Energy-Optimized Image Communication on Resource-Constrained Sensor Platforms

Dong-U Lee [1], Hyungjin Kim [1,2], Steven Tu [1], Mohammad Rahimi [2]
Deborah Estrin [1,3], and John D. Villasenor [1]
[1] Electrical Engineering Department, University of California, Los Angeles
[2] Center for Embedded Networked Sensing, University of California, Los Angeles
[3] Computer Science Department, University of California, Los Angeles
dongu@icsl.ucla.edu, hjkimnov@ee.ucla.edu, steventu@ucla.edu, mhr@cens.ucla.edu
destrin@cs.ucla.edu, villa@icsl.ucla.edu

ABSTRACT

Energy-efficient image communication is one of the most important goals for a large class of current and future sensor network applications. This paper presents a quantitative comparison between the energy costs associated with 1) direct transmission of uncompressed images and 2) sensor platform-based JPEG compression followed by transmission of the compressed image data. JPEG compression computations are mapped onto various resource-constrained sensor platforms using a design environment that allows computation using the minimum integer and fractional bit-widths needed in view of other approximations inherent in the compression process and choice of image quality parameters. Detailed experimental results examining the tradeoffs in processor resources, processing/transmission time, bandwidth utilization, image quality, and overall energy consumption are presented.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*software libraries*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*computation of transforms*; I.4.2 [**Image Processing and Computer Vision**]: Compression—*approximate methods*

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

While it is intuitive that image communication based on compression followed by transmission is generally more energy efficient than direct transmission of uncompressed images, there has been very little quantitative study of the specific energy tradeoffs characterizing these two approaches in the context of resource-constrained sensor platforms. This is due in part to the challenges involved in implementing compression algorithms such as JPEG on sensor platforms. There is a wealth of literature and experience regarding compression in other environments, but sensor platforms have much more stringent limitations on memory, processing speed, and energy consumption. In addition, unlike devices such as camera-equipped mobile phones which are also limited in some respects, acquisition and dissemination of imagery is the primary function of an important class of sensor platforms, giving the efficiency with which these tasks are performed primary importance.

The JPEG algorithm is utilized for the work presented here. While the newer JPEG2000 standard offers some significant advantages over the "old" JPEG, it is much less widely deployed and also requires full-frame processing which can be challenging in light of memory limitations. As described in [1] and elsewhere, JPEG involves partitioning of an image into blocks of size 8×8, computation of the discrete cosine transform (DCT) for each block, quantization, and then entropy coding. Loss is introduced both in the quantization process and in the DCT which, while in theory lossless, in practice introduces loss due to finite precision. Quality control is effected though adjustment of the quantization step sizes. Choosing small quantization step sizes leads to more accurate coefficient representation and higher image quality, but also results in larger compressed image files.

Energy-aware data compression has previously been examined by Barr and Asanović in [2], and by Sadler and Martonosi in [3]. Both papers investigate the effectiveness of various lossless data compression algorithms such as "LZO" and "bzip2" on constrained embedded platforms. The results demonstrate significant energy benefits when transmitting/receiving compressed data over uncompressed data, primarily due to the higher energy costs associated with communication versus computation. In comparison to lossless data compression, lossy image compression involves a wider range of tradeoffs because the quality of the image is related to the energy consumed during compression and transmission. Wu and Abouzeid [4] have studied the problem of energy-efficient image transmission in a multi-hop wireless sensor network using JPEG2000 compression on a StrongARM SA-1000 processor. For a given image quality requirement and transmission distance, an algorithm for finding the best set of JPEG2000 compression parameters is

.

described. Results indicate that large fractions of the total energy are spent on computation due to the high complexity of JPEG2000. In [5] Taylor and Dey examine effects on energy, quality, and speed using JPEG compression when 1) different quantization tables are used and 2) computation on the high-frequency components is selectively omitted. An adaptive system is described in which quantization and high-frequency handling are varied in order to meet image quality and energy consumption constraints.

In contrast with the previous work in this area, we direct attention to the issue of mapping JPEG onto resource limited processors using a design environment that makes specific use of native word lengths of the target processor. In traditional JPEG implementations, the precision used in computing the DCT is often far greater than is necessary in light of the rounding occurring in the quantization process. Furthermore, the precision is often excessive in light of the approximations inherent in the particular DCT algorithm being used. For example, the DCT is often implemented using fast algorithms based on integer multiplies that only roughly approximate samples of a scaled cosine function. Thus, the present paper provides the following contributions:

- We adopt an energy-aware approach ensuring that the JPEG computations utilize the minimum precision needed to ensure that quantization, not insufficient precision, remains the dominant source of error. To accomplish this we develop a design framework that analytically determines the optimum integer and fractional bit-widths for the signal paths in the compression process and is able to guarantee a specified precision.

- We utilize this framework to automatically generate platform-targeted JPEG C code and perform experiments using the Atmel ATmega128, TI MSP430, TI TMS320C64x, and Analog Devices Blackfin ADSP-BF533 processors to measure the energy savings resulting from the precision optimization process.

- Having obtained optimized JPEG implementations, we then measure the energy consumed by the compression process and subsequent transmission of the compressed data and compare this with the energy required to transmit the images in their uncompressed state. While the general result that compression/transmission is typically more energy efficient than transmission without compression is expected, we are not aware of any previous publications examining the specific quantitative aspects of the associated tradeoffs in terms of processor resources and overall energy consumption in the context of sensor platforms running platform-targeted software.

2. PLATFORM-TARGETED JPEG

2.1 JPEG Compression

JPEG is the most widely used lossy image compression standard. The main steps of JPEG compression are illustrated in Figure 1. If the original image is in RGB format, it is converted into YC_bC_r format where Y is the luminance component and C_b and C_r are chrominance components. Since the human eye is less sensitive to spatial detail in the chrominance components, those images are typically subsampled. The three images are then tiled into sections of 8×8 blocks (in pixels) and processed separately as described below.

Each tile is level shifted to zero mean: e.g. for a pixel depth of 8 bits, 128 would be subtracted from each pixel to bring the range to $[-128, 127]$. The level shifted block is then converted into frequency space by performing a 2D DCT, which is commonly accomplished by performing 1D DCTs on the rows followed by 1D DCTs on the columns. After the DCT, quantization is performed on the block by dividing each sample by a quantization step size defined in accordance with a quantization table, and rounding the result of the division to the nearest integer. Quantization tables are designed such many of the higher frequency components will be rounded to zero. The standard quantization table T specified in the JPEG standard for luminance is given by

$$T = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}. \quad (1)$$

The values in the table corresponding to low frequency components (upper left) are smaller than those corresponding to high frequency components (lower right), with the result that low frequency information will be retained more accurately (because it is quantized using a smaller step size) than high frequency information. To adjust image quality, all values in the table are rescaled in inverse proportion to a quality setting Q_{tab} that ranges from 1 (very poor image quality) to 100 (very good image quality), with a constant of proportionality typically introduced so that $Q_{tab} = 50$ corresponds to no rescaling of the table. The final step in JPEG compression is entropy coding, a lossless compression process involving run-length encoding and Huffman coding. In the present work we focus primarily on the optimization of the DCT and quantization steps, as these are lossy steps in JPEG compression and thus offer the largest opportunity for potential impact in terms of overall energy utilization by addressing precision issues.

2.2 Design Flow Overview

The DCT and quantization steps consist of large numbers additions and multiplications involving real numbers. The most straightforward way of implementing such computations is to use floating-point representations. However, most embedded processors found in sensor networks are highly resource-constrained lack dedicated floating-point hardware. Emulating floating-point operations via integer operations retains the high precision associated with floating point but is extremely slow, especially on 8-bit processors such as the ATmega128. Moreover, floating-point accuracy is rarely required in embedded environments, meaning valuable processor cycles and memory are wasted for computing overly precise results. To avoid this waste, we implement the JPEG computations in fixed-point arithmetic with specific consideration of the precision needed and the native word-length of the processor.

In systems with a short-word-length fixed-point proces-

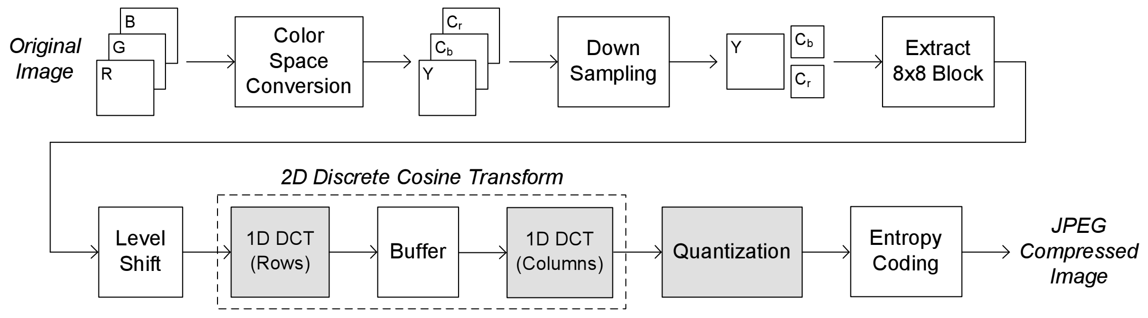

Figure 1: The JPEG compression algorithm.

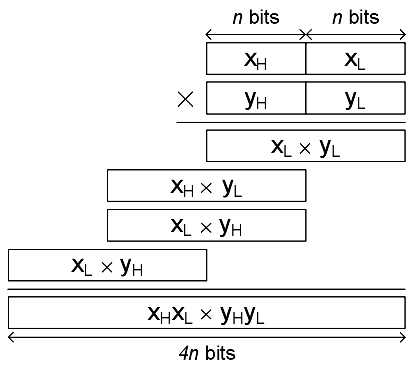

Figure 2: Performing $2n$-bit by $2n$-bit multiplication on an n-bit processor.

sor, multi-word arithmetic, which uses multiple words to execute operations larger than the natural processor word-length can be exploited [6], though due attention must be given to avoid overuse of and/or unnecessarily wide multi-word methods. Figure 2 shows an example of how $2n$-bit $\times$ $2n$-bit (i.e. 2-word $\times$ 2-word) multiplication can be accomplished on an n-bit processor. Assuming that the processor supports an "add with carry" instruction (which most processors do), four multiplications and six additions are required in total. More generally, an L-word by M-word addition requires $\max(L, M)$ additions, while an L-word by M-word multiplication requires $L \times M$ multiplications and $2 \times (L \times M - 1)$ additions. In order to minimize execution time, it is desirable to minimize the number of words used for each multi-word arithmetic operation, while simultaneously meeting the user-specified output error criterion.

Figure 3 depicts the design flow for determining the number of required bits (or words) involved for each fixed-point operation. The "Range Analysis" step analyzes the dynamic range of each signal and computes the minimal integer bit-widths that avoid overflow. We define a "signal" to be an operand of an addition/multiplication operation or a result of an operation. Next, "Precision Analysis" is performed which computes the fractional bits required to each signal. Simulated annealing is employed for this step, which finds the fractional bits in conjunction with constraints such as the output error requirement and costs associated with arithmetic operations of the target platform. Once the signal bit-widths are found, an appropriate C code fragment of the algorithm is generated.

In [7], a similar design flow was examined for generating

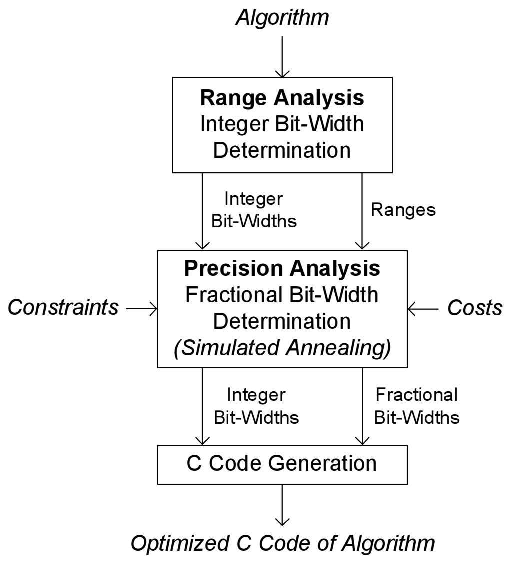

Figure 3: Bit-width optimization flow.

optimized hardware designs on field-programmable gate arrays. approach described here however, differs considerably from that in [7] in that 1) in the present work we target software programs on embedded processors which impose numerous constraints (limited processor word-length, fixed data types, instruction latencies, etc.), and 2) while [7] aims to minimize circuit area, the aim in the present paper is to minimize processor cycles and therefore energy.

2.3 Range Analysis

The aim of range analysis is to analyze the dynamic range of each signal and to compute the required number of integer bits that avoid overflow. We assume two's complement representation throughout. The bit-width of a signal x is denoted by B_x, while its integer bit-width (IB) and fractional bit-width (FB) are denoted by IB_x and FB_x respectively, i.e. $B_x = IB_x + FB_x$. In two's complement fixed-point, the IB of a signal x is given by

$$IB_x = \lceil \log_2(X) \rceil + 1 \qquad (2)$$

where $X = \max(|x_{\min}|, |x_{\max}|)$.

The range analysis method involves examining the local minima, local maxima, and the minimum and maximum input values at each signal. The local minima and local maxima are found by computing the roots of the derivative at

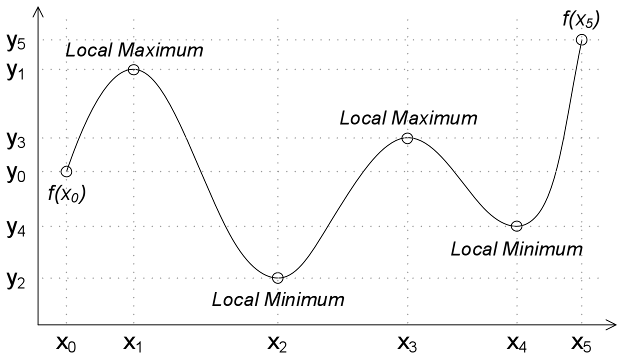

Figure 4: Local minima and maxima of a signal. The range of this signal is $[y_2, y_5]$.

the signal. Consider the signal shown in Figure 4 where the input interval is over $[x_0, x_5]$. The local minima and maxima can be found by computing the roots of the derivative. In Figure 4 the local minima, local maxima $f(x_0) = y_0$, and $f(x_5) = y_5$ are the potential candidates for the minimum and maximum values of the signal. The minimum value is given by $\min(y_0, y_1, y_2, y_3, y_4, y_5)$, while the maximum value is given by $\max(y_0, y_1, y_2, y_3, y_4, y_5)$. For this particular example, the output range would be $[y_2, y_5]$. This analysis methodology works for designs with differentiable signals, which is the case for the algorithms considered in this paper. For more general designs where signals cannot be differentiated, one could apply other methods such as affine arithmetic [8].

2.4 Precision Analysis

The goal of precision analysis is to determine the minimum fractional bit-widths (FBs) of all signals that still enable satisfying the user-specified accuracy constraint at the output. Quantization of signals is generally performed using truncation or round-to-nearest. Although round-to-nearest results in smaller errors, it requires an extra addition for adding the rounding constant.

2.4.1 Error Models

When performing truncation or round-to-nearest to a signal z, the error $\hat{\epsilon}_z$ due to quantization is given by:

$$\text{Truncation: } \hat{\epsilon}_z = \max(0, 2^{-FB_z} - 2^{-FB_{z'}}) \tag{3}$$

$$\text{Round-to-nearest: } \hat{\epsilon}_z = \begin{cases} 0, & \text{if } FB_z \geq FB_{z'} \\ 2^{-FB_z - 1}, & \text{otherwise} \end{cases} \tag{4}$$

where $FB_{z'}$ is the full precision of z before quantization. With the addition $z = x + y$ and the multiplication $z = x \times y$, $FB_{z'}$ is defined as follows

$$z = x + y : FB_{z'} = \max(FB_x, FB_y) \tag{5}$$

$$z = x \times y : FB_{z'} = FB_x + FB_y. \tag{6}$$

For the addition operation $z = x + y$, the error ϵ_z at the output z is given by

$$\begin{aligned} z &= x + y = x + y + \epsilon_x + \epsilon_y + \hat{\epsilon}_z \\ \Rightarrow \epsilon_z &= \epsilon_x + \epsilon_y + \hat{\epsilon}_z \end{aligned} \tag{7}$$

where ϵ_x and ϵ_y are errors associated with signals x and y.

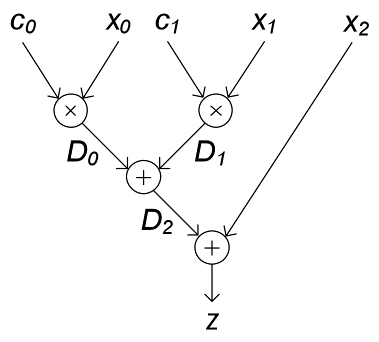

Figure 5: Example for precision analysis: $z = c_0 \times x_0 + c_1 \times x_1 + x_2$.

Table 1: Signal ranges and required integer bit-widths (IBs) for the example in Figure 5.

Signal	Range	IB
x_0	$[-0.5, 0.5)$	0
x_1	$[-2, 3)$	3
x_2	$[-4, 5)$	4
c_0	2.14532	3
c_1	0.07937	-2
D_0	$[-1.07266, 1.07266)$	2
D_1	$[-0.15874, 0.23811)$	-1
D_2	$[-1.23140, 1.31077)$	2
z	$[-5.23140, 6.31077)$	4

Similarly, for multiplication,

$$\begin{aligned} z &= xy \\ &= xy + x\epsilon_y + y\epsilon_x + \epsilon_x\epsilon_y + \hat{\epsilon}_z \\ \Rightarrow \epsilon_z &= x\epsilon_y + y\epsilon_x + \epsilon_x\epsilon_y + \hat{\epsilon}_z. \end{aligned} \tag{8}$$

It is clear that ϵ_z is at its maximum when x and y are at their maximum absolute values. The expressions in Eqn. (7) and Eqn. (8) form the basis for the performing precision analysis on complex computations such as the DCT.

2.4.2 Precision Analysis Example

We describe the precision analysis procedure with the following example:

$$z = c_0 \times x_0 + c_1 \times x_1 + x_2 \tag{9}$$

which occurs in the computation of the DCT used in this work. x_0, x_1, and x_2 are inputs, c_0 and c_1 are constants, and z is the output we would like to evaluate as shown in Figure 5. Instead of evaluating the expression in Eqn. (9) directly, the output of each addition and multiplication operator is stored in a temporary signal ($D_{0..2}$) to allow quantization to take place. For example, assume the ranges of the inputs are $x_0 = [-0.5, 0.5)$, $x_1 = [-2, 3)$, and $x_2 = [-4, 5)$ and that the two constants are $c_0 = 2.14532$ and $c_1 = 0.07937$. The range analysis approach described in Section 2.3 then gives the ranges and integer bit-widths (IBs) shown in Table 1. A zero or negative integer bit-width means that the magnitude of the signal will always be less than 1, and the specific negative value identifies the number of positions to the right of the binary point that will always be zero.

In order to compute the required fractional bit-widths (FBs) using the expressions in Eqn. (7) and Eqn. (8), it is first necessary to derive the error expressions to each signal. We assume that round-to-nearest is performed for the constants and truncation is performed for all other signals. The following error expressions can be derived for each signal in this example:

$$\epsilon_{D_0} = c_0\epsilon_{x_0} + \max(|x_0|)\epsilon_{c_0} + \epsilon_{c_0}\epsilon_{x_0} + \hat{\epsilon}_{D_0} \quad (10)$$
$$\epsilon_{D_1} = c_1\epsilon_{x_1} + \max(|x_1|)\epsilon_{c_1} + \epsilon_{c_1}\epsilon_{x_1} + \hat{\epsilon}_{D_1} \quad (11)$$
$$\epsilon_{D_2} = \epsilon_{D_0} + \epsilon_{D_1} + \hat{\epsilon}_{D_2} \quad (12)$$
$$\epsilon_z = \epsilon_{D_2} + \epsilon_{x_2} + \hat{\epsilon}_z. \quad (13)$$

For a desired worst-case error requirement ϵ_{req}, one must satisfy $\epsilon_z \leq \epsilon_{req}$. This is an optimization problem, where the goal is to find the minimal FBs for a given error requirement ϵ_z and output fractional bit-width FB_z. Though such optimization problems can be addressed in a number of different ways, we address it using simulated annealing, which is known to provide near-optimal results at reasonable computational cost [7]. Inequalities such $\epsilon_z \leq \epsilon_{req}$ are supplied as the constraint function and the latency costs associated with multi-word arithmetic are supplied as the cost function for the annealing process. In the case of the ATmega128 processor, an 8-bit addition requires a single cycle and an 8-bit multiplication consumes two cycles. In addition, the following extra constraints are added.

1. Bit-widths must conform to the C language integer data types, which are limited to 8, 16, 32, and 64 bits.

2. Bit-widths of the two operands for an addition or multiplication must be the same. This is because the C language performs additions and multiplications using operands with the same data types. If the types are different, compilers will cast the smaller operand to the bigger operand during compilation. Forcing the operand widths to be the same simply anticipates a step the compiler will take anyway, and allows the optimization algorithm to use precision opportunities that will be introduced in compilation.

3. The two operands of an addition must share the same FB since binary points must be aligned for additions. Due to constraint 2, this also means the addition operands share the same IB.

Table 2 shows the set of bit-widths obtained after the optimization process for error requirements of $\epsilon_{req} = 2^{-5}$ and $\epsilon_{req} = 2^{-10}$ with a fixed output bit-width of $B_z = 16$. The shifts shown in the last column are required to perform quantization and binary point alignments which will be discussed in Section 2.5. Compared to Table 1, the IBs of D_1 and D_2 have increased from -1 and 2 to 2 and 4 respectively, which is necessary to satisfy constraint 3. Requirements imposed by constraint 1 and constraint 2 have also been met. While 16 bits are required for all signals in the case of $\epsilon_{req} = 2^{-10}$, some signals in the case of $\epsilon_{req} = 2^{-5}$ can be represented by 8 bits. The optimization tool preferentially targeted reduction of the multiplications rather than of additions since multiplications cost more processor cycles. This will lead to a reduction in overall latency due to the reduced number of words involved in during multi-word arithmetic.

Table 2: Signal bit-widths for the example in Figure 5 after optimization for error requirements of $\epsilon_{req} = 2^{-5}$ and $\epsilon_{req} = 2^{-10}$ with a fixed output bit-width of $B_z = 16$.

ϵ_{req}	Signal	B	IB	FB	$\ll$
2^{-5}	x_0	8	0	8	0
	x_1	8	3	5	0
	x_2	16	4	12	0
	c_0	8	3	5	0
	c_1	8	−2	10	0
	D_0	16	2	14	−1
	D_1	16	2	14	1
	D_2	16	4	12	2
	z	16	4	12	0
2^{-10}	x_0	16	0	16	0
	x_1	16	3	13	0
	x_2	16	4	12	0
	c_0	16	3	13	0
	c_1	16	−2	18	0
	D_0	16	2	14	15
	D_1	16	2	14	17
	D_2	16	4	12	2
	z	16	4	12	0

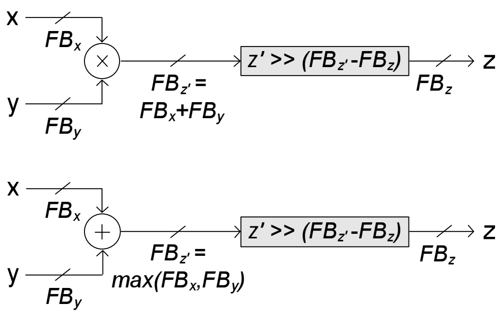

Figure 6: Computing the number of shifts required for multiplication and addition.

2.5 Fixed-Point to Integer Mapping

Internally, fixed-point processors perform integer operations. Although fixed-point libraries for the C language are available, they generally do not provide support for negative IBs. This is problematic given the potential resource savings that can be achieved by exploiting in the leading zeros which, as Table 1 demonstrates, can occur commonly. This can be addressed by performing integer arithmetic with an implicit binary point.

Shifts are conducted after each operation to perform quantization and binary point alignment. Figure 6 illustrates how the shifts are computed for each operation. For instance, for the expression $z = x \times y$, the FBs of x, y and z are known a priori from the precision analysis phase. The intermediate result z' will contain the multiplication full precision of $FB_x + FB_y$ fractional bits. To adjust the number of fractional bits according to what is needed for z, z' is simply shifted to the right by $FB_{z'} - FB_z$ bits.

Consider the $\epsilon_{req} = 2^{-5}$ case in Table 2 when $x_0 = 0.44140625$, $x_1 = -1.84375$, and $x_2 = 2.122314453125$ which have FBs of 8, 5, and 12 bits, respectively. We use the notation $\bar{x}$ to represent a fixed-point quantity x in integers with an implicit binary point. The corresponding integer representations of the three inputs are $\bar{x}_0 = x_0 \ll FB_{x_0} = 113$, $\bar{x}_1 = -59$, and $\bar{x}_2 = 8693$. Likewise, $\bar{c}_0 = \lfloor(2.14532 \ll 5) + 0.5\rfloor = 69$ and $\bar{c}_1 = 81$. The steps below illustrate how $\bar{z}$ is computed.

$$\bar{D}_0 = (\bar{x}_0 \times \bar{c}_0) \ll 1 = 15594 \quad (14)$$

$$\bar{D}_1 = (\bar{x}_1 \times \bar{c}_1) \gg 1 = -2389 \quad (15)$$

$$\bar{D}_2 = (\bar{D}_0 \times \bar{D}_1) \gg 2 = 3301 \quad (16)$$

$$\bar{z} = (\bar{D}_2 \times \bar{x}_2) = 11994 \quad (17)$$

To compute the fixed-point quantity z we simply shift $\bar{z}$ by FB_z bits to the right and keep the fractions, i.e. $z = 11994 \gg 12 = 2.92822265625$.

2.6 Optimized DCT and Quantization

For an input matrix $x(i,j)$ and an output matrix $z(m,n)$, the $N \times N$ 2-D DCT is defined as

$$z(m,n) = \frac{2}{N}K(m)K(n)\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} x(i,j) \times \cos\frac{\pi m(2i+1)}{2N}\cos\frac{\pi n(2j+1)}{2N} \quad (18)$$

where $K(0) = 1/\sqrt{2}$, and $K(m) = 1$ and $K(n) = 1$ for $m \neq 0$ and $n \neq 0$. For the case of an 8×8 DCT, a direct implementation of Eqn. (18) would require 4096 multiplications (though some of them are trivial). However, since the transform kernel in the summation is separable, the 2D DCT can be computed by performing 1D transforms of the rows and then columns, or vice versa. Further reductions in complexity can be realized by using fast algorithms and, in the case of JPEG, by exploiting the specific multiplications that occur with the DCT length is 8. Among the many fast $N = 8$ DCT algorithms that have been proposed, we use the LLM algorithm described by Loeffler *et al.* [9], which requires 12 multiplications and 32 additions for the 8-point 1D DCT. For the complete 8×8 2D DCT, this must be performed once for each of the 16 rows and columns, leading to a total of 192 multiplications and 512 additions.

To obtain optimized DCT and quantization, the bit-width optimization methods described earlier were applied to the operations in the LLM algorithm to determine appropriate bit-widths given the precision requirement and to automatically generate the C code appropriate to the target platform. In addition to the three constraints described in Section 2.4, we added an extra constraint where shifts involving 32-bit integers were constrained to multiples of bytes. This was introduced since arbitrary bit shifts on 32-bit integers using processors without hardware shifters can be extremely slow. Constraining such shifts to be multiples of bytes allows the processor to perform shifts simply by moving memory locations. To avoid division in the quantization process, the DCT outputs were multiplied by the reciprocal of the quantization table values. Due to the nature of the LLM algorithm, the resulting DCT outputs are scaled by a factor of 8. This factor is compensated in the quantization tables.

Since the bit-widths of the standard C data types such as "int" or "long int" can differ depending on the target platform, we use the "inttypes.h" standard C header, which enforces target-independent fixed size integer type definitions, and use the data types "int8_t", "int16_t", "int32_t" and "int64_t", which correspond respectively to two's complement 8, 16, 32, and 64-bit integers.

3. EXPERIMENTAL RESULTS

3.1 Compression

Image compression experiments were performed using grayscale images, as this is sufficient to allow exploring the compression speed, image quality, and energy consumption tradeoffs of interest in the present work. As is common in JPEG implementations, the JPEG standard table shown in Eqn. (1) was mapped to $Q_{tab} = 50$, and quantization tables for other quality settings were generated by linearly scaling the JPEG standard table [10].

Table 3 provides the latency, code size, execution time, energy consumption, and peak signal to noise ratio (PSNR) for three test images at $Q_{tab} = 50$ for the DCT and quantization steps in JPEG. The Bird, Camera, and Goldhill images from the "GreySet1" image archive hosted at the Waterloo BragZone [11] are used. The code size shown includes both instructions and data. The ATmega128 at 8 MHz with an active power consumption of 22 mW [12] is used as the target platform. Compilation is performed via WinAVR 20060421 with "-O3" optimization setting. As a basis for comparison, the library released by the Independent JPEG Group (IJG) [10] is used. The first row of the table gives the requirements for a floating point implementation in which C single-precision computations are mapped to the processor using standard compilation approaches. As noted earlier, such an approach delivers far more precision than is needed and in this case uses over an order of magnitude more time than all of the optimized versions. IJG has released two speed-optimized integer versions of JPEG code - a "slow" version with a DCT that favors accuracy over speed, and a "fast" version that makes the opposite tradeoff. The PSNR results in the table show that the "slow" version incurs no PSNR penalty over a full floating point implementation, while the "fast" version causes a 5 to 6 dB loss, which is very significant in dB terms and visually. When mapped into the ATmega128, the IJG "fast" version requires about half as many clock cycles as the "slow" version.

One of the most powerful advantages of the custom precision approach is that it offers the implementer a range of operating point choices and provides an energy optimized implementation for the chosen operating point. Rows 4-6 of Table 3 illustrate three such operating points. The improvements in the mappings in rows 4-6 of the table arise because the IJG implementations use fixed integer data types, whereas we customize the data types for each operation via the approach described in Section 2. Row 4 gives the results for a "slow" algorithm in which the constraint is that there must be no PSNR loss with respect to the floating point implementation. This is achieved using 38,364 cycles, which is approximately 25% lower than the corresponding IJG algorithm in row 2 of the table. Row 5 of the table illustrates an intermediate solution in which a PSNR loss of several tenths of a dB is allowed, and the cycle count is then reduced to approximately 23,000. Row 6 of the table gives the result of a "fast" optimization. Notably, this this more than twice as fast as the IJG "fast" algorithm in row 3, while still producing PSNR values that are only about 1 dB reduced from the full floating point PSNR, and 4 to 5 dB

Table 3: Comparisons between IJG's 8×8 DCT and quantization implementation and the proposed on the ATmega128. The PSNR is computed using 128×128 Bird, Camera, and Goldhill images at $Q_{tab} = 50$.

	Method	Type	DCT [Cycles]	Quantization [Cycles]	Total [Cycles]	Code Size [Bytes]	Execution Time [ms]	Energy [μJ]	PSNR [dB]		
									Bird	Camera	Goldhill
1	IJG	Float	580,106	244,840	824,946	5,964	103.12	2268.60	31.8	28.3	27.0
2		Slow	31,378	17,039	48,417	3,355	6.05	133.15	31.8	28.3	27.0
3		Fast	8,131	17,831	25,962	1,670	3.25	71.40	25.8	23.7	23.8
4	Proposed	Slow	21,172	17,192	38,364	3,524	4.80	105.50	31.8	28.3	27.0
5		Medium	20,718	2,810	23,528	3,318	2.94	64.73	31.3	28.0	26.6
6		Fast	8,768	2,385	11,153	2,662	1.39	30.67	30.6	27.5	26.1

Table 4: Variation in entropy coding latency with Q_{tab} on the ATmega128. Numbers have been averaged over the three images.

Q_{tab}	10	30	50	70	90
Latency [Cycles]	12124	19265	21504	28431	40893

above the PSNR results of the "fast" IJG algorithm in row 3 of the table. Thus, in this example the custom precision approach has enabled algorithm mapping that is both faster and gives more accurate DCT and quantization results (thus the higher PSNR). It is important to emphasize that these results are not intended to suggest that the IJG algorithm is "bad" in any sense; in fact it is widely recognized to be quite good. Rather, the key point conveyed in the table is that when platform-specific architectural attributes are considered in combination with precision requirements consistent with the application and precision of the underlying data, substantial savings can be realized.

Table 4 shows the variation in entropy coding latency as a function of different quality factors Q_{tab}. The latencies have been averaged over the three images. The general trend that higher quality settings require entropy coding of more bits is of course expected because the associated image files are larger. These latencies apply to both the IJG JPEG implementation as well as the implementation presented here, as it was the DCT and quantization steps that were subject to optimization, not the entropy coding. The value of this table is that it provides information that, in combination with Table 3, allows comparison of the overall latency requirements for the entire end-to-end JPEG algorithm for the IJG and DCT-optimized approaches. One example of such a comparison is given in Table 5, which provides a comparison of the DCT computation using the IJG "slow" algorithm and the proposed "fast" approach for the case where $Q_{tab} = 50$. As was discussed in association with rows 2 and 6 of Table 3, the proposed "fast" algorithm leads to PSNR values that are less than 1 dB lower than the IJG "slow" algorithm, which in most applications would not be visually significant. As Table 5 shows, the proposed "fast" algorithm requires about 1/4 as many cycles for DCT and quantization and the same number of cycles for entropy coding (because no entropy coding optimization was performed). The total number of cycles is approximately halved, with the energy consumption reduced accordingly.

Figure 7 provides a comparison of the image quality associated with different PSNR levels and algorithm mappings for the Camera image. Images shown are (a) the original, (b) the result of the IJG "slow" mapping (row 2 of Table 3) and the proposed "slow" mapping (row 4 of Table 3) at $Q_{tab} = 50$ (both "slow" methods give exactly the same image with a PSNR of 28.3 dB, but the proposed "slow" mapping requires approximately 10,000 fewer cycles than the IJG approach), (c) the result of the proposed "fast" mapping also at $Q_{tab} = 50$, giving a PSNR of 27.5 dB, and (d) an example of a the quality associated with a poor image quality setting of $Q_{tab} = 10$. In image (d) the proposed "fast" mapping was used, giving a PSNR of 22.8 dB, though if the IJG "slow" mapping were used with this Q_{tab} setting the PSNR would be 22.9 dB, which is almost identical.

Table 6 provides comparisons among various platforms for performing DCT and quantization on a single 8×8 block via the proposed "fast" method. The voltage, clock speed, and active power specifications were obtained from [12] for the Atmel ATmega128 and the TI MSP430F1611, and from [13] for the Analog Devices Blackfin ADSP-BF533 and the TI TMS320C6414T. Compilations are performed with the fastest optimization settings using WinAVR 20060421, IAR Embedded Workbench 3, Analog Devices Visual DSP++ 4.5, and TI Code Composer Studio 3.1. The cycle counts given in the latency column are the result of the optimization methods described in present paper, and differ among the various processors due to architectural differences such as the native world-length. The most interesting information in the table is in the final two columns, which give execution time and energy respectively. The lowest power processors (the ATmega128 and MSP430F1611) actually have the highest energy results, in part because they operate a lower speeds and require a longer time to complete the processing. In this respect, the ratio of power to speed (which is given in the fourth column of the table for all the processors) is at least as important as the power consumption alone. It is also important to note that there is a wide range of operating voltages, from 0.8 V in the case of the ADSP-BF533 to 2.7 V for the ATmega128. Thus, a conclusion that the ATmega128 is 30 times less energy efficient than the ADSP-BF533 must be tempered by the fact that the voltage ratio between these two processors is more than a factor of 3.

3.2 Compression and Transmission

In many applications what is of most interest is not the energy cost to compress alone, but also the energy cost to deliver an image to a different location. Transmission energy

Table 5: Comparisons between IJG's slow JPEG and proposed fast JPEG compression for an 8×8 block on the ATmega128 at $Q_{tab} = 50$. Entropy coding cycles have been averaged over the three images.

Method	DCT & Quantization [Cycles]	Entropy Coding [Cycles]	Total [Cycles]	Code Size [Bytes]	Execution Time [ms]	Energy [μJ]
IJG Slow	48,417	21,504	69,921	19,197	8.74	192.28
Proposed Fast	11,153	21,504	32,657	18,504	4.08	89.81

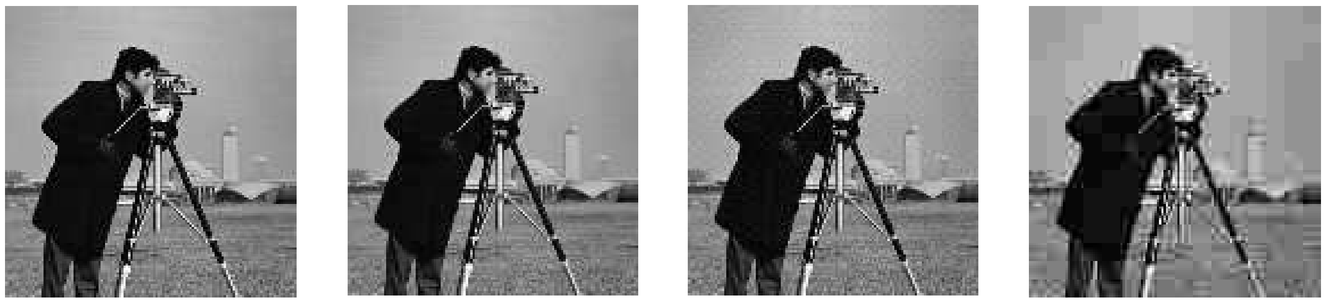
(a) Original (b) Slow: $Q_{tab} = 50$ (c) Proposed Fast: $Q_{tab} = 50$ (d) Proposed Fast: $Q_{tab} = 10$

Figure 7: Comparisons between the original 128×128 Camera and its JPEG compressed versions. The JPEG images correspond to (b) the IJG "slow" algorithm (row 2 of Table 3) and the proposed "slow" algorithm (row 4 of Table 3) - both "slow" methods give exactly the same image, but the proposed method is faster by about 10,000 cycles than the IJG algorithm, (c) the proposed "fast" mapping at $Q_{tab} = 50$, and (d) the proposed "fast" at $Q_{tab} = 10$. The PSNRs of the three JPEG images are 28.3 dB, 27.5 dB, and 22.8 dB. The original image size is 131 kb, images (b) and (c) have compressed size 21 kb, and the last JPEG size is 9 kb.

Table 6: Comparisons between various processors when performing fast DCT and quantization on an 8×8 block.

Processor	Voltage [V]	Clock Speed [MHz]	Active Power [mW]	Power / Speed [mW/MHz]	Latency [Cycles]			Execution Time [μs]	Energy [μJ]
					DCT	Quantization	Total		
ATmega128	2.7	8	22	2.75	8,768	2,385	11,153	1,394.16	30.67
MSP430F1611	1.8	8	3	0.37	5,113	2,266	7,379	922.38	2.78
ADSP-BF533	0.8	100	24	0.24	2,855	1,313	4,168	41.68	1.00
ADSP-BF533	1.4	756	644	0.85	2,855	1,313	4,168	5.51	3.55
TMS320C6414T	1.1	400	673	1.68	1,382	505	1,887	4.72	3.17

Table 7: Specifications of Mica2, MicaZ, and Telos motes [12].

Mote	Mica2	MicaZ	Telos
Processor	ATmega128	ATmega128	MSP430F1611
Radio	CC1000	CC2420	CC2420
Data Rate [kbps]	38.4	250	250
Processor [mW]	22	22	3
Transmit [mW]	69	57	35
Receive [mW]	41	63	39

is obviously highly platform-specific, so to examine this we consider three commonly used motes with the specifications as provided in Table 7 [12] (the radio transmit power for the data in the table is 0 dBm).

Figure 8 is a bar graph giving the total time for compression and transmission using an average of the results for the same three test images considered in Table 3. The upper and lower parts of each bar indicate the time used by transmission and compression respectively. Results are given for different image resolutions (square images of linear sizes 64, 128, and 256), and within each resolution for low ($Q_{tab} = 10$), medium ($Q_{tab} = 50$), and high ($Q_{tab} = 90$) image quality settings. Results for transmitting images with no compression are also provided. Not surprisingly, whether or not it is faster to compress and then transmit or to transmit without compression is at least partially dependent on the specific processor/radio combination. In the case of the Mica2 mote in combination with the ATmega128 processor and CC1000 radio, compression is always advantageous; with the Telos mote in combination with the MSP430F1611 processor and CC2420 radio the opposite is true. This is largely due to the significantly higher data rate of the Telos versus the Mica2. More specifically, the compression process is unable to reduce the size of the file by an amount equal to the number of bits that can be transmitted by the Telos

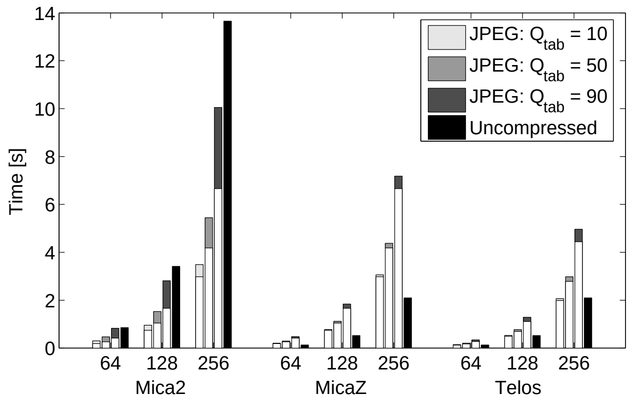

Figure 8: Average time for compressing and transmitting an image. Results are given for different image resolutions (square images of linear sizes 64, 128, and 256), and within each resolutions for low ($Q_{tab} = 10$), medium ($Q_{tab} = 50$), and high ($Q_{tab} = 90$) image quality settings. Results for transmitting images with no compression are also provided. The upper and lower parts of each bar indicate the portions used by transmission and compression respectively.

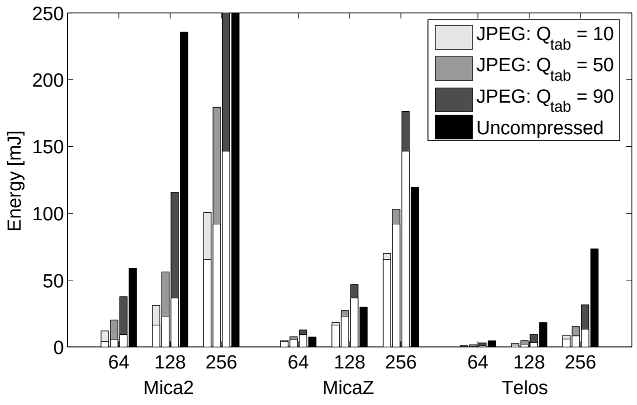

Figure 9: Average energy for compressing and transmitting an image. The upper and lower parts of each bar indicate the portions used by transmission and compression respectively. The two cropped Mica2 bars are 380.2 mJ and 942.1 mJ.

during the time otherwise consumed by compression. In all cases, Figure 8 highlights large reduction transmission time when JPEG is employed. This of course leads to reduction in bandwidth usage and thus less contention in the channel which is a common problem in wireless sensor networks.

When energy is considered instead of time, a very different set of results is produced as shown in Figure 9. The image resolutions and quality settings are the same as those in Figure 8. The energy values in the figure include the cost to compress and to transmit; the potential costs to receive and decompress are not considered. While reception/decompression are certainly part of the overall process of communicating an image, the value and significance of energy costs at a receiver are highly system dependent, and in many environments images would be received at highly capable nodes in which energy was less problematic. For systems such as multi-hop environments in which other energy-constrained nodes would be relaying the images, information such as that in Table 7 can be used to obtain an end-to-end cost. Since compression costs are only associated with the initial node, the time and energy savings of using JPEG would increase with the number of hops.

The energy results in Figure 9 are notable in comparison with the time results in Figure 8 in several respects. First, while for the Telos avoiding compression at all was the most time-efficient approach, for energy efficiency compression offers significant benefits. As would be expected, these benefits are larger when more compression is applied. Secondly, for the Mica2, compression was advantageous from a time standpoint, and Figure 9 shows that it is advantageous from an energy standpoint as well. Finally, results for the MicaZ are more nuanced. For the low and medium image quality levels, compression offers an energy advantage. For the highest quality level, compression is not justified because transmitting the uncompressed image consumes less energy. As the figure shows, this is because at that quality level the compression process alone (even without considering transmission of the compressed image) consumes more energy than transmission without compression. The results in Figure 9 correspond to a radio power of 0 dBm. When large transmission distances are desired, the radio power will have to be increased accordingly, which of course leads to an increase in the overall transmission cost per bit. Hence with increasing radio power, the relative cost of transmitting uncompressed images as opposed to JPEG-compressed images would increase.

4. CONCLUSIONS

We have explored the energy tradeoffs involved in JPEG compression on energy-constrained platforms followed by wireless transmission of the compressed images. A design approach based on precision-optimized custom arithmetic was used to obtain energy-minimal, platform-specific implementations of the DCT and quantization steps in the JPEG algorithm. Comparisons with traditional JPEG libraries show speed and energy improvements ranging from factors of 2 to 5 depending on which portion of the algorithm was considered. Comparisons across different platforms show that the JPEG energy consumption is actually higher on “low” power platforms due to the longer times needed for these platforms to perform the computation tasks to the desired precision. Time and energy requirements for the combination of compression/transmission were also investigated for several different processor/radio combinations. The most energy-efficient and time-efficient approaches among the options of 1) transmission of uncompressed images or 2) compression followed by transmission were identified for a variety of processor/radio combinations.

5. ACKNOWLEDGMENTS

The authors thank David Choi for his assistance. The support of the National Science Foundation (Grant number CCR-0120778 and CCF-0541453) and the Office of Naval Research (Contract number N00014-06-1-0253) is gratefully acknowledged.

6. REFERENCES

[1] G. Wallace, “The JPEG still picture compression standard,” *Communications of the ACM*, vol. 34, no. 4, pp. 30–44, April 1991.

[2] K. Barr and K. Asanović, “Energy-aware lossless data compression,” *ACM Trans. Computer Systems*, vol. 24, no. 4, pp. 250–291, August 2006.

[3] C. Sadler and M. Martonosi, “Data compression algorithms for energy-constrained devices in delay tolerant networks,” in *Proc. ACM Conf. on Embedded Networked Sensor Systems*, 2006.

[4] H. Wu and A. Abouzeid, “Power aware image transmission in energy constrained wireless networks,” in *Proc. IEEE Int’l Symp. on Computers and Communications*, vol. 2, 2004, pp. 202–207.

[5] C. Taylor and S. Dey, “Adaptive image compression for wireless multimedia communication,” in *Proc. IEEE Int’l Conf. on Communications*, vol. 6, 2001, pp. 1925–1929.

[6] T. Lukasiak, “Extended-precision fixed-point arithmetic on the Blackfin processor platform,” *Analog Devices Engineer To Engineer Note (EE-186)*, 2003.

[7] D. Lee, A. Abdul Gaffar, R. Cheung, O. Mencer, W. Luk, and G. Constantinides, “Accuracy-guaranteed bit-width optimization,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, October 2006.

[8] L. de Figueiredo and J. Stolfi, “Self-validated numerical methods and applications,” in *Brazilian Mathematics Colloquium monograph.* IMPA, Brazil, 1997.

[9] C. Loeffler, A. Ligtenberg, and G. Moschytz, “Practical fast 1-D DCT algorithms with 11 multiplications,” in *Proc. IEEE Int’l Conf. on Acoustics, Speech, and Signal Processing*, vol. 2, 1989, pp. 988–991.

[10] T. Lane, P.Gladstone, L. Ortiz, J. Boucher, L. Crocker, J. Minguillon, G. Phillips, D. Rossi, and G. Weijers, “The independent JPEG group’s JPEG software release 6b,” 1998, http://www.ijg.org.

[11] J. Kominek, “Waterloo BragZone,” *University of Waterloo*, 1995, http://links.uwaterloo.ca/bragzone.base.html.

[12] J. Polastre, R. Szewczyk, and D. Culler, “Telos: enabling ultra-low power wireless research,” in *Proc. ACM/IEEE Int’l Symp. on Information Processing in Sensor Networks*, 2005, pp. 364–369.

[13] BDTI, “Processor overviews,” 2006, http://www.bdti.com/procsum/index.htm.