The Extreme Benchmark Suite: Measuring High-Performance Embedded Systems

by

Steven Gerding

B.S. in Computer Science and Engineering, University of California at Los Angeles, 2003

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

Author .....
Department of Electrical Engineering and Computer Science
August 31, 2005

Certified by .....
Krste Asanović
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The Extreme Benchmark Suite: Measuring High-Performance Embedded Systems

by

Steven Gerding

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The Extreme Benchmark Suite (XBS) is designed to support performance measurement of highly parallel "extreme" processors, many of which are designed to replace custom hardware implementations. XBS is designed to avoid many of the problems that occur when using existing benchmark suites with nonstandard and experimental architectures. In particular, XBS is intended to provide a fair comparison of a wide range of architectures, from general-purpose processors to hard-wired ASIC implementations. XBS has a clean modular structure to reduce porting effort, and is designed to be usable with slow cycle-accurate simulators. This work presents the motivation for the creation of XBS and describes in detail the XBS framework. Several benchmarks implemented with this framework are discussed, and these benchmarks are used to compare a standard platform, an experimental architecture, and custom hardware.

Thesis Supervisor: Krste Asanović
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

I would first and foremost like to thank my advisor, Krste Asanović, without whom XBS and countless batches of homebrew would not have been possible. His genius and concern for his students are great assets to those who are fortunate enough to work with him.

I would also like to thank the SCALE group at MIT for their input and assistance with this work; especially Chris Batten and Ronny Krashinsky, whose knowledge of the vector-thread architecture is unparalleled (probably because they invented it). Elizabeth Basha and Rose Liu also deserve a great deal of credit for the Bluespec implementation of the 802.11a transmitter.

Finally, I must extend my deepest gratitude to Mom, Dad, Alison, and Melody, who have never wavered in their support of me in all of my ventures in life. I love you guys.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The vast majority of microprocessors manufactured today are slated for use in embedded computing environments [40]. An embedded processor is one that will not be used to run general purpose programs, but instead, used as a component of a larger system. Examples of such systems include digital cameras, cellular phones, wireless basestation routers, gaming machines, and digital video recorders. The applications that embedded processors must execute vary widely and include such tasks as image compression, digital signal processing, encryption, and 3D graphics processing. There are, however, certain characteristics that many embedded applications share such as real-time deadlines and abundant thread and data level parallelism. These traits are generally not exhibited by general purpose programs.

Designers of these systems must choose between myriad embedded processors, general purpose processors, and custom hardware implementations for inclusion in their products. These decisions are made based on price, performance, and power considerations. Unfortunately, obtaining a fair comparison of these diverse platforms is not possible with existing benchmark suites. This work discusses the problems involved in benchmarking these platforms and presents a new benchmark suite designed to overcome them. This benchmark suite is then used to compare a standard architecture, an experimental architecture, and custom hardware implementations.

1.1 Motivation

Much research is being conducted on embedded processors and new architectures are constantly being developed in industry and academia. Frequently, these platforms are in early stages of development and may not have the same associated infrastructure that they would at a more advanced stage. For example, there may not exist a compiler to take advantage of advanced features of the architecture. In addition, the architecture may not yet be implemented on real hardware, existing only as a cycle-accurate simulator or an FPGA prototype. Finally, the architecture may not have a full-featured operating system. These situations are even more likely with non-standard and experimental architectures, for which compiler and operating system technology is not as advanced.

While these architectures are arguably the most interesting, they are notoriously difficult to evaluate using existing benchmark suites for several reasons:

- Without a compiler that can utilize all of the features of an architecture, each benchmark must be hand-tuned to achieve full performance. This is usually not a problem for small benchmarks, but full application benchmarks can contain tens of thousands of lines of code. Usually, a few small kernels will account for a large portion of the run-time, so hand-tuning is still feasible; however, existing benchmark suites, assuming that the entire application will be compiled, rarely have a structure that makes these kernels apparent.

- If an architecture exists only as a slow cycle-accurate simulator, the time required to run a large benchmark can be prohibitively long. Some of this time is unavoidable: the benchmark code needs to be simulated in order to evaluate the architecture. However, a large portion of a benchmark's run time can be spent generating inputs and verifying outputs. These tasks are not required in real-world situations, and so are not interesting to simulate; unfortunately this is unavoidable with existing benchmark suites.

- Non-standard architectures running non-standard operating systems may not

support the full gamut of system calls, most notably those relating to I/O. If a benchmark depends on unsupported system calls, it will not be able to run on such an architecture without some modification. This modification could be time-consuming and could possibly compromise the integrity of the benchmark.

- For benchmark scores to be meaningful, it must somehow be determined that the device being tested is a valid implementation of the benchmark algorithm. While some existing benchmark suites provide facilities for verification of the benchmark's output, it is usually the case that a representative set of input used for performance testing will not adequately stress all features of an implementation. It is therefore important that a benchmark suite include separate verification and performance test cases.

To obtain a fair comparison of embedded processors, general purpose processors, custom hardware, and experimental non-standard architectures in the context of embedded applications, a new benchmark suite must be developed. Such a suite must address the problems described above and include kernels and applications that are representative of current embedded workloads.

1.2 Related Work

Numerous other benchmark suites are currently used to measure the performance of microprocessors. The most widely used of all benchmark suites is the Standard Performance Evaluation Corporation (SPEC) CPU benchmark suite [14]. The SPEC benchmark suite has gone through several revisions since its inception in the late 1980s; even in the latest version, however, very few of its benchmarks represent applications that would be run on an embedded processor. For this reason, the embedded processor community has turned to other benchmark suites that have been specifically targeted to the embedded application domain.

The EDN Embedded Microprocessor Benchmark Consortium (EEMBC) benchmark suite [5] has become an industry standard for evaluating embedded processors.

The EEMBC benchmarks include applications and kernels from the signal processing, networking, and image and video compression domains which are representative of the types of tasks performed by embedded systems. In addition, EEMBC realizes the importance of hand-optimizing benchmarks to achieve optimum performance. For this reason, they allow portions of the benchmark code to be hand-written in assembly and publish both the out-of-the-box (compiled) and optimized benchmark scores. Unfortunately, it is not made clear which parts of the benchmark it is beneficial, or even legal to optimize. This complicates hand-optimization of large applications, and can lead to illegitimate optimizations being unwittingly performed that compromise the integrity of the benchmark. EEMBC partially solves the legitimacy problem by requiring optimized benchmark implementations to be certified by the EEMBC Certification Laboratory before scores are published. This process, however, is expensive and could be avoided if performing the illicit optimizations was not possible in the first place.

In addition to EEMBC, other attempts have been made to develop a benchmark suite to evaluate embedded microprocessors, primarily by the academic sector. One such suite is the MiBench embedded benchmark suite [13]. MiBench is very similar to the EEMBC benchmark suite and contains many of the same applications and kernels. The MiBench benchmarks, however, are freely available to academic researchers whereas access to the EEMBC benchmarks requires membership in the consortium. MiBench, in contrast to EEMBC, does not make any allowances for hand-optimization of benchmarks and is intended only for platforms with full compiler support.

Another benchmark suite intended for the embedded domain is the MediaBench suite of benchmarks [31] developed at UCLA. MediaBench contains several applications from the signal processing and image and video processing domains which are generally representative of real-world embedded workloads. However, like MiBench, it is primarily intended to be compiled rather than hand-optimized.

One of the earliest attempts to create an embedded benchmark suite was DSPstone [43]. The purpose of the DSPstone project was to evaluate the performance of DSP compilers relative to the performance of hand-optimized assembly code. At the

time of its development, embedded systems were primarily used in signal processing applications, which is the application domain DSPstone focuses on. The designers of DSPstone make the distinction between kernel and application benchmarks in their literature, but the "application" benchmarks presented do not consist of multiple interacting kernels, as real-world embedded applications do.

Before benchmark suites tailored to the embedded application domain were available, the most commonly cited benchmark for embedded processor performance was Dhrystone [41]. Dhrystone is a single-program synthetic benchmark that was originally developed to mimic the proportions of high level language constructs appearing in programs. Its usefulness, however, has been widely discredited [42] and so the metric has been generally abandoned.

The problem of dramatically long benchmark runtimes when using simulators was addressed by the MinneSPEC [36] project. Minnespec sought to reduce the runtime of SPEC benchmarks by creating new, smaller input sets that remained representative of the full input sets traditionally used with SPEC. However, because MinneSPEC uses the SPEC benchmarks, it is generally not suitable for evaluating embedded systems.

ALPBench [32], a new multimedia benchmark suite, recognizes the importance of hand-optimization to achieve optimum performance. The ALPBench benchmarks are modified to exploit thread and data level parallelism using POSIX threads and Intel SSE2 instructions, respectively. While exposing the parallelism inherent in benchmarks is an important step towards reducing porting effort, the ALPBench framework is limited to Intel x86 architectures running UNIX compatible operating systems and does not make any provisions for non-standard and experimental architectures.

Other benchmark suites, however, have addressed the need to compare hardware implementations and experimental architectures against more standard platforms. The RAW benchmark suite [6] aims to compare reconfigurable hardware implementations to benchmarks running on conventional microprocessors. VersaBench [38] measures the "versatility" of an architecture, i.e., its ability to perform well on a wide range of different applications, and is intended to be run on experimental platforms. Both of these benchmark suites contain applications from the embedded domain, such

as 802.11a and FFT, as well as applications intended for more traditional platforms, including the SPEC benchmarks and the N-Queens problem.

1.3 Organization

The next chapter describes the Extreme Benchmark Suite, our solution to the problems with existing benchmark suites described above. Chapter 3 introduces the benchmarks that currently comprise the suite, and Chapter 4 describes their implementation on several different platforms. The experiments run using the Extreme Benchmark Suite are described in Chapter 5 and the results of those experiments are presented. Finally conclusions and future work are discussed in Chapter 6. For reference material relating to the Extreme Benchmark Suite, refer to the Extreme Benchmark Suite technical report [10].

Chapter 2

The Extreme Benchmark Suite

In the previous chapter, we discussed why existing benchmark suites are not suitable for non-standard and experimental architectures. To address this need, we present the Extreme Benchmark Suite (XBS), a new benchmark suite designed to measure the performance of highly parallel, experimental, and non-standard "extreme" architectures. XBS is also flexible enough to work with conventional processors and custom hardware implementations, and is intended to provide a fair comparison of a wide range of architectures.

XBS has a clean, modular structure to reduce porting effort. The input generation and output verification facilities are separate from the actual benchmark code, and the important kernels of application benchmarks are included independently as benchmarks themselves. These features greatly simplify porting the benchmark suite to new platforms, especially non-standard and experimental architectures. XBS also provides mechanisms for full correctness verification, debugging, and representative performance testing of all benchmarks.

This chapter first describes the organization of the benchmark set as a whole. The general structure of an XBS benchmark is then discussed. Finally, the abstraction used for input and output data streams is explained. Chapter 3 describes in detail the benchmarks currently implemented within the XBS framework. Chapter 4 goes on to chronicle the implementation of these benchmarks on several different architectures.

2.1 Benchmark Hierarchy

XBS Benchmarks are organized into three hierarchical levels: kernels, applications, and scenarios.

- **Kernels** represent small compute-intensive pieces of code that perform a single algorithm. Examples include FFT, matrix multiply, Quicksort, DCT, RSA encryption, and image convolution.

- **Applications** combine multiple kernels to perform an identifiable task such as perform JPEG compression of an image or decode an MP3 audio file.

- **Scenarios** combine multiple simultaneously running applications to measure the performance of processors on typical workloads. An example would be running a WCDMA link while decoding an MPEG video file.

This organizational structure vastly reduces the effort of porting application benchmarks to a new architecture by identifying the important kernels and providing an environment in which to develop and test them independently. As an example, consider the JPEG image compression application. This application contains almost 24,000 lines of code, but approximately 90% of the execution time is spent in just 4 kernels. If an architecture requires hand-optimization of benchmarks to reach its full performance potential, the ability to develop, debug, and test these 4 kernels independently will greatly simply the implementation of the JPEG compression application.

In addition to making porting applications easier, the three levels of the hierarchy provide different insights into the performance of an architecture. Kernels test the ability of an architecture to complete a single homogeneous task. Generally, the instruction cache will not be taxed because of the small code size of kernel benchmarks, but the size of the data cache could be a factor that impacts performance. The primary determinants of an architecture's performance on kernel benchmarks will be raw compute and memory bandwidth. Kernels also provide a simple performance filter for early designs: if an architecture cannot satisfy the timing requirements of

the kernel, it has no hope of delivering adequate performance on an application which includes that kernel.

Application benchmarks reveal the ability of an architecture to run multiple, interacting tasks. Because the code size of an application is much larger than that of a kernel, the instruction cache of an architecture will be stressed more heavily, and may need to be shared by multiple kernels. The application's data will usually be touched by more than one kernel, and may need to be passed from one kernel to another. If there is parallelism present between the kernels of an application, as is generally the case, the architecture may be able to run several of the kernels concurrently to increase performance.

Scenario benchmarks test an architecture's ability to run several distinct tasks at the same time. This makes it necessary to share the architecture's resources during the entire execution of the benchmark. These resources may include processor cycles, memory bandwidth, and space in the instruction and data caches. Unlike application benchmarks, which contain kernels specifically designed to operate with each other, scenarios are comprised of applications that were designed independently, making resource contention less predictable and possibly more severe.

2.2 Benchmark Structure

All XBS benchmarks have the structure shown in Figure 2-1. An Input Generator first generates an input file or a set of input files for the benchmark. These files are then read by the Test Harness which initiates benchmark execution on the Device Under Test (DUT) and supplies it with the input data. When the benchmark code has completed, the test harness reports timing information and writes the generated output data to a set of output files. These files, along with the input files, are then fed into the Output Checker which determines if the benchmark output is correct.

The XBS philosophy is to make the Test Harness as simple as possible. The only tasks for which the Test Harness is responsible are supplying the DUT with input, extracting the output from the DUT, and timing the execution of the benchmark

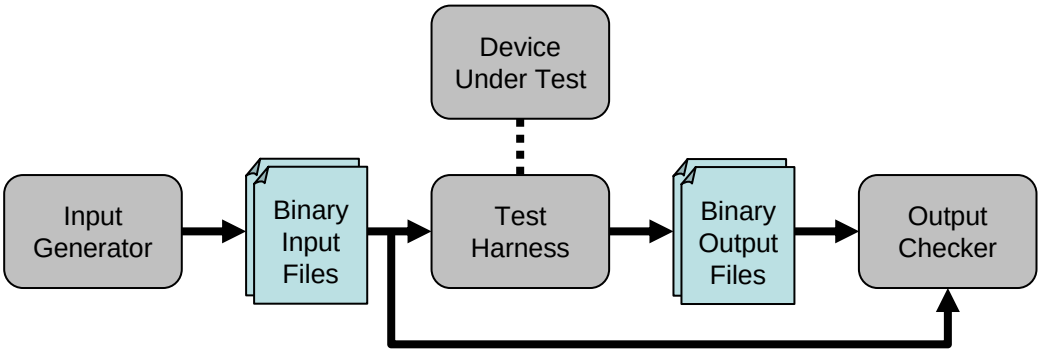

Figure 2-1: XBS IO Structure

code. All other complexity is relegated to the Input Generator and Output Checker.

The advantages to structuring the benchmarks in this manner are two-fold. First, the effort involved in porting XBS to new architectures is significantly reduced. Because the Input Generator and Output Checker only need to communicate with the Test Harness through binary files, they can be run on a different machine from the DUT. Only the Test Harness needs to be ported to the new architecture, and it is designed to be as minimal as possible.

The other advantage of allowing input generation and output verification to take place on a separate machine is that if the benchmark is to be run on a slow cycle-accurate simulator, these processes need not be run on the simulator itself. This is significant because the input generation and output verification can require much more processing than the benchmark code itself. If it were to be run on the simulator, the time required to benchmark the architecture would grow dramatically. Because the architecture would not be required to execute the input generation and output verification code in a real-world situation, we are not interested in its performance on this code and it is therefore wasteful to run it on the simulator.

The three components of an XBS benchmark, the Input Generator, Output Checker, and Test Harness, are described in detail in the following sections.

2.2.1 Input Generator

As previously mentioned, the Input Generator for each benchmark generates the set of input files on which the benchmark will operate. It is usually necessary for the Input Generator to create multiple input sets for verification, performance, and power testing. For verification purposes, the Input Generator should not be concerned with creating an input set that is representative of real-world data. Instead, the input set should be designed to stress every component of the benchmark code. This should include known corner cases of the algorithm, as well as random test cases. The random test cases, though randomly generated, should be repeatable so that debugging is possible if the random test reveals an error in the implementation.

In contrast to the verification set, the input set for performance testing should match real-world data as closely as possible so that any data-dependent control paths in the implementation are taken a representative number of times. There are a number of strategies that can be employed to create representative input sets, such as:

- Using actual input data files. For example, actual RGB photographs could be used as the input set for a JPEG encoder.

- Generating random data with a specific probability of emitting a meaningful value. For example, there is a data-dependent shortcut in the IDCT kernel (see Section 3.3.4) if all of the AC terms of a row are zero. The Input Generator should therefore make these terms zero with the same probability as in actual JPEG image data.

- Using a function that models real-world data to generate inputs. For example, the inputs to the ENCODE kernel (see Section 3.1.4) are quantized data blocks. The Input Generator for ENCODE can model this data using a function derived from the quantization coefficients.

- Using the inverse of the benchmark as an Input Generator. For example, the Input Generator for the decryption end of the Rijndael [4] AES standard could be the encryption end of the same standard.

It is possible that a benchmark is used in different real-world situations, in which case different input sets could be generated representing the different situations. A command line parameter is used to tell the Input Generator which input set to create (i.e. random or directed verification, or performance). In addition to this parameter, the Input Generator takes as input a seed for random number generation, and, if necessary, the size of the input set to generate (e.g., the width and height of an image).

2.2.2 Output Checker

The Output Checker for a benchmark reads in the benchmark's input and output files and determines whether the output is correct. The definition of correctness can vary depending on the benchmark. For some benchmarks, there is only one correct sequence of output bits for a given set of inputs. An example of such a benchmark would be decoding a losslessly compressed file. In such an application, if even one bit of the output file were incorrect, the implementation would be considered a failure.

Other benchmarks may not define correctness as rigidly. For example, an implementation of Fast Fourier Transform (FFT) could be considered correct if the output values lay anywhere within an acceptable range. Sometimes this range is defined by an accepted standard (such as the IEEE 1180 [15] standard for IDCT correctness), and sometimes it is obtained from real-world accuracy requirements. The check for correctness can be arbitrarily complex, so the Output Checker can require significantly more implementation and computational effort than the actual benchmark code.

In most cases, the first step in determining correctness is to obtain a reference version of the output data that is known to be correct. The Output Checker can either include an embedded reference version of the benchmark, or access pre-computed output files. In some cases, however, the benchmark's correctness may not be as closely tied to a reference version of the output. For example, a compression benchmark might only specify that the output be a valid compressed version of the input with a certain compression ratio rather than be concerned with the actual output bits

themselves. In such a situation, the Output Checker might operate by decompressing the output file and comparing it with the input file.

In addition to verifying an implementation of a benchmark, the Output Checker can also be used as an aid in debugging an implementation while it is still in development. The output checker is well suited to this task because it has access to both the implementation's output and the reference output; it can therefore disclose the values and positions of the output records that do not agree. Debugging information can be enabled in the Output Checker using a command line flag.

2.2.3 Test Harness

The Test Harness is responsible for interfacing with the actual benchmark code running on the DUT. It must supply input to and receive output from the DUT, as well as time the execution of the benchmark. Unlike the Input Generator and Output Checker, the Test Harness depends on the specifics of the DUT and must be re-written for every platform to which the benchmark is ported. For this reason, the Test Harness is designed to be as simple as possible.

The Test Harnesses for different platforms can vary significantly. For example, a Test Harness for a conventional architecture may load the entire input stream into a memory buffer and pass a pointer to that buffer to the code running on the DUT, while a Test Harness for a hardware implementation might supply small segments of the input data to the DUT when it requests them.

To reduce the effort in porting the Test Harness to new architectures, an XBS Test Harness Library is provided that implements common tasks that the Test Harness may need to perform, such as reading input files, writing output files, and starting and stopping timers. The functions in this library need to be re-written to support each architecture, but once the library exists for an architecture, the Test Harnesses for the individual benchmarks can be easily implemented by making use of the library functions. Refer to [10] for a complete description of the XBS Test Harness Library.

2.3 Input/Output Bit Streams

Each benchmark is structured as one or more compute tasks that perform all I/O through one or more bit streams. Each bit stream contains multiple "records" where a record is an atomic data unit that is consumed or produced by the benchmark. Examples of records include an audio sample, an image, or a network packet. For some benchmarks, the record format is fixed whereas for others the implementation can select any suitable encoding. Each record contains one or more primitive data objects, such as integers, floating-point numbers, or character strings. For some benchmarks, the endianess of multi-byte objects is fixed by the application (e.g., big-endian for network packet words), in other cases the benchmark may specify that the target native endianess can be used.

Bit streams are an abstraction of the various possible forms of I/O. For example, a bit stream might be implemented as an array in memory accessed directly by load and store instructions or DMA engines, as network messages sent over an inter-processor communications network, or as a high speed serial link connected directly to the hardware implementation. This allows XBS benchmarks to be run on non-standard architectures that may not have typical I/O facilities, and provides for a fair comparison of these platforms.

Although bit stream implementations can take many forms, benchmark run rules impose some restrictions. Each input and output bit stream has an associated buffer, and the cost of the buffer must be included in the cost of the implementation.

For an input bit stream, the implementation can randomly access data within its input buffer, but can only refill the buffer by reading bits sequentially from the input stream. The simplest valid input buffer is a single flip-flop. The workstation reference implementations have a buffer that holds the entire input stream in memory. Similarly, for an output bit stream, the simplest valid output buffer is a single flip-flop, but workstation reference implementations place the entire benchmark output into a single memory buffer. The implementation may overwrite data in the input buffer or read data from the output buffer.

The implementations are not charged for the cost of providing the physical port for bit stream input on an input stream, or for bit stream output on an output stream. The cost of this port is small because bit stream I/O is sequential. Small buffers, such as individual flip-flops, often already have a separate wire for input/output in the correct location. For larger RAM-based buffers, adding a wide input/output sequential access port that is time-multiplexed with the implementations accesses would add little overhead to the existing buffer. For the workstation reference implementations, it is difficult to account for the true cost of bringing the data into and out of main memory, but usually there is free main memory bandwidth to DMA data in and out of memory.

Implementations can use parallel links to increase the throughput of a single bit stream, but only by using a data-oblivious striping strategy. That is, each link i transfers the sequential next N_i bits of the bit stream regardless of the data contents. N_i can be different for different links but must be constant for each link across a benchmark run.

Chapter 3

Benchmarks

This chapter describes the benchmarks currently implemented within the XBS framework. There are two application benchmarks and 14 kernel benchmarks chosen from the image processing and communications/digital signal processing domains. The first application benchmark described is CJPEG, the compression end of the popular JPEG image compression standard. CJPEG consists of four communicating kernels: RGBYCC, FDCT, QUANTIZE, and ENCODE.

From the communications domain, XBS includes the 802.11A TRANSMITTER application benchmark, the transmitting end of the IEEE 802.11a wireless communication standard. 802.11A TRANSMITTER is comprised of six kernel benchmarks: SCRAMBLER, CONVOLUTIONAL ENCODER, INTERLEAVER, MAPPER, IFFT, and CYCLIC EXTEND.

In addition to these application benchmarks, XBS contains four kernels that are not associated with any application benchmark. They are VVADD, FIR, TRANSPOSE, and IDCT. At this time, no scenario benchmarks have been implemented, but future versions of XBS are slated to include them, as they are a good analogue of real-world situations.

3.1 CJPEG

CJPEG is the encoder end of the JPEG [37] image compression standard. Our imple-

mentation is based on the work done by the Independent JPEG Group [39].

JPEG is a lossy compression standard that utilizes psychophysical characteristics of the human visual perception system to discard information that will not significantly affect the quality of the image. Information is lost by reducing the resolution of the chrominance fields and by heavily quantizing the high frequency AC components of the data.

The top level schematic of CJPEG is shown in Figure 3-1. The RGB input image is first converted to the Y/Cb/Cr color space. The frequency domain representation of each color component is then calculated in 8x8 blocks using a Forward Discrete Cosine Transform. Next, the coefficients of the frequency domain representation are quantized to allow for compression. Finally, the quantized data is encoded using Differential Pulse Code Modulation, Run-Length Encoding, and Huffman Coding to create the JPEG image.

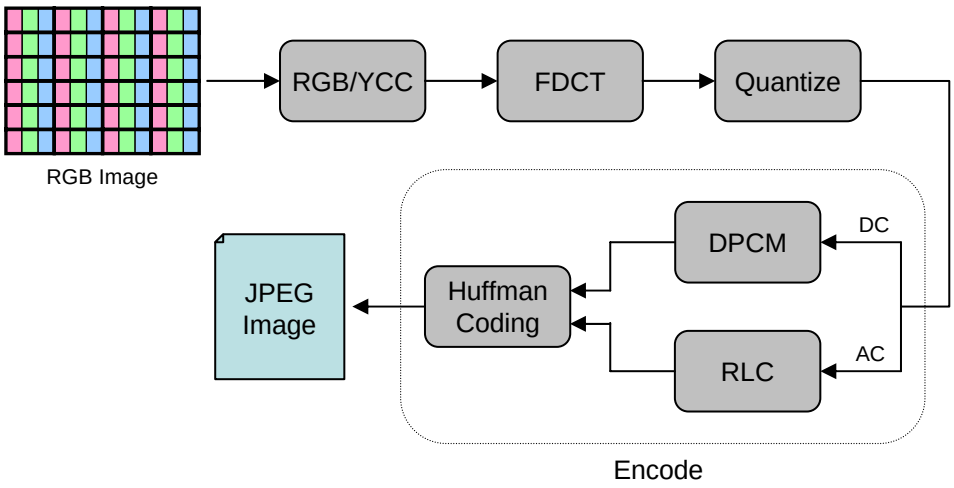

Figure 3-1: CJPEG Top Level Schematic

The following sections provide descriptions of the kernels of CJPEG:

3.1.1 RGBYCC

The first step in JPEG compression is to convert the pixels from the RGB color space to the Y/Cb/Cr color space. This step is performed because the human visual

system is more sensitive to luminance (the Y channel) than chrominance (the Cb and Cr channels), so separating the pixels into these components allows the chrominance channels to be heavily compressed without significantly impacting the image quality.

The conversion between RGB and Y/Cb/Cr is a simple linear transformation. The relationship between the two color spaces is given below:

$$\begin{aligned} Y &= 0.2989\ R + 0.5866\ G + 0.1145\ B \\ Cb &= -0.1687\ R - 0.3312\ G + 0.5000\ B \\ Cr &= 0.5000\ R - 0.4183\ G - 0.0816\ B \end{aligned}$$

3.1.2 FDCT

Once the pixels have been split into their Y, Cb and Cr components, they are transformed from their spatial domain representation to their frequency domain representation using the Discrete Cosine Transform, or DCT. The individual color components are split into 8x8 blocks which are fed into the DCT. The frequency domain representation of a block of pixels is expressed as an 8x8 block of coefficients representing the weighted sum of the DCT basis functions shown in Figure 3-2.

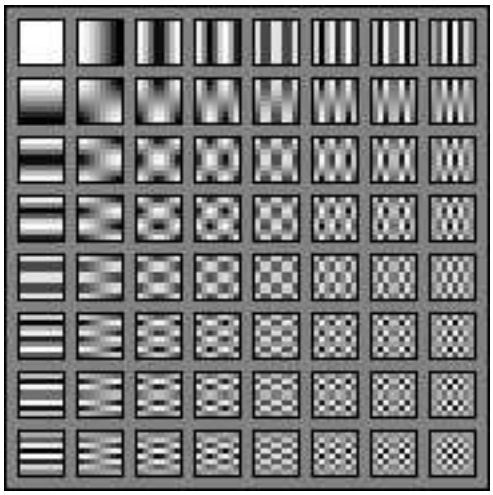

Figure 3-2: DCT Basis Functions

The two-dimensional DCT of an 8x8 block is defined by the following equation:

$$F(u,v) = \frac{\Lambda(u)\Lambda(v)}{4}\sum_{i=0}^{7}\sum_{j=0}^{7}\cos(\frac{(2i+1)\cdot u\pi}{16})\cdot\cos(\frac{(2j+1)\cdot v\pi}{16})\cdot f(i,j)$$

$$\Lambda(\xi) = \begin{cases} \frac{1}{\sqrt{2}} & for\ \xi = 0 \\ 1 & otherwise \end{cases}$$

The above equation can be factored into a series of one-dimensional DCTs as follows:

$$F(u,v) = \frac{1}{2}\sum_{i=0}^{7}\Lambda(u)\cdot\cos(\frac{(2i+1)\cdot u\pi}{16})\cdot f(i,j)$$

$$G(i,v) = \frac{1}{2}\sum_{j=0}^{7}\Lambda(v)\cdot\cos(\frac{(2j+1)\cdot v\pi}{16})\cdot f(i,j)$$

This factorization implies that the two-dimensional DCT can be accomplished by first performing the one-dimensional DCT of each row of the block, and then performing the one-dimensional DCT of each column. The Loeffler, Ligtenberg, and Moschytz [33] (or LLM) DCT algorithm exploits this property to calculate the DCT of an 8x8 block efficiently using fixed-point integer arithmetic. The LLM algorithm uses 12 multiplies and 32 adds and has the property that no data path contains more than one multiplication, meaning that the algorithm is very accurate even using fixed-point arithmetic. This is the XBS reference algorithm for FDCT.

3.1.3 QUANTIZE

After the blocks have been transformed to their frequency domain representations, they are quantized by dividing each coefficient in the block by a quantization factor. These quantization factors are chosen such that the higher frequency components of the block (see Figure 3-2) are quantized more heavily than the lower frequency components because the human visual system is more sensitive to errors in the lower

frequencies. The objective of quantization is to reduce a large number of the higher frequency coefficients to low values that can be represented by only a few bits, or, in the best case, reduce them to zero. The more heavily the coefficients are quantized, the better the compression ratio will be, but the more the image quality will be degraded.

3.1.4 ENCODE

Once the data blocks have been quantized and contain a large number of zero and low valued coefficients, they are encoded to achieve the actual compression in the JPEG standard. There are three different types of encoding that are used to compress the data: Differential Pulse Code Modulation, Run-Length Encoding, and Huffman Coding.

3.1.4.1 Differential Pulse Code Modulation

Differential Pulse Code Modulation (or DPCM), is used to encode the DC coefficients of the blocks (the coefficient in the upper left corner of Figure 3-2). The DC values are not heavily quantized, meaning they have relatively high values compared to the other coefficients. This means that using Huffman Coding on these values directly would not be beneficial. However, it is observed that in many images, the DC values of successive blocks, though high, are close to one another. It is therefore advantageous to encode the DC value of a block as its difference from the DC value of the previous block, and Huffman Code that difference.

3.1.4.2 Run-Length Encoding

As stated in section 3.1.3, quantization leads to many of the AC values being reduced to zero. If there are long runs of consecutive zeros, they can be heavily compressed by skipping over the zero valued coefficients and outputting the number of zeros skipped when a non-zero coefficient is encountered; this is Run-Length Encoding (RLE).

The higher the frequency of the coefficient (i.e. the closer it is to the bottom right

of Figure 3-2), the more likely it is to be zero. Because we want to maximize the number of consecutive zeros, it is beneficial to reorder the coefficients to group the high frequency coefficients together. The reordering chosen is a zig-zag back and forth across the block from the DC component to the highest frequency AC component. This is shown in Figure 3-3.

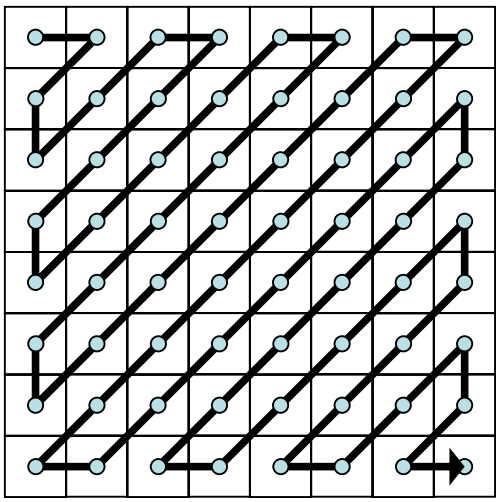

Figure 3-3: Zig-Zag Reordering

3.1.4.3 Huffman Coding

Huffman Coding replaces each value with a variable length code that is chosen based on the frequency with which the value appears. However, instead of the encoding the data directly, the length in bits of each value is Huffman encoded and followed in the output stream by the significant bits of the value.

After DPCM and Run-Length Encoding have been performed on the DC and AC coefficients respectively, the data consists of mostly low values (either the small difference between successive DC components, the number of consecutive zeros seen, or AC values that have been heavily quantized). Because the coefficients have primarily low values, many of them will be of the same (small) lengths in bits, making these lengths a good candidate for Huffman Coding.

3.2 802.11A TRANSMITTER

IEEE 802.11a [16] is a popular wireless communication standard that operates in the 5 GHz band using Orthogonal Frequency Division Multiplexing (OFDM). OFDM is an efficient multi-carrier modulation technique in which the base band signal is a composite of multiple data-encoded sub-carriers. The basic unit of transmission in 802.11a is the OFDM symbol, a collection of 53 sub-carriers (48 of which carry encoded data). The 802.11A TRANSMITTER application benchmark is responsible for transforming binary input data (that would be received from the MAC layer) into OFDM symbols suitable for transmission.

The 802.11a standard defines many different supported data rates, however, to avoid making the benchmark unnecessarily complicated, the XBS 802.11A TRANSMITTER benchmark and its kernels support only the 24 Mbps data rate. Because of this choice of data rate, it is defined that the system shall use 16-QAM modulation, a convolutional coding rate of 1/2, and shall assign 192 coded bits to each OFDM symbol (4 per data-carrying sub-carrier).

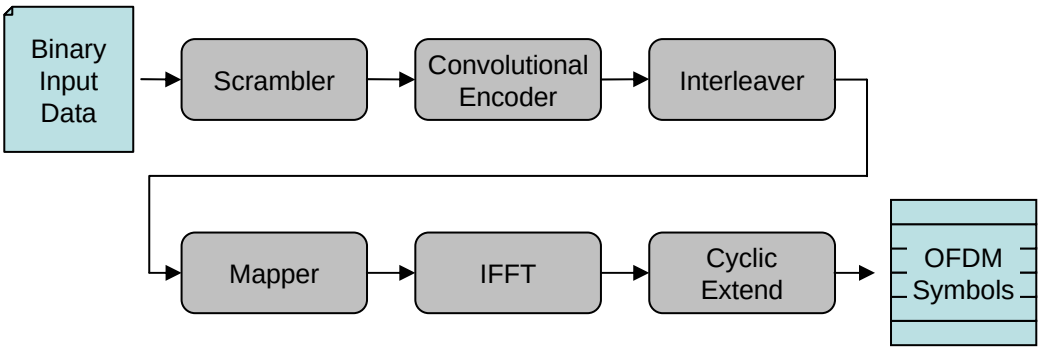

Figure 3-4: 802.11A TRANSMITTER Top Level Schematic

The top-level schematic of the 802.11A TRANSMITTER benchmark is shown in Figure 3-4. The binary input data first passes through the SCRAMBLER which XORs it with a scrambling sequence. The CONVOLUTIONAL ENCODER then performs convolutional coding on the data with a rate of 1/2. Next, the INTERLEAVER reorders the coded data and the MAPPER maps it to frequency domain OFDM symbols. These

symbols are then transformed to their time domain representation by the IFFT. Finally a guard interval is added in the CYCLIC EXTEND stage.

The following sections provide descriptions of the kernels of 802.11A TRANSMITTER:

3.2.1 SCRAMBLER

The SCRAMBLER kernel scrambles the input data by XORing it with a scramble sequence. This sequence is generated from a 7-bit seed value and repeats itself every 127 bits. Each bit of the sequence is generated according to the following formula:

$$scramble\_sequence_i = scramble\_sequence_{i-7} \oplus scramble\_sequence_{i-4}$$

The scramble sequence for times -7 through -1 is passed to the SCRAMBLER as an initialization seed. The SCRAMBLER algorithm is shown pictorially in Figure 3-5.

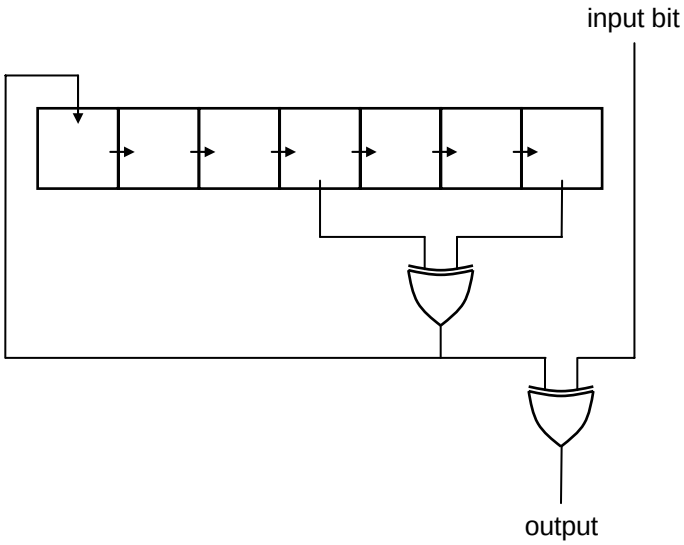

Figure 3-5: SCRAMBLER Algorithm

3.2.2 CONVOLUTIONAL ENCODER

The CONVOLUTIONAL ENCODER kernel adds redundant information to the data stream in such a way that forward error correction can be performed by the receiver if any bits are lost in the transmission. For every input bit, it generates two output bits, the even bit and the odd bit, that are functions of the current input bit and the previous 6 inputs. These output bits are defined as follows:

$$\begin{aligned} even\_output_i &= input_i \oplus input_{i-2} \oplus input_{i-3} \oplus input_{i-5} \oplus input_{i-6} \\ odd\_output_i &= input_i \oplus input_{i-1} \oplus input_{i-2} \oplus input_{i-3} \oplus input_{i-6} \end{aligned}$$

This algorithm is shown pictorially in Figure 3-6. The input data stream is defined to be zero for all times less than $i = 0$ for the purpose of processing of inputs 0 through 5.

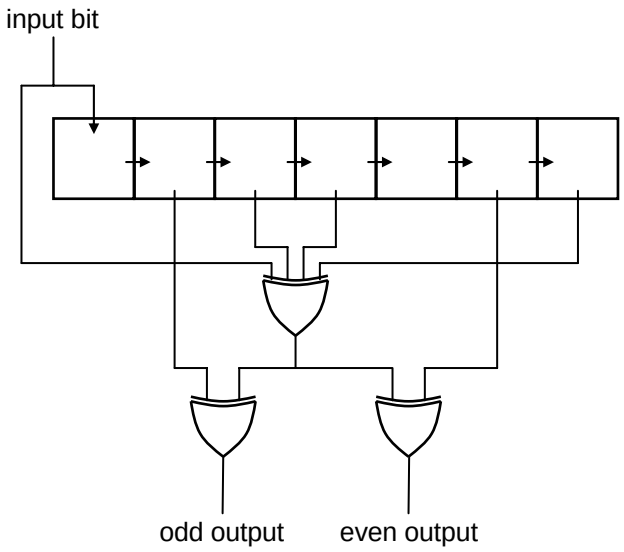

Figure 3-6: CONVOLUTIONAL ENCODER Algorithm

3.2.3 INTERLEAVER

The INTERLEAVER kernel reorders the data bits that will later be mapped to an OFDM symbol. This reordering serves two purposes: making sure adjacent coded bits are mapped to non-adjacent sub-carriers, and ensuring that adjacent bits are mapped to alternating more and less significant bits of the 16-QAM constellation. This is necessary because if several adjacent coded bits are lost, the receiver will not be able to perform error correction on the received data. The interleaving of the bits within an OFDM symbol is defined according to the following rule:

$$
\begin{aligned}
X &= 12 * (input\_index \mod 16) + (input\_index/16) \\
output\_index &= 2 * (X/2) + ((X + 192 - (X/12)) \mod 2) \\
& \quad symbol[output\_index] \leftarrow symbol[input\_index]
\end{aligned}
$$

3.2.4 MAPPER

The MAPPER kernel is responsible for mapping the binary data generated up to this point in the transmission process to sub-carriers of the OFDM symbol. It does this by dividing the data into groups of 4 bits and converting these groups to complex numbers representing points in the 16-QAM modulation constellation, and inserting these complex numbers into the appropriate indices of the OFDM symbol. In addition, the mapper inserts special pilot sub-carriers to enable synchronization at the receiver.

The sub-carriers of the OFDM symbol and the values mapped to them are shown in Figure 3-9. The data bits are converted to complex numbers according to the mappings shown in Figure 3-7. The values of the four pilot sub-carriers (11, 25, 39, and 53) are taken from the pilot vector shown in Figure 3-8. The pilot value used for a symbol is the pilot vector entry indexed by the symbol number, modulo 127. All of the pilot carriers in a symbol use the same value except for sub-carrier 53, which reverses the polarity of the value used by the other pilots.

Input Bits ($b_0b_1b_2b_3$)	Real	Imaginary
0000	$\frac{-3}{\sqrt{10}}$	$\frac{-3}{\sqrt{10}}$
0001	$\frac{-1}{\sqrt{10}}$	$\frac{-3}{\sqrt{10}}$
0010	$\frac{1}{\sqrt{10}}$	$\frac{-3}{\sqrt{10}}$
0011	$\frac{3}{\sqrt{10}}$	$\frac{-3}{\sqrt{10}}$
0100	$\frac{-3}{\sqrt{10}}$	$\frac{-1}{\sqrt{10}}$
0101	$\frac{-1}{\sqrt{10}}$	$\frac{-1}{\sqrt{10}}$
0110	$\frac{1}{\sqrt{10}}$	$\frac{-1}{\sqrt{10}}$
0111	$\frac{3}{\sqrt{10}}$	$\frac{-1}{\sqrt{10}}$

Input Bits ($b_0b_1b_2b_3$)	Real	Imaginary
1000	$\frac{-3}{\sqrt{10}}$	$\frac{1}{\sqrt{10}}$
1001	$\frac{-1}{\sqrt{10}}$	$\frac{1}{\sqrt{10}}$
1010	$\frac{1}{\sqrt{10}}$	$\frac{1}{\sqrt{10}}$
1011	$\frac{3}{\sqrt{10}}$	$\frac{1}{\sqrt{10}}$
1100	$\frac{-3}{\sqrt{10}}$	$\frac{3}{\sqrt{10}}$
1101	$\frac{-1}{\sqrt{10}}$	$\frac{3}{\sqrt{10}}$
1110	$\frac{1}{\sqrt{10}}$	$\frac{3}{\sqrt{10}}$
1111	$\frac{3}{\sqrt{10}}$	$\frac{3}{\sqrt{10}}$

Figure 3-7: 16-QAM Mapping Table (Normalized by $\frac{1}{\sqrt{10}}$)

```
pilot_vector[] = { 1, 1, 1, 1,  -1,-1,-1, 1,  -1,-1,-1,-1,   1, 1,-1, 1,
                  -1,-1, 1, 1,  -1, 1, 1,-1,   1, 1, 1, 1,   1, 1,-1, 1,
                   1, 1,-1, 1,   1,-1,-1, 1,   1, 1,-1, 1,  -1,-1,-1, 1,
                  -1, 1,-1,-1,   1,-1,-1, 1,   1, 1, 1, 1,  -1,-1, 1, 1,

                  -1,-1, 1,-1,   1,-1, 1, 1,  -1,-1,-1, 1,   1,-1,-1,-1,
                  -1, 1,-1,-1,   1,-1, 1, 1,   1, 1,-1, 1,  -1, 1,-1, 1,
                  -1,-1,-1,-1,  -1, 1,-1, 1,   1,-1, 1,-1,   1, 1, 1,-1,
                  -1, 1,-1,-1,  -1, 1, 1, 1,  -1,-1,-1,-1,  -1,-1,-1    }
```

Figure 3-8: Pilot Vector

3.2.5 IFFT

Once the frequency domain representation of the OFDM symbol has been created, it must be transformed to its time domain representation so that it may be transmitted. To accomplish this, a 64-point Inverse Fast Fourier Transform (IFFT) is used. The XBS reference version of the IFFT kernel uses a 3 stage radix-4 algorithm. In each stage, 16 radix-4 calculations are performed, each of which takes 4 complex inputs and produces 4 complex outputs.

The radix-4 calculation itself uses 8 multiplies, 16 adds, and 8 subtracts, however, in a fixed point implementation, extra shifts and adds are required for rounding and scaling. It takes 4 complex inputs: A, B, C and D, of which the real and imaginary components are 16-bits each. The data flow diagram of the radix-4 calculation, not including the operations necessary to support fixed point, is shown in Figure 3-10.

OFDM Sub-carrier	Complex Value	OFDM Sub-carrier	Complex Value
0	(0, 0)	32	(0, 0)
1	(0, 0)	33	map(in[96:99])
2	(0, 0)	34	map(in[100:103])
3	(0, 0)	35	map(in[104:107])
4	(0, 0)	36	map(in[108:111])
5	(0, 0)	37	map(in[112:115])
6	map(in[0:3])	38	map(in[116:119])
7	map(in[4:7])	39	(pilot[symbol], 0)
8	map(in[8:11])	40	map(in[120:123])
9	map(in[12:15])	41	map(in[124:127])
10	map(in[16:19])	42	map(in[128:131])
11	(pilot[symbol], 0)	43	map(in[132:135])
12	map(in[20:23])	44	map(in[136:139])
13	map(in[24:27])	45	map(in[140:143])
14	map(in[28:31])	46	map(in[144:147])
15	map(in[32:35])	47	map(in[148:151])
16	map(in[36:39])	48	map(in[152:155])
17	map(in[40:43])	49	map(in[156:159])
18	map(in[44:47])	50	map(in[160:163])
19	map(in[48:51])	51	map(in[164:167])
20	map(in[52:55])	52	map(in[168:171])
21	map(in[56:59])	53	(-pilot[symbol], 0)
22	map(in[60:63])	54	map(in[172:175])
23	map(in[64:67])	55	map(in[176:179])
24	map(in[68:71])	56	map(in[180:183])
25	(pilot[symbol], 0)	57	map(in[184:187])
26	map(in[72:75])	58	map(in[188:191])
27	map(in[76:79])	59	(0, 0)
28	map(in[80:83])	60	(0, 0)
29	map(in[84:87])	61	(0, 0)
30	map(in[88:91])	62	(0, 0)
31	map(in[92:95])	63	(0, 0)

Figure 3-9: Sub-carrier Mapping Table

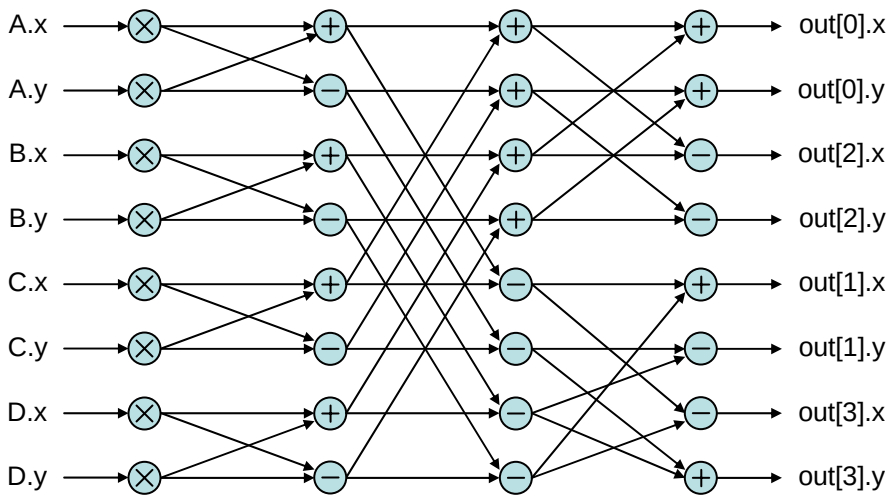

Figure 3-10: IFFT radix-4 Calculation Data Flow Diagram

At the end of the final stage of the IFFT, the data must be reordered. The reordering is accomplished by swapping OFDM symbol element i with element $64 - i$ for elements 1-63. Element zero is unchanged by the reordering.

3.2.6 CYCLIC EXTEND

After the OFDM symbol has been converted to its time domain representation, it is cyclically extended to form a guard interval. This is accomplished simply by prepending a copy the last 16 complex pairs of the symbol to the front of the symbol and appending a copy of the first complex pair to the end, thus lengthening the symbol from 64 complex pairs to 81. This process is shown pictorially in Figure 3-11.

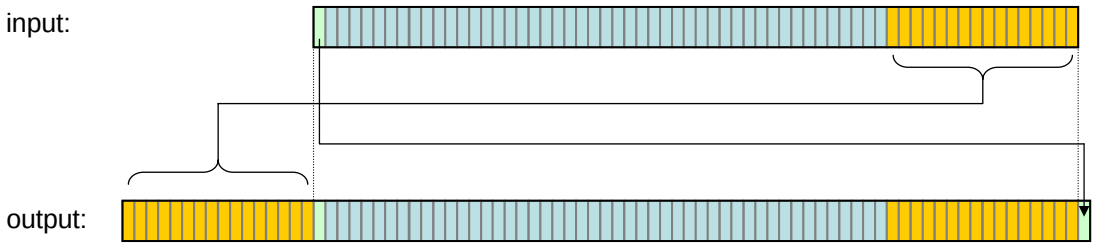

Figure 3-11: Cyclic Extension

The purpose of the guard interval is to combat inter-symbol interference if the

channel creates echoes of the signal [18]. The guard interval is designed to be longer than the impulse response of the transmission channel to protect against echo delays. Addition of the guard interval reduces the data capacity of the channel by approximately 21%, but is important for robustness.

3.3 Other Kernels

In addition to the two application benchmarks described above, XBS includes several kernels that are not currently a part of an XBS application benchmark. These kernels were chosen either because they provide a unique insight into the performance of an architectural feature, or because they are part of an XBS application benchmark to be implemented in the future. The four kernels are VVADD, FIR, TRANSPOSE, and IDCT.

3.3.1 VVADD

The VVADD kernel performs the addition of two vectors. It takes as input two vectors of 32-bit integers and outputs a vector representing the sum of its inputs. The operation can be expressed as:

$$output_i = input1_i + input2_i$$

The VVADD kernel is very simple, but it is an important component of several applications. It also useful to test an architecture's performance on a purely data parallel operation with a small ratio of computation to memory accesses.

3.3.2 FIR

The Finite Impulse Response (or FIR) filter is a digital filter that is frequently used in signal processing applications such as digital data modems. A FIR filter consists of a delay line of a fixed length, and a corresponding set of coefficients, or taps. At the

beginning of the computation, all of the values in the delay line are defined to be zero. For each input sample, the values in the delay line are shifted down one and the input is placed in the newly vacated delay line slot. The output for the sample is calculated by taking the dot product of the values in the delay line and their corresponding taps. The outputs can therefore be expressed in terms of the inputs and taps as:

$$output_i = \sum_{j=0}^{num\_taps-1} input_{i-j} \cdot tap_j$$

In addition to being a useful component of many DSP applications, the FIR kernel tests an architecture's ability to perform multiply-accumulate operations.

3.3.3 TRANSPOSE

The TRANSPOSE kernel computes the out-of-place transposition of a square matrix of 32-bit values. It takes as input the number of elements on a side of the matrix and the matrix itself and outputs the transpose of the matrix. The operation can be expressed as:

$$output_{i,j} = input_{j,i}$$

Matrix transposition is a useful operation in many applications including 3D graphics pipelines. In addition, if an architecture makes use of structured memory accesses to increase performance, the TRANSPOSE kernel forces the architecture to use multiple memory access patterns, thus stressing the flexibility of its memory system.

3.3.4 IDCT

The IDCT kernel performs an Inverse Discrete Cosine Transform on an 8x8 block of data. Because it performs the inverse operation of the FDCT kernel, it transforms the

frequency domain representation of a block of pixels to its spatial domain representation. As is the case for the FDCT kernel, the XBS reference version of IDCT uses the LLM fixed point algorithm. IDCT exhibits computation and memory access patterns similar to FDCT, except that it has a data-dependent shortcut when processing the rows of the input block. If all of the AC coefficients in a row are zero, the IDCT of that row can be trivially calculated. The IDCT kernel is used in the JPEG decoding application.

Chapter 4

Implementation

In order to evaluate the Extreme Benchmark Suite, the benchmarks described in Chapter 3 were implemented on three different platforms: an Intel XEON processor running Linux, the SCALE vector-thread architecture, and custom ASIC hardware. These three platforms were chosen because they represent a standard architecture, a non-standard experimental architecture, and a custom hardware implementation, respectively. This demonstrates the ability of XBS to fairly compare a wide variety of platforms.

This chapter briefly describes the three platforms and discusses the general optimization techniques used for each. The specific method by which each XBS benchmark was optimized on each platform is also discussed. Chapter 5 goes on to describe the experiments carried out using these implementations.

4.1 SCALE

The SCALE vector-thread architecture [30] unifies the vector and multi-threaded execution models to exploit the different types of structured parallelism present in embedded applications. SCALE is especially well suited for executing loops, and, in contrast to conventional vector machines, has the ability to handle loops with arbitrary intra-iteration control flow and cross-iteration dependencies. It was chosen as a platform for evaluating XBS because it is an experimental architecture targeted

for the embedded application domain. In addition, the SCALE platform is in the early stages of development; there is no compiler that can target its advanced architectural features and it exists only as a cycle-accurate simulator. These aspects, as explained in Chapter 2, make XBS an ideal benchmark suite for evaluating this architecture.

The SCALE programming model consists of two interacting units: a scalar control processor (CP) and a virtual processor vector (VPV). The CP executes standard scalar instructions and issues commands to the VPV, directing it to perform computation and memory accesses. The VPV is comprised of an array of virtual processors (VPs), which have a set of registers and are capable of executing atomic instruction blocks (AIBs) of RISC-like instructions. These instructions may target any one of 4 execution clusters within the VP. To initiate execution of an AIB, the CP issues a **vector fetch** command to the VPV, instructing each VP to execute the block. VPs have no automatic program counter, so each AIB must be executed to completion and any further AIBs must be explicitly fetched, either by the CP of the VPs themselves. The latter is referred to as a **thread fetch**.

In addition to vector fetch directives, the CP can also instruct the VPV to execute vector memory operations. The simplest vector memory operation is the **unit stride vector load** which loads contiguous elements of an array in memory into successive VPs in the VPV. The **strided vector load** loads non-contiguous elements of an array that are separated by a constant stride into the VPV. The **vector segment load** loads contiguous blocks of elements into the VPV, with successive registers in each VP receiving successive elements of the block. Finally, the **vector segment strided load** loads non-contiguous blocks of elements into the VPV. For each of the vector load instructions described, there exists an analogous vector store instruction. The VPs may also issue load and store commands which will affect only themselves, not the entire VPV.

The registers in a VP are divided into two categories: private and shared. The state in shared registers is not preserved across AIBs; they are used for temporary intra-AIB state and by the CP to pass data to the VPs. Private registers do preserve their state between successive AIBs, and can therefore be used for long-lived state

within a VP. To enable loops with cross-iteration dependencies to be vectorized, the VPs are able to communicate directly with each other in a limited capacity. Each VP may send data to the following VP and receive data from the previous VP; this path is referred to as the **cross-VP network**.

The following sections describe how the XBS benchmarks were implemented and optimized on the SCALE vector-thread architecture:

4.1.1 CJPEG

The primary kernels of CJPEG all possess significant parallelism, however, while some exhibit structured vector parallelism, others have cross-iteration dependencies that would prohibit vectorization on traditional vector machines. The SCALE vector-thread architecture, on the other hand, has facilities for dealing with these hazards while still allowing the parallelism inherent in the algorithm to be extracted.

Following are descriptions of how each of the kernels of the CJPEG benchmark were implemented on SCALE:

4.1.1.1 RGBYCC

The description of the RGBYCC kernel in Section 3.1.1 makes it clear that the algorithm is trivially vectorizable. As such, the implementation strategy for this kernel on SCALE was to vectorize over the pixels. Each VP was made responsible for one pixel, consuming 3 input bytes and producing 3 output bytes.

The first step in the color conversion is to load the R, G, and B pixel values into the VPs. Because these components are stored in interleaved order, this load can be accomplished using an 8-bit *vector segment load* with a segment size of 3. Next, an AIB is fetched that computes the Y output value and begins calculating the Cb and Cr values. This is followed by an 8-bit *unit-stride vector store* that writes the computed Y values out to memory. Two other AIBs and vector stores are then executed, one for the Cb values and one for the Cr values. The calculation is divided amongst three AIBs to allow useful computation to overlap with the vector store

memory commands.

4.1.1.2 FDCT

As described in Section 3.1.2, the FDCT kernel performs a 2-D DCT on its input block by first doing a series of 1-D DCTs on the rows of the block, and then a series of 1-D DCTs on the columns of the block. These 1-D DCTs are independent of one another and can be processed concurrently.

The SCALE version of FDCT first loads each row of the input block into a VP using a 16-bit *vector segment* load with a segment size of 8. Two AIBs are then fetched to compute the 1-D DCTs of the rows, and the data is written back to memory using a 16-bit *vector segment store*.

To load the columns of the input block into the VPs, it is not possible to use a *vector segment load* because the block is stored in row major order. Instead, 8 separate *unit stride vector loads* are issued to load the data. Next, an AIB is fetched to calculate the even numbered output values for each column, and 4 *unit stride vector stores* are executed to write this data to memory. Finally, an AIB is fetched to calculate the odd numbered outputs, and these values are stored using 4 more *unit stride vector stores*. The computation can be split between two AIBs in such a way because the calculations of the even and odd numbered values are largely independent (except for the fact that they take the same inputs). Splitting the calculation allows processing of the odd numbered outputs while the even numbered outputs are being stored.

4.1.1.3 QUANTIZE

The primary reason that the QUANTIZE kernel is a major component of the runtime of the CJPEG application is the large number of integer divides that it must perform. On SCALE, divide is a very expensive operation that is unpipelined, performed iteratively, and requires tens of cycles to complete. For this reason, any algorithmic change that would reduce the number of divides would result in a significant performance increase.

A key observation towards developing such an algorithm is that the quantization factors change infrequently with respect to the number of times they are used in divide operations. We can take advantage of this fact by calculating the reciprocal of the quantization factors when they are changed, and multiplying by these reciprocals instead of dividing by the quantization factors themselves. This allows all of the divides in the quantization process to be replaced with multiplies (a low latency, pipelined operation on SCALE) with the small overhead of calculating the reciprocals when the quantization factors are changed.

When the quantization table is changed, the following two parameters are calculated for every quantization factor q in the table:

$$\begin{aligned} multiplier_q &= \left(\left(\frac{2^{31}}{q} \ll (1 + \log_2 q)\right) + 2^{17}\right) \gg 16 \\ shift\_amount_q &= \log_2 q + 16 \end{aligned}$$

These parameters are defined such that multiplying an input coefficient by $multiplier_q$ and right-shifting the result by $shift\_amount_q$ has the same effect as dividing the coefficient by q. The multiply and shift, however, take an order of magnitude fewer cycles to complete than the divide.

Once the divides have been eliminated, the quantization process is trivially and efficiently vectorizable. *Unit-stride vector loads* can be used to load the q, $multiplier_q$, $shift\_amount_q$, and input values. Once the inputs have been loaded, an AIB is fetched to carry out the quantization process and the results are written to memory using a *unit-stride vector store*.

4.1.1.4 ENCODE

As stated in Section 3.1.4, the ENCODE kernel of the CJPEG application benchmark uses DPCM and Huffman Coding to encode the DC coefficients and Run-Length Encoding and Huffman Coding to encode the AC coefficients. Because there are 63 AC coefficients for every DC coefficient, the effort employed in optimizing ENCODE

for SCALE was spent on the RLE and Huffman Coding of the AC coefficients.

The difficulty in vectorizing the encoding of the AC coefficients, as with many compression and encryption algorithms, lies in the data dependencies between the coefficients. There are four different data values that cause these dependencies:

1. **The run-length of zeros.** The symbols that the RLE scheme outputs for each non-zero coefficient are a function of the number of zero-valued coefficients that appear before it.

2. **The pointer to the next output location.** Because each coefficient may cause a variable number of bytes to be outputted, the output pointer is not known until the previous coefficient has been processed.

3. **The number of bits generated but not yet output.** The values outputted by the Huffman Coding scheme are a variable number of bits in length. If the number of bits generated is not a multiple of 8 after any coefficient is done being processed, the next coefficient will need to know how many extra bits have been generated so it can align its output bits appropriately.

4. **The actual bits generated but not yet output.** If the Huffman Coding of a coefficient creates bits that cannot be written to memory because a full byte is not ready for output, the processing of the next coefficient will need to output these bits in addition to its own.

Despite the fact that these values need to be computed serially, there is still a significant amount of parallelism available. Unfortunately, the cross-iteration dependencies would make vectorization on a traditional vector machine impossible. However, SCALE provides facilities for VPs handling different iterations to synchronize and communicate, thus allowing the parallelism that does exist to be extracted while still processing the serial portions of the algorithm correctly.

The strategy for implementing the ENCODE kernel on SCALE was to assign each VP one AC coefficient in a stripmined loop and use the *cross-VP network* to communicate the inter-coefficient dependencies. The first step is to initialize the cross-VP

queue with zero for the starting zero run-length, the initial output pointer which is an input to the function, and the initial output buffer state (i.e. the number of bits generated but not output by previous invocations of the algorithm, and these bits themselves). An 8-bit *unit-stride vector load* is then issued to give each VP the zig-zag index at which it will load its coefficient from the input block. Next, an AIB is fetched that initiates a thread of computation on each VP which will process the coefficient.

The first thing the thread does is load the coefficient from the input block at its zig-zag index and compare it with zero. If the coefficient is zero, the VP will receive and increment the zero run-length, pass along the other cross-VP values unchanged, and terminate the thread. If the coefficient is not zero, the VP will immediately send zero as the zero run-length and proceed to compute the absolute value, ones compliment, and bit length of the input (all of which can be done without receiving information from the previous VP). The VP will then receive the zero run-length from the previous VP, form the output symbol, and perform Huffman encoding on it using a table lookup. At that point, no further processing can occur without completely serializing with the previous VP, so the VP will receive the output pointer and the output buffer state. Finally, the VP will create its output, update the cross-VP values, and send them to the next VP.

4.1.2 802.11a transmitter

To port the 802.11a transmitter application benchmark to the SCALE vector-thread architecture, the scrambler, convolutional encoder, interleaver, mapper, and ifft kernels were hand-optimized using assembly language. Because the cyclic extend kernel creates the guard interval using `memcpy`, a library function already optimized for SCALE, no further optimization was required.

Instead of running each module on the entire input stream before proceeding to the next module, the SCALE implementation processes only as much input as it can hold in its cache at a time. Once this data has been processed by all of the modules, the implementation moves on to the next input chunk. This method maximizes

performance by reducing the amount of data that bounces in and out of the cache. While all of the modules are trivially vectorizable at the OFDM symbol level (because successive OFDM symbols are independent of one another), some possess parallelism at a finer granularity. For these modules, we vectorize at that finer granularity so as to maximize the vector length for a given amount of data held in the cache.

Following are descriptions of how each of the kernels of the 802.11A TRANSMITTER were implemented on the SCALE vector-thread architecture:

4.1.2.1 SCRAMBLER

As described in Section 3.2.1, the canonical definition of the SCRAMBLER kernel calculates the output one bit at a time from the input and current scramble sequence. At each step it also generates the next bit of the sequence. It is inefficient to process the input bit-by-bit, so a method must be devised to process the input 8, 16, or 32 bits at a time. The first step towards such a method is to observe that the sequence generator described will repeat itself every 127 bits. This means that the entire sequence can be pre-computed for each 7 bit seed and stored in 128 bits of memory (the 127 bit sequence with the first bit of the sequence concatenated on the end). Because there are 128 possible 7 bit seeds, the entire table of pre-computed sequences occupies 2048 bytes of memory. While this may seem to be a large increase in code size, it does not affect performance because each message that passes through the transmitter uses only one initialization seed; the entire table therefore need not reside in the cache.

Given the 128 bit pre-computed sequence, the output of the SCRAMBLER can be easily computed from the input 16 bytes at a time by XORing the input with the sequence, shifting the 128 bit sequence left 1 bit, and appending bit 1 (the second bit) to the right end of the sequence.

To implement this algorithm on SCALE, we first load the pre-computed sequence for the initialization seed from the table and write it to a set of 4 shared registers. An AIB is then fetched in which each VP shifts the shared sequence registers into its own private registers based on its VP number (VPN) so that each VP has its own copy of

the sequence registers aligned to the correct starting point. For example, the fifth VP (VP 4) should have its sequence registers in a state as if 4 iterations of the algorithm described above had already taken place. Each VP's private sequence registers are created as follows (all right shifts are logical):

$$
\begin{aligned}
private\_seq\_reg0 &= (shared\_seq\_reg0 \ll VPN) | (shared\_seq\_reg1 \gg (31 - VPN)) \\
private\_seq\_reg1 &= (shared\_seq\_reg1 \ll VPN) | (shared\_seq\_reg2 \gg (31 - VPN)) \\
private\_seq\_reg2 &= (shared\_seq\_reg2 \ll VPN) | (shared\_seq\_reg3 \gg (31 - VPN)) \\
private\_seq\_reg3 &= (shared\_seq\_reg3 \ll VPN) | (shared\_seq\_reg0 \gg (30 - VPN))
\end{aligned}
$$

Once the sequence registers have been initialized, a stripmined loop processes the input data 16 bytes at a time. Each VP first uses a 32-bit *vector segment load* with a segment size of 4 to obtain its input data. An AIB is then fetched that XORs the 4 input words with the sequence registers to create 4 words of output and aligns the sequence registers for the next iteration. This alignment proceeds in the same manner as the initialization of the sequence registers described above, except that instead of using VPN as the shift amount, the current vector length (VLEN) is used. This is because to each VP, it must appear as if VLEN iterations of the loop have occurred between each coefficient it processes. The vector length is stored in a shared register and is updated by the CP at the beginning of each iteration of the stripmined loop.

The algorithm described above can only process input in chunks of 16 bytes. If the number of input bytes is not a multiple of 16, the remaining bytes must be processed on the control processor. To do this, the SCALE implementation first determines what the VPN of the next VP to execute would have been, had there been 16 more bytes of input data. The sequence registers are read out of this VP. The remaining input data is then XORed with the sequence registers in 32-bit portions until less than 4 bytes remain, at which point the last few bytes are processed one at a time.

4.1.2.2 CONVOLUTIONAL ENCODER

As with the SCRAMBLER kernel, the reference version of the CONVOLUTIONAL ENCODER kernel creates each bit of output individually from a set of input bits. In the case of the CONVOLUTIONAL ENCODER, however, a difficulty arises when we try to process multiple input bits at once because each input bit generates two output bits (the even and odd bits) which must appear interleaved in the output stream. To address this problem, the input bits must be *twinned* so that the position of each output bit is aligned with a copy of the corresponding input bit. This means that input bit 0 must be copied to positions 0 and 1, input bit 1 must be copied to positions 2 and 3, input bit 2 must be copied to positions 4 and 5, and so on as shown in Figure 4-1. Without special instructions, each bit must be copied individually and the process is very computationally expensive. To avoid having to carry out this computation, the twinned pattern for every possible input byte can be pre-computed and stored in a table. When an input byte is processed, this pattern can be loaded from the table, thus saving dozens of bit manipulation instructions. The table requires 256 16-bit entries and occupies 512 bytes of memory; it can therefore reside in the cache without significantly affecting performance.

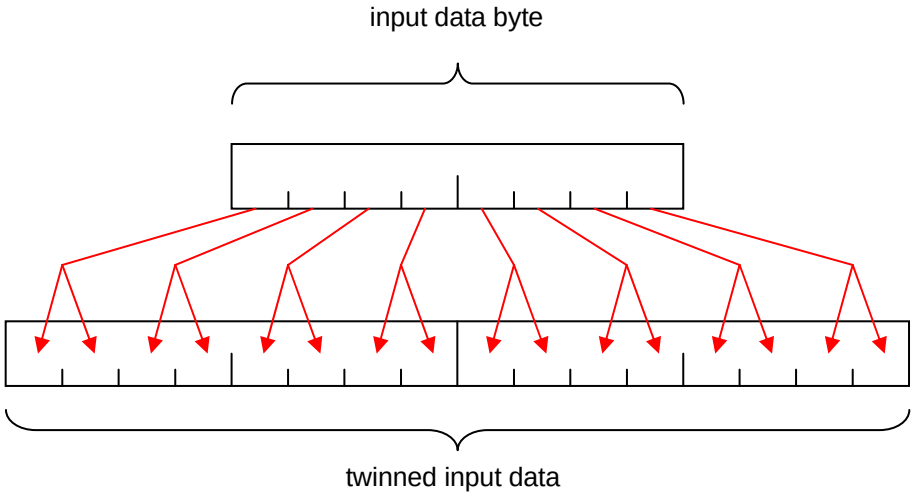

Figure 4-1: Input twinning in SCALE CONVOLUTIONAL ENCODER implementation.

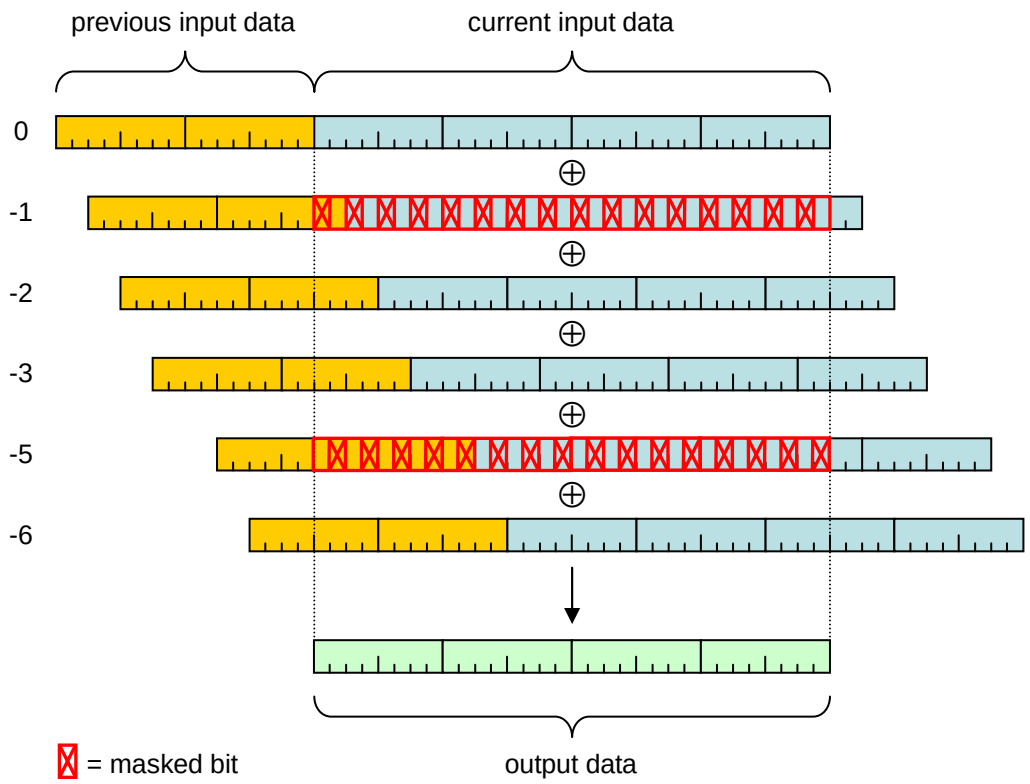

Figure 4-2: CONVOLUTIONAL ENCODER algorithm used in the SCALE implementation.

The SCALE implementation of the CONVOLUTIONAL ENCODER processes 16 bits of input data and generates 32 bits of output data at a time. In a stripmined loop, each VP reads in two bytes of input data and one byte of previous input data. Two AIBs are then fetched that twin the input data (as described above) and carry out the convolution using a series of shifts and XORs. This process is shown pictorially in Figure 4-2. After twinning, the current input data and previous input data are 32 bits wide and 16 bits wide respectively. Bit masks are applied in the -1 and -5 steps because the even output bits are not XORed with the -1 history position and the odd output bits are not XORed with the -5 history position. Once the output bits have been calculated, they are written to memory using a *unit stride vector store*. Because this algorithm only handles inputs in chunks of 2 bytes, if the number of input bytes is odd, the final byte must be processed on the control processor.

4.1.2.3 INTERLEAVER

As stated in Section 3.2.3, the INTERLEAVER kernel of the 802.11A TRANSMITTER reorders the 192 bits of a symbol according to the rule:

$$
\begin{aligned}
X &= 12 * (input\_index \mod 16) + (input\_index/16) \\
output\_index &= 2 * (X/2) + ((X + 192 - (X/12)) \mod 2) \\
& \quad symbol[output\_index] \leftarrow symbol[input\_index]
\end{aligned}
$$

However, it is inefficient to process the input bit-by-bit, so this rule cannot be implemented directly. Instead, assume that we are to read in 6 32-bit words of input data and generate 24 output bytes at a time. The outputs are formed from the input as shown in Figure 4-3. Further inspection reveals that this unrolled computation can be expresses as a loop, as shown in Figure 4-4. This loop forms the basis for optimizing the INTERLEAVER kernel.

In implementing the INTERLEAVER on the SCALE vector-thread architecture, we vectorized over the OFDM symbols, i.e., each VP is made responsible for one symbol (6 input words and 24 output bytes) and executes the loop shown in Figure 4-4 in its entirety. Each VP is first loaded with its input values using two 32-bit *vector segment strided loads* with a segment size of 3 and a stride of 24 (the number of bytes in an OFDM symbol). The VP then executes the loop, at each iteration outputting the three generated output bytes using an 8-bit *vector segment strided store* with a segment size of 3 and a stride of 24. The load could have been accomplished with a single *vector segment strided load* with a segment size of 6, but this would have required that all of the input words be loaded into the same cluster, which would have made load balancing of the bit extraction operations across the clusters more difficult.

```
    out00 = {in0[31]|in0[15]|in1[31]|in1[15]|in2[31]|in2[15]|in3[31]|in3[15]}
    out01 = {in4[31]|in4[15]|in5[31]|in5[15]|in0[30]|in0[14]|in1[30]|in1[14]}
    out02 = {in2[30]|in2[14]|in3[30]|in3[14]|in4[30]|in4[14]|in5[30]|in5[14]}

    out03 = {in0[29]|in0[13]|in1[29]|in1[13]|in2[29]|in2[13]|in3[29]|in3[13]}
    out04 = {in4[29]|in4[13]|in5[29]|in5[13]|in0[28]|in0[12]|in1[28]|in1[12]}
    out05 = {in2[28]|in2[12]|in3[28]|in3[12]|in4[28]|in4[12]|in5[28]|in5[12]}

    out06 = {in0[27]|in0[11]|in1[27]|in1[11]|in2[27]|in2[11]|in3[27]|in3[11]}
    out07 = {in4[27]|in4[11]|in5[27]|in5[11]|in0[26]|in0[10]|in1[26]|in1[10]}
    out08 = {in2[26]|in2[10]|in3[26]|in3[10]|in4[26]|in4[10]|in5[26]|in5[10]}

    out09 = {in0[25]|in0[09]|in1[25]|in1[09]|in2[25]|in2[09]|in3[25]|in3[09]}
    out10 = {in4[25]|in4[09]|in5[25]|in5[09]|in0[24]|in0[08]|in1[24]|in1[08]}
    out11 = {in2[24]|in2[08]|in3[24]|in3[08]|in4[24]|in4[08]|in5[24]|in5[08]}

    out12 = {in0[23]|in0[07]|in1[23]|in1[07]|in2[23]|in2[07]|in3[23]|in3[07]}
    out13 = {in4[23]|in4[07]|in5[23]|in5[07]|in0[22]|in0[06]|in1[22]|in1[06]}
    out14 = {in2[22]|in2[06]|in3[22]|in3[06]|in4[22]|in4[06]|in5[22]|in5[06]}

    out15 = {in0[21]|in0[05]|in1[21]|in1[05]|in2[21]|in2[05]|in3[21]|in3[05]}
    out16 = {in4[21]|in4[05]|in5[21]|in5[05]|in0[20]|in0[04]|in1[20]|in1[04]}
    out17 = {in2[20]|in2[04]|in3[20]|in3[04]|in4[20]|in4[04]|in5[20]|in5[04]}

    out18 = {in0[19]|in0[03]|in1[19]|in1[03]|in2[19]|in2[03]|in3[19]|in3[03]}
    out19 = {in4[19]|in4[03]|in5[19]|in5[03]|in0[18]|in0[02]|in1[18]|in1[02]}
    out20 = {in2[18]|in2[02]|in3[18]|in3[02]|in4[18]|in4[02]|in5[18]|in5[02]}

    out21 = {in0[17]|in0[01]|in1[17]|in1[01]|in2[17]|in2[01]|in3[17]|in3[01]}
    out22 = {in4[17]|in4[01]|in5[17]|in5[01]|in0[16]|in0[00]|in1[16]|in1[00]}
    out23 = {in2[16]|in2[00]|in3[16]|in3[00]|in4[16]|in4[00]|in5[16]|in5[00]}
```

Figure 4-3: Unrolled representation of the INTERLEAVER algorithm.

```
for(i = 0; i < 7; i++){
    out[3i+0] = {in0[31]|in0[15]|in1[31]|in1[15]|in2[31]|in2[15]|in3[31]|in3[15]}
    out[3i+1] = {in4[31]|in4[15]|in5[31]|in5[15]|in0[30]|in0[14]|in1[30]|in1[14]}
    out[3i+2] = {in2[30]|in2[14]|in3[30]|in3[14]|in4[30]|in4[14]|in5[30]|in5[14]}

    in0 <<= 2; in1 <<= 2; in2 <<= 2; in3 <<= 2; in4 <<= 2; in5 <<= 2;
}
```

Figure 4-4: The INTERLEAVER algorithm expressed as a loop.

4.1.2.4 MAPPER

As is evident from the description in Section 3.2.4 and Figure 3-9, the mapper has limited structured parallelism within an OFDM symbol. For this reason, the SCALE implementation of the mapper kernel vectorizes over the OFDM symbols, with each VP being responsible for one symbol in a stripmined loop.

Because each output symbol of the mapper includes an entry from the pilot vector (Figure 3-8) indexed by the symbol number, these entries are first loaded into the VPs using a 16-bit *unit stride vector load*. To ensure that this vector load does not attempt to read off of the end of the pilot vector, two identical copies of the table are stored contiguously in memory. If, during the course of the stripmined loop, the pointer to the next element to be loaded is incremented into the second table, it is reset to the corresponding position in the first table. This is permitted because the symbols index the pilot vector at their symbol number modulo 127, which is the size of the table.

Once each VP has acquired its pilot value, it builds its OFDM symbol in the 15 sections shown in Figure 4-5. The sections were chosen this way because it allowed all of the sections to be computed using only five simple AIBs, thus reducing code size and saving space in the SCALE AIB cache. To process each section (except for sections 1, 8, and 15), a 16-bit *strided vector load* is first performed to load the input data into the VPs. The appropriate AIB for the section is then fetched, which will perform the mapping using 4 table lookups. Finally, a 32-bit *vector segment strided store* is issued to write the output data to memory. The segment size for this store is 4 for the sections that do not contain pilot carriers, and 5 for the sections that do. Sections 1, 8, and 15 do not use any input data and serve only to write zeros to the appropriate places in the output symbol. Processing these section therefore does not require the vector load, only an AIB that loads zeros into the store-data registers and a *vector segment strided store* to write them to memory.

Section	Output Indices	Input Bytes	Description
1	0-5	∅	6 leading zeros
2	6-9	0-1	No pilots
3	10-14	2-3	Positive pilot at output 11
4	15-18	4-5	No pilots
5	19-22	6-7	No pilots
6	23-27	8-9	Positive pilot at output 25
7	28-31	10-11	No pilots
8	32	∅	Zero DC coefficient
9	33-36	12-13	No pilots
10	37-41	14-15	Positive pilot at output 39
11	42-45	16-17	No pilots
12	46-49	18-19	No pilots
13	50-54	20-21	Negative pilot at output 53
14	55-58	22-23	No pilots
15	59-63	∅	5 trailing zeros

Figure 4-5: Sections of the OFDM symbol used in the SCALE MAPPER.

4.1.2.5 IFFT

Although the IFFT kernel can be vectorized at the level of OFDM symbols, it also exhibits parallelism at a finer granularity: the radix-4 calculations. Because the input and output data of the IFFT is relatively large, we chose to vectorize over the radix-4 calculations to achieve long vector lengths without requiring many symbols to reside in the cache at once.

The strategy for implementing IFFT on SCALE was to have a single set of AIBs that perform the radix-4 calculation and use the different memory access patterns available in the SCALE ISA to group the data points appropriately. To maximize the possible vector length, each of the three stages runs once on the entire input set rather than having one OFDM symbol's worth of data passing through all 3 stages before proceeding to the next symbol.

To determine which memory access patterns and vector lengths are appropriate for each stage, we must understand which of the complex data points must be grouped together within a VP in each stage. Figure 4-6 shows which points are needed for each radix-4 calculation in the 3 stages of the algorithm.

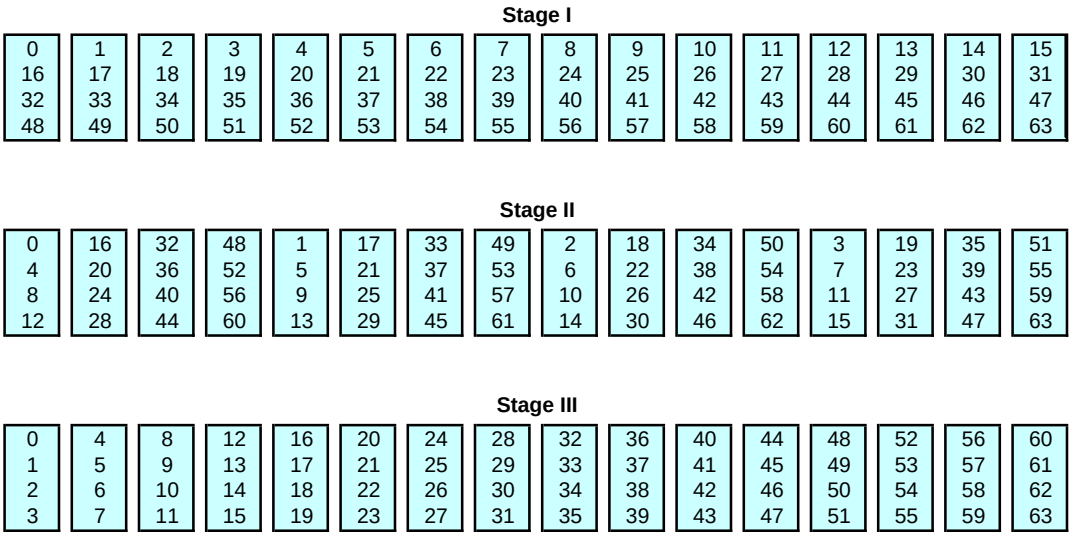

Figure 4-6: Grouping of complex data points for each radix-4 calculation in IFFT.

To load the data points into the VPs in Stage I, four *unit-stride vector loads* are performed to load the correct data into the A, B, C, and D radix-4 inputs respectively. Once the radix-4 calculation has been performed, its outputs are stored back to memory using vector stores which mirror these vector loads. This memory access pattern is shown pictorially in Figure 4-7. Because inputs 0 and 16 are loaded into the same VP, we must limit the vector length to 16 during Stage I.

If we were to adopt the same strategy for Stage II as we used for Stage I, we would be limited to a vector length of four because input 4 needs to be loaded into the same VP as input 0. A vector length of four is not sufficient to obtain acceptable performance, so another strategy was devised. This was to carry out Stage II in 4 different phases, as shown in Figure 4-7. To load the input values into the VPs in this pattern, four *strided vector loads* are performed each phase with a stride of 16 to load the correct data into the A, B, C, and D inputs. As is evident from the diagram, this approach does not limit the vector length and allows it to be set as high as the hardware will support. As in Stage I, the output values are vector stored in a manner analogous to the way in which the inputs were loaded.

The memory access pattern required for Stage III allows utilization of the *vector*

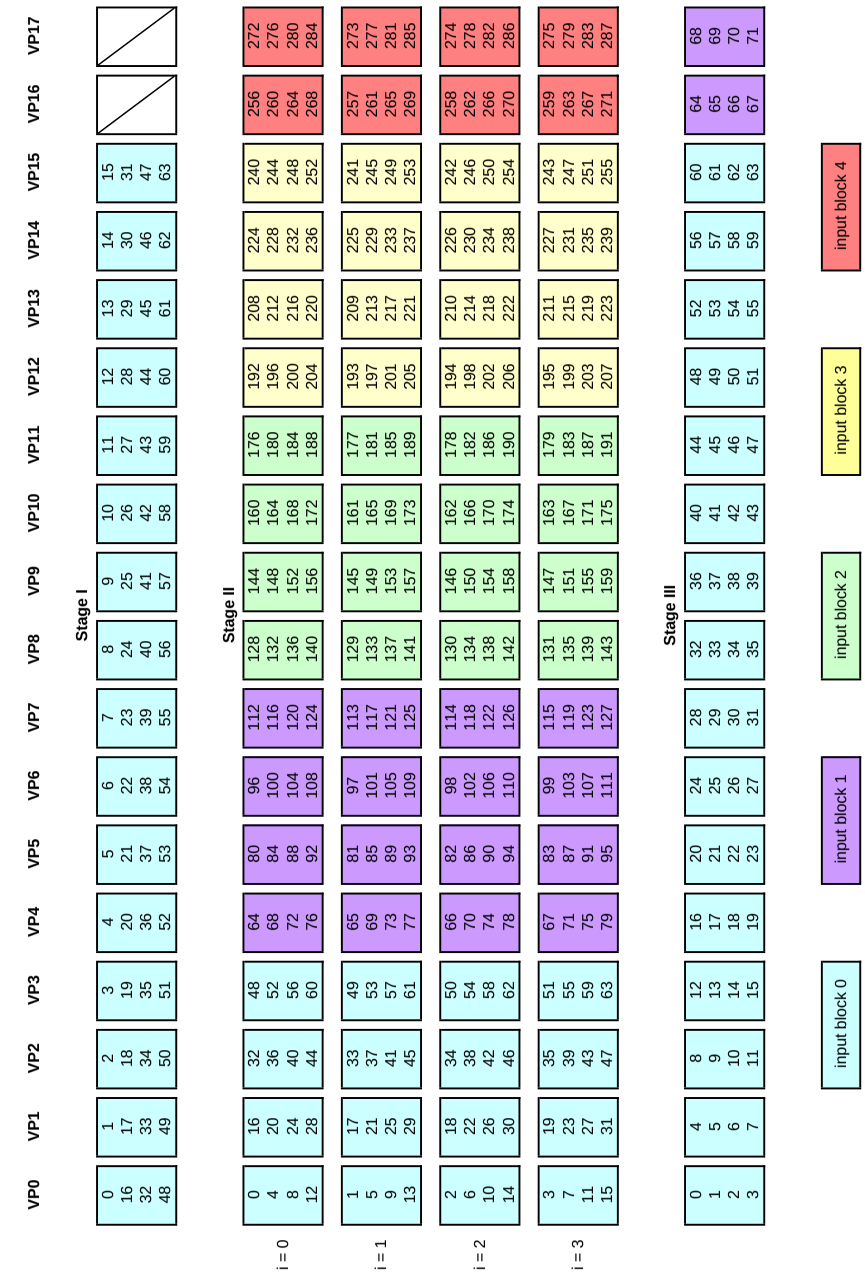

Figure 4-7: Grouping of complex data points for each radix-4 calculation in the SCALE implementation of IFFT.

segment load feature of the SCALE ISA to simply and efficiently load the input values into the required VPs. A single segment load with a segment size of 4 will load the correct input values into the A, B, C, and D inputs (provided that A, B, C, and D are consecutive private registers) of the radix-4 calculation. Without the *vector load segment* command, four separate, strided loads would be required to gather the data for this stage.

In contrast to Stages I and II, the outputs of Stage III are not stored in the same way that the inputs were loaded. This is because the output pattern of Stage III forms the first step of the reordering shown in Figure 4-8. The structure of this reordering makes it difficult to vectorize directly; however, if the reordering is split into two steps as shown in Figure 4-9, it can be vectorized. The first step is accomplished by storing the outputs of stage III to memory using four *unit-stride vector stores*. The second step loads 16 complex pairs of data into each VP using 2 *vector segment loads* of size 8, calls an AIB to reorder the data within each VP, and stores the elements back to memory using 2 *vector segment stores*.

The process described above will nearly accomplish the required reordering; the only problem is element zero, which must be mapped back to sub-carrier zero but cannot be directly because it is temporarily stored in sub-carrier 48. To map element zero to its correct destination, it is stored to sub-carrier 64 (the next OFDM symbol's zeroth position) in step 2, and another pass is made over the data to shift each sub-carrier zero up one OFDM symbol. Importantly, this allows all of the step 2 reordering AIBs to be the same (as is evidenced by the congruence of the step 2 reordering patterns in Figure 4-9).

It should be noted that because the output of Stage III must be stored to locations 16 elements apart, the vector length in Stage III is limited to 16. The vector length during the second step of the reordering is not limited algorithmically.

Figure 4-10 shows how the radix-4 calculation (first shown in Figure 3-10) was split between 6 AIBs. The computation was divided as such for two reasons: first, to allow the AIBs to fit in the SCALE AIB cache, and second, to allow computation to begin before all of the input data is loaded. For example, in Stage I, AIB `radix4_a`

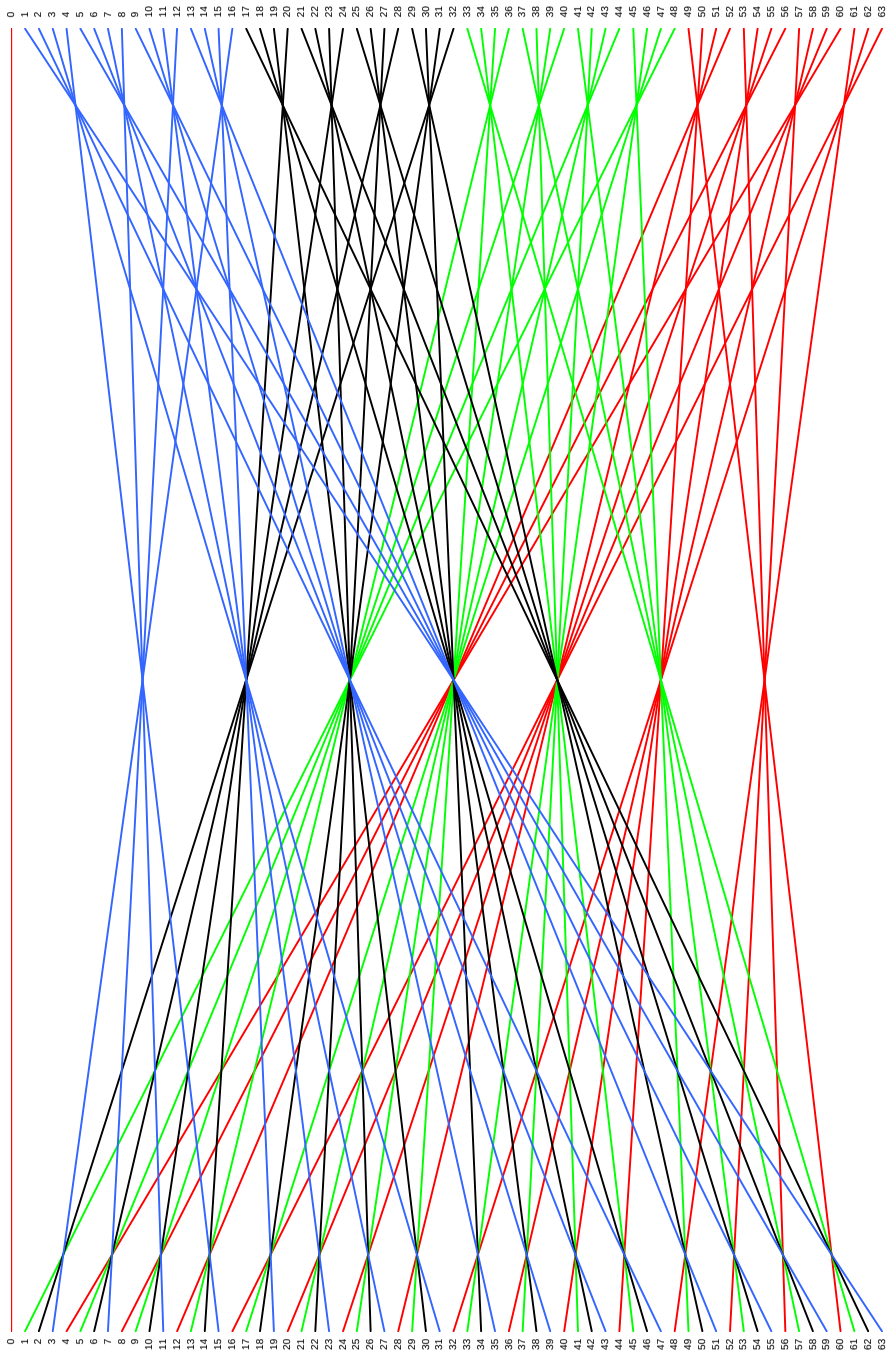

Figure 4-8: The data reordering that must be carried out at the end of the third stage of the IFFT benchmark (as described in Section 3.2.5).

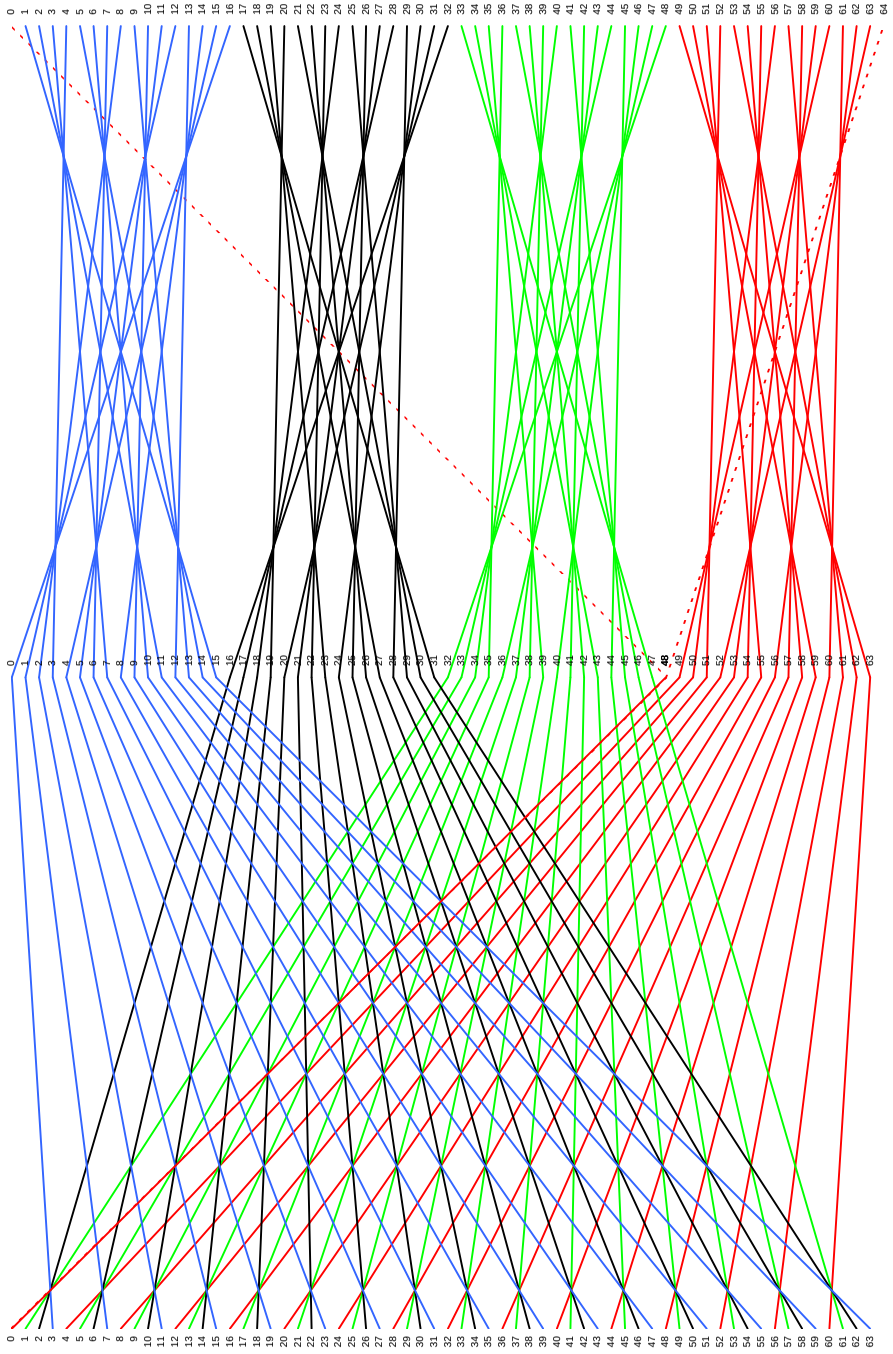

Figure 4-9: The reordering shown in Figure 4-8 split into two stages to allow for vectorization.

can be executed after the A inputs have been loaded, but before the B, C, and D inputs have been loaded. This allows some computation to take place in parallel with memory accesses.

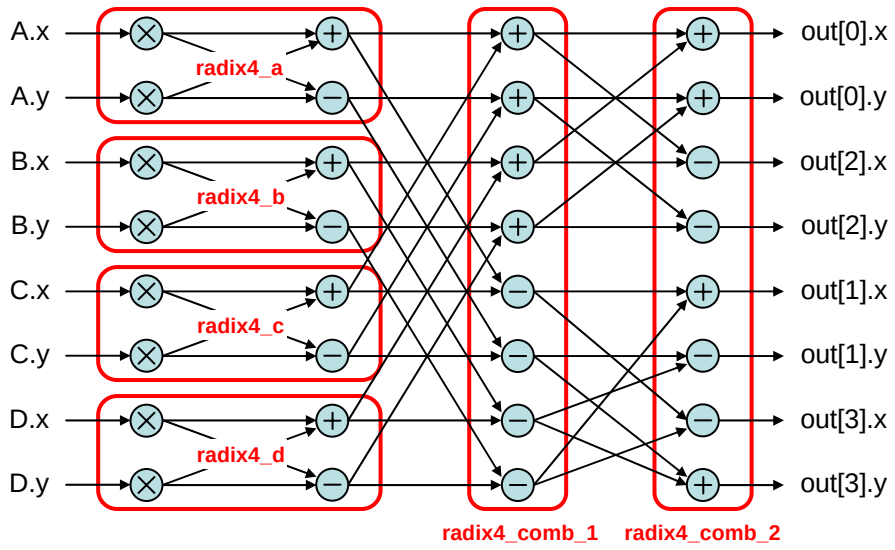

Figure 4-10: The division of the IFFT radix-4 calculation into AIBs.

4.2 Intel® XEON™

The Intel® XEON™ [29] family of processors execute the Intel IA-32 instruction set and were designed for dual-processor server and workstation platforms. The XEON was chosen as a platform on which to evaluate XBS because it represents a conventional architecture used in a wide variety of computing environments.

Although there exists a very sophisticated compiler that targets the XEON, hand-tuning is still necessary to achieve full performance on many applications. To optimize the XBS benchmarks for this platform, we used the Intel® Integrated Performance Primitives (IPP) library [26]. The IPP library contains assembly optimized routines from the signal processing, image and video compression, mathematical, and cryptographic computing domains. The library has versions of these routines for many different Intel processors and uses a common API to simplify porting an IPP optimized

application between platforms. To achieve maximum performance on the optimized routines, the IPP library makes use of numerous performance tuning techniques [28], including:

- Pre-fetching and cache blocking
- Avoiding data and trace cache misses
- Avoiding branch misprediction
- Utilizing Intel® MMX™ and Streaming SIMD Extensions (SSE) instructions
- Using Hyper-Threading Technology
- Algorithmic tuning and memory management

The IPP optimizations applied to the XBS benchmarks in order to optimize performance on the XEON platform are listed in the following sections. Unfortunately, Intel does not provide documentation of the specific optimization techniques used for each IPP routine; this information will therefore be absent from the descriptions below.

4.2.1 CJPEG

The IPP optimized version of the CJPEG application benchmark is provided by Intel [27]. The IPP library contains many optimized routines that have been specifically developed for JPEG compression, and these routines are used to optimize the CJPEG benchmark. For information on these, and other IPP optimized image processing routines, refer to the *IPP Image and Video Processing Reference Manual* [25].

Intel provides IPP optimized routines for all four of the CJPEG application benchmark's kernels: RGBYCC, FDCT, QUANTIZE, and ENCODE. In addition, the IPP library includes an optimized version of the downsampling process. Downsampling, as mentioned in Section 3.1, reduces the resolution of the Cb and Cr color components. While downsampling accounts for a smaller fraction of the CJPEG run-time

than any of the four kernels included in XBS, it can still affect the performance of CJPEG, especially once the other kernels have been optimized.

4.2.2 802.11A TRANSMITTER

Of the six kernels which comprise the 802.11A TRANSMITTER benchmark, only one has an optimized implementation in the IPP library: IFFT. The other kernels in the XEON optimized version of 802.11A TRANSMITTER are therefore optimized using only C code. For more information on the IPP optimized version of IFFT refer to the *IPP Signal Processing Reference Manual* [24].

Besides IFFT, the three kernels that were optimized are SCRAMBLER, CONVOLUTIONAL ENCODER, and INTERLEAVER. The optimizations performed were primarily to process the data 16 or 32 bits at a time instead of one bit at a time as the kernels are defined in Section 3.2. These optimizations are nearly identical to those described above for the SCALE architecture, less vectorization, and so will not be covered again here. The other two kernels of the 802.11A TRANSMITTER application benchmark, MAPPER and CYCLIC EXTEND, are already defined to process more than one bit at a time, and were therefore not optimized for the XEON platform.

4.3 Hardware ASIC

The ASIC implementations of the XBS benchmarks were developed using the Verilog, VHDL, and Bluespec [8] hardware description languages. These implementations are simulated using Synopsys VCS [20] to verify correctness and measure performance. To determine cycle time and area, the implementations are synthesized using the Synopsys Design Compiler [21] and placed and routed using Cadence Encounter [22].

Verilog and VHDL are well known hardware description languages and, as such, will not be introduced here. Bluespec is a new hardware description language being developed at MIT and Bluespec, Inc. The Bluespec language is intended to address the problem of complexity in large hardware design projects by allowing high-level system descriptions to be translated into high-quality RTL. Bluespec modules consist

of explicitly declared state elements which are acted upon by guarded atomic actions. Hardware designs written in Bluespec can be translated into Verilog for use in the tool chain described above.

The following sections describe how the XBS benchmarks were implemented in hardware:

4.3.1 CJPEG

The hardware ASIC version of the CJPEG application benchmark is a VHDL open source implementation freely available from the OPENCORES.ORG [35] repository of open source IP cores. The following sections provide a brief overview of the design choices made for each kernel of CJPEG; for more complete information, consult the documentation and source code available from the project's web site [34].

4.3.1.1 RGBYCC

The first process in the ASIC implementation of CJPEG converts the input pixels from their RGB color representation to their Y/Cb/Cr representation and downsamples the chrominance components. The conversion takes three cycles per pixel, reading a new pixel from the input data stream each iteration. When it has accumulated a complete block of 64 pixels, it signals the FDCT to begin processing and moves on to the next block.

4.3.1.2 FDCT

The FDCT kernel was implemented using the Xilinx LogiCORE 2-D Discrete Cosine Transform module [23]. The module takes as input a block of 64 8-bit signed values and outputs the frequency domain representation of the block as 64 19-bit signed values. The parameters with which the module was instantiated are listed below:

- Coefficient Width = 24
- Precision Control: Round

- Internal Width = 19
- Clock Cycles per input = 9

The module instantiation exhibits the following latencies:

- Latency = 95 cycles
- Row Latency = 15 cycles
- Column Latency = 15 cycles

For more information on the workings of this LogiCORE library module, refer to the product specification cited above.

4.3.1.3 QUANTIZE

As discussed in Section 4.1.1.3, nearly all of the complexity of the QUANTIZE kernel lies in the many divisions that must be performed. Like the SCALE version, the ASIC version of QUANTIZE takes advantage of the observation that the quantization factors are infrequently changed by pre-computing and multiplying by the reciprocals of the quantization factors. Each reciprocal is computed as a fraction with a denominator of 16384 ($= 2^{14}$). To perform the quantization, the coefficient to be quantized is multiplied by the numerator of this fraction and shifted right by 14 bits.

4.3.1.4 ENCODE

After quantization, the coefficients are encoded using DPCM, RLE and Huffman Coding. The hardware implementation of the ENCODE kernel operates sequentially on the coefficients of the block, updating the count of consecutive zeros and, if necessary, outputting the appropriate Huffman symbols for the data. This process can take place in parallel with the FDCT for the following block of data.

4.3.2 802.11A TRANSMITTER

The hardware ASIC version of the 802.11A TRANSMITTER application benchmark was implemented in Bluespec as part of a class project [2] in the Complex Digital Systems class at MIT. The sections below briefly describe the implementation of the SCRAMBLER, CONVOLUTIONAL ENCODER, INTERLEAVER, MAPPER, IFFT, and CYCLIC EXTEND kernels in hardware. For more details, consult the report cited above.

4.3.2.1 SCRAMBLER

Instead of scrambling each input bit one at a time as shown in Figure 3-5, the hardware implementation of the SCRAMBLER kernel is designed to process 24 bits of input in one pass. At the beginning of each message, the implementation first generates the entire 127-bit scramble sequence from the seed and stores it in a circular shift register. After the initialization stage, 24 bits of data are read in per cycle and XORed with the corresponding bits of the scramble sequence to form the output. At the end each cycle, the circular shift register is rotated left by 24 bits to prepare for the next set of input bits.

4.3.2.2 CONVOLUTIONAL ENCODER

Like the SCRAMBLER kernel, the hardware implementation of the CONVOLUTIONAL ENCODER encodes one 24-bit frame of input data every cycle rather than processing the input one bit and a time. Because the convolutional coding rate is 1/2, this means that the module generates 48 bits of output data each cycle.

4.3.2.3 INTERLEAVER

The INTERLEAVER must process the data in OFDM symbols, 192 bits at a time. Because the previous module (the CONVOLUTIONAL ENCODER) generates 48 bits of data at a time, the INTERLEAVER must buffer this input until it has received 4 cycles worth. At this point, it can reorder the data according to the formula in Section

3.2.3 and output the interleaved data. Because the reordering pattern is static and known ahead of time, the hardware for the interleaver simply hardwires the input bit positions to their corresponding output positions.

4.3.2.4 MAPPER

The MAPPER kernel reads in 192 bits of interleaved, encoded data and outputs 64 complex pairs to the following stage (the IFFT kernel). The hardware implementation of the MAPPER is able to process 48 input bits each cycle except for one dead cycle during which the module sends its output to the next stage. The MAPPER can therefore process one OFDM symbol every 5 cycles.

4.3.2.5 IFFT

The IFFT kernel accounts for the majority of the area and sets the critical path of the hardware implementation of the 802.11A TRANSMITTER benchmark. The IFFT has one 16 node stage of the computation fully unrolled in hardware. This means that the hardware exists to carryout 16 radix-4 calculations simultaneously. Because the calculation of the IFFT requires three stages, the data passes through the one instantiated stage three times, being reordered at each iteration so that the correct points are involved in each radix-4 calculation.

4.3.2.6 CYCLIC EXTEND

The CYCLIC EXTEND kernel is simple to implement in hardware. The complex pairs that must be copied are simply hardwired to the appropriate locations according to the description in Section 3.2.6.

Chapter 5

Experimental Results

This chapter describes a set of experiments designed both to evaluate the Extreme Benchmark Suite itself, and the platforms to which it has been ported. The SCALE, XEON, and ASIC implementations of the XBS benchmarks described in Chapter 4 were utilized in our experiments.

In order to evaluate XBS as a benchmark suite, we must first show that the claims made in Chapters 1 and 2 regarding the motivation for the design of XBS are supported by the data. In addition, we must show that benchmarks provide an insight into the relative strengths and weaknesses of the platforms being compared. To this end, experiments were designed to answer the following questions:

- What percentage of the application benchmarks' runtime is accounted for by the kernels?
- How much performance is gained from hand-optimization over compiled code?
- How do the different platforms compare in their ability to run the XBS benchmarkss?

The technical details of each platform used in our experiments are described in the next section. The following three sections describe the experiments conducted to determine the runtime breakdown of the application benchmarks, measure the

speedup gained by hand-optimization, and compare the architectures being benchmarked. Each of these sections discusses the experimental setup, presents the data collected from the experiments, and provides analysis of that data.

It should be noted that we do not, in these experiments, examine the power and energy expended by the platforms while running the XBS benchmarks. Power and energy are, however, important factors in embedded system design and, as such, future versions of XBS will be equiped to measure these parameters.

5.1 Experimental Setup

The details of the three platforms on which we evaluated XBS are listed in the following sections:

5.1.1 SCALE

As stated in Section 4.1, a hardware implementation of the SCALE vector-thread architecture is not yet available. Instead, the experiments involving the SCALE platform were carried out in simulation. The SCALE simulator is a detailed cycle-level, execution-driven microarchitectural simulator based on the prototype SCALE design. It models many of the microarchitectural details of the CP and VTU and is complemented by a cycle-based memory system simulation which models the multi-requester, multibanked, non-blocking, highly-associative CAM-based cache and a detailed memory controller and DRAM model [30]. Based on preliminary estimates, the SCALE chip will be able to run at 400 MHz, and this clock rate is assumed in the results in the following sections.

While there is not yet a compiler that can target the advanced architectural features of SCALE, a version of gcc has been created to compile scalar code to be run on the CP. This code is then linked with hand-written assembly code that targets the VTU to create the optimized SCALE implementations of the XBS benchmarks. Basic operating system facilities, including I/O, are provided by a proxy kernel that intercepts system calls made by applications running on the SCALE simulator.

5.1.2 XEON

The XEON platform tested in out experiments is a dual-processor, rack mounted server that is part of a compute farm in the Computer Architecture Group at MIT. Each processor operates at 3.6 GHz, although the XBS benchmarks use only one processor at a time. The machine runs Red Hat Enterprise Linux 3 [19] and uses version 2.4.21 of the Linux kernel.

All code for the XEON platform was compiled using gcc version 2.95.3 with -O3 optimizations enabled. The optimized versions of CJPEG and 802.11A TRANSMITTER make use of the Intel Integrated Performance Primitives libraries, as described in Section 4.2, which are dynamically linked at runtime.

5.1.3 ASIC

As stated in Section 4.3, the hardware ASIC implementations of the CJPEG and 802.11A TRANSMITTER benchmarks were written in the VHDL and Bluespec hardware description languages, respectively. After synthesizing the two designs using TSCM 130 nm process parameters, it was determined that they could run at the following clock rates:

CJPEG	214.6 MHz
802.11A TRANSMITTER	30.4 MHz

The ASIC implementation runtimes listed in Section 5.4 were calculated assuming these clock rates. The cycle counts used in these calculations were obtained from the simulators for each benchmark which were generated by VCS. The seemingly slow clock rate of the 802.11A TRANSMITTER implementation is due to the fact that the IFFT stages were left unpipelined to save area and power. Despite the slow clock rate, the design is still easily able to meet the real-time timing constraints of the application.

Kernel	SCALE	XEON
RGBYCC	19.3%	11.0%
FDCT	26.3%	8.0%
QUANTIZE	14.8%	31.8%
ENCODE	27.4%	35.9%
kernel total	**87.8%**	**86.7%**
other	12.2%	13.3%

Table 5.1: CJPEG runtime percentage breakdown.

5.2 Application Runtime Breakdown

It was claimed in Section 2.1 that the effort involved in porting and hand-optimizing application benchmarks is drastically reduced by separating out the important kernels so that they may be developed and tested independently. This is only true if a very large portion of the application's runtime is accounted for by these kernels. To show that this holds true for the two application benchmarks in XBS, CJPEG and 802.11A TRANSMITTER, the benchmarks were instrumented with IPM [1] timers to measure the contribution of each kernel to the total runtime.

The instrumented benchmarks were tested on the XEON and SCALE platforms with a variety of input sizes. For each platform, the runtime breakdown remained relatively constant over the different input sizes. The percentage of application runtime for each kernel is listed for the CJPEG benchmark in Table 5.1 and for the 802.11A TRANSMITTER benchmark in Table 5.2. The percentages shown in these tables are the geometric mean of the runtime percentages for the different input sizes.

Although the runtime breakdown can vary between platforms, it is clear that the kernels account for the vast majority of the overall application runtime (more than 85% for CJPEG and almost 100% for 802.11A TRANSMITTER). This means that significant performance incrases can be attained by optimizing only the kernels of an application. This can greatly reduce the effort in porting an application such as CJPEG that is comprised of nearly 24,000 lines of code, but has only 4 important kernels.

Kernel	SCALE	XEON
SCRAMBLER	10.3%	12.5%
CONVOLUTIONAL ENCODER	24.3%	29.8%
INTERLEAVER	16.9%	22.1%
MAPPER	4.0%	3.3%
IFFT	43.6%	31.2%
CYCLIC EXTEND	0.9%	1.0%
kernel total	**100.0%**	**99.9%**
other	0.0%	0.1%

Table 5.2: 802.11A TRANSMITTER runtime percentage breakdown.

5.3 Hand Optimization Speedup

As discussed in Sections 1.1 and 2.1, many embedded platforms rely on hand-optimized assembly code to achieve their full performance potential. This can be due to the fact that a compiler that is able to target the advanced architectural features of the platform does not exist. However, even if such a compiler does exist, a hand-tuned implementation of an application may still run significanly faster than the compiled version. The extra labor cost of hand-tuning applications for a specific platform is justified in the embedded computing domain, where the same application will be shipped on millions of units.

To demonstrate the importance of hand-tuning to the performance of the XBS benchmarks, the kernel benchmarks were tested on the SCALE and XEON platforms in both their unoptimized and optimized implementations. Some of the kernels were optimized using assembly code, some were optimized using only C code, and some have both assembly optimized and C optimized versions. Figures 5-1, 5-2, 5-3, and 5-4 show the speedups of the XBS kernel benchmarks on the two platforms over a range of input sizes. The implementations of the benchmarks in these charts are given the following codes:

- **REF** – Unoptimized reference version
- **OPTC** – Optimized using C source code optimizations only
- **VTU** – Assembly optimized for SCALE utilizing the vector-thread unit

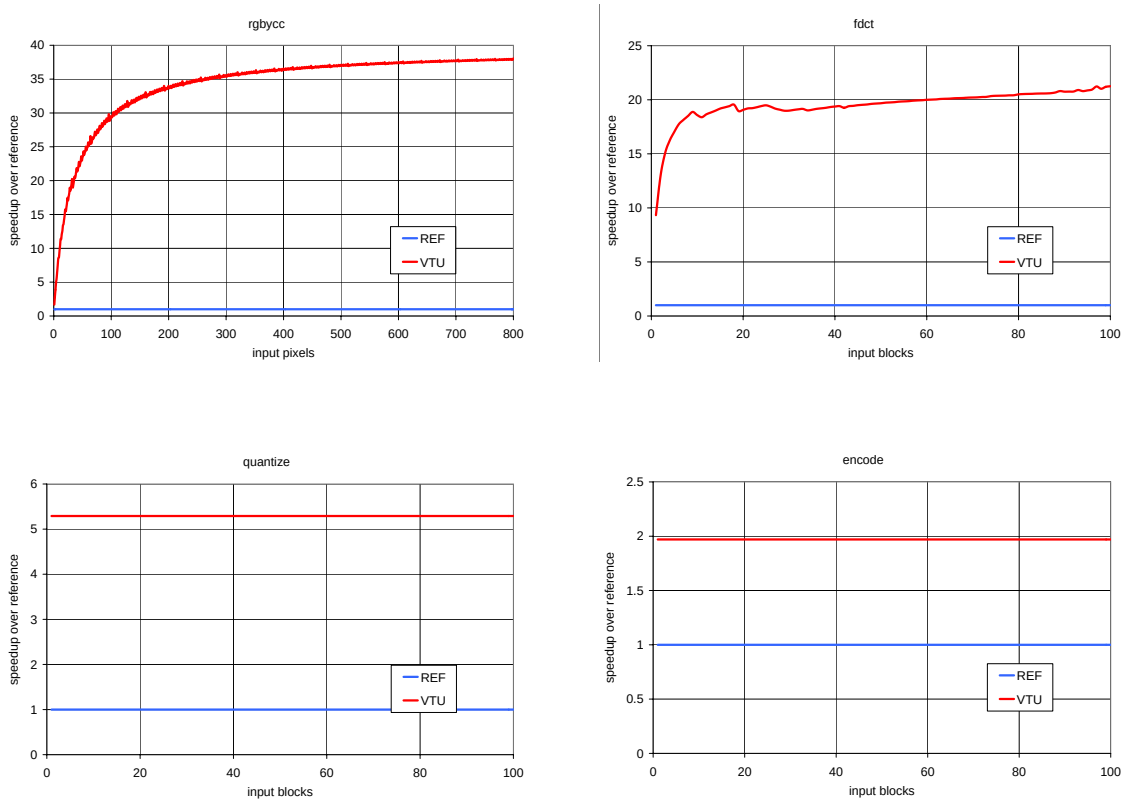

Figure 5-1: SCALE cjpeg charts

- **IPP** – Optimized for XEON using the Intel Integraed Performance Primitives library

Figure 5-1 shows the speedup of the assembly optimized kernels of CJPEG over the compiled reference versions on the SCALE platform. The method employed in optimizing each kernel is described in Section 4.1. The RGBYCC and FDCT kernels are able to achieve large speedups because these processes are highly vectorizable. As the size of the input increases, the optimized versions of RGBYCC and FDCT become faster relative to the reference versions because they are able to process the data using a larger vector length. The optimized versions of QUANTIZE and ENCODE, on the other hand, are only able to process one block of input at a time so their performance does not improve with an increase in input size. Because ENCODE contains a large amount of code that must be executed serially (as described in Section 4.1.1.4), the optimized version is only able to achieve a speedup of 2x over the reference version.

The performance of the SCALE implementations of the 802.11A TRANSMITTER

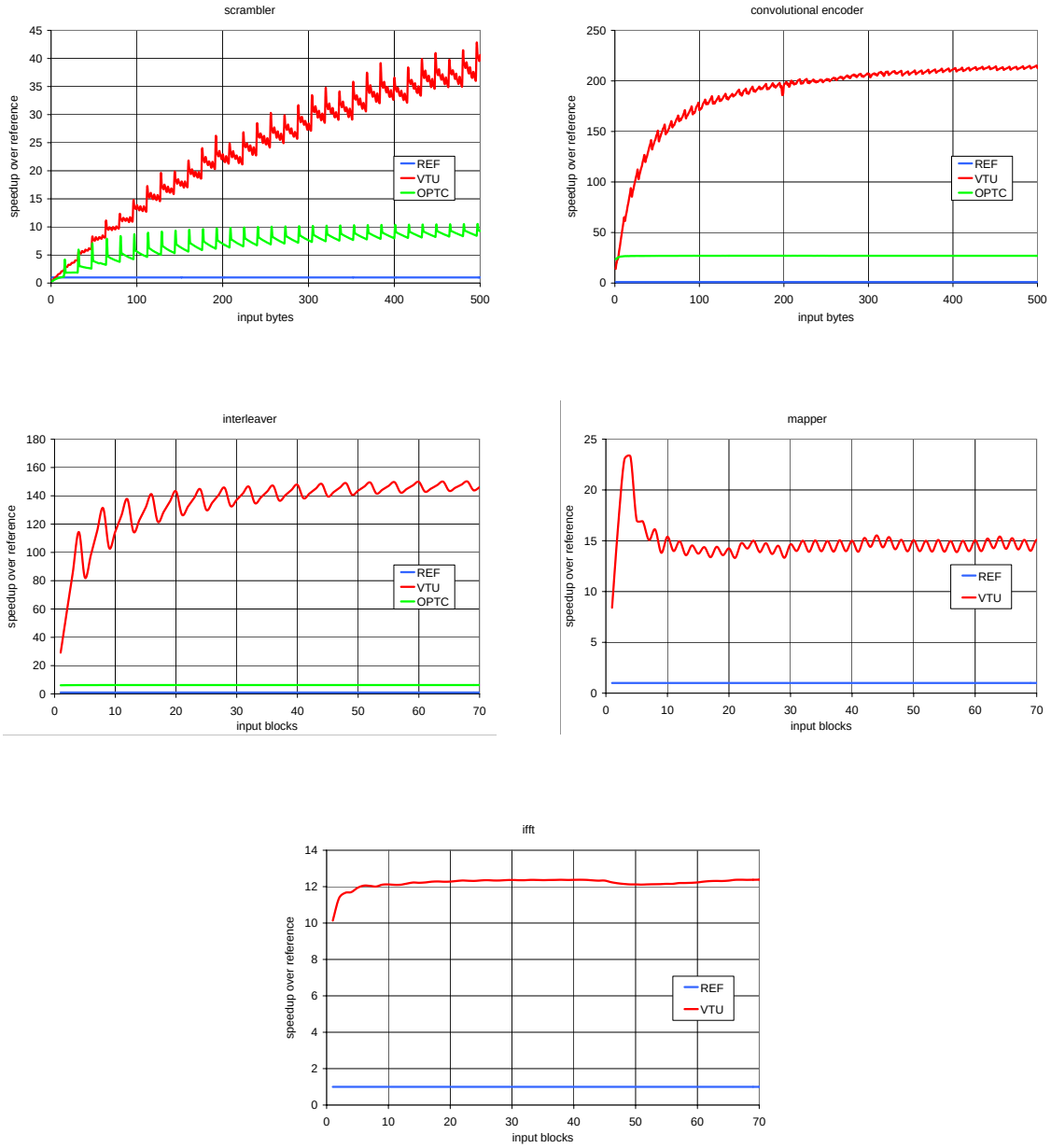

Figure 5-2: SCALE 802.11a transmitter charts

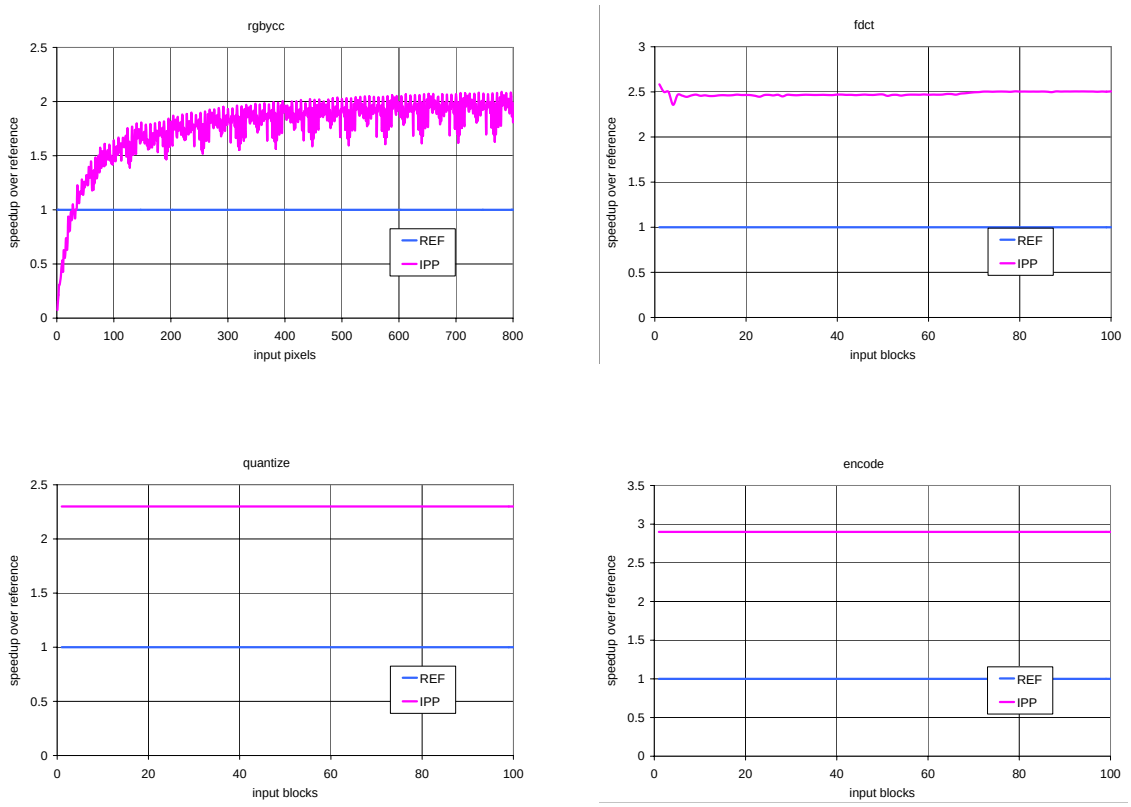

Figure 5-3: XEON cjpeg charts

kernels is shown in Figure 5-2. The SCRAMBLER, CONVOLUTIONAL ENCODER, and INTERLEAVER kernels have C code optimized implementations in addition to the reference and assembly language optimized versions. As with the kernels of CJPEG, the relative performance of the optimized versions improves as the input size is increased. Unlike the CJPEG kernels, however, the optimized implementations of the SCRAMBLER, INTERLEAVER, and MAPPER appear to exhibit a periodic component in their performance. This can be explained by the fact that these implementations process the input in chunks and when the size of the input is not a multiple of the size of the chunk, the remainder must be processed as a special case, which can be inefficient. This means that, for example, if the chunk size was 16 bytes, the implementation would be able to process 64 or 80 bytes very efficiently, but would not fare well on inputs of 63 bytes because 15 bytes would need to be special-cased.

Figure 5-3 shows the CJPEG kernels optimized for the XEON platform using the Intel IPP library. Because Intel does not provide documentation of the specific op-

timizations performed on each routine, we cannot definitively explain the trends of these charts. However, from the periodicity exhibited by the RGBYCC implementation, it is likely that it was designed to process the input in chunks, as described in the preceding paragraph. Because the optimized routines for FDCT, QUANTIZE, and ENCODE are only able to process one input block at a time, their relative performance does not vary with the input size.

Finally, the XEON optimized implementations of the 802.11A TRANSMITTER kernels are shown in Figure 5-4. As stated in Section 4.2.2, the SCRAMBLER, CONVOLUTIONAL ENCODER, and INTERLEAVER kernels were C code optimized for the XEON and the IFFT kernel was optimized using an IPP library routine. The charts of the C code optimized kernels follow nearly the same pattern as their SCALE counterparts in Figure 5-2. Although the IPP version of IFFT can process only one block of input at a time, its relative performance does increase with the input size because the rather signficant cost of the initialization code becomes amortized.

5.4 Platform Comparison

To compare the SCALE, XEON, and ASIC platforms, the two XBS application benchmarks were run on all three platforms over a range of input sizes.

The input images used to test the CJPEG implementations range in size from 128x128 (48 KB) to 1024x1024 (3 MB). To evaluate the 802.11A TRANSMITTER implementations, input sequences of 240 bytes to 480 KB were used. Becuase each input byte to the 802.11A TRANSMITTER is transformed into $\frac{1}{12}^{th}$ of an output OFDM symbol, or 27 bytes, the sizes of the outputs in our experiment range from 6.3 KB to 12.4 MB.

The runtimes (in milliseconds) of the CJPEG application benchmark on each of the three platforms are shown in Table 5.3 and Figure 5-5. As expected, the optimized SCALE and XEON implementations each out-perform their unoptimized counterparts (by roughly a factor of 3.1x). The ASIC implementation runs 2.6x faster than SCALE VTU and slightly out-performs XEON REF, but is 2.5x slower than XEON

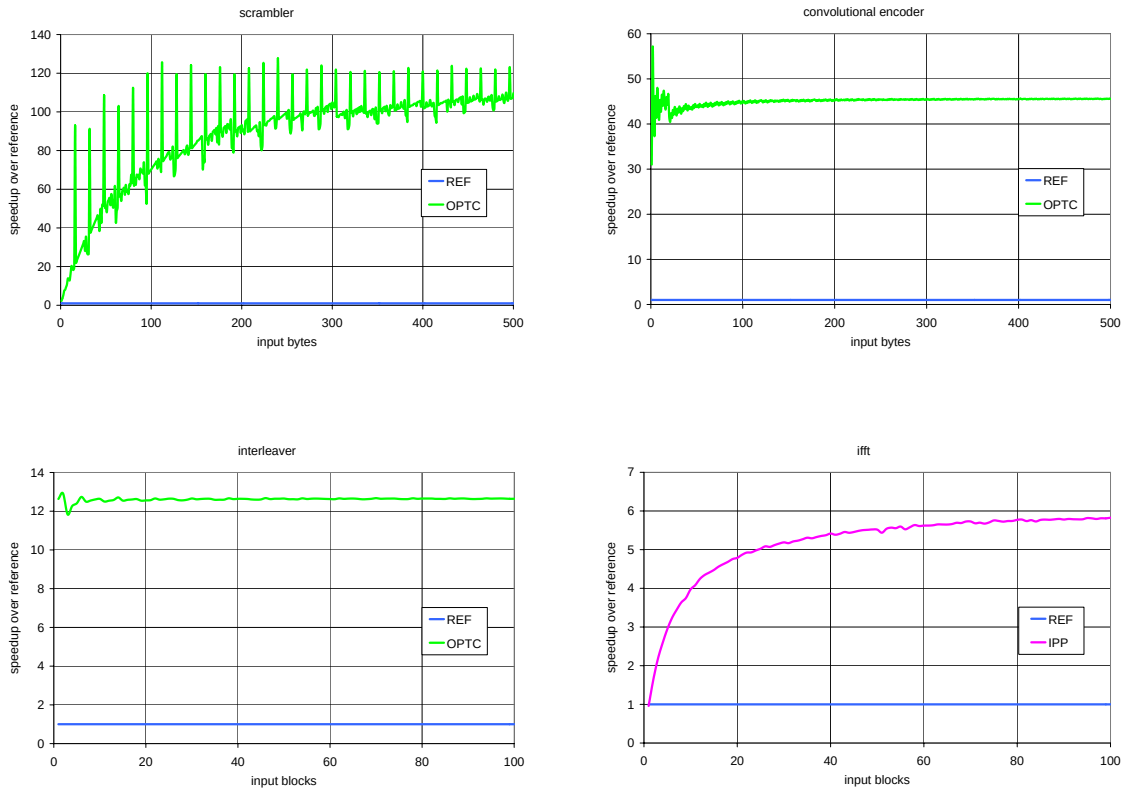

Figure 5-4: XEON 802.11a transmitter charts.

IPP. The XEON performs well relative to the other platforms on CJPEG because its high clock rate allows it to process the serial portions of the algorithm much more quickly than its competitors.

Table 5.4 and Figure 5-6 show the runtimes of the 802.11A TRANSMITTER benchmark on the three platforms. For the SCALE and XEON platforms, the OPTC versions were used in place of the REF versions because they represent the highest performance on these platforms without hand-optimization. As with CJPEG, the XEON optimized implementation out-performs the unoptimized version by a factor of approximately 3.1x. The SCALE VTU version of 802.11A TRANSMITTER, however, runs 12.3x faster than the SCALE OPTC version. This large speedup is due to the high degree of parallelism present in the algorithm, which allows the SCALE VTU implementation to take advantage of SCALE's numerous compute resources.

In contrast to CJPEG, the ASIC version of 802.11A TRANSMITTER significantly out-performs its competitors (XEON IPP by a factor of 5.1x and SCALE VTU by

Image Size	**XEON**		**SCALE**		**ASIC**
(pixels)	IPP	REF	VTU	REF	
128 x 128	0.52	1.37	3.09	9.35	1.11
128 x 256	0.97	2.88	6.07	18.68	2.26
256 x 128	0.96	2.94	6.12	18.77	2.28
256 x 256	1.90	5.93	12.02	37.35	4.57
256 x 1024	7.42	23.12	47.08	148.34	18.25
1024 x 256	7.27	23.36	46.46	147.63	18.55
512 x 512	7.28	23.18	46.20	147.42	18.27
512 x 1024	14.15	46.15	91.86	294.12	36.51
1024 x 512	14.22	46.71	92.41	294.64	37.08
1024 x 1024	28.68	93.21	184.12	588.35	74.15

Table 5.3: CJPEG runtimes (ms)

Input Size	**XEON**		**SCALE**		**ASIC**
(bytes)	IPP	OPTC	VTU	OPTC	
240	0.03	0.08	0.10	1.10	< 0.01
480	0.06	0.17	0.18	2.16	0.01
1200	0.14	0.42	0.44	5.29	0.03
2400	0.26	0.84	0.85	10.56	0.05
4800	0.53	1.66	1.69	21.09	0.11
12000	1.29	4.20	4.20	52.66	0.27
24000	2.84	8.39	8.41	105.38	0.53
48000	5.33	16.83	16.77	210.70	1.07
120000	13.50	42.14	41.95	526.53	2.67
240000	27.01	84.39	83.70	1052.78	5.35
480000	55.03	167.73	167.39	2105.88	10.70

Table 5.4: 802.11A TRANSMITTER runtimes (ms)

16.1x). The SCALE VTU implementation exhibits nearly the same performance as the XEON OPTC version, but is about 3 times slower than XEON IPP. The XEON does not perform as well as it does on CJPEG relative to the other platforms because the SCALE and ASIC implementations can exploit the abundant parallelism present in the 802.11A TRANSMITTER benchmark more effectively than the XEON, thus making up for their slower clock frequencies.

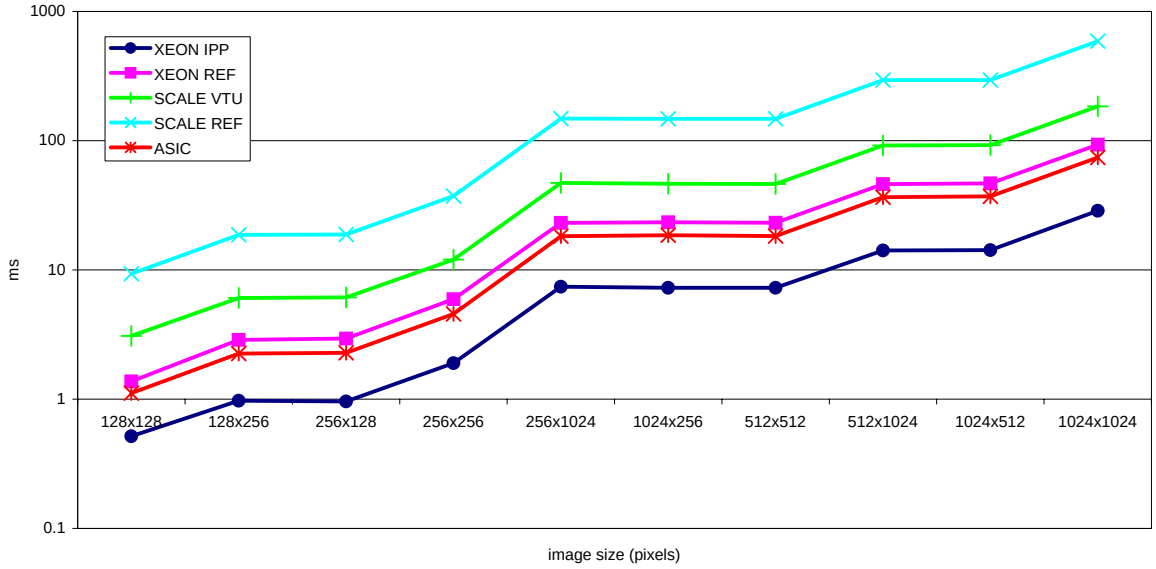

Figure 5-5: CJPEG runtimes

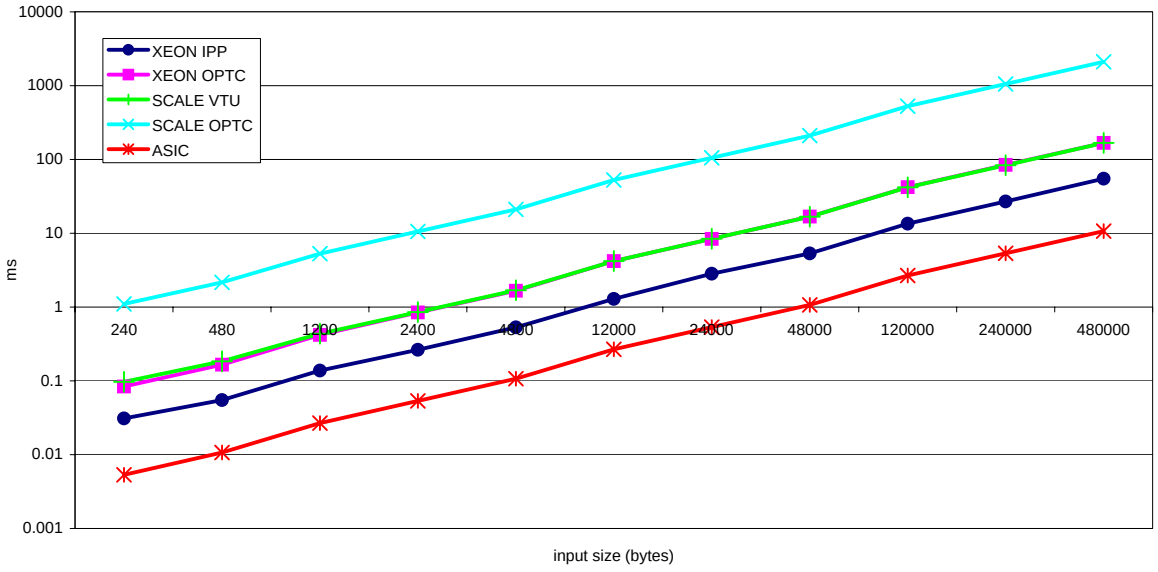

Figure 5-6: 802.11A TRANSMITTER runtimes

Chapter 6

Conclusions

This thesis has presented the Extreme Benchmark Suite, a new benchmark suite designed to evaluate highly parallel, experimental, and non-standard architectures used in embedded computing environments. The difficulties involved in using conventional benchmark suites for this task were discussed, and the design of XBS was developed to address these issues in Chapters 1 and 2. The CJPEG and 802.11A TRANSMITTER application benchmarks, taken from the image processing and communications embedded computing domains, were presented along with their associated kernels in Chapter 3. Chapter 4 went on to chronicle the implementation of these benchmarks on three platforms: the SCALE vector-thread architecture, the INTEL XEON processor, and ASIC hardware. Finally, Chapter 5 presented the experiments designed to evaluate XBS and the platforms to which it has been ported.

From the results of our experiments, it is clear that the statments made as motivation for the creation of XBS are supported by the data; namely, that hand-tuning is required to achieve full performance on embedded applications and that porting an application to a new architecture can be dramatically simplified if the computationally intensive kernels are exposed. In addition, from our experience in porting the XBS benchmarks to the three platforms studied, we have verified that the separation of input generation, output verification, and actual benchmark code greatly reduces porting effort.

By implementing the XBS benchmarks on a standard platform (XEON), an ex-

perimental architecture (SCALE), and in custom hardware, we have shown that the XBS framework is flexible enough to provide a fair comparison of a wide range of embedded computing platforms. For these reasons, it is clear that XBS can be an invaluable tool for benchmarking embedded systems.

6.1 Future Work

Although XBS in its current state is a fully featured, usable system, it is still very much a work in progress. We plan to add more kernel and application benchmarks to broaden the range of embedded application domains that XBS encompasses. Two applications that are slated for inclusion in the near future are MPEG [7] video compression and the OpenGL ES [12] graphics pipeline for embedded systems. MPEG has some of the same kernels as JPEG and also includes motion estimation. OpenGL ES contains such kernels as matrix transformation, vertex shading, rasterization, and dithering [11].

In addition to adding new benchmarks, we plan to implement the XBS benchmarks on other conventional and experimental architectures. One standard platform to which XBS will be ported is the PowerPC architecture [3] used in Apple Macintosh computers. The XBS kernels can be optimized for this architecture using the Apple Vector Libraries [17], which provide vector optimized digital signal processing, image processing, and mathematical routines. A non-conventional architecture to which we plan to port XBS is the RAW tiled architecture [9]. RAW exploits instruction level parallelism using an array of communicating tiles and should perform well on the XBS benchmarks which exhibit significant parallelism.

We also plan to implement several scenario benchmarks (see Section 2.1) within the XBS framework. Scenarios model the real-world environments of embedded systems more closely than application benchmarks can alone, and would provide valuable insight into the performance of an architecture. Finally, we plan to distribute XBS over the web so that other researchers and embedded processor designers can benchmark their own systems and compare results.

Bibliography

[1] Krste Asanović. IPM: Interval Performance Monitoring. http://www.cag.lcs.mit.edu/∼krste/ipm/IPM.html.

[2] Elizabeth Basha, Steve Gerding, and Rose Liu. 6.884 Final Project - Hardware Implementation of an 802.11a Transmitter. http://web.mit.edu/∼sgerding/www/papers/80211a_transmitter-6_884-2005.pdf, 2005.

[3] International Business Machines Corporation. Power Architecture. http://www-03.ibm.com/chips/power/.

[4] J. Daemen and V. Rijmen. AES proposal: Rijndael. http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf, 1999.

[5] EDN Embedded Microprocessor Benchmark Consortium. http://www.eembc.org.

[6] J. Babb et al. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, 1999.

[7] J.L. Mitchel et al. *MPEG Video Compression Standard*. Chapman & Hall, 1997.

[8] L. Augustsson et al. Bluespec: Language definition. http://www.bluespec.org/, 2001.

[9] Michael Taylor et al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, March/April 2002.

[10] Steve Gerding and Krste Asanović. The Extreme Benchmark Suite. http://web.mit.edu/∼sgerding/www/papers/xbs_tr.pdf.

[11] Khronos Group. OpenGL ES 2.X and the OpenGL ES Shading Language for programmable hardware. http://www.khronos.org/opengles/2_X/.

[12] Khronos Group. OpenGL ES Overview. http://www.khronos.org/opengles/.

[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *4th IEEE International Workshop on Workload Characteristics*, 2001.

[14] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[15] IEEE. *IEEE standard 1180. Standard Discrete Cosine Transform Accuracy Test*, 1990.

[16] IEEE. *IEEE standard 802.11a supplement. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. High-speed Physical Layer in the 5 GHz Band*, 1999.

[17] Apple Computer Inc. Vector Libraries. http://developer.apple.com/hardware/ve/vector_libraries.html.

[18] CommStack Inc. Why License CommStacks OFDM Modem IP Cores? http://www.commstack.com/WhyLicenseCommStacksOFDMModemIP Coresv2.pdf.

[19] Red Hat Inc. Red Hat Enterprise Linux. http://www.redhat.com/software/rhel/.

[20] Synopsys Inc. Discovery Verification Platform - VCS. http://www.synopsys.com/products/ simulation/.

[21] Synopsys Inc. Synopsys Products: Design Compiler. http://www.synopsys.com/products/logic/design_compiler.html.

[22] Synopsys Inc. Synopsys Products: Design Compiler. http://www.cadence.com/products/digital_ic/rtl_compiler/index.aspx.

[23] Xilinx Inc. LogiCORE 2-D Discrete Cosine Transform(DCT) V2.0 Product Specification. http://www.xilinx.com/ipcenter/catalog/logicore/docs/da_2d_dct.pdf, 2002.

[24] Intel. Intel® Integrated Performance Primitives for Intel® Architecture Reference Manual Volume 1: Signal Processing. ftp://download.intel.com/support/performancetools/libraries/ipp/ippsman.pdf'.

[25] Intel. Intel® Integrated Performance Primitives for Intel® Architecture Reference Manual Volume 2: Image and Video Processing. ftp://download.intel.com/support/performancetools/libraries/ipp/ippiman.pdf'.

[26] Intel. Intel Integerated Performance Primitives. http://www.intel.com/software/products/ipp/.

[27] Intel. Intel Integerated Performance Primitives JPEG Samples. http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/219805.htm.

[28] Intel. Intel Integerated Performance Primitives Product Overview. http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/219892.htm.

[29] Intel. Intel® XEON™ Processor Product Information. http://www.intel.com/products/processor/xeon/index.htm.

[30] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. In *Proceedings of the* 31^{st} *Annual International Symposium on Computer Architecture*, 2004.

[31] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture*, 1997.

[32] M. Li, R. Sasanka, S. Adve, Y. Chen, and E. Debes. The ALPBench Benchmark Suite for Multimedia Applications. Technical Report UIUCDCS-R-2005-2603, University of Illinois at UrbanaChampaign, July 2005.

[33] C. Loeffler, A. Ligtenberg, and G. Moschytz. Practical Fast 1-D DCT Algorithms with 11 Multiplications. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1989.

[34] Victor Lopez Lorenzo. OPENCORES.ORG JPEG Hardware Compressor. http://www.opencores.org/projects.cgi/web/jpeg/overview/, 2005.

[35] OPENCORES.ORG. OPENCORES.ORG: free open source IP cores and chip design. http://www.opencores.org/.

[36] A. J. Klein Osowski and David J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, 2002.

[37] W.B. Pennebaker and J.L. Mitchel. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[38] R. Rabbah, I. Bratt, K. Asanović, and A. Agarwal. Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Report MIT-LCS-TM-646, Massachusetts Institute of Technology, June 2004.

[39] The Independent JPEG Group. http://ijg.org/.

[40] Jim Turley. Embedded Processors by the Numbers. *Embedded Systems Programming*, 12(5), 1999.

[41] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10), 1984.

[42] Alan R. Weiss. Dhrystone Benchmark: History, Analysis, "Scores" and Recommendations. http://www.eembc.org/techlit/Datasheets/dhrystone_wp.pdf.

[43] V. Zivojnovic, J. Martinez, C. Schlger, and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing Applications & Technology* , 1994.