Steady-State Convergence in a Self-Adaptive Virtual Machine: An Empirical Study

Robert A. James
StarForth Project
rajames440@gmail.com https://github.com/rajames440

December 16, 2025

Abstract

We document empirical observations of steady-state behavior in a minimal virtual machine equipped with adaptive runtime mechanisms. The system implements seven configurable feedback loops that modify execution behavior based on runtime metrics including instruction heat, transition probabilities, and window-based history capture.

Using controlled workloads and high-resolution instrumentation, we measured runtime stability across 360 experimental runs spanning 128 distinct runtime configurations. Measurements include execution time, performance variance, and a dimensionless performance statistic K derived from effective window utilization.

Results demonstrate deterministic convergence to stable execution regimes under fixed workload conditions. The statistic K exhibits measurable invariance (coefficient of variation $< 1\%$) across replicate runs and predictable modulation with window size. System behavior shows attractor-like characteristics, with configurations settling into reproducible states despite dynamic adaptation mechanisms.

These observations suggest that adaptive runtime systems can achieve measurable stability without sacrificing dynamic behavior. The existence of empirical invariants in a minimal implementation indicates that such properties may be potentially non-accidental patterns worthy of further investigation across diverse platforms.

1 Introduction

Adaptive runtime systems typically face a tradeoff between dynamic optimization and behavioral predictability. Systems that adjust execution strategy based on runtime feedback often exhibit complex transient behavior, making performance analysis difficult and reproducibility questionable. While such systems may eventually settle into stable operating regimes, the conditions under which stability emerges and the properties of such regimes remain poorly characterized.

The central question motivating this work is whether adaptive behavior can converge to measurable steady states despite continuous feedback-driven adjustments. If stable regimes exist, can they be characterized by empirical invariants? And if so, what does this imply for the design and analysis of adaptive systems?

We approach these questions through controlled experimentation on a minimal virtual machine (VM) instrumented with seven configurable feedback mechanisms. These mechanisms—including execution heat tracking, rolling history windows, linear decay, and transition prediction—operate concurrently and influence lookup latency, caching behavior, and runtime state. The system serves as a test platform for observing emergence of stable behavior under controlled workload conditions.

Through 360 experimental runs across 128 runtime configurations, we observe deterministic convergence to reproducible execution states. A dimensionless performance statistic K, derived from the ratio of intrinsic system scale to effective window utilization, exhibits coefficient of variation below 1% across replicate runs. The statistic follows predictable patterns as window size varies, with resonance-like peaks and anti-resonance troughs occurring at specific parameter values.

Scope and limitations. This paper presents empirical observations and descriptive models. We do not claim first-principles derivation, universality across all platforms, or explanatory theory for the observed phenomena. The results document what was measured under controlled conditions on a specific hardware platform with deterministic workloads. Generalization to production systems, diverse architectures, and non-deterministic workloads requires further validation.

Contributions. This paper makes the following empirical contributions:

1. Documentation of measurable steady-state convergence in an adaptive VM with concurrent feedback mechanisms
2. Identification of empirical invariants (K-statistic, characteristic length scale $\lambda_0 = 256$ bytes, modulation frequency $f_0 = 2/3$ cycles/window) that remain consistent across workloads and configurations
3. Characterization of attractor-like behavior, including dual-mode states and deterministic transitions
4. Experimental methodology for measuring runtime stability in adaptive systems

This paper documents the architecture, methodology, and empirical observations of such behavior.

2 Related Work

2.1 Adaptive Runtime Systems

Self-tuning runtime environments have been explored across multiple domains. The Jikes RVM [1] demonstrated adaptive compilation with dynamic method selection. The HotSpot JVM employs tiered compilation and speculative optimization based on runtime profiling [11]. These systems optimize for common-case performance through feedback-driven specialization, though their convergence properties and steady-state characteristics are not typically characterized quantitatively.

PyPy's tracing JIT [3] uses execution traces to guide optimization, exhibiting adaptive behavior through guard-based speculation. SPUR [2] extended this model with staged compilation. Both demonstrate that adaptive mechanisms can improve performance, though behavioral stability under sustained workloads remains an open research question.

2.2 Control-Theoretic Approaches

Control theory has been applied to runtime resource management in systems such as AutoPilot [5] and PCCP [8]. These approaches model runtime behavior using feedback control loops with explicit stability analysis. Our work differs in observing emergent stability without explicit control-theoretic design, focusing on measurement rather than controller synthesis.

2.3 Virtual Machine Instrumentation

High-resolution VM instrumentation has enabled detailed performance analysis. Valgrind [10] provides fine-grained execution tracing. Pin [9] enables dynamic binary instrumentation. DynamoRIO [4] supports runtime code manipulation. These tools focus on observation and profiling; our instrumentation additionally feeds back into execution behavior.

2.4 Operating System and Runtime Co-design

Exokernel [6] explored application-level resource management. Singularity [7] demonstrated language-runtime-OS integration. L4Re/Fiasco.OC [12] provides microkernel-based isolation with fine-grained resource control. These projects demonstrate that runtime-OS boundaries are negotiable; our work examines adaptive behavior within a single runtime layer.

Our work complements these prior efforts by documenting measurable steady-state properties in a minimal adaptive system. We focus on empirical characterization rather than performance optimization, stability guarantees, or architectural innovation.

3 System Architecture

3.1 Virtual Machine Overview

The experimental platform is a FORTH-79 compliant virtual machine implementing a minimal stack-based execution model. The VM provides:

- Two primary stacks: data stack (operands) and return stack (control flow)
- Dictionary-based word lookup with linear search by default
- Threaded code interpreter executing words sequentially
- 5 MB virtual address space with byte-addressable memory
- Block-based persistent storage subsystem (1 KB blocks)

Memory model. The VM maintains strict separation between VM addresses (byte offsets) and host addresses (C pointers). All word code operates on VM offsets, with canonical accessor functions mediating memory operations. This design enables deterministic address arithmetic independent of host pointer size.

Execution model. Each word in the dictionary consists of a name, flags, and either native code or a compiled sequence of word references. The interpreter fetches the next word, looks it up in the dictionary, and executes it. Execution proceeds until an explicit halt (BYE) or error condition.

Determinism boundary. The base VM is deterministic: identical bytecode and identical workloads produce identical execution traces. Adaptive mechanisms (Section 3.3) introduce feedback-driven behavior that remains deterministic for fixed initial conditions but varies with configuration.

3.2 Instrumentation and Feedback

The VM is instrumented to capture execution metadata without altering semantic behavior. Instrumentation operates at three granularities:

Per-word execution tracking. Each dictionary entry maintains an `execution_heat` accumulator incremented on invocation. Heat values decay over time according to configurable policies (Section 3.3), implementing a computational analog of thermal dynamics.

Rolling history window. A circular buffer (default 4096 entries) records the sequence of executed word IDs. This "Rolling Window of Truth" captures short-term execution patterns and provides statistical context for adaptive decisions. The window supports lock-free snapshots via double-buffering, enabling concurrent reads by analysis threads.

Word transition metrics. A 16×16 matrix tracks pairwise word-to-word transition frequencies. Entries represent observed counts of word i being followed by word j. This data structure functions analogously to a memristive crossbar pattern, with transition probabilities serving as adaptive weights.

Heartbeat coordination. A background thread (when enabled) runs at configurable intervals (10-1000 ms) to:

1. Apply heat decay updates
2. Evaluate adaptive tuning heuristics

3. Update window size and decay parameters
4. Capture snapshot metrics for external analysis

Instrumentation overhead is minimal (¡ 2% runtime impact measured via DoE with all loops disabled vs. uninstrumented baseline).

3.3 Adaptive Mechanisms

Seven feedback loops (L1–L7) provide tunable adaptation. Each loop can be independently enabled or disabled, yielding $2^7 = 128$ distinct configurations. An eighth mechanism (L8) implements meta-level adaptive selection among strategies.

L1: Execution Heat Tracking. Increments per-word heat counters on each invocation. Provides foundational signal for all downstream mechanisms.

L2: Rolling Window of Truth. Maintains circular buffer of execution history. Window size is adaptive: starts large (4096 entries) and may shrink if pattern diversity saturates.

L3: Linear Heat Decay. Applies uniform decay to all heat values periodically. Decay rate is configurable; typical values are 0.1–1.0 heat units per heartbeat tick. Prevents unbounded heat accumulation.

L4: Pipelining Metrics. Records word-to-word transition frequencies. Enables speculative prefetching and locality prediction (not evaluated in this paper; infrastructure present for future work).

L5: Window Width Inference. Applies Levene's test to detect variance heterogeneity in execution timing. If variance is stable, window may shrink to reduce memory footprint. If variance increases, window expands to capture more history.

L6: Decay Slope Inference. Fits exponential regression to heat trajectory over recent heartbeat ticks. If heat is growing, decay rate increases to stabilize. If heat is declining, decay rate decreases to preserve signal.

L7: Adaptive Heartrate. Adjusts heartbeat interval based on system load and execution phase. During steady execution, heartbeat slows (saves power). During transients (detected via heat volatility), heartbeat accelerates (improves responsiveness).

L8: Adaptive Strategy Selector. Meta-mechanism that switches among predefined loop combinations based on workload classification. Not a feedback loop itself, but a controller selecting which loops to activate.

Hot-words cache. When L1 is enabled, frequently-executed words (top 16 by heat) are cached separately for fast lookup. Cache uses frequency-based eviction: if a word's heat drops below the cache minimum, it is evicted. This implements a physics-driven caching policy without explicit LRU bookkeeping.

All adaptive mechanisms respect the determinism boundary: given fixed loop configuration and fixed initial state, behavior is reproducible. Variance arises only from configuration changes or non-deterministic workloads (none used in this study).

4 Formal Definitions and Metrics

Terminology note. This paper uses physics-inspired terminology ("memristive," "resonance," "frequency") descriptively to characterize observed patterns, not to claim physical equivalence or derivation from first principles. These terms serve as convenient analogs for empirical phenomena in adaptive software systems. We make no claim that the observed behaviors reflect universal physical laws.

4.1 Steady-State Definition

We define steady-state operationally through observable criteria rather than theoretical equilibrium conditions.

Definition 1 (Steady State). A VM execution is in steady state if the following conditions hold over a measurement window of duration T:

1. **Execution completion:** The workload completes without error or timeout
2. **Temporal stability:** Total execution time t has coefficient of variation $\text{CV} = \sigma_t/\mu_t < 0.05$ across replicate runs (¡ 5% relative variance)
3. **Metric reproducibility:** Performance statistic K (defined below) varies by less than 1% across replicates with identical configuration
4. **Heat convergence:** Sum of all execution heat values H_{total} stabilizes such that $|\Delta H_{\text{total}}|/H_{\text{total}} < 0.01$ over the final 10% of execution

This definition avoids requiring mathematical equilibrium ($dH/dt = 0$) which is never achieved in discrete systems. Instead, it captures ”stable enough to measure” regimes.

Time window assumptions. All measurements occur after a warm-up period of at least 1000 word executions to allow feedback loops to stabilize. Measurements span the full workload execution (typically 10^5–10^6 word executions for our test suite).

4.2 The K-Statistic

We introduce a dimensionless performance statistic K that characterizes window utilization efficiency.

Definition 2 (K-statistic). Let W_{config} be the configured rolling window size (bytes) and W_{actual} be the effective window size utilized by the system. The K-statistic is:

$$K = \frac{\Lambda_{\text{eff}}}{W_{\text{actual}}} \tag{1}$$

where Λ_{eff} is an empirically-determined characteristic length scale (units: bytes).

Operational measurement. W_{actual} is not directly observable. We infer it from the relationship:

$$W_{\text{actual}} = \Lambda_{\text{eff}}/K \tag{2}$$

where K is computed from runtime metrics (execution time, heat distribution entropy, transition matrix rank). Details in Appendix A.

Empirical observation. Across 360 experimental runs, we observe:

- $\Lambda_{\text{eff}} = 256 \pm 8$ bytes (3% uncertainty)
- K exhibits coefficient of variation $\text{CV} < 1\%$ for fixed configuration
- K varies systematically with W_{config}, following a modulated inverse relationship (Section 6)

Invariance claim (carefully stated). Under fixed workload and fixed loop configuration, K is reproducible to within measurement precision. K is not universal across configurations but is a stable observable for each configuration.

We make no claim that K represents a fundamental constant. It is an empirical statistic useful for characterizing steady-state behavior.

4.3 James Law (Descriptive Form)

Empirical data suggest K follows a modulated inverse relationship with window size:

$$K(W) = \frac{\Lambda_{\text{eff}}}{W} \times [1 + A(W)\sin(2\pi f_0 \log_2(W) + \varphi)] \tag{3}$$

where:

- $\Lambda_{\text{eff}} = 256$ bytes (intrinsic scale)
- W is configured window size (bytes)
- $f_0 = 2/3$ cycles/window (observed frequency)
- $A(W) = A_{\max}\exp(-W/W_{\text{decay}})$ (damping envelope)
- $A_{\max} \approx 0.3$, $W_{\text{decay}} \approx 50000$ bytes
- φ is phase offset (configuration-dependent)

This functional form fits experimental data with $R^2 > 0.99$ (Section 6). We refer to this empirical relationship as the *James Law* for convenient reference, with no implication of universality.

Physical interpretation (speculative). The inverse baseline Λ_{eff}/W may reflect fundamental scaling; the sinusoidal modulation may arise from cache line interference or page boundary effects. However, the mechanism remains unproven. The law is presented as a descriptive model, not an explanatory theory.

4.4 Measurement Constraints

Temporal resolution. Timing measurements use 64-bit nanosecond counters with Q48.16 fixed-point arithmetic (16 fractional bits). This provides ∼15 picosecond theoretical resolution, though practical resolution is limited by timer granularity (∼1 nanosecond on x86-64 with `CLOCK_MONOTONIC`).

Heat quantization. Heat values are 64-bit unsigned integers, incremented by 1 per invocation. For typical workloads (10^5 invocations), this provides 16 decimal digits of precision—far exceeding measurement noise.

Noise sources. Identified sources of measurement variance include:

1. OS scheduler preemption (mitigated via CPU pinning and isolation)
2. Cache state variability (mitigated via cold-start initialization between runs)
3. Branch predictor state (not mitigated; assumed negligible for deterministic workloads)
4. Background system activity (mitigated via dedicated test machine)

Despite mitigation, residual noise is observed. Coefficient of variation across replicates ranges from 0.1% (best case, anti-resonance) to 6.9% (worst case, resonance with high loop activity).

What is not measured. The following are explicitly not measured in this study:

- Power consumption (no hardware instrumentation)
- Memory bandwidth utilization (no PMU access)
- Instruction-level microarchitecture events (no perf integration)

- Long-term drift (¿ 1 hour execution; workloads complete in seconds)

Threats from unmeasured variables. It is possible that unmeasured hardware effects (e.g., thermal throttling, DVFS transitions) contribute to observed behavior. We assume such effects are negligible for our short-duration workloads (¡ 1 second runtime) but cannot rule them out.

See Section 9 (Threats to Validity) for comprehensive discussion of measurement limitations.

5 Experimental Methodology

5.1 Hardware Environment

All experiments were conducted on a Beelink MINI S (AZW MINI S) compact workstation with the following specification:

- **Processor:** Intel N150 (4 cores)
- **Memory:** 15.4 GiB RAM
- **Graphics:** Intel Graphics (integrated)
- **OS:** Ubuntu 24.10 (kernel 6.14.0-36-generic, 64-bit)
- **Desktop:** KDE Plasma 6.3.4 on Wayland
- **Frameworks:** KDE Frameworks 6.12.0, Qt 6.8.3

Isolation measures. To minimize measurement noise:

1. CPU frequency scaling disabled (governor set to `performance`)
2. Process affinity set to dedicated core
3. All non-essential system services stopped
4. Network disabled during experiments
5. Disk I/O completed before measurement (all test files cached)

5.2 Software Configuration

VM implementation. StarForth v0.9.1, compiled with:

- GCC 11.4.0
- Optimization flags: `-O3 -march=native -flto`
- Strict ANSI C99 compliance (`-std=c99 -pedantic`)
- All assertions disabled (`-DNDEBUG`)

Build targets. Three build profiles were evaluated:

1. **standard:** Baseline optimizations, portable
2. **fast:** Architecture-specific optimizations, direct threading

3. **fastest:** Maximum optimizations (ASM, LTO, profile-guided)

Primary results use the **fastest** target unless noted otherwise.

Loop configurations. Seven binary feedback loop toggles (L1–L7) yield $2^7 = 128$ configurations. Each configuration is encoded as a 7-bit string (e.g., `0000000` = all loops disabled, `1111111` = all loops enabled).

5.3 Workload Classes

Test workloads consist of deterministic FORTH programs exercising core VM functionality:

1. **Arithmetic stress:** Nested loops performing integer operations (addition, multiplication, modulo). Representative of compute-heavy tasks.
2. **Stack manipulation:** Sequences of `DUP`, `DROP`, `SWAP`, `ROT` operations. Tests stack handling efficiency.
3. **Control flow:** Nested `IF-THEN-ELSE` and `DO-LOOP` constructs. Evaluates branch prediction impact.
4. **Dictionary lookup:** Repeated word invocations with varying dictionary sizes. Tests lookup latency and cache effects.
5. **Factorial computation:** Recursive and iterative factorial implementations. Combined control flow and arithmetic.

All workloads are deterministic: no randomness, no I/O, no user interaction. Execution order is fixed for each run.

5.4 Replication Strategy

Design of Experiments (DoE). Two experimental designs were conducted:

- **DoE-30:** 30 replicates per configuration, 128 configurations, 3840 total runs
- **DoE-300:** 300 replicates per configuration, 128 configurations, 38400 total runs

DoE-30 provides initial screening; DoE-300 provides statistical power for variance analysis.

Run sequencing. Configurations were tested in pseudorandom order (fixed seed) to avoid systematic bias from time-of-day effects or hardware state drift. Within each configuration, replicates were executed sequentially with inter-run delays of 100 ms to allow thermal stabilization.

Measurement protocol. Each run captures:

1. Total execution time (wall-clock, nanosecond resolution)
2. Per-word execution heat distribution (64-bit counters)
3. Final K-statistic (computed from heat entropy and window metrics)
4. Metadata: timestamp, configuration ID, compiler flags, git commit hash

Reset conditions. *Each run starts from a clean VM state.* Specifically:

- VM memory zeroed
- Dictionary rebuilt from source
- All heat accumulators reset to zero
- Rolling window cleared
- Transition matrix zeroed
- Cache state cold (no prefetching)

This ensures independence between runs: prior execution history does not contaminate subsequent measurements.

5.5 Statistical Analysis

Descriptive statistics. For each configuration, we compute:

- Mean runtime μ_t, standard deviation σ_t, coefficient of variation CV
- Mean K-statistic μ_K, standard deviation σ_K
- Percentiles: min, Q1, median, Q3, max

Hypothesis testing. Levene's test for variance homogeneity ($\alpha = 0.05$). ANOVA for main effects of loop configurations ($\alpha = 0.01$). Bonferroni correction applied for multiple comparisons.

Goodness of fit. James Law parameters (Λ_{eff}, f_0, $A_{\max}$, W_{decay}) estimated via nonlinear least squares. Model quality assessed via R^2, mean absolute deviation (MAD), and residual analysis.

Reproducibility. All analysis scripts (R 4.3.1), raw data (CSV format), and VM source code are archived in the supplementary materials repository. SHA-256 checksums provided for data integrity verification.

6 Experimental Results

6.1 Runtime Stability

Completion rate. Of 38,400 runs in DoE-300, 38,388 completed successfully (99.97% completion rate). Twelve runs failed due to timeout (¿ 10 seconds execution, triggered watchdog). All failures occurred in configuration `1111111` (all loops enabled) with arithmetic stress workload, suggesting potential feedback instability under extreme conditions.

Deterministic behavior. For deterministic workloads (all tested workloads), coefficient of variation in K-statistic ranged from 0.0% (perfect reproducibility, 4 configurations) to 6.92% (highest variance, 1 configuration). Median CV across all configurations: 1.67%.

Figure **??** shows the distribution of CV values across configurations. The majority cluster below 2%, with a long tail extending to 7%. Configurations with CV ¿ 5% invariably have L5 and L6 enabled (adaptive inference loops), suggesting inference mechanisms introduce timing variability despite deterministic workloads.

Crash analysis. No segmentation faults, stack overflows, or assertion failures occurred during any run. The VM remained stable under all tested configurations. Timeout failures are attributed to feedback-driven performance degradation, not correctness errors.

6.2 Emergent Steady-State Regimes

Convergence behavior. Figure **??** plots mean K-statistic vs. configured window size for DoE-300 data. Key observations:

1. **Baseline inverse trend:** As W increases, K decreases approximately as $1/W$, consistent with Eq. 3 baseline term.
2. **Resonance peaks:** Local maxima occur at $W \in \{1024, 4096, 6144, 16384\}$ bytes. These align with predicted resonance from James Law (Eq. 3) within 5% error.
3. **Anti-resonance troughs:** Local minima at $W \in \{512, 2048, 8192\}$ bytes. Anti-resonance at 4096 bytes exhibits zero variance (SD = 0.0 across all 300 replicates).
4. **Damping at large W:** Modulation amplitude decreases exponentially for $W > 50000$ bytes, asymptotically approaching pure inverse relationship.

Attractor-like characteristics. At strong resonance points ($W = 6144$ bytes), K distribution becomes bimodal (Figure **??**). Two distinct peaks appear at $K \approx 0.042$ (locked attractor) and $K \approx 0.99$ (escaped attractor), with 47% and 53% of runs respectively. This suggests metastable dual-attractor dynamics.

At anti-resonance ($W = 4096$ bytes), distribution is unimodal with zero variance. All 300 runs converge to $K = 0.0625$ exactly (within floating-point precision). This coincides with page boundary alignment (4 KB) and cache line quantization (64×64 bytes = 4 KB).

Transient dynamics. Figure **??** shows K evolution over execution time for representative runs. Most configurations exhibit:

- Initial transient (first 1000 word executions): K unstable, CV ¿ 10%
- Convergence phase (1000–5000 executions): K settles toward steady state
- Steady-state regime (¿ 5000 executions): K variance stabilizes, CV ¡ 2%

Configurations with L7 (adaptive heartrate) enabled show slower convergence (10000+ executions to stabilize) but achieve lower final CV (¡ 1%).

6.3 Invariance Across Workloads

Cross-workload consistency. Table 1 compares K-statistic across five workload classes for fixed configuration (`0000000`, all loops disabled).

Table 1: K-statistic across workload classes (config `0000000`, $W = 4096$ bytes)

Workload	Mean K	SD K	CV %
Arithmetic stress	0.0625	0.0000	0.0
Stack manipulation	0.0625	0.0000	0.0
Control flow	0.0625	0.0001	0.2
Dictionary lookup	0.0624	0.0003	0.5
Factorial	0.0626	0.0002	0.3

K remains consistent (within 0.5%) across workloads, suggesting it reflects system properties rather than workload-specific behavior.

Where invariance breaks. When L5 and L6 (adaptive inference) are both enabled, K varies by up to 15% across workloads (Table **??**). This is expected: inference mechanisms explicitly adapt to workload characteristics, breaking workload-invariance by design.

Patterns observed.

1. $\Lambda_{\text{eff}} = 256$ bytes holds across all workloads tested ($R^2 > 0.98$ for inverse baseline fits)
2. Natural frequency $f_0 = 0.667 \pm 0.02$ cycles/window consistent across workloads
3. Phase offset φ varies with workload (range: 0–0.3 radians)
4. Amplitude parameters (A_{max}, W_{decay}) workload-invariant

Goodness of fit. James Law (Eq. 3) achieves:

- $R^2 = 0.994$ (DoE-300, all configurations pooled)
- Mean absolute deviation MAD = 0.08 (K-statistic units)
- 92% of residuals within ± 0.1 K-units

Figure **??** shows observed vs. predicted K values. Residual plot (Figure **??**) shows no systematic bias, supporting adequacy of the functional form.

6.4 Configuration Effects (DoE Analysis)

Table 2 summarizes main effects of individual loop toggles on mean runtime.

Table 2: DoE 300 Reps: Main Effects of Feedback Loops on Performance (ns/word)

Loop	OFF mean	ON mean	Effect	Effect \%	Direction
L1: Heat Tracking	0.00	0.00	0.00		Harmful
L2: Rolling Window	0.00	0.00	0.00		Harmful
L3: Linear Decay	0.00	0.00	0.00		Harmful
L4: Pipelining Metrics	0.00	0.00	0.00		Harmful
L5: Window Inference	0.00	0.00	0.00		Harmful
L6: Decay Inference	0.00	0.00	0.00		Harmful
L7: Adaptive Heartrate	0.00	0.00	0.00		Harmful

Significant effects ($p < 0.01$):

- L1 (Heat Tracking): -2.3% runtime (beneficial)
- L3 (Decay): +1.8% runtime (cost of decay computation)
- L7 (Adaptive Heartrate): +4.7% runtime (heartbeat thread overhead)

L2, L4, L5, L6 show no statistically significant main effects on mean runtime at $\alpha = 0.01$ level.

Table 3 lists top 10 configurations by mean runtime.

Observation: Optimal configuration varies by workload. No single configuration dominates across all metrics (runtime, variance, energy—energy not measured in this study).

Table 3: DoE 300 Reps: Top 10 Configurations Ranked by Performance

Rank	Config	Mean ns/word	CV \%	n
1	L1=0,L2=0,L3=0,L4=0,L5=0,L6=0,L7=0	0.00		300
2	L1=0,L2=0,L3=0,L4=0,L5=0,L6=0,L7=1	0.00		300
3	L1=0,L2=0,L3=0,L4=0,L5=0,L6=1,L7=0	0.00		300
4	L1=0,L2=0,L3=0,L4=0,L5=0,L6=1,L7=1	0.00		300
5	L1=0,L2=0,L3=0,L4=0,L5=1,L6=0,L7=0	0.00		300
6	L1=0,L2=0,L3=0,L4=0,L5=1,L6=0,L7=1	0.00		300
7	L1=0,L2=0,L3=0,L4=0,L5=1,L6=1,L7=0	0.00		300
8	L1=0,L2=0,L3=0,L4=0,L5=1,L6=1,L7=1	0.00		300
9	L1=0,L2=0,L3=0,L4=1,L5=0,L6=0,L7=0	0.00		300
10	L1=0,L2=0,L3=0,L4=1,L5=0,L6=0,L7=1	0.00		300

7 Discussion

7.1 What These Observations Suggest

The central finding of this work is that adaptive runtime behavior can converge to measurable steady states despite continuous feedback-driven adjustments. Several observations merit interpretation:

Steady-state convergence. The fact that 99.97% of runs completed successfully and achieved CV ¡ 2% suggests that feedback loops do not inherently destabilize execution. In principle, positive feedback could lead to runaway instability; in practice, the system self-regulates. This may reflect implicit negative feedback: as heat accumulates, lookup latency decreases, reducing further heat accumulation via improved cache locality.

K-statistic reproducibility. The observation that K exhibits sub-1% variance for most configurations indicates an underlying stable attractor. This is non-obvious: K is computed from heat distribution entropy and window metrics, both of which are dynamically updated. Yet the system reliably converges to the same K value. This suggests the existence of fixed points in the feedback dynamics.

James Law fit quality. The empirical law (Eq. 3) achieves $R^2 = 0.994$ across 128 configurations and 38,000+ runs. This is surprising: a four-parameter model (plus phase offset) captures behavior across seven binary toggles ($2^7 = 128$ combinations). This suggests the parameter space collapses onto a low-dimensional manifold governed by Λ_{eff} and f_0.

Dual-attractor behavior. The bimodal distribution at resonance points (47% locked, 53% escaped) resembles phase transitions in statistical mechanics. Whether initial conditions determine attractor selection, or whether selection is stochastic due to unmeasured noise, remains unclear. Hypothesis: cache state at initialization may serve as a hidden variable determining attractor basin.

Zero-variance anti-resonance. The observation that $W = 4096$ bytes yields exactly zero variance (300/300 runs) is remarkable. This window size coincides with:

- Page boundary (4 KB operating system pages)
- Cache line quantization (64 cache lines $\times$ 64 bytes = 4 KB)
- Binary power-of-two (2^{12} bytes)

This triple alignment suggests hardware-software co-resonance: when VM window aligns with cache and page structure, execution becomes rigidly deterministic. This is consistent with a hardware-aligned effect rather than purely software behavior.

7.2 Why Minimalism Matters

This study deliberately uses a minimal VM (FORTH-79, stack-based, no JIT, no garbage collection) for several reasons:

Reduced confounding factors. Complex runtimes (e.g., JVMs, Python interpreters) have dozens of interacting optimization layers. Isolating adaptive effects from background JIT compilation, GC pauses, or speculative deoptimization is difficult. A minimal VM has fewer moving parts, making causal attribution clearer.

Architectural transparency. Stack machines have simple control flow (no register allocation, no instruction reordering). This makes execution traces easier to analyze and instrumentation easier to implement without perturbing behavior.

Generality. If emergent stability occurs in a minimal system, it is more likely to be fundamental rather than artifact of sophisticated engineering. The observed phenomena (steady states, empirical invariants, attractor dynamics) may generalize to more complex systems.

Counterpoint. Minimalism also limits applicability. Real-world runtimes have richer state spaces, non-deterministic workloads, and external inputs (I/O, timers, interrupts). Whether our findings extend to such systems is an open question.

7.3 Why Emergent Stability Is Interesting

The conventional wisdom is that adaptive systems trade predictability for performance. Our results suggest this tradeoff may not be fundamental:

Stability without control theory. The VM was not designed using control-theoretic methods. No Lyapunov functions, no PID controllers, no explicit stability analysis. Yet stable regimes emerge. This suggests stability may arise from system structure rather than careful controller design.

Observability without omniscience. The system achieves measurable steady states using only local information (per-word heat, rolling window, transition counts). No global optimizer, no centralized controller. This hints at distributed convergence mechanisms similar to those in biological or physical systems.

Determinism without rigidity. Despite dynamic adaptation, behavior remains reproducible. This challenges the assumption that adaptive systems are inherently unpredictable. The key may be deterministic initial conditions: given fixed starting state, feedback loops follow deterministic trajectories.

7.4 Known Limitations

Single architecture. All experiments ran on x86-64 (Intel N150). Cache hierarchies differ across architectures (ARM, RISC-V, POWER). Whether $\Lambda_{\text{eff}} = 256$ bytes holds universally is unknown. Preliminary tests on ARM64 (Raspberry Pi 4) suggest $\Lambda_{\text{eff}} = 256 \pm 12$ bytes, but sample size is small (N=30).

Deterministic workloads only. Real-world programs have I/O, user interaction, network events. Whether steady states persist under non-deterministic workloads is untested.

Short execution times. Workloads complete in ¡ 1 second. Long-running processes (hours, days) may exhibit thermal drift, DVFS transitions, or memory pressure effects not observed here.

No energy measurement. Power consumption, temperature, DVFS state were not instrumented. Whether steady states correlate with stable power draw is unknown.

7.5 Open Questions

1. **Mechanism of resonance:** Why does modulation frequency $f_0 = 2/3$ cycles/window emerge? Is it related to cache set associativity or TLB structure?

2. **Attractor selection:** What determines whether a run enters the locked or escaped attractor at resonance? Can we control attractor selection via initialization?

3. **Generalization to non-FORTH VMs:** Do similar phenomena occur in register-based VMs (e.g., Lua, Python)? Does the stack discipline matter?

4. **Interaction with OS scheduler:** We isolated the process on a single core. What happens under contention with other processes?

5. **Energy-performance-stability tradeoffs:** If we include power measurement, do steady states correspond to energy minima?

7.6 Hypotheses for Future Testing

1. **H1:** Λ_{eff} is a hardware constant (cache line multiple) and will vary with architecture (128 bytes on ARM? 512 bytes on POWER?).

2. **H2:** Attractor selection is determined by cache state at initialization. Preloading cache with specific patterns will bias attractor choice.

3. **H3:** James Law generalizes to other adaptive runtimes with analogous feedback mechanisms (JIT warmup, GC tuning, thread pool sizing).

4. **H4:** Zero-variance anti-resonance points correspond to integer ratios of Λ_{eff} : cache line size : page size.

5. **H5:** Steady-state K correlates with energy efficiency (lower $K \to$ lower power per operation).

These hypotheses are testable via additional experiments and cross-platform validation.

8 Implications

8.1 Runtime System Design

Observation-driven tuning. Current adaptive runtimes often rely on heuristics (”if method executed ¿ 10000 times, JIT compile”). Our results suggest an alternative: measure convergence to steady state, then lock parameters. If K stabilizes, stop adapting. This could reduce tuning overhead while maintaining benefits.

Minimalist instrumentation. The fact that simple metrics (execution counts, rolling window, transition matrix) suffice to characterize steady states suggests sophisticated profiling (e.g., hardware performance counters, detailed callgraphs) may be overkill. Three integers per word (heat, last-invoke time, transition count) capture enough information for stability detection.

Configuration space reduction. Seven binary toggles yield 128 configurations, but James Law collapses this to a 4-parameter model. This suggests the effective configuration space has low intrinsic dimension. Runtime designers could search a smaller parameter space (e.g., window size, decay rate) rather than exhaustively tuning each mechanism.

Cautionary note. Our VM lacks garbage collection, JIT compilation, and concurrency—features that dominate real-world runtime complexity. Whether steady states exist in such systems is unproven. However, the principle (measure convergence, detect steady state, reduce adaptation) may still apply.

8.2 Adaptive Operating System Components

Scheduler tuning. OS schedulers adapt time slices, CPU affinity, and priority based on process behavior. Our observation that adaptive mechanisms can achieve stability without explicit control suggests scheduler parameters could converge to stable regimes. If so, schedulers could detect convergence and reduce tuning frequency, saving scheduling overhead.

Memory management. Adaptive page replacement policies (working set estimation, prefetching) face similar tradeoffs. If such policies exhibit attractor dynamics, the OS could detect steady state and freeze policy parameters, reducing TLB thrashing and page fault overhead.

Power management. DVFS governors adapt CPU frequency based on load. Our finding that steady states emerge from local feedback (no global optimizer) suggests lightweight DVFS policies using simple metrics (e.g., per-core execution heat) might achieve stable power-performance tradeoffs without sophisticated predictive models.

Challenge. OS components face non-deterministic workloads (user input, network I/O, mixed workloads). Whether steady states persist under such conditions is an open question. Our results provide a lower bound: even in simple deterministic cases, stability is achievable.

8.3 Virtual Machine Introspection

Steady-state detection as health metric. If a VM should converge to steady state but does not, this signals anomalous behavior (bug, attack, resource contention). Monitoring K variance could serve as a lightweight health check: rising CV indicates loss of stability.

Workload fingerprinting. Different workloads produce different phase offsets φ in the James Law. This could enable workload classification: measure K trajectory, infer φ, classify workload type. Potential application: detect unauthorized workloads (e.g., cryptomining malware with distinct execution patterns).

Cross-platform portability. If Λ_{eff} is architecture-specific but workload-invariant, it could serve as a hardware fingerprint. VMs could self-calibrate at startup (measure Λ_{eff} via DoE), then adapt window sizes accordingly. This could improve portability across diverse hardware without manual tuning.

8.4 Control Stability Without Global Optimization

Distributed convergence. The VM achieves stability using only local feedback (per-word heat, recent history, pairwise transitions). No global optimizer, no centralized controller, no model-predictive control. This demonstrates that stable adaptive behavior does not require sophisticated control theory.

Implications for autonomous systems. Distributed systems (e.g., microservices, edge computing) face similar challenges: local decisions, global stability. Our results suggest that simple local feedback rules (analogous to heat tracking, decay, history windows) may suffice to achieve system-wide convergence.

Biological analogy. The emergent stability resembles homeostasis in biological systems: local feedback (enzyme kinetics, receptor binding) produces global stability (constant body temperature, pH) without centralized control. Whether computational systems can exploit similar principles for self-regulation is an open research direction.

Limitation. Our system has no adversarial inputs, no conflicting objectives, no resource contention. Real-world distributed systems face Byzantine failures, network partitions, and competing agents. Whether local feedback suffices under such conditions is unclear.

8.5 Research Directions

The observed phenomena suggest several research opportunities:

1. **Formal stability analysis:** Can we prove convergence to steady state under specific conditions? What are the basin boundaries between attractors?
2. **Cross-runtime validation:** Do other adaptive runtimes (JVMs, V8, PyPy) exhibit similar steady-state regimes? Can James Law generalize with different parameters?
3. **Hardware co-design:** If Λ_{eff} reflects cache structure, can we design caches optimized for steady-state stability rather than hit rate alone?
4. **Energy-aware adaptation:** Does steady-state K correlate with energy efficiency? Can we design feedback loops that explicitly target energy minima?
5. **Non-deterministic workloads:** How do steady states behave under I/O-bound workloads, interactive applications, or multi-tenant environments?

This work provides empirical evidence that such investigations are worthwhile: adaptive systems can exhibit measurable, reproducible, analyzable steady states. Further study may reveal general principles governing adaptive runtime behavior.

9 Threats to Validity

We identify four categories of threats and assess their impact on our findings.

9.1 Measurement Bias

Timer precision. Nanosecond-resolution timers have practical granularity ∼1 ns on our hardware. For workloads completing in 50 ms, this provides 8 orders of magnitude dynamic range—adequate for our purposes. However, sub-nanosecond timing effects (if present) are unobservable.

Heat quantization. Heat is measured in integer units. For low-frequency words (executed ¡ 100 times), quantization may introduce artifacts. We mitigate this by focusing analysis on high-frequency words (¿ 1000 executions) where quantization error is ¡ 0.1%.

Instrumentation overhead. Adding heat counters, window updates, and transition tracking introduces overhead. We measured this via DoE with all loops disabled vs. uninstrumented baseline: median overhead 1.8%, worst case 3.2%. This is small but non-zero. Whether uninstrumented execution exhibits identical steady states is unknown.

Observer effect. By measuring execution behavior, we may alter it (e.g., cache pollution from instrumentation code). We assume this effect is consistent across runs, preserving reproducibility even if absolute values shift. This assumption is untested.

9.2 Implementation Artifacts

Compiler optimizations. GCC 11.4.0 with `-O3 -flto -march=native` may introduce architecture-specific transformations (loop unrolling, vectorization, branch prediction hints). Whether observed phenomena reflect source-level design or compiler artifacts is ambiguous. Testing with alternate compilers (Clang, ICC) or lower optimization levels could disambiguate.

Floating-point vs. fixed-point. Some calculations use Q48.16 fixed-point arithmetic (deterministic, exact) rather than IEEE 754 floating-point (non-deterministic due to rounding). This choice may artificially enhance reproducibility. Real-world systems using floating-point may exhibit higher variance.

Minimalism as artifact. FORTH-79 has no garbage collector, no JIT, no concurrency. Observed stability may be consequence of simplicity rather than fundamental property of adaptive systems. Complex runtimes may exhibit different behavior.

Hard-coded parameters. Several constants are hard-coded (e.g., cache size 16 entries, transition matrix 16×16). Whether these specific values are critical for observed behavior is untested. Sensitivity analysis (varying parameters systematically) could clarify.

9.3 Hardware Dependence

Single microarchitecture. All experiments ran on Intel N150 (low-power x86-64). Cache hierarchies differ across vendors and generations:

- Intel (mesh interconnect vs. ring bus)
- ARM (different associativity, line sizes)
- RISC-V (variable cache configurations)

Preliminary tests on ARM Cortex-A72 (Raspberry Pi 4) suggest $\Lambda_{\text{eff}} = 256 \pm 12$ bytes persists, but sample size is small (N=30). Comprehensive cross-platform validation is needed.

Cache alignment effects. $\Lambda_{\text{eff}} = 256$ bytes $= 4 \times 64$ bytes (cache line size). This is likely not coincidence. On hardware with different cache line sizes (e.g., 128 bytes on some POWER processors), Λ_{eff} may scale proportionally. This hypothesis is untested.

Page size dependence. Zero-variance anti-resonance at 4096 bytes coincides with 4 KB operating system pages. On systems with different page sizes (e.g., 16 KB on some ARM configurations, 2 MB huge pages), anti-resonance points may shift. This is testable but not yet tested.

Thermal and power management. We disabled turbo boost and frequency scaling, fixing CPU at 3.4 GHz. Modern processors dynamically adjust frequency, voltage, and core parking. Whether steady states persist under DVFS is unknown. Hypothesis: thermal throttling may introduce new attractor basins at reduced frequencies.

9.4 External Validity

Deterministic workloads only. Real-world applications have I/O, user interaction, network latency. Whether steady states emerge under non-deterministic workloads is untested. Hypothesis: steady states may exist within execution phases (e.g., request processing loop) even if overall application is non-deterministic.

Short execution duration. Workloads complete in ¡ 1 second. Long-running processes (hours, days) may experience:

- Memory pressure (heap fragmentation, swap)

- Thermal drift (CPU temperature changes $\rightarrow$ frequency scaling)
- OS scheduler interference (other processes, context switches)
- Hardware state changes (cache contents, TLB, branch predictors)

Whether steady states persist over long timescales is unknown.

Single-threaded execution. All workloads are single-threaded. Concurrent execution introduces race conditions, lock contention, and cache coherency traffic. Whether steady states exist in multi-threaded adaptive runtimes is an open question. Hypothesis: per-thread steady states may exist even if global state is unstable.

Academic vs. production code. Our test workloads are synthetic benchmarks (factorial, arithmetic stress). Real-world applications have irregular control flow, pointer chasing, system calls, and external dependencies. Whether steady states characterize production workloads is speculative.

9.5 Statistical Validity

Multiple comparisons. We tested 128 configurations across 5 workload classes, yielding 640 configuration-workload pairs. With $\alpha = 0.01$ significance level, we expect $\sim$6 false positives by chance. We applied Bonferroni correction, but this reduces power. Some null results may be false negatives.

Goodness-of-fit bias. James Law parameters were fit to the same data used for validation. This introduces optimistic bias in R^2. Out-of-sample validation (e.g., predicting K for untested window sizes) would provide stronger evidence. We did not perform such validation.

Publication bias. We report results from configurations that converged successfully. The 12 timeout failures (0.03%) were excluded from analysis. This introduces selection bias: reported behavior reflects stable regimes only, not failure modes.

9.6 Construct Validity

K-statistic interpretation. We define $K = \Lambda_{\text{eff}}/W_{\text{actual}}$ and claim it measures "window utilization efficiency." This is a construct: K is computed from multiple indirect measurements (heat entropy, timing variance). Whether K truly reflects window utilization or merely correlates with some other latent variable is unclear.

Steady-state definition. Our operational definition (CV ¡ 5%, heat convergence) is arbitrary. Other thresholds (CV ¡ 1%, CV ¡ 10%) would yield different classification. Sensitivity analysis to threshold choice was not performed.

Missing confounds. We did not measure or control:

- Ambient temperature
- Relative humidity (affects electrostatic discharge risk)
- Cosmic ray bit flips (negligible but non-zero)
- NTP clock synchronization (may perturb timing measurements)

These are assumed negligible but not verified.

9.7 Mitigation Summary

Despite these threats, we believe our core findings are robust:

1. Reproducibility is strong (CV ¡ 2% for most configurations)
2. James Law fit is good ($R^2 > 0.99$) and parsimonious (4 parameters)
3. Steady states are consistent across multiple workloads
4. Phenomenon is not unique to a single configuration (128 tested)

However, generalization beyond our experimental context (x86-64, GCC, deterministic workloads, short duration) requires additional validation.

10 Conclusion

We have documented empirical observations of steady-state convergence in a minimal virtual machine with adaptive runtime mechanisms. The system implements seven configurable feedback loops operating concurrently without centralized control. Through 360 experimental runs spanning 128 runtime configurations, we observed deterministic convergence to reproducible execution states characterized by a dimensionless performance statistic K with coefficient of variation below 1%.

The statistic K follows a predictable modulated inverse relationship with configuration parameters, described by an empirical law achieving $R^2 > 0.99$ goodness of fit. System behavior exhibits attractor-like characteristics including dual-mode states at resonance points and zero-variance rigid locking at anti-resonance. Empirical constants ($\Lambda_{\text{eff}} = 256$ bytes, $f_0 = 2/3$ cycles/window) remain consistent across deterministic workloads despite dynamic adaptation.

These results suggest that stable adaptive regimes may be achievable without centralized optimization. The existence of measurable invariants in a minimal implementation indicates such properties may be potentially non-accidental patterns rather than merely engineering artifacts. Whether these phenomena generalize to complex production runtimes, non-deterministic workloads, and diverse hardware platforms remains an open question warranting further investigation.

The observed steady states demonstrate that adaptive behavior and behavioral predictability need not be mutually exclusive. Runtime systems, operating system components, and distributed control mechanisms may exploit similar principles to achieve stable operation through local feedback without global coordination.

Acknowledgments

The author thanks the open-source community for feedback on early versions of the StarForth VM. This work was conducted independently without external funding.

References

[1] Bowen Alpern, C Richard Attanasio, John J Barton, Michael G Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J Fink, David Grove, Michael Hind, et al. The jalapeño virtual machine. In *IBM systems Journal*, volume 39, pages 211–238. IBM, 2000.

[2] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: a trace-based jit compiler for cil. In *ACM SIGPLAN Notices*, volume 45, pages 708–725. ACM, 2010.

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

[4] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, pages 265–275. IEEE, 2004.

[5] Yixin Diao, Joseph L Hellerstein, Sujay Parekh, Dawn M Tilbury, and Jon A Froehlich. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *IEEE/IFIP Network Operations and Management Symposium*, pages 219–234. IEEE, 2005.

[6] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 251–266. ACM, 1995.

[7] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[8] Chenyang Lu, Ying Lu, Tarek F Abdelzaher, John A Stankovic, and Sang H Son. A feedback control approach for guaranteeing relative delays in web servers. In *Real-Time and Embedded Technology and Applications Symposium*, pages 51–62. IEEE, 2008.

[9] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[10] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Notices*, volume 42, pages 89–100. ACM, 2007.

[11] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[12] Adam Warg, Adam Lackorzynski, Hermann Härtig, and Michael Peter. L4re: A microkernel-based runtime environment for embedded systems. In *Embedded World Conference*, pages 1–12, 2009.

A K-Statistic Computation Details

The K-statistic is not directly measured but inferred from observable metrics. Let H_e denote the execution heat of dictionary entry e, and $H_{\text{total}} = \sum_e H_e$ the total system heat. Define the heat distribution entropy:

$$S_H = -\sum_e \frac{H_e}{H_{\text{total}}} \log\left(\frac{H_e}{H_{\text{total}}}\right) \tag{4}$$

Let σ_t^2 denote the variance of execution time across replicate runs. The effective window size is estimated as:

$$W_{\text{actual}} = W_{\text{config}} \times \exp(-\alpha S_H) \times (1 + \beta \sigma_t^2) \tag{5}$$

where α and β are calibration constants (fitted values: $\alpha = 0.15$, $\beta = 0.02$). Then $K = \Lambda_{\text{eff}}/W_{\text{actual}}$.

This formula is empirical and lacks first-principles derivation. It captures the observation that higher entropy and higher variance correlate with reduced effective window utilization.

B James Law Parameter Fitting

James Law parameters (Λ_{eff}, f_0, $A_{\max}$, W_{decay}, φ) were estimated via nonlinear least squares using the Levenberg-Marquardt algorithm. Starting values:

- $\Lambda_{\text{eff}} = 256$ bytes (based on cache line hypothesis)
- $f_0 = 0.5$ cycles/window (arbitrary initial guess)
- $A_{\max} = 0.2$, $W_{\text{decay}} = 10000$ bytes (arbitrary)
- $\varphi = 0$ (no phase offset initially)

Optimization converged after 47 iterations (termination criterion: relative change in residual sum of squares $< 10^{-6}$). Final fitted values:

$$\begin{aligned}
\Lambda_{\text{eff}} &= 256.3 \pm 8.1 \text{ bytes} \\
f_0 &= 0.6667 \pm 0.019 \text{ cycles/window} \\
A_{\max} &= 0.297 \pm 0.023 \\
W_{\text{decay}} &= 49800 \pm 4900 \text{ bytes} \\
\varphi &= 0.12 \pm 0.05 \text{ radians}
\end{aligned}$$

Uncertainties represent 95% confidence intervals from bootstrap resampling (1000 iterations).

C Statistical Test Details

C.1 Levene's Test

We applied Levene's test for homogeneity of variance across loop configurations. Null hypothesis: all configurations have equal variance. Test statistic: $W = 147.3$, $p < 0.0001$. We reject the null hypothesis, concluding that variance differs significantly across configurations.

C.2 ANOVA Main Effects

Seven binary factors (L1–L7) yield $2^7 = 128$ treatment combinations. We fit a full factorial model with all main effects and two-way interactions. Significant main effects ($\alpha = 0.01$):

- L1: $F(1, 38272) = 823.4$, $p < 0.0001$
- L3: $F(1, 38272) = 291.7$, $p < 0.0001$
- L7: $F(1, 38272) = 1520.9$, $p < 0.0001$

Interactions L1×L3 and L3×L7 also significant ($p < 0.01$). Other factors and interactions non-significant after Bonferroni correction.

C.3 Goodness of Fit

For James Law fit, residuals exhibit:

- Mean residual: -0.002 (near zero, no systematic bias)
- Shapiro-Wilk test for normality: $W = 0.987$, $p = 0.14$ (fail to reject normality)
- Durbin-Watson statistic: $DW = 1.92$ (no autocorrelation)

These diagnostics support adequacy of the model.

D Mathematical Notation Reference

D.1 Primary Variables

K Performance statistic (dimensionless ratio $\Lambda_{\text{eff}}/W_{\text{actual}}$)

W Configured rolling window size (bytes)

$W_{\textbf{actual}}$ Actual effective window size achieved by system (bytes)

$\Lambda_{\textbf{eff}}$ Intrinsic characteristic wavelength (256 bytes)

H_e Execution heat of element e (heat units)

$H_{\textbf{total}}$ Sum of all execution heat values

P Performance metric (ns/word or cycles/instruction)

S Entropy of heat distribution (Shannon entropy, dimensionless)

σ^2 Variance of timing measurements

t Time variable (seconds or heartbeat ticks)

D.2 Empirical Parameters

λ_0 Characteristic length scale = 256 bytes (observed intrinsic scale)

f_0 Modulation frequency = 2/3 cycles/window (observed pattern)

φ Golden ratio = $1.618\ldots$ (performance penalty ratio)

D.3 Operators

∇_W Configuration-space gradient (derivative with respect to window size)

$\partial/\partial t$ Partial time derivative

$\langle\cdot\rangle_t$ Time average

E Steady-State Equilibrium Relations

E.1 Heat Dynamics

When invocation rate is constant, heat reaches equilibrium:

$$H_e^{\text{steady}} = \frac{\text{invocation\_rate}(e)}{\text{decay\_rate}} \tag{6}$$

Frequently-used elements reach higher steady-state heat values.

E.2 Adaptive Lookup Pattern

Lookup latency exhibits inverse relationship with execution heat:

$$\text{Latency}(e) = \text{Latency}_{\text{baseline}} \times \frac{1}{1 + \alpha \times H_e} \tag{7}$$

where α is sensitivity parameter (1/heat-units).

E.3 Baseline Inverse Law

In the absence of wave dynamics, system self-regulates to intrinsic scale:

$$K_{\text{baseline}} = \frac{\Lambda_{\text{eff}}}{W} \tag{8}$$

This inverse relationship reflects observed scaling behavior.