A Hardware Spinal Decoder

Peter A. Iannucci, Kermin Elliott Fleming, Jonathan Perry, Hari Balakrishnan, and Devavrat Shah
Massachusetts Institute of Technology
Cambridge, Mass., USA
{iannucci,kfleming,yonch,hari,devavrat}@mit.edu

ABSTRACT

Spinal codes are a recently proposed capacity-achieving rateless code. While hardware encoding of spinal codes is straightforward, the design of an efficient, high-speed hardware decoder poses significant challenges. We present the first such decoder. By relaxing data dependencies inherent in the classic M-algorithm decoder, we obtain area and throughput competitive with 3GPP turbo codes as well as greatly reduced latency and complexity. The enabling architectural feature is a novel "α-β" incremental approximate selection algorithm. We also present a method for obtaining hints which anticipate successful or failed decoding, permitting early termination and/or feedback-driven adaptation of the decoding parameters.

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

Categories and Subject Descriptors: B.4.1 [Data Communications Devices]: Receivers; C.2.1 [Network Architecture and Design]: Wireless communication

General Terms: Algorithms, Design, Performance

Keywords: Wireless, rateless, spinal, decoder, architecture

1. INTRODUCTION

At the heart of every wireless communication system lies a *channel code*, which incorporates methods for error correction. 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).

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.

Our interest is in *rateless codes*, defined as codes for which any encoding of a higher rate is a prefix of any lower-rate encoding (the "prefix property"). Rateless codes are interesting because they offer a way to achieve high throughput over *time-varying* wireless networks: a good rateless code inherently sends only as much data as required to communicate reliably under any given channel conditions. As conditions change, a good rateless code adapts naturally.

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 requirement to a manageable level.

This paper presents the design, implementation, and 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 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 novel 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.

ANCS'12, October 29–30, 2012, Austin, Texas, USA.

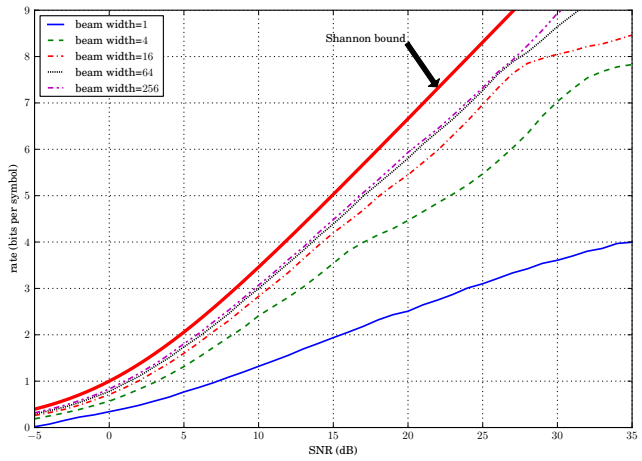


Figure 1: Coding efficiency achieved by the spinal decoder increases with the width of the explored portion of the tree. Hardware designs that permit wide exploration are desirable.

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.

2. 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 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 polynomial 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.

2.1 Spinal Codes

For context, we review the salient details of spinal codes [21, 20].

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 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 2. The

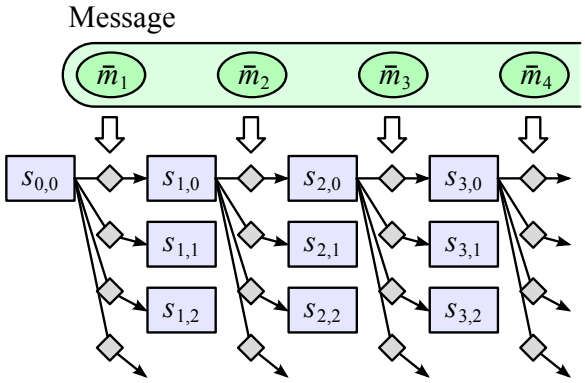


Figure 2: Computation of pseudo-random words $s_{i,j}$ in the encoder, with hash function application depicted by a diamond. Each $\bar{m}_i$ is k message bits.

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 [21].

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 alive 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 signal variance that remains after subtracting the corresponding encoded message from the received signal. 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,

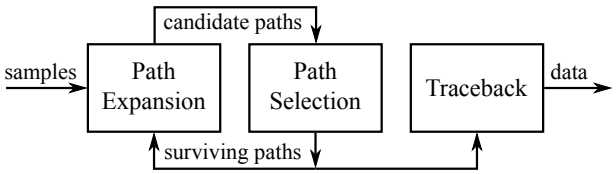


Figure 3: Block diagram of M-algorithm hardware.

$\text{Var}(y - \hat{x}) = P \cdot (2 + \frac{1}{\text{SNR}})$. The sum of squared differences is an estimator of this variance and discriminates between the two cases.

2.2 Existing M-Algorithm Implementations

The decoder described above is essentially the M-algorithm (MA) [2]. A block diagram of MA is shown in Figure 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 [18] 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 [9] or BCJR [3] 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 [10] [22] 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 novel work in this paper will focus on achieving high-quality decoding while minimizing the size of the sort network that must be constructed.

The M-algorithm implementation in [22] 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 is, to the best of our knowledge, 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 [15] where they are 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.

3. 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 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 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 §5.

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 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. We show in §6 how we keep this memory small and minimize the time spent tracing back through the memory, while also 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 would require pipelining dozens of simultaneous decodes, with a commensurate storage requirement.

3.1 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

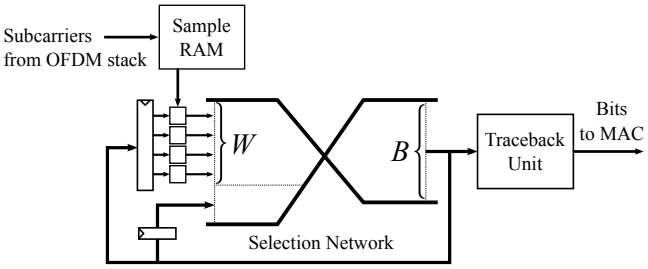


Figure 4: The initial decoder design with W workers and no pipelining.

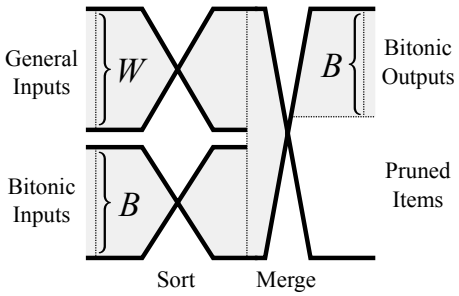


Figure 5: Detail of the incremental selection network.

as many cycles are needed to complete a decode. This design is depicted in Figure 4. The procedure for merging into the register is detailed in §4.1.

4. 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. Thus, during each cycle, the selection network must combine a number of fresh inputs with the surviving inputs from previous cycles, and prune away inputs which score poorly.

4.1 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 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.

4.2 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 ($B \cdot 2^k \gg W$).

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

Table 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.

4.3 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 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 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

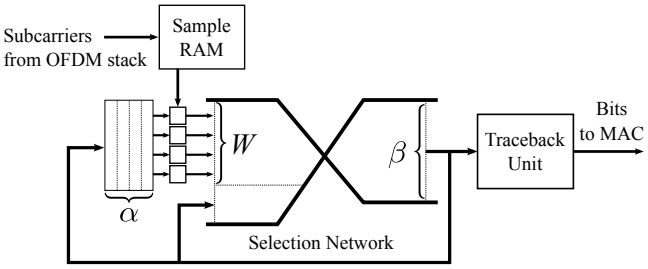


Figure 6: A parametric α-β spinal decoder with W workers. A shift register of depth α replaces the ordinary register in Figure 4. Individual stages of the selection pipeline are not shown, but the width of the network is reduced from B to $\beta = B/\alpha$.

is pruned, the packet will fail to decode until more passes arrive. A detailed analysis is given in §4.6.

Figure 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 §4.1.

4.4 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 9. Combining the two proposed optimizations achieves performance that is slightly better than a random shuffle.

4.5 Further Optimization

A further reduction of the α-β selection network is possible by concatenating multiple smaller selection networks as shown in Figure 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.

4.6 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*

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

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.

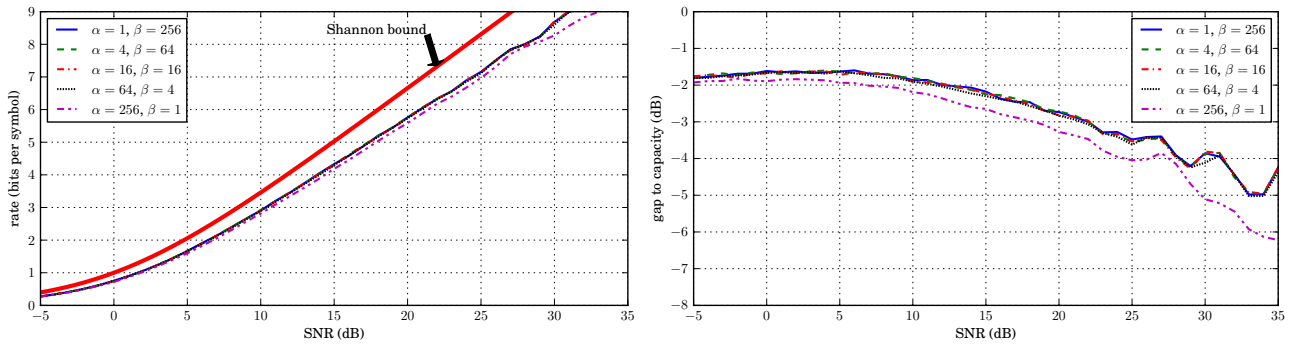
PROOF. Suppose that the outputs are sorted from best to worst. Suppose also that the input consists of a random permutation of $(1, \ldots, N)$. For general input, we can imagine that each input has been replaced by the position at which it would appear in a list sorted from best to worst. Under this 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

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

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



(a) Bits per Symbol

(b) Gap to Capacity

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

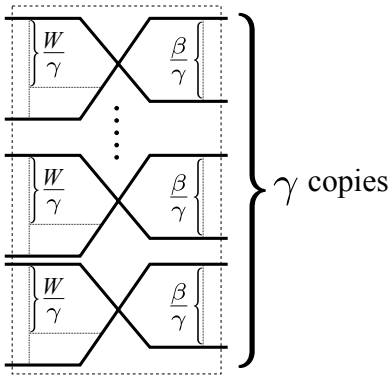


Figure 8: Concatenated selection network, emulating a β, W network with γ smaller selection networks.

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

$$\mathbb{E}\left[\sum_{i=1}^{m} \mathbb{1}_{\{i \text{ in output}\}}\right] = \sum_{i=1}^{m} \mathbb{P}(i \text{ in output})$$

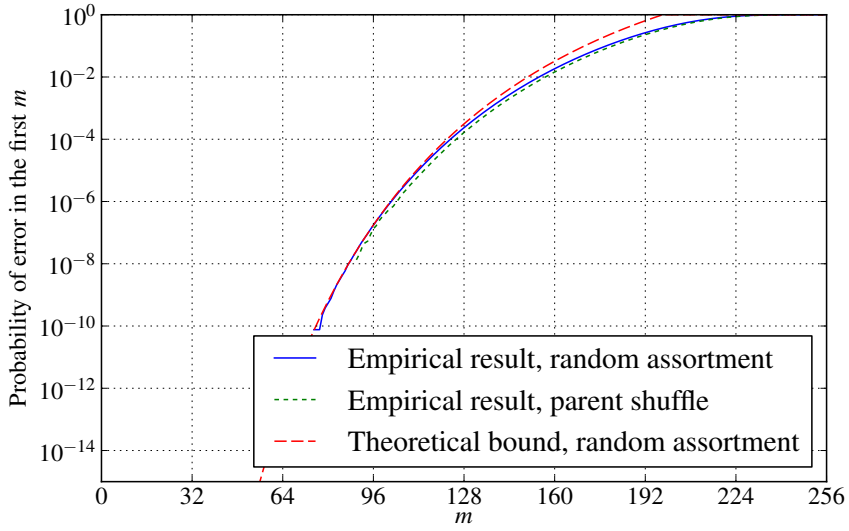
$$= m - \sum_{i=1}^{m}\sum_{j=\beta}^{n} \frac{\binom{n-1}{j}\binom{N-n}{i-j-1}}{\binom{N-1}{i-1}}$$

5. Path Expansion

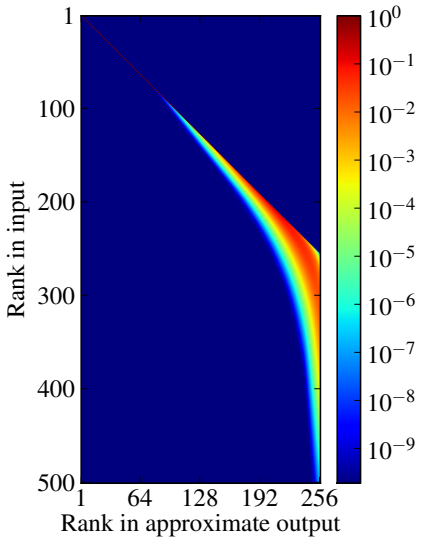
Thanks to the optimizations of §4, path metric units occupy a large part of the area of the final design. The basic worker is shown in Figure 10. This block encodes the symbols corresponding to k bits of data by hashing and mapping them, then subtracts them from the received samples and computes the squared residual. 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.

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 [13]. 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 $8 \times 8 \rightarrow 16$ 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.



(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 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.

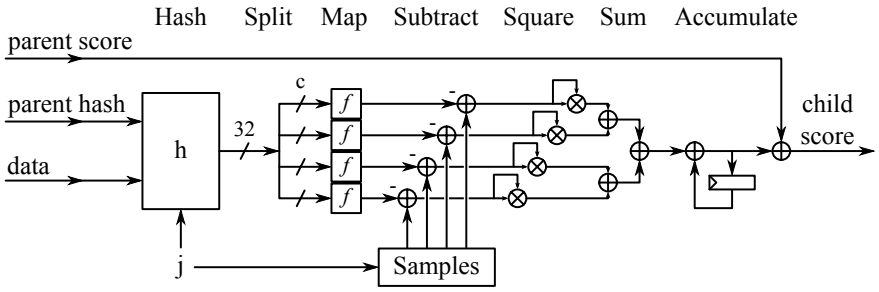


Figure 10: Schematic of the path metric unit. Over one or more cycles indexed by j, the unit accumulates squared differences between the received samples in memory and the symbols which would have been transmitted if this candidate were correct. Not shown is the path for returning the child hash, which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

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 §6). 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.

6. Online Traceback

The final stage of the M-algorithm decoder is traceback. Ideally, at the end of decoding, traceback begins from the most likely child, outputting the corresponding set of k bits and recursing up the tree to the node'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. For each new beam, a traceback of c steps is performed starting from the best candidate, and k bits are produced. The variable c represents the effective constraint length of the code: the maximum number of steps until all surviving paths converge on a single, likely ancestor. Beyond this point of convergence, the paths are identical. Because only c steps of traceback need to be performed to find this ancestor, only c beams' worth of data need to be maintained. For many codes, c is actually quite small. For example, convolutional code traceback lengths are limited to $\log_2 s$, where s is the number of states in the code. In spinal codes, particularly with our selection approximations, it is possible for bad paths to appear with very long convergence distances. However, in practice we find that convergence is usually quite rapid, on the order of 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 [7, 19]. 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 [7]. Even hybrid backtrace/register-exchange architectures [5] 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 required storage is on the order of tens of kilobits. Figure 13 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. [14]. If the packet being processed will be decoded correctly, parent and child backtracks should be similar. Figure 11 shows a distribution of convergence distances under varying channel conditions, confirming this intuition.

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 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.

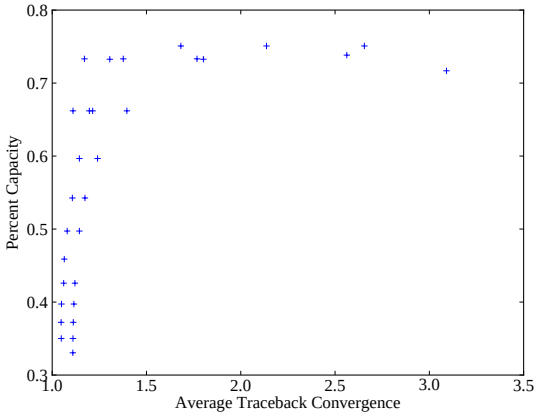


Figure 11: Average convergence distance between adjacent tracebacks, collected for various SNRs and numbers of passes. Near capacity, tracebacks begin to take longer to converge.

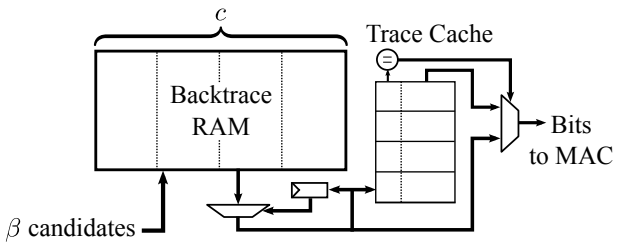


Figure 12: Traceback Microarchitecture. Some control paths have been eliminate to simplify the diagram.

7. EVALUATION

7.1 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 [23] and USRP2 [8]. We use the USRP2 to feed IQ samples to an Airblue [17]-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 3dB. Additionally, from a hardware standpoint, some interesting decoder 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 [16]. 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-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.

7.2 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 [4] are a capacity-approaching code currently deployed in most modern cellular standards.

There are several metrics against which one might compare hard-

	3G Turbo (1dB)	3G Turbo(1dB) [6]	Spinal (1dB)	Spinal(-5dB)
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 2: Memory Usage for turbo and spinal decoders supporting 5120 bit packets. Memory area accounts for more than 50% of turbo decoder area.

ware implementations of spinal and turbo codes: implementation area, throughput, latency, and power consumption.

A fundamental difference between turbo and spinal decoders is that the former are *iterative*, while spinal decoders are sequential and thus can be streaming. This means that a 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 2, 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 1dB from -5dB, 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.

7.3 Performance of Hardware Decoder

Figure 13 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. 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 3dB.

Table 3 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 [6] 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.

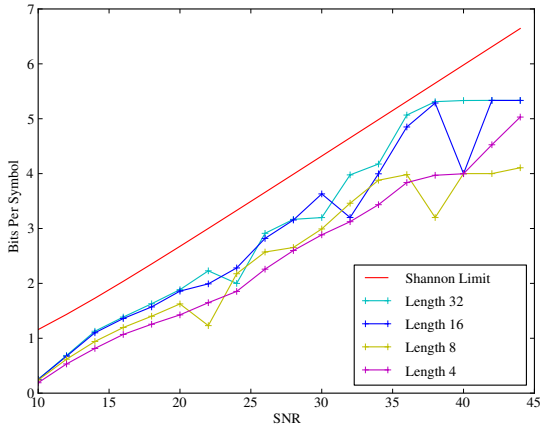


Figure 13: Throughput of the hardware decoder with various traceback lengths.

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 14 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 decoders with throughputs greater than 100 Mbps.

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 3: 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.

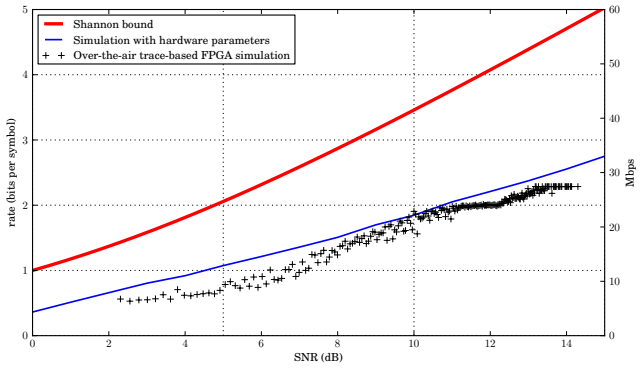


Figure 14: Performance of $B = 4$, $\beta = 4$, $\alpha = 1$ decoder over the air versus identically parameterized C++ model. Low code efficiency is due to the narrow width of the decoder, which yields a high throughput implementation.

7.4 On-Air Validation

The majority of the performance results presented in this paper 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 accurately model both the hardware and the wireless channel, it is important to validate the simulation results with on-air testing.

Figure 14 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.

7.5 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 [12], 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 [11], is an excellent candidate for this role. As Figure 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.

8. Conclusion

Spinal codes are, in theory and simulation, a promising new capacity-achieving code. In this paper, we have 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 a novel "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.

Acknowledgments

We thank Lixin Shi for helpful comments. Support for P. Iannucci and J. Perry came from Irwin and Joan Jacobs Presidential Fellowships. An Intel Fellowship supported K. Fleming. Additional support for J. Perry came from the Claude E. Shannon Research Assistantship.

References

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

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

[3] 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.

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

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

[6] C.-C. Cheng, Y.-M. Tsai, L.-G. Chen, and A. 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.

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

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

[9] G. J. Forney. The Viterbi Algorithm. In *Proceedings of the IEEE*, volume 61, pages 268–278. IEEE, Mar. 1973.

[10] L. Gonzalez Perez, E. Boutillon, A. Garcia Garcia, J. Gonzalez Villarruel, and R. 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.

[11] J. Hagenauer and P. Hoeher. A Viterbi Algorithm with Soft-Decision Outputs and its Applications. In *Proc. IEEE GLOBECOM*, pages 1680–1686, Dallas, TX, Nov. 1989.

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

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

[14] C.-C. Lin, C.-C. Wu, and C.-Y. Lee. A Low Power and High-speed Viterbi Decoder Chip for WLAN Applications. In *ESSCIRC '03*, pages 723 –726, 2003.

[15] E. C. Lin, K. Yu, R. A. Rutenbar, and T. Chen. A 1000-word vocabulary, speaker-independent, continuous live-mode speech recognizer implemented in a single FPGA. In *FPGA*, pages 60–68, 2007.

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

[17] 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.

[18] M. C. Ng, M. Vijayaraghavan, G. Raghavan, N. Dave, J. Hicks, and Arvind. From WiFI to WiMAX: Techniques for IP Reuse Across Different OFDM Protocols. In *MEMOCODE'07*.

[19] 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, Nov. 1991.

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

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

[22] 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.

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