SparSDR: Sparsity-proportional Backhaul and Compute for SDRs

Moein Khazraee, Yeswanth Guddeti, Sam Crow
Alex C. Snoeren, Kirill Levchenko[†], Dinesh Bharadia, and Aaron Schulman

UC San Diego [†]University of Illinois at Urbana-Champaign

ABSTRACT

We present SparSDR, a resource-efficient architecture for software-defined radios whose backhaul bandwidth and compute power requirements scale in inverse proportion to the sparsity (in time and frequency) of the signals received. SparSDR requires dramatically fewer resources than existing approaches to process many popular protocols while retaining both flexibility and fidelity. We demonstrate that our approach has negligible impact on signal quality, receiver sensitivity, and processing latency.

The SparSDR architecture makes it possible to capture signals across bandwidths far wider than the capacity of a radio's backhaul through the addition of lightweight frontend processing and corresponding backend reconstruction to restore the signals to their original sample rate. We employ SparSDR to develop two wideband applications running on a USRP N210 and a Raspberry Pi 3+: an IoT sniffer that scans 100 MHz of bandwidth and decodes received BLE packets, and a wideband Cloud SDR receiver that requires only residential-class Internet uplink capacity. We show that our SparSDR implementation fits in the constrained resources of popular low-cost SDR platforms, such as the AD Pluto.

CCS CONCEPTS

• **Hardware** → **Digital signal processing**; **Hardware accelerators**; • **Networks** → Wireless access points, base stations and infrastructure.

KEYWORDS

Software Defined Radio; Sparsity; Cloud SDR; FFT; FPGA

ACM Reference Format:
Moein Khazraee, Yeswanth Guddeti, Sam Crow, Alex C. Snoeren, Kirill Levchenko, Dinesh Bharadia, and Aaron Schulman. 2019. SparSDR: Sparsity-proportional Backhaul and Compute for SDRs. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307334.3326088

1 INTRODUCTION

Software-defined radios (SDRs) have long promised extreme flexibility: in principle, they can capture, decode, and analyze a variety of signals across arbitrary frequency ranges. SDRs derive their versatility from the decoupling of the radio frontend—including analog RF components and digitizers—from the signal processing backend, which is typically deployed on general-purpose computing devices like CPUs [31], GPUs [19], or re-programmable DSPs [3]. When equipped with wideband analog-to-digital converters (ADCs), SDRs can capture a broad range of frequencies, and support for new protocols can be added by implementing the requisite signal processing in software [24, 32].

The universality of the SDR architecture, however, comes at a price: raw samples need to be backhauled between the frontend and the signal-processing backend. Most existing SDR implementations are inefficient in that they require backhaul capacity and processing performance in proportion to the SDR's sampling frequency, irrespective of the (typically far more limited) bandwidth of the signal being captured. As a result, wideband (e.g., 100-MHz) SDR frontends often "throw away" information via digital downsampling in order to reduce the datarate of the backhaul stream.

For example, the popular USRP N210 frontend captures 14-bit I/Q samples at 100 Msps, but must perform 4× digital downsampling to 25 Msps in order to operate within the capacity of its 1-Gbps backhaul link. Even at this reduced bandwidth, signal processing requires desktop-class compute performance, foreclosing the possibility of using embedded processors (e.g., the CPU on a Raspberry Pi). Unfortunately, this limitation renders the N210 impractical for infrastructure deployments. For instance, a general-purpose IoT gateway needs to support frequency-hopping protocols such as Bluetooth and ZigBee that span 80 MHz of bandwidth (Fig. 1 left). Such an application requires 2.5 Gbps of backhaul capacity and server-class processing capability. (Although a single transmitter could be aliased into a narrower bandwidth, the full 80-MHz bandwidth is required to support simultaneous transmissions.)

However, it is well known that the RF spectrum is sparsely occupied across both time and frequency [17, 23, 28]. We present SparSDR, a sparsity-proportional architecture for wideband SDRs that takes advantage of this fact. In SparSDR, we propose a signal-processing extension for existing SDRs whose backhaul and compute requirements scale in proportion to the bandwidth of the captured *signal*, not the RF spectrum sampled (Fig. 1 right). We demonstrate that for many commonly used bands, SparSDR significantly reduces backhaul and compute requirements. Because SparSDR preserves the raw samples of the captured signal, it does not sacrifice the flexibility or protocol independence of SDR.

The added processing in SparSDR must satisfy the following requirements: (1) it must not significantly reduce overall signal quality, (2) it must allow for signals that have low SNR, such as

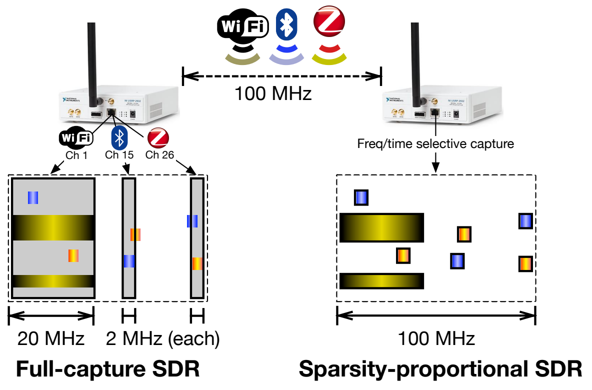


Figure 1: SparSDR's goal is to make SDRs capture primary transmissions rather than entire channels.

below-the-noise transmissions from IoT protocols, and (3) it must fit in the unused processing resources of existing SDR compute platforms. We satisfy all three of these constraints by repurposing a classic signal analysis technique, short-time Fourier transform (STFT) [13, 33]. STFT is traditionally used for power-spectrum analysis: observing the primary frequency components that exist in a signal, and how they change over time. The key property of STFT that makes it possible to achieve sparsity proportionality is that only the primary frequency components need to be backhauled to reverse the transform and recover the raw samples.

Although STFT is reversible, existing algorithms generally recreate the entire raw sample stream—hence, while the backhaul requirements may be reduced, the signal processing task remains proportional to the original sample rate, not the bandwidth of primary signals in the capture. In SparSDR, we introduce a sparsity-proportional reconstruction algorithm that recovers only a portion of the captured signal. Specifically, it reverses only part of the STFT, effectively using STFT itself to identify and isolate the primary signals. The algorithm also corrects for the phase and frequency offsets introduced by STFT-based downsampling.

In this paper, we evaluate the tradeoffs in the parametrization of STFT-based downsampling; the size of the FFT and the type of window used can both significantly impact the downsampling size, signal quality, and latency. We find that by applying a simple threshold to detect primary signals, it is possible to detect signals below the noise floor. We also demonstrate that STFT fits in the unused processing resources of current SDR frontends. Therefore, by replacing the downsampling found in existing SDRs with STFT, we can backhaul and process signals with bandwidth and performance requirements inversely proportional to the sparsity of the signals.

Using SparSDR, we demonstrate that a USRP N210 can can continuously capture 100 MHz while sending 14-bit I/Q samples—4× its backhaul capacity. Specifically, we show that it is possible to simultaneously receive 450 Bluetooth Low Energy (BLE) transmissions per second across the full 2.4-GHz band using a Raspberry Pi 3+ connected to a SparSDR-enabled USRP N210 operating at its full 100-MHz bandwidth. We also show that SparSDR makes it possible to build a Cloud SDR platform out of low-end wideband SDRs (e.g., an Analog Devices Pluto), residential-class backhaul, and inexpensive computing platforms (i.e., the Raspberry Pi 3+). The limited backhaul bandwidth in residential networks has traditionally forced crowd-sourced SDRs to operate with only narrowband (3.2-MHz) USB SDR dongles [25, 29]. Finally, we demonstrate that battery-powered wideband SDRs have sufficient processing resources to incorporate SparSDR's additional stages, including the USRP E310 and USB-powered SDRs such as the AD Pluto and USRP B210.

In summary, our contributions are as follows:

(1) We describe how the STFT can be employed as a lightweight frequency/time-selective downsampling algorithm that produces protocol-independent raw samples that requires backhaul proportional to the bandwidth of the captured signal. We also present a lightweight time-domain reconstruction algorithm which uses partial STFTs to downsample to the primary signals in a capture (Section 3).
(2) We evaluate the tradeoffs associated with STFT-based downsampling and demonstrate that the resource-constrained USRP N210—and even the ~$100 AD Pluto—have sufficient unused resources to support SparSDR (Section 4).
(3) We evaluate SparSDR in two case studies: an IoT gateway built using a USRP N210 connected to a Raspberry Pi, and a wideband Cloud SDR that operates over residential Internet-class backhaul links (Section 6).

SparSDR is open source. We have developed a SparSDR module for GNU Radio, making it easy to drop into existing projects. The hardware implementation is parametrized to enable porting to new SDR platforms (Section 5); we provide example hardware implementations for the USRP N210 and AD Pluto. The hardware and software source can be found at:

https://github.com/ucsdsysnet/sparsdr

2 MOTIVATION

In this section, we motivate the need for frequency and time *sparsity-proportional* SDRs. First, we provide an example of how wideband SDR captures are sparse in frequency and time. Then, we describe how simple downsampling in today's SDRs selects only one contiguous band at a time—indiscriminately throwing away signals—and leaves many spare processing resources.

2.1 Popular bands are sparsely occupied

The motivation for SparSDR comes from the observation that there is a mismatch between the sparsity of spectrum usage and the full-capture design of SDRs. Commodity SDR RF frontends are often built with wide-band ADCs that can capture more than 50 MHz, making it possible for them to capture many channels and protocols simultaneously. In the following experiment, we demonstrate that the popular 100-MHz wide 2.4-GHz ISM band is sparsely occupied—even though it is shared by a variety of protocols such as WiFi (three orthogonal 20-MHz channels), Bluetooth (79 1-MHz channels), and ZigBee (sixteen 2-MHz channels).

We use the OneRadio[1] wideband SDR to capture a 45-second 125-MHz bandwidth snapshot of the entire 2.4-GHz band in an office building during business hours. Fig. 2 shows a 150-millisecond portion of the capture. Even in this short period, it is evident that although the 2.4-GHz band is active (there are WiFi packets being

[1]http://www.oneradiocorp.com/oneradio-receiver/technology

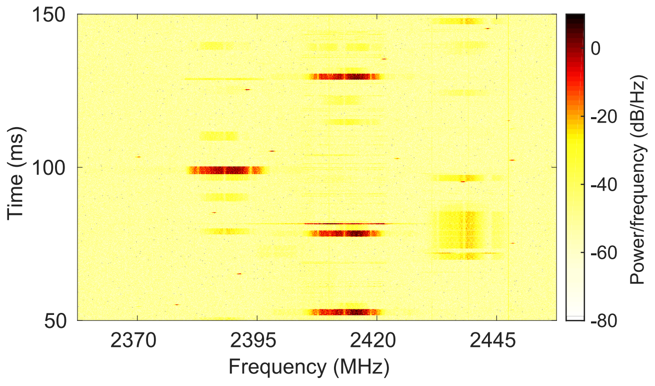


Figure 2: A snapshot of the 2.4-GHz band collected by a OneRadio wideband (125-MHz) SDR shows that even popular unlicensed bands are sparsely occupied.

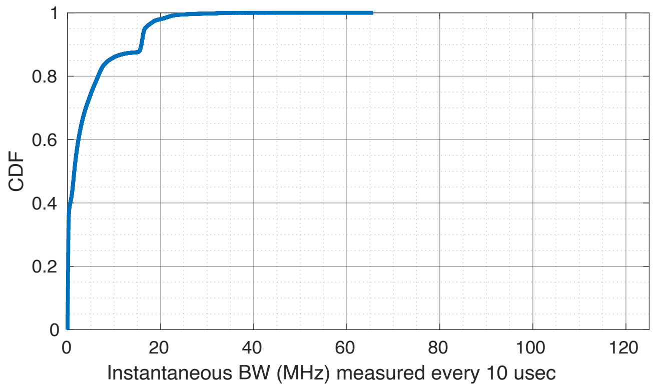


Figure 3: Distribution of occupied bandwidth over 10 microsecond intervals at 2.4 GHz. We use a relative power threshold of 5 dB above the maximum observed noise floor to determine if a frequency is occupied.

transmitted), its usage is sparse in frequency and time. Fig. 3 shows the distribution of instantaneous bandwidth occupancy every 10 microseconds across the entire 45-second capture. This figure shows that the wide-band (125-MHz) capture is sparse: the band is unoccupied for almost 40% of the time and the occupancy is less than 18 MHz (approximately one WiFi channel) 85% of the time.

2.2 SDRs need smarter downsampling

SDR frontends often downsample to reduce the capture bandwidth to allow for the use of inexpensive and widely available backhaul links. These inexpensive links limit the capture bandwidth: the USRP N210 can only capture 25 Msps on its gigabit-Ethernet backhaul, and the AD Pluto can only achieve ~5 Msps over its USB 2.0 backhaul (with four bytes per I/Q sample). Downsampling is also necessary to operate within the compute resources available. For instance, processing a 100-Msps capture on a desktop-class 3-GHz CPU only allows for 30 cycles per sample and is simply infeasible on an inexpensive embedded processor like that found on the Raspberry Pi.

This is an unfortunate state of affairs, because many signals are indiscriminately thrown away during downsampling. However, the fact that the downsampling logic is often implemented on an FPGA at the SDR frontend presents an opportunity to modify the frontend to downsample more intelligently. Many SDR frontends have additional processing resources intended to support smarter downsampling, the implementation must be compact in order to fit into the FPGAs of existing SDRs.

3 DESIGN

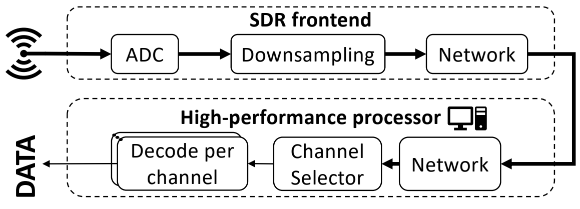
SparSDR is an add-on for existing software-defined radios, shown in Fig. 4, that makes them *sparsity-proportional*: they only require backhaul capacity and compute resources in inverse proportion to the sparsity of the signals they receive. The primary component of SparSDR is a frequency-and-time selective downsampling step that replaces the basic downsampling logic in the SDR frontend. With this modification, the SDR frontend only backhauls samples at frequencies that have active signals. To support frequency-and-time selective downsampling, SparSDR also inserts an extra processing step to reconstruct the raw time-domain samples at the backend compute platform (e.g., CPU, GPU, or DSP). We introduce an efficient reconstruction algorithm that recreates the time-domain signals while maintaining the same level of sparsity. Unlike compressed sensing, SparSDR does not require new frontend hardware (i.e., random sampling ADCs), nor any assumptions regarding the sparsity of the captured signal, and the reconstruction algorithm can operate within the constrained performance of embedded platforms (e.g., Raspberry Pi).

Here we focus exclusively on making the receiver side of SDRs sparsity-proportional. SDR receivers are are more computationally intensive than transmitters: receivers produce a constant stream of samples that must be processed in real time. We note, however, that SparSDR can be modified to operate on the transmit side of SDRs by reversing its steps: The frequency-and-time selective downsampling algorithm can be reversed to construct a signal that is sparse in the frequency domain, and upsample it to the full bandwidth of the transmit front-end (e.g., DACs). A potential application could be to efficiently transmit multiple, concurrent BLE packets (2 Msps) at varying frequencies in the 80-MHz ISM band.

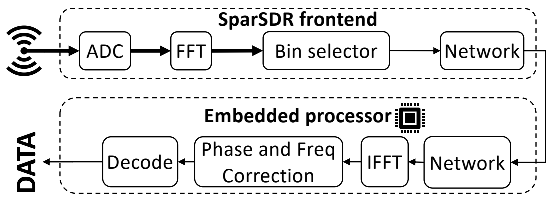
3.1 Frequency-and-time downsampling

We begin by describing SparSDR's downsampling algorithm that generates a sample stream whose bandwidth is inversely proportional to the signal sparsity in the captured spectrum. To retain the flexibility and protocol independence of SDR, SparSDR's downsampling must be general: it cannot be specific to a particular communication protocol.

We draw inspiration from a popular spectrum analysis measurement: power spectral density. Typically observed with a spectrum analyzer, the power spectrum reveals the frequency and magnitude of received signals. A common tool for performing power spectral analysis is the Short-Time Fourier Transform (STFT) [33]. STFT divides the sample stream into overlapping shorter windows of equal length and then computes the Fourier transform on each shorter window. This process reveals the frequencies of the signals that are active in that short window of time. For instance, we use the STFT to generate Fig. 2, showing sparse usage of the 2.4-GHz spectrum.



(a) Full-capture SDR



(b) Sparsity-proportional SDR (SparSDR)

Figure 4: (a) Full-capture SDRs require backhaul and compute resources that are equal to the bandwidth of the downsampled ADC capture. (b) SparSDR introduces new blocks that make resource usage proportional to the sparsity of the captured signals.

Our key insight is that the STFT algorithm can be repurposed to downsample an SDR frontend's ADC capture in frequency and time. We can then detect the frequency bins which are active and backhaul only those bins. The resulting downsampling is proportional to the activity in spectrum. For example, for the capture shown in Fig. 2 we would backhaul just the few active frequency and time components during the transmissions. Importantly, STFT is invertible and protocol independent. Even when backhauling only the active frequency and time components, we can reconstruct the active signals accurately.

A natural question is then: Can we compute STFT-based downsampling in real time on SDR frontends? The STFT transform is computed by repeatedly performing an FFT operation on short windows of a capture. The use of FFTs means the STFT computation can be hardware-accelerated to operate in real time. Specifically, we rely on pipelined streaming FFT hardware implementations that take in one time-domain sample and output one FFT frequency bin per clock cycle. Efficiency improves with increasing FFT (capture-window) length, but it also increases decoding latency (detailed in Section 5.1). Additional latency may be problematic if a transmitter needs to respond quickly after receiving a packet (e.g., by sending an acknowledgment).

An effective STFT-based downsampling approach must address several complexities, however, that we discuss below in turn: (1) the FFT must be parameterized to produce a sparse frequency-domain representation of each signal contained in the capture; (2) a detection algorithm is needed to select the active frequency bins to be backhauled; and (3) the frequency bins must be efficiently reconstructed back into raw samples. We discuss each of these issues in turn in the remainder of this section.

3.2 Sparse representation of signals with STFT

We first describe how we parameterize the STFT in SparSDR to produce a sparse frequency-domain representation of a capture without harming signal quality. In particular, we select an STFT windowing function and determine the overlap between windows.

A naive approach, commonly called rectangular windowing, would be to use no windowing function and divide the capture into non-overlapping short windows of samples. However, rectangular windowing does not produce a sparse representation in the frequency domain. Fig. 5 shows the frequency response curves of the rectangular window compared to other common STFT window functions. The rectangular window leaks energy into adjacent bins with its main lobe at -13db. Due to the sharp transitions at the edges

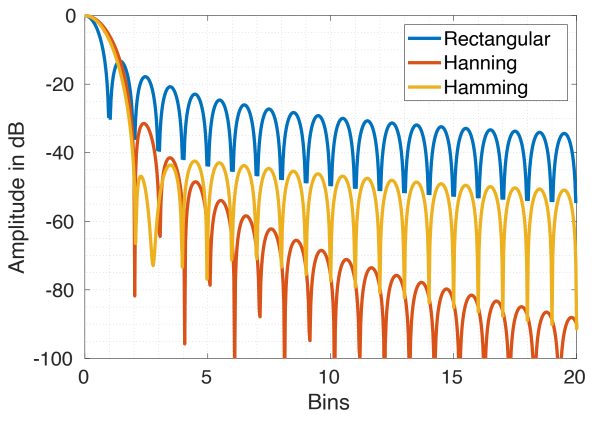


Figure 5: Frequency response of common window functions

of the rectangular window, the leakage does not degrade rapidly, even at bins that are far from the center. This leakage negates the benefits of STFT-based downsampling because the signal's energy is spread over many frequency bins, each of which would need to be backhauled in SparSDR. Moreover, leakage poses a problem for signals that are close in frequency which would therefore be mapped to the same STFT bins.

In contrast to the rectangular window, generalized cosine windows attenuate the amplitude of the samples at the edges of the STFT. Fig. 5 plots the frequency response of two common cosine windows, Hanning and Hamming. The figure shows that cosine windows can provide sparse representation of signals in the frequency domain by reducing energy leakage into adjacent frequency bins. These window functions are preferable for SparSDR because they reduce the number of bins that must be backhauled to reconstruct the signal [27, Ch. 7, p. 468]. For example, Fig. 5 suggests that SparSDR need only backhaul a few bins adjacent to the signal if Hamming or Hanning windowing functions are employed.

Unfortunately, there is a complication with the direct application of cosine windows: they significantly attenuate the time-domain samples at the edges of each window. Simply inverting the window only amplifies the noise (quantization) at the edges of each window. We observe, however, that samples at the edges of a cosine window can be perfectly recovered by overlapping sequential windows by half of the window length (the overlapped region of the sinusoids adds up to one). While overlapping windows by 50% introduces a bandwidth and computation overhead of 2× compared to a non-overlapping rectangular window, we demonstrate in Section 4.1 the

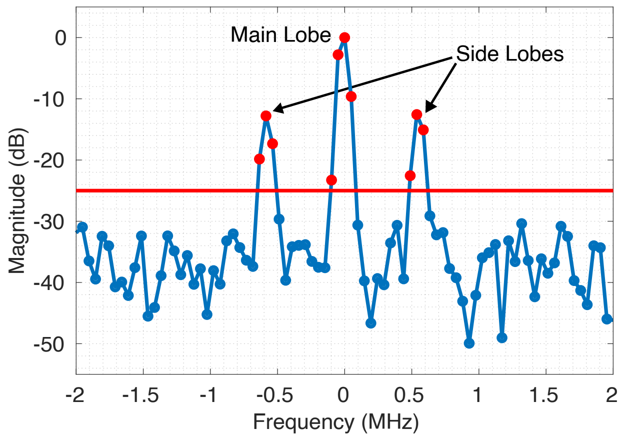


Figure 6: An example of SparSDR's threshold-based energy detector applied to an STFT of a Bluetooth signal. Thresholding a Bluetooth signal results in backhauling at most 10 bins (0.5%) of a 2048-bin STFT for a 100 MHz capture.

tradeoff is worthwhile for SparSDR: overlapping cosine windows provide more than a 2× reduction in backhaul bandwidth.

Compared to the gradual decline in frequency response of the Hamming window, we find the Hanning window's steep decline is better-suited for SparSDR. Although the Hamming window has a lower first sidelobe, Hanning requires backhauling fewer bins because the energy is concentrated in the closest few bins. Therefore, we select the Hanning window as the default window function for SparSDR. If custom windowing is desired, SparSDR's software can load new windowing coefficients into the SparSDR frontend FPGA.

3.3 Universal signal detection for STFT

Next we describe how SparSDR selects the frequency bins in an STFT that contain signals of interest. A typical approach to detect a signal is to use time-domain correlation, namely searching for a protocol-specific synchronization sequence (e.g., a preamble) [25]. We eschew time-domain correlation in SparSDR precisely because it requires protocol-specific processing, which would undermine the flexibility of SDR frontends. Time-domain correlations are not universal: they requires the knowledge of the correlation sequence for each protocol. Said differently, it can only detect signals whose correlation sequences are known *a priori*. Furthermore, time-domain correlation requires per-protocol processing, making real-time multi-protocol frontends hard to implement.

Instead, SparSDR uses a threshold-based energy detector, an efficient and universal method of detecting signals. It is universal because it does not depend on any signal-specific patterns, and efficient because it does not require per-protocol computation. SparSDR compares the magnitude of each FFT bin with a threshold. If the magnitude is above the threshold, the bin is backhauled, otherwise the bin is dropped.

Fig. 6 shows an example of what SparSDR's energy-based thresholding looks like for a 1-MHz Bluetooth signal. In this example, we simulated capturing the Bluetooth with 100 MHz of bandwidth and a 2048-bin STFT. Notice that SparSDR does not need to backhaul the entire 1 MHz Bluetooth signal (~20 bins). The reason is, Bluetooth

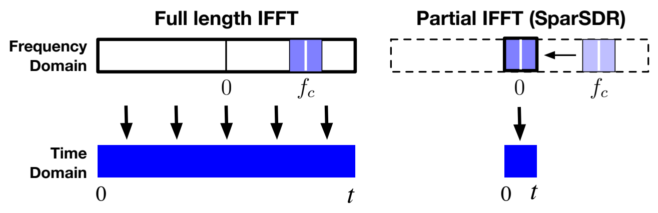


Figure 7: Compute-efficient reconstruction with partial IFFT

is frequency modulated, so each STFT only contains a few symbols. Therefore, SparSDR only needs to backhaul the main lobe and the side lobes for those symbols, which together only occupy only half of the Bluetooth bandwidth (10 bins).

While energy-based signal detection can miss decodable signals that are below the noise—such as distant ZigBee transmissions—we observe that this is mitigated by the natural oversampling SparSDR provides by performing the STFT on the full capture bandwidth. STFT bins are averaged over many time-domain samples (e.g., 2048) captured at a high sample rate (e.g., 100 Msps) that is several times faster than required for ZigBee. We evaluate the benefits of oversampling for energy-based detection in Section 4.3.

The thresholds for energy-based signal detection must be set carefully to avoid detecting spurious signals. A receiver may have non-uniform noise in the frequency domain, and harmonics from out-of-band transmitters may alias into the capture bandwidth. To address this issue, SparSDR provides a configurable threshold for each FFT bin that can be updated constantly as conditions change.

To help select bin thresholds, SparSDR monitors the average energy of each bin. The monitoring is implemented on the SDR frontend's FPGA as an exponentially weighted moving average (EWMA). The benefit of using a weighted average is that it has a streaming implementation that requires only limited FPGA resources. When a new FFT value is available for bin n, the FPGA updates the bin average using the following formula:

$$Avg\,(n)_{new} = \alpha \cdot Avg\,(n)_{old} + (1 - \alpha) \cdot Value\,(n)\,.$$

The length of the averaging window (α) is configurable at the SDR backend. A short window is useful for monitoring the noise floor of a bin that has dynamic transmissions. Such monitoring is useful in ISM bands, where the noise floor must be distinguished from signals originating from many transmitters that are received at different energy levels. A longer window is effective for determining if a bin is occupied by a constant transmitter (e.g., TV and radio broadcasts). Such constant transmissions consume a large amount of backhaul resources and might not be of interest for a particular application of SparSDR. To make it possible to ignore these signals, we support a bin-masking feature: masking acts like a notch filter to avoid backhauling specific signals that are not of interest.

3.4 Efficient reconstruction of partial STFTs

The final component of SparSDR is the efficient reconstruction of time-domain signals from the frequency-domain STFT bins that are backhauled to the processing backend. These time-domain signals need to be recovered at their original sample rate (e.g., 2 Msps for BLE) in order to prepare them for decoding or other application-specific signal processing. To achieve our vision of

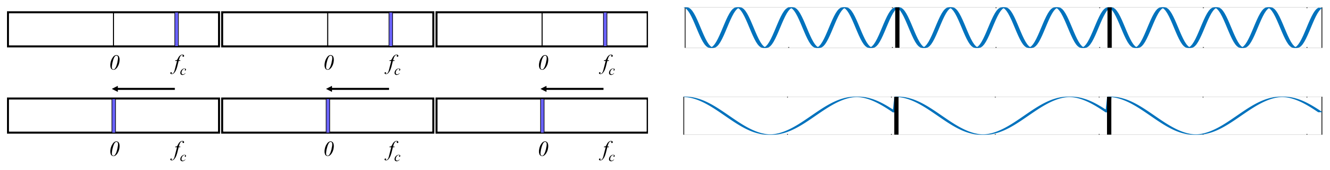


Figure 8: Downconverting the frequency-domain bins results in a discontinuity in the time domain.

sparsity-proportional compute, this reconstruction step must be efficient: the compute resources required should be proportional to the data rate of the partial STFTs that are being backhauled.

A straightforward approach to reconstruction would be to perform an IFFT that is the same length as the FFT executed on the SDR frontend. This would require performing a full-length IFFT on the bins backhauled from the SDR frontend, with other bins set to zero (Fig. 7 left). The time-domain samples output by this full-length IFFT must be downconverted to baseband by multiplying by a sinusoid, and then downsampled from the full capture rate to the signal's sample rate. Clearly, the amount of processing is independent of the number of bins backhauled.

A more efficient system would directly reconstruct the time-domain samples at (or close to) the desired signal sample rate by performing an IFFT with a length equal to the number of backhauled frequency-domain bins from an STFT. This approach results in a sparsity-proportional reconstruction process as well, where the computation required is proportional to the number of active bins. As shown in the right-hand side of Fig. 7, SparSDR, performs an IFFT on only part of the full window length, simultaneously downconverting the center frequency of the signal to baseband and downsampling to an appropriate sample rate.[2]

3.5 Phase offset compensation

Indeed, a partial IFFT can reconstruct the time-domain signal. However, it also introduces a time-domain discontinuity in the signal. The reason is that taking the partial IFFT implicitly downconverts the signal to baseband. This change in center frequency creates phase discontinuities at the boundaries of the time-domain windows. To better understand this artifact, consider the example depicted in Fig. 8. A sinusoid at frequency f_c is converted to the frequency domain by an STFT across three windows (top left). The time-domain signal after an IFFT is a perfect sinusoid (top right). If we perform a simple downconversion by shifting the bins to the center frequency (lower left), changing the frequency introduces a phase discontinuity at each window boundary (lower right).

The discontinuity in phase is due to the windowing effects of the STFT. Downconverting a signal in the time domain involves multiplying the signal by a sinusoid $e^{(-j2\pi f_c)}$ at the signal's center frequency f_c. In the frequency domain, this operation is equivalent to shifting the frequency axis by f_c as follows:

$$X(f - f_c) \leftrightarrow x(t)\, e^{-j2\pi f_c t}.$$

[2]If the frequency offset and sample rate are not an integer multiple of the frequency bin spacing, the time-domain samples must be corrected with additional downconversion and downsampling. Fortunately, these corrections do not add significant computational overhead because they are performed after the initial STFT-based downsampling.

In a windowed FFT, the start time of each segment increments for each subsequent segment. Therefore, each segment has an additional phase term which is proportional to the time at which the segment was captured t_{start} and the shift in frequency f_c. For each segment $x_k(t) = x(t_{start} + (0 : \frac{N-1}{f_s}))$, the downconversion can be computed as:

$$\begin{aligned} X_k(f - f_c) &\leftrightarrow x_k(t)\, e^{-j2\pi f_c \left(0 : \frac{N-1}{f_s}\right)} \\ &= x\left(t_{start} + (0 : \frac{N-1}{f_s})\right) e^{-j2\pi f_c \left(0 : \frac{N-1}{f_s}\right)}. \end{aligned}$$

When taking an IFFT of part of a full-length FFT window, this introduces a downconversion that is equivalent to multiplying each window with a sinusoid of zero initial phase $e^{j2\pi f_c (0:(l-1)/f_s)}$, where l is the length of the partial IFFT. In other words, taking a partial IFFT is equivalent to multiplying the signal in the time domain with a sinusoid at f_c whose phase begins at zero at the start of each window—independent of the time at which the window was captured. Therefore, SparSDR can compensate for the phase discontinuity by introducing an additional phase offset of $e^{-j2\pi f_c t_{start}}$. The result extends to the overlapped STFT windows that are needed to faithfully reconstruct signals; the phase correction must be applied before overlapping the windows.

4 EVALUATION

In this section we evaluate how the parameterization of SparSDR affects the efficiency of downsampling in terms of backhaul throughput. We evaluate three parameters: STFT window type, STFT length, and threshold value. This investigation reveals the tradeoffs that must be made between efficiency and reconstruction accuracy.

4.1 STFT window function

In the first experiment, we evaluate how the selection of a window function affects SparSDR's ability to backhaul only the bins needed to faithfully reconstruct a signal. For each window function, we observe the decrease in the reconstruction error of a sinusoid as we increase the number of bins used for reconstruction. For all window functions, we use a fixed STFT length of 2048, and for the cosine windows we add 50% overlap. We measure the reconstruction error as the Error Vector Magnitude (EVM) [11, Chapter 5.1.3] of the ideal sinusoid compared to the reconstructed signal.

Fig. 9 shows the results of this experiment. As more bins are used, the rectangular and Hamming windowed STFT do not provide a significant reduction in error. However, the Hanning window provides a significant drop in reconstruction error with each new bin that is added: backhauling 30 bins for every 2048 time-domain samples results in an extremely low reconstruction error (-60 dB).

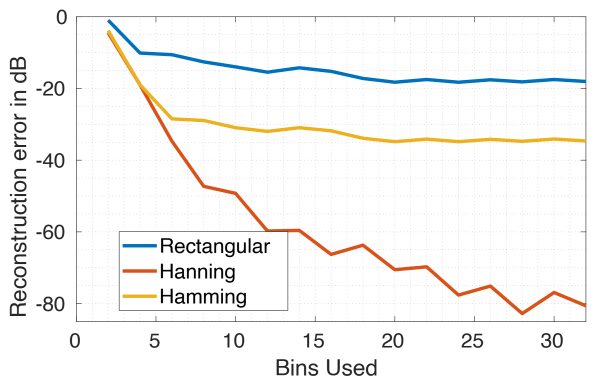


Figure 9: Window function affects the number of bins that must be backhauled to accurately reconstruct a sinusoid.

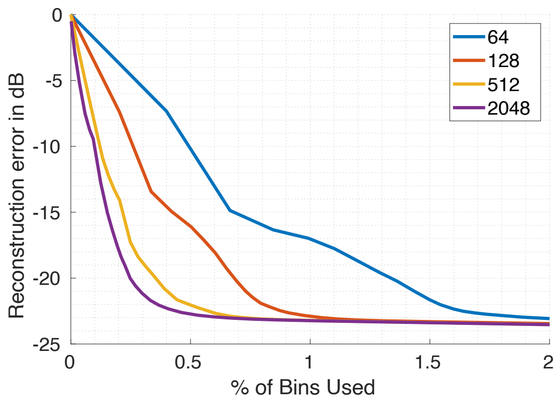


Figure 10: Longer STFTs result in improved backhaul efficiency without degrading reconstruction fidelity.

(Two bins must be backhauled for each STFT bin due to the required 50% overlap.)

4.2 STFT length

Next we evaluate how the STFT length affects the fraction of the STFT bins that need to be backhauled to faithfully reconstruct the signal. The benefit of using a longer STFT is that it increases the STFT's frequency precision: each frequency bin represents a smaller frequency band. Although longer STFTs require backhauling more bins to represent a fixed bandwidth, these bins also represent an equally longer period of time. Therefore, the backhaul bandwidth is proportional to the fraction of the STFT bins, regardless of the STFT length.

We expect that increasing STFT length will reduce the fraction of bins that need to be backhauled. The reason is that the number of bins into which a signal falls depends mainly on the window type. For example, Hanning window's main lobe primary leaks into two adjacent bins for STFT lengths below 2048. Therefore, a signal that spans n bins would require backhauling two additional bins on each side. This results in an overhead of $4/(n+4)$. As the STFT length decreases, the number of bins needed for a signal also decreases, yielding an increase in the relative overhead.

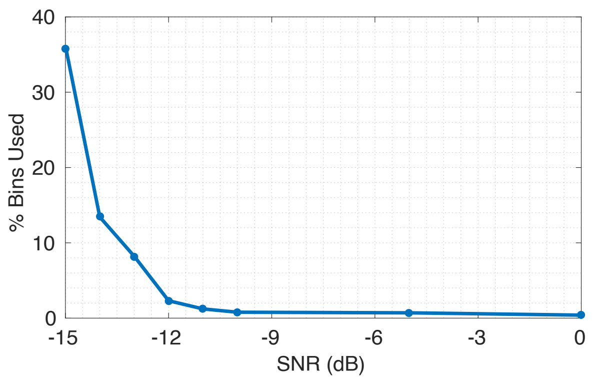


Figure 11: Average fraction of STFT bins that were above the threshold to achieve BER of less than 10^{-5} for different SNR values. Evaluated signal is IEEE 802.15.4 (ZigBee) PHY with 2 MHz of bandwidth.

To compare the performance of different STFT lengths, we generate a 1-MHz Bluetooth signal at 100 Msps using MATLAB. This is a realistic signal that requires backhauling multiple bins to reconstruct with high fidelity. Fig. 10 plots the results of capturing this signal using four STFT lengths (64, 128, 512, 2048) with a 50%-overlapped Hanning window. We reconstruct the signal with an increasing fraction of the STFT length, and compute the reconstruction error. Fig. 10 shows the results of this experiment. As expected, larger STFTs achieve a lower reconstruction error rate for any given fraction of bins, and achieve very low error with a small fraction of bins (-23 dB while using only half a percent in this instance).

4.3 Threshold value

Finally, we evaluate how the energy threshold value we select affects the fidelity of the reconstructed signals. Lowering the threshold to a level close to noise increases the rate of spuriously backhauled bins. Increasing the threshold significantly above the noise may reduce the sensitivity to weak signals.

Detecting signals in SparSDR is made easier because the SDR frontend oversamples (captures at a sample rate higher than the required for the signal). Performing an STFT on oversampled signals introduces averaging gain in the frequency domain, effectively lowering the noise floor. This separates weak signals from the noise floor, making them easier to detect. To be precise, there is $10 \cdot \log_{10}\left(\text{capture}_{\text{bw}}/\text{signal}_{\text{bw}}\right)$ averaging gain added by performing an STFT. Hence, if the capture bandwidth is wide enough, below-the-noise signals, such as weak ZigBee transmissions, can be detected.

In the following experiment, we evaluate the effectiveness of thresholding to detect below-the-noise signals (i.e., negative SNR). We generated IEEE 802.15.4 (ZigBee) signals and added noise that was equal to or higher power than the signal. We detected the signals with threshold values in steps of 1 dB for each SNR, reconstructed the signals with only these bins, and decoded the signals to measure their BER. We found the maximum threshold that resulted in a Bit Error Rate (BER) of less than 10^{-5}: we call this the "maximum decodable threshold". Building on the results of the prior experiments in this section, these experiments are run with a Hanning-windowed 2048-bin STFT. The 2 Msps ZigBee samples

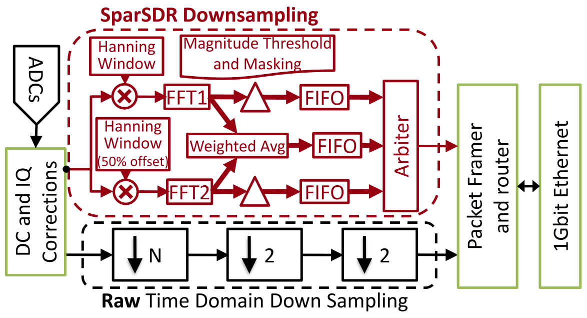


Figure 12: Overview of SparSDR's FPGA-based downsampling pipeline. The time-domain downsampling module is replaced by the frequency-domain SparSDR downsampling module.

were then upsampled to the full capture bandwidth of 100 Msps before applying the STFT.

In this experiment, the averaging gain results in an increase of $10 \cdot \log_{10}(50) = 17$ dB in SNR, effectively separating the signal from background noise. As described in Sec. 3.3, the bandwidth of the signal is 2% of the full capture bandwidth, so we expect 2% or fewer of the bins to be above the threshold.

Fig. 11 presents the average fraction of the STFT bins that were above the maximum decodable threshold for SNRs ranging from -15 to 0 dB. At -12 dB SNR and above, the minimum decodable threshold selects at most 2% of the bins. Between -12 dB and -15 dB SNR, there is a rapid increase in the number of bins above the minimum decodable threshold. For SNR -15 dB and below, backhauling all the bins is not sufficient to capture a decodable signal.

5 IMPLEMENTATION

In this section, we describe how to add SparSDR to an SDR's FPGA-based frontend and backend software stack. The frontend modification replaces the existing downsampling modules in the FPGA with SparSDR hardware modules, and backend reconstruction is performed on the host within GNU Radio. Specifically, we detail the downsampling hardware design, backhaul packet format, and backend reconstruction software. We demonstrate that SparSDR fits in the resources of typical SDRs, and that the design is portable to many different SDR platforms.

5.1 SparSDR's downsampling hardware

We begin by describing how SparSDR's frontend architecture can downsample the full capture bandwidth in real time using the limited resources of an SDR frontend's FPGA. Our implementation fits into the spare resources of two resource-limited SDRs: the classic USRP N210 and low-end AD Pluto. Further, we describe how the SparSDR implementation is portable to other SDR platforms because of its parametrized design, as well as being lightweight enough to fit in their spare resources. Finally, we describe how STFT length and other parameters affect the FPGA resource requirements.

Architecture. Fig. 12 shows the overall architecture of the SparSDR downsampler for the USRP N210. The first step is to transform the time-domain samples captured by the ADC into the frequency domain. The challenge is that the N210's FPGA runs at the same clock frequency as the full-capture sample rate; therefore the design needs to be fully pipelined so that at every clock cycle a new time sample is accepted, and an output frequency sample is passed through the SparSDR downsampling filter.

To address this challenge, we use the streaming version of the Xilinx FFT core. Since SparSDR requires overlapping windows by 50%, we instantiate two of these FFT modules and set their start triggers to fire half a window size apart. We store the parameters of SparSDR locally in block RAMs. These include constant values such as the Hanning-window coefficients, per-bin threshold values, and bin masks. We implement windowing in a streaming manner as well by reading two coefficients half a window apart and multiplying them by the time-domain samples before they are input into the FFT. The outputs of the two FFT modules are also half a window apart. Each output bin is individually compared to the threshold value and mask. If the magnitude of the bin is above the threshold and the bin is not masked, then that bin is sent to the network module so it can be backhauled to the SDR host.

Due to the sparsity of SparSDR downsampling, it is not possible to predict when a bin will be above the threshold or which of the two overlapped FFTs will have an above-threshold bin. To resolve this problem and make reconstruction more straightforward, we include an arbiter module that sends all the active bins from one FFT window before switching to the next overlapped window generated by the other FFT module. On top of that, the periodic FFT bin magnitude averages have higher priority than FFT bin values. The arbiter orchestrates the order which data is sent to the networking module. This reordering and interruption by average samples, as well as potential network bottlenecks, requires FIFOs for the FFT outputs and also the averaging module.

Backhaul packet format. Next, we describe the format for backhauling STFT bins from the SDR frontend to the host. The information needed to reconstruct the signal from the frequency domain bins is as follows: alongside the I/Q value of the FFT bin, we need to send the STFT bin index and the timestamp of the STFT. The I/Q data for an FFT bin is 4 bytes, and we we use an additional 4 bytes per FFT bin for the metadata. The maximum FFT size we support in the USRP N210 implementation of SparSDR is 2048, so 11 bits are used for the FFT index, 1 bit is used to distinguish an FFT sample from an average, and 20 bits remain for the timestamp. To make it possible for the host to distinguish between the two overlapped FFTs, the timestamp includes the start time of each half window.

To make the timestamp range as large as possible, we synchronize the FFT bin 0 output to occur when the USRP N210's internal timer lower bits are zero. Hence, the timestamp of each half window has trailing zeros that can be dropped. For an FFT of size 2048, this process results in a total timestamp size of 30 bits—20 bits of metadata and 10 bits from the zeros for each half window—sufficient for 10.7 seconds at the 10-ns sample clock interval of the USRP N210.

Portability. To make SparSDR portable to other popular SDRs that have different resource constraints, we parameterize the hardware code. There are three design parameters that can be passed to the SparSDR modules: the maximum length of the FFT that an implementation should support (fixed at synthesis time to a power of two), the size of the FIFO buffer for sending FFT samples, and the

	Use CLB Logic				Use XtremeDSP Slices			
MAX FFT size	256	512	1024	2048	256	512	1024	2048
DSP	12	16	16	20	30	38	40	48
Block RAM	1	3	4	7	1	3	4	7

Table 1: FPGA resource requirements for different maximum FFT window sizes.

Module	Slice Reg	LUT	LUT RAM	BRAM	DSP48A	Instances
SparSDR Top	623	1022	128	0	16	1
FFT	4006	4121	1191	7	20	2
Thresholds Mem	0	0	0	4	0	1
Averages Mem	0	0	0	2	0	2
Masks mem	0	0	0	1	0	1
FFT Samples FIFO	23	61	0	4	0	2
Avg. Samples FIFO	25	67	0	7	0	1
Backhaul send FIFO	29	71	0	34	0	1

Table 2: Resource utilization for different modules in our USRP N210 implementation.

size of the FIFO buffer for sending average samples. These parameters are governed by the maximum sample rate of the SDR frontend, as well as the spare resources available on the FPGA (see below). In addition, to achieve timer and FFT output synchronization, FFT module latency values (shown in Table 4) need to be updated in a specific Verilog file. These latencies can be different for different FPGAs.

The primary challenge of porting SparSDR to a new platform is understanding the top module of the new platform and how the samples are backhauled. For example, porting from the USRP N210 to the AD Pluto took about two weeks. Most of the time was spent understanding the hardware design and software stack of the Pluto, followed by developing an FFT wrapper for the AXI-Stream version of the Xilinx FFT core. Additional hardware development effort would be required to use an FFT module other than the legacy or AXI-Stream versions of the Xilinx FFT module.

Resource utilization. Driven by our evaluation that demonstrates that longer FFTs produce more efficient backhaul usage (Section 4.2), we tried to fit the largest FFT length possible into the remaining FPGA resources. As shown in Table 1, the maximum FFT length dictates the resources needed for the FFT module. Furthermore, changing the length proportionally changes the required memory for thresholds, masking, and averaging (Table 2) because these units use Block RAMs to hold the constant coefficients or values for each bin. Finally, the FIFO sizes determine how well the system can tolerate bursts of data for FFT output and running averages.

Satisfying these constraints resulted in a length-2048 FFT for the USRP N210 and length-1024 FFT for the AD Pluto. In addition to the scalable FFT-related modules, the top module requires a fixed amount of resources for a command decoder as well as windowing and average calculations which require DSP blocks. The sampling rates of the N210 and Pluto platforms are 100 Msps and 61.44 Msps respectively. The top of Table 3 shows the resource utilization of SparSDR on these platforms compared to their baseline implementations. The bottom of Table 3 shows available FPGA resources of some popular SDR modules. Our lightweight implementation can easily fit into high-end FPGAs like the one in the USRP X310.

Implementation / SDR module	Slice registers	LUT	BRAM	DSP	Price ($)
Baseline N210	20287	31248	41 x 18Kb	31 x 18b*18b	-
N210 + SparSDR	29066	39935	115 x 18Kb	87 x 18b*18b	-
Baseline Pluto	14365	7801	4 x 18Kb	56 x 18b*25b	-
Pluto + SparSDR	23103	13682	33 x 18Kb	74 x 18b*25b	-
AD Pluto	32500	17600	120 x 18Kb	80 x 18b*25b	150
USRP N210	47744	47744	126 x 18Kb	126 x 18b*18b	2000
USRP E310	106400	53200	280 x 18Kb	220 x 18b*25b	3050
USRP B210	184304	92152	268 x 18Kb	180 x 18b*18b	1200
USRP X310	508400	254200	1540 x 18Kb	1590 x 18b*25b	5400

Table 3: Resource requirements for our SparSDR implementations, and available FPGA resources of various popular SDR models.

FFT size	8	16	32	64	128	256	512	1024	2048
Latency (cycles)	49	105	137	215	343	613	1125	2162	4210

Table 4: Latency of the FFT module for different window sizes.

Latency. Table 4 shows the latency, for different window sizes, added by the FFT module with a maximum FFT window size of 2048. Windowing and thresholding add another 5 cycles to these FFT latencies. The monitoring module calculates and updates the running average per bin, but it is not on the signal latency path. Since the arbiter orders the windows, there is a half window delay for sending the second half of the previous window before starting the new window. The other source of delay is periodic transmission of full average window values. As such, the primary parameter to adjust the latency added by SparSDR's downsampling hardware is FFT length. In addition, FFT and average sample FIFOs add some delay, but these delays are negligible compared to the latency introduced by the network module. For instance, the network module may not transmit a packet to the host until it is filled, and there are kernel latencies on the host side.

5.2 SparSDR's host-based reconstruction

The software implementation of SparSDR reconstruction can provide real-time performance even on an embedded processor, and is integrated into GNU Radio. We designed a custom GNU Radio block that performs SparSDR's reconstruction. The SparSDR block produces reconstructed I/Q samples that can be passed to existing signal processing software (e.g., decoders).

SparSDR's host software can send commands to the SDR frontend to update SparSDR's parameters without resynthesizing the hardware. These parameters include FFT length, threshold and mask values, windowing coefficients, averaging weight, and interval between average samples. The software can also disable output of FFT samples or average samples. This allows each application to tailor these parameters in real time based on the performance tradeoffs described earlier in this section.

The software also provides error reporting. Namely, if the SDR frontend tries to send bins faster than the network and software can handle, the software detects the overflow. If the user desires, the software can configure the parameters to avoid overflow and resume the capture.

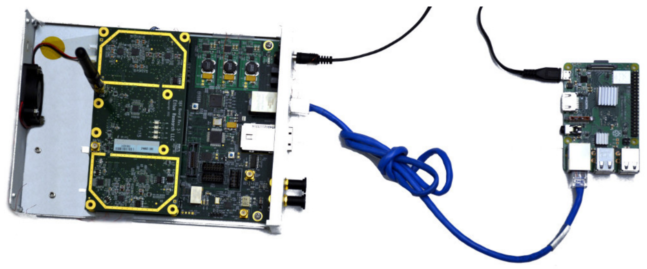


Figure 13: SparSDR reconstructs 100-MHz captures in near real time on a Raspberry Pi.

6 CASE STUDIES

In this section we demonstrate the power of SparSDR with two case studies: an SDR-based IoT gateway and a Cloud SDR for the VHF and UHF bands. We select these applications because they both involve processing sparse signals across wide capture bandwidth on platforms with limited resources. Specifically, the IoT gateway has constrained compute resources, while the Cloud SDR has constrained backhaul bandwidth. A summary of the benefits of using SparSDR instead of a full-capture SDR for these applications is as follows:

- **IoT Gateway:** Using SparSDR, Bluetooth signal processing can be performed on an SDR host with compute resources proportional to the activity rather than the 80-MHz bandwidth of Bluetooth. We demonstrate that reconstruction processing is so lightweight that a Raspberry Pi 3+ can process 450 BLE packets per second from channels spread across the entire 80-MHz spectrum.
- **Cloud SDR:** With SparSDR, we perform an over-the-air experiment that demonstrates that most VHF and UHF bands ($\leq$ 10 MHz) are sparse, and therefore only require residential-class Internet uplink speeds to backhaul the signals to the cloud for processing.

6.1 IoT gateway

For many IoT applications, sensors intermittently upload data to an IoT gateway [2]. Monitoring all of the transmissions in the 100-MHz ISM band from Bluetooth [4], BLE [10], and ZigBee [34] simultaneously creates new capabilities. For instance, such a system could reduce the energy wasted by sensors channel hopping to find a channel in common with the IoT gateway. It also makes it possible to build a universal IoT gateway that is compatible with the myriad current and future IoT protocols [1, 26].

Today the only SDR platform that could achieve this would be a wideband SDR frontend with at least 10 Gbps of backhaul capacity, such as the USRP X310. Processing this sample stream would require a desktop-class processor. We demonstrate that the sparsity-proportional compute enabled by SparSDR makes it possible to monitor all of the popular IoT frequencies, and even decode signals, with the embedded-class processor of a Raspberry Pi 3+.

To demonstrate the benefits of SparSDR as an IoT gateway, we perform three controlled experiments on the following testbed: We connect a USRP N210 with the SparSDR module to a Raspberry Pi 3+ as shown in Fig. 13. We use the SBX-120-MHz RF frontend on the USRP N210 operating at full bandwidth and full (100-Msps)

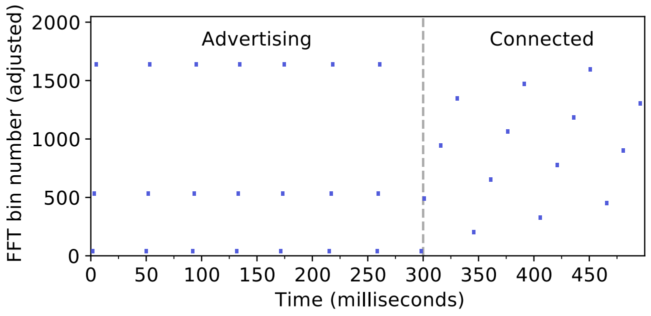


Figure 14: IoT transmissions are sparse in time and frequency. For both BLE modes of operation (advertising and connected) SparSDR only backhauls active FFT bins—about 45 bins—of a 2048 length FFT capturing 100 MHz.

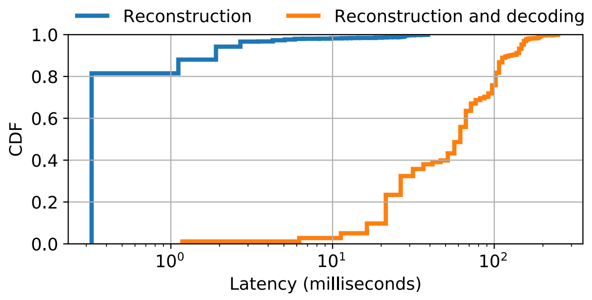


Figure 15: Added latency from reconstruction is significantly less than decode latency.

sampling rate. Our "sensors" are up to three ESP32 modules that we use to transmit BLE packets. First, we evaluate SparSDR's ability to receive BLE's wideband frequency-hopping transmissions. Then, we demonstrate how lightweight the reconstruction processing is and how much delay it adds to the system. Finally, we stress-test SparSDR's backhaul and processing capabilities.

Receiving wideband frequency-hopping transmissions. First we demonstrate that SparSDR is able to continuously monitor the full 80-MHz band for BLE packets with embedded-class compute. We set up one ESP32 as a BLE sensor that advertises a GATT service. Another ESP32 connects to the first and exchanges data. Fig. 14 shows the STFT bins that are backhauled to the SDR host during this experiment. SparSDR only backhauls bins during the short advertisement packets across all three advertising channels. It works similarly during the frequency-hopping exchange across all 40 BLE channels. BLE transmissions are ideal for SparSDR because they are sparse in both time and frequency. A single BLE transmitter uses at most 2 MHz of bandwidth at a time, or about 45 active FFT bins, which translates to a peak backhaul throughput of 280 Mbps. This is amortized over time due to the bursty nature of BLE signals.

Lightweight reconstruction and induced latency. To evaluate the latency introduced by reconstruction in SparSDR, we transmit BLE packets to SparSDR from an ESP32 on a fixed band. We then decode the BLE packets on the host with *gr-bluetooth*. We measure two different latencies: first, the time from the first bin backhauled for a packet to when that sample is reconstructed, and second to

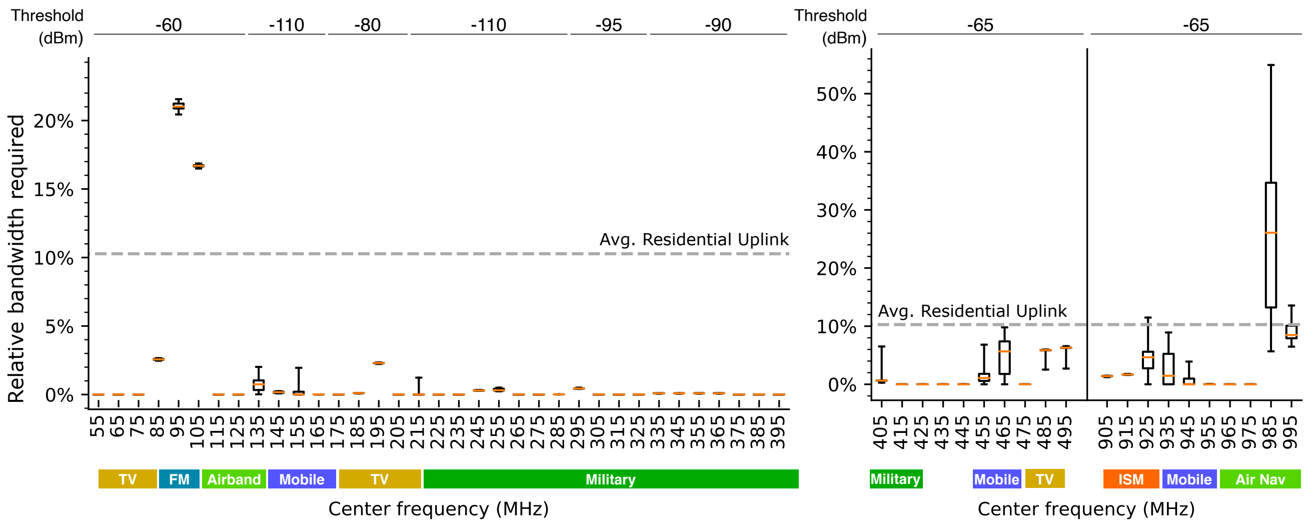


Figure 16: The distribution of SparSDR's backhaul for capturing transmissions in 10-MHz bands from 50 MHz to 1 GHz. Bandwidth is plotted relative to the 320 Mbps needed by a standard 10-MHz SDR. We also indicate the capacity of a typical residential-class Internet uplink.

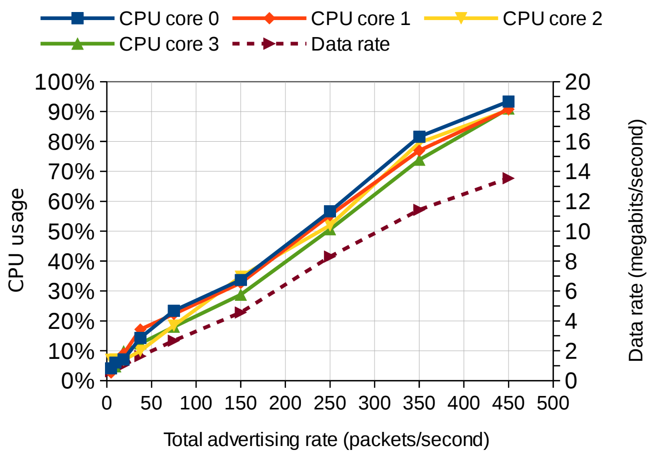


Figure 17: SparSDR's backhaul and compute scale linearly with the rate of received BLE advertising packets.

when the packet is fully decoded. Fig.15 shows the distribution of SparSDR's reconstruction latency. The reconstruction latency is ~200 μs for 80% of the packets, while decoding latency is between 10 ms to 100 ms for 80% of the packets. This demonstrates that the reconstruction latency is negligible compared to the Bluetooth decode latency.

Backhaul and processing requirements on RPi. Next we evaluate how efficiently SparSDR reduces backhaul and processing requirements. We set up three ESP32s to advertise on the three BLE advertising channels. With just a Raspberry Pi, we monitor 100 MHz of bandwidth and decode up to 450 BLE advertisement packets per second. Fig. 17 shows the CPU utilization and required backhaul data rate as the BLE packet rate increases from the three transmitters. The reconstruction and decoding processes use all four CPU cores and do not fully saturate the CPU until the packet rate reaches 450 packets per second. We verify that the decoder was operating without errors. It is important to note that the Raspberry Pi 3+ has a 1-Gbps network interface, but it is connected using USB 2.0 so it can only achieve 224 Mbps.

We acknowledge that the current system is not a full IoT Gateway. In these experiments we only focus on the receive side, and do not implement a multi-protocol scheduler. For multi-protocol decoding, there is a need to either run a process for each protocol or operate a scheduler that distributes packets among a limited number of decoders. Designing such a system is beyond the scope of this paper, but may be a potential area for future work.

6.2 Cloud SDR

The next case study shows that SparSDR makes it possible to expand Cloud SDR deployments to wideband SDRs. We demonstrate that SparSDR significantly reduces the backhaul bandwidth requirements of a Cloud SDR: specifically, backhauling the VHF and UHF bands only requires residential-class Internet uplink speeds.

We set up a USRP N210-based SparSDR receiver to measure the backhaul data rate required to capture real-world over-the-air signals from a roof-mounted antenna. We mount the antennas on a building in a populated area near waterways, an airport, and a large military base. We use a dipole VHF antenna with a WBX-40-MHz frontend to capture signals below 400 MHz, and wideband discone antenna with an SBX-120 MHz for the UHF frequencies. We measure the backhaul data rate for each of the 10-MHz bands from 50 MHz to 1 GHz.

Fig. 16 shows the observed distributions of the per-second backhaul data rate computed over a period of one minute. For each 10-MHz band, the box plot shows the 0%, 25%, 50% and 75%, and 100% quartiles of the data rate. The y axes plots the SparSDR backhaul rate relative to the full-capture data rate (320 Mbps). The plot also shows how the backhaul data rate compares to the bandwidth of a typical U.S. residential internet connection uplink (32.88 Mbps)[3]. We break the frequency range into 10-MHz bands because that is the approximate allocation size of the different uses of the VHF and UHF bands (shown on the bottom of the figure). Fig. 16 also shows the absolute power threshold value that we employ for each band. These thresholds are selected by finding the minimum threshold value that does not cause overflow due to noise. We determine the absolute power by calibrating the USRP power measurements with a signal generator.

The primary result of this case study is that SparSDR's backhaul data rate is often significantly below the backhaul bandwidth required for backhauling a single 10-MHz band, or even several adjacent 10-MHz bands. The "Mobile" and "ISM" bands are the best use cases for SparSDR because they contain dynamic transmissions. For many of these bands, the entire distribution of throughput measurements are well below the typical residential uplink speed.

As expected, the bands that have constant broadcasters (e.g., FM) do not benefit much from sparsity-proportional downsampling. This is also why we do not present the bands between 500 MHz and 900 MHz because they are occupied by constant LTE downlink channels and broadcast TV stations.

7 RELATED WORK

While SparSDR is the first work that demonstrates a lightweight and near-real-time SDR implementation whose backhaul bandwidth and computation resource requirements vary with signal sparsity, SDR architectures have been investigated extensively. We survey a few of the most relevant pieces of related work below.

SDRs exploiting frequency-time sparsity. We acknowledge that compressed sensing [7, 8] has a similar goal to SparSDR; however, it requires *a priori* knowledge of how sparse the signal is, so it is not adaptable to dynamic conditions. Furthermore, compressed sensing requires random-sampling-based ADCs which are hard to create and require computationally intensive algorithms to reconstruct the original signal [22]. Compressed sensing has therefore been very costly and difficult to realize in real time. Similarly, while sparse FFT [14] can be used in place of STFTs in SparSDR, they require strict sparsity assumptions. In general, there is no way to ensure *a priori* that the captured spectrum will meet the sparsity constraints.

There has been prior work that detected signals to take advantage of sparsity. For example, Narayanan and Kumar detect narrowband signals are over time by correlating the received samples with a predefined preamble sequence common to various protocols in [25]. Their work is complementary to SparSDR in that they can detect all signals in a given narrowband in the time domain, whereas SparSDR detects signals in the frequency domain.

In contrast to SparSDR, BigBand [15] uses a complementary approach to wideband spectrum sensing and decoding. Hassanieh *et al.* alias multiple RF signals into a capture band, and reconstruct the individual signals within the aliased capture. BigBand could use SparSDR to reduce its backhaul and computation requirements at arbitrary capture bandwidths.

Crowdsourced Cloud SDRs and IoT SDRs. Cloud SDR systems such as ElectroSense [29] and RadioHound [21] have emerged from the development of low-cost SDRs such as the RTL-SDR[4] and low-cost hosts such as the Raspberry Pi [20]. Researchers have also made the case for a universal IoT SDR built on such low-cost platforms [9, 25]. Research on crowdsourced SDR infrastructure has focused on giving inexpensive SDRs the capabilities of premium SDRs such as the USRP, namely full-spectrum tuning range [21, 29] and accurate frequency measurement [6]. With crowdsourced SDR infrastructure, researchers have demonstrated that by combining information from many networked SDRs they can estimate the occupancy of the spectrum [18], reveal anomalous transmissions [30], and even jointly decode wideband signals received from many narrowband sensors [5]. Existing implementations of these applications require making a tradeoff: Either the low-cost SDRs do a significant fraction of the processing locally—only allowing a wideband SDR to be used for one application at a time—or the SDRs must backhaul raw I/Q samples, limiting their deployments to locations with well-provisioned backhaul (i.e., universities and businesses). SparSDR significantly reduces the backhaul requirements of wideband SDRs so they can operate across residential-class access links as well.

C-RAN. There are several proposals for compressing the backhaul of cellular networks from SDR-like basestations that send raw I/Q samples to the cloud for centralized processing. These captures are not sparse: the entirely of the basestation's bandwidth is occupied by the LTE signal. Also, the compression algorithm is running on carrier-grade equipment that is well-provisioned compared to popular SDRs [12, 16].

8 CONCLUSION

SparSDR presents an ideal architecture for upcoming applications like Cloud SDR and IoT gateways. It can be implemented on many SDR frontend FPGAs to make their backhaul capacity requirements inversely proportional to the sparsity—both in time and frequency—of the signal in any part of the RF spectrum, instead of the full ADC capture bandwidth. Further, SparSDR employs a reconstruction process whose computational requirements also scale in response to the sparsity. Hence, it significantly decreases processing demands—so much so that many applications can be run on an embedded processor. These two features are key to enabling a scalable, low-cost SDR solution. SparSDR delivers near-real-time performance without sacrificing signal quality or flexibility. We have released SparSDR under an open source license to enable others to develop new applications on top of it.

ACKNOWLEDGEMENTS

We thank our shepherd Tam Vu, and the anonymous reivewers for their detailed feedback. Also, we thank the members of the Communication Sensing and Networking group at UC San Diego for their comments on early versions of this paper. We are appreciative of the generosity of Ryan Kastner and Alireza Khodamoradi for lending us their wideband SDR equipment. This work was supported in part by NSF Award CNS-1646493.

[3]https://www.speedtest.net/reports/united-states/2018/fixed

[4]https://www.rtl-sdr.com/

REFERENCES

[1] G. Aloi, G. Caliciuri, G. Fortino, R. Gravina, P. Pace, W. Russo, and C. Savaglio. A mobile multi-technology gateway to enable IoT interoperability. In *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 259–264. IEEE, 2016.

[2] L. Atziori, A. Iera, and G. Morabito. The Internet of things: A survey, 2010.

[3] M. Bansal, A. Schulman, and S. Katti. Atomix: A framework for deploying signal processing applications on wireless infrastructure. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[4] Bluetooth SIG. *Bluetooth Core Specification*, 12 2016. Rev. 5.0.

[5] R. Calvo-Palomino, D. Giustiniano, V. Lenders, and A. Fakhreddine. Crowdsourcing spectrum data decoding. In *INFOCOM*, 2017.

[6] R. Calvo-Palomino, F. Ricciato, D. Giustiniano, and V. Lenders. Ltess-track: A precise and fast frequency offset estimation for low-cost sdr platforms. In *WINTECH*, 2017.

[7] E. Candes and J. Romberg. Sparsity and incoherence in compressive sampling, 2006.

[8] D. L. Donoho. Compressed sensing. *IEEE Trans. Inform. Theory*, 52:1289–1306, 2006.

[9] P. Dutta and I. Mohomed. emergence of the IoT gateway platform. *GetMobile: Mobile Computing and Communications*, 19(3):33–36, 2015.

[10] C. Gomez, J. Oller, and J. Paradells. Overview and evaluation of Bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[11] Q. Gu. *RF System Design of Transceivers for Wireless Communications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[12] B. Guo, W. Cao, A. Tao, and D. Samardzija. CPRI compression transport for LTE and LTE-A signal in C-RAN. In *7th International Conference on Communications and Networking in China*, pages 843–849, Aug. 2012.

[13] F. J. Harris. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE*, 66(1):51–83, jan 1978.

[14] H. Hassanieh, P. Indyk, D. Katabi, and E. Price. Simple and practical algorithm for sparse Fourier transform. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.

[15] H. Hassanieh, L. Shi, O. Abari, E. Hamed, and D. Katabi. GHz-Wide sensing and decoding using the sparse Fourier transform. In *Proc. IEEE Conference on Computer Communications (INFOCOM)*, 2014.

[16] C. I, J. Huang, R. Duan, C. Cui, J. Jiang, and L. Li. Recent progress on C-RAN centralization and cloudification. *IEEE Access*, 2:1030–1039, 2014.

[17] M. H. Islam, C. L. Koh, S. W. Oh, X. Qing, Y. Y. Lai, C. Wang, Y.-C. Liang, B. E. Toh, F. Chin, G. L. Tan, et al. Spectrum survey in singapore: Occupancy measurements and analyses. In *Cognitive Radio Oriented Wireless Networks and Communications, 2008. CrownCom 2008. 3rd International Conference on*, pages 1–7. IEEE, 2008.

[18] A. P. Iyer, K. Chintalapudi, V. Navda, R. Ramjee, V. N. Padmanabhan, and C. R. Murthy. Specnet: Spectrum sensing sans frontieres. In *Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX, 2011.

[19] J. Kim, S. Hyeon, and S. Choi. Implementation of an SDR system using graphics processing unit. *IEEE Communications Magazine*, Mar. 2010.

[20] M. KING. Raspberry Pi to production. https://www.rigado.com/download/raspberry-pi-to-production-whitepaper/.

[21] N. Kleber, J. D. Chisum, A. Striegel, B. M. Hochwald, A. Termos, J. N. Laneman, Z. Fu, and J. Merritt. RadioHound: A pervasive sensing network for sub-6 GHz dynamic spectrum monitoring. *CoRR*, 2016.

[22] J. Laska, S. Kirolos, Y. Massoud, R. Baraniuk, A. Gilbert, M. Iwen, and M. Strauss. Random sampling for analog-to-information conversion of wideband signals. In *IEEE Dallas Circuits and Systems Workshop (DCAS)*, volume 1, pages 119–122, 2006.

[23] M. A. McHenry, P. A. Tenhula, D. McCloskey, D. A. Roberson, and C. S. Hood. Chicago spectrum occupancy measurements & analysis and a long-term studies proposal. In *Proceedings of the First International Workshop on Technology and Policy for Accessing Spectrum*, TAPAS '06, New York, NY, USA, 2006. ACM.

[24] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, May 1995.

[25] R. Narayanan and S. Kumar. Revisiting software defined radios in the IoT era. In *Proc. Workshop on Hot Topics in Networks (HotNets)*. ACM, 2018.

[26] R. Narayanan and C. S. R. Murthy. A probabilistic framework for protocol conversions in IIoT networks with heterogeneous gateways. *IEEE Communications Letters*, 21(11):2456–2459, 2017.

[27] A. V. Oppenheim and R. W. Schafer. Digital signal processing (book). *Research supported by the Massachusetts Institute of Technology, Bell Telephone Laboratories, and Guggenheim Foundation. Englewood Cliffs, N. J., Prentice-Hall, Inc., 1975. 598 p*, 1975.

[28] K. Patil, R. Prasad, and K. Skouby. A survey of worldwide spectrum occupancy measurement campaigns for cognitive radio. In *Devices and Communications (ICDeCom), 2011 International Conference on*, pages 1–5. IEEE, 2011.

[29] S. Rajendran, R. Calvo-Palomino, M. Fuchs, B. V. den Bergh, H. CordobÃĺs, D. Giustiniano, S. Pollin, and V. Lenders. Electrosense: Open and big spectrum data. *IEEE Communications Magazine*, Jan. 2018.

[30] S. Rajendran, W. Meert, V. Lenders, and S. Pollin. SAIFE: Unsupervised wireless spectrum anomaly detection with interpretable features. In *Proc. IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, 2018.

[31] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker. Sora: High-performance software radio using general-purpose multi-core processors. *Commun. ACM*, 54(1):99–107, Jan. 2011.

[32] D. L. Tennenhouse and V. G. Bose. SpectrumWare - A software-oriented approach to wireless signal processing. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 1995.

[33] P. Welch. The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. *IEEE Transactions on Audio and Electroacoustics*, 15(2):70–73, jun 1967.

[34] ZigBee Alliance. *ZIGBEE SPECIFICATION*, 9 2012. Rev. 20.