

# Understanding Sources of Inefficiency in General-Purpose Chips

By Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz

## Abstract

Scaling the performance of a power limited processor requires decreasing the energy expended per instruction executed, since  $\text{energy/op} \times \text{op/second}$  is power. To better understand what improvement in processor efficiency is possible, and what must be done to capture it, we quantify the sources of the performance and energy overheads of a 720p HD H.264 encoder running on a general-purpose four-processor CMP system. The initial overheads are large: the CMP was 500× less energy efficient than an Application Specific Integrated Circuit (ASIC) doing the same job. We explore methods to eliminate these overheads by transforming the CPU into a specialized system for H.264 encoding. Broadly applicable optimizations like single instruction, multiple data (SIMD) units improve CMP performance by 14× and energy by 10×, which is still 50× worse than an ASIC. The problem is that the basic operation costs in H.264 are so small that even with a SIMD unit doing over 10 ops per cycle, 90% of the energy is still overhead. Achieving ASIC-like performance and efficiency requires algorithm-specific optimizations. For each subalgorithm of H.264, we create a large, specialized functional/storage unit capable of executing hundreds of operations per instruction. This improves energy efficiency by 160× (instead of 10×), and the final customized CMP reaches the same performance and within 3× of an ASIC solution's energy in comparable area.

## 1. INTRODUCTION

Most computing systems today are power limited, whether it is the 1 W limit of a cell phone system on a chip (SoC), or the 100 W limit of a processor in a server. Since power is  $\text{ops/second} \times \text{energy/op}$ , we need to decrease the energy cost of each op if we want to continue to scale performance at constant power. Traditionally, chip designers were able to make increasingly complex designs both by increasing the system power, and by leveraging the energy gains from technology scaling. Historically each factor of 2 in scaling made each gate evaluation take 8× less energy.<sup>7</sup> However, technology scaling no longer provides the energy savings it once did,<sup>9</sup> so designers must turn to other techniques to scale energy cost. Most designs use processor-based solutions because of their flexibility and low design costs, however, these are usually not the most energy-efficient solutions. A shift to multi-core systems has helped improve the efficiency of processor systems but that approach is also going to hit a limit pretty soon.<sup>8</sup>

On the other hand, using hardware that has been customized for a specific application (an Application Specific Integrated Circuit or ASIC) can be three orders of magnitude better than a processor in both energy/op and ops/area.<sup>6</sup> This paper compares ASIC solutions to processor-based solutions, to try to understand the sources of inefficiency in general-purpose processors. We hope this information will prove to be useful both for building more energy-efficient processors and understanding why and where customization must be used for efficiency.

To build this understanding, we start with a single video compression application, 720p HD H.264 video encode, and transform the hardware it runs on from a generic multiprocessor to a custom multiprocessor with ASIC-like specialized hardware units. On this task, a general-purpose software solution takes 500× more energy per frame and 500× more area than an ASIC to reach the same performance. We choose H.264 because it demonstrates the large energy advantage of ASIC solutions (500×) and because there exist commercial ASICs that can serve as a benchmark. Moreover, H.264 contains a variety of computational motifs, from highly data-parallel algorithms (motion estimation) to control intensive ones (Context Adaptive Binary Arithmetic Coding [CABAC]).

To better understand the potential of producing general-purpose chips with better efficiency, we consider two broad strategies for customized hardware. The first extends the current trend of creating general data-parallel engines on our processors. This approach mimics the addition of SSE instructions, or the recent work in merging graphic processors on die to help with other applications. We claim these are similar to general functional units since they typically have some special instructions for important applications, but are still generally useful. The second approach creates application-specific data storage fused with functional units. In the limit this should be an ASIC-like solution. The first has the advantage of being a programmable solution, while the second provides potentially greater efficiency.

The results are striking. Starting from a 500× energy penalty, adding relatively wide SSE-like parallel execution engines and rewriting the code to use them improves performance/

A previous version of this paper was published in *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ACM, NY.

area by  $14\times$  and energy efficiency by  $10\times$ . Despite these customizations, the resulting solution is still  $50\times$  less energy efficient than an ASIC. An examination of the energy breakdown in the paper clearly demonstrates why. Basic arithmetic operations are typically 8–16 bits wide, and even when performing more than 10 such operations per cycle, arithmetic unit energy comprises less than 10% of the total. One must consider the energy cost of the desired operation compared with the energy cost of one processor cycle: for highly efficient machines, these energies should be similar.

The next section provides the background needed to understand the rest of the paper. Section 3 then presents our experimental methodology, describing our baseline, generic H.264 implementation on a Tensilica CMP. The performance and efficiency gains are described in Section 4, which also explores the causes of the overheads and different methods for addressing them. Using the insight gained from our results, Section 5 discusses the broader implications for efficient computing and supporting application driven design.

## 2. BACKGROUND

We first review the basic ways one can analyze power, and some previous work in creating energy-efficient processors. With this background, we then provide an overview of H.264 encoding and its main compute stages. The section ends by comparing existing hardware and software implementations of an H.264 encoder.

### 2.1. Power-constrained design and energy efficiency

Power is defined to be energy per second, which can be broken up into two terms,  $\text{energy/op} * \text{ops/second}$ . Thus there are two primary means by which a designer can reduce power consumption: reduce the number of operations per second or reduce the energy per operation. The first approach—reducing the operations per second—simply reduces performance to save power. This approach is analogous to slowing down a factory’s assembly line to save electricity costs; although power consumption is reduced, the factory output is also reduced and the energy used (i.e., the electricity bill) per unit of output remains unchanged. If, on the other hand, a designer wishes to maintain or improve the performance under a fixed power budget, a reduction in the fundamental energy per operation is required. It is this reduction in energy per operation—not power—that represents real gains in efficiency.

This distinction between power and energy is an important one. Even though designers typically face physical *power* constraints, to increase efficiency requires that the fundamental *energy* of operations be reduced. Although one might be tempted to report power numbers when discussing power efficiency, this can be misleading if the performance is not also reported. What may seem like a power efficiency gain may just be a modulation in performance. Using energy per operation, however, is a performance-invariant metric that represents the fundamental efficiency of the work being done. Thus, even though the designer may be facing a *power* constraint, it is *energy per operation* that the designer needs to focus on improving.

Reducing the energy required for the basic operation can be achieved through a number of techniques, all of which fundamentally reduce the overhead affiliated with the work being done. As one simple example, clock gating improves energy efficiency by eliminating spurious activity in a chip that otherwise causes energy waste.<sup>8</sup> As another example, customized hardware can increase efficiency by eliminating overheads. The next section further discusses the use of customization.

### 2.2. Related work in efficient computing

Processors are often customized to improve their efficiency for specific application domains. For example, SIMD architectures achieve higher performance for multimedia and other data-parallel applications, while DSP processors are tailored for signal-processing tasks. More recently, ELM<sup>1</sup> and AnySP<sup>24</sup> have been optimized for embedded and mobile signal processing applications, respectively, by reducing processor overheads. While these strategies target a broad spectrum of applications, special instructions are sometimes added to speed up specific applications. For example, Intel’s SSE4<sup>10</sup> includes instructions to accelerate matrix transpose and sum-of-absolute-differences.

Customizable processors allow designers to take the next step, and create instructions tailored to applications. Extensible processors such as Tensilica’s Xtensa provide a base design that the designer can extend with custom instructions and datapath units.<sup>15</sup> Tensilica provides an automated ISA extension tool,<sup>20</sup> which achieves speedups of  $1.2\times$  to  $3\times$  for EEMBC benchmarks and signal processing algorithms.<sup>21</sup> Other tools have similarly demonstrated significant gains from automated ISA extension.<sup>4, 5</sup> While automatic ISA extensions can be very effective, manually creating ISA extensions gives even larger gains: Tensilica reports speedups of  $40\times$  to  $300\times$  for kernels such as FFT, AES, and DES encryption.<sup>18, 19, 22</sup>

Recently researchers have proposed another approach for achieving energy efficiency—reducing the cost of creating customized hardware rather than customizing a processor. Examples of the latter include using higher levels of abstraction (e.g., C-to-RTL<sup>13</sup>) and even full chip generators using extensible processors.<sup>16</sup> Independent of whether one customizes a processor, or creates customized hardware, it is important to understand in quantitative terms the types and magnitudes of energy overheads in processors.

While previous studies have demonstrated significant improvements in performance and efficiency moving from general-purpose processors to ASICs, we explore the reasons for these gains, which is essential to determine the nature and degree of customization necessary for future systems. Our approach starts with a generic CMP system. We incrementally customize its memory system and processors to determine the magnitude and sources of overhead eliminated in each step toward achieving a high efficiency 720p HD H.264 encoder. We explore the basic computation in H.264 next.

### 2.3. H.264 computational motifs

H.264 is a block-based video encoder which divides each

video frame into  $16 \times 16$  macro-blocks and encodes each one separately. Each block goes through five major functions:

- i. IME: Integer Motion Estimation
- ii. FME: Fractional Motion Estimation
- iii. IP: Intra Prediction
- iv. DCT/Quant: Transform and Quantization
- v. CABAC: Context Adaptive Binary Arithmetic Coding

IME finds the closest match for an image block versus a previous reference image. While it is one of the most compute intensive parts of the encoder, the basic algorithm lends itself well to data-parallel architectures. On our base CMP, IME takes up 56% of the total encoder execution time and 52% of total energy.

The next step, FME, refines the initial match from integer motion estimation and finds a match at quarter-pixel resolution. FME is also data parallel, but it has some sequential dependencies and a more complex computation kernel that makes it more difficult to parallelize. FME takes up 36% of the total execution time and 40% of total energy on our base CMP design. Since FME and IME together dominate the computational load of the encoder, optimizing these algorithms is essential for an efficient H.264 system design.

IP uses previously encoded neighboring image blocks within the current frame to form an alternate prediction for the current image-block. While the algorithm is still dominated by arithmetic operations, the computations are much less regular than the motion estimation algorithms. Additionally, there are sequential dependencies not only within the algorithm but also with the transform and quantization function.

Next, in DCT/Quant, the difference between a current and predicted image block is transformed and quantized to generate coefficients to be encoded. The basic function is relatively simple and data parallel. However, it is invoked a number of times for each  $16 \times 16$  image block, which calls for an efficient implementation. For the rest of this paper, we merge these operations into the IP stage. The combined operation accounts for 7% of the total execution time and 6% of total energy.

Finally, CABAC is used to entropy-encode the coefficients and other elements of the bit-stream. Unlike the previous algorithms, CABAC is sequential and control dominated. While it takes only 1.6% of the execution time and 1.7% of total energy on our base design, CABAC often becomes the bottleneck in parallel systems due to its sequential nature.

## 2.4. Current H.264 implementations

The computationally intensive H.264 encoding algorithm poses a challenge for general-purpose processors, and is typically implemented as an ASIC. For example, T. C. Chen et al. implement a full-system H.264 encoder and demonstrate that real-time HD H.264 encoding is possible in hardware using relatively low power and area cost.<sup>2</sup>

H.264 software optimizations exist, particularly for motion estimation, which takes most of the encoding time. For example, sparse search techniques speed performance of IME and FME by up to  $10 \times$ .<sup>14, 25</sup> Combining aggressive

algorithmic modifications with multiple cores and SSE extensions leads to highly optimized H.264 encoders on Intel processors.<sup>3, 12</sup>

Despite these optimizations, software implementations of H.264 lag far behind dedicated ASICs. Table 1 compares a software implementation of a 480p SD encoder<sup>12</sup> to a 720p HD ASIC implementation.<sup>2</sup> The software implementation employs a 2.8GHz Intel Pentium 4 executing highly optimized SSE code. This results in very high energy consumption and low area efficiency. It is also worth noting that the software implementation relies on various algorithmic simplifications, which drastically reduce the computational complexity, but result in a 20% decrease in compression efficiency for a given SNR. The ASIC hardware, on the other hand, consumes over  $500 \times$  less energy and is far more efficient in its use of silicon area and has a negligible drop in compression efficiency.

## 3. EXPERIMENTAL METHODOLOGY

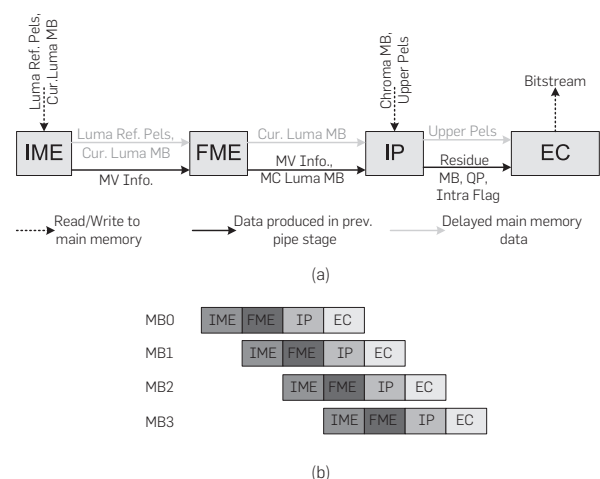
To understand what is needed to gain ASIC level efficiency, we use existing H.264 partitioning techniques, and modify the H.264 encoder reference code JM 8.6<sup>11</sup> to remove dependencies and allow mapping of the five major algorithmic blocks to the four-stage macro-block (MB) pipeline shown in Figure 1. This mapping exploits task level parallelism at the macro-block level and significantly reduces the inter-processor communication bandwidth requirements by sharing data between pipeline stages.

**Table 1. Intel's optimized H.264 encoder versus a 720p HD ASIC.**

	FPS	Area (mm <sup>2</sup> )	Energy/Frame (mJ)
Intel (720 × 480 SD)	30	122	742
Intel (1280 × 720 HD)	11	122	2023
ASIC	30	8	4

The second row gives Intel's SD data scaled to HD. ASIC data is scaled from 180 down to 90nm.

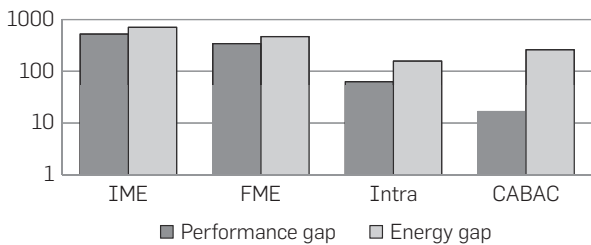
**Figure 1. Four stage macroblock partition of H.264. (a) Data flow between stages. (b) How the pipeline works on different macroblocks. IP includes DCT+Quant. EC is CABAC.**



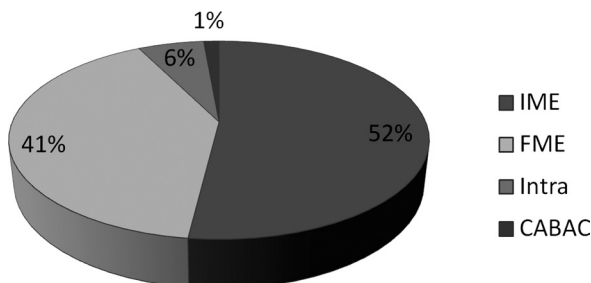
In the base system, we map this four-stage macro-block partition to a four-processor CMP system where each processor has 16KB 2-way set associative instruction and data caches. Figure 2 highlights the large efficiency gap between our base CMP and the reference ASIC for individual 720p HD H.264 subalgorithms. The energy required for each RISC instruction is similar and as a result, the energy required for each task (shown in Figure 3) is related to the cycles spent on that task. The RISC implementation of IME, which is the major contributor to performance and energy consumption, has a performance gap of 525× and an energy gap of over 700× compared to the ASIC. IME and FME dominate the overall energy and thus need to be aggressively optimized. However, we also note that while IP, DCT, Quant, and CABAC are much smaller parts of the total energy/delay, even they need about 100× energy improvement to reach ASIC levels.

At approximately 8.6B instructions to process 1 frame, our base system consumes about 140 pJ per instruction—a reasonable value for a general-purpose system. To further analyze the energy efficiency of this base CMP implementation we break the processor’s energy into different functional units as shown in Figure 4. This data makes it clear how far we need to go to approach ASIC efficiency. The energy spent in instruction fetch (IF) is an overhead due to the programmable nature of the processors and is absent in a custom hardware state machine, but eliminating all this overhead only increases the energy efficiency by less than one third. Even if we eliminate everything but the functional unit energy, we still end up with energy savings of only 20×—not nearly enough to reach ASIC levels.

**Figure 2. The performance and energy gap for base CMP implementation when compared to an equivalent ASIC. Intra combines IP, DCT, and Quant.**



**Figure 3. Processor energy breakdown for base implementation, over the different H.264 subalgorithms. Intra combines IP, DCT, and Quant.**



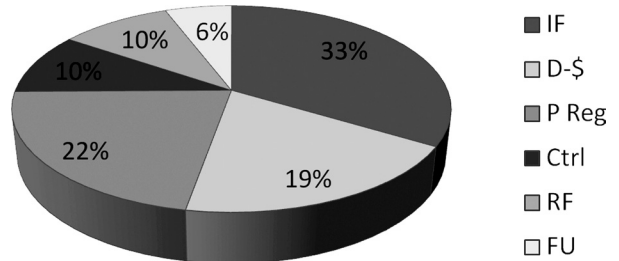
The next section explores what customizations are needed to reach the efficiency goals.

#### 4. CUSTOMIZATION RESULTS

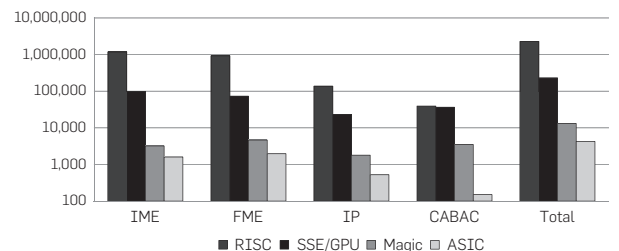
At first, we restrict our customizations to datapath extensions inspired by GPUs and Intel’s SSE instructions. Such extensions are relatively general-purpose data-parallel optimizations and consist of single instruction, multiple data (SIMD) and multiple instruction issue per cycle (we use long instruction word, or LIW), with a limited degree of algorithm-specific customization coming in the form of operation fusion—the creation of new instructions that combine frequently occurring sequences of instructions. However, much like their SSE and GPU counterparts, these new instructions are constrained to the existing instruction formats and datapath structures. This step represents the datapaths in current state-of-the-art optimized CPUs. In the next step, we replace these generic datapaths by custom units, and allow unrestricted tailoring of the datapath by introducing arbitrary new compute operations as well as by adding custom register file structures.

The results of these customizations are shown in Figures 5 through 7. The rest of this section describes these results in detail and evaluates the effectiveness of these three customization strategies. Collectively, these results describe how efficiencies improve by 170× over the baseline of Section 3.

**Figure 4. Processor energy breakdown for base implementation. IF is instruction fetch/decode. D-\$\$ is data cache. P Reg includes the pipeline registers, buses, and clocking. Ctrl is miscellaneous control. RF is register file. FU is the functional units.**



**Figure 5. Each set of bar graphs represents energy consumption (μJ) at each stage of optimization for IME, FME, IP and CABAC respectively. The first bar in each set represents base RISC energy; followed by RISC augmented with SSE/GPU style extensions; and then RISC augmented with “magic” instructions. The last bar in each group indicates energy consumption by the ASIC.**



#### 4.1. SSE/GPU style enhancements

Using Tensilica's TIE extensions we add LIW instructions and SIMD execution units with vector register files of custom depths and widths. A single SIMD instruction performs multiple operations (8 for IP, 16 for IME, and 18 for FME), reducing the number of instructions and consequently reducing IF energy. LIW instructions execute 2 or 3 operations per cycle, further reducing cycle count. Moreover, SIMD operations perform wider register file and data cache accesses which are more energy efficient compared to narrower accesses. Therefore all components of instruction energy depicted in Figure 4 get a reduction through the use of these enhancements.

We further augment these enhancements with operation fusion, in which we fuse together frequently occurring complex instruction sub-graphs for both RISC and SIMD instructions. To prevent the register file ports from increasing, these instructions are restricted to use up to two input operands and can produce only one output. Operation fusion improves energy efficiency by reducing the number of instructions and also reducing the number of register file accesses by internally consuming short-lived intermediate data. Additionally, fusion gives us the ability to create more

energy-efficient hardware implementations of the fused operations, e.g., multiplication implemented using shifts and adds. The reductions due to operation fusion are less than 2× in energy and less than 2.5× in performance.

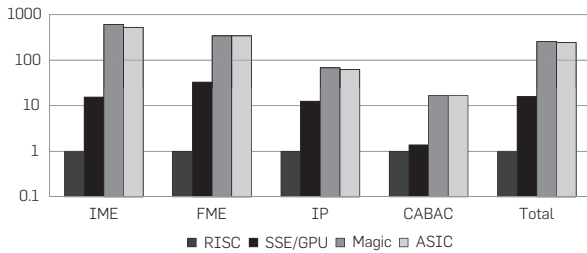
With SIMD, LIW and Op Fusion support, IME, FME and IP processors achieve speedups of around 15×, 30× and 10×, respectively. CABAC is not data parallel and benefits only from LIW and op fusion with a speedup of merely 1.1× and almost no change in energy per operation. Overall, the application gets an energy efficiency gain of almost 10×, but still uses greater than 50× more energy than an ASIC. To reach ASIC levels of efficiency, we need a different approach.

#### 4.2. Algorithm specific instructions

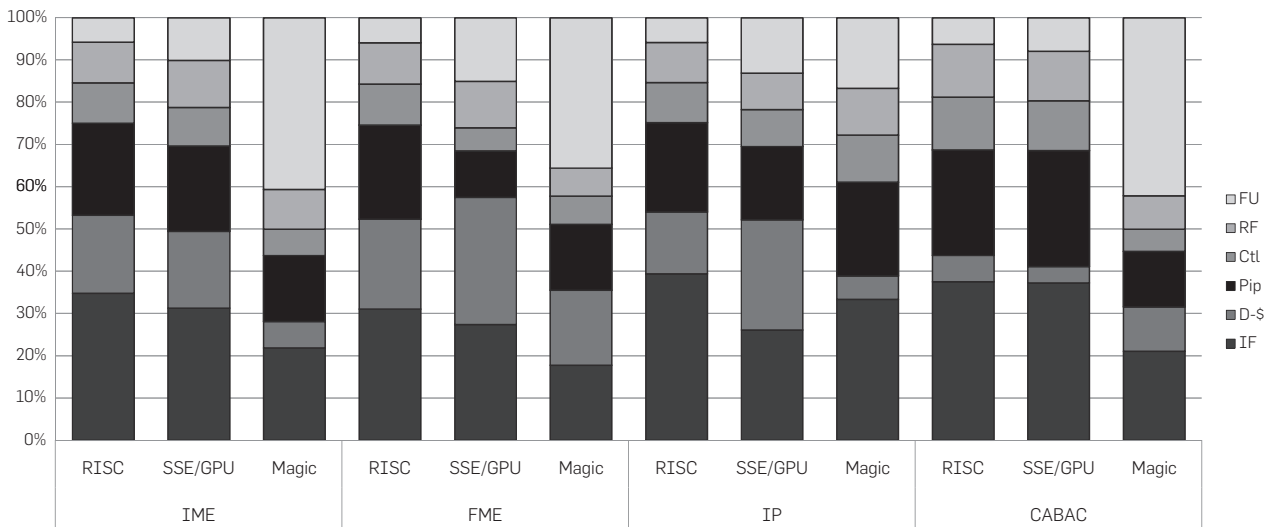
The root cause of the large energy difference is that the basic operations in H.264 are very simple and low energy. They only require 8–16 bit integer operations, so the fundamental energy per operation bound is on the order of hundreds of femtojoules in a 90 nm process. All other costs in a processor—IF, register fetch, data fetch, control, and pipeline registers—are much larger (140 pJ) and dominate overall power. Standard SIMD and simple fused instructions can only go so far to improve the performance and energy efficiency. It is hard to aggregate more than 10–20 operations into an instruction without incurring growing inefficiencies, and with tens of operations per cycle we still have a machine where around 90% of the energy is going into overhead functions. It is now easy to see how an ASIC can be 2–3 orders of magnitude lower energy than a processor. For computationally limited applications with low-energy operations, an ASIC can implement hardware which both has low overheads, and is a perfect structural match to the application. These features allow it to exploit large amounts of parallelism efficiently.

To match these results in a processor we must amortize the per-instruction energy overheads over hundreds of these

**Figure 6. Speedup at each stage of optimization for IME, FME, IP and CABAC.**



**Figure 7. Processor energy breakdown for H.264. IF is instruction fetch/decode. D-\$ is data cache. Pip is the pipeline registers, buses, and clocking. Ctl is random control. RF is the register file. FU is the functional elements. Only the top bar or two (FU, RF) contribute useful work in the processor. For this application it is hard to achieve much more than 10% of the power in the FU without adding custom hardware units.**

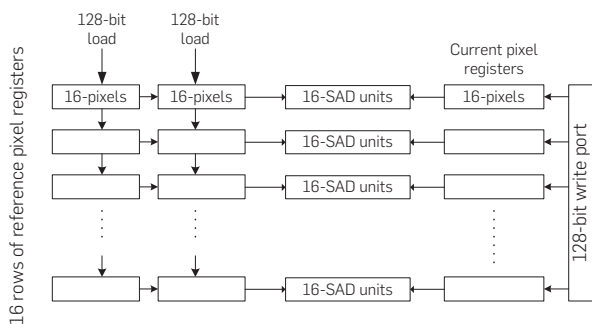


simple operations. To create instructions with this level of parallelism requires custom storage structures with algorithm-specific communication links to directly feed large amounts of data to custom functional units without explicit register accesses. These structures also substantially increase data-reuse in the datapath and reduce communication bandwidth and power at all levels of the memory hierarchy (register, cache, and memory).

Once this hardware is in place, the machine can issue “magic” instructions that accomplish large amounts of computation at very low cost. This type of structure eliminates almost all the processor overheads for these functions by eliminating most of the communication overhead associated with processors. We call these instructions “magic” because they can have a large effect on both the energy and performance of an application and yet they would be difficult to derive directly from the code. Such instructions typically require an understanding of the underlying algorithms, as well as the capabilities and limitations of existing hardware resources, thus requiring greater effort on the part of the designer. Since the IP stage uses techniques similar to FME, the rest of the section will focus on IME, FME, and CABAC.

**IME Strategy:** To demonstrate the nature and benefit of magic instructions we first look at IME, which determines the best alignment for two image blocks. The best match is defined by the smallest sum-of-absolute-differences (SAD) of all of the pixel values. Since finding the best match requires scanning one image block over a larger piece of the image, one can easily see that while this requires a large number of calculations, it also has very high data locality. Figure 8 shows the custom datapath elements added to the IME processor to accelerate this function. At the core is a  $16 \times 16$  SAD array, which can perform 256 SAD operations in 1 cycle. Since our standard vector register files cannot feed enough data to this unit per cycle, the SAD unit is fed by a custom register structure, which allows parallel access to all 16-pixel rows and enables this datapath to perform one 256-pixel computation per cycle. In addition, the intermediate results of the pixel operations need not be stored since they can be reduced in place (summed) to create the single desired output. Furthermore, because we need to check many possible

**Figure 8. Custom storage and compute for IME  $4 \times 4$  SAD. Current and ref-pixel register files feed all pixels to the  $16 \times 16$  SAD array in parallel. Also, the ref-pixel register file allows horizontal and vertical shifts.**



alignments, the custom storage structure has support for parallel shifts in all four directions, thus allowing one to shift the entire comparison image in only one cycle. This feature drastically reduces the instructions wasted on loads, shifts, and pointer arithmetic operations as well as data cache accesses. “Magic” instructions and storage elements are also created for other major algorithmic functions in IME to achieve similar gains.

Thus, by reducing instruction overheads and by amortizing the remaining overheads over larger datapath widths, this functional unit finally consumes around 40% of the total instruction energy. The performance and energy efficiency improve by 200–300 $\times$  over the base implementation, match the ASIC’s performance and come within 3 $\times$  of ASIC energy. This customized solution is 20–30 $\times$  better than the results using only generic data-parallel techniques.

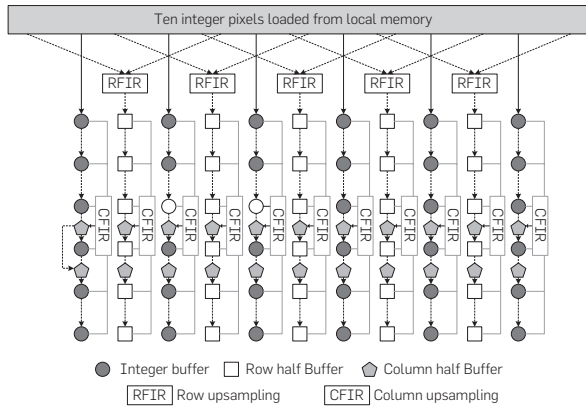
**FME Strategy:** FME improves the output of the IME stage by refining the alignment to a fraction of a pixel. To perform the fractional alignment, the FME stage interpolates one image to estimate the values of a  $4 \times 4$  pixel block at fractional pixel coordinates. This operation is done by a filter and upsample block, which again has high arithmetic intensity and high data locality. In H.264, upsampling uses a six tap FIR filter that requires one new pixel per iteration. To reduce IFs and register file transfers, we augment the processor register file with a custom 8 bit wide, 6 entry shift register structure which works like a FIFO: every time a new 8 bit value is loaded, all elements are shifted. This eliminates the use of expensive register file accesses for either data shifting or operand fetch, which are now both handled by short local wires. All six entries can now be accessed in parallel and we create a six input multiplier/adder which can do the calculation in a single cycle and also can be implemented much more efficiently than the composition of normal 2-input adders. Finally, since we need to perform the upsampling in 2-D, we build a shift register structure that stores the horizontally upsampled data, and feeds its outputs to a number of vertical upsampling units (Figure 9).

This transformation yields large savings even beyond the savings in IF energy. From a pure datapath perspective (register file, pipeline registers, and functional units), this approach dissipates less than 1/30th the energy of a traditional approach.

A look at the FME SIMD code implementation highlights the advantages of this custom hardware approach versus the use of larger SIMD arrays. The SIMD implementation suffers from code replication and excessive local memory and register file accesses, in addition to not having the most efficient functional units. FME contains seven different sub-block sizes ranging from  $16 \times 16$  pixel blocks to  $4 \times 4$  blocks, and not all of them can fully exploit the 18-way SIMD datapath. Additionally, to use the 18-way SIMD datapath, each sub-block requires a slightly different code sequence, which results in code replication and more I-fetch power because of the larger I-cache.

To avoid these issues, the custom hardware upsampler processes  $4 \times 4$  pixels. This allows it to reuse the same computation loop repeatedly without any code replication,

**Figure 9. FME upsampling unit. Customized shift registers, directly wired to function logic, result in efficient upsampling. Ten integer pixels from local memory are used for row upsampling in RFIR blocks. Half upsampled pixels along with appropriate integer pixels are loaded into shift registers. CFIR accesses six shift registers in each column simultaneously to perform column upsampling.**



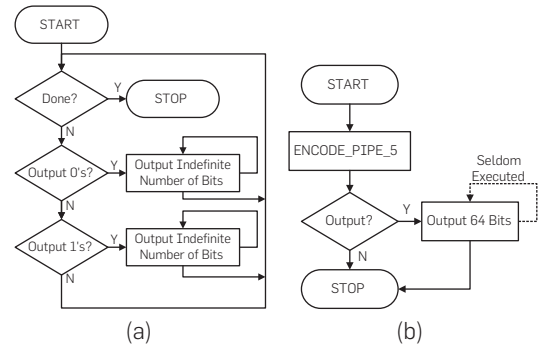
which, in turn, lets us reduce the I-cache from a 16KB 4-way cache to a 2KB direct-mapped cache. Due to the abundance of short-lived data, we remove the vector register files and replace them with custom storage buffers. The “magic” instruction reduces the instruction cache energy by 54× and processor fetch and decode energy by 14×. Finally, as Figure 7 shows, 35% of the energy is now going into the functional units, and again the energy efficiency of this unit is close to an ASIC.

**CABAC Strategy:** CABAC originally consumed less than 2% of the total energy, but after data-parallel components are accelerated by “magic” instructions, CABAC dominates the total energy. However, it requires a different set of optimizations because it is control oriented and not data parallel. Thus, for CABAC, we are more interested in control fusion than operation fusion.

A critical part of CABAC is the arithmetic encoding stage, which is a serial process with small amounts of computation, but complex control flow. We break arithmetic coding down into a simple pipeline and drastically change it from the reference code implementation, reducing the binary encoding of each symbol to five instructions. While there are several if-then-else conditionals reduced to single instructions (or with several compressed into one), the most significant reduction came in the encoding loop, as shown in Figure 10a. Each iteration of this loop may or may not trigger execution of an internal loop that outputs an indefinite number of encoded bits. By fundamentally changing the algorithm, the while loop was reduced to a single constant time instruction (ENCODE\_PIPE\_5) and a rarely executed while loop, as shown in Figure 10b.

The other critical part of CABAC is the conversion of non-binary-valued DCT coefficients to binary codes in the binarization stage. To improve the efficiency of this step, we create a 16-entry LIFO structure to store DCT coefficients. To each LIFO entry, we add a single-bit flag to identify zero-valued DCT coefficients. These structures, along with their

**Figure 10. CABAC Arithmetic Encoding Loop (a) H.264 reference code. (b) After insertion of “magic” instructions. Much of the control logic in the main loop has been reduced to one constant time instruction ENCODE\_PIPE\_5.**



corresponding logic, reduce register file energy by bringing the most frequently used values out of the register file and into custom storage buffers. Using “magic” instructions we produce Unary and Exponential-Golomb codes using simple operations, which help reduce datapath energy. These modifications are inspired by the ASIC implementation described in Shojania and Sudharsanan.<sup>17</sup> CABAC is optimized to achieve the bit rate required for H.264 level 3.1 at 720p video resolution.

**Magic Instructions Summary:** To summarize, the magic instructions perform up to hundreds of operations each time they are executed, so the overhead of the instruction is better balanced by the work performed. Of course this is hard to do in a general way, since bandwidth requirements and utilization of a larger SIMD array would be problematic. Therefore we solved this problem by building custom storage units tailored to the application, and then directly connecting the necessary functional units to these storage units. These custom storage units greatly amplified the register fetch bandwidth, since data in the storage units is used for many different computations. In addition, since the intra-storage and functional unit communications were fixed and local, they can be managed at ASIC-like energy costs.

After this effort, the processors optimized for data-parallel algorithms have a total speedup of up to 600× and an energy reduction of 60–350× compared to our base CMP. For CABAC total performance gain is 17× and energy gain is 8×. Figure 7 provides the final energy breakdowns. The efficiencies found in these custom datapaths are impressive, since, in H.264 at least, they take advantage of data sharing patterns and create very efficient multiple-input operations. This means that even if researchers are able to create a processor which decreases the instruction and data fetch parts of a processor by more than 10×, these solutions will not be as efficient as solutions with “magic” instructions.

Achieving ASIC-like efficiency required 2–3 special hardware units for each subalgorithm, which is significant customization work. Some might even say we are just building an ASIC in our processor. While we agree that creating “magic” instructions requires a thorough understanding of

the application as well as hardware, we feel that adding this hardware in an extensible processor framework has many advantages over just designing an ASIC. These advantages come from the constrained processor design environment and the software, compiler, and debugging tools available in this environment. Many of the low-level issues, like interface design and pipelining, are automatically handled. In addition, since all hardware is wrapped in a general-purpose processor, the application developer retains enough flexibility in the processor to make future algorithmic modifications.

## 5. ENERGY-EFFICIENT COMPUTERS

It is important to remember that the “overhead” of using a processor depends on the energy required for the desired operation. Floating point (FP) energy costs are about  $10\times$  the small integer ops we have explored in this paper, so machines with 10 wide FP units will not be far from the maximum efficiency possible for that class of applications. Similarly, customizing the hardware will not have a large impact on the energy efficiency of an application dominated by memory costs; an ASIC and a processor’s energy will not be that different. For these applications, optimization that restructures the algorithm and/or the memory system is needed to reduce energy, and can yield large savings.<sup>23</sup>

Unfortunately, as we drive to more energy-efficient solutions, we will find ways to transform FP code to fixed point operations, and restructure our algorithms to minimize the memory fetch costs. Said differently, if we want ASIC-like energy efficiencies— $100\times$  to  $1000\times$  more energy efficient than general-purpose CPUs—we will have to transform our algorithms to be dominated by the simple, low-energy operations we have been studying in this paper. Since the energy of these operations is very low, any overhead, from the register fetch to the pipeline registers in a processor, is likely to dominate energy costs. The good news is that this large overhead per instruction makes estimating the energy savings easy—you simply look at the performance gains—but the bad news is that adding state-of-the-art data-parallel hardware like wide SIMD units and media extensions will still leave you far from the desired efficiency.

It is encouraging that we were able to achieve ASIC energy levels in a customized processor by creating customized hardware that easily fit inside a processor framework. Extending a processor instead of building an ASIC seems like the correct approach, since it provides a number of software development advantages and the energy cost of this option seems small. However, building such custom datapaths still requires a significant effort and thus the key challenge now is to build a design system that lets application designers create and exploit such customizations with much greater ease. The key is to find a parameterization of the space which makes sense to application designers in a specific application domain.

For example, often a number of algorithms in a domain share similar data flow and computation structures. In H.264 a common computational motif is based on a convolution-like data flow: apply a function to all the data, then perform a reduction, then shift the data and add a small

amount of new data, and repeat. A similar pattern of convolution-like computations also exists in a number of other image processing and media processing algorithms. While the exact computation is going to be different for each particular algorithm, we believe that by exploiting the common data-flow structure of these algorithms we can create a generalized convolution abstraction which application designers can customize. If this abstraction is useful for application designers, one can imagine implementing it by creating a flexible hardware unit that is significantly more efficient than a generic SIMD/SSE unit. We also believe that similar patterns exist in other domains that may allow us to create a set of customized units for each domain.


Even if we could come up with such a set of customized functional units, it is likely that some degree of per algorithm configurability will be required. For example, in a convolution engine, the convolution size and resulting datapath size could vary from algorithm to algorithm and thus potentially needs to be tuned on a per processor basis. This leads to the idea of creating a two-step design process. The first step is when a set of chip experts design a processor generator platform. This is a meta-level design which “knows” about the special functional units and control optimization, and provides the application designer an application-tailored interface. The application designers can then co-optimize their code and the interface parameters to meet their requirements. After this co-optimization, an optimized implementation based on these parameters is automatically generated. In fact, such a platform will also help in building the more generic domain customized functional units mentioned earlier by facilitating the process of rapidly creating and evaluating new designs.

A reconfigurable processor generator alone is not a sufficient solution, since one still needs to take one or more of these processors and create a working chip system. Designing and validating a chip is an extremely hard and expensive task. If application customization will be needed for efficiency—and our data indicates it will be—we need to start creating systems that will efficiently allow savvy application experts to create these optimized chip level solutions. This will require extending the ideas for extensible processors to full chip generation systems. We are currently working on creating this type of system.<sup>16</sup>

## Acknowledgments

This work would have not been possible without great support and cooperation from many people at Tensilica including Chris Rowen, Dror Maydan, Bill Huffman, Nenad Nedeljkovic, David Heine, Govind Kamat, and others. The authors acknowledge the support of the C2S2 Focus Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation subsidiary, and earlier support from DARPA. This material is based upon work partially supported under a Sequoia Capital Stanford Graduate Fellowship. The National Science Foundation under Grant #0937060 to the Computing Research Association also supports this material for the CIFellows Project. Any opinions, findings,



and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the view of the National Science Foundation or the Computing Research Association. 

#### References

- Balfour, J., Dally, W., Black-Schaffer, D., Parikh, V., Park, J. An energy-efficient processor architecture for embedded systems. *Comput. Archit. Lett.* 7.1 (2007), 29–32.
- Chen, T.C. Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder. *IEEE Trans. Circuits Syst. Video Technol.* 16, 6 (2006), 673–688.
- Chen, Y.-K., Li, E.Q., Zhou, X., Ge, S. Implementation of H.264 encoder and decoder on personal computers. *J Vis. Commun. Image Represent.* 17 (2006), 509–532.
- Clark, N., Zhong, H., Mahlke, S. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Trans. Comput.* 54, 10 (2005), 1258–1270.
- Cong, J., Fan, Y., Han, G., Zhang, Z. Application-specific instruction generation for configurable processor architectures. In *12th International Symposium on Field Programmable Gate Arrays* (2004), 183–189.
- Davis, W., Zhang, N., Camera, K., Chen, F., Markovic, D., Chan, N., Nikolic, B., Brodersen, R. A design environment for high throughput, low power, dedicated signal processing systems. In *Custom Integrated Circuits Conference (CICC)* (2001).
- Dennard, R., Gaensslen, F., Yu, H., Rideout, V., Bassous, E., LeBlanc, A. Design of ion-implanted MOSFET's with very small physical dimensions. *Proc. IEEE (reprinted from IEEE J Solid-State Circuits, 1974)*, 87, 4 (1999), 668–678.
- Esmailzadeh, H., Blem, E., Amant, R.S., Sankaralingam, K., Burger, D. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture* (June 2011).
- Horowitz, M. Scaling, power and the future of CMOS. In *Proceedings of the 20th International Conference on VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems* (2007), 23–23.
- Intel Corporation. Motion Estimation with Intel Streaming SIMD Extensions 4 (Intel SSE4) (2008).
- ITU-T. Joint Video Team Reference Software JM8.6 (2004).
- Iverson, V., McVeigh, J., Reese, B. Real-time H.264/AVC Codec on Intel architectures. In *IEEE International Conference on Image Processing ICIP'04* (2004).
- Kathail, V. Creating power-efficient application engines for SOC design. *SOC Central* (2005).
- MPEG, I., VCEG, I.-T. Fast integer pel and fractional pel motion estimation for JVT. *JVT-F017* (2002).
- Rowen, C., Leibson, S. Flexible architectures for engineering successful SOCs. In *Proceedings of the 41st Annual Design Automation Conference* (2004), 692–697.
- Shacham, O., Azizi, O., Wachs, M., Qadeer, W., Asgar, Z., Kelley, K., Stevenson, J., Solomatnikov, A., Firoozshahian, A., Lee, B., Richardson, S., Horowitz, M. Why design must change: Rethinking digital design. *IEEE Micro* 30, 6 (Nov.–Dec. 2010), 9–24.
- Shojania, H., Sudharsanan, S. A VLSI architecture for high performance CABAC encoding. In *Visual Communications and Image Processing* (2005).
- Tensilica Inc. Implementing the advanced encryption standard on Xtensa processors. *Application Notes* (2009).
- Tensilica Inc. Implementing the fast Fourier transform (FFT). *Application Notes* (2005).
- Tensilica Inc. The what, why, and how of configurable processors (2008).
- Tensilica Inc. Xtensa LX2 benchmarks (2005).
- Tensilica Inc. Xtensa processor extensions for data encryption standard (DES). *Application Notes* (2008).
- Williams, S., Olike, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (2007).
- Woh, M., Seo, S., Mahlke, S., Mudge, T., Chakrabarti, C., Flautner, K. AnySP: Anytime anywhere anyway signal processing. *SIGARCH Comp. Arch. News* 37, 3 (2009), 128–139.
- Yin, P., Tourapis, H.-Y.C., Tourapis, A.M., Boyce, J. Fast mode decision and motion estimation for JVT/H.264. In *Proceedings of IEEE International Conference on Image Processing* (2003).

**Rehan Hameed** (rhameed@stanford.edu), Stanford University, Stanford, CA.

**Wajahat Qadeer** (wqadeer@stanford.edu), Stanford University, Stanford, CA.

**Megan Wachs** (wachs@stanford.edu), Stanford University, Stanford, CA.

**Omid Azizi** (oazizi@gmail.com), Hicamp Systems, Menlo Park, CA.

**Alex Solomatnikov** (Solomatnikov@gmail.com), Hicamp Systems, Menlo Park, CA.

**Benjamin C. Lee** (benjamin.c.lee@duke.edu), Duke University, Durham, NC.

**Stephen Richardson** (steveri@stanford.edu), Stanford University, Stanford, CA.

**Christos Kozyrakis** (kozyraki@stanford.edu), Stanford University, Stanford, CA.

**Mark Horowitz** (horowitz@stanford.edu), Stanford University, Stanford, CA.

© 2011 ACM 0001-0782/11/10 \$10.00

# Take Advantage of ACM's Lifetime Membership Plan!

- ◆ **ACM Professional Members** can enjoy the convenience of making a single payment for their entire tenure as an ACM Member, and also be protected from future price increases by taking advantage of **ACM's Lifetime Membership** option.
- ◆ **ACM Lifetime Membership** dues may be tax deductible under certain circumstances, so becoming a Lifetime Member can have additional advantages if you act before the end of 2011. (Please consult with your tax advisor.)
- ◆ Lifetime Members receive a certificate of recognition suitable for framing, and enjoy all of the benefits of **ACM Professional Membership**.

Learn more and apply at:  
<http://www.acm.org/life>



Association for  
Computing Machinery

Advancing Computing as a Science & Profession