

Efficient Series History and Incremental Technical Analysis for Bar-Loop Interpreters

HOOX Research / PYNE Contributors
jango_blockchained
Independent Open Source Research
contact@hoox.sh

August 2026

Abstract

Bar-loop interpreters for series-oriented domain-specific languages evaluate every statement once per time bar and materialize history via the operator $x[n]$ (“ n bars back”). Naive implementations store unbounded per-series lists and recompute rolling technical-analysis (TA) windows from scratch on each bar, inducing $O(n \cdot s)$ memory growth and $O(n \cdot p)$ work for period- p indicators over n bars and s series. We present the series-history and incremental-TA stack shipped in PYNE, an independent open toolchain for the Pine Script™ language [HOOX Project, 2026, TradingView, Inc., 2024]. The design combines (i) a default-on series cap that trims chronological host lists to $\max(SERIES_MAX, max_bars_back) + slack$, yielding $\sim 78\times$ long-run list-memory reduction at 20 000 bars; (ii) an opt-in chronological ring buffer (`ChronologicalSeriesBuffer`) with true $O(1)$ lookback under modular indexing; (iii) call-site incremental kernels for the hot `ta` surface—including SMA/E-MA/RMA, RSI, Bollinger Bands, highest/lowest, KAMA, and StochRSI—with kernel micro-benchmarks up to $\sim 717\times$ versus full-window rebuilds; and (iv) host-side parse/AST caching, lazy calendar materialization, and light plot modes that cut multi-plot residual wall time. On the interpret path at 2 000 bars we report median latencies of 15.2 ms (minimal), 23.6 ms (`ta_sma`), 153 ms (`ta_combo`), and 69 ms (`strategy_ish`); cProfile attributes $\sim 75\%$ of residual `ta_combo` wall time to plot packing rather than TA kernels. We formalize correctness conditions under bounded history, give recurrences and stability notes for each kernel class, and situate the work in incremental computation and streaming algorithms.

Keywords. incremental computation; technical analysis; series buffers; ring buffers; memory-bounded interpreters; financial time series.

Disclaimer. Pine Script™ and TradingView® are trademarks of TradingView, Inc. PYNE is an independent, unofficial implementation and is not affiliated with, authorized by, sponsored by, or endorsed by TradingView, Inc.

Contents

1	Introduction	3
2	Background	3
2.1	Pine series semantics	3
2.2	Bar-loop evaluation model	4
2.3	Cost model of interpretation	4

3	Problem Formalization	5
3.1	Memory: unbounded series growth	5
3.2	Time: full-window TA recompute	5
3.3	Correctness constraints	5
4	Series Cap Design	6
4.1	Policy	6
4.2	Correctness argument for bounded history	6
4.3	When lookback exceeds the cap	6
4.4	Structural memory win	6
5	Ring Buffer Series	7
5.1	ChronologicalSeriesBuffer	7
5.2	Dual-write transition	7
6	Incremental TA Framework	8
6.1	Generic pattern	8
6.2	Case study: SMA (sliding sum)	8
6.3	Case study: EMA and RMA (Wilder)	9
6.4	Case study: RSI	9
6.5	Case study: Bollinger Bands (mean + variance)	9
6.6	Case study: highest / lowest (monotonic deque)	10
6.7	Case study: KAMA	10
6.8	Case study: StochRSI	10
6.9	Additional kernels	11
6.10	Numerical stability notes	11
7	Interpreter Hot Paths	11
7.1	Dispatch	11
7.2	Plot packing	11
7.3	Light plots and lazy calendar	11
8	Parse/AST Caching Interaction with Runtime Multi-Run	12
9	Evaluation	13
9.1	Methodology	13
9.2	Memory	13
9.3	Kernel speedups	13
9.4	End-to-end interpret latency	13
9.5	Compile path comparison	14
9.6	Feature flags summary	14
10	Related Work	14
11	Limitations	15
12	Conclusion	15
A	Memory bound (informal)	16
B	Environment reproduction notes	16

1 Introduction

Financial charting languages such as Pine Script™ [TradingView, Inc., 2024] expose a *series* data model in which every scalar expression is implicitly a time series indexed by bar. Evaluation proceeds as a *bar loop*: the host advances an OHLCV cursor and re-executes the script body for each bar $i \in \{0, \dots, n-1\}$. History access $x[n]$ denotes the value of x that was current n bars earlier; qualifiers `var` and `varip` persist state across bars and intra-bar ticks respectively; and the sentinel `na` propagates through arithmetic [TradingView, Inc., 2024, HOOX Project, 2026].

This model is attractive for domain users—indicators are written as if they were spreadsheet formulas—but is expensive for a naive interpreter. Historically, the dominant costs on PYNE’s interpret path were:

1. **Series history growth.** Every named series and host OHLCV field appended one sample per bar, so memory scaled as $\Theta(n \cdot s)$ for s live series.
2. **Full-window TA recompute.** Indicators such as $\text{SMA}(p)$, Bollinger Bands, and Kaufman adaptive moving average [Kaufman, 2013] rebuilt period- p windows from the entire prefix on each bar, producing $\Theta(n \cdot p)$ or even $\Theta(n^2)$ work for nested rebuilds.
3. **Plot packing and call dispatch.** Multi-plot scripts spent a majority of residual wall time in result capture and `visit_Call` envelopes rather than pure arithmetic.

PYNE addresses these costs with a layered runtime stack measured across performance rounds 4–7 [HOOX Project, 2026]. Contributions of this paper:

- **Bounded series history** via default-on `PYNE_SERIES_CAP`, with a correctness argument relating lookback depth, recursive incremental state, and out-of-range `na` semantics (Section 4).
- **Chronological ring buffers** (`ChronologicalSeriesBuffer` / `PYNE_SERIES_RING`) enabling $O(1)$ Pine-offset lookback and a dual-write migration path off newest-first dequeues (Section 5).
- **A generic incremental TA framework** of call-site state tuples σ and updates $\sigma' = \text{update}(\sigma, x_i)$, covering sliding sums, exponential smoothers, Wilder-class oscillators, variance bands, amortized dequeues, and adaptive kernels (Section 6).
- **Interpreter and multi-run host optimizations:** type-keyed dispatch, light plots, lazy calendar, and a SHA-256 LRU AST cache with call-site scrubbing on hit (Sections 7–8).
- **Empirical evaluation** of memory, kernel micros, and end-to-end latency, including a residual bottleneck map that isolates plot packing as $\sim 75\%$ of multi-plot interpret wall time (Section 9).

Scope. We focus on the *interpret* (AST-walking) path and on the series/TA subsystem shared with the Numba compile path [Lam et al., 2015], which always emits incremental kernels for hot TA. Grammar, strategy broker semantics, and LSP tooling are treated elsewhere in the PYNE paper series.

2 Background

2.1 Pine series semantics

Definition 2.1 (Series and history operator). A series $\mathbf{x}$ is a sequence $(x_0, x_1, \dots, x_i)$ of values observed up to the current bar index i . The history operator is

$$x[n] = \begin{cases} x_{i-n} & \text{if } 0 \leq n \leq i \text{ and } x_{i-n} \neq \text{na}, \\ \text{na} & \text{otherwise (including } n < 0). \end{cases} \quad (1)$$

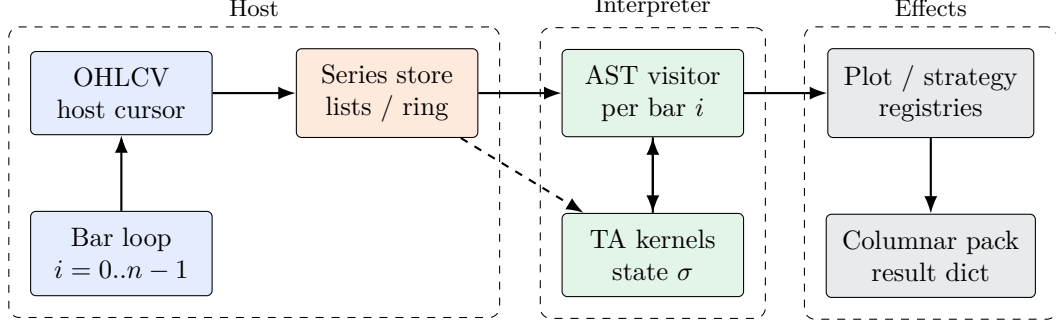


Figure 1: Bar-loop interpret architecture with series store, call-site TA state, and plot packing. The host advances OHLCV once per bar; the visitor re-executes the script body; incremental kernels read/write state σ keyed by call site.

Pine indices are *most-recent-first* in the programmer model ($x[0]$ is current), while host chronological lists store oldest-first so that physical index $-(n + 1)$ realizes lookback n [TradingView, Inc., 2024].

Definition 2.2 (Persistence qualifiers). The qualifier `var` initializes a declaration site once and carries the value across subsequent bars. `varip` additionally carries across ticks within an incomplete bar. Bare assignment re-evaluates every bar.

Definition 2.3 (na propagation). Arithmetic and many builtins propagate `na`: if any operand is `na`, the result is `na` (with Pine-specific exceptions for functions that treat missing samples specially). Out-of-range history must yield `na`, never a silent zero [HOOX Project, 2026].

These rules interact with memory bounding: truncating history deeper than any live look-back must not invent samples, and recursive smoothers must either retain enough raw history *or* retain closed-form incremental state (Section 4).

2.2 Bar-loop evaluation model

Figure 1 sketches the interpret architecture. For each bar i :

1. The host advances OHLCV series (`open`, `high`, `low`, `close`, `volume`, `time`) and optional script-local series.
2. The evaluator walks the script AST via a visitor, resolving names against a context map and call-site TA state.
3. Builtins (notably `ta`) either update incremental kernels or fall back to full-window helpers when `PYNE_TA_INCREMENTAL = 0`.
4. Plot/strategy side effects append into result registries for packing into columnar outputs.

The compile path lowers the same semantics to NumPy/Numba kernels that process entire arrays in one call [Harris et al., 2020, Lam et al., 2015]; that path always uses incremental kernels for hot TA and is not gated by the interpret flag.

2.3 Cost model of interpretation

Let n be the number of bars, s the number of live series (host + local), c the number of call nodes executed per bar, p a typical TA period, and q the number of `plot` (or similar) emissions per bar. A crude cost model for a naive interpreter is

$$T_{\text{naive}}(n) = \underbrace{O(n \cdot c \cdot d)}_{\text{AST dispatch depth } d} + \underbrace{O(n \cdot \sum_k p_k)}_{\text{full-window TA}} + \underbrace{O(n \cdot q \cdot w)}_{\text{plot capture width } w} + \underbrace{O(n \cdot s)}_{\text{series append}}. \quad (2)$$

Memory is $M_{\text{naive}} = \Theta(n \cdot s)$ plus plot buffers $\Theta(n \cdot q)$.

With series caps of capacity K , incremental TA of $O(1)$ or $O(p)$ amortized per call, and light plots that skip columnar capture, the improved model is

$$T_{\text{opt}}(n) = O(n \cdot c \cdot d') + O(n \cdot \sum_k u_k) + O(n \cdot q \cdot w') + O(n \cdot s), \quad (3)$$

where $u_k \in \{1, p_k\}$ is the per-bar update cost of kernel k , $d' \leq d$ after dispatch tuning, $w' \ll w$ under light plots, and $M_{\text{opt}} = \Theta(K \cdot s) + \Theta(|\sigma|)$ with $|\sigma|$ the aggregate incremental state (typically $O(p)$ per call site).

Dominant historical costs. Profiling rounds identified four hotspots: series list growth, TA full recompute, plot packing, and the `visit_Call` envelope. Kernel-only wins therefore do not automatically dominate end-to-end multi-plot scripts (Section 9).

3 Problem Formalization

3.1 Memory: unbounded series growth

Proposition 3.1 (Linear series memory). *Under uncapped chronological lists, if a script maintains s series each of length n after n bars, then pointer storage alone is $\Theta(n \cdot s)$ words. For $n = 20\,000$, $s = 6$ OHLCV-style lists, this is $\sim 9.6 \cdot 10^5$ pointers (~ 7.7 MiB on 64-bit), exclusive of object payloads.*

Many Pine scripts also allocate local series for intermediate expressions; s can reach dozens to hundreds on complex strategies, so memory pressure is not limited to OHLCV.

3.2 Time: full-window TA recompute

Definition 3.2 (Full-window recompute). A kernel f_p is a *full-window recompute* if at bar i it scans $\{x_{i-p+1}, \dots, x_i\}$ (or the entire prefix) without reusing work from bar $i - 1$, costing $\Omega(p)$ or $\Omega(i)$ arithmetic per bar.

Proposition 3.3 (Quadratic nested rebuild). *If each bar i rebuilds an indicator by scanning a prefix of length i (e.g., nested KAMA or StochRSI implementations that re-seed from bar 0), total work over n bars is $\Theta(n^2)$.*

Even “only” $\Theta(n \cdot p)$ is painful for $n = 5\,000$ and $p = 200$ when dozens of indicators share a script. Time-series textbooks emphasize online/streaming updates for exactly this reason [Box and Jenkins, 1970, Tsay, 2010].

3.3 Correctness constraints

Any optimization must preserve:

1. **Lookback identity:** for all $n \leq K$ (cap), $x[n]$ equals the uncapped value when the sample exists.
2. **na discipline:** OOB and missing samples are `na`, never 0.
3. **TA parity:** incremental kernels match full-window oracles within floating-point noise [Higham, 2002, IEEE Computer Society, 2019] under default flags.
4. **Call-site isolation:** two syntactically distinct calls to `ta.sma(close, 14)` maintain independent state.

4 Series Cap Design

4.1 Policy

PYNE formalizes host list capping behind `PYNE_SERIES_CAP` (default `on`). Let

$$K = \max(K_0, \text{max_bars_back}(\text{src})), \quad (4)$$

where $K_0 = \text{SERIES_MAX}$ defaults to 256 (overridable via `PYNE_SERIES_MAX`) and `max_bars_back` is parsed from the script source when present. Chronological lists are grown to $K + \Delta$ with slack $\Delta = 64$, then trimmed in place with `del lst[:drop]` back to K . Slack amortizes the $O(K)$ cost of prefix deletion so trims are not performed every bar.

PineSeries wrappers retain a history floor of 1000 samples (raised when K is larger) so `close[n]` for $n \leq 999$ remains available even when list caps are tight—matching pre-cap host defaults.

4.2 Correctness argument for bounded history

Proposition 4.1 (Window kernels under cap). *Let f_p depend only on the last p finite samples of $\mathbf{x}$. If $p \leq K$ and the cap retains the newest K samples, then f_p at every bar with $i \geq p - 1$ equals the uncapped result.*

Proof sketch. At bar i , the uncapped window is $\{x_{i-p+1}, \dots, x_i\}$. The cap retains $\{x_{i-K+1}, \dots, x_i\}$ (or the full prefix if shorter). Since $p \leq K$, the window is a suffix of the retained segment. $\square$

Proposition 4.2 (Recursive kernels with incremental state). *Let $y_i = g(y_{i-1}, x_i)$ be a first-order recurrence (EMA, RMA, ...) with call-site state storing y_{i-1} . After warm-up, y_i is independent of raw samples older than 1, hence independent of K provided state is not reset.*

Remark 4.3 (Full-recompute residual). If `PYNE_TA_INCREMENTAL` = 0, recursive kernels reseed from the retained list prefix. When $n \gg K$, the re-seeded trajectory can diverge from an uncapped full recompute. Default-on incremental mode closes this residual for production hosts; the flag remains for debugging and oracle tests.

Proposition 4.4 (OOB is na). *For lookback $n \geq$ retained length, $x[n] = \text{na}$. Implementations must not coerce missing history to 0, which would bias sums, ranges, and boolean guards.*

4.3 When lookback exceeds the cap

If a script requests `close[500]` while $K = 256$, the host returns `na` unless `max_bars_back` (or `PYNE_SERIES_MAX`) raises K . This matches Pine’s resource model: deep history is an explicit request, not an implicit infinite tape. Scripts that need deep history should set `max_bars_back` accordingly; the cap then becomes a *floor* rather than a truncation surprise.

4.4 Structural memory win

Simulating 20 000 bars $\times$ 6 OHLCV-style lists with $K = 256$, $\Delta = 64$:

Mode	Final list len	~pointer bytes (6 lists)	Append wall
Uncapped	20 000	~ 960 000	0.020 s
Capped	305	~ 14 640	0.021 s

The memory ratio is $\sim 78\times$ at 20k bars and scales as n/K asymptotically. Trim CPU is negligible relative to AST and TA work.

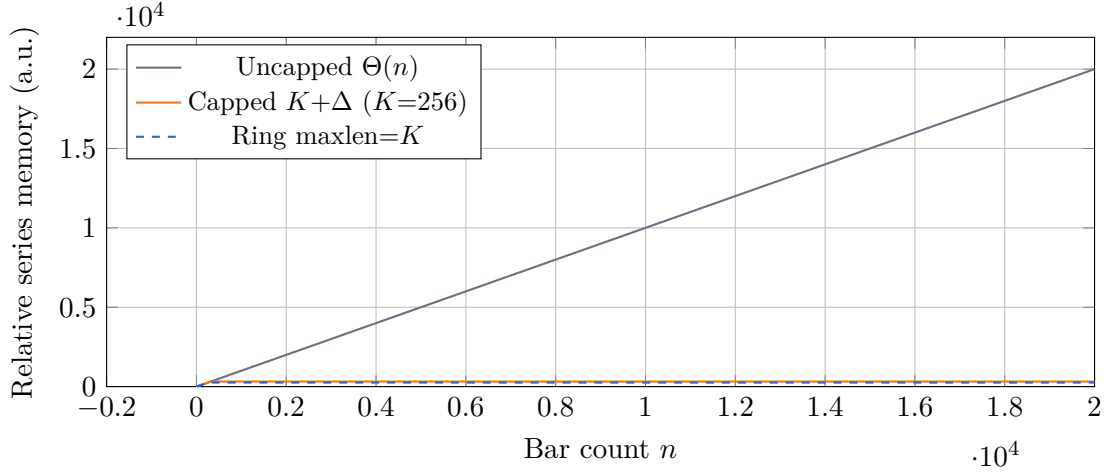


Figure 2: Series growth versus capped lists and fixed-capacity ring buffers. Uncapped memory grows linearly with bar count; the default series cap plateaus near $K + \Delta$; a modular ring plateaus exactly at K .

5 Ring Buffer Series

5.1 ChronologicalSeriesBuffer

The Phase 2.2 design introduces a chronological (oldest-first) buffer with modular ring capacity [HOOX Project, 2026]:

$$\text{append}(v) \quad O(1), \quad (5)$$

$$\text{lookback}(n) \quad O(1) \text{ via physical index } \text{end} - n, \quad (6)$$

$$\text{series}[n] \quad \text{na on negative / OOB / missing.} \quad (7)$$

When $\text{maxlen} = N$ is set, `append` overwrites the oldest slot after the ring fills—true $O(1)$ memory and time without prefix `del`. When maxlen is `None`, the buffer grows as a plain list (still $O(1)$ end-index lookback).

Why not deque alone? CPython `collections.deque` offers $O(1)$ append/pop at both ends, but *middle* indexing is $O(n)$. Modular list indexing is $O(1)$ for arbitrary lookback depth within the window—important for scripts that probe `close[k]` for moderate k on every bar.

5.2 Dual-write transition

Production hosts historically maintained *two* series representations:

- **PineSeries**: newest-first deque (`appendleft`), used by context OHLCV and `series[n]`.
- **current_series** lists: chronological, used by `ta` via `_as_series`.

Materializing a newest-first deque into chronological order required `list(reversed(...))`, a residual tax on full-recompute paths.

The migration plan is staged:

1. **Phase A (shipped)**: factory `make_pine_series()` returns `RingPineSeries` when `PYNE_SERIES_RING = 1`, else classic `PineSeries`. Default is `off` until the corpus is green under ring mode.
2. **Phase B**: zero-copy `_as_series` for objects marked `chrono_order=True`.

Algorithm 1 Generic bar-mode incremental update

Require: call-site map S , key k , input x_i , update U , init σ_0

1: **if** $k \notin S$ **then**

2: $S[k] \leftarrow \sigma_0$

3: **end if**

4: $(\sigma', y_i) \leftarrow U(S[k], x_i)$

5: $S[k] \leftarrow \sigma'$

6: **return** y_i

▷ may be na during warm-up

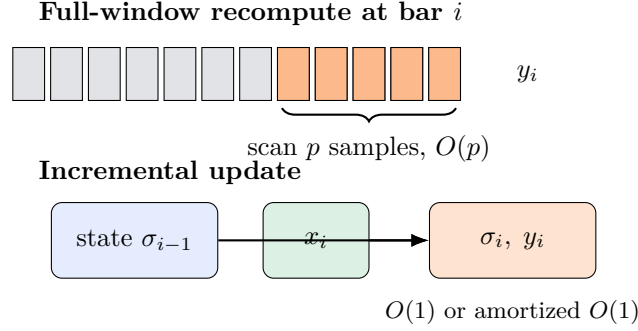


Figure 3: Incremental versus full-window TA. Full recompute re-scans the period window on every bar; incremental kernels carry a state tuple and apply a constant-time (or amortized constant-time) update.

3. **Phase C:** single buffer shared by context and `current_series`, dropping dual update.

`RingPineSeries` exposes a `NewestFirstHistoryView` so existing duck-typed code (`history[0] == current`) continues to work during the transition. `Ring maxlen` tracks `pineries_history_length(...)` so it composes with the series cap rather than fighting it.

6 Incremental TA Framework

6.1 Generic pattern

Definition 6.1 (Incremental kernel). An incremental kernel is a pair (σ_0, U) where σ_0 is initial state and $U(\sigma, x_i) = (\sigma', y_i)$ produces updated state and output. State is stored in a call-site map keyed by (builtin name, syntactic slot, parameters).

Flag `PYNE_TA_INCREMENTAL` (default **on**) selects incremental updates; setting it to 0 forces full-window oracles for differential testing. The compile path always emits incremental Numba kernels for hot TA regardless of this flag [Lam et al., 2015, HOOX Project, 2026].

Figure 3 contrasts full-window scans with incremental updates.

6.2 Case study: SMA (sliding sum)

Let $p \geq 1$. Maintain a deque window W of the last p samples and a running sum Σ :

$$\Sigma_i = \Sigma_{i-1} + x_i - x_{i-p} \quad (\text{when } |W| = p), \quad (8)$$

$$y_i = \Sigma_i / p. \quad (9)$$

Warm-up returns **na** until p finite samples accumulate. Samples that are **na** are typically excluded from the count or propagate **na** per Pine rules.

Complexity: $O(1)$ arithmetic and deque operations per bar after warm-up.

Algorithm 2 Incremental SMA with running sum

Require: state ($window, \Sigma, count$), input x , period p

```
1: if  $x$  is na then
2:   return (state, na) ▷ policy: skip or propagate
3: end if
4: append  $x$  to  $window$ ;  $\Sigma \leftarrow \Sigma + x$ ;  $count++$ 
5: if  $count > p$  then
6:    $u \leftarrow \text{popleft}(window)$ ;  $\Sigma \leftarrow \Sigma - u$ ;  $count--$ 
7: end if
8: if  $count < p$  then
9:   return (state, na)
10: end if
11: return (state,  $\Sigma/p$ )
```

6.3 Case study: EMA and RMA (Wilder)

The exponential moving average with smoothing $\alpha \in (0, 1]$ is

$$y_i = \alpha x_i + (1 - \alpha) y_{i-1}, \quad (10)$$

seeded by SMA of the first p samples for Pine-compatible EMA with $\alpha = 2/(p + 1)$. Wilder's RMA [Wilder, 1978] uses $\alpha = 1/p$ (equivalently $y_i = y_{i-1} + (x_i - y_{i-1})/p$).

State is a single scalar y_{i-1} plus a warm-up counter— $O(1)$ memory and time. Numerical stability: (10) is a contractive affine map; error is not amplified [Higham, 2002]. Catastrophic cancellation is not an issue for the recurrence itself, though very large $|x_i|$ can still lose precision in IEEE-754 binary64 [IEEE Computer Society, 2019].

6.4 Case study: RSI

Relative Strength Index [Wilder, 1978] maintains average gain G and average loss L via RMA:

$$u_i = \max(x_i - x_{i-1}, 0), \quad d_i = \max(x_{i-1} - x_i, 0), \quad (11)$$

$$G_i = \text{RMA}_p(u)_i, \quad L_i = \text{RMA}_p(d)_i, \quad (12)$$

$$\text{RSI}_i = G_i / L_i, \quad \text{RSI}_i = 100 - 100 / (1 + \text{RSI}_i). \quad (13)$$

When $L_i = 0$, RSI is defined as 100 if $G_i > 0$, else typically 0 or na per implementation policy. Incremental form stores $(G_{i-1}, L_{i-1}, x_{i-1})$.

6.5 Case study: Bollinger Bands (mean + variance)

Bollinger Bands [Bollinger, 2001] are

$$\text{mid}_i = \text{SMA}_p(x)_i, \quad \sigma_i = \text{stdev}_p(x)_i, \quad \text{upper/lower}_i = \text{mid}_i \pm k\sigma_i. \quad (14)$$

PYNE maintains sliding sum Σ and sum of squares Q :

$$\Sigma_i = \Sigma_{i-1} + x_i - x_{i-p}, \quad (15)$$

$$Q_i = Q_{i-1} + x_i^2 - x_{i-p}^2, \quad (16)$$

$$\sigma_i^2 = Q_i/p - (\Sigma_i/p)^2 \quad (\text{population form; sample form uses } p-1). \quad (17)$$

Alternatively, Welford's online algorithm updates mean and M_2 with better numerical properties for one-pass streams [Higham, 2002, Press et al., 2007]; for fixed-length sliding windows, a

Algorithm 3 Sliding window maximum (highest)

Require: deque D of indices (decreasing values), input x_i , period p

```
1: while  $D$  nonempty and  $x_{D.back} \leq x_i$  do  
2:   pop back  $D$   
3: end while  
4: push  $i$  to back of  $D$   
5: while  $D.front \leq i - p$  do  
6:   pop front  $D$   
7: end while  
8: return  $x_{D.front}$ 
```

▷ na if window incomplete

compensated sliding variance or periodic full refresh reduces cancellation when Q and $(\Sigma/p)^2$ are large and close.

Round 7 ships a dedicated `_bb_inc_update` that nests SMA+stdev slots and last-sample hygiene so Volatility paths avoid reverse materialization. Kernel micro-benchmarks report $\sim 217\times$ versus naive full `_sma+_stdev` rebuilds at $n = 2000$ (Section 9).

6.6 Case study: highest / lowest (monotonic dequeues)

Range extrema over a sliding window admit classic amortized $O(1)$ updates via monotonic dequeues [Aho et al., 2006]:

Each index is pushed and popped at most once, so amortized cost is $O(1)$ per bar. Lowest is symmetric (increasing deque).

6.7 Case study: KAMA

Kaufman’s adaptive moving average [Kaufman, 2013] computes an efficiency ratio from change over noise:

$$ER_i = \frac{|x_i - x_{i-L}|}{\sum_{j=0}^{L-1} |x_{i-j} - x_{i-j-1}|}, \quad (18)$$

$$SC_i = (ER_i \cdot (\text{fast} - \text{slow}) + \text{slow})^2, \quad (19)$$

$$y_i = y_{i-1} + SC_i \cdot (x_i - y_{i-1}). \quad (20)$$

A naive implementation rebuilds the noise sum from scratch each bar ($O(L)$ or worse if re-seeded from bar 0). The incremental kernel keeps a price ring of length $L+1$ and a running absolute-delta sum, achieving $O(1)$ per bar after seeding. Micro-benchmarks: $\sim 121\times$ at $n = 2000$, length 10.

6.8 Case study: StochRSI

Stochastic RSI applies a stochastic oscillator to an RSI series. Nested full rebuilds yield near-quadratic cost. The incremental form maintains:

- an RSI window (or Wilder RSI state, depending on oracle),
- a stochastic ring over the last `stoch_len` RSI values,
- optional signal smoothing in call-site state ($0.33x + 0.67\text{prev}$ in PYNE’s simple-path oracle).

Round 7 reports $\sim 717\times$ kernel speedup for `stochrsi(14,14)` at $n = 2000$. Note: the interpret oracle for this path uses simple average gain/loss RSI rather than Wilder `ta.rsi`; the incremental path matches that oracle bit-for-bit, not necessarily live TradingView® StochRSI-on-RMA [HOOX Project, 2026].

6.9 Additional kernels

Across rounds 1–7 the incremental surface grew to include (non-exhaustively): WMA, VWMA, TR/ATR, change, stoch %K, cum, VWAP, MOM, SWMA, barsince, highestbars/lowestbars, linreg, CCI, TSI, ROC, WPR, dev, HMA, rising/falling, median, percentrank, CMO, and compile-side ports of the same recurrences. Typical gains range from $\sim 1.2\times$ (already $O(p)$ windows with better hygiene) to two–three orders of magnitude for former full-prefix rebuilds.

6.10 Numerical stability notes

- **Sliding variance:** prefer compensated updates or periodic resync when $|x|$ is large [Higham, 2002].
- **EMA/RMA:** contractive; warm-up seed choice dominates early bars.
- **Ratios (RSI, ER):** guard zero denominators explicitly; do not rely on IEEE infinities for Pine na semantics.
- **Parity budget:** compile vs. interpret max abs error $\sim 10^{-11}$ on standard kernels (float noise), 0 on pure integer-path RSI/TSI variants in Numba goldens.

7 Interpreter Hot Paths

7.1 Dispatch

Expression evaluation is visitor-based [Gamma et al., 1994, Nystrom, 2021]. Hot improvements include type-keyed dispatch tables (avoiding long `isinstance` chains), operator maps for binary ops, and a scalar fast path that skips series coercion when both operands are plain floats. Mixed expression micro-benchmarks improved $\sim 2\text{--}7\times$ across rounds.

The cumulative cProfile view on multi-plot TA scripts still shows `visit_Call` as a large envelope: thousands of calls per bar for nested builtins, each paying Python function and argument-binding overhead. This is structural to AST interpretation; the compile path eliminates it.

7.2 Plot packing

For `ta_combo`-class scripts (multiple TA values, ~ 8 plots), variant isolation shows:

Variant (interpret, $n = 2\,000$)		med wall
<code>ta_combo</code> with <code>8×plot(...)</code>	~ 480 ms (profile host)	
Same TA, no plots		~ 102 ms

Thus plot path + host result packing account for $\sim 75\text{--}80\%$ of multi-plot wall time once TA is incremental. cProfile ranks `_builtin_plot` / `_capture_plot` immediately after the visitor envelope.

7.3 Light plots and lazy calendar

- **Lazy calendar** (always on): defer expensive calendar/timezone materialization until a plot or time builtin requires it.
- **PYNE_LIGHT_PLOTS:** skip columnar capture and registry bookkeeping when the consumer only needs OK/fail (corpus runs, smoke tests). Multi-plot scripts see roughly 10–20% wall reduction.

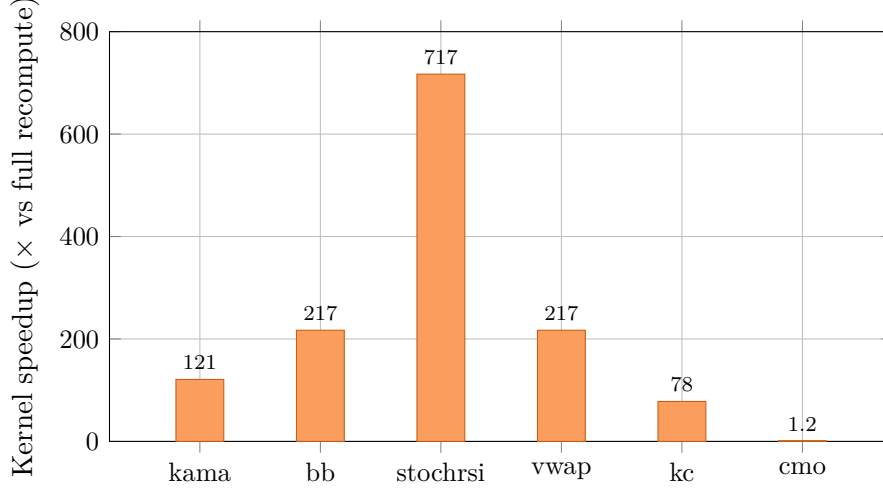


Figure 4: Kernel micro-benchmark speedups (incremental vs. full-window rebuild) at $n \approx 2000$ bars. Values from performance rounds 4–7: KAMA $\sim 121\times$, Bollinger $\sim 217\times$, StochRSI $\sim 717\times$, VWAP $\sim 217\times$, KC $\sim 78\times$, CMO $\sim 1.2\times$ (already near- $O(p)$).

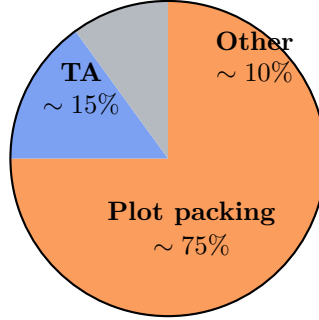


Figure 5: Approximate latency breakdown for interpret `ta_combo` after incremental TA: plot packing dominates residual wall time; pure TA kernels are a minority slice; “Other” covers dispatch, series updates, and host bookkeeping.

These are host-level levers: they do not change TA math, but they reclaim the dominant residual identified by profiling.

8 Parse/AST Caching Interaction with Runtime Multi-Run

Product hosts often parse once and evaluate many times (parameter sweeps, watchlists, recharts). PYNE maintains a process-level LRU cache keyed by SHA-256 of source text. Multi-parse of identical source improves by $\sim 1000\times$ on cache hit.

Call-site scrub on hit. AST nodes may accumulate mutable annotations (resolved call sites, cached locations). Returning the same `Script` object on a cache hit without scrubbing caused empty-plot poisoning on the second `Runtime.run` of identical source: call-site maps and plot registries interacted with stale node state. The fix clears Pine call-site annotations on cache hit (`_scrub_pine_call_sites`) and pairs multi-run goldens with `clear_parse_cache()` isolation where tests construct independent Runtime instances [HOOX Project, 2026].

Interaction with series/TA state. Parse caching is orthogonal to series caps and TA state: those live in the evaluator/runtime instance. However, multi-run correctness requires that

Table 1: Series list memory at $n = 20\,000$ bars, $s = 6$ OHLCV-style lists, $K = 256$, slack $\Delta = 64$.

Configuration	Len/list	$\sim$ pointers (total)	Ratio vs uncapped
Uncapped lists	20 000	$\sim 960\text{k}$	$1\times$
PYNE_SERIES_CAP (default)	≤ 320	$\sim 14.6\text{k}$	$\sim \mathbf{78}\times$
Ring maxlen = K	256	$\sim 12.3\text{k}$	$\sim 78\times$

Table 2: Kernel micro-benchmarks (bar-walk growing prefix, $n = 2\,000$, median).

Kernel	Full recompute	Incremental	Speedup
kama(10)	1099 ms	9.1 ms	$\sim \mathbf{121}\times$
bb(20) (pure full vs inc)	2458 ms	11.3 ms	$\sim \mathbf{217}\times$
stochrsi(14,14)	10 196 ms	14.2 ms	$\sim \mathbf{717}\times$
cmo(14)	13.5 ms	11.4 ms	$\sim 1.2\times$
vwap (earlier round)	1669 ms	7.7 ms	$\sim 217\times$
kc(20,2) (earlier round)	2054 ms	26 ms	$\sim 78\times$

(1) AST mutation not leak across runs, and (2) evaluator call-site maps reset or key correctly when scripts are re-bound. Warm product paths prefer the compile cache (near-zero warm compile, sub-millisecond runs for pure TA) when scripts are compilation-eligible.

9 Evaluation

9.1 Methodology

Unless noted, numbers are from PYNE performance rounds 4–7 on Linux x86_64 CPython, using `scripts/bench_pipeline.py` and kernel micro-benches (median of small iteration counts after warmup). Absolute milliseconds vary by host; we report the Round 7 synthesis table for end-to-end interpret latency and kernel ratios for structural wins. Verify suite at synthesis: 627 passed (series cap/ring, parse cache, lazy plots, TA incremental, order fills, evaluator, compiler residual kernels, corpus residuals).

9.2 Memory

Table 1 summarizes series list memory. Figure 2 plots the asymptotic shape.

9.3 Kernel speedups

Table 2 and Figure 4 report incremental vs. full-recompute micro-benchmarks.

9.4 End-to-end interpret latency

Round 7 net benchmarks at $n = 2\,000$ bars (synthesis medians):

Kernel-level T2 wins (KAMA/BB/StochRSI) do not appear in `ta_combo` because that script exercises `sma/ema/rsi/atr/stddev/bb/highest/lowest`—already incremental before Round 7. Wall-time flatness of the synthesis suite is expected; structural wins show up in kernel micros and memory, while plot packing dominates multi-plot residual (Figure 5).

Table 3: Interpret Runtime median latency @ 2000 bars (Round 7 synthesis).

Script	R6 med (ms)	R7 med (ms)	Notes
minimal	14.92	15.21	host + one plot floor
ta_sma	22.53	23.62	single SMA path
ta_combo	137.23	152.85	plot-dominated residual
strategy_ish	63.07	68.90	broker + plots

9.5 Compile path comparison

Warm compile of `ta_combo` is ~ 0.01 ms with IR/disk cache; run median ~ 1.06 ms at 5000 bars in nopython mode—orders of magnitude below interpret. Cold JIT remains highly variable (hundreds of ms to seconds) depending on Numba cache state [Lam et al., 2015]. Product guidance: prefer warm compile for multi-run TA; use interpret for compilation-ineligible scripts and fidelity debugging.

9.6 Feature flags summary

Flag	Default	Role
PYNE_TA_INCREMENTAL	ON	interpret incremental kernels
PYNE_SERIES_CAP	ON	trim <code>current_series</code> lists
PYNE_SERIES_MAX	unset ($K_0=256$)	absolute cap override
PYNE_SERIES_RING	OFF	chronological ring wrappers
PYNE_LIGHT_PLOTS	OFF	skip columnar plot capture

10 Related Work

Incremental computation. Classical incremental computation maintains derived results under input changes [Acar, 2008, Ramalingam and Reps, 1993, Reps et al., 1983]. Our setting is the special case of *append-only time*: each bar extends every series by one sample, so self-adjusting computation and dynamic dependency graphs are unnecessary—fixed recurrences and sliding windows suffice. Streaming algorithms for sliding aggregates (sums, extrema, quantiles) are textbook [Box and Jenkins, 1970, Knuth, 1997].

Technical analysis libraries. Wilder’s original systems [Wilder, 1978], Bollinger Bands [Bollinger, 2001], and Kaufman’s adaptive methods [Kaufman, 2013] define the mathematical objects. Vectorized libraries (pandas, TA-Lib, NumPy-first stacks) compute indicators over whole arrays [Harris et al., 2020, McKinney, 2017], which matches PYNE’s compile path but not the bar-loop interpret path where scripts interleave TA with control flow, strategy orders, and mutable `var` state.

Memory-bounded interpreters and ring buffers. Ring buffers are standard in signal processing and market-data feeds. Our contribution is integrating them with Pine’s $x[n]$ lookback semantics, `na` discipline, dual newest-first/chrono representations, and a flag-gated migration that preserves corpus goldens.

Compiler/JIT for DSLs. Numba [Lam et al., 2015] and related JITs provide the compile backend. Partial evaluation and online compilation literature [Jones et al., 1993] motivates caching parse/AST/IR across multi-run product workloads.

Numerical accuracy. Higham’s treatment of floating-point stability [Higham, 2002] and IEEE-754 [IEEE Computer Society, 2019] underwrite our parity budgets and sliding-variance cautions.

11 Limitations

1. **Ring buffer default-off.** `PYNE_SERIES_RING` remains opt-in until dual-write is removed and the full corpus is green under ring mode.
2. **Plot-path residual.** Incremental TA shifts the bottleneck to plot packing; structural wins there are only partially landed (`PYNE_LIGHT_PLOTS`, registry upsert hygiene).
3. **Nested full-history leftovers.** Some builtins (e.g., certain `obv/mode/range` paths) still recompute more than necessary.
4. **Oracle quirks.** StochRSI incremental matches the simple-RSI full path, not necessarily Wilder-based live platform StochRSI. ATR Wilder re-baseline is intentionally gated on dedicated goldens.
5. **Full recompute + cap.** With incremental TA disabled, recursive kernels can diverge from uncapped trajectories when $n \gg K$.
6. **Host absolute variance.** End-to-end ms figures vary by machine load and CPython version; treat ratios and asymptotic claims as primary.
7. **No whole-script auto-vectorization on interpret.** Control flow and strategy side effects keep the bar loop; users wanting array throughput should use the compile path.

12 Conclusion

Bar-loop series languages induce characteristic memory and compute pathologies: unbounded history and full-window TA rebuilds. PYNE demonstrates that a modest set of systems techniques—bounded series caps, chronological rings, call-site incremental kernels, parse caching, and light plot modes—restores linear-time, memory-bounded evaluation for the hot surface of technical analysis while preserving Pine’s lookback and `na` semantics.

Empirically, series caps deliver $\sim 78\times$ long-run list-memory reduction; incremental kernels reach two–three orders of magnitude on former quadratic rebuilds (KAMA, BB, StochRSI); and residual multi-plot interpret time is dominated by plot packing rather than arithmetic. The compile path, always incremental for hot TA, remains the throughput vehicle for multi-run product workloads.

Future work includes default-on rings after corpus ratification, deeper plot pipeline restructuring, remaining nested-kernel residuals, and tighter dual-host parity for worker deployments.

Acknowledgments

We thank the PYNE open-source contributors and performance-agent rounds for measurement infrastructure, goldens, and iterative kernel landings.

A Memory bound (informal)

Proposition A.1 (Bounded series store). *Under `PYNE_SERIES_CAP` with capacity K and slack Δ , and with s chronological lists, the number of retained samples satisfies*

$$\forall t, \quad \sum_{j=1}^s |L_j(t)| \leq s(K + \Delta). \quad (21)$$

With a modular ring of maxlen K per series, the bound tightens to $s \cdot K$ and each lookback of depth $< K$ is $O(1)$.

Proposition A.2 (Incremental TA time). *If every TA call site k admits an update of cost $u_k \in O(1) \cup O(p_k)$ amortized, then total TA work over n bars is $O(n \sum_k u_k)$, independent of full prefix length.*

B Environment reproduction notes

Listing 1: Flag surface (interpret path)

```
1 # Default-on incremental TA and series cap
2 # PYNE_TA_INCREMENTAL=1
3 # PYNE_SERIES_CAP=1
4 # Optional:
5 # PYNE_SERIES_MAX=512
6 # PYNE_SERIES_RING=1
7 # PYNE_LIGHT_PLOTS=1
```

Listing 2: Minimal Pine series / TA example

```
1 //@version=5
2 indicator("Demo", overlay=true)
3 len = input.int(14, "Length")
4 v = ta.sma(close, len)
5 plot(v)
```

References

- Umut A. Acar. Incremental computation. In *Encyclopedia of Algorithms*. Springer, 2008.
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, Boston, 2nd edition, 2006.
- John Bollinger. Bollinger on bollinger bands. *McGraw-Hill*, 2001.
- George E. P. Box and Gwilym M. Jenkins. Time series analysis: Forecasting and control. *Holden-Day*, 1970.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1994.
- Charles R. Harris et al. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi: 10.1038/s41586-020-2649-2.
- Nicholas J. Higham. Accuracy and stability of numerical algorithms. *SIAM*, 2002.
- HOOX Project. PYNE: Independent open toolchain for the Pine Script language. <https://github.com/hoox-sh/pyne>, 2026. Version 0.3.0; PyPI package `hoox-pyne`.

- IEEE Computer Society. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019*, 2019. doi: 10.1109/IEEESTD.2019.8766229.
- Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial evaluation and automatic program generation. Prentice Hall, 1993.
- Perry J. Kaufman. Trading systems and methods. *Wiley*, 2013.
- Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997.
- Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6. ACM, 2015. doi: 10.1145/2833157.2833162.
- Wes McKinney. Python for data analysis. O’Reilly, 2017.
- Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021.
- William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007.
- G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. *Proceedings of POPL*, 1993.
- Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL*, 1983.
- TradingView, Inc. Pine Script language reference manual. <https://www.tradingview.com/pine-script-docs/>, 2024. Accessed 2026.
- Ruey S. Tsay. *Analysis of Financial Time Series*. Wiley, 3rd edition, 2010.
- J. Welles Wilder. New concepts in technical trading systems. *Trend Research*, 1978.