A Programmable Vector Coprocessor Architecture for Wireless Applications

Yuan Lin, Nadev Baron, Hyunseok Lee, Scott Mahlke, Trevor Mudge
Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109-2122
{linyz,nbaron,leehzz,mahlke,tnm}@umich.edu

ABSTRACT

The physical layers of most wireless protocols are traditionally implemented in ASICs due to the heavy computation requirements. These solutions are costly to design and hardwired solutions that offer no post-programmability. In this paper, we introduce a flexible coprocessor architecture customized for wireless protocols. To accomplish the design, a complete baseband physical layer for the 802.11b and Hiperlan2 protocols was developed and the computational characteristics of the time-critical code analyzed. In particular, we studied the behavior of the Viterbi decoding stage and the FIR filter. We propose an fully programmable vector coprocessor architecture that achieves real-time performance for these computation-heavy algorithms. The design consists of a memory streaming macro-pipelined vector architecture that effectively exploits the high degree of data-level parallelism within these algorithms. It offers high performance while maintaining full programmability.

1. INTRODUCTION

With the growing popularity of wireless networking in both the home and office, the wireless communications semiconductor marketplace is an area with huge growth opportunities. Next generation wireless standards are being designed to support a wide variety of multimedia services with very high data rates using different multiplexing and modulation techniques. Seamless switching between multiple protocols, which would allow devices to support and change to other protocols, is highly desirable. Thus, one central challenge in the design of hardware to support these protocols is programmability. At the same time, the hardware must meet strict performance, cost, and power goals to achieve the desired data rates, be economically feasible, and be usable in mobile platforms.

Traditional hardware systems for wireless communications are designed using a mixture of DSPs, and ASICs. The heterogeneous nature of these solutions make them difficult to design and verify. Further, mapping each protocol onto these solutions is tedious and error-prone as each component is programmed or designed separately. While operations at the Media Access Control (MAC) layer are typically of millisecond levels, operations of the Physical (PHY) layer are of microsecond levels. This makes it very hard to perform the PHY layer algorithms on today's DSPs or even FPGAs in real-time. More recent wireless platforms including the Imagine processor [1] [19], PICO Chip [9], and HiperSonic [12]. Most of these processors fall into two categories: 1) General purpose procesors for cellular protocols, which are less computationally intensive than wireless LAN protocols; or 2) Integrated ASIC and DSP solutions for wireless LAN.

In this paper, we propose a coprocessor architecture that has been customized to the computation requirements of wireless protocols. In contrast to recently proposed solutions, a coprocessor model is adopted that can be integrated with a conventional general purpose processor, such as an ARM. A programmable vector computation engine is chosen as the underlying architecture for the coprocessor. The coprocessor design consists of a pipeline of vector engines in which data is streamed through the pipeline. The vector architecture effectively exploits the data parallelism within the application to achieve scalable performance, while still maintaining a high degree of programmability.

This paper represents a work-in-progress snapshot of our current design. Our work is based on a homegrown wireless testbed that is being developed and analyzed to understand the computational structure of the varying wireless protocols. The paper begins with an overview of the testbed, followed by a description of the coprocessor architecture together with some of the issues that arise when mapping two of the key computation kernels (FIR and viterbi) onto the coprocessor. Finally, we present some preliminary experimental data on the effectiveness of the architecture at meeting the real-time performance goal.

2. WIRELESS TESTBED

2.1 Testbed Flows

The testbed consists of two wireless protocols, each uses different modulation and multiplexing technique. The first is the IEEE 802.11b[7] which uses Direct Sequence Spread Spectrum (DSSS) and Complementary Code Keying (CCK) modulation. The second protocol is the European HiperLAN/2 which uses Orthogonal Frequency Division Multiplexing (OFDM) and rectangular Quadratic Amplitude Modulation (QAM).

Figure 1. shows a system level diagram of the IEEE 802.11b Simulink model. The Physical Layer Service Data Unit (PSDU) is generated with random bits. Next, the Physical Layer Convergence Protocol (PLCP) forms the PLCP protocol data unit (PPDU) by adding preamble header information to each PSDU. Once the PPDU is formed it is modulated and spread using CCK. Finally, before white Gaussian noise is added, the chips are upsampled and filtered using FIR interpolation and then mixed to channel

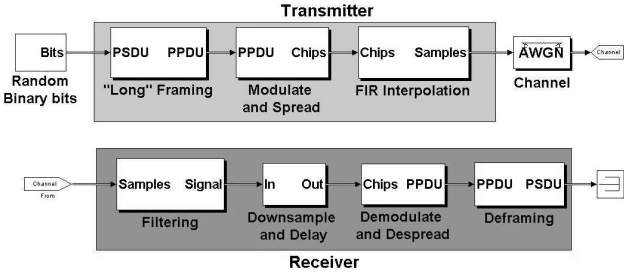

Figure 1: IEEE 802.11b Simulink Model

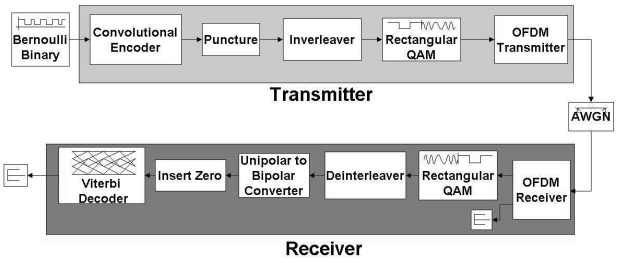

Figure 2: HiperLAN/2 Simulink Model

frequency,
At the receiver part, the same basic algorithms are performed, however in reverse order. Samples are moved back to baseband, filtered and downsampled. Data is demodulated and despread, and the header and preamble information are separated from the PSDU.

Figure 2. shows the system level diagram of the HiperLAN/2 Simulink model. Random packets of bits are encoded using a Convolutional encoder with code rate of 1/2. The encoded bits are punctured with a 4/6 P2 puncturing rate and are passed to an interleaver. Once interleaving is done, the data is passed to a 16-ary rectangular QAM modulator and finally to an OFDM transmitter where pilots are added to the data and Inverse Fast Fourier Transform (IFFT) is performed. The receiver part performs a Fast Fourier Transform (FFT), separates the pilots, modulate and decode the information using a Viterbi decoder.

Complete end to end PHY layer models of those two protocols were built in Simulink. From this stand alone C code was produced directly.

2.2 Computation Requirements

Both HiperLAN/2 and the IEEE 802.11b[6] protocols can operate at different modes and rate and with different modulation techniques. Higher data rates usually require more complex and computationally intense algorithms which traditionally could only be implemented using hard coded devices.

Figure 3 shows a system level profiling of three protocols; IEEE 802.11 - 1Mbps, IEEE 802.11b - 11Mbps and HiperLAN/2 - 36Mbps. It shows the runtime computation distribution of various functions in the protocols. Out of all of the profiled functions, many are not shown in the diagram. They are grouped under 'Others' because they contribute to less the 2 percent of the total running time. Figure 3 shows that for the IEEE 802.11 at both 1Mbps and 11Mbps, the most computation-heavy algorithm is the DF2T Filter, which requires about 25 - 45 percent of the overall computation time. Moreover, FIR interpolation function, shown in Figure 3 in plain white, requires about 5 percent, and together with the DF2T filter sum up to about 50 percent. For the HiperLAN/2 protocol, on the other hand, the most computation-heavy algorithm is the Viterbi decoder, which takes more than 30 percent of overall run time. Although generation of the channel frequency and mixing to the channel frequencies (Generate Channel algorithms and Mix to Channel as shown in the Figure 3) are also very computationally intensive algorithms they were not addressed because they can be replaced by using the very efficient Direct Digital Synthesizer (DDS).

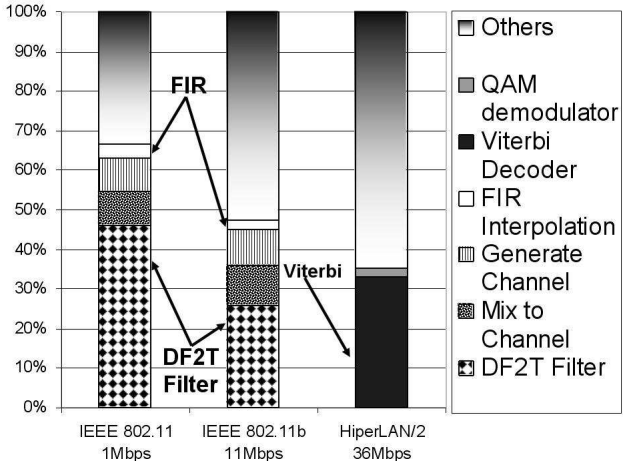

Figure 3: System Level Profiling

	Numbers of calls	Samples per call	Total samples per second
IEEE 802.11 1Mbps	960	9,224	88,535,040
IEEE 802.11b 11Mbps	20,993	10,304	216,311,872

Table 1: One second of DF2T Filter

Tables 1, 2 and 3 show real-time requirements, in terms of samples per second, for the DF2T, FIR Interpolator and, in terms of bits per second, for the Viterbi decoder. Each sample can be represented either as an integer or floating point number, depending on implementation. In order to achieve real-time requirements for both the IEEE 802.11 and the IEEE 802.11b, we require an IIR Direct Form II Transpose (DF2T) filter with order of 84 that can operate at about 200Mps (200 million samples per second). In addition, real-time HiperLAN/2 implementation requires a Viterbi decoder with k=7 capable of operating at about 70Mbps. FIR interpolation with order 84 and interpolator factor of 8, is yet another algorithm which both the IEEE 802.11 and the IEEE 802.11b spend much time in, and needs to be preformed at about 100Mps. Any reconfigurable architecture capable of supporting those protocols, must satisfy these filters and Viterbi decoder real-time requirements.

The following two sections provide an overview of the two algorithms that are the most computationally intensive in wireless protocols.

	Numbers of calls	Samples per call	Total Samples per second
IEEE 802.11 1Mbps	1,080	9,224	99,601,920
IEEE 802.11b 11Mbps	9,620	10,304	99,124,480

Table 2: One second of FIR Interpolation

	Numbers of calls	Bits per call	Total bits per second
HiperLAN/2 36Mbps	250010	288	72,002,880

Table 3: One second of Viterbi Decoder

2.3 Viterbi Algorithm

Viterbi algorithm is a decoding method for the convolutional code that is frequently used in digital wireless communication systems such as wireless LAN. From the received bit sequence contaminated by channel noise, we can recover the original bit sequence with minimum error by the use of convolution and the Viterbi algorithm. Whereas several flip flops and adder are sufficient to realize the convolutional encoder, the decoding procedure requires highly intensive computation, because the decoding procedure requires that an optimum code sequence be found that minimizes the error probability from all possible code sequences. [18]

In Viterbi algorithm, it is possible to classify the operations into three steps, BMC (Branch Metric Calculation), ACS (Add Compare Selection) and Back Tracking (BT). BT is also called Survival Path Tracking(SPT). Generally, the error cost of all available code sequences are calculated at the BMC step by the bit level comparison between input sequence and all sequences in the candidate code set. The candidate code set is refined at the ACS step by accumulating error cost and selecting the local optimal code sequence. The BT is the final step to find a globally optimal code sequence that minimize error probability. The localized data dependency of the BMC and ACS steps means they can be easily parallelized and pipelined.

2.4 Digital Filters

The function of a filter is to selectively bypass input signal terms within specific frequency range. The unwanted signals like noise and other adjacent frequency terms inducing distortion are suppressed by the filter. In order to perform filtering operation over a digital signal we use two types of digital filter, finite impulse response(FIR) and infinite impulse response(IIR) filters. The IIR filter has a much sharper cut off characteristic for a given filter order. The FIR filter has a better linear phase response characteristic. In Wireless LAN system, the FIR filter is used for modulating baseband signal into the specific channel frequency. It filters out the undesired imaginary term and harmonics term generated by the upsampling procedure. The role of the IIR filter in the Wireless LAN receiver is to demodulate the signal in a desired channel band. The basic operations of these filters are multiplication and summation [22].

3. CO-PROCESSOR SYSTEM AND ARCHITECTURE

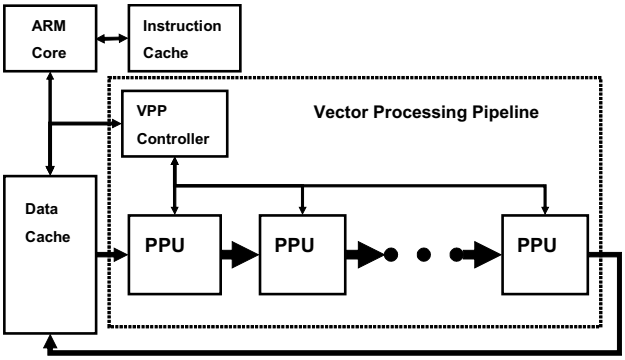

Figure 4: VPP Microarchitecture

This section introduces a programmable vector coprocessor architecture VPP (Vector Processing Pipeline). It consists of three parts: VPP programming model, the microarchitecture overview, and the algorithm mappings.

3.1 VPP Programming Model

Wireless protocols can be categorized as streaming applications because they fit the producer-consumer model and have a high degree of data parallelism. Data streams in, gets processed by various signal processing algorithms, and streams out. VPP architecture exploits these characteristics by mapping applications onto a macro-pipeline, where each stage of the pipeline executes one major step (often a function) in the application. To support this architecture, two instruction set architectures (ISAs) are required. The first describes the overall macro-pipeline behavior (macro ISA), and the second describes each individual pipeline stage actions (micro ISA). This is very similar to the Imagine Stream Architecture [1], VIRAM vector architecture [14], as well as various tiled processor architectures [3] [17]. The key difference is that our architecture is a rolled-out simple feed-forward macro pipeline. We do not have any routing network between streaming stages, making the architecture and ISAs simpler.

The macro ISA is implemented as a set of coprocessor instructions on the host processor (ARM-10 in our design). They are simple scalar instructions. The micro ISA is implemented as vector SIMD style instructions. This architecture fits very well with the wireless applications. For example, as mentioned in previous sections, the Viterbi decoding algorithm consists of three steps: branch metric calculation, add-compare-select, and backtracking. This is mapped onto three macro pipeline stages, one for each step. The macro ISA initializes each macro-pipeline stage and controls data streams to and from memory. The micro ISA describes the actual functionalities of these three steps on each macro pipeline stage.

3.2 Microarchitecture Overview

Figure 4 presents the overall coprocessor architecture. The VPP consists of two major parts: the VPP controller and a set of Pipeline Processing Units (PPUs). Both structures are explained in detail in sections below. Overall, this is a stream architecture. Data are fetched from the data cache directly into the macro pipeline. After computation, data are stored directly back into memory from the pipeline. The

VPP controller interacts with host processor's core and controls flow of data through the PPUs. It contains registers to keep internal states.

PPUs are vector processing stages of the macro pipeline. There is no cache structure in each PPU, because of the streaming nature of the applications. Instead, each PPU contains an instruction buffer, a vector register file, an input queue and an output queue. PPUs are connected with buses between them. Each PPU can only receive data from the PPU before it, and send data to the PPU after it. The first and last PPUs read and write to the host data cache directly. We can implement fast and high bandwidth data bus between PPUs because there is no routing network, each PPU only has one source and one drain. The number of PPUs in the macro pipeline is dependent on the application characteristics and desired performance.

3.2.1 Coprocessor Interface

The proposed coprocessor architecture is a general architectural platform that can be applied to any host processor with coprocessor support. For this study, we implemented it as an ARM-10 coprocessor. The VPP controller communicates with the ARM through standard ARM coprocessor interface [21]. When the ARM encounters a coprocessor instruction, it dispatches the instruction to VPP. The instruction is then further decoded and executed by the VPP controller. When the macro instruction finishes execution, an acknowledgment is send back to the ARM core processor.

The macro ISA is implemented with ARM coprocessor instructions. There are two basic types of instructions: coprocessor initialization and coprocessor execution. Initialization instructions load application code into the instruction buffers of the PPUs, reset the internal states of the PPUs, and set register with the memory addresses in data cache for stream data. Execution instructions start macro pipeline execution. One of these execution instructions can potentially run for thousands of cycles. To keep program consistency, the ARM core stalls its own pipeline until these coprocessor instructions finish execution.

3.2.2 System Architecture

The VPP communicates with ARM code through the VPP controller. This is a very simple structure. It consists of registers to keep track of the streaming data's memory addresses. It is the only structure that can access each individual PPU pipeline stage. The purpose is to upload application code into the instruction buffers of the PPUs, as well as some constant values into the data buffers of the PPUs. For a given application, we only need to initialize the PPU macro pipeline once for it to receive and process incoming streaming data. The time requirements for initialization are minimal. Therefore, the bus bandwidth connecting VPP controller and PPUs can be relatively small because it does not affect overall performance.

The VPP controller is responsible for loading data from the data cache, and storing data out to the data cache. The memory addresses are stored in the internal registers of the VPP controller. The data bus between the data cache and PPU cannot be as high a bandwidth as the internal bus between PPUs, resulting in greater memory fetch latency. This latency can be mostly hidden away through careful macro-pipeline workload balancing.

The VPP controller is also responsible for data movement between PPUs. In every clock cycle, it checks the output queue of each PPU stage for new data. If there is new data in stage i, it then checks if the input queue in stage $i+1$ is full. If it is not full, the VPP controller then transfers the data over. Conversely, if the input queue is full, then the controller stalls stages $0 \ldots i$. It then proceeds with the data transfer.

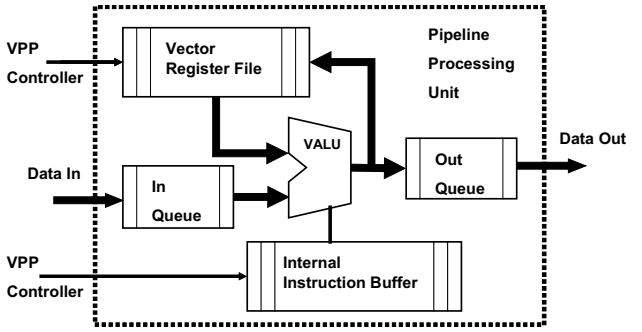

Figure 5: PPU architecture

3.2.3 Pipeline Processing Unit (PPU) Architecture

Each PPU is a simple vector processor as shown in Figure 5. A PPU consists of four different fully vectorized storage elements: input queue (IQ), output queue (OQ), vector data register file (VRF), and instruction buffer (IB). Because none of these memory structures are caches, we can implement high degree of vectorization very efficiently. The VRF consists of 64 register vectors. We find that this size is sufficient for all applications studied so far. One of the vector registers can be also used as an array of scalar registers, while the rest can only be accessed at the vector boundary. Both the IQ and OQ also consist of arrays of register vectors. The size of the IQ and OQ are heavily dependent on the macro-pipeline workload. If the pipeline stages are completely balanced, then theoretically we do not need these queue structures. However, most algorithms cannot be perfectly pipelined into stages. In general, larger queues help alleviate these bottlenecks. In the applications studied, we found that relatively small queue sizes are sufficient enough. Queue sizes are evaluated in more detail in Section 4.

The VALU is a fully vectorized ALU. It can do up to 64 integer or floating point operations in parallel. It supports the most common signal processing operations, including multiply-and-accumulate (MAC). However, it does not support special algorithm-specific operations like the butterfly operation which is common on most ASIC implementations of Viterbi algorithms [15] [20] [4]. Instead, the butterfly operation is done by combination of multiple element vector instructions. Because of the wide collection of signal processing algorithms in current wireless protocols, we choose this approach, though less efficient, so that the architecture remains general-purpose enough to support multiple algorithms and protocols.

3.2.4 PPU Vector Microarchitecture ISA

As mentioned previously, the PPU supports an internal vector microarchitecture ISA. It supports many common vector instructions, including predicated vector execution, vector element permutation, vector reduction and expansion, and loop counters.

This architecture implements a simple 1-level masking. Masked vector execution has be implemented in numerous vector processors in the past, including the original CRAY-1 [8]. To accomplish masked execution, each PPU also has a mask vector. Whenever a vector comparison is done, the results are stored in the mask vector. Each instruction in the IB has a conditional field, which is used to conditionally execute elements of the vector if the conditional field matches the corresponding values in the mask vector. These instructions are used to implement simple 1-level if-then-else, instead of more time consuming and inefficient branch instructions. Multi-level if-then-else conditionals still require branch instructions. However, in the wireless applications we've studied, multi-level if-then-else statements are very rare, making 1-level masking a good design decision.

Vector permutation instructions are similar to those implemented in VIRAM architecture [13]. These instructions split the elements from one vector into two vectors. For further detail on exact operations, please refer [13]. These instructions are used to implement butterfly operation of Viterbi decoding algorithm.

Vector reduction instructions reduce a vector into a scalar value. These instructions are relatively slow, because it takes at best O(log(V)) time to reduce a vector of size V. They are included because they are useful for many signal processing applications. Fortunately, for the applications we studied, such operations do not come up very often. Vector expansion instructions expand a scalar value into a vector and are used in a variety of signal processing algorithms.

1-level loop counter instructions are also implemented in this architecture. They consist of two instructions: a loop header instruction to set the loop counter, and a loop tail instruction to check loop count. Through behavioral characterization, we found that most wireless applications have very regular control flow. Most branches are results of 1-level loop iterations. Therefore, we only implemented simple branch and jump instructions, with no hardware branch predictors. Most of the control flow can be mapped into loops and masked execution.

3.3 Mapping Algorithms to Architecture

For this study, we mapped the three most computationally intensive signal processing algorithms onto our proposed pipeline architecture. These algorithms are normally implemented with ASICs in most commercial products.

3.3.1 *Viterbi Decoding Algorithm*

The Viterbi algorithm is a maximum likelihood state tracing algorithm. It operates on a trellis state diagram. Figure 6 shows the general diagram for an 8 state trellis. Each vertical column represents the possible states at different times. The edges describe the possible state transitions. The convolution encoder encodes data bits with these state transitions. The gray circles and bold edges in the figure represent the correct state transitions that the encoder took to encode the data bits. The Viterbi decoder's job is to recover this exact path based on the encoded data, by calculating the most likely transitioning edge at each time step. It does this by calculating likelihood costs of every possible state transition at every time step. And at the end of the trellis diagram, it traces back from the most likely state to find the transition trace. Please refer to [10] for further detailed explanations of the algorithm.

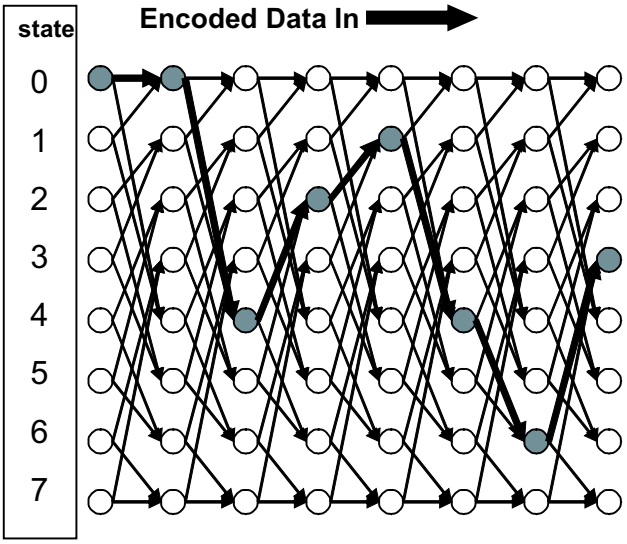

Figure 6: Viterbi Algorithm Example

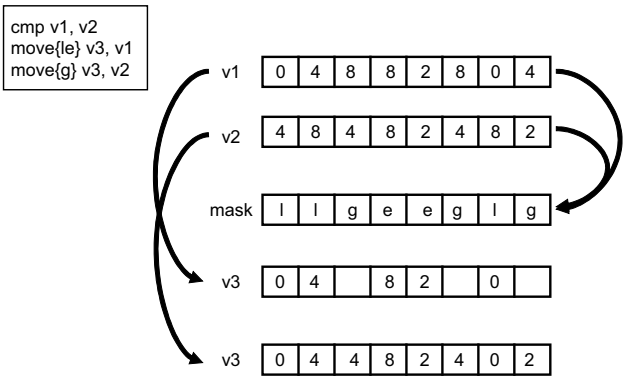

Figure 7: Conditional Move

The implementation of the Viterbi decoding algorithm consists of three steps: Branch Metric Calculation(BMC), Add-Compare-Select (ACS), and Back Tracking (BT). For this study, each step is mapped onto one PPU. We implemented the Viterbi Algorithm for 1/2 convolution encoding rate, with K = 7. This uses a 64-state trellis transition diagram. We used this setup because it is the one defined and used in the Hiperlan2 specifications [16]. For this setup, each column in the trellis diagram corresponds to 2 bits of encoded data, and 1 bit of decoded data.

The BMC stage is responsible for calculating the likelihood costs associated with every out-going transition edge for the current state column. This can be vectorized very easily because edge cost calculations are independent of each other. This stage outputs two vectors of size 64 to the second stage, ACS. These two vectors contain all 128 transition edge values.

The ACS stage calculates the cost values for the next column of 64 states based on the 64 current state values and the 128 transition edge values. It adds the current state value to these transition edge values. Because there are two input edges for every state, it then compares and chooses the lower cost as the next state value for all 64 next states.

In a conventional architecture, we would then need to iterate through all 128 transition edge values, and then iterate 64 times to find the values of next states. For our architecture, through the use of masked vector execution, we can do such an operation very efficiently. Figure 7 shows the general idea for this operation. V1 and V2 are the two potential next state vector values, mask is the mask vector, and V3 is the nextstate value. A compare instruction is first performed to set the mask vector. In Figure 7, 'l' is for less-than, 'e' is for equal, and 'g' is for greater-than. V1 and V2 are conditionally moved into V3 based on the result of the comparison. This takes three steps on a 64-wide PPU, whereas it takes hundreds of cycles on a conventional processor. With this architectural mapping, the ACS step sends out a vector of 64 to the final stage (BT) containing next state cost values. Because this step is the most computationally intensive stage, it is mapped onto two PPUs.

The BT stage traces back from the end of trellis diagram to find the maximum likelihood transition. This stage requires the values of all states in the trellis diagram. Unfortunately, this step is highly serial. To map to the coprocessor architecture, BT receives the whole trellis diagram values from ACS. It then iterates through the diagram backward, producing one data bit per iteration. After it iterates through the whole trellis diagram, it then outputs the decoded data vector. This stage by nature cannot take advantage of PPU's vector architecture. However, because the algorithm is pipelined, we can still hide some of the latency of BT stage. This shows another advantage of this macro pipeline architecture.

3.3.2 Filters

We also mapped the DF2T filter onto this architecture. Unlike the Viterbi algorithm, filters are relatively simple algorithms. Most filter algorithms consist of simple 1-level loops, making loop counters particularly useful. In addition, MAC operations are very common. Thus, PPUs also support vector MAC operations. We mapped this algorithm onto three PPUs. The first PPU reads in the inputs and spread each bit of the input into its own vector. The second PPU adds the inputs to the internal filter states. The third PPU calculates the final output values based on the inputs and internal filter states. Output is then streamed out of the VPP, and back into memory.

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

Both the wireless protocol physical layers studied in this paper are build in Matlab with Simulink. The Matlab code is then converted into C code for program behavioral characterization, profiling, and hotspot identification. For the architectural study, we used ARM Development Suite as our baseline model [2]. We build a behavior coprocessor simulator which interfaces with the ARM simulator core. For this study, we assume a 500 MHZ operating frequency.

4.2 Overall Performance

We used Matlab generated C code for profiling purposes only. We found out that the C code itself is very inefficient. Using the internal vector ISA, we hand-coded the Viterbi decoding algorithm and DF2T filter. Figure 8 shows our simulation results. The top two graphs are the performance results for Viterbi and DF2T filter while varying PPU's ALU vector width. The ALU vector width defines how many data operations to execute in parallel. The middle two graphs are the performance results while varying cache memory peak bandwidth. Memory peak bandwidth is the transfer rate of data from cache to PPU. The bottom two graphs are the performance results of varying the In-queue and Out-queue size, for different memory bandwidth. For our results, Viterbi Algorithm's performance metric is bits-per-second. For DF2T filter, the performance metric is sample-per-second.

From our experimental results, we determined that the optimal architecture configuration is: 64 wide vector ALU; 8 entries In-queue and Out-queue; 500MBps memory bandwidth; 64-entry vector register file; and 64 entry instruction buffer. Viterbi decoder is mapped onto four PPUs, and the DF2T filter is mapped onto three PPUs. Using this as our prototype architecture, we achieved the following results. For the Viterbi decoder, we were able to achieve nearly 40Mbps. This meets the real-time processing requirement for the 802.11b protocol. A similar Viterbi algorithm running on an ARM-10 processor can achieve around 10Kbps of data throughput. We have achieved a respectable performance compared to ASICs while maintaining a level of programmability. For the DF2T filter, we achieved near 100Mps(samples per second). This is still 5-6x off from real-time performance, as 802.11b and Hiperlan2 require around 160Mps to 200Mps. However, the overall results are still very promising as there are a number of algorithmic improvements that can be employed to parallelize Viterbi or more efficiently realize the filters [5] [11]. We examine the performance in more detail in the remainder of this section.

4.3 Degree of Vectorization

From Figure 8, we see a near linear relationship with the degree of vectorization for both the Viterbi decoder and the filter algorithm. The best performances are achieved with 64-wide vectors. This is partially because the implementation of our Viterbi algorithm has 64 internal states, making it a good fit for 64-wide vector operation. The filter also has high degree of data parallelism. Running a 64-wide vector on a 32-wide ALU requires two operations and additional overhead. One alternative to a high degree of vectorization is to have a deeper macro pipeline. Given the flexibility of the architecture, we are not limited to algorithms with a certain degree of vectorization. Some initial study with different Viterbi configurations suggested that we can achieve around 5Mbps for K=9 (256 states), and 10Mbps for K=8 (128 states). They key point is that the architecture itself is flexible, it is up to the compilers and programmers to efficiently utilize it.

4.4 Memory Bandwidth

Because this is a streaming architecture, we need to examine the effects of memory bandwidth on the architecture. From Figure 8, we can see that memory is not a major bottleneck in this architecture for both Viterbi and the filter. This is because both algorithms are inherently computationally intensive. They both require so much processing power that memory latency costs can be hidden using queues. To maintain top performance, the Viterbi decoder requires around 300MBps, and DF2T filter requires between 500 MBps and 1GBps. This memory bandwidth requirement is reasonable given modern stream memory architec-

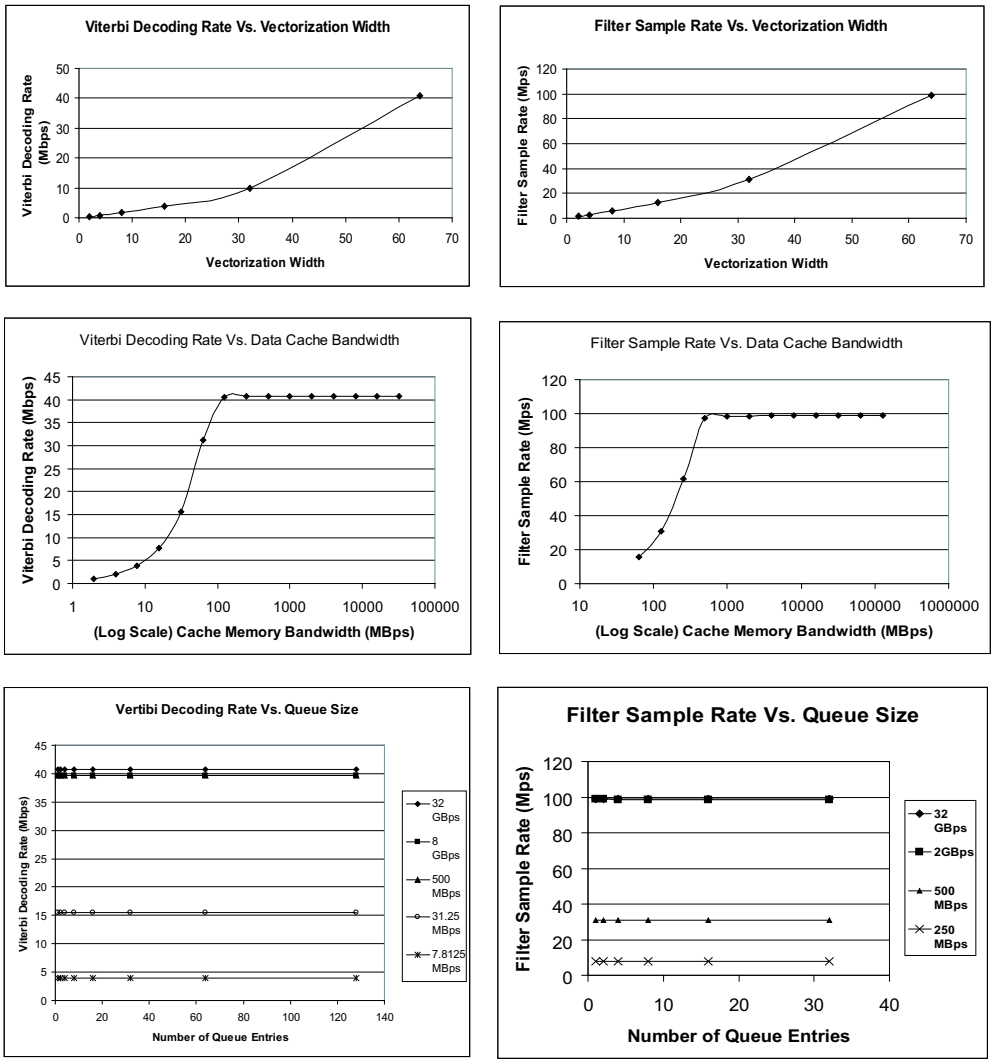

Figure 8: Viterbi Decoder Results

tures. It should be noted that this memory bandwidth is peak bandwidth. Average bandwidth is even lower for these benchmarks. For a 1/2 rate 11Mbps Viterbi decoder, it only needs 22Mbps of average memory bandwidth. This kind of application fits well on a coprocessor architecture, because a typical coprocessor cannot get high average memory bandwidth from the cache because it may be serving other processors.

4.5 Queue Size

Each PPU has an input queue and an output queue. In this study, we also examine the effect of size of the queue to overall architectural behavior. If we have perfect workload balance between macro pipeline stages, queues at the input and output of each PPU would be unneccessary. Therefore, examining the effect of queue size is an effective way to determine the quality of our application mapping. Figure 8 shows the performance for different queue sizes for different memory bandwidth. Both the filter and Viterbi decoder are hand coded. The computational load for each PPU in the pipeline is carefully balanced for maximum throughput and efficiency. For both of these algorithms, we do not need more then 2 or 3 queue entries. If we were to mapp Viterbi algorithm on three PPUs instead of four, with ACS stage being the most computationally intensive stage, we would see benefits of having a larger queue.

5. CONCLUSION

In this paper, we propose a programmable coprocessor architecture that is customized to the critical computation found in wireless applications. The design consists of a pipeline of vector engines through which data is streamed. Each vector engine is fully programmable and efficiently exploits the large levels of data parallelism found in the applications.

A preliminary evaluation of the coprocessor architecture was performed on two important components of the 802.11 and Hiperlan2 protocols. For the Viterbi decoder, we were able to achieve around 40Mbps. This is close to real-time processing requirement for 802.11b protocols. A similar

Viterbi algorithm running on an ARM-10 processor can achieve around 10Kbps of data throughput. The coprocessor achieved respectable performances compared to ASICs, while maintaining the flexibility of general purpose processors. For the DF2T filter, we achieved around 100Mps(samples per second). This is still 2x off real-time performances, as 802.11b and Hiperlan2 requires around 160Mps and 200Mps. However, the overall results are still very promising. We have yet to explore the benefits of any algorithmic improvements as the code we studied was directly generated from Matlab without optimizations. There has been substantial research on parallelizing and more efficient filter design, which will be examined as part of our future research.

6. REFERENCES

[1] J. H. Ahn et al. Evaluating the imagine stream architecture. In *ISCA*, Jun. 2004.

[2] ARM. Developer suite. Version 1.2.

[3] M. Bedford et al. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA*, Jun. 2004.

[4] M. A. Bickerstaff et al. A unified turbo/viterbi channel decoder for 3gpp mobile wireless in 0.18um cmos. *IEEE Journal of Solid-State Circuits*, Nov. 2002.

[5] P. J. Black and T. H. Meng. A 140-mb/s, 32-state, radix-4 viterbi decoder. *IEEE Journal of Solid-State Circuits*, Dec. 1992.

[6] L. M. S. Committee. Ansi/ieee std 802.11, 1999 edition, part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications.

[7] L. M. S. Committee. Ansi/ieee std 802.11, 1999 edition, part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. higher-speed physical layer extension in the 2.4 ghz band.

[8] Cray Research Inc. *The Cray-1 Computer System*, Publication No.2240008b 1976.

[9] A. Duller, G. Panesar, and D. Towner. Parallel processing - the picochip way. In *Communicating Process Architectures*, 2003.

[10] G. Fettweis and H. Meyr. High-speed parallel viterbi decoding: Algorithm and vlsi-architecture. *IEEE Communication Magazine*, May 1991.

[11] F. J. Harris, C. Dick, and M. Rice. Digital receivers and transmitters using polyphase filter banks for wireless communications. *IEEE Transactions on Microwave Theory and Techniques*, April 2003.

[12] J. Kneip et al. Single chip programmable baseband assp for 5 ghz wireless lan applications. *IEICE TRANS. ELECTRON*, Feb 2002.

[13] C. Kozyrakis. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, University of California at Berkeley, 2002.

[14] C. Kozyrakis and D. Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *MICRO*, Nov. 2002.

[15] X. Liu and M. C. Papaefthymiou. Design of a high-throughput low-power is95 viterbi decoder. In *DAC*, Jun. 2002.

[16] E. P. B. R. A. Networks. *HIPERLAN Type 2; Physical layer*, 2001.

[17] J. Oliver et al. Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor. In *ISCA*, Jun. 2004.

[18] J. G. Proakis. *Digital Communications, 4th Ed.* McGraw-Hill, 2001.

[19] S. Rajagopal, S. Rixner, and J. R. Cavallaro. A programmable baseband processor design for software defined radios. In *Proceedings of the 45th Midwest Symposium on Circuits and Systems*, Aug. 2002.

[20] S. Ranpara and D. S. Ha. A low-power viterbi decoder design for wireless communications applications. In *Int. ASIC Conference*, Sept. 1999.

[21] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, London, UK, 2000.

[22] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.