Scheduling FFT-Based Spectral Analysis in the Audio Processing Loop

Christian Kauten
Arhythmetic Units
arhythmetic.units@gmail.com

October 1, 2026 / Release v2.3.0, manuscript version 5

Abstract

An FFT-based spectrum analyzer can be inexpensive on average yet concentrate most of its work in one audio-processing call. This paper presents CoopFFT (Cooperative Fast Fourier Transform), which spreads that work across successive sample calls without a separate analysis thread. A resumable radix-2 FFT preserves its traversal state between calls; windowing, spectrum reconstruction, smoothing, and output preparation are scheduled alongside it. Each frame completes over one chosen update interval, with a bounded operation count per call and an explicit delay before the spectrum becomes available. We implement the method in two VCV Rack visualizers and evaluate it using matched batch controls and optimized native FFT libraries. In a scalar 4096-point comparison, distributing the same implementation reduces the typical peak block time from 52.2 to 6.67 μs while increasing aggregate instrumented cost by 18%. With four aligned analyzers on one Rack engine thread, CoopFFT reduces the corresponding peak from 890 μs for native vDSP batch processing to 201 μs, at the cost of more work and about 30 ms of additional publication delay. The measurements demonstrate a useful tradeoff between processing bursts, total cost, and result latency; they do not show a consistent reduction in missed deadlines.

Keywords: CoopFFT; spectral analysis; cooperative scheduling; real-time audio; VCV Rack.

1 Introduction

A spectrum analyzer need not generate audio to interfere with audio processing. If its FFT runs on the audio engine thread, the engine must finish that work before it can deliver the next block of sound. Computing a spectrum only a few dozen times per second can make the average cost look modest while leaving occasional blocks much more expensive than their neighbors. Several analyzers updating together can compound the burst.

This problem motivated CoopFFT (Cooperative Fast Fourier Transform), the analysis approach implemented in Fourier, a spectrum analyzer for the VCV Rack modular synthesizer. Rack invokes a module's processing function once per sample [1]. The FFT was therefore built to advance a little on each invocation: execute some butterflies, save the current position, and resume when the next sample arrives. Analysis remains in the ordinary processing loop. The display receives the completed spectrum later, while audio processing continues between pieces of the transform. CoopFFT names the resumable FFT and its surrounding analysis pipeline.

The distinction is between the amount of work and when that work happens. A conventional radix-2 FFT takes $O(N \log N)$ operations for N samples. Making it resumable does not reduce that count. It makes the computation available for scheduling over an interval in which the application can afford to wait for the result. Spectral visualization is a natural application: its output is a sequence of display updates rather than an audio sample that must be returned immediately. Whether the added delay is acceptable still depends on the desired display response.

Scheduling butterflies alone is insufficient. Windowing and arranging the input, reconstructing a real spectrum, computing magnitudes, and preparing output can each leave a large loop at a frame boundary. The implementation studied here distributes those operations too. It selects a frame, preserves its input while new samples arrive, advances the analysis in dependency order, and publishes the completed result at the end of a fixed number of calls. Figure 1 illustrates this change in execution.

There is a competing advantage to doing the work at once: optimized FFT libraries use vectorization and carefully organized memory access. A custom resumable transform may give up some of that efficiency. A useful evaluation must therefore ask both whether distribution reduces bursts within the same implementation and whether the resulting tradeoff remains worthwhile against native libraries. Comparing only a custom batch FFT with its incremental version would answer the first question but leave the second open.

This paper makes three contributions. It presents an implementable schedule for complete windowed analysis within a sample-processing loop; explains its work bound, input lifetime, and publication delay; and measures the tradeoff in controlled analysis and complete Rack modules. The method uses established FFT mathematics and cooperative scheduling. Its contribution is their application to the complete analyzer

and the evidence needed to judge that choice, rather than a new Fourier transform or a claim that distribution is always faster.

The results support a focused conclusion: distributing analysis can buy substantial processing headroom in burst-heavy configurations, but that headroom has costs in total work and result age. Native batch and hybrid implementations remain useful alternatives. Host scheduling also matters: lower typical processing peaks did not consistently produce fewer late blocks in the measured system.

2 Background and Related Work

The FFT reduces the arithmetic required to evaluate the discrete Fourier transform [2]; real-input algorithms exploit conjugate symmetry to avoid redundant work [3]. Repeating a windowed transform at a fixed hop produces a short-time spectral representation [4]. Window choice controls the familiar tradeoff between frequency resolution and spectral leakage [5]. The present work preserves these analysis choices and changes the execution schedule.

Time distribution has direct precedents. Lo and Lee describe FFT methods for online spectral analysis [6]. Hurchalla applies a time-distributed FFT to low-latency convolution [7]. Battenberg and Avižienis compare cooperative and preemptive implementations of partitioned convolution on conventional operating systems [8]. Their cooperative design distributes work within the host's processing thread and uses FFTW for small transform leaves. These studies establish both the idea of spreading FFT work and the importance of choosing an execution granularity. The complete-analysis schedule below is a specialization for spectrum and spectrogram displays, not an invention of time-distributed computation.

Other methods change the computation across successive frames. The sliding DFT updates a spectrum as samples enter and leave a moving window [9]. Such reuse is attractive for frequent updates, but it brings different numerical and windowing considerations. Here each selected frame is transformed afresh. The method supports the analyzer's existing windows and arbitrary integer hops without maintaining a recursive spectral estimate. Sliding transforms are an alternative design, not a measured baseline in this study.

Finally, execution efficiency is distinct from scheduling flexibility. FFTW selects transform decompositions for the target machine [10]; PFFFT and Apple's vDSP provide native optimized implementations [11, 12]. We compare with all three and include hybrids that distribute the surrounding analysis while keeping a native FFT call intact. This exposes the practical question: how much of the benefit requires interrupting the transform itself, and how much can be obtained by scheduling the work around it?

3 CoopFFT: Scheduling the Analysis

3.1 Frames and processing calls

Let $x[n]$ be a real signal sampled at f_s Hz. The analyzer uses a power-of-two transform length N and an integer hop $H \geq 1$, the number of samples between successive frame endpoints.

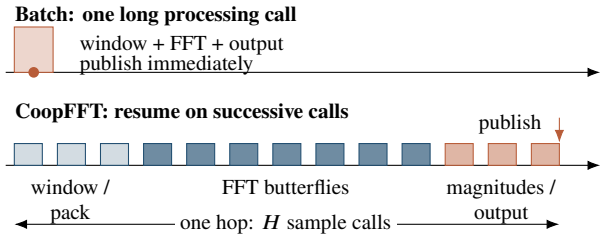

Figure 1. The same selected frame can be computed at its endpoint or spread over the following hop. Input capture continues in both cases. Bars are conceptual work slices, not timing measurements; stages and bar heights are not to scale. CoopFFT publishes the result on the last call.

Frame r ends at $t_r = rH$ and contains

$$u_r[m] = x[t_r - N + 1 + m]w[m], \quad 0 \leq m < N, \tag{1}$$

where $w[m]$ is the selected window, including any fixed gain normalization. Its spectrum is

$$X_r[k] = \sum_{m=0}^{N-1} u_r[m]e^{-j2\pi km/N}. \tag{2}$$

Deferring computation changes when this result is available, not the samples in the sum. In particular, later input must not overwrite a sample before the scheduled windowing operation has read it.

A *sample call* advances the engine by one sample. A *block* contains D such calls and is the unit delivered by the audio host. The hop and block size need not match or divide one another. At 48 kHz, a block of 64 samples has a time budget of 1.333 ms; a hop of 1024 samples gives 21.33 ms between spectra. These are different clocks: distributing a frame over its hop can move work out of one block and into subsequent blocks.

3.2 A resumable FFT

An iterative radix-2 FFT traverses stages, groups within a stage, and pairs within a group. A butterfly reads two coefficients, multiplies one by a precomputed twiddle factor, and replaces the pair by their sum and difference. Saving the current stage, group, and pair is enough to resume the traversal. The coefficient array remains in place between calls. Pausing introduces no new transform mathematics: when resumed, the next butterfly sees the same operands that an uninterrupted traversal would have produced.

For real input, pair the windowed samples as $z[m] = u_r[2m] + ju_r[2m+1]$ and compute an $M = N/2$ point complex FFT. Standard real-spectrum reconstruction then produces the $K = M + 1$ bins from DC through Nyquist [3]. The complex traversal contains

$$B = \frac{M}{2}\log_2 M = \frac{N}{4}\log_2(N/2) \tag{3}$$

butterflies. The traversal indices require constant storage; the coefficient and lookup arrays require $O(N)$ storage. Any

Stage	One operation	Credits
Prepare	Window, pack, and permute an input pair	pM
Transform	Execute one complex butterfly	B
Magnitude	Reconstruct one bin; compute magnitude and optional prefix sum	K
Output	Smooth one bin and write its display data	oK

Table 1. CoopFFT stages for real input. $M = N/2$ pairs and $K = M + 1$ bins are processed in order. The last column gives the total credits per stage; p and o are preparation and output weights.

partition of this ordered traversal preserves the result within the same arithmetic implementation. This does not imply bitwise agreement with a different FFT library.

3.3 Include the work around the FFT

The analyzer divides each frame into four stages (Table 1). First, it windows pairs of retained input samples and stores them in the FFT's permuted order. Next, it executes the butterflies. It then reconstructs positive-frequency bins and computes their magnitudes. Finally, it applies enabled smoothing and emits display data one bin at a time. Each stage finishes before the next begins; a sample call can cross a stage boundary if it has work left to do.

Frequency smoothing uses prefix sums of the magnitudes. Once those sums are complete, the mean over any contiguous bin range needs two prefix reads and a subtraction rather than a loop over that range. Temporal smoothing updates one stored value per bin. Thus neither form of smoothing adds an unscheduled whole-spectrum pass on the publication call. The display callback must also perform bounded work per bin; an arbitrary expensive callback would defeat the schedule.

To account for operations with different costs, the implementation assigns integer *credits*. Most operations use one credit. Rebuilding window coefficients uses $p = 4$ credits per input pair, compared with $p = 1$ when the cache is ready. Output uses $o = 2$ credits per bin for Fourier's four-channel display conversion and $o = 1$ otherwise. A weighted operation executes at its first credit; the remaining credits defer later work. These are engineering weights, not measurements of equal execution time. The total is

$$W = pM + B + (1 + o)K. \tag{4}$$

The weights affect where work falls within the hop without changing the computed spectrum or the chosen publication time.

3.4 An exact completion schedule

Number a frame's sample calls $s = 1, \ldots, H$, starting with the call that captures its endpoint. Assign call s the quota

$$q_s = \left\lfloor \frac{sW}{H} \right\rfloor - \left\lfloor \frac{(s-1)W}{H} \right\rfloor. \tag{5}$$

The cumulative count after s calls is $\lfloor sW/H \rfloor$, so all W credits have been consumed after exactly H calls. Every call receives either $\lfloor W/H \rfloor$ or $\lceil W/H \rceil$ credits. An integer quotient and remainder implement the rule without dividing on every sample:

```
// Once per frame configuration:
base = W / H; remainder = W % H;
// At frame start: error = 0.
// On each sample call:
quota = base;
error += remainder;
if (error >= H) { error -= H; ++quota; }
advance_analysis(quota);
if (++phase == H) { publish(); phase = 0; }
```

Here `advance_analysis` consumes credits in the stage order above, preserving the current stage and its cursor. Input capture occurs on every call, including calls whose quota is zero. Publication waits for the fixed boundary even if the last weighted operation has already executed.

For a clean scalar frame with $N = 4096$ and $H = 1024$, $W = 17410$: 1022 calls consume 17 credits and two consume 18. For fixed configuration, any 64 consecutive calls consume at most $\lceil 64W/H \rceil = 1089$ credits, including blocks that cross a frame boundary. More generally, when H is proportional to N, the maximum per-sample quota grows as $O(\log N)$ while a batch frame still requires $O(N \log N)$ work at once. The schedule bounds operation counts, not elapsed time: instruction costs, memory effects, and operating-system interruptions are outside that arithmetic statement.

Keeping the frame clock separate from arithmetic completion is essential. A fixed rounded quota such as $\lceil B/H \rceil$ butterflies per call can finish early. Restarting immediately would change the actual update rate and any smoothing calibrated to it. Equation (5) distributes the remainder across the intended interval and retains the requested cadence.

3.5 Input lifetime and result delay

The frame selected at t_r is published at $t_r + H - 1$. Measured from its newest sample, the delay is therefore

$$A_{\text{endpoint}} = (H - 1)/f_s. \tag{6}$$

Measured from the window center, add $(N-1)/(2f_s)$. For the 4096-point, 1024-hop example at 48 kHz, these ages are 21.31 and 63.97 ms, respectively. Even an immediate batch transform has the window-center age; distribution adds the time spent advancing the frame. These are logical sample-clock ages, excluding computation time, audio buffering, and repaint.

A preallocated input ring with capacity $N_{\max} + H_{\max}$ retains every selected sample until analysis finishes. At most $H - 1$ later samples arrive before publication, so the oldest sample of the N-sample frame cannot be overwritten. Windowing can therefore proceed incrementally without a bulk frame copy. The first frames use zero padding for input that predates startup. With fixed settings, only one frame is in progress and successive results are published exactly H samples apart.

4 Implementation in VCV Rack

CoopFFT is implemented by the header-only C++ class `SpectrumAnalysis<T>` and used by two modules in the Arhythmetic Units plugin. Fourier displays four spectra, using the four

lanes of Rack's SIMD sample type as independent channels. Spectre displays a scalar spectrogram. Neither module has an audio output, but both analyze input on the engine thread. The scheduling code depends on arithmetic operations on `T`, not on Rack or its graphics API.

Storage and maximum-size twiddle and permutation tables are prepared before sample processing. Smaller transforms reuse those tables by changing indices and strides. Window coefficients are cached and rebuilt incrementally after a relevant setting change. The scalar path retains the ordinary butterfly traversal; dense SIMD work is dispatched in contiguous segments within the same sample quota to reduce repeated setup. This preserves the scheduling bound without requiring a function dispatch for every individual butterfly.

At each frame boundary the modules latch analysis settings. Length changes invalidate input and averaging history logically; reset or sample-rate changes cancel partial analysis. Smoothing and display-coordinate caches rebuild in scheduled per-bin operations. Steady-state analysis and live window, length, and smoothing changes use prepared storage. Increasing the sample rate can require growing Fourier's input ring in the host's sample-rate callback; that lifecycle operation is outside the per-sample guarantee.

Publishing a spectrum must not reintroduce a large copy or expose unfinished data to the display. Each bin is written into producer-owned storage. At completion, a single-producer/single-consumer mailbox exchanges ownership using acquire/release atomics. The UI keeps its current slot while the engine fills another. Fourier publishes a completed curve; Spectre publishes completed history columns and constructs its image on the UI thread. A slow consumer can miss intermediate updates, but it does not observe a partially written spectrum. This ownership rule is separate from the arithmetic schedule.

The deployed defaults also make the latency choice concrete. Fourier uses $N = 2048$ and a 30 ms hop ($H = 1440$ at 48 kHz); Spectre uses $N = 2048$ and $H = 1024$. Both default to a flattop window. The schedule adds nearly one hop before publication, not one hop of audio output latency: these modules produce display data only. Display polling and rendering add further delay. The study below measures processing and logical publication age; it does not evaluate the perceptual acceptability of that display response.

5 Experimental Method

The evaluation asks two questions: does spreading a fixed implementation reduce processing bursts, and how does that tradeoff compare with native FFT-based analysis inside a real host engine? We use the retained study collected on September 30, 2026: two separately prepared sessions on one Apple M1 Pro laptop running macOS 26.6.2, with Apple Clang 21.0.0. The measured C++ uses the Rack build flags, including `-O3` and `-funsafe-math-optimizations`; dependency identities and complete compile commands are archived. Both sessions use the same executable and sources. Each planned configuration was run in two fresh processes per session, giving four processes per comparison row. The complete campaign contains 194 planned cells and 776 processes, including scaling, numerical stress, and lifecycle cases beyond the selections reported here.

5.1 Comparisons and workloads

The *CoopFFT batch* and *CoopFFT scheduled* controls use the same scalar implementation, storage, arithmetic, and frame endpoints. Batch spends all credits in the endpoint call; scheduled execution spends them according to Equation (5). This isolates placement within the implementation. Unless marked batch, CoopFFT denotes the scheduled mode.

The native controls use PFFFT, FFTW, or vDSP for the transform. Each provider has a batch path and a *hybrid*: the hybrid distributes input preparation and output processing but executes the native FFT and any required reordering as one indivisible call. Within each provider, the two paths share retained input, windowing, magnitude calculation, and smoothing. Native layouts are used directly where possible. vDSP uses vectorized preparation and batched four-channel transforms; FFTW uses reusable real plans, including batched plans for four channels, with single-threaded `FFTW_MEASURE` planning and no imported wisdom. Plan creation occurs before measurement.

Controlled comparisons use 48 kHz input, 64-sample blocks, a Hann window, and no smoothing unless stated otherwise. The main scalar comparison uses $N = 4096$, $H = 1024$. A four-channel comparison uses $N = 16384$, $H = 240$ and varies the hybrid's completion interval while keeping the hop fixed. These calls feed a controlled output sink. They measure complete analysis, but omit module input conditioning and display conversion.

Host comparisons run complete Fourier modules through Rack's `stepBlock()` entry point, including worker synchronization. Native controls replace the analysis inside the same module shell, retaining input conditioning, display conversion, and publication. The reported host rows use Fourier's defaults, one or four engine threads, and four or sixteen modules. Aligned modules share frame endpoints; staggered modules start at different phases. No background modules or display consumer are active in these rows. The host is paced at the nominal block period; controlled analysis runs continuously. Results from those different execution regimes are not treated as interchangeable timings.

5.2 Measurements and replication

We measure the duration of each whole processing block. To keep sparse FFT bursts visible, the primary statistic takes the longest block intersecting each complete frame-to-frame hop, then reports the median of those maxima. We call this the *typical hop peak*. A block crossing a hop boundary counts in both intervals. The same full-hop interval is used for immediate and delayed publication. Thus a batch method is not favored merely because it publishes sooner, and a rare frame boundary does not disappear beneath a callback percentile.

Aggregate cost is the sum of instrumented block wall times divided by engine samples, in ns/sample. It includes dispatch, timing overhead, and interruptions; it is not an independently

measured throughput or CPU utilization figure. A *compute-budget exceedance* occurs when a timed block takes longer than D/f_s. A *release miss* occurs when the paced host finishes after its scheduled deadline, including wake lateness and any backlog. Neither statistic is an audio-device underrun count.

Every process warms up for 16 nominal hops. Setup, correctness replay, and result serialization are outside the timed blocks. Within each session we average the two process costs and take the median of their hop-peak summaries; the published result weights the two sessions equally. Bracketed ranges show the two session summaries, not confidence intervals. We retain all processes and slow observations. With two sessions on one machine, treating individual blocks as independent replicates would overstate the evidence [13].

The operator prepared the machine offline, on AC power with Low Power Mode disabled, and closed unnecessary applications. Pre/post snapshots still show macOS service activity. No affinity, frequency lock, or real-time thread priority was applied. Timings therefore describe this host and build under the recorded conditions rather than worst-case execution time.

5.3 Numerical verification

Every measured path has a separate output replay checked against a direct DFT at small lengths and a separately structured double-precision FFT at larger lengths. Complete-module references also model input conditioning and display conversion. Relative vector ℓ_2 and ℓ_∞ errors must stay within 3×10^{-4} for float or 10^{-10} for double; a zero reference requires zero output. Long module decays have a separately flagged region affected by flush-to-zero and display mapping, checked against an explicit absolute floor and excluded from relative-accuracy claims.

The source archive, compact evidence, process summaries, and deterministic table generator are retained with the paper. Its build reads those results; it does not run a new benchmark. The original raw sessions remain local, so the repository is not presented as a public raw-data deposit.

6 Results

6.1 Distribution reduces bursts, with overhead

Table 2 isolates work placement and compares native alternatives. Scheduling scalar CoopFFT reduces the typical hop peak from 52.2 to 6.67 μs, a factor of 7.82. Aggregate instrumented cost rises from 68.3 to 80.3 ns/sample, an increase of about 18%. The same FFT arithmetic is being performed in smaller pieces; the result is less concentrated execution rather than less total work.

All three native hybrids also lower the peak relative to their own batch paths. PFFFT, for example, moves from 13.7 to 8.15 μs while cost rises from 41.3 to 49.8 ns/sample. Native hybrids can therefore recover much of the burst reduction with lower aggregate cost than CoopFFT. CoopFFT has the lowest typical peak at this scalar configuration, but it has the highest cost. There is no single winner on both measures.

Across the campaign's three baseline lengths, four implementation families, and two sessions, the distributed or hybrid member has the lower session hop-peak summary in all 24 comparisons. These are related observations on one machine, not 24 independent replications. Moreover, every typical peak in Table 2 is small relative to the 1,333 μs block budget. This experiment establishes the scheduling effect; complete-host measurements determine whether the saved headroom is substantial in a patch.

Analysis path	Cost	Peak [sessions]
CoopFFT batch	68.3	52.2 [51.2–53.1]
CoopFFT scheduled	80.3	**6.67** [6.25–7.08]
PFFFT batch	**41.3**	13.7 [13.1–14.3]
PFFFT hybrid	49.8	8.15 [7.4–8.9]
vDSP batch	43.6	17.1 [16.8–17.4]
vDSP hybrid	59.4	8.92 [8.85–8.98]
FFTW batch	**41.3**	21.8 [20.7–22.9]
FFTW hybrid	46.6	12.5 [11.6–13.4]

Table 2. Scalar analysis, $N = 4096$, $H = 1024$, $D = 64$, Hann, no smoothing. Cost is instrumented ns/sample; peak is the typical hop peak in μs. Brackets give the two session summaries. Each row contains four processes, 16,384 blocks, and 1,024 complete hops. No block exceeded the 1,333 μs budget. Lower is better; bold marks minima at displayed precision.

6.2 A shorter schedule delivers earlier, larger bursts

A native hybrid can finish within H, $H/2$, or $H/4$ sample calls, then only capture input until the next frame endpoint. Table 3 shows this tradeoff for four-channel vDSP analysis at a larger transform and shorter hop. Reducing the completion interval from 240 to 60 calls cuts endpoint-to-publication age from 4.98 to 1.23 ms. The typical peak rises from 68.6 to 121 μs, approaching the batch peak of 125 μs. The unchanged native transform remains indivisible; the surrounding work becomes more densely packed.

Finish after	Age (ms)	Cost	Peak
Batch	**0**	**544**	125
H	4.98	589	**68.6**
$H/2$	2.48	594	95.4
$H/4$	1.23	568	121

Table 3. Four-channel vDSP analysis, $N = 16384$, $H = 240$, $D = 64$, Hann. Age is endpoint-to-publication delay; cost is ns/sample for all four channels; peak is μs. Each row contains four processes and 3,840 blocks. Only the completion interval changes between hybrid rows. Lower is better; bold marks minima at displayed precision.

This configuration also shows a limit of CoopFFT. Its four-channel SIMD path costs 1080 ns/sample with a 93.6 μs typical peak. The full-hop vDSP hybrid costs 589 ns/sample with a 68.6 μs peak. Here the native hybrid is better on both recorded measures. The benefit of interrupting every butterfly therefore depends on transform size, hop, channel organization, and native kernel cost.

6.3 The effect survives one host configuration

With four aligned Fourier modules on one Rack engine thread, CoopFFT has a 201 μs typical hop peak, compared with 890 μs for native vDSP batch and 423 μs for the vDSP hybrid (Table 4). These occupy approximately 15%, 67%, and 32% of the block budget. Unlike the scalar comparison, the difference is a substantial fraction of the time available to the rest of the patch. The cost is higher aggregate processing: CoopFFT uses 2300 ns/sample, versus 1730 for batch and 2040 for hybrid. Both distributed paths also defer publication by nearly the 30 ms hop.

Analysis	Cost	Peak [sessions]	Misses
CoopFFT scheduled	2300	**201** [197–206]	**18**
vDSP batch	**1730**	890 [885–894]	20
vDSP hybrid	2040	423 [417–428]	38

Table 4. Actual Rack engine: one thread, four aligned four-channel Fourier modules, $N = 2048$, $H = 1440$, $D = 64$, flattop. Cost is block wall ns/sample for the complete engine; peak is μs. Brackets give session summaries. Misses are late synthetic releases across 16,384 blocks per row, including wake lateness, not audio underruns. Lower is better; bold marks minima at displayed precision.

With four engine threads and four aligned modules, the typical peaks converge to about 100–115 μs. Rack synchronizes its workers between samples [14], but the experiment cannot isolate that cost from core placement or power-state effects. The empty four-thread engine itself consumes about three CPU-seconds per second of nominal audio, indicating that its workers remain active between paced releases. The one-thread engine can sleep. The contrast therefore changes more than available parallelism.

Alignment also matters. With sixteen modules on four threads, CoopFFT's typical peak is 144 μs, compared with 261 μs for aligned vDSP batch. Staggering the batch modules lowers their peak to 171 μs. Phase distribution can reduce the coincidence of complete transforms without making them resumable, although an individual module cannot generally choose the phases of unrelated modules.

Lower typical peaks do not establish better deadline reliability. In the one-thread, four-module comparison, CoopFFT, vDSP batch, and hybrid miss 18, 20, and 38 synthetic releases respectively, across 16,384 blocks per row. With sixteen aligned modules on four threads, CoopFFT misses 19 releases and vDSP batch misses none. Empty-engine controls also have late releases. Wake delays, interruptions, and backlog remain relevant even when the normal processing burst is much smaller than the deadline.

6.4 Operating and numerical limits remain visible

Direct paced calls to a default Fourier module have a typical peak of 41.6 μs. At $N = 16384$ with a 5 ms hop, the four-port module reaches 997 μs; enabling both smoothing modes raises this to 1220 μs and produces 196 compute-budget exceedances in 3,840 blocks. Scheduling distributes the work but cannot make arbitrarily demanding settings affordable. No paced native complete-module control was collected at this extreme configuration.

All numerical replays pass their declared acceptance policies. The largest ordinary float relative vector errors are below 4.1×10^{-5}, against a 3×10^{-4} budget. These results include high overlap, non-power-of-two hops, silence, and serialized lifecycle changes. Relative error can reach one in the separately flagged long-decay tail; that region is covered only by its absolute floor. The tests substantiate the implemented analysis within those policies, not uniform relative accuracy for every weak spectral component.

7 Discussion and Limitations

The useful design choice is how much result delay to exchange for a smaller processing burst. For a visualizer whose native batch analysis comfortably fits the host budget, the added state and overhead of a resumable transform may be unnecessary. When several transforms coincide or one transform occupies a large share of the block, distribution can leave substantially more headroom. The one-thread Rack result demonstrates that regime, while the native hybrid results show that scheduling the surrounding stages may already be sufficient.

The FFT kernel and the scheduling policy should therefore be chosen together. An efficient native kernel reduces total work, but an indivisible call sets a lower limit on how finely that work can be spread. A resumable kernel gives more control over placement, but repeated indexing, dispatch, and smaller memory operations can cost more. The measured four-channel example shows that a sufficiently efficient native hybrid can beat CoopFFT even on peak time. This is why operation-count bounds and timing measurements answer different questions.

The delay also belongs in the design decision. A frame timestamp identifies the analyzed signal, not when the user sees it. In the retained headless consumer experiment at 60 Hz, default Fourier snapshots have median endpoint-age bounds of 38–40 ms for CoopFFT and the full-hop hybrid, versus 8–10 ms for native batch. That roughly 30 ms difference matches the schedule; it excludes repaint. Slower or stalled consumers skip some published frames. No listening, interaction, or visual-response study was performed, so acceptable display latency remains an application requirement.

The evidence has three main limits. First, both sessions were collected on one machine, binary, and date, with residual operating-system activity. Their ranges describe repeatability under those conditions, not portability or rare-event probabilities. Second, the engine was driven by a synthetic release schedule without a physical audio device or GUI rendering. Release misses cannot be interpreted as audible dropouts. Third, the evaluation compares a specific resumable radix-2 implementation with specific native adapters. It does not test a background analysis worker, sliding-spectrum reuse, or native sub-FFT leaves, and it does not establish an optimal granularity.

The implementation has corresponding boundaries. Credits are not equal-time units, and their weights were tuned during earlier engineering on the same machine. Lifecycle callbacks, UI work, other shared controls, and external interruptions

fall outside the per-sample accounting. Input retention and mailbox ownership solve particular data-lifetime problems; they do not make the entire plugin a hard real-time system. The result is a bounded schedule for the ordinary analysis path, with measured rather than guaranteed execution times.

These limits suggest a direct next experiment: compare batch, hybrid, and fully distributed analysis under matched complete-module workloads on several hosts and architectures, including demanding smoothing settings and actual audio-device deadlines. Shorter completion intervals and native FFT leaves could then be selected against explicit peak-cost and display-age targets. The present evidence is sufficient to motivate that comparison without assuming its outcome.

8 Conclusion

CoopFFT spreads spectral analysis across the audio processing loop by saving traversal state and resuming on successive sample calls. Scheduling windowing, magnitudes, smoothing, and output in the same way brings the complete analysis under one schedule. A balanced integer quota completes each analysis over a chosen hop and makes the added publication delay explicit.

The measured benefit is smaller processing bursts, with costs in overhead and result age. Matched controls establish the placement effect; native controls and Rack measurements show when it matters and when a hybrid is preferable. The central lesson is to evaluate total cost, peak block time, and spectrum age together when choosing how an analyzer should share an audio engine.

Availability

Source, compact evidence, build instructions, and the separately retained technical derivations are available in the Fourier repository. This manuscript is a preprint and has not been peer reviewed by a venue.

References

[1] VCV, "Plugin API guide," *VCV Rack Manual*. Accessed September 28, 2026. https://vcvrack.com/manual/PluginGuide.

[2] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. doi:10.1090/S0025-5718-1965-0178586-1.

[3] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus, "Real-valued fast Fourier transform algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 6, pp. 849–863, 1987. doi:10.1109/TASSP.1987.1165220.

[4] J. B. Allen and L. R. Rabiner, "A unified approach to short-time Fourier analysis and synthesis," *Proceedings of the IEEE*, vol. 65, no. 11, pp. 1558–1564, 1977. doi:10.1109/PROC.1977.10770.

[5] F. J. Harris, "On the use of windows for harmonic analysis with the discrete Fourier transform," *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, 1978. doi:10.1109/PROC.1978.10837.

[6] P.-C. Lo and Y.-Y. Lee, "Real-time FFT algorithm applied to on-line spectral analysis," *Circuits, Systems, and Signal Processing*, vol. 18, no. 4, pp. 377–393, 1999. doi:10.1007/BF01200789.

[7] J. Hurchalla, "A time distributed FFT for efficient low latency convolution," in *129th Audio Engineering Society Convention*, paper 8257, 2010. https://secure.aes.org/forum/pubs/conventions/?elib=15679.

[8] E. Battenberg and R. Avižienis, "Implementing real-time partitioned convolution algorithms on conventional operating systems," in *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, Paris, France, September 2011. https://ericbattenberg.com/pdf/partconvDAFx2011.pdf.

[9] E. Jacobsen and R. Lyons, "The sliding DFT," *IEEE Signal Processing Magazine*, vol. 20, no. 2, pp. 74–80, 2003. doi:10.1109/MSP.2003.1184347.

[10] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. doi:10.1109/JPROC.2004.840301. Planner API: https://www.fftw.org/fftw3_doc/Planner-Flags.html (accessed September 30, 2026).

[11] J. Pommier and contributors, "PFFFT: A pretty fast FFT," software; VCV Rack integration, `dsp::RealFFT`. Accessed September 30, 2026. https://github.com/VCVRack/pffft. Rack FFT wrapper source.

[12] Apple, "Using Fourier Transforms," *vDSP Programming Guide*, Accelerate framework documentation. Accessed September 30, 2026.

[13] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*, pp. 63–74, 2013. doi:10.1145/2464157.2464160. Corrected author manuscript: https://kar.kent.ac.uk/33611/.

[14] VCV, "Rack engine implementation," source revision `8c33d966d329e4a6e354593b2b5f9ac2df5a03bd`, inspected September 30, 2026. https://github.com/VCVRack/Rack/blob/8c33d966d329e4a6e354593b2b5f9ac2df5a03bd/src/engine/Engine.cpp.