen` to generate Rust types from our C headers (§5.1.1), the assistant invented a builder option that does not exist and over-engineered a separate generated-types module before we settled on the simpler integration the chapter describes.

The unifying pattern is by now clear: the failures cluster precisely where novelty, sparse or runtime-only documentation, and correctness-criticality coincide, the exact inverse of the conditions that made AI valuable in §A.3. That intersection is the danger zone, and learning to recognize it is what motivates our best practices.

A.5. Workflow and Practices That Worked

The split between §A.3 and §A.4 is only useful if it can be acted on in the moment, while the code is being written. A single principle organized our workflow: we calibrated how much we trusted the assistant to the cost of checking its work. Where an independent oracle was cheap, we delegated broadly and judged only the result; where verification was expensive, we kept a short leash.

The first practice followed directly: we treated the compiler, the test suite, and the eBPF verifier as non-negotiable ground truth, and gated every multi-file change on them before it was allowed to land. Running `cargo build`, `cargo clippy`, and the test suite after each change converted the assistant's confident-but-occasionally-wrong output into something we could accept quickly, because we did not need to read every line when an impartial arbiter would reject a mistake. This was what made broad delegation safe rather than reckless; when we once asked a single instruction to validate and update documentation across fifty-one files, it was `cargo doc` and not a human reviewer that caught the regressions.

The second practice was to work in two distinct registers depending on which side of that boundary we were on. On disposable scaffolding, such as the simulation and plotting infrastructure of §4.7, we gave the assistant free rein and reviewed only whether

the output was correct. On correctness-critical paths, such as the verifier interactions of §5.1.2, the atomic-swap protocol of §5.2.1, and the Kalman covariance math, we kept the leash short: small diffs, an explicit account of the reasoning, and human review of every line. The boundary between the two registers was precisely the boundary between high- and low-leverage work.

A harder-won practice was knowing when to abandon an AI thread entirely. In the danger zone, the assistant could enter a confidently-wrong spiral in which each new suggestion was as plausible and as broken as the last, and the marginal cost of one more prompt quietly exceeded the cost of simply writing the code ourselves. We learned to recognize that spiral early, cut our losses, and write the tricky core by hand, reserving the assistant for the scaffolding around it.

Two habits of *how* we prompted transferred across the whole project. The first was to lead with a reviewed plan before writing any code: we repeatedly had the assistant produce an implementation plan, revised it against our intent, and only then let it execute, which front-loaded alignment and surfaced wrong approaches while they were still cheap to discard. The second was to invite disconfirming evidence rather than agreement. When a hypothesis was on the table, we asked the assistant to check it against the actual code and data and to say plainly when the evidence contradicted our intuition, which is what made it useful as a skeptic rather than a mirror, as the refuted overhead theory of §A.3 illustrates. None of these practices are specific to SACCADE, and we distill them into concrete recommendations in §A.7.

A.6. Threats to These Observations

These observations carry the obvious caveats of their origin, and we restate them plainly so they are not mistaken for more than they are. The most fundamental is the absence of a control: as §A.1 notes, this is an $n = 1$ account of one developer building one roughly ten-thousand-line codebase over a single three-month window. No second developer built SACCADE without assistance, and our own fluency with Rust, eBPF, and the problem itself grew over exactly the same period, so we cannot cleanly separate what the assistant contributed from what we simply learned to do.

The tooling, moreover, was a moving target. We adopted three successive releases of Claude Opus over the course of the work, and the agentic platform around them evolved in step. Specific failures we report, such as the verifier confusion of §A.4 and the hallucinated APIs, may already be artifacts of a particular model version rather than durable properties; these observations are best read as a snapshot of a fast-moving capability frontier, not as fixed laws.

Finally, this is a retrospective report, and so it is vulnerable to the usual recall and selection biases: vivid successes and frustrations are over-weighted against the unremarkable majority of interactions, and we did not record in advance where we expected the assistant to help or fail. The single thesis that organizes these sections, that leverage tracks the cost of verification, is itself a lens that shaped what we chose to notice. SACCADE is also unusual software, and the boundary we drew between §A.3 and §A.4 may shift in domains with denser documentation and cheaper oracles than systems programming affords. We therefore offer these observations, as we did at the outset, as a starting point for others' experimentation rather than as settled findings.

A.7. Recommendations for Future Practitioners

If the preceding sections distill to a handful of transferable rules, they are these.

Build the oracle before you delegate. The first and most valuable investment is not a better prompt but cheaper verification: a fast test suite, continuous-integration gates, and a willingness to treat the compiler and the eBPF verifier as impartial arbiters (§A.5). The amount of work you can safely hand to an assistant is bounded by how quickly an independent check can reject its mistakes, so verification infrastructure is a critical feature, not an afterthought.

Lean in hardest where work is voluminous, well-trodden, and verifiable. Delegate broadly on scaffolding, on learning unfamiliar APIs and idioms, on mechanical refactors and translations, and on generating and testing hypotheses (§A.3). In this register the assistant is a genuine force multiplier, and the cost of a wrong answer is both low and quickly caught.

Take the wheel where mistakes stay silent. Where novelty, sparse documentation, and correctness-criticality coincide (§A.4), the worst errors produce no crash and no warning, only quietly wrong results: the Kalman covariance that was no longer PSD, a counter event attributed to the wrong slot. Shorten the leash here to small diffs, explicit reasoning, and human review of every line, and keep the discipline to abandon a confidently-wrong thread and write the tricky core yourself.

Treat the assistant as a skeptic, not an oracle. Lead with a reviewed plan before writing any code, and actively invite disconfirming evidence rather than agreement (§A.5); the assistant's most valuable contributions were the ones that refuted our intuitions, not the ones that flattered them.

Underneath all four rules is a single observation. In every case, from the scaffolding we trusted on sight to the numerical code we checked line by line, the leverage AI offered tracked the cost of verifying its output. The skill that matters, then, is not prompt-craft but the judgment to read that gradient in the moment and to calibrate trust accordingly, granting broad autonomy where an oracle is cheap and keeping a firm hand where it is not. This is also the one lesson we expect to outlast the particular models and platforms of §A.6: the tools will keep improving, but the discipline of matching trust to verifiability should remain.

APPENDIX B

LLM Scheduler Prompts and Configuration

B.1. LLM Configuration

SACCADE draws a deliberate distinction between the model used to *generate* schedules and the latency model used to *simulate* their runtime cost; the motivation for this split is discussed in §5.4.1.

Schedule generation queries Google's Gemma 4 `gemma-4-26b-a4b-it` model [16], accessed through the OpenRouter inference service over an OpenAI-compatible chat completions API. The client authenticates with a bearer token, applies a 10 s connection timeout and a 600 s response timeout, and retries transient transport failures (timeouts, dropped connections, HTTP 429, and 5xx responses) up to four times with exponential backoff between 2 s and 30 s.

In simulation, the measured wall-clock latency of each LLM call is instead replaced by a sample drawn from a per-call-type latency distribution. These distributions were profiled on our local inference server, which runs the 4-bit quantized, 8 billion parameter version of Gemma 4 (Ollama blob `c6eb396dbd59`) under Ollama version 0.20.2, using the `q7_llm_latency.py` harness. Latency is sampled separately for each call type; `static_setup`, `wrr_setup`, and `dynamic_update`; each with a `_reason` companion when the reasoning variant is enabled.

B.2. Prompts

B.2.1. Shared System Message

All interactions share the same system message with the following substitutions:

- `<NUM_SLOTS>`: integer number of hardware counter slots the profiler can activate simultaneously.
- `<GUIDANCE>`: optional free-text string supplied by the user at scheduler construction time. The entire "User guidance:" line is omitted when no guidance is provided.

```
You are an expert Linux performance profiling assistant.
You generate hardware performance counter observation schedules
for a sampling profiler that activates <NUM_SLOTS> counters
simultaneously. The profiler cycles through the schedule
indefinitely, rotating which counters are active to build a
complete picture of program behavior over time.

User guidance: <GUIDANCE>
```

B.2.2. Initial Schedule Generation

The static scheduler (§4.6.3.2) and dynamic scheduler (§4.6.3.3) use the same prompt to generate their initial schedule. The conversation has four turns: the shared system message (§B.2.1), the user message below, an assistant turn carrying a few-shot example (`EXAMPLE_JSON`), and a short closing user instruction. The substitutions are:

- `<EVENT_LIST>`: one line per available hardware counter in the form "ID: name – description", drawn from the system's `perf` event library.
- `<NUM_SLOTS>`: as in §B.2.1.

- `EXAMPLE_JSON`: a pretty-printed JSON object of the form `{"steps": [{"duration_ms": <int>, "events": [<id>, ...]}, ...]}`, constructed from the first events in `<EVENT_LIST>` with step durations [100, 25, 75] ms. The example deliberately repeats the first (high-priority) event across all three steps to demonstrate a tilted, non-uniform schedule rather than a flat round-robin. It is supplied as an assistant turn, not embedded in the user message.

The output is further constrained by a JSON schema enforced through the server's structured-output mode (§B.2.6), so the explicit "respond only with JSON" instructions of earlier revisions are no longer required.

```
Available performance counters (format -- ID: name: description):
<EVENT_LIST>
Generate a cyclic measurement schedule. Every counter must appear at
least once across the full cycle -- this coverage floor is mandatory.
Within that constraint, bias the marginal budget (extra repetitions,
longer durations) toward counters that reveal bottlenecks or are
expected to be bursty, high-rate, or high-variance: cache misses, TLB
pressure, branch mispredictions, memory stalls, and ratio denominators
such as instructions and cpu-cycles. Important counters SHOULD recur
across multiple steps. Uniform allocation is NOT required and is
generally suboptimal -- deliberately tilt the schedule.

Each step activates <NUM_SLOTS> counters and runs for a specified
duration (aim for 10-1000 ms per step; vary durations to give more
observation time to high-priority steps). Each step must have
duration_ms (milliseconds) and events (an array of exactly <NUM_SLOTS>
counter IDs). Use each event ID at most once per step. Use only event
IDs from the list above.
```

The assistant then replies with `EXAMPLE_JSON`, and the conversation closes with the user instruction:

```
Good. Now produce the full, deliberately-tilted schedule covering ALL
the counters listed above -- not just those in the example.
```

B.2.3. Priority Weight Assignment

The weighted round robin scheduler (§4.6.3.1) uses its own prompt to generate the weights. As with the initial-schedule prompt, the few-shot example is supplied as an assistant turn and the output is constrained by a JSON schema (§B.2.6). The substitutions are:

- `<EVENT_LIST>`: as in §B.2.2.
- `<NUM_SLOTS>`: as in §B.2.1.
- `EXAMPLE_JSON`: a pretty-printed JSON object of the form `{"weights": [{"event_id": <int>, "weight": <int>}, ...]}`, constructed from the first four events in `<EVENT_LIST>` with weights [9, 6, 3, 1]. The spread across the full range discourages the model from clustering every counter near a single weight, which would degrade to round-robin sampling.

```
Available performance counters (format -- ID: name: description):
<EVENT_LIST>
Assign a priority weight (integer 1-10) to each counter based on its
expected information value. Use the FULL 1-10 range -- do NOT cluster
all counters near one value, as that degrades to round-robin sampling.
Rank higher (8-10) counters that are bursty, high-variance, or form
critical ratio denominators -- cpu-cycles, instructions, cache misses,
branch mispredictions, TLB misses, and memory stalls. Rank lower (1-3)
counters that are stable, low-rate, or redundant given others already
measured. Every counter must appear exactly once in your response.
Use only event IDs from the list above.
```

The assistant replies with `EXAMPLE_JSON`, and the user closes with:

```
Good. Now produce the COMPLETE weights list for ALL the counters listed
above, using the full 1-10 range with meaningful differentiation.
```

B.2.4. Dynamic Schedule Update

The dynamic LLM scheduler (§4.6.3.3) updates its schedule using the prompt below, again followed by an assistant few-shot example and a closing user instruction, with the output constrained by a JSON schema (§B.2.6). The substitutions are:

- `<EVENT_LIST>`: as in §B.2.2.
- `<NUM_SLOTS>`: as in §B.2.1.
- `<OBS_LIST>`: one line per counter: `ID (name): rate=R ev/ns, uncertainty=U, samples=N`, or `ID (name): not yet observed` if the counter has never been active.
- `CURRENT_SCHEDULE_JSON`: pretty-printed JSON of the schedule currently in use, in the same `{"steps": [...]}` schema as §B.2.2.

```
Available performance counters (format -- ID: name: description):
<EVENT_LIST>
Recent observations (rate in events/ns, uncertainty in
                     [0=confident, 1=unknown]):
<OBS_LIST>
Current schedule (for reference):
<CURRENT_SCHEDULE_JSON>

Generate an updated cyclic schedule for the same <NUM_SLOTS>-slot
profiler. INCREASE sampling (more repetitions, longer durations) for
counters that have high uncertainty or volatile rates -- they need more
observations. DECREASE sampling for counters that are confident and
stable. Bootstrap any "not yet observed" counter soon -- give it an
early slot. Every counter must still appear at least once across the
full cycle. Each step must have duration_ms (milliseconds) and events
```

```
(an array of exactly <NUM_SLOTS> counter IDs). Use each event ID at most
once per step. Use only event IDs from the list above.
```

The assistant replies with the few-shot example, and the user closes with:

```
Good. Now produce the full updated schedule covering ALL the counters
listed above -- not just those in the example.
```

B.2.5. Reasoning Analysis

The reasoning variants of the static and dynamic schedulers (Sections 4.6.3.2 and 4.6.3.3) precede schedule generation with a single free-form call that imposes no JSON schema, inviting the model to analyze the counters in prose. The resulting analysis is then replayed into the schedule-generation conversation as an assistant turn, so the model treats it as its own chain of thought. The substitutions `<EVENT_LIST>` and `<NUM_SLOTS>` are as above.

```
Available performance counters (format -- ID: name: description):
<EVENT_LIST>
Before generating a schedule, analyze these counters in prose. Consider:
1. Which counters are likely bursty or high-variance (spiky event rates)?
2. Which counters form meaningful pairs or ratios (e.g. misses/references,
   instructions/cycles) that must be co-sampled to be interpretable?
3. Which counters are ratio denominators (cpu-cycles, instructions,
   cache-references) and therefore especially important to sample
   frequently?
4. Which counters are stable, low-rate, or redundant given others?
5. Given <NUM_SLOTS> simultaneous slots, what allocation strategy would
   maximize information gain -- which counters deserve to recur, and
   which are lower priority?

Do NOT produce JSON. Reason freely in prose; the schedule will be
generated in the next step based on your analysis.
```

When reasoning is enabled, the schedule-generation conversation (§B.2.2) is extended: after the assistant few-shot example, a user turn invites the analysis ("Good. First, reason through which counters matter most and how to allocate the sampling budget across the cycle."), the model's prose is inserted as an assistant turn, and a final user turn requests the schedule ("Now produce the full, deliberately-tilted schedule as JSON, following your reasoning above and covering ALL the counters listed."). The dynamic update conversation (§B.2.4) is extended analogously, with the invitation phrased in terms of which counters need more or less sampling given the recent observations.

B.2.6. Structured Output Schema

The schedule- and weight-generation calls request structured output from the inference server by supplying a strict JSON schema. The server constrains decoding to that schema, which guarantees syntactically valid JSON, enforces the required object shape, and restricts every event ID to the set of valid counters. Schedule steps are required to contain exactly `<NUM_SLOTS>` events (enforced via `minItems` and `maxItems`), and each event ID is drawn from an enumerated list of valid IDs so the model cannot emit an out-of-range counter. The free-form reasoning call (§B.2.5) is the only LLM call that omits a schema.

Even with schema enforcement, every response is validated again on the client: unknown IDs are filtered, duplicates removed, short steps padded with unused counters, and zero durations clamped. A response that still cannot be turned into a usable schedule triggers the correction prompt below.

B.2.7. Parse-Error Correction

When a validated response still cannot be parsed into a usable schedule or weight list, we retry following the logic in §5.4.2. Before re-prompting, the malformed response is appended to the conversation as an assistant turn, unless it was empty, in which case it is omitted to avoid teaching the model to repeat empty output. We then send the correction prompt below, with the substitution:

- `<REASON>`: the validation error, truncated at the first newline so the raw model response that triggered the failure is omitted.

```
Your response contained an error: <REASON>
Fix the error and respond again.
```