Computer Generation of Efficient Software Viterbi Decoders *

Frédéric de Mesmay, Srinivas Chellappa,
Franz Franchetti, and Markus Püschel

Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA 15213, USA
{fdemesma, schellap, franzf, pueschel}@ece.cmu.edu

Abstract. This paper presents a program generator for fast software Viterbi decoders for arbitrary convolutional codes. The input to the generator is a specification of the code and a single-instruction multiple-data (SIMD) vector length. The output is an optimized C implementation of the decoder that uses explicit Intel SSE vector instructions. At the heart of the generator is a small domain-specific language called VL to express the structure of the forward pass. Vectorization is done by rewriting VL expressions, which a compiler then translates into actual code in addition to performing further optimizations specific to the vector instruction set. Benchmarks show that the generated decoders match the performance of available expert hand-tuned implementations, while spanning the entire space of convolutional codes. An online interface to the generator is provided at www.spiral.net.

Key words: Library generation, high performance software, vectorization, domain-specific language, Viterbi algorithm

1 Introduction

The Viterbi algorithm is a maximum likelihood sequence decoder introduced by Andrew Viterbi [1], and finds wide usage in communications, speech recognition, and statistical parsing. In the past, the high throughput requirements for decoding demanded dedicated hardware implementations [2]. However, the dramatically growing processor performance has started to change this situation: intensive processing is now often done in software for reasons of cost and flexibility. A prominent example is software defined radio [3].

Unfortunately, developing a generic high-throughput software Viterbi decoder is difficult. The reason is that the best performance can only be achieved by using vector instructions (such as Intel's Streaming SIMD Extensions, SSE), which most modern processors provide. To take advantage of these instructions,

* This work was supported by NSF through awards 0325687 and 0702386, and by DARPA through the Department of Interior grant NBCH1050009.

the programmer has to explicitly issue them using an intrinsics interface or directly write assembly code. Achieving performance gains with these instructions first requires proper restructuring of the dataflow of the decoder with respect to the vector length. Then, the programmer must deal with the intricacies of the instruction sets as the available instructions differ between platform vendors and vector lengths. Finally, because the total vector length in bits is fixed, there is a degree of freedom arising from the tradeoff between speed and precision (e.g., 4-way vectorization implies 32 bits per element while the faster 16-way vectorization implies only 8 bits per element).

Because of the difficulty in handling these issues, existing approaches are based on manually writing specific assembly routines for each decoder (e.g., [4]), which involves considerable effort given the large set of different codes and platforms used in real-world applications.

Contribution of this paper. In this paper, we present a method to automatically generate fast implementations of software Viterbi decoders. Our generator takes as input, a specification of the convolutional code and the vector length to be used. The output is a C program for the corresponding Viterbi decoder implemented using SSE. Note that the methods presented in this paper are generic enough to apply to other instruction sets. Only the performance critical forward pass of the decoder is actually generated – the infrastructure and the traceback stages are reused from the high-performance decoders by Karn [4].

Our generator consists of three components:

1. A domain specific language, called VL, to describe the forward pass at a high level of abstraction. The language is a generalization of the Kronecker product formalism used to describe fast Fourier transforms [5].
2. A VL rewriting system that restructures the forward pass depending on the target vector length.
3. A compiler that takes VL as input, outputs C code including SSE intrinsics, and performs various low level optimizations.

As we will show, our generator can handle arbitrary convolutional codes, arbitrary vector length, and produces decoders with excellent performance. It is implemented as part of the Spiral program generation system [6]. An online interface is provided at `www.spiral.net/software/viterbi.html`.

Related work. Our approach is similar to the one taken by Spiral to generate programs for linear transforms [6]. Spiral uses the domain-specific language SPL [7] to explore alternative algorithms for a given transform, and vectorize and parallelize fast Fourier transform algorithms (FFTs) [8, 9]. While our VL is closely related to SPL due to the inherent similarities between Viterbi decoding and FFTs (which were already noted in [10, 11]), a significant difference with previous work is that the selection between alternative algorithms is not present. However, the wide range of convolutional codes that one would want to generate still offers a compelling case for on-demand generation of high-performance code.

VL is a subset of the Operator Language (OL) that aims at generalizing SPL to capture more general computations. [12] presents the general OL framework while this this paper focuses on issues specific to Viterbi decoding.

To achieve high performance, other decoding algorithms for convolutional codes also exist. Examples include the lazy Viterbi (efficient for long constraint lengths) and the Fano algorithm (efficient for good signal-to-noise ratios) [13, 14]. This paper only considers "standard" Viterbi decoders.

Organization of this paper. In Section 2 we provide background on convolutional codes and the Viterbi decoding algorithm. In Section 3, we introduce the language VL to describe the forward pass of Viterbi decoding and explain how to generate scalar code. Vectorization through VL rewriting is covered in Section 4. In Section 5, we benchmark the performance of our generated decoders, and conclude in Section 6.

2 Background: Viterbi Decoders

In this section, we provide background on convolutional codes and Viterbi decoders. We then introduce the butterfly representation for Viterbi decoders.

2.1 Encoding Convolutional Codes

The purpose of forward error-correcting codes (FEC) is to prevent corruption of a message by adding redundant information before the message is sent over a noisy channel. At the receiving side, the redundant data is used to reconstruct the original message despite errors. In this paper, we focus only on a single type of FEC, namely convolutional codes. These codes are heavily used in telecommunications standards such as GSM and CDMA.

A convolutional encoder takes as input a bit stream and convolves it with a number of fixed bit sequences to obtain the output bit stream. Since convolution is equivalent to polynomial multiplication, each of these fixed bit sequences is called a (binary) *polynomial* although it is represented by a single integer.

Formally, a convolutional code is specified by N integers smaller than 2^K, denoted with $p_1, \ldots, p_N$. Such a code is said to have a *constraint length* K and a *rate* $1/N$, i.e., for each input bit, the encoder produces N output bits.

Finite State Machine (FSM) representation.. The encoding process can be described using a FSM with 2^{K-1} states that outputs N bits on each transition (Fig. 1). The precise layout depends on the convolutional code itself but each state always has a 0-transition (input bit is 0) and a 1-transition (input bit is 1) to other states. The initial encoding state is assumed to be 0 and the input stream is padded with $K-1$ trailing zeros which guarantees that the final state is also 0.

More precisely, there exists a 0-transition between states n and m if $m \equiv 2n \mod 2^{K-1}$. Similarly, there exists a 1-transition between states n and m if $m \equiv (2n+1) \mod 2^{K-1}$. Denoting the bit-wise AND as &, the bit-wise XOR as $\oplus$ and the XOR on all bits by $\bigoplus$, the output bit $b^\ell_{n\to m}$, corresponding to the polynomial p_ℓ when transitioning from state n to state m is computed as

$$b^\ell_{n\to m} = \bigoplus \Big(p_\ell \& \big(2n \oplus (m \& 1)\big) \Big) \ .$$

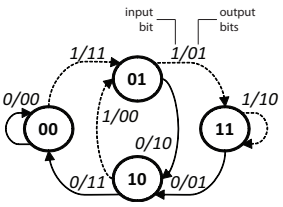


Fig. 1: FSM representation of the encoder $r = 1/2$, $K = 3$ with polynomials 7 and 5.

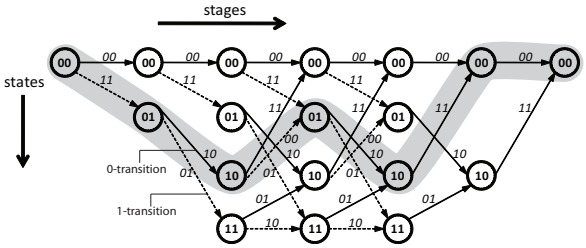


Fig. 2: Viterbi trellis corresponding to the encoder of Fig. 1. The highlighted path shows that the message 1010_2 (first padded to 101000_2 to guarantee that the final state is 0) is encoded as $11\,10\,00\,10\,11\,00_2$.

Viterbi trellis. An equivalent representation of the encoding process "unrolls" the finite state machine in time to yield the Viterbi *trellis*, as shown in Fig. 2. Each path from the initial state to the final state represents a possible message. Note that the number of vertical stages, called the *frame length* F, is independent of the convolutional code but agreed upon by the two communicating parties.

The different states of the encoder are placed vertically, the different time steps, or *stages* are placed horizontally. The initial state (first stage) when starting a frame is 0. The zero padding explained previously implies that the last $K - 1$ transitions are 0-transitions, guaranteeing that the final state is also 0.

2.2 Viterbi Decoding

The Viterbi algorithm is a dynamic programming method that performs maximum likelihood sequence decoding (MLSD) on a convolutionally encoded stream. Intuitively, the decoder receives a bit stream and has to find the path in the Viterbi trellis that best corresponds to it, which would ideally be the same path the encoder originally took. The best visualization of the Viterbi algorithm again uses the Viterbi trellis but its purpose is now reversed: the incoming message is fixed and the path is to be found. It is composed of three phases, the *branch metric* computation, the *path metric* computation, and the *traceback*.

Branch metrics computation. In the first phase, the Viterbi algorithm assigns a cost, called the *branch metric*, to each edge in the trellis. This value represents how well the received bits would match if we knew the encoder took the transition corresponding to a given edge. It is computed by taking the Hamming distances between the bits the transition should output and the actually received ones (Fig. 3).

Path metrics computation. After the previous phase, the problem is equivalent to finding the shortest path between the entry and the exit vertices on a directed acyclic graph with weighted edges. Therefore, the second phase is a breadth-first forward traversal of the graph. It progressively computes the *path metric*, which is the shortest path to get from the root to each vertex. If a state has the path metric π, there exists one message that ends in the state with π

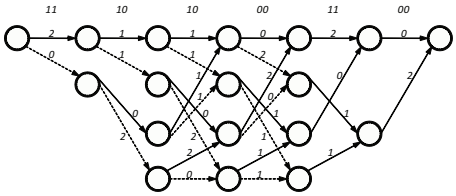


Fig. 3: Branch metrics computation. Assuming the decoder just received the message $11\,10\,10\,00\,11\,00_2$ which is the previous example message with two bit flips (corresponding to injected errors), the Hamming distance between the bits actually received and the output bits corresponding to each arrow (shown in Fig. 2) is computed.

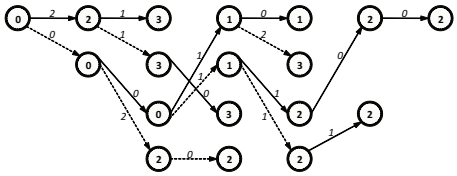


Fig. 4: Path metrics computation and traceback. Using the branch metric, the shortest path from the initial state to all states is computed, leaving only one predecessor for each node. Finally, only one complete path remains: it can be simply read off by starting in the final state and going backwards. Here, this path corresponds to the original message: 101000_2.

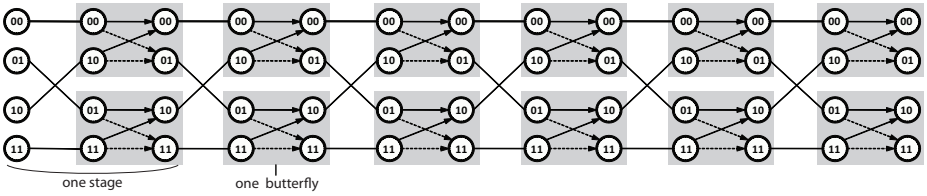


Fig. 5: Each stage in the Viterbi trellis consists of a perfect shuffle and 2^{K-2} parallel butterflies (here $K = 3$ and $F = 6$).

corrupted bits and this message is less or equally corrupted than all other possible messages. While computing this, the predecessor of each node is remembered as a *decision bit*[1] (Fig. 4).

Traceback. The decision bits describe the ancestor of each vertex. Given this information and the final state, one can reconstruct the shortest path, called the *survivor path* by reading off predecessors.

In a software Viterbi decoder, it is important to perform branch and path metrics computations simultaneously to improve the ratio of operations over memory accesses. The fusion of these two phases is called the *forward pass*.

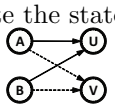
2.3 Viterbi Butterflies

The trellis shown on Fig. 2 has a regular structure except for the initial and final stages. The initial stage can be handled like all other stages by inserting prohibitively high path metrics as appropriate. Handling the final stage like all other stages simply involves computing all path metrics—the useless ones are automatically discarded.

Closer inspection of the trellis structure now shows that each stage of the forward pass can be decomposed in two phases: a fixed permutation called a *perfect*

[1] The structure of the FSM guarantees that there are exactly two incoming edges into each vertex, except for the leftmost nodes in the trellis where there is only one.

shuffle and a parallel operation on 2^{K-2} 2-by-2 substructures called *butterflies* (Fig. 5). In the following, we denote the states of a butterfly as shown below:



During the path metric computation, each butterfly does two *Add-Compare-Select* operations to compute the path metrics π_U and π_V from the path metrics π_A and π_B and the branch metrics $\beta_{A\to U}$, $\beta_{A\to V}$, $\beta_{B\to U}$ and $\beta_{B\to V}$:

$$\begin{cases} \pi_U = \min_{d_U}(\pi_A + \beta_{A\to U}, \pi_B + \beta_{B\to U}) \\ \pi_V = \min_{d_V}(\pi_A + \beta_{A\to V}, \pi_B + \beta_{B\to V}) \ . \end{cases} \tag{1}$$

Note that the minimum operator $\min_d(a, b)$ actually performs both the compare and select operations simultaneously. It returns the actual minimum of a and b and stores the binary decision in the decision bit d.

Simplification. Other effects, notably polynomial symmetries and soft decisions, actually modify the above expression. However, for reasons of brevity, we will not elaborate on them here.

3 Generating Scalar Code

The goal of this paper is to enable computer generation of efficient software implementations of the Viterbi decoder for arbitrary convolutional codes. To achieve this, we introduce a domain-specific language, called Viterbi language (VL), to concisely describe the (most critical) forward pass of the decoder and its associated compiler that translates the description into actual C code. Both are described in this section.

There are two main reasons for using a domain-specific language. First, it structures and simplifies the implementation of our software generator. Second, it enables the SIMD vectorization of the forward pass through rewriting VL expressions rather than optimizing C code. The vectorization is explained in Section 4.

For reasons that will become clear, VL is closely related to the signal processing language (SPL), a domain-specific language that was designed to generate high performance implementations of linear transforms [6, 7]. We start with a brief introduction to SPL, motivate VL and explain the compilation process.

3.1 Fast Transform Algorithms: SPL

Linear transforms in signal processing are usually presented as summations, but can be equivalently viewed as a matrix. A linear transform computes $y = Tx$, where x is the complex input vector, y the complex output vector, and T the fixed transform matrix. For example, the n-point Fourier and Walsh-Hadamard transforms are defined by the following $n \times n$ matrices [5]:

$$\mathbf{DFT}_n = [e^{-2\pi kl\sqrt{-1}/n}]_{0\le k,l<n} \ ,$$
$$\mathbf{WHT}_n = \begin{bmatrix} \mathbf{WHT}_{n/2} & \mathbf{WHT}_{n/2} \\ \mathbf{WHT}_{n/2} & -\mathbf{WHT}_{n/2} \end{bmatrix}, \quad \mathbf{WHT}_1 = [1] \ .$$

A fast algorithm for a transform T reduces the number of operations required for computing Tx and can be viewed as a factorization of T into a product of sparse structured matrices.

SPL. SPL is the language used to describe such algorithms. It is based on matrix algebra and captures structured matrices. Parametrized symbols are used to represent frequently occurring matrices:

- The $n \times n$ identity matrix is denoted with I_n.
- The *stride permutation* matrix L^n_k reads the input at stride k and stores it at stride 1. In particular $\mathrm{L}^n_{n/2}$ is the *perfect shuffle* that interleaves the first half of a vector with the second half.
- The *butterfly*[2] matrix F_2 corresponds to a DFT on two points: $F_2 = \left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$
- The $n \times n$ *bit-reversal* permutation is denoted with R_n and the *twiddle* matrix T^n_i is a particular diagonal matrix. Their exact form is not important here.

Further, SPL uses matrix constructs to build matrices from other matrices. An example is the product AB which effectively composes two matrix vector multiplications: $(AB)x = A(Bx)$. The product can be indexed as in $\prod_{i=0}^{n} A_i = A_1 A_1 \dots A_n$.

Finally, the *Kronecker product* (also called *tensor product*) of two matrices and its indexed variant are defined as

$$A \otimes B = [a_{k,l}B]\,, \quad A \otimes_j B_j = [a_{k,l}B_j]\,, \quad A = a_{k,l}\ .$$

Most importantly,

$$\mathrm{I}_n \otimes A = \begin{pmatrix} A & & \\ & \ddots & \\ & & A \end{pmatrix}\ .$$

Pease algorithms. The SPL expressions for the Pease $O(n \log n)$ algorithms for the Fourier and Walsh-Hadamard transforms is shown below:

$$\mathbf{WHT}_{2^n} \rightarrow \prod_{i=0}^{n-1} \left((\mathrm{I}_{2^{n-1}} \otimes F_2)\, \mathrm{L}^{2^n}_{2^{n-1}} \right)\ , \tag{2}$$

$$\mathbf{DFT}_{2^n} \rightarrow R_{2^n} \prod_{i=0}^{n-1} \left(T^n_i (\mathrm{I}_{2^{n-1}} \otimes F_2)\, \mathrm{L}^{2^n}_{2^{n-1}} \right)\ . \tag{3}$$

The associated dataflow for the Pease WHT is shown in Fig. 6 (the Pease DFT is very similar). Note the similarity to the Viterbi trellis shown in Fig. 5, but remember that the butterflies operate differently. The resemblance between the DFT, the WHT and the Viterbi forward pass was already noted in [10, 11]. Omega networks also share this dataflow [15].

3.2 Representing the Viterbi Algorithm

The Viterbi algorithm is not a linear transform and therefore, does not fit the earlier framework. However, as discussed above, the forward pass (branch and path metric computation, excluding the traceback) is closely related to the Pease algorithms (2) and (3).

[2] The Viterbi and DFT butterflies are different but related as we will see.

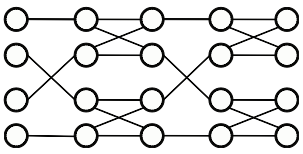


Fig. 6: Dataflow of the Pease algorithm for the $\mathbf{WHT}_4$.

<op> ::=	$\mathbf{F}_{K,F}$	*Viterbi forward pass*
	$\mid \mathrm{I}_n$	*identity*
	$\mid \mathrm{L}^{mn}_n$	*stride permutation*
	$\mid B_{i,j}$	*Viterbi butterfly*
	$\mid$ <op> <op>	*composition*
	$\mid \prod$ <op>	*iterative composition*
	$\mid$ <op> $\otimes$ <op>	*tensor product*

Table 1: Definition of the Viterbi Language in Backus-Naur form

From now on, we will only consider the forward pass, excluding the traceback. The reason is that the traceback is both trivial and computationally much cheaper than the forward pass, requiring $O(F)$ operations versus $O(2^K F)$ for the forward pass. Hence, in practice, except for very short constraint lengths, a generic traceback is not the performance bottleneck.

Butterflies similarities. The DFT butterfly F_2 is an operator that takes two inputs x_0 and x_1 and produces two outputs y_0 and y_1:

$$\begin{cases} y_0 = x_0 + x_1 \\ y_1 = x_0 - x_1 \end{cases} .$$

Similarly, we view the j-th Viterbi butterfly decoding the i-th codeword as an operator $B_{i,j}$ that consumes and produces two path metrics as in (1). The difference between F_2 and $B_{i,j}$ is that, depending on its position, the Viterbi butterfly uses values from some external arrays to compute the branch metrics, and it also writes values to an external decision bit array (through the "select" part of the minimum operator).

Viterbi language (VL). In Table 1, we give the grammar in Backus-Naur form of a domain specific language called VL tailored to describe the operations performed during the forward pass of the Viterbi algorithm. VL uses parts of SPL but also includes the Viterbi butterfly. In SPL, the composition operation is equivalent to matrix multiplication, but this is not true of VL. We will occasionally refer to elements of VL as operators.

Forward pass algorithm. Using VL, The forward pass of a Viterbi decoder with constraint length K, frame length F, denoted $\mathbf{F}_{K,F}$ can be expressed in a way that is similar to the Pease algorithms (2) and (3):

$$\mathbf{F}_{K,F} \rightarrow \prod_{i=1}^{F} \left((\mathrm{I}_{2^{K-2}} \otimes_j B_{F-i,j})\, \mathrm{L}^{2^{K-1}}_{2^{K-2}} \right) \quad . \tag{4}$$

3.3 Compiler

The VL compiler is responsible for producing efficient code from an algorithm expressed in VL. By translating an operator A into code, we mean creating the code for the function `A` that takes the input vector `x` and the output vector `y` as parameters and performs $y = A(x)$.

To generate the code, the compiler traverses the VL expression tree top-down, matching sub-trees with the templates shown in Table 2 and specializing

Table 2: Translating VL expressions to code. x denotes the input and y the output vector. C and D are generic operators optionally parametrized by their superscript and of domain and range optionally specified by their subscript. `x[b:e]` denotes the sub-vector of x starting at `b` and ending at `e`.

construct	code
$y = (CD)x$	`t = D(x);` `y = C(t);`

$y = \prod_{i=0}^{l-1} C^i x$	`y = C(l-1, x);`
	`for (i=l-2;i>=0;i--)`
	`  y = C(i, y);`

$y = (\mathrm{I}_m \otimes_j C_n^j)x$	`for (j=0;j<m;j++)`
	`  y[j*n:j*n+n-1] =`
	`  C(j, x[j*n:j*n+n-1]);`

$y = \mathrm{L}_m^{mn}\, x$	`for (i=0;i<m;i++)`
	`  for (j=0;j<n;j++)`
	`  y[i+m*j]=x[n*i+j];`

$y = B_{i,j}x$	*see equation* (1)

them if needed. Plugging this code inside the generic traceback would yield a correct, albeit unoptimized implementation.

In practice, various optimizations are performed such as loop unrolling, array scalarization, strength reduction, copy propagation, precomputation and common sub-expression elimination. While some of these optimizations may be left to a C compiler, performing them inside the VL compiler typically yields better results, as C compilers generally have conservative aliasing assumptions. Most importantly, it also performs loop merging which means that perfect shuffles are never explicitly performed but merged (i.e., translated into readdressing) with the subsequent computation.

4 Generating Vector Code

The vast majority of current processors provide additional instructions working on vector registers, often branded as "multimedia" extensions like Intel's MMX and SSE or AMD's 3DNow!. The speedups for suitably structured applications like Viterbi decoders can be significant: for example, [4] achieves up to a 16x speedup using SSE. We first provide some background on these instructions, then show how to automatically take advantage of them by rewriting VL expressions. Finally we tackle overflows, a side effect of vectorizing the Viterbi algorithm.

4.1 Background: Short-Vector Instructions

Single-instruction multiple-data (SIMD) vector instructions instructions perform operations on short vectors in parallel. We call the length of the vector ν and

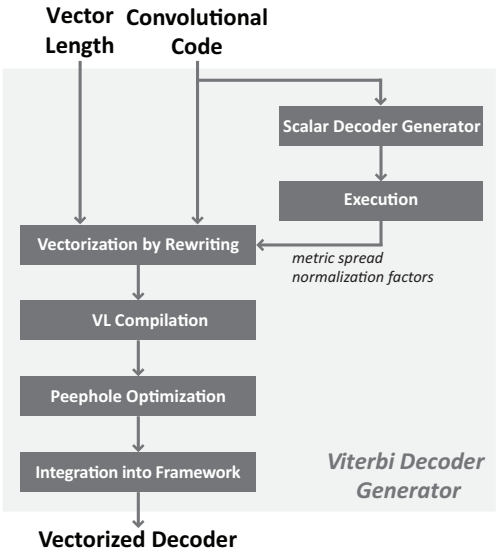


Fig. 7: Automatic generation of vectorized Viterbi decoders.

the instructions ν-way. For instance, if $\nu = 4$, a point-wise addition of 4 scalars could be done with a single vector instruction instead of four *scalar* instructions. Also, vector instruction sets offer ways to reorganize (*shuffle*) the data within a short vector.

The *vector length* ν depends on the exact instruction set. The larger ν, the more operations can be performed simultaneously. In general, the expected speedup of efficiently *vectorized* code over its scalar counterpart can be up to the order of ν which makes it critical in high-performance applications. For example, Intel-compatible processors offer integer vector instructions for $\nu = 4 - 16$.

The drawback of these instructions is that they add complexity at various levels. First, they only operate on ν *contiguous* elements, which means algorithms have to be restructured to expose this parallelism. This step involves algorithmic knowledge, and compilers often fail at doing it automatically. Second, since compilers fail, it has to be done by hand which is complex and time-consuming. Third, vector instructions are processor specific, so using them requires precise knowledge and reduces portability.

4.2 Generating Vectorized Decoders: Overview

Fig. 7 gives an overview of our approach to generating a vectorized Viterbi decoder. The user specifies as input the convolutional code (i.e., the rate $1/N$, K, the polynomials p_ℓ and the frame length F) and the vector length ν. The generator outputs a ν-way vectorized Viterbi decoder, implemented using SSE.

As shown in Fig. 7, first, the generator instantiates the appropriate VL expression (4) and generates a scalar decoder (Section 3). This decoder is then executed to obtain the normalization factors needed to prevent overflows (Section 4.4). The ν-way vectorization is then performed by first rewriting the scalar

VL algorithm. The result is then compiled into source code using the previous template matching system. Finally, a peephole optimizer ensures the features available in the instruction set are fully exploited. The resulting implementation of the forward pass is inserted into a generic framework to obtain the complete decoder.

Our implementation targets 4 to 16-way Intel SSE but the generic principles behind the generator makes it easy to retarget to another instruction set.

4.3 Vectorizing Rewriting System

In this section we explain how to automatically vectorize the forward pass of the Viterbi decoding using VL. First we identify some VL expressions, called vector base cases, that can directly be mapped into vector code. Vectorization is then achieved by rewriting the forward pass algorithm (equation (4)) into an equivalent VL expression that consists exclusively of vector base cases. This expression is then mapped into vector code and further optimizations are performed. The details are explained next.

Base cases. For the vectorization of the Viterbi algorithm, three types of base cases are required.

- One construct that can be implemented with all ν-way short vector instruction sets is $C \otimes I_\nu$ with C being any side-effect free VL operator. When implementing $C \otimes \mathrm{I}_\nu$, the template system first implements C with its corresponding scalar template (Table 2) and then replaces all scalar variables and scalar operations inside the code by their ν-way vector counterparts. For instance, L^4_2 is a permutation of four elements whereas $\mathrm{L}^4_2 \otimes \mathrm{I}_\nu$ is a permutation of four vectors of ν elements each. To denote that the construct is a base case, we write $C \bar{\otimes} I_\nu$.
- The Viterbi kernel has side effects and thus does not fall into the previous category. We denote by $\vec{B}^\nu_{i,l}$ the vector code that executes ν Viterbi kernels over contiguous elements.
- Another class of vector base cases is the perfect shuffle of 2ν elements, written as $\vec{\mathrm{L}}^{2\nu}_\nu$. We use the method in [16] to automatically generate efficient vector code for these permutations from the definition of the instruction sets.

Vector tags. The vectorization subsystem first *tags* a given VL expression with the vector length ν, which is denoted like this: $\underline{A}_\nu$. The full expression is then rewritten using algorithms, manipulation and tag propagation rules until all tags disappear. At this point, the expression only consists of vector base cases that can be implemented using the template system. The same approach has been successfully applied with linear transforms for vectorization [9] and parallelization[3] [8].

[3] This paper does not handle the parallelization of the Viterbi algorithm because, in traditional settings, it is not relevant. Even in multi-core systems, the cost of exchanging data over the interconnect is too high to split the trellis handling over multiple cores. It is more practical to parallelize by assigning different frames to different cores.

Table 3: Manipulation (left) and vectorization (right) rules. C and D are generic operators optionally parametrized by their superscript.

$$\mathrm{I}_{mk} \otimes_j C^j = \mathrm{I}_m \otimes_{j_1} (\mathrm{I}_k \otimes_{j_2} C^{j_1 k + j_2})$$
$$\mathrm{L}^{kmn}_{km} = (\mathrm{I}_k \otimes \mathrm{L}^{mn}_m)(\mathrm{L}^{kn}_k \otimes \mathrm{I}_m)$$
$$(\mathrm{I}_m \otimes C)(\mathrm{I}_m \otimes D) = (\mathrm{I}_m \otimes CD)$$
$$(\mathrm{I}_m \otimes_j C^j_n)\, \mathrm{L}^{mn}_m = \mathrm{L}^{mn}_m (C^j_n \otimes_j \mathrm{I}_m)$$

$$\underline{CD}_{\ \nu} \to \underline{C}_{\ \nu}\ \underline{D}_{\ \nu}$$
$$\underline{\prod C}_{\ \nu} \to \prod \underline{C}_{\ \nu}$$
$$\underline{I_m \otimes_j C^j}_{\ \nu} \to I_m \otimes_j \underline{C^j}_{\ \nu}$$
$$\underline{C \otimes \mathrm{I}_\nu}_{\ \nu} \to C \bar{\otimes} \mathrm{I}_\nu$$
$$\underline{B_{i,l\nu+j} \otimes_j \mathrm{I}_\nu}_{\ \nu} \to \vec{B}^\nu_{i,l}$$
$$\underline{\mathrm{L}^{2\nu}_\nu}_{\ \nu} \to \vec{\mathrm{L}}^{2\nu}_\nu$$

Rules. There are three different kinds of rules:

- The algorithms describe how to implement a specification using VL. In this paper, we only use the Viterbi algorithm (4).
- The manipulation rules (Table 3 left) are basic mathematical identities that can be proved from the definitions of the symbols.
- The tag propagation rules (Table 3 right) describe how the tags interact with the other symbols.

Viterbi vectorization. Using all the manipulation rules, the system automatically derives the following equality, which we call the *partial tensor flip*, that holds for any parametrized operator C^j and integers m, n and ν such that ν divides m:

$$(\mathrm{I}_m \otimes_j C^j)\, \mathrm{L}^{mn}_m = \left(\mathrm{I}_{m/\nu} \otimes_{j_1} \mathrm{L}^{n\nu}_\nu \left(C^{j_1\nu + j_2} \otimes_{j_2} \mathrm{I}_\nu\right)\right) (\mathrm{L}^{mn/\nu}_{m/\nu} \otimes \mathrm{I}_\nu) \ .$$

Because of this transformation, the rewriting systems returns the following vectorized form of the algorithm in (4) and consists exclusively of base cases that can be mapped to code as explained above:

$$\underline{\mathbf{F}_{K,F}}_{\ \nu} \to \prod_{i=1}^{F} \left[\left(\mathrm{I}_{2^{K-2}/\nu} \otimes_{j_1} \vec{\mathrm{L}}^{2\nu}_\nu \vec{B}^\nu_{F-i,j_1}\right) (\mathrm{L}^{2^{K-1}/\nu}_{2^{K-2}/\nu} \bar{\otimes} \mathrm{I}_\nu)\right] \ .$$

In words, at each stage, this algorithm first permutes full vectors then iteratively ($i = 1, \ldots, F$) computes $2^{K-2}/\nu$ independent Viterbi butterflies and performs in-vector permutations $\mathrm{L}^{2\nu}_\nu$ on each result. Remember that in the final code, the initial permutation in each i-step is never performed but merged with the subsequent butterfly computations.

Code generation. Using the previously explained template system, code can be generated for the algorithm above. In practice though, an additional pass with a peephole optimizer is inserted to handle the specifics of the instruction set which is unfortunately very irregular.

4.4 Overflows

The path metrics increase on average with the stage number. For implementation however, they must stay within a finite window of representable values. The

precision offered by short vector instructions might not be sufficient to guarantee the absence of overflows so the algorithm must sometimes be slightly modified.

For instance, with Intel's SSE, all vector operations are performed in 128 bits vector registers. Therefore, in 4-way mode, elements are 32-bits long whereas in 16-way mode, elements are 8-bits long. In this last case, metrics are likely to overflow the window of 256 "legal" values.

There are known methods to help rescaling these metrics (see [17]) but they are based on empirical properties of the code which is why we need to first generate a scalar version of the decoder (i.e., in which overflows will not occur) and only then generate the vectorized version once these properties are determined. We will not detail this process further even though it is fully automated.

5 Results

In this section, we analyze the performance of our generated Viterbi decoders. We compare against existing optimized implementations, show the generality of our generator, and show the speedup obtained by vectorization.

Experimental Setup. All experiments are performed on an Intel Core 2 Extreme X9650. All code is compiled using the Intel Compiler (`icc`) 10.1 with performance flags (`-fast -fomit-frame-pointer -fno-alias`). The performance in each case is measured by entirely decoding (forward pass and traceback) multiple frames. Initialization and precomputation (one time costs) are excluded.

Our generator supports any valid combination of rate, polynomials, frame length, and constraint length $K \geq 6$[4]. Vectorization is available for all convolutional codes and for processors that are SSE-compatible through 4-way, 8-way and 16-way intrinsics.

Benchmarking. We first compare our generated decoders against Karn's hand-written decoders [4]. Karn's forward error correction software supports four codes ($1/2$, $K = 7$ nicknamed "Voyager", $1/2$, $K = 9$, $1/3$, $K = 9$ and $1/6$, $K = 15$ nicknamed "Cassini") available for different vector lengths. Not all vector lengths are supported for all codes. The forward pass in [4] is written separately in assembly for each combination of code and vector length.

In Fig. 8, we show the performance results for these four codes and for all vector lengths. A missing bar signifies that the implementation is not provided by Karn. Analysis of the plots shows that our generated decoders have roughly equal performance compared to Karn's software.

Performance of supported codes. To show the generality of our generator and the consistent performance, we generated decoders for known "good" codes (see [18]) of rate $1/2$ collected and all four vector lengths 1, 4, 8, 16. Fig. 9a shows the performance results and, as expected, the lines show the exponential decay in performance when the constraint length increases. Similar graphs are observed for other rates.

[4] The limitation is an artifact of the actual implementation. The methods presented in this paper are applicable for all constraint lengths.

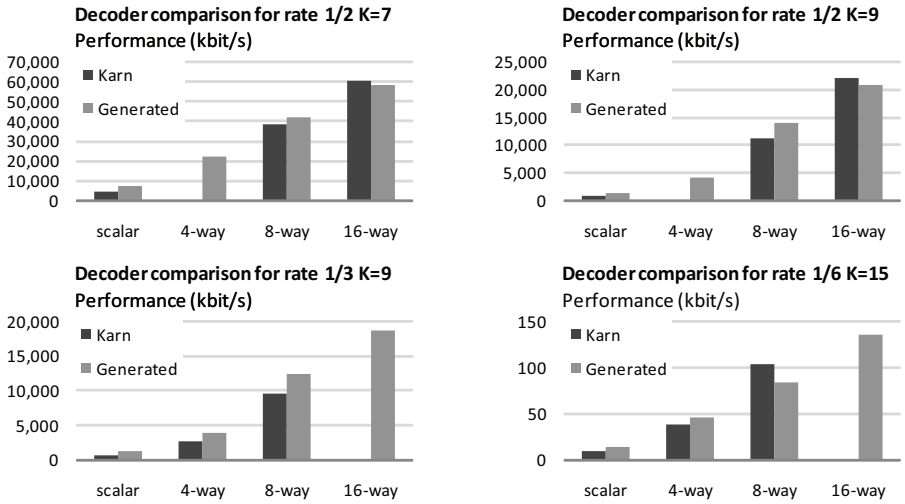


Fig. 8: Performance comparison between the generated and hand-optimized decoders.

Quality of Vectorization. Fig. 9b shows the speedup achieved by vectorization in Fig. 9a. The baselines are the non-vectorized scalar decoders. We observe a consistent speedup of about 3.5 for 4-way, 6 for 8-way, and 10 for 16-way vectorization. The smaller gains for longer vectors is expected since they require more involved shuffle operations. The peak for both 16-way and 8-way with short constraint length is caused by the reduction of the memory footprint due to the use of shorter data types.

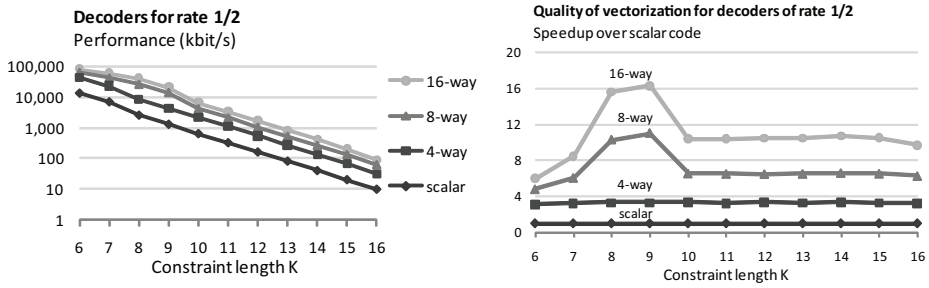
Code Generation Time. Note that, in the previous graphs, we present 40 different optimized decoders, all of which were generated in less than one hour. We estimate that it would take an expert more than a day to produce each one of the forward passes in assembly, which implies an improvement in the development time in the range of three orders of magnitude.

6 Conclusion

We presented a framework and its implementation that completely automates the implementation of fast software Viterbi decoders for modern computing platforms with SIMD instruction sets by generating the performance critical forward pass. The basic idea is to construct a domain specific mathematical language to express the forward pass, to vectorize by rewriting in this language, and to use a backend for low level optimizations. The same approach could be used for parallelization but it is more efficient (and trivial) to parallelize across frames.

Our framework enables the instant generation of any decoder across a wide spectrum of parameters. The generated decoders' performance is on-par with specialized expert implementations. Further, it enables fast porting to new architectures as only small changes are needed to support a new instruction set.

We invite the reader to visit the online interface to our generator at `www.spiral.net/software/viterbi.html`.



(a) Performance (b) Normalized perf. over scalar code

Fig. 9: Performance of various generated decoders for rate 1/2. Note that each point on these graphs actually replace one manually tuned assembly code.

References

1. Viterbi, A.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. Information Theory, IEEE Transactions on **13**(2) (Apr 1967)
2. Gemmeke, T., Gansen, M., Noll, T.: Implementation of scalable power and area efficient high-throughput viterbi decoders. Solid-State Circuits **37**(7) (Jul 2002)
3. Mitola III, J.: Software Radio Architecture. John Wiley & Sons (2002)
4. Karn, P.: FEC library version 3.0.1, `http://www.ka9q.net/code/fec/` (Aug 2007)
5. Van Loan, C.: Computational frameworks for the fast Fourier transform. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1992)
6. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., et al.: SPIRAL: Code generation for DSP transforms. Proc. of the IEEE **93**(2) (2005)
7. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: a language and compiler for DSP algorithms. SIGPLAN Not. **36**(5) (2001) 298–308
8. Franchetti, F., Voronenko, Y., Püschel, M.: FFT program generation for shared memory: SMP and multicore. In: Supercomputing (SC). (2006)
9. Franchetti, F., Voronenko, Y., Püschel, M.: A rewriting system for the vectorization of signal transforms. In: High Performance Computing for Computational Science (VECPAR). Volume 4395 of LNCS., Springer (2006) 363–377
10. Forney, G.D., J.: The viterbi algorithm. Proc. of the IEEE **61**(3) (March 1973)
11. Rader, C.: Memory management in a viterbi decoder. Communications, IEEE Transactions on [legacy, pre - 1988] **29**(9) (Sep 1981) 1399–1401
12. Franchetti, F., de Mesmay, F., McFarlin, D., Püschel, M.: Operator language: A program generation framework for fast kernels. In: IFIP Working Conference on Domain Specific Languages (DSL WC). Volume 5658 of LNCS., Springer (2009)
13. Feldman, J., Abou-Faycal, I., Frigo, M.: A fast maximum-likelihood decoder for convolutional codes. Proc. of Vehicular Technology Conference (2002) 371–375
14. Fano, R.: A heuristic discussion of probabilistic decoding. Information Theory, IEEE Transactions on **9**(2) (Apr 1963) 64–74
15. Lawrie, D.: Access and alignment of data in an array processor. Computers, IEEE Transactions on **C-24**(12) (Dec. 1975) 1145–1155
16. Franchetti, F., Püschel, M.: Generating SIMD vectorized permutations. In: Inter. Conf. on Compiler Construction (CC). Volume 4959 of LNCS., Springer (2008)
17. Hekstra, A.: An alternative to metric rescaling in viterbi decoders. Communications, IEEE Transactions on **37**(11) (Nov 1989) 1220–1222
18. Chambers, W.: On good convolutional codes of rate 1/2, 1/3, and 1/4. Singapore ICCS/ISITA '92. 'Communications on the Move' (Nov 1992) 750–754 vol.2