Integrating Spinal Codes into Wireless Systems

by

Peter Anthony Iannucci

B.S. EECS and Physics
Massachusetts Institute of Technology, 2011

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2013

Certified by .....
Hari Balakrishnan
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Integrating Spinal Codes into Wireless Systems

by

Peter Anthony Iannucci

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Rateless spinal codes [47] promise performance gains for future wireless systems. These gains can be realized in the form of higher data rates, longer operational ranges, reduced power consumption, and greater reliability. This is due in part to the manner in which rateless codes exploit the instantaneous characteristics of the wireless medium, including unpredictable fluctuations. By contrast, traditional rated codes can accommodate variability only by making overly conservative assumptions.

Before spinal codes reach practical deployment, they must be integrated into the networking stacks of real devices, and they must be instantiated in compact, efficient silicon. This thesis addresses fundamental challenges in each of these two areas, covering a body of work reported in previous publications by this author and others [27, 26]. On the networking side, this thesis explores a rateless analogue of link-layer retransmission schemes, capturing the idea of rate adaptation and generalizing the approach of hybrid ARQ/incremental redundancy systems such as LTE [29]. On the silicon side, this thesis presents the development of a VLSI architecture that exploits the inherent parallelism of the spinal decoder.

Thesis Supervisor: Hari Balakrishnan
Title: Professor of Computer Science and Engineering

Acknowledgments

This thesis draws upon work and material from prior publications [27, 26] with thanks to (and permission from) to Jonathan Perry, Kermin Elliott Fleming, Devavrat Shah, and Hari Balakrishnan.

Regarding chapters 2 and 3 [27], I thank the reviewers and Bob Iannucci for helpful comments. Two generous graduate fellowships supported this work: the Irwin and Joan Jacobs Presidential Fellowship (Perry and Iannucci), and the Claude E. Shannon Assistantship (Perry). I thank the members of the MIT Center for Wireless Networks and Mobile Computing, including Amazon.com, Cisco, Google, Intel, MediaTek, Microsoft, and ST Microelectronics, for their interest and support.

Regarding chapters 4 and 5 [26], I thank Lixin Shi for helpful comments and Joseph Lynch for helping to collect hardware results. Support for my work came from an Irwin and Joan Jacobs Presidential Fellowship and from the National Science Foundation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It hardly bears repetition that wireless devices grow in pervasiveness and criticality. Rateless codes promise faster transfers, longer range, lower power, and better reliability for wireless systems under fading conditions. The purpose of this thesis is to defeat two practical barriers to the widespread deployment of rateless codes in general and spinal codes [47] in particular.

1.1 Ratelessness in the Wireless Network Stack

For network applications to perform well on wireless devices, network protocols must cope with substantial variability in the radio frequency (RF) channel. Realized channel conditions depend on a large number of unpredictable spatial factors, as well as interference from sources internal or external to the wireless network. Variation may occur over both human and sub-packet time scales. In addition to channel effects, non-idealities in the receiver introduce further discrepancies between the transmitted signal and the signal that the receiver demodulates. Because such distortions and discrepancies are inevitable, the choice of an appropriate *data rate* for a wireless system is fundamental to obtaining good performance: at a higher rate, the system is more susceptible to decoding failures due to noise and interference, and at a lower rate, it can be made more robust. Designers prefer the highest rate that meets their reliability goals.

A natural question to ask is whether one can completely hide this unreliability in the network – that is, whether a system can be made arbitrarily reliable. In theory, perfectly reliable communication is possible over an additive Gaussian channel in the limit of infinite block length. However, real wireless channels exhibit complicated fading behaviors, and real packets are of finite length, e.g. 1500 bytes. Traditional rated block codes suffer a nonzero outage probability in these settings, because a deep fade may prevent decoding and such a system cannot adapt to correct the mistake. For this reason, virtually every wireless network uses an automatic repeat request (ARQ) protocol in addition to a rate adaptation strategy. When decoding succeeds, the receiver sends an acknowledgement (ACK) message back to the sender to indicate this condition, and the sender repeats the coded block if no ACK arrives within a timeout period, or if a negative ACK (NAK) arrives. Relying on coding alone without feedback is called forward error correction (FEC).

Some networks use a refinement called hybrid ARQ (HARQ), in which a large but not unlimited amount of redundancy is encoded at the sender and portions of it are sent incrementally when a NAK arrives. *Rateless* codes, including LT and Raptor [50, 44], Strider [13, 20], and spinal codes [47], extend HARQ by providing a limitless amount of redundancy at the sender. By contrast, a HARQ system built on a traditional *rated* code (LDPC [17], turbo [6], convolutional [52], etc.) must eventually resort to reusing some redundancy when the effective rate of the system drops below the rate of the underlying rated *mother code*[1], degrading performance at sufficiently low signal-to-noise ratios (SNRs).

Certain rateless codes have two additional properties, which we term "adaptation-freedom" and "computation-variability". An adaptation-free rateless code bypasses the need to choose an appropriate signalling constellation (i.e. a map from bits to volts and back to bits) for the current channel conditions. These codes use one signalling constellation all the time, producing each output *symbol* in the same way,

[1]Mother code: a fixed-rate forward error correction scheme yielding a large supply of redundancy. In a HARQ system, we might initially send only a fraction of this redundancy, yielding a short transmission with an effectively higher-rate encoding. We would only send all of the redundancy if conditions were poor enough to warrant it.

whereas traditional wireless communication systems use a dense constellation at high SNR and a sparse constellation at low SNR. At low SNR, dense constellations are computationally complex to demap; at high SNR, sparse constellations make inefficient use of available bandwidth. Adaptation-free rateless codes have a particularly nice structure: regardless of signal conditions, the signal produced by the sender will be exactly the same, with bandwidth used efficiently and demapping costs kept low. Adaptation-free codes are also regret-free: the receiver never wishes that the sender had chosen a different modulation or code rate.

The computation-variable property captures the effect of the receiver's computational capabilities on the throughput of the system. Spinal codes, which use a heuristic decoder parameterized by a search-breadth parameter B, achieve a level of performance dependent on how hard the receiver tries to decode: that is, how large B is [47]. Strider uses an iterative turbo decoder whose performance depends on the number of iterations used. The sender does not know or need to know how capable the receiver may be for the system to work.

Regardless of the underlying code, a half-duplex sender must determine the appropriate amount of redundancy to send between acknowledgement events, since it cannot receive acknowledgements while transmitting. If it sends too little, then waiting for an acknowledgement will be wasteful because the result will most likely be a NAK. If it sends too much, then time is wasted sending unnecessary redundancy when an acknowledgement event would most likely yield an ACK. This trade-off is modulated by the cost of feedback: if delays are long or synchronization overhead is large, then acknowledgement events should be scheduled sparingly; conversely, if acknowledgements take little time or their cost can be mitigated, then frequent pauses for acknowledgement can be exploited to increase throughput.

For an adaptation-free, computation-variable rateless code, the probability of successful decoding for any given reception is determined by (1) the number of symbols received; (2) the instantaneous channel distortions of each symbol; and (3) the low-level details of the implementation of the decoder and how much computation it uses. Chapter 2 presents a code-agnostic link-layer protocol which adaptively determines how

much incremental redundancy to send between pauses so that throughput is maximized. It strategically controls (1) by observing how the rateless code accommodates (2) and (3). These observations summarize the distortions and decoder specifics in a way that frees the protocol from a need to anticipate every possible scenario and parameter.

While ARQ-based and rateless systems can recover after deep fades without dropping any packets – a feature not shared by FEC systems – they do so at the cost of two unpleasant properties: feedback delays during which the channel is idle, and unpredictable timing jitter. Chapter 3 presents an evaluation showing that the link-layer protocol of chapter 2 achieves high throughput and low delay.

1.2 Spinal Codes in Hardware

The second practical barrier to widespread deployment of rateless codes is the availability of suitable hardware encode and decode logic. In general, the challenges include parallelizing the required computation, and reducing the storage requirement to a manageable level.

At the transmitter, an encoder takes a sequence of message bits (e.g., belonging to a single packet or link-layer frame) and produces a sequence of coded bits or coded symbols for transmission. At the receiver, a decoder takes the (noisy or corrupted) sequence of received symbols or bits and inverts the encoding operation to produce its best estimate of the original message bits. If the recovered message bits are identical to the original, then the reception is error-free; otherwise, the communication is not reliable and additional actions have to be taken to achieve reliability (these actions may be taken at the physical, link, or transport layers of the stack).

Chapter 4 shows that spinal codes are remarkably amenable to efficient implementation in hardware. The encoder is straightforward, but the decoder is tricky. Unlike convolutional decoders, which operate on a finite trellis structure, spinal decoders developed thus far operate on an exponentially growing tree. The amount of exploration at the decoder has an effect on throughput: if a decoder computes sparingly, it will require more symbols to decode and thus achieve lower throughput. This effect is

shown in Figure 4-1. A naïve decoder targeted to achieve the greatest possible coding gain would require hardware resources to store and sort upwards of a thousand tree paths per bit of data, which is beyond the realm of practicality. Chapter 4 shows that this level of computation is not necessary, and that some architectural finesse leads to a spinal decoder that is competitive in area and throughput with LTE turbo codes. Finally, chapter 5 validates the design with an FPGA implementation and on-air testing.

Chapter 2

No Symbol Left Behind[1]

Rateless codes offer a natural way to adapt to channel variations via the *prefix property*: in a rateless code, an encoding with a higher rate is a prefix of any lower-rate encoding. Rather than discarding symbols which fail to decode, the receiver uses them again to form part of a lower-rate encoding once additional symbols arrive. No symbol is left unused in decoding a rateless message. The greater the number of symbols received, the higher the probability of a successful decoding.

Ratelessness and adaptation-freedom do not excuse the sender and receiver from adapting to channel conditions. Instead, they unify the processes of sensing conditions and reacting to them. For a good rateless code, the number of symbols required for decoding closely tracks changes in the prevalent channel conditions. This means that a rateless sender and receiver can skip the customary estimation of channel quality and focus on the central problem of estimating, based on feedback, the number of symbols to be transmitted.

A comprehensive rateless link protocol would include a mechanism for reliable feedback and a mechanism for dealing with channel contention. It would allow application-level end-to-end latency constraints to impose limits on packet aggregation, and it would provide theoretical guarantees that a suitable performance metric is optimized. It would circumscribe what the sender needs to know about the channel

in order to optimize performance, and it would include the cost of conveying this information to the sender in the optimization.

This chapter identifies efficiency as the right performance metric and shows how the sender can maximize it given the feedback delay and a cumulative probability distribution function called the decoding CDF. Efficiency, given by the fraction of channel occupation time which is strictly necessary for reliable delivery, provides a code-neutral measure of protocol overhead. This chapter and the following one also develop an efficiency-maximizing protocol, **RateMore**, and presents its implementation and evaluation over spinal codes [47], Strider [20], and Raptor codes [50, 44] on stationary and fading channels. RateMore accommodates soft single-hop latency constraints and provides reliable delivery. Multi-hop latency constraints and unreliable service classes are interesting unexplored directions.

RateMore assumes common half-duplex radios with 802.11-style framing and acknowledgments, and considers operation above a minimum signal-to-noise ratio (SNR) such that ACKs sent at the lowest 802.11 convolutional code rate are reliable. RateMore also assumes that once a sender has contended for the medium, it uses 802.11's short inter-frame space (SIFS) mechanism to bypass contention during feedback hand-offs until the transaction is complete. Under these conditions, the feedback delay is known to the transmitter a priori via a calculation reviewed below. The resulting analysis shows that the decoding CDF and the feedback delay are sufficient knowledge for the sender to maximize the average throughput of the link over all possible strategies of transmitting data and pausing for feedback.

The decoding CDF is not only sufficient, but it is practical to obtain. Chapter 3 demonstrates that the sender can obtain approximate yet satisfactory knowledge of the decoding CDF (e.g. performing within 1.57% of full knowledge) from just a handful of acknowledgments.

The evaluation of chapter 3 compares RateMore with two alternatives borrowed from fixed-rate coding. The first is an analogue of automated repeat requests (ARQ) fashioned after 802.11, and the second is incremental redundancy after the pattern of the 3GPP cellular standard, sending a fixed number of additional coded symbols be-

tween pauses for feedback. RateMore proceeds from a sound mathematical framework and outperforms both of these ad hoc approaches.

These results show that RateMore achieves an efficiency of over 90% across all the experiments. The experiments also show that RateMore reduces overhead by 2.6× to 3.9× compared to 802.11-style classical ARQ and by 2.8× to 5.4× compared to 3GPP-style "Try-after-n" hybrid ARQ. These translate to throughput improvements of up to 26% even under "static" channel conditions. Over fluctuating Rayleigh-fading channels, RateMore performs within 1.57% of the ideal adaptation.

2.1 Related Work

2.1.1 Type-I HARQ

Hybrid ARQ (HARQ) systems [37] marry forward error correction with ARQ to raise the probability of successful reception. Type-I HARQ systems transmit coded messages and retransmit on failures, and are widely used in wireless standards such as Wi-Fi [28]. A common design technique is to specify several modes, or "bit rates", each suitable for a small range of channel conditions. Bit rate adaptation algorithms then choose what mode to use by picking a combination of modulation (signalling constellation) and code.

2.1.2 Type-II HARQ

Type-II HARQ allows the exchange of coded data over several round-trip interactions between the transmitter and receiver. Using more interactions has been shown to allow improvement over ARQ's performance under poor conditions, while not sacrificing performance under favorable conditions. Two approaches exist for combining transmissions. In HARQ with Chase Combining [10], the retransmission repeats parts of the original packet, which are then combined at the receiver. HARQ with Incremental Redundancy (IR) sends different coded bits in every transmission. This approach has been used successfully in the 3GPP LTE protocol [29], and a protocol

has been built to achieve IR-HARQ over 802.11 networks in ZipTx [36].

Soljanin et al. discuss the design of HARQ with incremental redundancy [51]. The develop such a protocol for an ensemble of LDPC codes as well as Raptor codes. The basic idea is to transmit only as many coded symbols as required for the receiver to decode the message with high probability using the best possible maximum-likelihood (ML) decoding rule, presupposing a very high SNR. If the receiver is successful, the transmitter moves to the next message. If not, the transmitter sends the next sets of coded symbols in such a way that given what receiver has received (the transmitter knows the past channel condition at the receiver), there is a high probability of successful ML decoding if the SNR is only moderately high. This is repeated until successful decoding is achieved.

Another related work is by Anastasopoulos [2]. The primary difference arises in the fact that their work does not take feedback delays into account. They model the decoding CDF in terms of an error exponent, and assume that the underlying code cannot decode with more than some maximum number of symbols. They model the channel as a Markov chain and try to infer its state. The approach taken by RateMore is different both in the learning strategy (learning the decoding CDF) and in applying a dynamic programming strategy to compute the transmission schedule.

Most proposed HARQ schemes have a small number of effective rates to choose from. This is achieved using rate-compatible puncturing [22, 33, 21], where the mother code is partitioned into a series of punctured words, such that the decoder has good probability of successfully processing any prefix of the series. In comparison, rateless codes do not require such gentle constructions, and natively offer a free choice of rates.

Systems with either Type-I or Type-II HARQ schemes use bit rate adaptation algorithms. These algorithms take a *reactive* approach to combating temporal channel variations: systems passively monitor channel conditions in the form of the signal-to-noise ratio [9, 25], frame loss rate [54, 7], or bit-error rate [53, 49].

2.1.3 Partial Packet Recovery

Partial packet recovery schemes are protocols designed to recover packets that were received with errors. Data is divided into small fragments, and some method is used to detect which fragments were received correctly. The transmitter then only retransmits erroneous fragments. Seda [18] adds an overhead of a CRC-8 and an identifier to each block, and sends correction blocks alongside fresh blocks. PPR [31] uses confidence information from the PHY layer instead of a CRC to determine which bits need retransmission, and incorporates a dynamic algorithm to determine what feedback should be sent by the receiver. Maranello [24] again uses CRC to protect each block, but reduces overhead by only transmitting CRCs when the packet contains errors.

Partial packet recovery can be viewed as a finer-grained Type-I HARQ: the PPR system sends smaller blocks, retransmitting a block on error. The difference to non-PPR HARQ is the aggregation of several blocks onto a single transmission, eliminating the fate-sharing of bits in the non-PPR case.

2.1.4 Rateless Codes

The following experiments use three recently developed rateless codes—Raptor codes, Strider [20], and spinal codes [45]—so we start by summarizing the key ideas in these codes and the salient features of the implementations of these codes used in our experiments.

Raptor code. Raptor codes [50, 14], which are built on LT codes [38], achieve capacity for the Binary Erasure Channel where packets are lost with some probability. Not much is known about how close Raptor codes come to capacity for additive Gaussian noise (AWGN) channels and binary symmetric channels (BSC). However, there have been several attempts made to extend Raptor codes for the AWGN channel [44, 51, 5]. We adopt a similar construction to [44] in this evaluation, with an inner LT code generated using the degree distribution in the Raptor RFC [39], and an outer LDPC code as suggested by Shokrollahi [50].

Strider. Strider realizes the layered approach to rateless codes of Erez et al [13]. This

approach combines existing fixed-rate base codes to produce symbols in a rateless manner. By carefully selecting linear combinations of symbols generated by the base codes, they show that the resulting rateless code can achieve capacity as the number of layers increases, provided the fixed-rate base code achieves capacity at some fixed SNR. The Strider implementation in our experiments was built using reference code from Gudipati [20], with recommended parameters.

Spinal codes. Spinal codes [45] use a hash function to generate transmitted symbols. The rich structure obtained using the hash function can be exploited to achieve rates very close to the channel capacity, with a practical encoder and decoder. In addition to their ability to approach capacity, spinal codes also work well on short messages (256-1024 bits), making them appealing from a latency perspective.

2.2 Optimal Transmission Schedule

We have yet to define the decoding CDF or to show how to coordinate the sender and receiver via a *transmission schedule* derived from this CDF. This section introduces the decoding CDF and applies it to the schedule optimization problem, treating the CDF as a known quantity. We also show how to compute the feedback delay. The solution of the optimization problem proceeds from these two inputs via a dynamic programming algorithm, which we demonstrate with an example. The next section will show how to learn the CDF using feedback from the receiver.

2.2.1 Decoding CDF

We abstract a rateless code as an encoder that produces an infinite sequence of encoded symbols (bits or constellation points/channel usages) from a finite message, and a decoder that takes any prefix of this infinite sequence and returns either nothing or the correct message. Supposing that all input messages are protected equally, and that the channel parameters drawn from some unknown distribution, the behavior of the code on this channel is characterized by a single function giving the probability

with which a message can be decoded correctly after a certain number of symbols have been received by the decoder.

We assume that this probability increases monotonically with the number of symbols received; otherwise, a better (and admittedly more expensive) decoder could attempt to decode with only the first symbol, then with the first two, and so on, guaranteeing monotonicity. If every message is eventually decoded, then the function can be viewed as the cumulative distribution for a random variable that we denote n. Changes to the code parameters, channel conditions, or code block length will affect this distribution.

The decoding CDF has three desirable features from the perspective of the protocol:

1. It succinctly captures all of the uncertainties in the system, including those due to fluctuating outcomes on a stationary channel, time-varying channel parameters, and uncertainty about these parameters. Moreover, a schedule determination algorithm that relies only on the decoding CDF is insulated from the details of the code.

2. It enables the protocol to explicitly compute, and thus maximize, the expected throughput of a transmission schedule.

3. It can be learned from receiver feedback. §2.3 shows how beliefs about the decoding CDF can be updated from the number of symbols needed to decode each packet. Alternatively, the CDF can be estimated using offline simulations of the behavior of the code. Either way, the sender and receiver have common knowledge of the decoding CDF.

2.2.2 Optimization Problem

Given a decoding CDF, we would like to obtain a rule for the sender that determines when it should transmit and when it should pause for feedback. This rule takes the form of a sequence of positive integers n_i, $i \in \{1, 2, \ldots\}$. Each n_i indicates cumulatively how many symbols should be transmitted before the sender begins its i^{th} pause.

If feedback were free (i.e., took 0 time), then a throughput-maximizing sender would pause after each symbol transmission. In reality, it takes many microseconds to turn the radios from transmit to receive mode, re-synchronize, complete any ongoing decoding operations to determine whether a positive ACK should be sent, encode the ACK, and turn the radios around again to return control to the sender. The *feedback delay* is the time between the sender's last symbol prior to a pause for feedback and the same sender's first useful symbol following the pause for feedback. With 802.11a/n timings and the most reliable ACK coding rate,

$$\begin{aligned} T_{\text{feedback}} &= \text{SIFS} + \text{preamble} + \text{ACK payload} + \text{SIFS} + \text{preamble} \\ &= 64\ \mu\text{s} + 4\ \mu\text{s} \cdot \left\lceil \frac{\text{ACK bits} + 6}{24} \right\rceil \end{aligned}$$

On a 20 MHz channel with the standard guard interval, 802.11 sends 12 million symbols per second. Thus, if the sender and receiver have only one coded transmission in play at a time, the cost of a single bit of feedback could equal the cost of 816 symbols. This number can be reduced by aggregating many packets to divide the large constant cost of feedback across more useful bits. We explore the details of such aggregation in §2.4. Even for Strider, the code with the largest packet size of those we considered, transmitted frames are only on the order of 3750 symbols each, so that a pause for feedback after each frame would occupy 18% of the total time spent on transmission plus feedback.

Let

$$n_f = \frac{T_{\text{feedback}}}{\#\text{ aggregated packets}} \cdot 12\text{ million } \frac{\text{symbols}}{\text{sec}}$$

This is the amortized cost of feedback in units of foregone symbols. Note that higher layers cannot tolerate an arbitrary increase in latency caused by excessive aggregation, so one can never drive the cost of feedback to zero. Suppose that n_f already takes into account as much amortization as is possible subject to higher-layer latency constraints.

We wish to send a packet whose length before any coding is b bits, after coding it using a rateless code. The decoding CDF for the rateless code specifies $\mathbb{P}(n > \cdot)$, the probability that decoding will be successful after receiving n symbols.

Consider a general transmission schedule for reliable delivery. The sender first transmits n_1 symbols, then pauses for feedback. If decoding fails, the sender transmits $n_2 - n_1$ symbols before pausing a second time. In sum, before the i^{th} pause, the sender transmits n_i symbols. We seek an assignment of values to n_1, n_2, ... that minimizes the average time spent delivering each b-bit packet.

Let

$$p_i = \mathbb{P}(\text{first success after } i \text{ feedback rounds})$$
$$q_i = \mathbb{P}(\text{stop after } i \text{ feedback rounds}).$$

These two quantities differ if the sender gives up on this packet without success and moves on to the next one. The sender will spend an average of $\sum_i q_i(n_i + i \cdot n_f)$ symbol-times on each message, including time spent transmitting and time spent waiting for feedback.

The *efficiency* of a transmission strategy is the fraction of this time *strictly necessary* for reliable delivery. Its formal definition is motivated by two observations.

1. For any feedback-based link-layer protocol, it is essential that the sender solicit feedback at least once for each network-layer packet to ensure that it has been received, so the sender can then proceed to the next packet.

2. It is necessary for the transmitter to transmit, on average, at least $\mathbb{E}[n]$ symbols. If the transmitter sends less than this number of symbols, then it follows that some fraction of packets are not decoded correctly.

Combining these two observations, we define efficiency as

$$\eta = \frac{\mathbb{E}[n] + n_f}{\sum_i q_i(n_i + in_f)} \tag{2.1}$$

Note that if n_f is nonzero, the only way to achieve perfect efficiency will be for the sender to guess n correctly every time. If n has positive entropy arising from unpredictable channel variations, as in practice, we will be unable to achieve 100% efficiency, but our goal is to come as close as possible to the ideal efficiency.

Minimizing the time spent delivering a message is equivalent to maximizing η. In principle, this problem seems like a difficult multi-dimensional search, but it turns out that the optimal n_i assignments can be obtained by dynamic programming using only the decoding CDF as input. The optimal substructure in this problem is revealed by interpreting the transmitter's decisions in the framework of a dynamic game, as we explain next.

2.2.3 Dynamic Game Formulation

Suppose that we are playing a game against nature, and that nature chooses n from the known decoding CDF distribution but does not tell us the value. Our first move is to transmit n_1 symbols; nature's behavior is to use its hidden knowledge of n to determine whether the game ends or continues. We then transmit $n_2 - n_1$ symbols, and nature once more determines whether the game ends, and so on.

Our score at the end of the game is the negative of the total time we spent transmitting symbols or waiting for feedback (we measure the time in units of "symbol-time"). The negative is because we want the game to end as soon as possible. Because nature only acts to end the game, the optimal strategy depends only on n_f and the distribution of n (the decoding CDF), but not on any information arising during gameplay.

To develop an optimal strategy for the game, we apply backward induction. Supposing we find ourselves at some node of the decision tree where i symbols have already been transmitted, we must decide how many additional symbols j_i^* we will transmit before pausing to minimize our expected time-to-completion. For some choice

of j, the corresponding expected time is

$$t_{ij} = j + n_f + \mathbb{P}(n > i + j \,|\, n > i) t_{(i+j)j^*_{i+j}}$$

That is, we pay an immediate cost of $j + n_f$, and with some probability given by the decoding CDF and expressed in the third summand above, the game will continue and we will incur additional costs according to our optimal strategy. The third term is somewhat counter-intuitive: it says that with probability $\mathbb{P}(n > i + j \,|\, n > i)$, the additional time required is $t_{(i+j)j^*_{i+j}}$; the tricky part is that the expression of t_{ij} now depends on $t_{\ell j'}$, where the index $\ell = i + j$ is *greater than* i.

We address this issue below, but for now observe that choosing

$$\begin{aligned} j^*_i &= \arg\min_{j>0} t_{ij} \\ t^*_i &= t_{ij^*_i} \text{ so that we can write} \\ t^*_i &= j^*_i + n_f + \mathbb{P}(n > i + j^*_i \,|\, n > i) t^*_{i+j^*_i} \end{aligned} \tag{2.2}$$

produces the optimal strategy[2].

The reason is that if we know the optimal strategy and the corresponding expected time for all $i' > i$, then we can compute the strategy and expected time for i. So, if n were bounded above by some finite value, one can use dynamic programming to find the optimal strategy to minimize the expected completion time for all i. We would then choose $n_1 = j^*_0$, $n_{i+1} = n_i + j^*_{n_i}$.

2.2.4 Finite Termination by Tail Fitting

Some finesse is required to terminate the infinite recursion onto larger and larger i. The problem is that one may easily encounter a point where there is no information available from the decoding CDF for some number of symbols, n^*, required to make progress; i.e., we have no information about $\mathbb{P}(n > n^* = i + j^*)$. What RateMore does in this situation is to revert to the best possible periodic-rate schedule given all

[2]The strategy is a subgame perfect equilibrium.

the information known. It produces a schedule where the sender pauses after sending j^* symbols, obtaining the feedback, and then continuing, until either the packet is decoded or the sender gives up. If the packet gets decoded, the sender will have obtained information about the decoding CDF for this point in the state space, which will improve the subsequent operation of the protocol.

To determine j^*, we replace the tail of our distribution for n with an *analytic form*, which produces a stationary optimal strategy. Rearranging the terms in Equation (2.2) and replacing the t's with a fixed (stationary) value on both sides, we find

$$\begin{aligned} t^* &= j^* + n_f + \mathbb{P}(n > i + j^* \,|\, n > i)t^* \\ 1 - \frac{j^* + n_f}{t^*} &= \mathbb{P}(n > i + j^* \,|\, n > i) \\ &= \frac{\mathbb{P}(n > i + j^*)}{\mathbb{P}(n > i)} \end{aligned}$$

The memoryless (geometric) distribution satisfies this property. We fit a geometric tail onto our distribution and compute j^* and t^* for it to bootstrap the recursion onto finite i.

We could instead have set the complementary CDF to be zero above some finite n, but this introduces undesirable oscillations into the computed expected time and strategy variables, and does not offer any insight into what the transmitter should do if it does eventually find itself in the position of choosing a j for some i larger than this limit. With the geometric tail, the transmitter simply falls back to sending the stationary number j^* of symbols before each pause. This distinction is especially important during the process of learning the empirical CCDF, because the sender will have no data for the probability of $n > n_{\max}$, where $n_{\max}$ is the largest n we have observed so far.

For the memoryless distribution with geometric parameter $0 < \beta < 1$, we solve to

obtain

$$\begin{aligned} \beta^{-j} + j \log \beta &= 1 - n_f \log \beta \\ \Rightarrow k &= \ln(k + \gamma), \text{ where} \\ k &\triangleq -j \log \beta \\ \gamma &\triangleq 1 - n_f \log \beta \end{aligned}$$

Solving iteratively,

$$k_1 = \gamma \qquad\qquad k_{i+1} = \ln(k_i + \gamma)$$

$$j^*_{\text{tail}} = -\frac{k_\infty}{\log \beta} \qquad\qquad t^*_{\text{tail}} = \frac{j^* + n_f}{1 - \beta^{j^*}}$$

In practice, the iteration converges rapidly.

Another useful special case of the dynamic programming algorithm is for $n = c + g$ where c is a constant and g is geometrically distributed with parameter β. In this case, we reuse the above calculation for j^*_{tail}, but take $n_i = c + i \cdot j^*_{\text{tail}}$. That is, the first step is to send $c + j^*_{\text{tail}}$ symbols, and each subsequent step is to send j^*_{tail} symbols. This form for the distribution of n turns out to be a reasonable practical approximation to the empirical behaviors of the spinal code and Strider.

2.2.5 An Example

This section will use Figure 2-1 to illustrate the behavior of the dynamic programming strategy using a synthetic decoding CDF for purposes of illustration. We proceed top-to-bottom, then left-to-right. The top-left plot shows a CDF and when we would pause for feedback with various delays, n_f. The mapping of colors[3] to feedback costs is shown on the right axis. The middle-left and bottom-left plots show the estimated remaining time and optimal forward strategy for the same CDF after some number

[3]Unfortunately, this chart is best viewed in color; we recognize that it is a problem when not viewed online or on a color print-out; note, however, that the different "levels" correspond to different feedback costs.

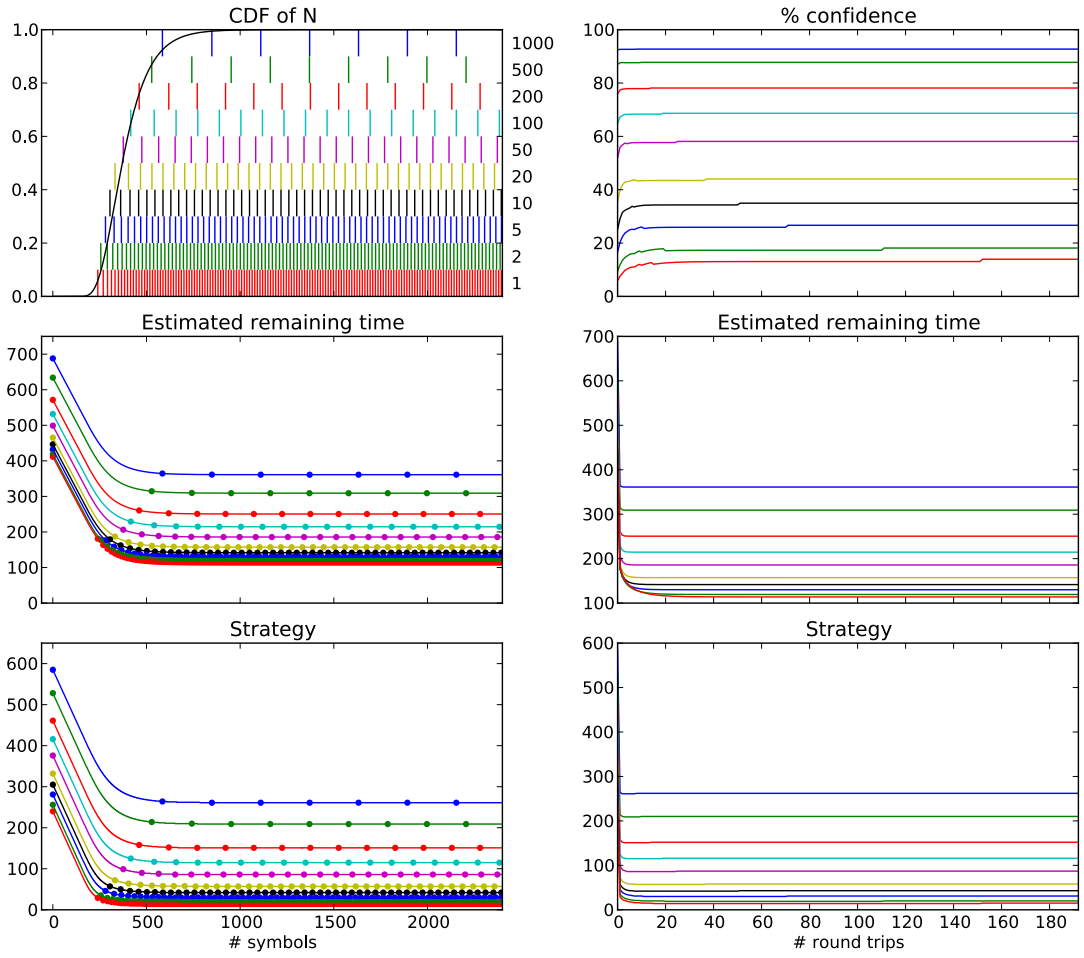


Figure 2-1: An illustration of the output of the dynamic programming algorithm, explained in §2.2.5.

of symbols have been transmitted. The circles show the best division into feedback intervals, starting at zero and stepping forward according to the bottom-left plot.

The top-right plot shows how the optimal strategy chooses a variable level of pre-feedback confidence of decoding. For the constant-plus-geometric distribution, for example, the lines would be flat. The other two plots on the right column show how tail-fitting gives us reasonable behavior even after dozens of round-trips (which might be reasonable if the cost of feedback is low enough).

2.3 Learning the Decoding CDF

The decoding CDF is the primary input to the dynamic programming algorithm, specifying the probability that a given number of symbols will be required by the receiver for successful decoding. These probabilities depend on current channel conditions. When the channel can be characterized by a single parameter, we can obtain CDFs for different values of that parameter using off-line simulations or experiments. For example, in the case of the additive white Gaussian noise channel (AWGN) we could obtain CDFs for a range of signal-to-noise ratio (SNR) values. In practice, however, wireless radios operate in complex environments, and we expect the channel to have too many parameters for such an approach to be practical; we expect these parameters to vary unpredictably over time; and we expect that off-line simulations will differ significantly from the actual implemented system.

A robust alternative to this approach is to *learn* the CDF on the fly: that is, to estimate the CDF based on the recent history of number of symbols required for successful decoding at the receiver. The sender always consults the strategy derived from the most recent CDF estimate. This form of online learning directly accommodates variations in channel conditions due to fading and mobility.

The most general empirical distribution for the probability of successful decoding after any number number of symbols is the multinomial distribution. Thus, a very general Bayesian approach would be to learn this multinomial distribution beginning from a Dirichlet prior. This entails maintaining a histogram over the number of symbols required for decoding so far. While straightforward, a model with such a large state-space leads to slow learning and slow adaptation to variations in the channel.

Ideally, we would like to maintain a parametric distribution with a minimal number of parameters as our surrogate for the actual CDF. With this in mind, we propose two approaches: (a) Gaussian approximation, and (b) Constant-plus-Geometric approximation. From these, we pick the Gaussian approximation, for the reasons explained below.

Gaussian approximation. Our inspection of the decoding CDFs indicates that the

Gaussian distribution should be a reasonable approximation at low SNR. Maximum-likelihood (ML) estimation for the Gaussian distribution requires nothing more than computing the empirical mean and variance of the number of symbols needed for successful decoding, which can be accomplished with a few accumulators. In the face of time-varying conditions, the empirical mean and variance can be *filtered* using a moving average or a similar scheme.

We chose to use Algorithm 1 for learning a Gaussian CDF. The algorithm keeps exponentially-weighted accumulators to estimate the mean, μ, and variance, σ^2. The parameter α ranges from 0 (no memory) to 1 (unlimited memory). The averages track the input with a time constant of $1/\ln(1/\alpha)$. This scheme has two advantages over a traditional exponentially-weighted moving average of the form $y \leftarrow (1-\alpha)x + \alpha y$. First, the start-up transient dies out more quickly because we weight our initial conditions less heavily. Second, the estimator's behavior is well-defined for $\alpha = 1$: inputs are retained forever, and we recover the ML estimator.

Algorithm 1 Gaussian learning with exponentially weighted moving average. Long-term performance is not sensitive to initial values.

function `init`():
  `samples` $\leftarrow 1$
  `sum` $\leftarrow 100$ (for instance)
  `sumsq` $\leftarrow$ `sum`$^2 + 10^2$ (for instance)
function `learn`(`sample`, α):
  `samples` $\leftarrow$ `samples` $\cdot\, \alpha + 1$
  `sum` $\leftarrow$ `sum` $\cdot\, \alpha +$ `sample`
  `sumsq` $\leftarrow$ `sumsq` $\cdot\, \alpha +$ `sample`^{2}
function `get_ccdf`(`x`):
  $\mu =$ `sum`/`samples`
  $\sigma^2 =$ `sumsq`/`samples` $- \mu^2$
  return `normal_ccdf`(μ, σ^2, `x`)

We present results in §3 showing that under the Gaussian approximation, performance of RateMore when the true decoding CDF is not available differs by only a few percent from performance when the true CDF is available. At low SNR, the Gaussian approximation with a memory parameter of $\alpha = 0.99$ performs within 1% of the known-CDF case. We also show that very aggressive learning rates of $\alpha = 0.80$

perform very well on simulated fading channels.

Constant-plus-Geometric approximation. The Constant-plus-Geometric approximation treats the random variable n as $n = c + g$, where c is a constant and g is a geometric random variable with parameter β. Like the Gaussian, this family of distributions has two parameters, c and β. We considered this approximation because of the analogy with the presence of *error exponents* in most good codes. When a code has an error exponent, the probability of a decoding error after transmitting n symbols scales as $\exp(-\gamma n)$ for some constant γ as long as n is large enough to push the rate below the channel capacity. This minimal n corresponds to the constant summand c, and the exponential drop in error probability with increasing n corresponds to the geometric drop in probability of unsuccessful decoding with increasing g.

Additionally, if this approximation is good, the the resulting dynamic programming strategy has a simple form: for the first transmission, send a number of symbols $c + j^*_{\text{tail}}$, then pause for feedback. If the receiver has not decoded the transmission, send j^*_{tail} more symbols before each subsequent request for feedback.

Unfortunately, despite this elegance, this distribution does not have a convenient maximum-likelihood estimator. The most likely value of the constant c is upper bounded by the min of n observed over all decoding attempts. In practice, this will lead the dynamic programming algorithm to under-estimate the number of symbols necessary for decoding. For this reason, we performed our experiments using the Gaussian approximation.

2.4 Packing Packets and Block ACKs

Rateless codes, particularly in high bandwidth situations, benefit from a method to pack symbols belonging to multiple distinct network-layer packets into the same link-layer frame, and using a "block ACK" scheme (as in 802.11e/n) to amortize the cost of sending feedback about the decoding state of each packet. Packing packets into a single frame is a useful amortization not only when the packets are small in size, but also when only a few more symbols are required to be transmitted to successfully

decode the packet. As link rates increase, it is likely that any code (whether rateless or using incremental redundancy) that decodes using symbols from previous frame transmissions will benefit from packing multiple distinct packets into the same frame and using block ACKs.

If per-packet latency were not a concern, we could pack as many packets as we wish into one frame, making n_f as small as we wish. In practice, however, we care about this metric: for example, if we consider a high-definition video/audio teleconference, the maximum wireless per-packet latency might be set to 50 ms, which at a bit rate of 1.5 Mbits/s works out to $n_f =$. RateMore's dynamic programming method assumes that n_f is exogenously given by such latency requirements, and that it has already been amortized over multiple in-flight packets packed into one frame.

To efficiently pack and unpack symbols from multiple packets into a single transmitted frame, we observe that because the sender's decoding CDF is learned from the receiver's feedback, both the sender and receiver can agree on the exact CDF and hence the exact feedback schedule currently in use. The sender encodes the length of the packet (in symbols) in the Signal field as in 802.11, which is transmitted inside a 4 µs OFDM symbol. From this field, the receiver of a packet knows immediately how many symbols it will receive in all. The problem is to allocate a certain number of symbols for packet 1, then the symbols for packet 2, and so on, as shown in Figure 2-2.

The solution to this problem is as follows. When the sender transmits the first frame, the receiver consults its copy of the feedback schedule to see how many symbols should have been included for a packet for which no symbols have previously been sent (obviously, this number would be the same for every packet). For example, the schedule may recommend sending 100 symbols before pausing for feedback. If a frame is received with length 500 symbols, the receiver infers that the sender wants to deliver 5 packets (as in Figure 2-2).

After the frame ends and the receiver finishes attempting to decode each packet, it sends an acknowledgment (ACK) frame to the sender including a short field (e.g., 6 bits) for each such packet. If the field for packet i is all ones (NAK), the packet has *not* been decoded successfully, so the sender should consult its feedback schedule (also

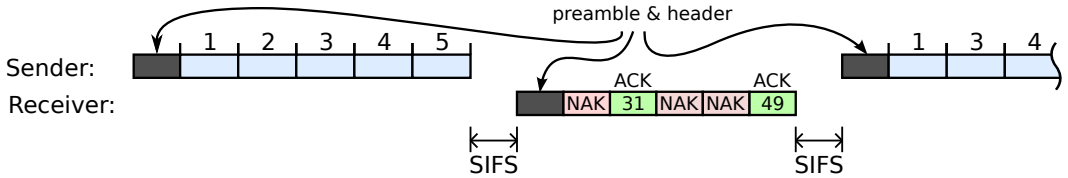


Figure 2-2: Symbols from many packets are aggregated into a frame in order to amortize the cost of the short inter-frame spaces (SIFS, 16µs) and preambles/headers.

known to the receiver) and include more symbols for that packet in the next frame. Otherwise, decoding was successful; in this case, the ACK field encodes the *fraction of this last frame's symbols for packet i that was needed to achieve a successful decoding.* Using this feedback, the sender can calculate the number of symbols that were needed to decode any given packet to estimate the decoding CDF using the learning algorithm (§2.3).

In general, because the sender and the receiver both agree on the updated CDF, the receiver will now know how many symbols for each unfinished packet will be in the next frame. For example, in Figure 2-2, three packets out of five were not decoded successfully. Each of those packets will have the *same* number of additional symbols sent in the next frame (for example, 20 symbols, according to the feedback schedule). In addition, the frame would include symbols from new packets; each of those new packets would send the same number of symbols given by the feedback schedule for new packets.

The key insight in this scheme is that we do not need to explicitly communicate the number of symbols per packet being transmitted because both parties know the schedule. This scheme works correctly if frames and ACKs are not lost, which can be engineered by ensuring that the preamble and headers are sent at low rates (one can extend this protocol to handle frame and ACK loss with some effort).

2.5 Software Implementation

Our implementation comprises several interconnected simulations written in C++ and Python.

Codes and Channel Models. To obtain raw decoding CDFs, we performed thou-

Code	Parameter	Value
Spinal	Message size	256 bits
	k	4 bits/half symbol
	B	256 candidates/beam
	Constellation	QAM-2^{20}
Strider	K	33 layers
	M	1500 bits/layer
	Message size	50490 bits
	Outer code	1/5-rate turbo
Raptor	Message size	9500 bits
	LT degree dist.	As per RFC [39]
	LDPC design	As per Shokrollahi [50]
	LDPC rate	0.95
	Constellation	QAM-64

Table 2.1: Parameters for the underlying rateless codes.

sands of experiments with the Spinal, Strider, and Raptor codes using the parameters shown in Table 2.1. The experiments produced decoding CDFs for SNRs from -5 to 35 dB on both stationary additive white Gaussian noise (AWGN) and fast-fading Rayleigh channels. The decoders had access to channel information. The slow-fading experiments used the decoding CDFs from the stationary channel, because in this regime the channel coefficients fluctuate over a time scale much longer than a packet (e.g., 45 ms for a receiver moving at 10 km/hr).

To simulate slow Rayleigh fading, we directly synthesized a Gaussian process with the appropriate Doppler spectrum and used the result as a time series of channel coefficients. Noise power was maintained at a constant fraction of the average signal power. Given the Rayleigh channel coefficient and the ratio of average signal power to noise power, we computed an appropriate instantaneous SNR. For the "ideal adaptation" results in the next section, we then selected the corresponding raw decoding CDF from our empirical data by interpolating between CDFs at the nearest SNRs. RateMore itself learns the CDFs online using the Gaussian approximation. Interestingly, despite the approximation, the system's performance is as good as using the raw CDFs.

Timing. To obtain accurate values for n_f as well as throughput and latency, we

built a timing model for transmission and feedback on a 20 MHz half-duplex OFDM channel designed to mimic 802.11a/n. We included the overhead associated with preamble synchronization, and assumed that all feedback was transmitted at the lowest 802.11a/n rate of 6 Mbps for reliable delivery. We did not include a simulation of contention, but we used the SIFS interval (16 µs) to determine how long the parties would have to wait for the ACK stage.

Because the nature of our transmission schedule is to periodically incur negative acknowledgements, we wanted to ensure that in such cases the communication proceeds to successful completion before the channel is relinquished. This avoids receivers having to maintain large buffers for several ongoing conversations. Therefore, in our simulation the transmitter resumes transmitting after SIFS elapses from the receipt of an ACK frame.

Dynamic programming algorithm. Our implementation allows for the possibility of "sparse" CDFs whose values are known only for some isolated values of the number of symbols n. For the Spinal code, we found that choosing the values of n corresponding to $1/8^{\text{th}}$-passes sacrificed virtually no performance while reducing the search space for the algorithm by a factor of roughly 8. For Strider, n should be a multiple of 3840 symbols.

The sender runs Algorithm 2 to determine how many symbols should be sent before each feedback. The points at which the CDF is known are given by x_i. The steady-state values j^*_{tail} and t^*_{tail} are computed as described in §2.2.4, and i_{tail} denotes the index into the x_i where recursion is terminated and the steady-state behavior takes over.

A value j_i returned from Algorithm 2 specifies that after a failed decode attempt with x_i symbols, the next decode attempt should occur with x_{i+j_i} symbols. A sender that gets a NAK after x_i symbols should transmit $x_{i+j_i} - x_i$ more symbols before the next feedback.

Given a feedback schedule, we use Equation (2.1) to compute the fraction η of transmission time that is well-spent, and the overhead $1 - \eta$ that is wasted.

Algorithm 2 Given decoding CDF (sampled at x_i) and n_f, compute the transmission schedule by minimizing Expected Time To Completion (ETTC).

j_i, $t_i \leftarrow j^*_{\text{tail}}$, t^*_{tail} for all $i \geq i_0$
for $i = i_{\text{tail}} - 1$ **to** 0 **do**
  # compute strategy after x_i transmitted symbols:
  for $j = 1$ **to** $i_{\text{tail}} - i$ **do**
    # when transmitting $x_{i+j} - x_i$ more symbols:
    $\text{ETTC}_j \leftarrow (x_{i+j} - x_i) + n_f + t_{i+j} \cdot \mathbb{P}(n > x_{i+j} \,|\, n > x_i)$
  end for
  # choose strategy that minimizes expected time:
  $j_i, t_i \leftarrow \arg\min\{\text{ETTC}_j\}, \min\{\text{ETTC}_j\}$
end for
return $j_0, j_1, \cdots$

ARQ and Try-after-n HARQ. We implemented two alternative feedback schedules for comparison, which we refer to as ARQ and Try-after-n HARQ. With ARQ, the sender picks some number of symbols to transmit, then waits for feedback in the form of an ACK. If it receives no ACK, it drops the packet and starts over. This scheme effectively chooses a rated version of the rateless code, and does not incorporate incremental redundancy. In principle, ARQ could choose a different rate for each SNR value. However, existing ARQ systems with rate adaptation, like 802.11, do not have such a large number of rates. 802.11n with one spatial stream has only eight. We therefore require ARQ to pick eight rates to cover the SNR range from -5 to 35 dB. These eight rates are selected to maximize efficiency over all SNRs. For the Spinal code, this maximization led us to select rates of $3.4, 6.2, 11, 19, 30, 43, 56$, and 72 Mbits/s.

For Try-after-n HARQ, we picked a family of eight parameter values, as before, and sent n additional symbols of incremental redundancy before each solicitation of feedback.

Chapter 3

Evaluating RateMore

We evaluate RateMore in simulation on stationary, slow-, and fast-fading AWGN channels. Our principal results are summarized in Table 3.1.

3.1 Baseline comparison to ARQ, Try-after-n HARQ

Figure 3-1 shows the overhead of RateMore compared to ARQ and "Try-after-n" HARQ running atop spinal codes with a feedback value $n_f = 20$. These experiments are using a stationary channel simulator that introduces Gaussian noise of different variances. We run 20 trials of 100 packets at each SNR.

Across a range of SNRs, the reduction in overhead is between 2.6× and 3.8× relative to ARQ, and between 2.8× and 5.4× relative to HARQ. These reductions are significant; they translate to link-layer throughput improvements up to 26%.

3.2 Slow-fading Rayleigh channel

Using our simulated Rayleigh fading coefficients for a Doppler velocity of 10 meters per second, we compared the throughput of RateMore running with known CDFs and SNR against the throughput of RateMore with Gaussian CDF learning. Figure 3-2 shows a typical trace of 100 milliseconds of adaptation on a fading channel. The average SNR is 15 dB. We found that long integration times (e.g. learning parameter

Fig., §	Experiment	Result
3-1, §3.1	RateMore vs. ARQ, Try-after-n HARQ	Overhead reduced by up to 5.4×; throughput increased by up to 26.6%.
3-2, 3-3, §3.2	Learning under slow Rayleigh fading	Throughput only 1.57% below case of known CDFs and SNR.
3-4, §3.3	RateMore efficiency for various n_f	Better than 88% efficiency even when $n_f = 100$: time is well spent.
3-5, §3.4	Impact of learning on throughput	RateMore's learning has an impact of only 0.25%-6%.
3-6, §3.5	Streaming with a latency requirement	Block ACK scheduling and code agnosticism enables efficient streaming.
3-7, §3.6	Streaming on a fast-fading channel	Less than 15% channel occupancy even at low bandwidth.

Table 3.1: Summary of principal results.

α close to 1) were not necessary to obtain good performance. In fact, with $\alpha = 0.8$, corresponding to an exponential time constant of only 4.5 packets, we found aggregate throughput with learning to be within 1.57% of throughput under known CDFs and SNR. Figure 3-3 is a detail of Figure 3-2.

3.3 Efficiency

Figure 3-4 shows the efficiency of RateMore (we defined this metric in Equation (2.1)) across a range of SNRs for different values of the feedback cost, n_f. These graphs are for spinal codes (the other codes show similar results). We have grouped the data according to "low", "medium", and "high" SNRs for illustration. In general, efficiency is extremely high, always over 95% at medium and high SNRs, and over 88% for low SNR values, even when n_f is as high as 100. The conclusion is that RateMore produces a good transmission schedule across different channel conditions.

3.4 How well does learning work?

Figure 3-5 shows the results of experiments that evaluate how much we lose in our *learning* the decoding CDF compared to *knowing* the true CDF. We include the

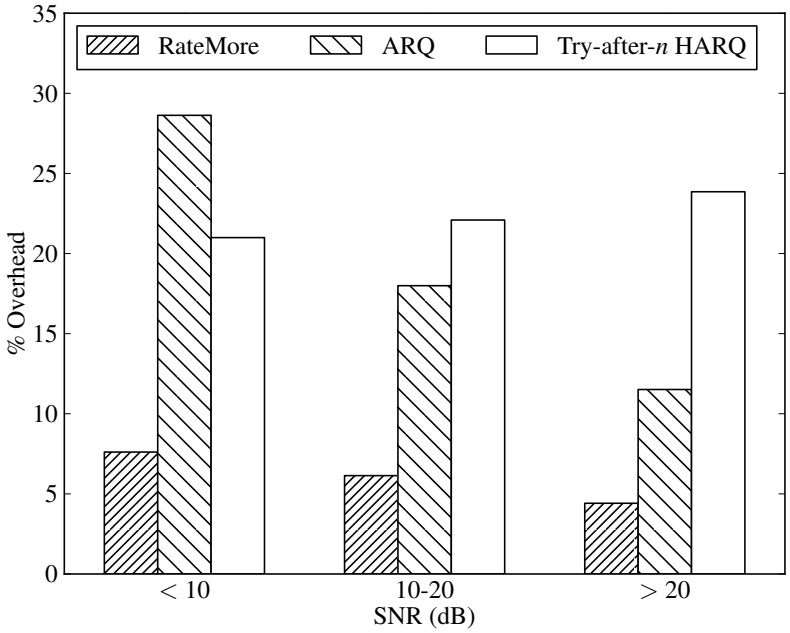


Figure 3-1: Performance with spinal codes and $n_f = 20$. At low, medium, and high SNR, respectively, RateMore reduces overhead by 3.8×, 2.9×, and 2.6× relative to ARQ and 2.8×, 3.6×, and 5.4× relative to Try-after-n HARQ.

comparison for spinal codes, but the results are similar for other codes. At low and medium SNRs, the cost of learning, i.e., how much we lose, is a negligible 2%. At higher SNR values, the cost is always less than 6%, which is still tolerable.

The conclusion is that our Gaussian approximation of the decoding CDF, which is a simple two-parameter fit (mean and variance) works extremely well, as does the simple filtering method to estimate the mean and variance.

3.5 Streaming with a low-latency requirement

For spinal codes, Strider, and Raptor, we compared overhead and channel occupation fraction using RateMore for streaming multimedia designed to emulate a Skype voice and HD video call (Figure 3-6). The main constraint here is latency: we require that the number of packets in flight, times the size of a packet, should represent no more than 100 ms of audio or audio and video. The Strider and Raptor codes we used have single-packet sizes larger than this limit for the low-rate voice call. In order to

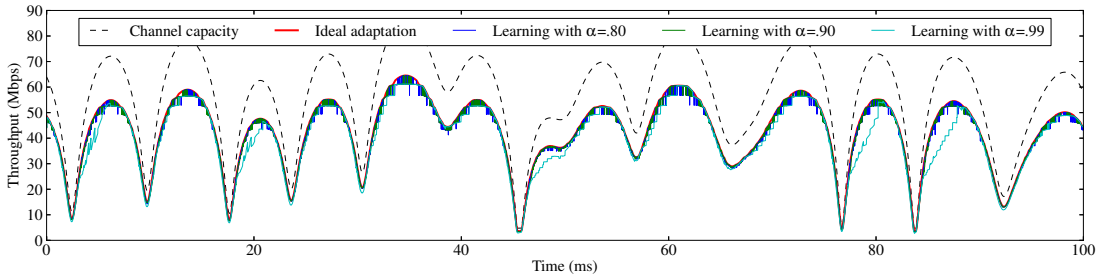


Figure 3-2: Learning performance of RateMore with spinal codes on a Rayleigh fading AWGN channel. Noise power is 15 dB below average faded signal power, giving an equivalent (in terms of capacity) stationary SNR of 12.8 dB. Doppler velocity is 10 m/s. The learning parameter α can be set to a very aggressive value of .8 for an aggregate throughput within 1.57% of the known-SNR "ideal adaptation" throughput. With $n_f = 10$.

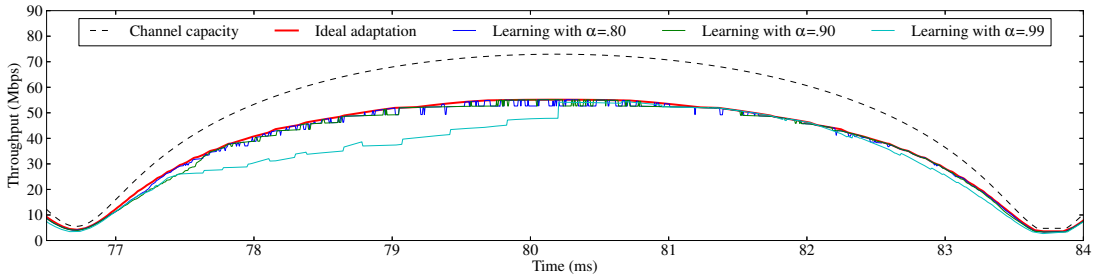


Figure 3-3: Detail of Figure 3-2.

meet the latency requirements, these codes would be forced to transmit partly-empty packets. Because RateMore is agnostic to the details of a specific rateless code, it enables a direct comparison of different rateless coding schemes with different packet sizes, including for instance the effects of relatively smaller packet sizes in spinal codes.

3.6 Streaming on a fast-fading channel

When run on CDFs generated for a fast-fading Rayleigh channel, RateMore still maintains low overhead and channel occupation (Figure 3-7).

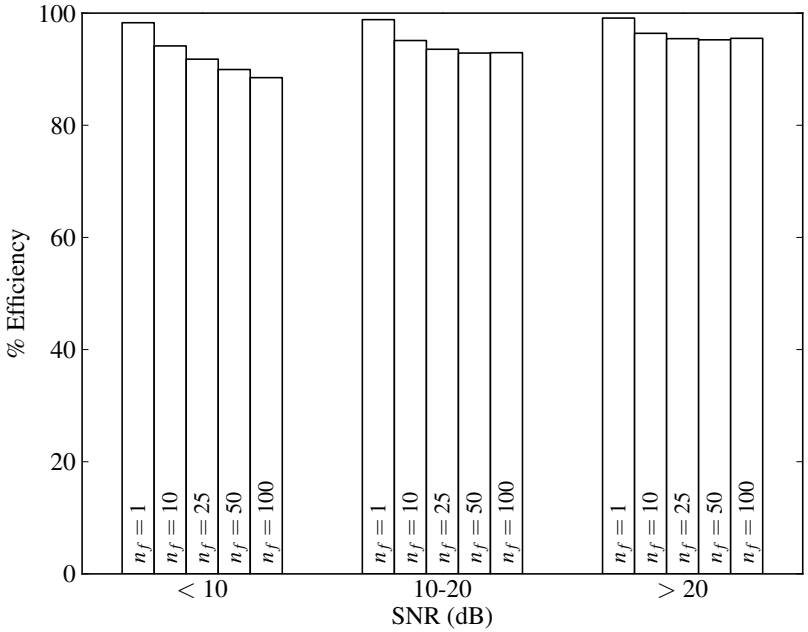


Figure 3-4: Even though n is random, RateMore makes good guesses and thus wastes very little time sending unnecessary symbols or waiting for unnecessary feedback.

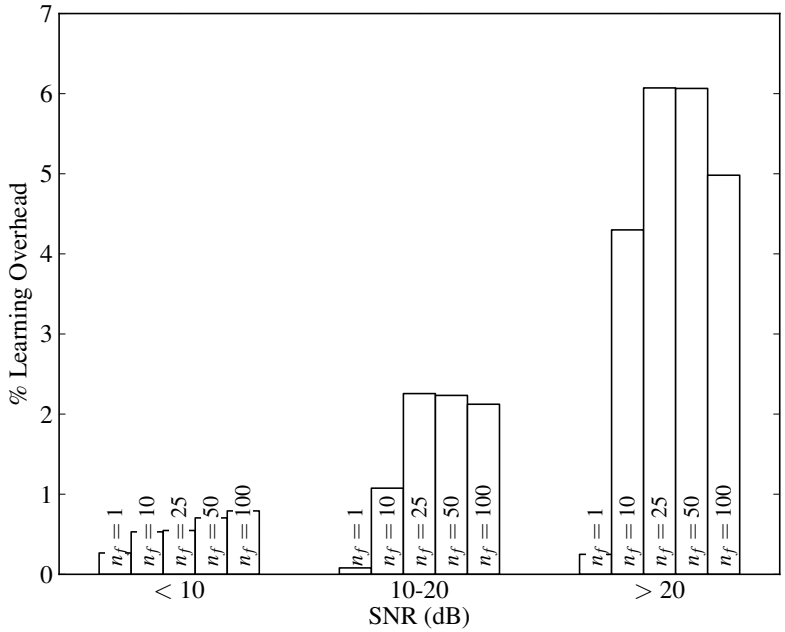


Figure 3-5: Gaussian learning performance on a stationary channel with spinal codes. Mean and variance are smoothed by an exponentially-weighted moving average with parameter $\alpha = .99$. Steady-state performance is comparable to performance when CDFs are fully known.

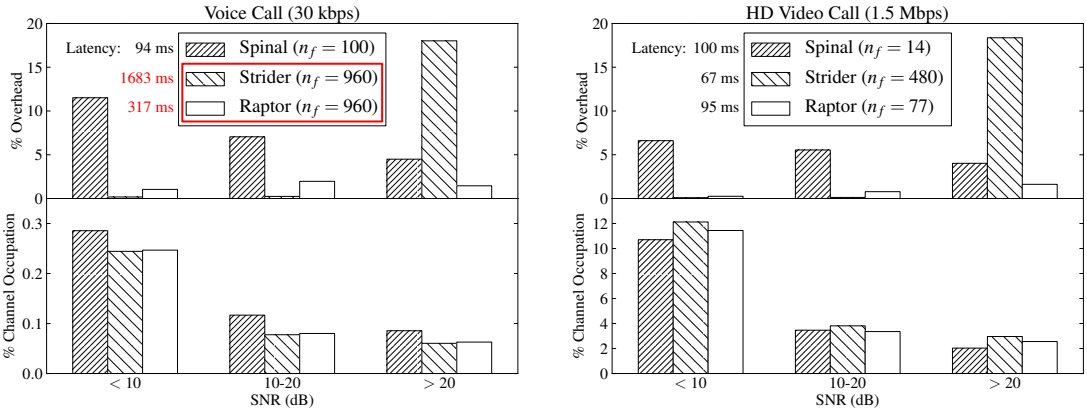


Figure 3-6: RateMore is agnostic to the details of a specific rateless code, and achieves low overhead in latency-critical environments. n_f has been normalized according to the largest number of parallel streams compatible with a given bandwidth-latency product. Cases are shown for a minimum-quality Skype voice call and for an HD video call. For an ideal user experience, latency should be less than 100 ms. Strider and Raptor require relatively large packets, which prevents the use of aggregation on the voice call – in fact, a single packet contains more than 100 ms of audio, as shown in red. In order to meet the latency target at the expense of channel occupation, packets could be fired off partly empty.

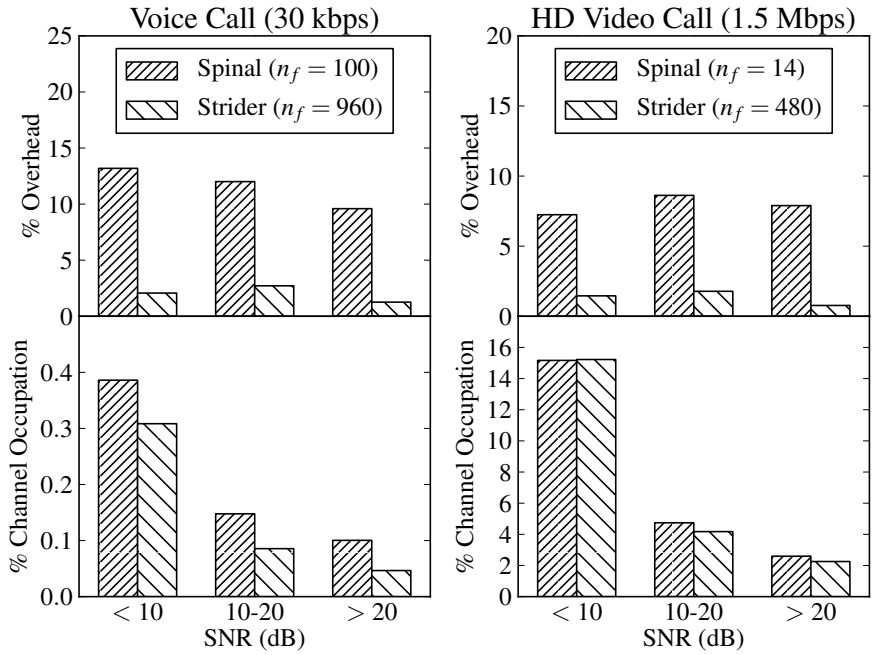


Figure 3-7: RateMore overhead and channel occupation on fast-fading Rayleigh channels. n_f has been normalized as in Figure 3-6

Chapter 4

Hardware Spinal Decoder[1]

The search for good, practical codes has a long history, starting from Shannon's fundamental results that developed the notion of *channel capacity* and established the *existence* of capacity-achieving codes. Shannon's work did not, however, show how to construct and decode practical codes, but it set the basis for decades of work on methods such as convolutional codes, low-density parity check (LDPC) codes, turbo codes, Raptor codes, and so on. Modern wireless communication networks use one or more of these codes.

In recent work, we proposed and evaluated in simulation the performance of *spinal codes*, a new family of rateless codes for wireless networks. Theoretically, spinal codes are the first rateless code with an efficient (i.e., polynomial-time) encoder and decoder that essentially achieve Shannon capacity over both the additive white Gaussian noise (AWGN) channel and the binary symmetric channel (BSC).

In practice, however, polynomial-time encoding and decoding complexity is a necessary, but hardly sufficient, condition for high throughput wireless networks. The efficacy of a high-speed channel code is highly dependent on an efficient hardware implementation. In general, the challenges include parallelizing the required computation, and reducing the storage required to manageable amounts.

This chapter and the following one present the design, implementation, and

[1]

evaluation of a hardware architecture for spinal codes. The encoder is straightforward, but the decoder is tricky. Unlike convolutional decoders, which operate on a finite trellis structure, spinal codes operate on an exponentially growing tree. The amount of exploration the decoder can afford has an effect on throughput: if a decoder computes sparingly, it will require more symbols to decode and thus achieve lower throughput. This effect is shown in Figure 4-1. A naïve decoder targeted to achieve the greatest possible coding gain would require hardware resources to store and sort upwards of a thousand tree paths per bit of data, which is beyond the realm of practicality.

Our principal contribution is a set of techniques that enable the construction of a high-fidelity hardware spinal decoder with area and throughput characteristics competitive with widely-deployed cellular error correction algorithms. These techniques include:

1. a method to select the best B states to maintain in the tree exploration at each stage, called "α-β" incremental approximate selection, and

2. a method for obtaining hints to anticipate successful or failed decoding, which permits early termination and/or feedback-driven adaptation of the decoding parameters.

We have validated our hardware design with an FPGA implementation and on-air testing. A provisional hardware synthesis suggests that a near-capacity implementation of spinal codes can achieve a throughput of 12.5 Megabits/s in a 65 nm technology while using substantially less area than competitive 3GPP turbo code implementations.

4.1 Background & Related Work

Wireless devices taking advantage of ratelessness can transmit at more aggressive rates and achieve higher throughput than devices using fixed-rate codes, which suffer a more substantial penalty in the event of a retransmission. Hybrid automatic repeat request (HARQ) protocols reduce the penalty of retransmission by puncturing a fixed-rate "mother code". These protocols typically also require the use of *ad hoc* channel quality

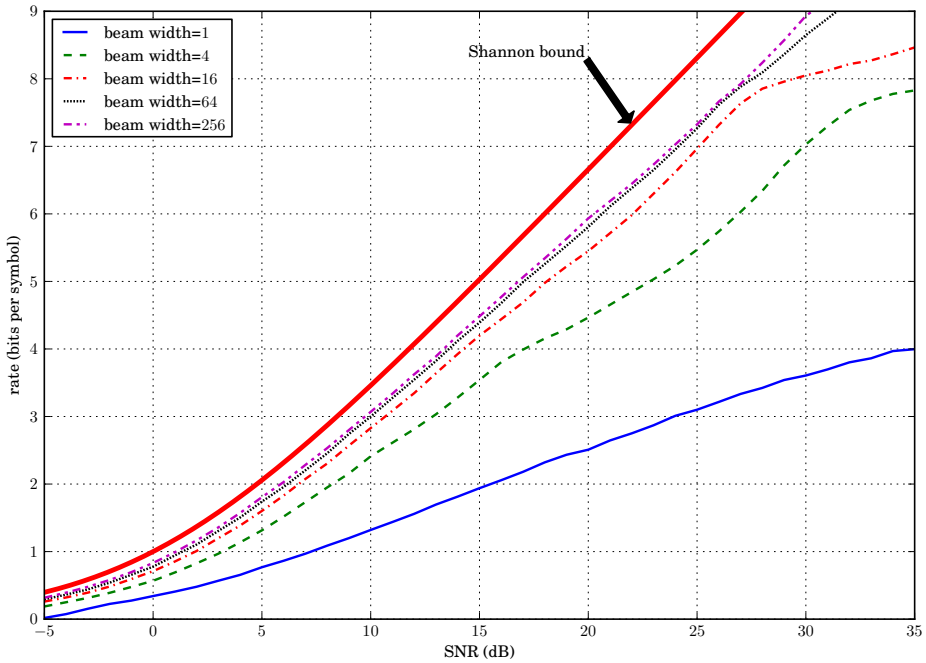


Figure 4-1: Coding efficiency achieved by the spinal decoder increases with the width of the explored portion of the tree. Hardware designs that can admit wide exploration are desirable.

indications to choose an appropriate signaling constellation, and involve a demapping step to convert I and Q values to "soft bits", which occupy a comparatively large amount of storage.

Spinal codes do not require constellation adaptation, and do not require demapping, instead operating directly on I and Q values. Spinal codes also impose no minimum rate, with encoding and decoding complexity linear in the number of symbols transmitted. They also retain the sequentiality, and hence potential for low latency, of convolutional codes while offering performance comparable to iteratively-decoded turbo and LDPC codes.

4.1.1 Spinal Codes

By way of background and for context, we review the salient details of spinal codes here [46, 45].

The principle of the spinal encoder is to produce pseudo-random bits from the message in a sequential way, then map these bits to output constellation points. As

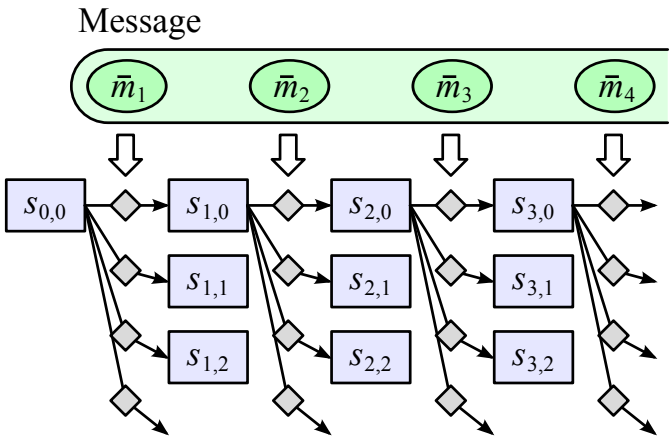


Figure 4-2: Computation of pseudo-random words $s_{i,j}$ in the encoder, with hash function application depicted by a diamond. Each $\bar{m}_i$ is k message bits.

with convolutional codes, each encoder output depends only on a prefix of the message. This enables the decoder to recover a few bits of the message at a time rather than searching the huge space of all messages.

Most of the complexity of the encoder lies in defining a suitable sequential pseudo-random generator. Most of the complexity of the decoder lies in determining the heuristically best (fastest, most reliable) way to search for the right message.

Encoder The encoder breaks the input message into k-bit pieces $\bar{m}_i$, where typically $k = 4$. These pieces are hashed together to obtain a pool of pseudo-random 32-bit words $s_{i,j}$ as shown in Figure 4-2. The initial value $s_{0,0} = 0$. Note that each hash depends on k message bits, the previous hash, and the value of j. The hash function need not be cryptographic.

Once a certain hash $s_{i,j}$ is computed, the encoder breaks it into c-bit pieces and passes each one through a *constellation map* $f(\cdot)$ to get $\lfloor 32/c \rfloor$ real, fixed-point numbers. The numbers generated from hashes $s_{i,0}$, $s_{i,1}$ ... are indexed by ℓ to form the sequence $x_{i,\ell}$.

The $x_{i,\ell}$ are reordered for transmitting so that resilience to noise will increase smoothly with the number of received constellation points. Symbols are transmitted in *passes* indexed by ℓ. Within a pass, indices i are ordered by a fixed, known permutation [46].

Decoder The algorithm for decoding spinal codes is to perform a pruned breadth-first search through the tree of possible messages. Each edge in this tree corresponds to k bits of the message, so the out-degree of each node is 2^k, and a complete path from the root to a leaf has N edges. To keep the computation small, only a fixed number B of nodes will be kept alive at a given depth in the tree. B is named after the analogy with beam search, and the list of B nodes is called the beam. At each step, we explore all of the $B \cdot 2^k$ children of these nodes and score each one according to the amount of received signal variance that remains after subtracting the corresponding encoded message. Lower scores (path metrics) are better. We then prune all but the B lowest-scoring nodes, and move on to the next k bits. With high probability, if enough passes have been received to decode the message, one of the B leaves recovered at the end will be the correct message. Just as convolutional codes can be terminated to ensure equal protection of the tail bits, spinal codes can transmit extra symbols from the end of the message to ensure that the correct message is not merely one of the B leaves, but the best one.

The decoder operates over received samples $y_{i,\ell}$ and candidate messages encoded as $\hat{x}_{i,\ell}$. Scores are sums of $(y_{i,\ell} - \hat{x}_{i,\ell})^2$. Formally, this sum is proportional to the log likelihood of the candidate message. The intuition is that the correct message will have a lower path metric in expectation than any incorrect message, and the difference will be large enough to distinguish if SNR is high or there are enough passes. "Large enough" means that fluctuations do not cause the correct message to score worse than B other messages.

To make this more concrete, consider the AWGN channel with $y = x + n$, where the noise n is independent of x. We see that $\mathrm{Var}(y) = \mathrm{Var}(x) + \mathrm{Var}(n) = P \cdot (1 + \frac{1}{\mathrm{SNR}})$, where P is the power of the received signal. If $\hat{x} = x$, then $\mathrm{Var}(y - \hat{x}) = \frac{P}{\mathrm{SNR}}$. Otherwise, $\mathrm{Var}(y - \hat{x}) = P \cdot (2 + \frac{1}{\mathrm{SNR}})$. The sum of squared differences is an estimator of this variance and discriminates between the two cases.

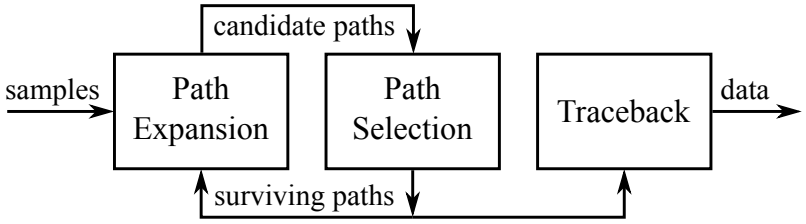


Figure 4-3: Block diagram of M-algorithm hardware.

4.1.2 Existing M-Algorithm Implementations

The decoder described above is essentially the M-algorithm (MA) [3]. A block diagram of MA is shown in Figure 4-3. In our notation, the expansion phase grows each of B paths by one edge to obtain $B \cdot 2^k$ new paths, and calculates the path metric for each one. The selection stage chooses the best B of these, and the last stage performs a Viterbi-style [42] traceback over a window of survivor paths to obtain the output bits.

There have been few recent VLSI implementations of MA, in part because modern commercial wireless error correction codes operate on a small trellis [1]. It is practical to instantiate a full Viterbi [16] or BCJR [4] decoder for such a trellis in silicon. MA is an approximation designed to reduce the cost of searching through a large trellis or a tree, and consequently it is unlikely to compete with the optimal Viterbi or BCJR decoders in performance or area for such codes. As a result, the M-algorithm is not generally commercially deployed. Existing academic implementations [19] [48] focus on implementing decoders for rate 1/2 convolutional codes.

These works recognize that the sorting network is the chief bottleneck of the system, and generally focus on various different algorithms for achieving implementations. However, these implementations deal with very small values of B and k, for instance $B = 16$ and $k = 2$, for which a complete sorting network is implementable in hardware. Spinal codes on the other hand require B and k to be much larger in order to achieve maximum performance. Much of the work in this chapter will focus on achieving high-quality decoding while minimizing the size of the sort network that must be constructed.

The M algorithm implementation in [48] leverages a degree of partial sorting among the generated $B \cdot 2^k$ nodes at the expansion stage. Although our implementation

does not use their technique, their work, to the best of our knowledge, is the first to recognize that a full sort is not necessary to achieve good performance in the M-algorithm.

The M-algorithm is also known as beam search in the AI literature. Beam search implementations do appear as part of hardware-centric systems, particularly in the speech recognition literature [35] where it is used to solve Hidden-Markov Models describing human speech. However, in AI applications, computation is typically dominated by direct sensor analysis, while beam search which appears at a high level of the system stack where throughput demands are much lower. As a result, there seems to have no attempt to create a full hardware beam search implementation in the AI community.

4.2 System Architecture

Our decoder is designed to be layered with an inner OFDM or CDMA receiver, so we are not concerned with synchronization or equalization. The decoder's inputs are the real and imaginary parts (I and Q) of the received (sub)carrier samples, in the same order that the encoder produced its outputs x_n. The first decoding step is to invert the encoder's permutation arithmetic and recover the matrix $y_{i,\ell}$ corresponding to $x_{i,\ell}$. Because of the sequential structure of the encoder, $y_{i,\ell}$ depends on $\bar{m}_{1\ldots i}$, the first ik bits of the message. Each depth in the decoding tree corresponds to an index i and some number of samples $y_{i,\ell}$.

The precise number of samples available for some i depends on the permutation and the total number of samples that have been received. In normal operation there may be anywhere from 0 to, say, 24 passes' worth of samples stored in the sample memory. The upper limit determines the size of the memory.

To compute a score for some node in the decoding tree, the decoder produces the encoded symbols $\hat{x}_{i,\ell}$ for the current i (via the hash function and constellation map) and subtracts them from $y_{i,\ell}$. The new score is the sum of these squared differences plus the score of the parent node at depth $i-1$. In order to reach the highest level of

performance shown in Figure 4-1, we need to defer pruning for as long as possible. Intuitively, this gives the central limit theorem time to operate – the more squared differences we accumulate, the more distinguishable the correct and incorrect scores will be. This requires us to keep a lot of candidates alive (ideally $B = 64$ to 256) and to explore a large number of children as quickly as possible.

There are three main implementation challenges, corresponding to the three blocks shown in Figure 4-3. The first is to calculate $B \cdot 2^k$ scores at each stage of decoding. Fortunately, these calculations have identical data dependencies, so arbitrarily many can be run in parallel. The calculation at each node depends on the hash $s_{i-1,0}$ from its parent node, a proposal $\hat{m}_i$ for the next k bits of data, and the samples $y_{i,\ell}$. We discuss optimizations of the path expansion unit in §4.4.

The second problem is to select the best B of $B \cdot 2^k$ scores to keep for the next stage of path expansion. This step is apparently an all-to-all shuffle. Worse yet, it is in the critical path, since computation at the next depth in the decoding tree cannot begin until the surviving candidates are known. In §4.3 we describe a surprisingly good approximation that relaxes the data dependencies in this step and allows us to pipeline the selection process aggressively.

The third problem is to trace back through the tree of unpruned candidates to recover the correct decoded bits. When operating close to the Shannon limit (low SNR or few passes), it is not sufficient, for instance, to put out the k bits corresponding to the best of the B candidates. Viterbi solves this problem for convolutional codes using a register-exchange approach reliant on the fixed trellis structure. Since the spinal decoding tree is irregular, we need a memory to hold data and back-track pointers. In §4.5, we show how we keep this memory small and minimize the time spent tracing back through the memory, while obtaining valuable decoding hints.

While we could imagine building $B \cdot 2^k$ path metric blocks and a selection network from $B \cdot 2^k$ inputs to B outputs, such a design is too large, occupying up to 1.2 cm^2 (for $B = 256$) in a 65 nm process. Worse, the vast majority of the device would be dark at any given time: data would be either moving through the metric units, or it would be at some stage in the selection network. Keeping all of the hardware busy

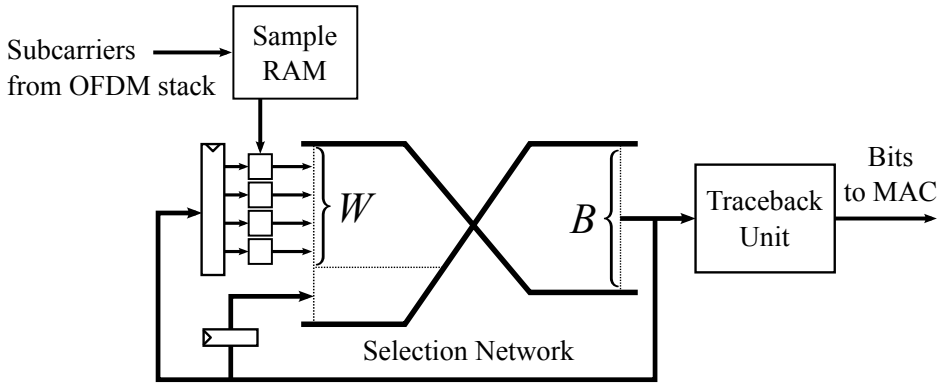


Figure 4-4: The initial decoder design with W workers and no pipelining.

would require pipelining dozens of simultaneous decodes, with a commensurate storage requirement.

4.2.1 Initial Design

The first step towards a workable design is to back away from computing all of the path metrics simultaneously. This reduces the area required for metric units and frees us from the burden of sorting $B \cdot 2^k$ items at once. Suppose that we have some number W of path metric units (informally, *workers*), and we merge their W outputs into a register holding the best B outputs so far. If we let $W = 64$, the selection network can be reduced in area by a factor of 78 and in latency by a factor of three relative to the all-at-once design, and workers also occupy 1/64 as much area. The cost is that 64 times as many cycles are needed to complete a decode. This design is depicted in Figure 4-4. The procedure for merging into the register is detailed in §4.3.1.

4.3 Path Selection

To address the problem of performing selection efficiently, we describe a series of improvements to the sort-everything-at-once baseline. We require the algorithm to be streaming, so that candidates are computed only once and storage requirements are minimal.

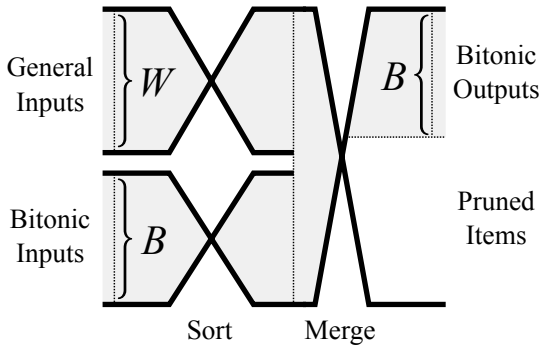


Figure 4-5: Detail of the incremental selection network.

4.3.1 Incremental Selection

This network design accepts W fresh items and B old items, and produces the B best of these, allowing candidates to be generated over multiple cycles (Figure 4-5). While the W fresh items are in arbitrary order, it is possible to take advantage of the fact that the B old items are previous outputs of the selection network, and hence can be sorted or partially sorted if we wish. In particular, if we can get independently sorted lists of the best B old candidates and the best B new candidates, we can merge the lists in a single step by reversing one list and taking the pairwise min. The result will be in bitonic order (increasing then decreasing). Sorting a bitonic list is easier than sorting a general list, allowing us to save some comparators. We register the bitonic list from the merger, and restore it to sorted order in parallel with the sorting of the W fresh items. If $W \neq B$, a few more comparators can be optimized away. We use the bitonic sort because it is regular and parametric. Irregular or non-parametric sorts are known which use fewer comparators, and can be used as drop-in replacements.

4.3.2 Pipelined Selection

The original formulation of the decoder has a long critical path, most of which is spent in the selection network. This limits the throughput of the system at high data rates, since the output of the selection network is recirculated and merged with the next set of W outputs from the metric units. This dependency means that even if we pipeline the selection, we will not improve performance unless we find another way to keep the

pipeline full.

Fortunately, candidate expansion is perfectly parallel and sorting is commutative. To achieve pipelining, we divide the $B \cdot 2^k$ candidates into α independent threads of processing. Now we can fill the selection pipeline by recirculating merged outputs for each thread independently, relaxing the data dependency. Each stage of the pipeline operates on an independent thread.

This increases the frequency of the entire system without introducing a dependency bottleneck. Registers are inserted into the pipeline at fixed intervals, for instance after every one or two comparators.

At the end of the candidate expansion, we need to eliminate $B(\alpha - 1)$ candidates. This can be done as a merge step after sorting the α threads at the cost of around $\alpha \log \alpha$ cycles of added latency. This may be acceptable if α is small or if many cycles are spent expanding candidates ($B \cdot 2^k \gg W$).

4.3.3 α-β Approximate Selection

We now have pipeline parallelism, which helps us scale throughput by increasing clock frequency. However, we have yet to consider a means of scaling the B and k parameters of the original design. An increase in k improves the maximum throughput of the design linearly while increasing the amount of computation exponentially, making this direction unattractive. For fixed k, scaling B improves decoding strength.

In order to scale B, we need to combat the scaling of sort logic, which is $\Theta(B \log B) + \Theta(W \log^2 W)$ in area and $\Theta(\max(\log B, \log^2 W))$ in latency. Selection network area can quickly become significant, as shown in Table 4.1. Fortunately, we can dodge this cost without a significant reduction in decoding strength by relaxing the selection problem.

First, we observe that if candidates are randomly assorted among threads, then on average $\beta \triangleq \frac{B}{\alpha}$ of the best B will be in each thread. Just as it is unlikely for one poker player to be dealt all the aces in a deck, it is unlikely (under random assortment) for any thread to receive significantly more than β of the B best candidates.

Thus, rather than globally selecting the B best of $B \cdot 2^k$ candidates, we can

Beam Width	8 Workers	16 Workers	32 Workers
8	14601		
16	22224	44898	
32	39772	61389	122575

Table 4.1: Area usage for various bitonic sorters in μm^2 using a 65 nm process. An 802.11g Viterbi implementation requires 120000 μm^2 in this process.

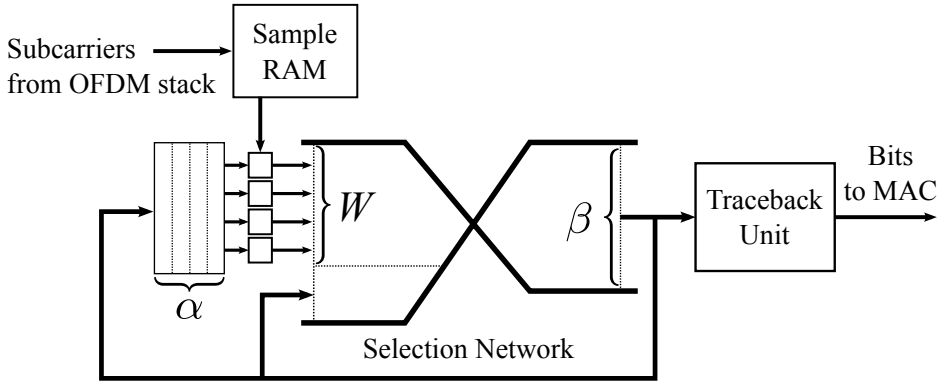


Figure 4-6: A parametric α-β spinal decoder with W workers. A shift register of depth α replaces the ordinary register in Figure 4-4. Individual stages of the selection pipeline are not shown, but the width of the network is reduced from B to $\beta = B/\alpha$.

approximate by locally selecting $\beta = \frac{B}{\alpha}$ from each thread. There are a number of compelling reasons to make this trade-off. Besides eliminating the extra merge step, it reduces the width of the selection network from B to β, since we no longer need to keep alive the B best items in each thread. This decreases area by more than a factor of α and may also improve operating frequency. We call the technique α-β selection.

The question remains whether α-β selection performs as well as B-best selection. The intuition about being dealt many aces turns out to be correct for the spinal decoder. The candidates which are improperly pruned (compared with the unmodified M-algorithm) are certainly not in the top β, and they are overwhelmingly unlikely to be in the top $B/2$. In the unlikely event that the correct candidate is pruned, the packet will fail to decode until more passes arrive. A detailed analysis is given in §4.3.6.

Figure 4-6 is a block diagram of a decoder using α-β selection. Since each candidate expands to 2^k children, the storage in the pipeline is not sufficient to hold the B

surviving candidates while their children are being generated. A shift register buffer of depth α placed at the front of the pipeline stores candidates while they await path expansion. We remark in passing that letting $\alpha = 1$, $\beta = B$ recovers the basic decoder described in §4.3.1.

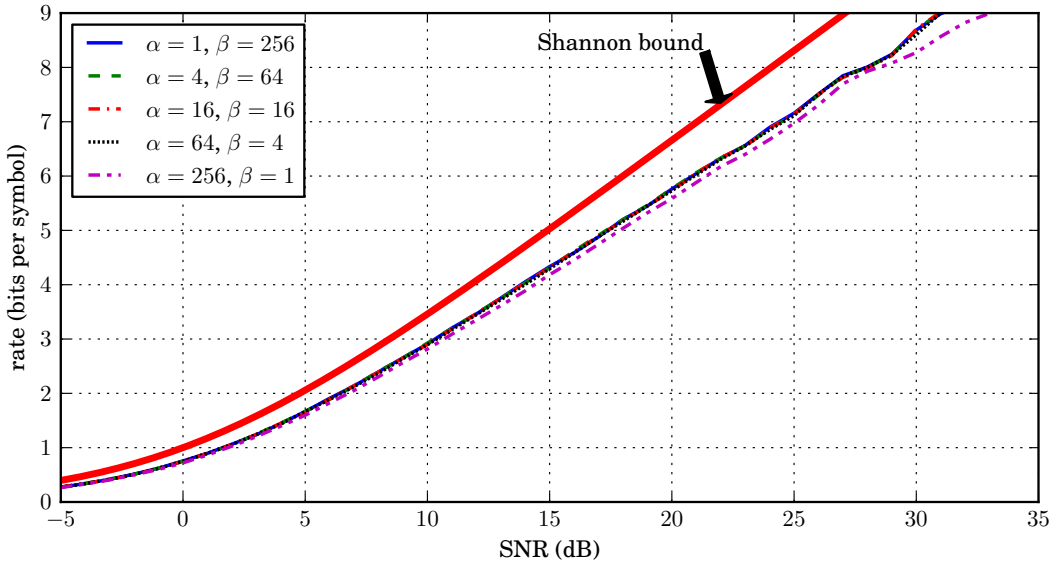
4.3.4 Deterministic α-β Selection

One caveat up to this point has been the random assortment of the candidates among threads. Our hardware is expressly designed to keep only a handful of candidates alive at any given time, and consequently such a direct randomization is not feasible. We would prefer to use local operations to achieve the same guarantees, if possible.

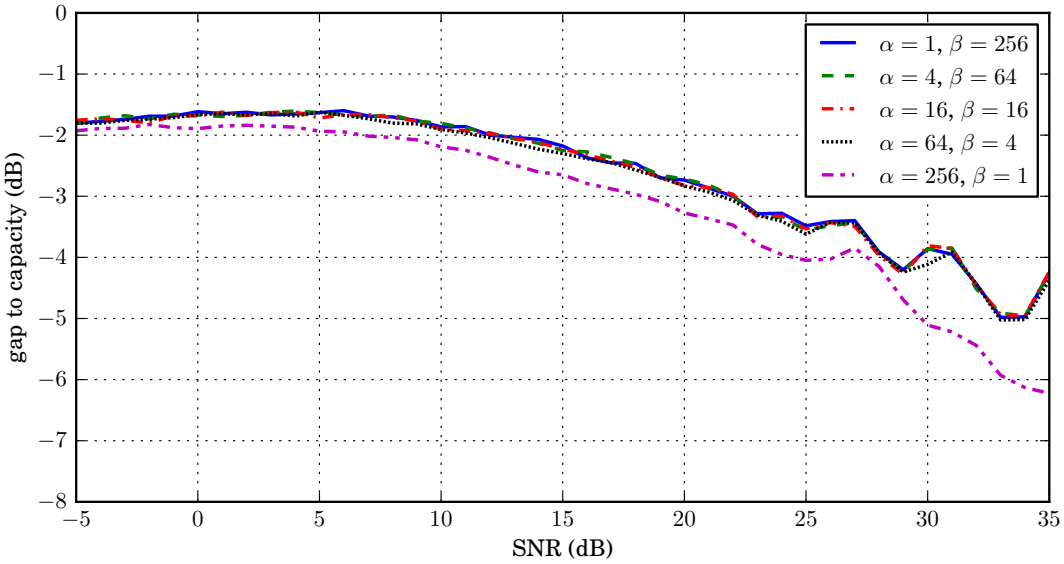
Two observations lead to an online sorting mechanism that performs as well as random assortment. The first is that descendants of a common parent have highly correlated scores. Intuitively, the goal is not to spread the candidates randomly, but to spread them *uniformly*. Consequently, we place sibling candidates in different threads. In hardware this amounts to a simple reordering of operations, and entails no additional cost.

The second observation is that we can randomize the order in which child candidates are generated from their parents by scrambling the transmitted packet. The hash function structure of the code guarantees that all symbols are identically distributed, so the scores of incorrect children are i.i.d. conditioned on the score of their parents. This guarantees that a round-robin assignment of these candidates among the threads is a uniform assignment. The children of the correct parent are not i.i.d., since one differs by being the correct child. By scrambling the packet, we ensure that the correct child is assigned to a thread uniformly. The scrambler can be a small linear feedback shift register in the MAC, as in 802.11a/g.

The performance tradeoffs for these techniques are shown in Figure 4-9. Combining the two proposed optimizations achieves performance that is slightly better than a random shuffle.



(a) Bits per Symbol



(b) Gap to Capacity

Figure 4-7: Decoder performance across α and β parameters. Even β=1 decodes with good performance.

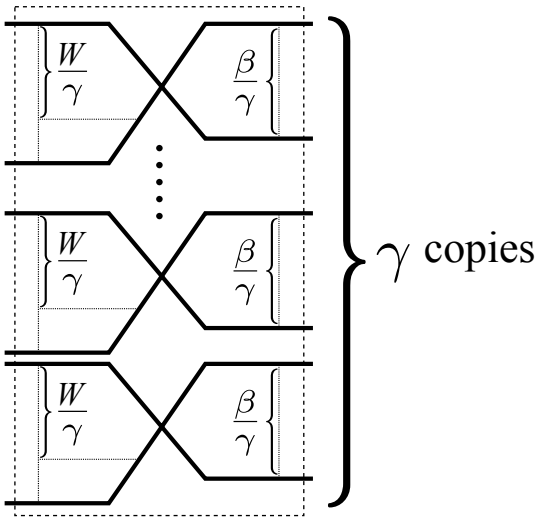


Figure 4-8: Concatenated selection network, emulating a β, W network with γ smaller selection networks.

4.3.5 Further Optimization

A further reduction of the α-β selection network is possible by concatenating multiple smaller selection networks as shown in Figure 4-8. This design has linear scaling with W. One disadvantage is that child candidates are less effectively spread among threads if a given worker only feeds into a single selection network. At the beginning of decoding, for instance, this would prevent the children of the root node from ever finding their way into the workers serving the other selection networks, since no wires cross between the selection networks or the shift registers feeding the workers. A cheap solution is to interleave the candidates between the workers and the selection networks by wiring in a rotation by $\frac{W}{2\gamma}$. This divides each node's children across two selection networks at the next stage of decoding. A more robust solution is to multiplex between rotated and non-rotated wires with an alternating schedule.

4.3.6 Analysis of α-β Selection

We consider pipelining the process of selecting the B best items out of N (i.e. $B \cdot 2^k$). Our building block is a network which takes as input W unsorted items plus β bitonically presorted items, and produces β bitonically sorted items.

Suppose that registers are inserted into the selection network to form a pipeline of

depth α. Since the output of the selection network will not be available for α clock cycles after the corresponding input, we will form the input for the selection network at each cycle as W new items plus the β outputs from α cycles ago. Cycle n only depends on cycles $n' \equiv n \pmod{\alpha}$, forming α separate threads of execution.

After N/W uses of the pipeline, all of the threads terminate, and we are left with α lists of β items. We'd like to know whether this is a good approximation to the algorithm which selects the $\alpha\beta$ best of the original N items. To show that it is, we state the following theorem.

Theorem 1. *Consider a selection algorithm that divides its N inputs among N/n threads, each of which individually returns the best β of its n inputs, for a total of $N\beta/n$ results. We compare its output to the result of an ideal selection algorithm which returns precisely the $N\beta/n$ best of its N inputs. On randomly ordered inputs, the approximate output will contain all of the best m inputs with probability at least*

$$\mathbb{P} \geq 1 - \sum_{i=1}^{m} \sum_{j=\beta}^{n} \frac{\binom{n-1}{j}\binom{N-n}{i-j-1}}{\binom{N-1}{i-1}} \tag{4.1}$$

For e.g. $N = 4096$, $n = 512$, $\beta = 32$, this gives a probability of at least $1 - 3.1 \cdot 10^{-4}$ for all of the best 128 outputs to be correct, and a probability of at least $1/2$ for all of the best 188 outputs to be correct. Empirically, the probability for the best 128 outputs to be correct is $1 - 2.4 \cdot 10^{-4}$, so the bound is tight. The empirical result also shows that the best 204 outputs are correct at least half of the time.

Proof. Suppose that the outputs are sorted from best to worst. Suppose also that the input consists of a random permutation of $(1, \ldots, N)$. For general input, we can imagine that each input has been replaced by the position at which it would appear in a list sorted from best to worst. Under this mapping, the exact selection algorithm would return precisely the list $(1, \ldots, N\beta/n)$. We can see that the best m inputs appear in the output list if and only if m is the m^{th} integer in the list. Otherwise,

some item $i \leq m$ must have been discarded by the algorithm. By the union bound,

$$\mathbb{P}(m^{\text{th}} \text{ output} \neq m) \leq \sum_{i=1}^{m} \mathbb{P}(i \text{ discarded})$$

An item i is discarded only when the thread it is assigned also finds at least β better items. So

$$\mathbb{P}(i \text{ discarded}) = \mathbb{P}(\exists \beta \text{ items} < i \text{ in same thread})$$

$$= \sum_{j=\beta}^{n} \mathbb{P}(\text{exactly } j \text{ items} < i \text{ in same thread})$$

What do we know about this thread? It was assigned a total of n items, of which one is item i. Conditional on i being assigned to the thread, the assignment of the other $n-1$ (from a pool of $N-1$ items) is still completely random. There are $i-1$ items less than i, and we want to know the probability that a certain number j are selected. That is, we want the probability of drawing exactly j colored balls in $n-1$ draws from a bucket containing $N-1$ balls, of which $i-1$ are colored. The drawing is without replacement. The result follows the hypergeometric distribution, so the number of colored balls is at least β with probability

$$\mathbb{P}(i \text{ discarded}) = \sum_{j=\beta}^{n} \frac{\binom{n-1}{j}\binom{N-n}{i-j-1}}{\binom{N-1}{i-1}}$$

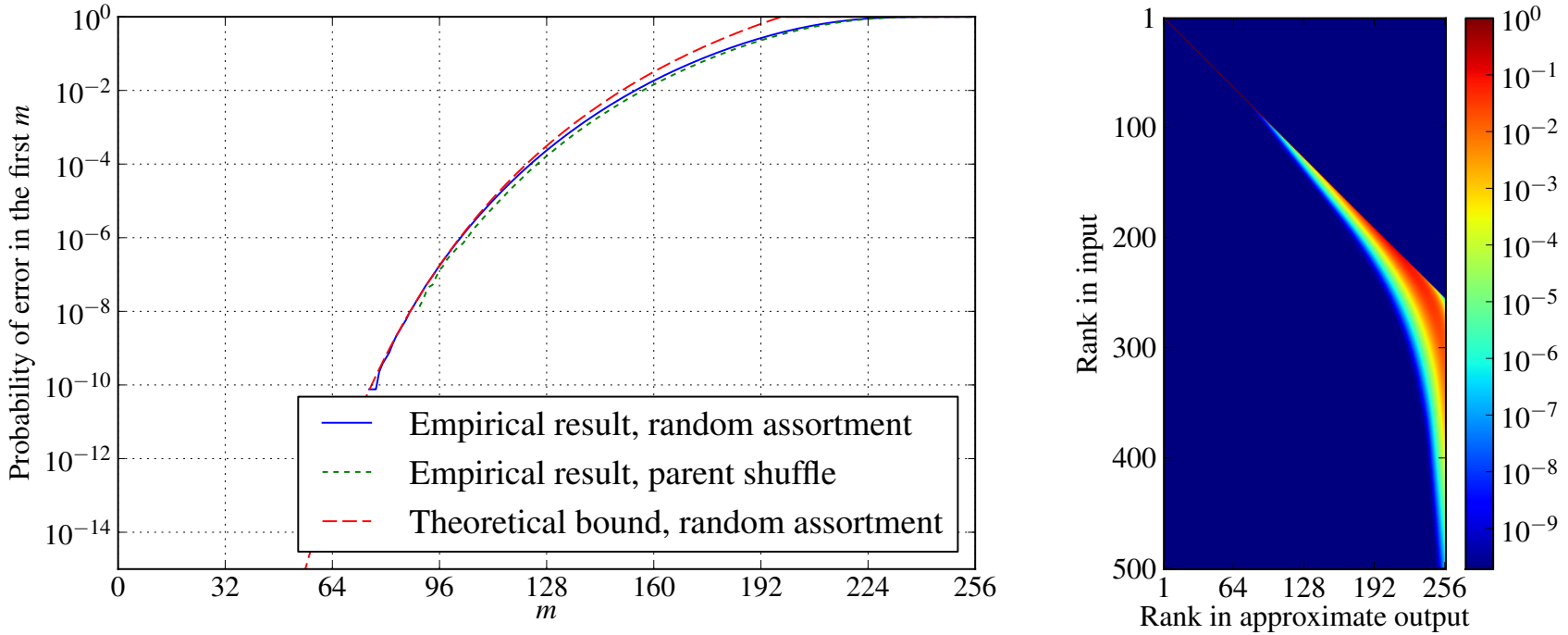
Thus, we have

$$\mathbb{P}(\text{best } m \text{ outputs correct}) =$$
$$1 - \mathbb{P}(m^{\text{th}} \text{ output} \neq m) \geq 1 - \sum_{i=1}^{m} \sum_{j=\beta}^{n} \frac{\binom{n-1}{j}\binom{N-n}{i-j-1}}{\binom{N-1}{i-1}}$$

□

Similarly, the expected number of the best m inputs which survive selection is

$$\begin{aligned}\mathbb{E}\left[\sum_{i=1}^{m}\mathbb{1}_{\{i \text{ in output}\}}\right] &= \sum_{i=1}^{m}\mathbb{P}(i \text{ in output}) \\ &= m - \sum_{i=1}^{m}\sum_{j=\beta}^{n}\frac{\binom{n-1}{j}\binom{N-n}{i-j-1}}{\binom{N-1}{i-1}}\end{aligned}$$



(a) Bound of Theorem 1 and empirical probability of losing one of the m-best inputs.

(b) Empirical log probability for each input position to appear in each output position.

Figure 4-9: Frequency of errors in α-β selection with $B \cdot 2^k = 4096$ and $\beta = 32$, $\alpha = 8$. The lower order statistics (on the left of each graph) are nearly perfect. Performance degrades towards the right as the higher order statistics of the approximate output suffer increasing numbers of discarded items. The graph on the left measures the probability mass missing from the main diagonal of the color matrix on the right. The derandomized strategy makes fewer mistakes than the fully random assortment.

4.4 Path Expansion

Thanks to the optimizations of §4.3, path metric units occupy a large part of the area of the final design. The basic worker is shown in Figure 4-10. This block encodes the symbols corresponding to k bits of data by hashing and mapping them, then computes the squared residual after subtracting them from the received samples. In the instantiation shown, the worker can handle four passes per cycle. If there are more than four passes available in memory, it will spend multiple cycles accumulating the result. By adding more hash blocks, we can handle any number of passes per cycle; however, we observe that in the case where many passes have been received and stored in memory, we are operating at low SNR and consequently low throughput. Thus, rather than accelerate decoding in the case where the channel and not the decoder is the bottleneck, we focus on accelerating decoding at high SNR, and we only instantiate one hash function per worker in favor of laying down more workers. We can get pipeline parallelism in the workers provided that we take care to pipeline the iteration control logic as well.

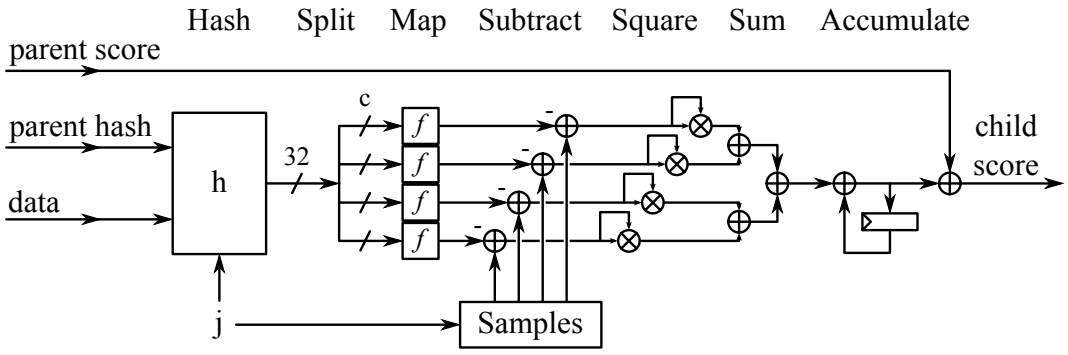


Figure 4-10: Schematic of the path metric unit. Over one or more cycles indexed by j, the unit accumulates squared differences between the received samples in memory and the symbols which would have been transmitted if this candidate were correct. Not shown is the path for returning the child hash, which is the hash for $j = 0$.

Samples in our decoder are only 8 bits, so subtraction is cheap. There are three major costs in the worker. The first is the hash function. We used the Jenkins one-at-a-time hash [32]. Using a smaller hash function is attractive from an area perspective, but hash and constellation map collisions are more likely with a weaker

hash function, degrading performance. We leave a satisfactory exploration of this space to future work.

The second major cost is squaring. The samples are 8 bits wide, giving 9 bit differences and nominally an 18 bit product. This can be reduced a little by taking the absolute value first to give $8 \times 8 \to 16$ bits, and a little further by noting that squaring has much more structure than general multiplication. Designing e.g. a Dadda tree multiplier for squaring 8 bits gives a fairly small circuit with 6 half-adders, 12 full-adders, and a 10 bit summation. By comparison, an 8×8 general Dadda multiplier would use 7 half-adders, 35 full-adders, and a 14 bit summation.

The third cost is in summing the squares. In Viterbi, the scores of all the live candidates differ by no more than the constraint length K times twice the largest possible log likelihood ratio. This is because the structure of the trellis is such that tracing back a short distance from two nodes always leads to a common ancestor. Thanks to two's complement arithmetic, it is sufficient to keep score registers that are just wide enough to hold the largest difference between two scores.

In our case, however, there is no guarantee of common ancestry, save for the argument that the lack of a recent common ancestor is a strong indication that decoding will fail (as we show in §4.5). As a consequence, scores can easily grow into the millions. We used 24 bit arithmetic for scores. We have not evaluated designs which reduce this number, but we nevertheless highlight a few known techniques from Viterbi as interesting directions for future work. First, we could take advantage of the fact that in low-SNR regimes where there are many passes and scores are large, the variance of the scores is also large. In this case, the low bits of the score may be swamped with noise and rendered essentially worthless, and we should right-shift the squares so that we accumulate only the "good" bits.

A second technique for reducing the size of the scores is to use an approximation for the x^2 function, like $|x|$ or $\min(|x|, 1)$. The resulting scores will no longer be proportional to log likelihoods, so the challenge will be to show that the decoder still performs adequately.

4.5 Online Traceback

The final stage of the decoder pipeline is traceback. Ideally, at the end of decoding, traceback starts from the most likely child, outputting the set of k bits represented by that child and recursing up the tree to that child's parents. The problem with this ideal approach is that it requires the retention of all levels of the beam search until the end of decoding. As a result, traceback is typically implemented in an online fashion wherein for each new beam, a traceback from the best candidate of c steps is performed, and k bits are output. The variable c represents the constraint length of the code, the maximum number of steps until all surviving paths converge on a single, likely ancestor. Prior to this ancestor, all paths are the same. Because only c steps of traceback need to be performed in-order to achieve this ancestor, only c beams worth of data need to be maintained. For many codes, c is actually quite short. For example, convolutional code traceback lengths are limited to 2^s, where s is the number of states in the code. In spinal codes, particularly with our selection approximations, bad paths with long convergence distances may be kept alive. However in practice, convergence occurs quite quickly in spinal codes, usually after one or two traceback steps.

Online traceback implementation are well-studied and appear in most implementations of the Viterbi algorithm. Viterbi implementations typically implement traceback using the register-exchange microarchitecture [12, 43]. However, spinal codes can have a much wider window of B live candidates at each backtrack step. Moreover, unlike convolutional codes wherein each parent may have only two children, in spinal codes, a parent may have 2^k children, which makes the wiring the register-exchange expensive. Therefore, we use the RAM-based backtrace approach [12]. Even hybrid backtrace/register-exchange architectures [8] are likely to be prohibitive in complexity. In this architecture, pointers and data values are stored in RAM and iterated over during the traceback phase. For practical choices of parameters the number of bits required is on the order of 10's of kilobits. Figure 5-1 shows empirically obtained throughput curves for various traceback lengths. Even an extremely short traceback length of four is sufficient to achieve a significant portion of channel capacity. Eight

steps represents a good tradeoff between decoding efficiency and area.

The traditional difficulty with traceback approaches is the long latency of the traceback operation itself, which must chase c pointers to generate an output. We note however, that c is a pessimistic bound on convergence. During most tracebacks, "good" paths will converge long before c. Leveraging this observation, we memoize the backtrack of the preceding generation, as suggested by Lin et al. [34]. If the packet being processed will be decoded correctly, parent and child backtracks should be similar. Figure 4-11 shows a distribution of convergence distances at varying channel conditions, confirming this intuition.

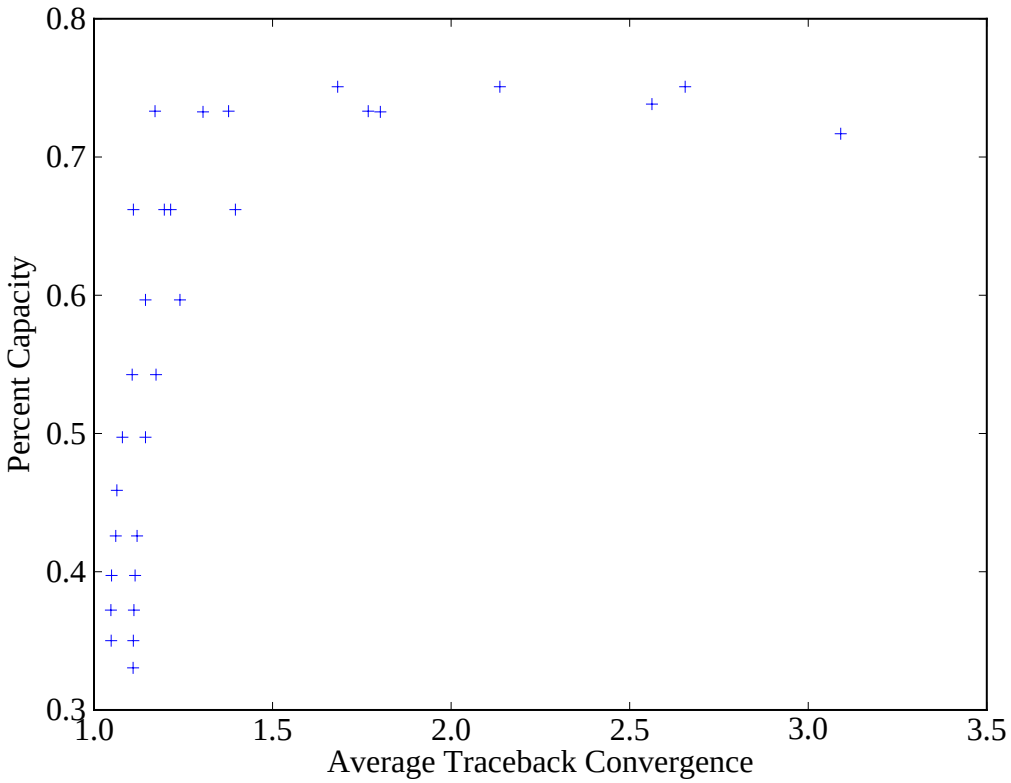


Figure 4-11: Average convergence distance between adjacent tracebacks, collected across SNR and length. Near capacity, tracebacks begin to take longer to converge.

If, during the traceback pointer chase, we encounter convergence with the memoized trace, we terminate the traceback immediately and return the memoized value. This simple optimization drastically decreases the expected traceback length, improving

throughput while simultaneously decreasing power consumption.

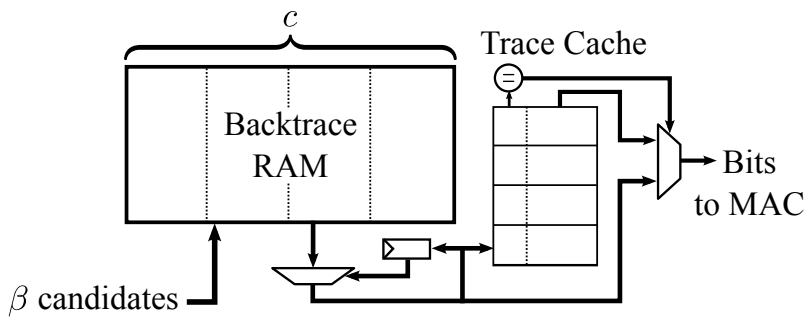


Figure 4-12: Traceback Microarchitecture. Some control paths have been eliminate to simplify the diagram.

Figure 4-12 shows the microarchitecture of our backtrace unit. The unit is divided in half around the traceback RAM. The front half handles finding starting points for traceback from among the incoming beam, while the back half conducts the traceback and outputs values. The relatively simple logic in the two halves permits them to be clocked at higher frequencies than other portions of the pipeline. Our implementation is fully parameterized, including both the parameters of the spinal code and the traceback length.

Chapter 5

Evaluating the Hardware Spinal Decoder

5.1 Hardware Platforms

We use two platforms in evaluating our hardware implementation. Wireless algorithms operate on the air, and the best way to achieve a high-fidelity evaluation is of wireless hardware is to measure its on-air performance. The first platform we use to evaluate the spinal decoder is a combination of an XUPV5 [55] and USRP2 [15]. We use the USRP2 to feed IQ samples to an Airblue [41]-based OFDM baseband implemented on the larger XUPV5 FPGA.

However, on-air operation is insufficient for characterizing and testing new wireless algorithms because over-air operation is difficult to control. Experiments are certainly not reproducible, and some experiments may not even be achievable over the air. For example, it is interesting to evaluate the behavior of spinal codes at low SNR, however the Airblue pipeline does not operate reliably at SNRs below 3 dB. Additionally, some hardware standpoint, some interesting hardware configurations may operate too slowly to make on-air operation feasible.

Therefore, we use a second platform for high-speed simulation and testing: the ACP [40]. The ACP consists of two Virtex-LX330T FPGAs socketed in to a Front-Side Bus. This platform not only offers large FPGAs, but also a low-latency, high-

	3G Turbo (1 dB)	3G Turbo (1 dB) [11]	Spinal (1 dB)	Spinal (-5 dB)
Parity RAM	118 Kb	86kB		
Systemic RAM	92 Kb	25kB		
Interleaver RAM	16 Kb			
Pipeline Buffer RAM	27 Kb	12kB		
Symbol RAM			41Kb	135Kb
Backtrace RAM			8Kb	8Kb
Total RAM	253 Kb	123 kB	49Kb	143Kb

Table 5.1: Memory Usage for turbo and spinal decoders supporting 5120 bit packets. Memory area accounts for more than 50% of turbo decoder area.

bandwidth connection to general purpose software. This makes it easy to interface a wireless channel model, which is difficult to implement in hardware, to a hardware implementation while retaining relatively high simulation performance. Most of our evaluations of the spinal hardware are carried out using this high-speed platform.

5.2 Comparison with Turbo Codes

Although spinal codes offer excellent coding performance and an attractive hardware implementation, it is important to get a feel for the properties of the spinal decoder as it compares to existing error correcting codes. Turbo codes [6] are a capacity-approaching code currently deployed in most modern cellular standards.

There are several metrics against which one might compare hardware implementations of spinal and turbo codes: implementation area, throughput, latency, and power consumption.

A fundamental different between turbo codes, and spinal codes is that the former is *iterative*, while spinal codes are streaming. This means that turbo implementation must fundamentally use more memory than a spinal implementation since turbo decoders must keep at least one pass worth of soft, extrinsic information alive at any point in time. Because packet lengths are large and soft information is wide, this extra memory can dominate implementation area. On the other hand, spinal

codes store much narrower symbol information. We therefore conjecture that turbo decoders must use at least twice the memory area of a spinal decoder with a similar noise floor. This conjecture is empirically supported by Table 5.1, which compares 3G-compliant implementations of turbo codes with spinal code decoders configured to similar parameters.

It is important to note that spinal decoder memory usage scales with the noise floor of the decoder since more passes must be buffered, while turbo codes use a constant memory area for any noise floor supported. If we reduce the supported noise floor to 1 dB from -5 dB, then the area required by the spinal implementation drops by around a factor of 4. This is attractive for short-range deployments which do not require the heavy error correction of cellular networks.

5.3 Performance of Hardware Decoder

Figure 5-1 shows the performance of the hardware decoder across a range of operational SNRs. Throughputs were calculated by running the full Airblue OFDM stack on FPGA and collecting packet error rates across thousands of packets, a conservative measure of throughput Generally the decoder performs well, achieving as much as 80% of capacity at relevant SNRs. The low SNR portion of the range is limited by Airblue's synchronization mechanisms which do not operate reliably below 3 dB.

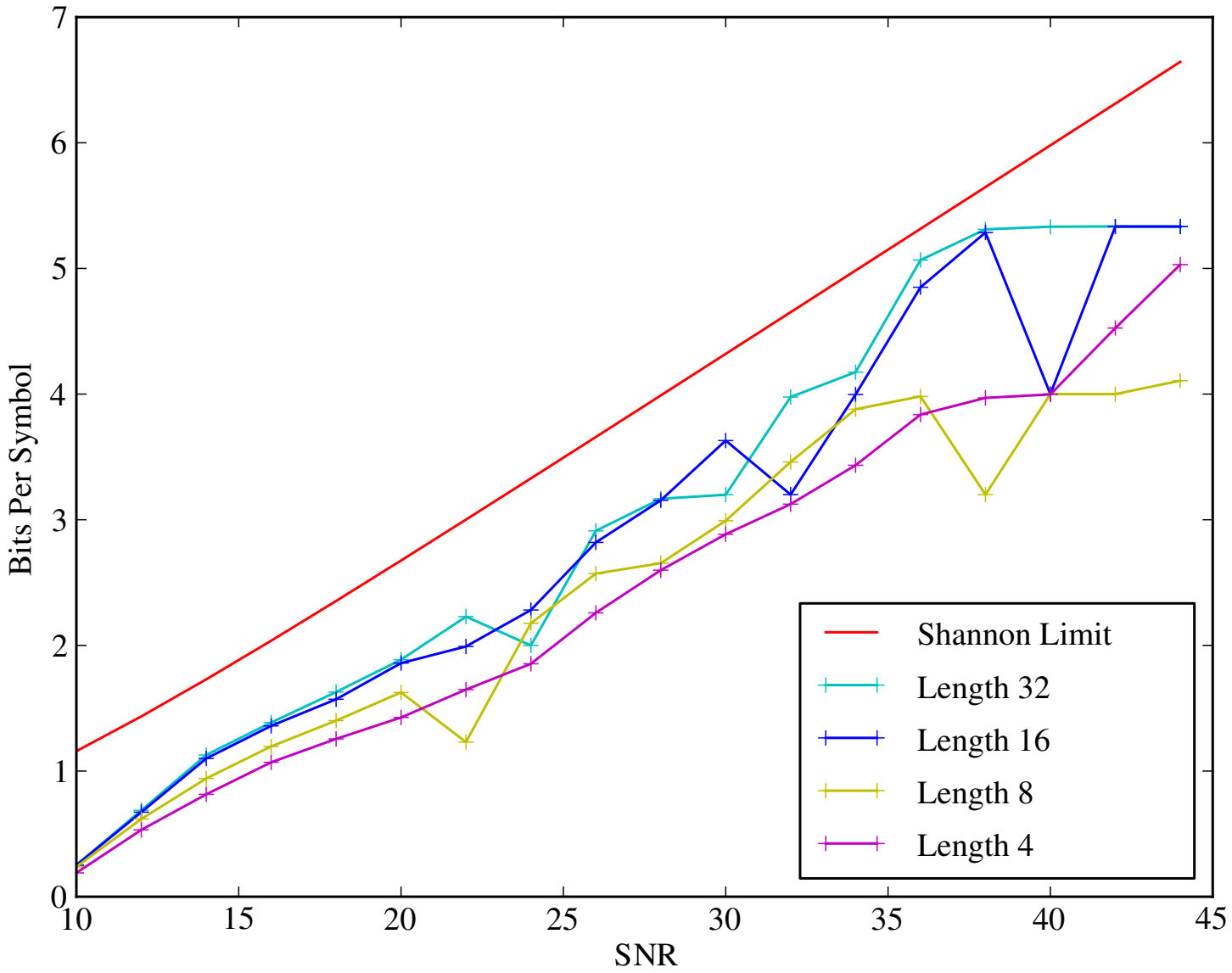


Figure 5-1: Throughput of hardware decoder with various traceback lengths.

Table 5.2 shows the implementation areas of various modules of our reference hardware decoder in a 65 nm technology. Memory area dominates the design, while logic area is attractively small. The majority of the area of the design is taken up by the score calculation logic. Individually, these elements are small. However there are β of them in our parameterized design. The α-β selection network requires one-fourth the design. In contrast, a full selection network for $B = 64$ requires around 360000 μm^2, much more than our entire decoder.

As a basis for comparison, state of the art turbo decoders [11] at the 65 nm node require approximately .3 mm^2 for the active portion of the decoder. The remaining area (also around .3 mm^2) is used for memory. Our design is significantly smaller in terms of area, using half the memory and around 80% the logic area. However, our design at 200 MHz, processes at a maximum throughput of 12.5 Mbps, which is somewhat lower than the Cheng et al., who approached 100 Mbps.

In our choice of implementation, we have attempted to achieve maximum decoding efficiency and minimum gap-to-capacity. However, maximum efficiency may not yield the highest throughput design. Should throughput be a priority, we note that there are several ways in which we could improve the throughput of our design. The most obvious direction is reducing B to 32 or 16. These decoders suffer slightly degraded performance, but operate 2 and 4 times faster. Figure 5-2 shows an extreme case of this optimization with $B = 4$. This design has low decoder efficiency, but much higher throughput. We note that a dynamic reduction in B can be achieved with relatively simple modifications to our hardware. A second means of improvement is optimizing the score calculators. There are three ways to achieve this goal. First, we can increase the number of score calculators. This is slightly unattractive because it also requires scaling in the sorting network. Second, the critical path of our design runs through the worker units and is largely unpipelined. Cutting this path should increase achievable clock period by at least a few nano-seconds. Related to the critical path is the fact that we calculate error metrics using Euclidean distance, which requires multiplication. Strength reduction to absolute difference has worked well in Viterbi and should apply to spinal as well. By combining these techniques it should be possible to build spinal

Module	Total (μm^2)	Combinational (μm^2)	Sequential (μm^2)	RAM (Kbits)
Selection Network	60700	25095	35907	
Backtrack	8575	3844	4720	8
Score Calculator	10640	8759	1881	
SampleRAM	5206	2592	2613	41
Total	245526	181890	63703	49

Table 5.2: Area usage for modules with $B = 64$, $W = \beta = 16$, $\alpha = 4$. Area estimates were produced using Cadence Encounter with a 65 nm process, targeting 200 MHz operating frequency. Area estimates do not include memory area.

decoders with throughputs greater than 100 Mbps.

5.4 On-Air Validation

The majority of the performance results presented in this chapter were generated via simulation, either using an idealized, floating-point C++ model of the hardware or using an emulated version of the decoder RTL on an FPGA with a software channel model. Although we have taken care to accurate model both hardware and wireless channel, it is important to validate the simulation results with on-air testing.

Figure 5-2 show a preliminary on-air throughput curve obtained by using the previously described USRP set-up plotted against an identically parameterized C++ model. The performance differential between hardware and software across a wide range of operating conditions is minimal, suggesting that our simulation-based results have high fidelity.

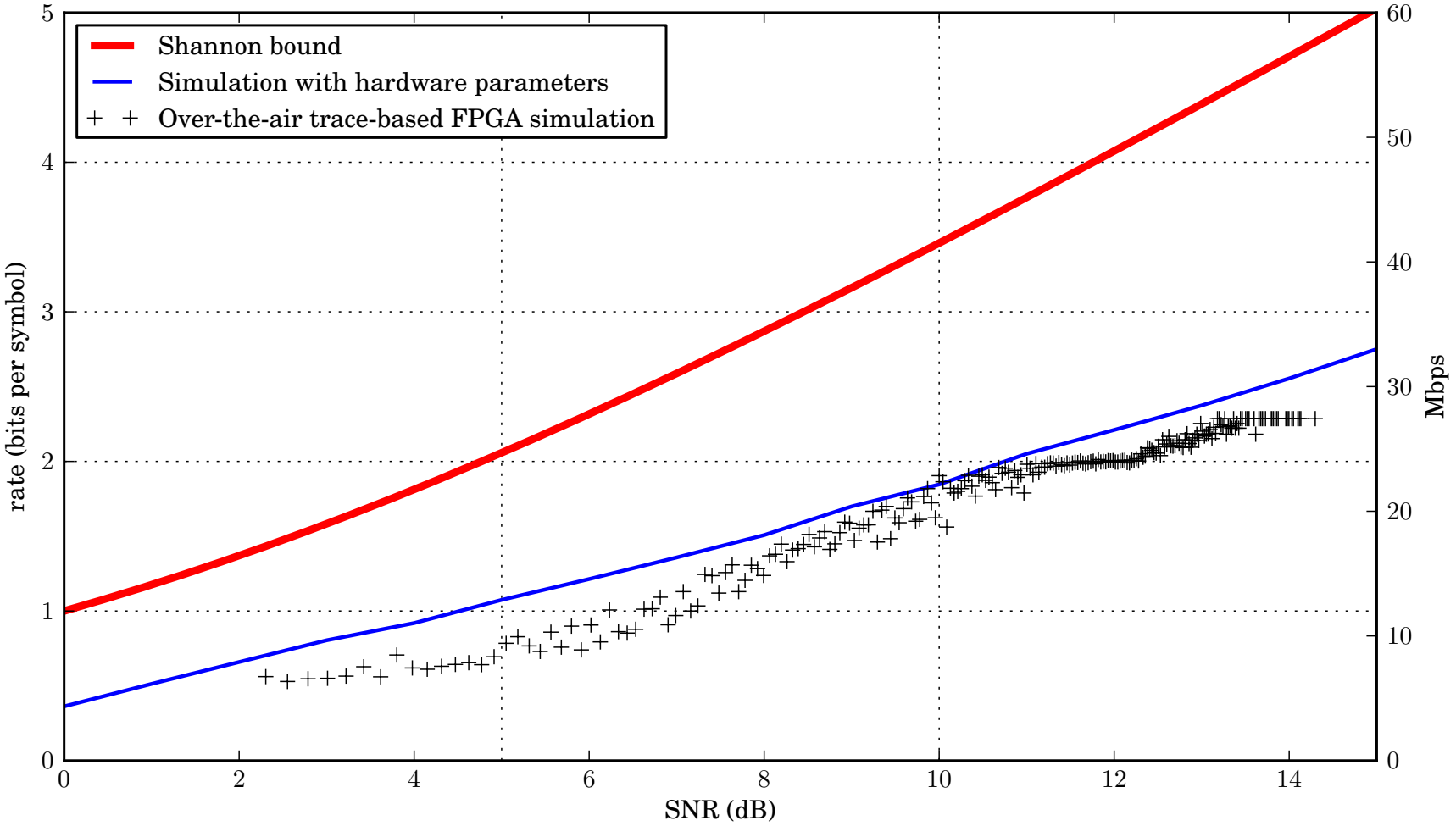


Figure 5-2: Performance of $B = 4$, $\beta = 4$, $\alpha = 1$ decoder over the air versus identically parameterized C++ model. Low code efficiency is due to the narrow width of the decoder, which yields a high throughput implementation when the number of workers is limited. A decoder with a larger beamwidth would achieve higher throughput on any given decode attempt, but it would also require more time between packets to complete decoding.

5.5 Integrating with Higher Layers

Error correction codes do not exist in isolation, but as part of a complete protocol. Good protocols require feedback from the physical layer, including the error correction block, to make good operational choices. Additionally, the spinal decoder itself requires a degree of control to decide when to attempt a decode when operating ratelessly. Decoding too early results in increased latency due to failed decoding, while decoding too late wastes channel bandwidth. It is therefore important to have mechanisms in the decoder, like SoftPHY [30], which can provide fine-grained information about the success of decoding.

Traceback convergence in spinal codes, which bears a strong resemblance to confidence calculation in SOVA [23], is an excellent candidate for this role. As Figure 4-11 shows, a sharp increase in convergence length suggests being near or over capacity. By monitoring the traceback cache for long convergences using a simple filter, the hardware can terminate decodes that are likely to be incorrect early in processing, preventing significant time waste. Moreover, propagating information about when convergences begin to narrow gives upper layers an excellent measure of channel capacity which can be used to improve overall system performance.

Chapter 6

Conclusion and Future Work

6.1 Link-layer Protocols

Practical rateless codes for wireless networks possessing the adaptation-free and computation-variable properties are an exciting recent development. They promise a better way to deal with the fundamental problem of time-varying channel conditions caused by mobility and interference. Chapters 2 and 3 presented the case for a new approach to link-layer protocols motivated by the lack of an obvious point to pause a half-duplex rateless sender to ask for feedback. We showed how the decoding CDF provides a code-independent way to encapsulate the essential information of the underlying code, and developed RateMore, which combines a dynamic programming strategy to compute the transmission schedule, a simple learning method to estimate the decoding CDF, and a block ACK protocol to amortize the cost of feedback.

Our results show that RateMore reduces overhead by between 2.6× and 3.9× compared to 802.11-style ARQ and between 2.8× and 5.4× compared to 3GPP-style "Try-after-n" HARQ, which are to our knowledge the best existing deployed approaches. We demonstrated the reduced overhead of RateMore in experiments using three different rateless codes (Raptor, Strider, and Spinal codes). These significant reductions in overhead translate to substantial throughput improvements. For example, for spinal codes, the throughput improvement is as high as 26% compared to both ARQ and "Try-after-n" HARQ.

We believe that RateMore provides a practically useful link-layer protocol to coordinate between sender and receiver in wireless networks using rateless codes, improving performance when channel conditions are variable. The most significant limitation of this protocol, and the most substantial avenue for future work in our view, concerns its reliance on reliable delivery of acknowledgement frames to synchronize explicit state updates. We would like to see an evolution of this protocol that coordinates the sender and receiver inferentially under the assumption that any frame may fail to decode. How can we build the ultimate robust signalling, synchronization, and medium access control scheme while still maximizing throughput? How does this scheme generalize to the multi-user setting, and can it be made to maximize fairness as well?

Another interesting direction is to adapt RateMore for use with adaptation-unfree codes: those requiring the choice of a signalling constellation for each transmission. With a fixed set of modulations and within a predetermined range of signal-to-noise ratios, it is quite possible for a rated code + HARQ system to exhibit characteristics of ratelessness: the amount of redundancy adjusts to accommodate channel variation, and decoding stops as soon as there is enough redundancy for it to succeed. LTE is a prime example of this, though its adaptation scheme is encumbered by a tangle of vendor-specific channel quality estimation schemes. An extension of this work to adaptation-unfree codes like LTE turbo codes would need to expand the protocol's state space to account for differences in the usefulness of each constellation to the demapper under various channel conditions.

6.2 Spinal Codes in Hardware

Spinal codes are, in theory and simulation, a promising new capacity-achieving code. In chapters 4 and 5, we developed an efficient microarchitecture for the implementation of spinal codes by relaxing data dependencies in the ideal code to obtain smaller, fully pipelined hardware. The enabling architectural features are an "α-β" incremental approximate selection algorithm, and a method for obtaining hints to anticipate

successful or failed decoding, which permits early termination and/or feedback-driven adaptation of the decoding parameters.

We have implemented our design on an FPGA and have conducted over-the-air tests. A provisional hardware synthesis suggests that a near-capacity implementation of spinal codes can achieve a throughput of 12.5 Megabits/s in a 65 nm technology, using substantially less area than competitive 3GPP turbo code implementations.

We conclude by noting that further reductions in hardware complexity of spinal decoding are possible. We have focused primarily on reducing the number of candidate values alive in the system at any point in time. Another important avenue of exploration is reducing the complexity and width of various operations within the pipeline. Both Viterbi and Turbo codes operate on extremely narrow values using approximate arithmetic. It should be possible to reduce spinal decoders in a similar manner, resulting in more area-efficient and higher throughput decoders.

Bibliography

[1] 3rd Generation Partnership Project. *Technical Specication Group Radio Access Networks, TS 25.101 V3.6.0*, 2001-2003.

[2] A. Anastasopoulos. Delay-optimal hybrid ARQ protocol design for channels and receivers with memory as a stochastic control problem. In *Communications, 2008. ICC'08. IEEE International Conference on*, pages 3536–3541, 2008.

[3] J. Anderson and S. Mohan. Sequential coding algorithms: A survey and cost analysis. *IEEE Trans. on Comm.*, 32(2):169–176, 1984.

[4] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate (Corresp.). *IEEE Trans. Info. Theory*, 20(2):284–287, 1974.

[5] R.J. Barron, C.K. Lo, and J.M. Shapiro. Global design methods for raptor codes using binary and higher-order modulations. In *IEEE MILCOM*, 2009.

[6] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1. In *ICC*, 2002.

[7] John Bicket. Bit-Rate Selection in Wireless Networks. Master's thesis, M.I.T., February 2005.

[8] P.J. Black and T.H.-Y. Meng. Hybrid Survivor Path Architectures for Viterbi Decoders. *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 1:433–436, 1993.

[9] J. Camp and E. Knightly. Modulation Rate Adaptation in Urban and Vehicular Environments: Cross-Layer Implementation and Experimental Evaluation. In *MobiCom*, 2008.

[10] David Chase. Code Combining: A Maximum-Likelihood Decoding Approach for Combining an Arbitrary Number of Noisy Packets. *IEEE Trans. on Comm.*, 33(5):385–393, May 1985.

[11] Chih-Chi Cheng, Yi-Min Tsai, Liang-Gee Chen, and Anantha P. Chandrakasan. A 0.077 to 0.168 nJ/bit/iteration scalable 3GPP LTE turbo decoder with an adaptive sub-block parallel scheme and an embedded DVFS engine. In *CICC*, pages 1–4, 2010.

[12] R. Cypher and C.B. Shung. Generalized Trace Back Techniques for Survivor Memory Management in the Viterbi Algorithm. In *Proc. IEEE GLOBECOM*, Sec 1990.

[13] Erez, U. and Trott, M. and Wornell, G. Rateless Coding for Gaussian Channels. *IEEE Trans. Info. Theory*, 58(2):530–547, 2012.

[14] O. Etesami, M. Molkaraie, and A. Shokrollahi. Raptor codes on symmetric channels. In *ISIT*, 2005.

[15] Ettus Research USRP2. `http://www.ettus.com/products`.

[16] G.D. Jr. Forney. The Viterbi Algorithm. In *Proceedings of the IEEE*, volume 61, pages 268–278. IEEE, March 1973.

[17] R. Gallager. Low-density parity-check codes. *IRE Trans. Information Theory*, 8(1):21–28, 1962.

[18] R.K. Ganti, P. Jayachandran, H. Luo, and T.F. Abdelzaher. Datalink streaming in wireless sensor networks. In *SenSys*, pages 209–222, 2006.

[19] L.F. Gonzalez Perez, E. Boutillon, A.D. Garcia Garcia, J.E. Gonzalez Villarruel, and R.F. Acua. VLSI Architecture for the M Algorithm Suited for Detection and Source Coding Applications. In *International Conference on Electronics, Communications and Computers*, pages 119 – 124, feb. 2005.

[20] A. Gudipati and S. Katti. Strider: Automatic rate adaptation and collision handling. In *SIGCOMM*, 2011.

[21] J. Ha, J. Kim, and S. McLaughlin. Rate-compatible puncturing of low-density parity-check codes. *IEEE Trans. Info. Theory*, 2004.

[22] J. Hagenauer. Rate-compatible punctured convolutional codes (rcpc codes) and their applications. *IEEE Trans. on Comm.*, 1988.

[23] Joachim Hagenauer and Peter Hoeher. A Viterbi Algorithm with Soft-Decision Outputs and its Applications. In *Proc. IEEE GLOBECOM*, pages 1680–1686, Dallas, TX, November 1989.

[24] B. Han, A. Schulman, F. Gringoli, N. Spring, B. Bhattacharjee, L. Nava, L. Ji, S. Lee, and R. Miller. Maranello: practical partial packet recovery for 802.11. In *NSDI*, pages 14–14, 2010.

[25] G. Holland, N. Vaidya, and P. Bahl. A Rate-Adaptive MAC Protocol for Multihop Wireless Networks. In *MobiCom*, 2001.

[26] Peter A. Iannucci, Kermin Elliott Fleming, Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. A hardware spinal decoder. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS '12, pages 151–162, New York, NY, USA, 2012. ACM.

[27] Peter A. Iannucci, Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. No symbol left behind: a link-layer protocol for rateless codes. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, Mobicom '12, pages 17–28, New York, NY, USA, 2012. ACM.

[28] IEEE Standard 802.11: Wireless LAN Medium Access Control and Physical Layer Specifications, August 1999.

[29] J.C. Ikuno, M. Wrulich, and M. Rupp. Performance and modeling of LTE H-ARQ. In *Proc. International ITG Workshop on Smart Antennas (WSA 2009), Berlin, Germany*, 2009.

[30] Kyle Jamieson. *The SoftPHY Abstraction: from Packets to Symbols in Wireless Network Design*. PhD thesis, MIT, Cambridge, MA, 2008.

[31] Kyle Jamieson and Hari Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *SIGCOMM*, 2007.

[32] B. Jenkins. Hash functions. Dr. Dobb's Journal, 1997.

[33] J. Li and K. Narayanan. Rate-compatible low density parity check codes for capacity-approaching ARQ scheme in packet data communications. In *Int. Conf. on Comm., Internet, and Info. Tech.*, 2002.

[34] Chien-Ching Lin, Chia-Cho Wu, and Chen-Yi Lee. A Low Power and High-speed Viterbi Decoder Chip for WLAN Applications. In *ESSCIRC '03*, pages 723 –726, 2003.

[35] Edward C. Lin, Kai Yu, Rob A. Rutenbar, and Tsuhan Chen. A 1000-word vocabulary, speaker-independent, continuous live-mode speech recognizer implemented in a single FPGA. In *FPGA*, pages 60–68, 2007.

[36] K C Lin, N Kushman, and D Katabi. ZipTx: Exploiting the Gap Between Bit Errors and Packet Loss. In *MobiCom*, 2008.

[37] C. Lott, O. Milenkovic, and E. Soljanin. Hybrid ARQ: theory, state of the art and future directions. In *2007 IEEE Information Theory Workshop on Information Theory for Wireless Networks*, pages 1–5. IEEE, July 2007.

[38] M. Luby. LT codes. In *FOCS*, 2003.

[39] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer. Raptor Forward Error Correction Scheme for Object Delivery. RFC 5053 (Proposed Standard), October 2007.

[40] Nallatech. Intel Xeon FSB FPGA Socket Fillers. *http://www.nallatech.com/intel-xeon-fsb-fpga-socket-fillers.html*.

[41] M. C. Ng, K. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan. Airblue: A System for Cross-Layer Wireless Protocol Development. In *ANCS'10*, San Diego, CA, 2010.

[42] Man Cheuk Ng, Muralidaran Vijayaraghavan, Gopal Raghavan, Nirav Dave, Jamey Hicks, and Arvind. From WiFI to WiMAX: Techniques for IP Reuse Across Different OFDM Protocols. In *MEMOCODE'07*.

[43] E. Paaske, S. Pedersen, and J. Sparsø. An Area-efficient Path Memory Structure for VLSI Implementation of High Speed Viterbi Decoders. *Integr. VLSI J.*, pages 79–91, November 1991.

[44] R. Palanki and J.S. Yedidia. Rateless codes on noisy channels. In *ISIT*, 2005.

[45] J. Perry, H. Balakrishnan, and D. Shah. Rateless spinal codes. In *HotNets-X*, October 2011.

[46] J. Perry, P. Iannucci, K. Fleming, H. Balakrishnan, and D. Shah. Spinal Codes. In *SIGCOMM*, August 2012.

[47] Jonathan Perry, Peter A. Iannucci, Kermin E. Fleming, Hari Balakrishnan, and Devavrat Shah. Spinal codes. *SIGCOMM Comput. Commun. Rev.*, 42(4):49–60, August 2012.

[48] M. Power, S. Tosi, and T. Conway. Reduced Complexity Path Selection Networks for M-Algorithm Read Channel Detectors. *IEEE Transactions on Circuits and Systems*, 55(9):2924 –2933, Oct. 2008.

[49] S. Sen, N. Santhapuri, R.R. Choudhury, and S. Nelakuditi. AccuRate: Constellation-based rate estimation in wireless networks. *NSDI*, 2010.

[50] A. Shokrollahi. Raptor codes. *IEEE Trans. Info. Theory*, 52(6), 2006.

[51] E. Soljanin, N. Varnica, and P. Whiting. Incremental redundancy hybrid ARQ with LDPC and Raptor codes. *IEEE Trans. Info. Theory*, 2005.

[52] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Info. Theory*, 13(2):260–269, 1967.

[53] M. Vutukuru, H. Balakrishnan, and K. Jamieson. Cross-Layer Wireless Bit Rate Adaptation. In *SIGCOMM*, 2009.

[54] S Wong, H Yang, S Lu, and V Bharghavan. Robust Rate Adaptation for 802.11 Wireless Networks. In *MobiCom*, 2006.

[55] Xilinx. Xilinx University Program XUPV5-LX110T Development System. *http://www.xilinx.com/univ/xupv5-lx110t.htm*.