# Understanding Sources of Inefficiency in General-Purpose Chips

Rehan Hameed[1], Wajahat Qadeer[1], Megan Wachs[1], Omid Azizi[1], Alex Solomatnikov[2], Benjamin C. Lee[1], Stephen Richardson[1], Christos Kozyrakis[1] and Mark Horowitz[1]

[1]Dept. of Electrical Engineering
Stanford University, Stanford, CA
{rhameed, wqadeer, wachs, oazizi,
bcclee, steveri, kozyraki, horowitz}@stanford.edu

[2]Hicamp Systems,
Menlo Park, CA
solomatnikov@gmail.com

## ABSTRACT

Due to their high volume, general-purpose processors, and now chip multiprocessors (CMPs), are much more cost effective than ASICs, but lag significantly in terms of performance and energy efficiency. This paper explores the sources of these performance and energy overheads in general-purpose processing systems by quantifying the overheads of a 720p HD H.264 encoder running on a general-purpose CMP system. It then explores methods to eliminate these overheads by transforming the CPU into a specialized system for H.264 encoding. We evaluate the gains from customizations useful to broad classes of algorithms, such as SIMD units, as well as those specific to particular computation, such as customized storage and functional units.

The ASIC is 500x more energy efficient than our original four-processor CMP. Broadly, applicable optimizations improve performance by 10x and energy by 7x. However, the very low energy costs of actual core ops (100s fJ in 90nm) mean that over 90% of the energy used in these solutions is still "overhead". Achieving ASIC-like performance and efficiency requires algorithm-specific optimizations. For each sub-algorithm of H.264, we create a large, specialized functional unit that is capable of executing 100s of operations per instruction. This improves performance and energy by an additional 25x and the final customized CMP matches an ASIC solution's performance within 3x of its energy and within comparable area.

## Categories and Subject Descriptors

C.5.4 [**Computer Systems Implementation]:** VLSI Systems – *customization, heterogeneous CMP*; C.1.3 [**Processor Architectures**]: Other Architecture Styles - *Heterogeneous (Hybrid) Systems*.

## General Terms

Algorithms, Measurement, Performance, Design, Experimentation.

## Keywords

ASIC, H.264, chip multiprocessor, high-performance, energy efficiency, customization, Tensilica.


*ISCA'10*, June 19–23, 2010, Saint-Malo, France.


## 1. INTRODUCTION

Most computing systems today are power limited, whether it is the 1W limit of a cell phone, or the 100W limit of a server. Since technology scaling no longer provides the energy savings, it once did [1], designers must turn to other techniques for continued performance improvements and tractable energy costs. One attractive option is to understand and to incorporate sources of ASIC efficiency, since general-purpose processors can be outclassed by three orders of magnitude in both performance and energy efficiency by ASIC designs [5].

The desire to achieve ASIC-like compute efficiencies with microprocessor-like application development cost is pushing designers to explore two new areas. One area aims to create CPU designs with much lower energy per instruction [6], while the other aims to create new design methodologies to reduce the cost of creating customized hardware. Examples of the latter include using higher levels of abstraction (e.g., C-to-RTL [8], [7]), and even full chip generators using extensible processors [2]. A critical first step in all of these approaches is to understand, in quantitative terms, the types and magnitudes of energy overheads in general-purpose processors. Once these are understood, it is then possible to explore ways to eliminate these overheads and assess the feasibility of creating an efficient, general-purpose machine.

This paper quantifies general-purpose overheads, exploring a series of customizations that reduce overheads to achieve ASIC-like efficiency. In particular, we consider three broad strategies: (1) techniques to exploit instruction- and data-level parallelism, such as VLIW and SIMD, (2) techniques to customize instructions by fusing complex, frequently occurring instruction sub-graphs, and (3) techniques to create application-specific data storage with fused functional units. These strategies span a range of general and domain-specific customization, incurring progressively greater design effort.

We evaluate these strategies by transforming a general-purpose, Tensilica-based, extensible CMP system into a highly efficient 720p HD H.264 encoder. We choose H.264 because it demonstrates the large energy advantage of ASIC solutions (500x) 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 (CABAC).

The results are striking. Starting from a 500x energy penalty, adding relatively wide (16x) SIMD execution units improves

Adding specific functions
- demonstrate user base
- demonstrate adaptability to chip design
- only so much space on the chip (more cores, less specific functionality / less cores, more SF)
* Reticle limit (used by Oracle) : Chips are built to manufacture processes for photographic chips ("cannot manage photography on anything wider than this").

performance by 10x and energy efficiency by 7x. Since SIMD units are often augmented with special fused instructions to accelerate important applications, we introduce our own custom fused instructions to improve both performance and energy efficiency by an additional 1.4x. Despite these customizations, which collectively improve energy efficiency by 10x, the resulting solution is still 50x less energy efficient than an ASIC.

An examination of the energy breakdown clearly demonstrates why. Since the SIMD unit customizes datapath widths of 8-12bits, functional unit energy comprises less than 10 percent of the total even when performing more than 10 operations per cycle. Thus, to create a truly efficient processor, one needs to construct instructions that aggregate enough computation to offset the energy overheads of flexible instruction and data fetch. Creating such "magic" instructions improves energy efficiency by another 18x and yields a solution within 3x of a full ASIC design.

While identifying the right customizations for a given application takes significant effort, it is hard to achieve ASIC-like efficiencies without them. The inescapable conclusion is that truly efficient designs will require application-specialized hardware. If energy efficiency is going to drive future computing design, then we need frameworks that allow application experts to easily (and at low cost) create customized solutions. The fact that, for our application, we can achieve good efficiency using processor instruction extensions is an encouraging sign.

Since our experiments use an extensible processor, the next section reviews some of the prior work in this area, provides an overview of H.264 encoding, and describes the performance of hardware and software solutions. Section 3 then presents our experimental methodology, describing our baseline, generic H.264 implementation on a Tensilica CMP and outlining our strategies for customizing this system. 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

Since we use an extensible processor for our case study, we first describe prior work on efficient computing, focusing on processor extensions. With this background, we then provide an overview of H.264 encoding and its main compute stages. The section ends by describing hardware and software implementations to demonstrate the performance advantages of an ASIC.

## 2.1 Related Work in Efficient Computing

General-purpose 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 to perform signal-processing tasks efficiently. More recently, ELM [6] and AnySP [10] have been optimized for embedded and mobile signal processing applications, respectively, by reducing processor overheads. While these strategies are meant to cover a broad spectrum of applications, special instructions are sometimes added to accelerate frequently used or critical operations for specific applications. For example, Intel's SSE4[11][12] 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 [9]. Extending the ISA for a given application can be done either manually or with automated tools. Tensilica provides an automated ISA extension tool [18], which achieves speedups of 1.2x to 30x for EEMBC benchmarks [17] and signal processing algorithms [16]. Other tools have similarly demonstrated significant gains from automated ISA extension [13][14]. While automatic ISA extensions can be very effective, manually creating ISA extensions gives even larger gains: Tensilica reports speedups of 40x to 300x for kernels such as FFT, AES and DES encryption [19][20][21].

Our work takes customizable processors, which are much less efficient than ASICs, and determines what is required to close that efficiency gap within a flexible framework. While previous studies have demonstrated significant improvements in performance and efficiency, 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, then customizes 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.

## 2.2 H.264 Algorithm and Computational Motifs

To understand how we customize a generic CMP to efficiently implement H.264, we must first understand the basic components of the H.264 algorithm. Five major functions comprise more than 99% of the total execution time in our base CMP implementation:

(i) IME: Integer Motion Estimation

(ii) FME: Fractional Motion Estimation

(iii) IP: Intra Prediction

(iv) DCT/Quant: Transform and Quantization and

(v) CABAC: Context Adaptive Binary Arithmetic Coding.

We implement the H.264 baseline profile at level 3.1; however, we use CABAC in place of CAVLC because CABAC is more complex and more challenging to improve [23][24]. CABAC is also more representative of advanced coding steps in other applications.

IME finds the closest match for an image-block from a previous reference image, and computes a vector to represent the observed motion. While it is one of the most compute intensive parts of the encoder, the basic algorithm lends itself well to data parallel architectures. When run 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 challenging 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 then uses previously encoded neighboring image-blocks within the current image to form a 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 quantized coefficients, which then go through the inverse quantization and inverse transform to generate the reconstructed pixels. The basic function is relatively simple and data parallel. However, it is invoked a number of times for each 16x16 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. This becomes particularly important because we need to speed up the application by around 250x on a four-processor system. After speedups in the first four functions, CABAC becomes the bottleneck and cannot be ignored.

## 2.3 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. Prior work has demonstrated efficient hardware architectures for various sub-algorithms in H.264 [33][34][35][36]. T.-C. Chen et al. implement a full-system H.264 encoder [4] and demonstrate that real-time HD H.264 encoding is possible in hardware using relatively low power and area cost. Later implementations employ clever algorithmic optimizations which sacrifice some signal-to-noise ratio (SNR) but significantly reduce energy and area [29][30]. While these optimizations are useful, our study works with the basic algorithms similar to those in [4]. Our aim is to understand the mechanisms behind high efficiency of custom hardware, and these insights are not likely to change significantly for a particular algorithmic variant.

There has also been H.264 software optimizations, particularly for motion estimation, which takes most of the encoding time. For example, sparse search techniques along with other algorithmic modifications speed up software performance of IME and FME by up to 10x with negligible loss in SNR [31] [32]. Combining aggressive algorithmic modifications with multiple cores and SSE extensions lead to highly optimized H.264 encoders on Intel processors [3][37].

Despite these optimizations, software implementations of H.264 lag far behind dedicated ASICs. Table 1 compares a software implementation of a 480p SD encoder [3] to a 720p HD ASIC implementation [4]. The software implementation employs a 2.8 GHz 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 to achieve real-time performance, but result in a 20% decrease in compression efficiency for a given SNR [3]. The custom ASIC hardware, on the other hand, consumes over 500x less energy and is far more efficient in its use of silicon area as shown by the area numbers in Table 1. The ASIC makes few algorithmic simplifications and consequently has a negligible drop in compression efficiency [4].

**Table 1. Intel's highly optimized, 2.8GHz Pentium 4 implementation of a 480p H.264 encoder versus a 720p HD ASIC. The second row presents Intel's SD data scaled to HD H.264. ASIC numbers have been scaled from 180nm to 90nm.**

| | Perf. (fps) | Area ($mm^2$) | Enrgy/frame (mJ) |
|---|---|---|---|
| Intel (720x480 SD) | 30 | 122 | 742 |
| Intel (1280x720 HD) | 11 | 122 | 2023 |
| ASIC | 30 | 8 | 4 |

# 3. EXPERIMENTAL METHODOLOGY

Our experiments use a CMP platform based on Tensilica's extensible RISC cores [2][39][40]. This baseline implementation defines the gap we seek to bridge between general-purpose computing and ASIC efficiencies. We use the extensible platform to implement three different classes of customizations, each more application specific than the previous one. We independently customize each processor's datapath using Tensilica's TIE language and optimize memory system parameters. To quickly simulate and evaluate different design options, we created a multiprocessor simulation framework that employs Tensilica's Xtensa Modeling Platform (XTMP) as its base. We use Tensilica's ISA extension framework to specify the number of VLIW slots, the width for the SIMD data paths, the number and size of register files, custom hardware instructions, and custom data storage elements. Tensilica's TIE compiler generates simulation models for different processor configurations and their energy explorer tool [22] estimates the energy and area of the resulting system. Its results are within 30% of the actual energy numbers [25], which is adequate since we are looking for more than two orders of magnitude improvements in energy efficiency.

## 3.1 Baseline H.264 Implementation

We use H.264 encoder reference code JM 8.6 for our experiments [38]. In the reference implementation, H.264's video encoding path is very long and suffers from sequential dependencies that restrict parallelism. We carefully analyze existing H.264 partitioning techniques and implement algorithmic changes in IME that remove some 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.

To build a base system, we map the four-stage macro-block partition of H.264 to a four-processor CMP system where each processor has 16KB 2-way set associative instruction and data caches. Table 2 presents our base system's performance and energy efficiency for the individual 720p HD H.264 sub-algorithms to highlight the large area and energy efficiency gap between our base CMP and the reference ASIC. At approximately 8.6B instructions to process one frame (IME), our base system consumes about 140 pJ/instruction—a reasonable value for a general-purpose system.

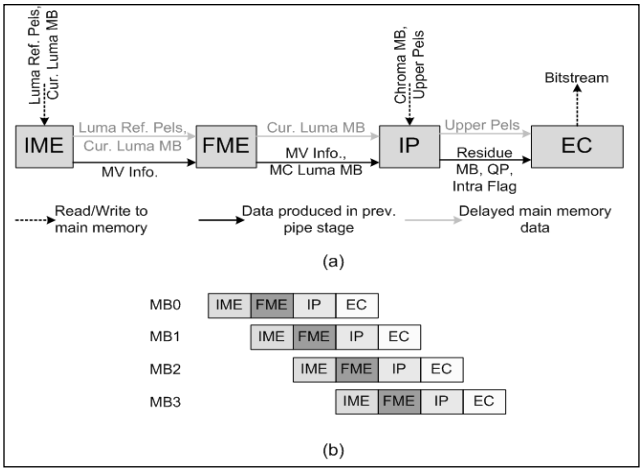


**Figure 1. Four stage macroblock partition of H.264. (a) Data flow between different pipeline stages. (b) How the four stage pipeline works on different macro blocks. The IP stage includes DCT+Quant. EC is the CABAC stage.**

**Table 2. Performance and energy for a generic Tensilica CMP implementation of H.264. Intra combines IP, DCT, and Quant. The gap numbers compare these values to an equivalent ASIC.**

| | Performance | | | | | |
|---|---|---|---|---|---|---|
| | MC/ MB | Frame /sec | Area ($mm^2$) | Energy/ Frame (mJ) | Perf. Gap | Energy Gap |
| **IME** | 2.10 | 0.06 | 1.04 | 1179 | 525.0x | 707x |
| **FME** | 1.36 | 0.08 | 1.04 | 921 | 342.0x | 468x |
| **Intra** | 0.25 | 0.48 | 1.04 | 137 | 63.0x | 157x |
| **CABAC** | 0.06 | 1.82 | 1.04 | 39 | 16.7x | 261x |

**Table 3. Datapath energy breakdown for our base implementation in mJ/frame. IF is instruction fetch/decode (including the I-cache). D-$ is the D-cache. Pip is the pipeline registers, buses, and clocking. Ctl is random control. RF is the register file. FU is the functional elements. Data estimates from processor simulations.**

| | IF | D-$ | Pip | Ctl | RF | FU | Total |
|---|---|---|---|---|---|---|---|
| **IME** | 410 | 218 | 257 | 113 | 113 | 68 | 1179 |
| **FME** | 286 | 196 | 205 | 90 | 90 | 54 | 921 |
| **Intra** | 54 | 20 | 29 | 13 | 13 | 8 | 137 |
| **CABAC** | 12 | 2 | 8 | 4 | 4 | 2 | 32 |
| **Total** | 762 | 436 | 499 | 220 | 220 | 132 | 2269 |

We analyze the performance and energy efficiency of this base CMP implementation and compare it to that of the ASIC. We allocate the processor's energy into different functional units as shown in Table 3, which reports the energy consumed by our base four-processor CMP system. As expected, the energy required for each task is related to the time required for that task, since the energy of each instruction is similar. The RISC implementations of IME and FME, which are the major contributors to performance and energy consumption, have a performance gap of 525x and an energy gap of over 700x with respect to the ASIC.

We also note that while IP, DCT, Quant and CABAC are much smaller parts of the total energy/delay, even they need about 100x energy improvements to reach ASIC-level values.

This data makes it clear how far we need to go to approach ASIC efficiency. Clearly, 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 2x. Even if we assume everything but the functional unit energy is overhead, we still end up with energy savings of only 20x—not nearly enough to reach ASIC levels. As the rest of this paper demonstrates, we need to both customize functional units (for correct bit widths, for efficient multi-input or output operations, etc.) and remove almost all other processor overheads (instruction fetches, register file accesses, etc.) to approach ASIC efficiency.

**Table 4. Different stages of specialization, and the types of optimizations implemented. Step 1 is very general; step 2 is often done in general-purpose SIMD units for important applications; step 3 builds application specific functional units.**

| | Step 1 | Step 2 | Step 3 |
|---|---|---|---|
| **Inst. decode logic** | App. class optimizations e.g. SIMD | App. class optimizations with custom fused instruction sub-graphs | Complex instrs performing multiple independent operations |
| **Register file** | App. specific register file size and width.<br>SIMD register file | Consume short-lived intermediate results without sending to register file | App. specific data storage structures and data supply networks |
| **Arithmetic datapath** | App. specific precision | Custom fused arithmetic operations | App. specific arithmetic blocks |

## 3.2 Customization Strategies

Table 4 defines three classes of processor customization. At the first stage we restrict ourselves to relatively general purpose datapath extensions such as SIMD and VLIW units; such extensions are frequently found in processor designs today and will be part of future efficient processors.

At the second stage, we add a limited degree of algorithm-specific customization. Operation fusion – the creation of new instructions that combine sequences of existing instructions – produces new functional units. We limit new instructions to operand requirements (i.e., two input operands, one output) that match those for existing instructions; new instructions must fit in existing instruction formats and datapath. This constraint is the same as that of Intel's SSE instructions. These customizations, at least for key functions, are also likely to exist in future processors.

Finally, at the third stage we allow unrestricted tailoring of the datapath according to algorithm needs by introducing arbitrary new compute operations as well as by complementing or even replacing the register files with custom storage structures. The results of these customizations shown in Figures 2, 3 and 4 are described in more detail in the next section.

# 4. RESULTS

We implement and evaluate the three-customization strategies of Table 4, detailing their effectiveness. For algorithm-specific

Power expenses:
x86 is spending most power on information fetching and decoding. (all the money going to the managers :P). x86 is a machine that fetches info and decodes them (power perspective).
GPUs: in multithreaded machines, use registers. With more threads, need more registers (one of the bounds on # of threads). Register file is huge. Hardware ('huge long wires'), hence costs energy to access the register file (so all power goes to reading and writing the register file). Operational frequency of access also higher due to multiple threads, as compared to x86.
(Now shuffling messages between departments is the money sink :P)
Dedicated unit for : Matrix factor multiply (FFT application on a single circuit board, faster implementation) -> lesser register file retrievals, less power usage. ASIC.
Things that benefit from hardware acceleration tend to be straightforward data patterns. Most problems are not a 100% that, making chip design challenging.
IF -> DECode -> ALU -> (MEM <-> D$) -> WB
-> special instruction 1 -> sp2 -> sp3

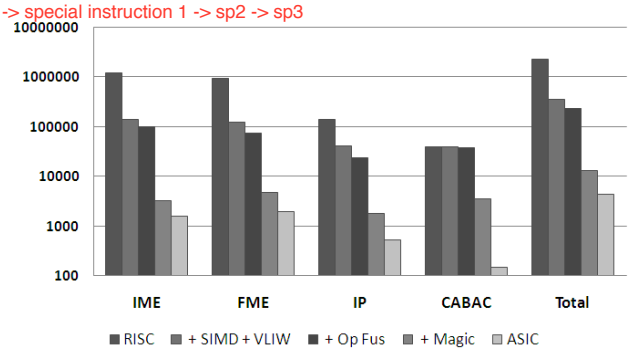


**Figure 2. Each set of bar graphs represents energy consumption (µJ) at each stage of optimization for IME, FME, IP and CABAC respectively. Each optimization builds on the ones in the previous stage with the first bar in each set representing RISC energy dissipation followed by generic optimizations such as SIMD and VLIW, operation fusion and ending with "magic" instructions**

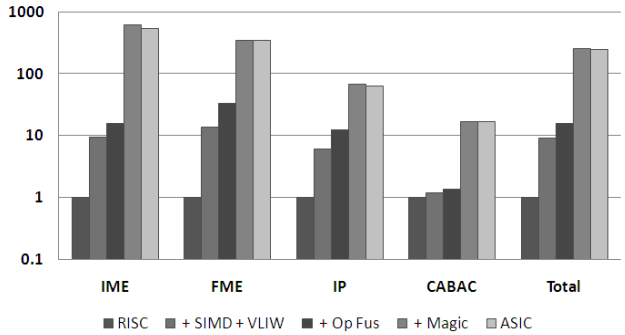


**Figure 3. Each set of bar graphs represents speedup at each stage of optimization. Each optimization builds on those of the previous stage with the first bar in each set representing RISC speedup, followed by generic optimizations such as SIMD and VLIW, then operation fusion and finally "magic" instructions**

instructions, we outline strategies for each major phase of computation. Collectively, these results describe how efficiencies improve by 170x over the baseline in Section 3.1.

## 4.1 SIMD and VLIW Enhancements

Using Tensilica's FLIX (Flexible Length Instruction eXtension) feature, we create processors with 2- and 3-slot VLIW instructions. Using TIE, we add SIMD execution units to the base processor with vector register files of custom depths and widths. As expected, DLP algorithms using SIMD units show a large decrease in processor energy; speedup increases as the number of instructions executed decreases. IME and FME use 16 and 18-way SIMD datapaths and achieve speedups of 10x and 14x. Intra/DCT/Quant using an 8-way SIMD datapath achieves a speedup of 6x. The SIMD units use custom-width functional units instead of standard 32-bit versions to enable more efficient computation, and generally run between 8 and 16 bits. As Figure 4 shows, even performing 16 concurrent operations barely increases the percentage energy used by the functional units, which still comprise around 10% of the total. Even the register file energy decreases by 4-6x using SIMD since we use 8-bit vector elements, and scale down register file depths, so its percentage contribution to the total energy does not increase considerably.

While SIMD only works for data-parallel algorithms, all H.264 sub-algorithms achieve speedups from VLIW instructions, with 2-slot VLIW offering higher energy efficiency than 3 slots. 2-slot VLIW gains up to 1.5x more performance. For CABAC, VLIW instructions increase the code size, and the resulting increase in cache size and cache access energy offsets any energy gains. SIMD and VLIW speed up the application by 10x, decreasing IF energy by 10x, but the percentage of energy going to IF does not change much. IF still consumes more energy than functional units. Furthermore, while CABAC is not initially an issue, its power dissipation is unchanged by these optimizations, and is now a major contributor to overall power dissipation.

## 4.2 Operation Fusion

The second customization strategy builds on the first and evaluates additional gains offered by the fusion of frequently occurring complex instruction subgraphs. Operation fusion is particularly interesting because it can be targeted by a number of automatic tools [22]. Fusion of complex subgraphs is useful because it reduces both instruction count and register file accesses—intermediate results are consumed within the fused operation and do not need to be stored in the register file. An additional benefit is the ability to create more energy efficient hardware implementations of the fused operations. For data parallel algorithms, we fuse together both RISC as well as SIMD operations. We pipeline our functional units to ensure fused operations do not increase clock cycle time.

To illustrate operation fusion, we present a pixel up-sampling example taken from FME:

$$x_n = x_{-2} - 5x_{-1} + 20x_0 + 20x_1 - 5x_2 + x_3$$

H.264 uses this equation to perform upsampling of pixels in the reference image frame. In the equation $x_n$ is the newly calculated up-sampled pixel, formed by applying an interpolation filter on pixels $x_{-2} \ldots x_3$ of the reference frame. Upsampling uses a major portion of FME compute time, so we want to enhance its performance and energy efficiency.

Before creating fused instructions, we split the equation into three parts based on computation similarities: $20x_0 + 20x_1$, $-5x_{-1} - 5x_2$, and $x_2 + x_3$. This allows us to keep the number of input operands per fused instruction equal to two and thus we do not increase the number of register file ports. Each instruction fuses addition/subtraction with multiplication, which is implemented using shift and adds. Figure 5 presents the newly created instructions.

Note that the two-input operand restriction is not broken because the accumulator register (acc), internal to the functional unit, is used implicitly. Similarly, the instruction supplies the constant multiplication factor directly, avoiding a register file access. These new instructions improve energy efficiency by reducing register file accesses by forwarding the result of the multiplication directly to an adder and by using an accumulator.

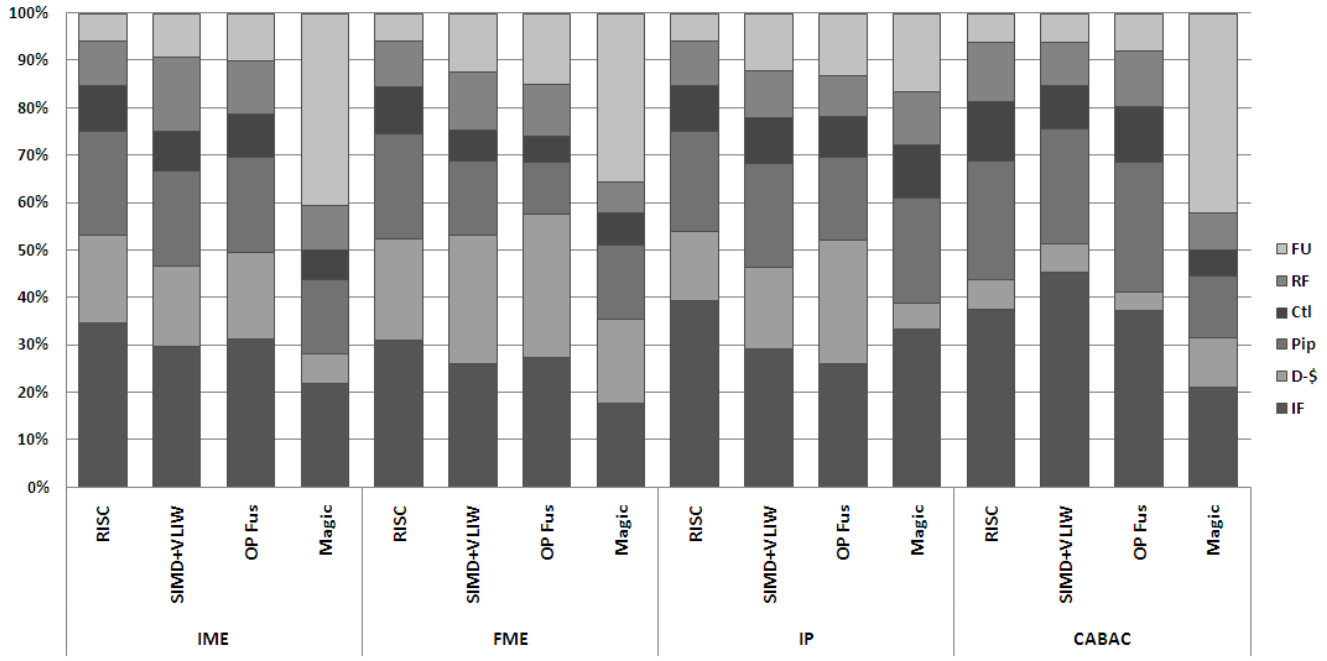


**Figure 4. Datapath energy breakdown for H.264. IF is instruction fetch/decode (including the I-cache). D-$ is the D-cache. Pip is the pipeline registers, busses, and clocking. Ctl is random control. RF is the register file. FU is the functional elements. Only the top bar (FU), or perhaps the top two (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. This data was estimated from processor simulations.**

```
acc = 0;
acc = AddShft(acc, x0, x1, 20);
acc = AddShft(acc, x-1, x2, -5);
acc = AddShft(acc, x-2, x3, 1);
xn = Sat(acc);
```

**Figure 5. FME upsampling after fusion of two multiplications and two additions. AddShft takes two inputs, multiplies both with the multiplicand and adds the result. Multiplication is performed using shifts and adds. Operation fusion results in 3 instructions instead of the RISC's 5 add/sub and 4 multiplication instructions.**

**Table 5. Fused operations added to each unit and the resulting performance and energy gains. FME required fusion of large subgraphs to get significant performance improvement.**

| | # of fused ops | Op Depth | Energy Gain | Perf Gain |
|---|---|---|---|---|
| **IME** | **4** | **3-5** | **1.5** | **1.6** |
| **FME** | **2** | **18-34** | **1.9** | **2.4** |
| **Intra** | **8** | **3-7** | **1.9** | **2.1** |
| **CABAC** | **5** | **3-7** | **1.1** | **1.1** |

Table 5 presents the number of fused operations created for each H.264 algorithm, the average size of the fused instruction subgraphs, and the total energy and performance gain achieved through fusion. Interestingly, IME and FME do not share any instructions, though Intra and FME share instructions for the Hadamard transform. DCT transform also implements the same transform instructions. CABAC's fused operations provide negligible performance and energy gains of 1.1x. Fused instructions give the largest advantage for FME, on average doubling the energy/performance advantage of SIMD/VLIW. Employing fused operations in combination with SIMD/VLIW results in an overall performance improvement of 15x for the H.264 encoder, and an energy efficiency gain of almost 10x, but still uses greater than 50x more energy than an ASIC.

The basic problem is clear. For H.264, the basic operations are very simple and low energy. In our base machine we over-estimate the energy consumed by the functional units, since we count the entire 32–wide functional unit energy. When we move to the SIMD machine, we tailor the functional unit to the desired width, which reduces the required energy. However, executing 10s of narrow width operations per instruction still leaves a machine that is spending 90% of its energy on overhead functions, with only 10% going to the functional units.

## 4.3 Algorithm Specific Instructions

To bridge the remaining gap, we must create instructions that can execute 100s of operations in a single instruction. To achieve this parallelism requires creating instructions that are tightly connected to custom data storage elements with algorithm-specific communication links to supply the large amounts of data required, and thus tend to be very closely tied to the specific algorithmic methods being optimized. These storage elements can then be directly wired to custom designed multiple input and possibly multiple output functional units, directly implementing the required communication for the function in hardware.

Once this hardware is in place, the machine can issue "magic" instructions that can accomplish large amounts of computation at very low costs. This type of structure eliminates almost all the

processor overheads for these functions by eliminating most of the communication overhead (register file, bus, and instruction fetch) associated with processors. We call these "magic" instructions, since these operations can have a large effect on both the energy and performance of an application and yet would be difficult to derive directly from the code. They typically require an understanding of the underlying algorithms and the capabilities and limitations of existing hardware resources, thus requiring greater effort on part of the designer. Since the IP stage uses some techniques similar to FME the rest of the section will focus on FME, IME and CABAC.

### 4.3.1 *FME Strategy*

To illustrate a "magic" instruction, we begin by returning to the pixel upsampling example. In H.264, upsampling uses an FIR filter that requires one new pixel per iteration. Thus after one upsampling step, we can reuse pixels $\mathbf{x_{-1}}$ … $\mathbf{x_3}$, and only need to load $\mathbf{x_4}$. Normal register files require us to do five register transfers for each upsampling step, significantly increasing the energy dissipated in the instruction fetch and decode logic and also in the register file. While some machines have indexing register files that help with this issue [6], we still need to read all the operations from the register file to perform the computation.

To reduce instruction fetches and register file transfers, we augment the processor register file with a custom 8-bit wide, six 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. Additionally, since all six entries can now be accessed in parallel we create a six input multiplier/adder which can be implemented much more efficiently (using carry-save addition) 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 (see **Figure** 6).

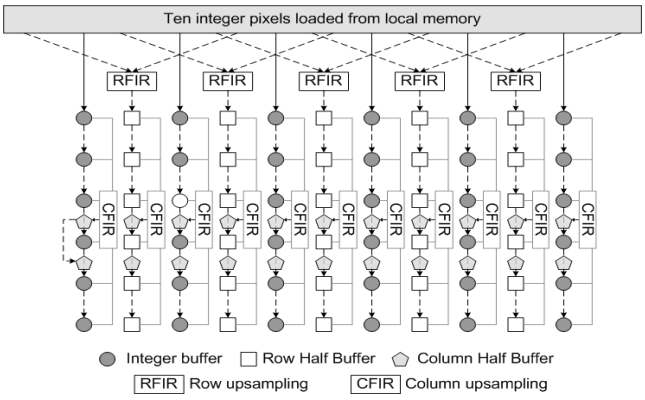


**Figure 6. FME upsampling unit, showing merged storage and computation. Customized shift registers directly wired to function logic result in efficient upsampling. Ten integer pixels from local memory are used for row upsampling in RFIR blocks. Half upsampled pixels along with appropriate integer pixels are loaded into shift registers. CFIR accesses six shift registers in each column simultaneously to perform column upsampling.**

This transformation yields large savings even beyond the savings in instruction fetch 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.

The FME SIMD code highlights the advantages of this approach over using 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 16x16 pixel blocks to 4x4 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. Next, FME fits a streaming data flow model where most of the intermediate data has a short life and is consumed by instructions that are only a few cycles behind; by storing such intermediate data in the register file, energy is wasted on unnecessary register file accesses. This intermediate data also leaves less space in the register file for non-intermediate data, resulting in additional loads and stores. Finally, not all computations are able to benefit from fusion because our register files can only supply two operands at a time.

To avoid these issues, our custom hardware upsampler processes 4x4 pixels. This allows us to reuse the same computation loop repeatedly without any code replication, 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 54x and processor fetch and decode energy by 14x. Finally, as Figure 4 shows, 35% of the energy is now going into the functional units.

### 4.3.2 *IME Strategy*

4x4 sum of absolute differences (SAD) calculations are important for IME. Figure 7 shows the custom datapath elements added to the IME processor to accelerate this function. The 16-way SIMD SAD unit of the fusion-optimized processor has been replaced by a 16x16 SAD unit, which can perform 256 SAD operations in one cycle. Since our standard vector register files cannot feed enough data to this unit per cycle, these registers have been replaced by state registers, which allow parallel access to all 16-pixel rows and enable this datapath to perform one 256-pixel computation per cycle. The fetch overhead of SAD operations is thus reduced by roughly 16x. Additionally, this custom storage structure has support for parallel shifts in all four directions, thus allowing much greater data reuse, and drastically reducing the cycles spent 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. More than 65% of total IME cycles are spent in overhead instructions. Thus, by reducing instruction overheads and by amortizing the remaining overheads over larger datapath widths, this strategy improves performance and energy efficiency by 20-30x.

The large number of parallel operations means that this functional unit finally consumes around 40% of the total instruction energy. This would be even higher, but we further reduced energy (approximately 30%) by employing reduced precision arithmetic where only 5 pixel-bits are used in distortion calculations instead of 8. This technique is also employed by our reference ASIC and

causes negligible drop in SNR [4]. These optimizations along with a small set of other custom operations enable the IME processor to match ASIC performance and come within 3x of ASIC energy

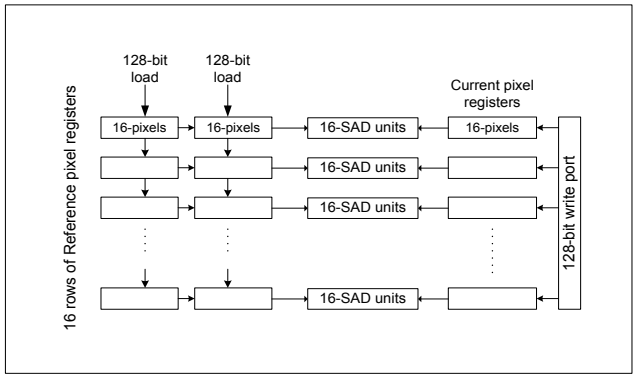


**Figure 7. Custom storage and compute blocks for IME's 4x4 SAD calculation. Current and reference-pixel register files allow parallel access to all pixel values to feed the 16x16 SAD array. In addition, the RefPixel Regfile supports operations to shift all pixel rows down by one row or shift all pixel columns right by one pixel location.**

### 4.3.3 *CABAC Strategy*

CABAC originally consumed less than 2% of the total energy. However, after adding "magic" instructions for data parallel components, CABAC dominates the total energy. However, it requires a different set of optimizations because it is highly 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 highly serialized process with small amounts of computation, but significant 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, which is written as a while loop over every bit of the **RANGE** in the reference code as shown in Figure 8. This loop (including the implicit doubly nested loops in **`put_one_bit_plus_outstanding`**) was reduced to a single constant time instruction and a rarely executed small while loop by fundamentally changing the algorithm as shown in Figure 9. Since we now do buffering on a 64-bit basis**, `word1full`** is rarely true, and **`wordsOustanding`** is almost never greater than 0.

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 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 [15]. CABAC is optimized to achieve the bit rate required for H.264 level 3.1 at 720p video resolution.

```
while (range < QUARTER) {
  if (low >= HALF) {
    put_one_bit_plus_outstanding(1);
    low -= HALF;
  } else if (low < QUARTER) {
    put_one_bit_plus_outstanding(0);
  } else {
    global_eep.Ebits_to_follow++;
    low -= QUARTER;
  }
  low <<= 1;
  range <<= 1;
}
```

**Figure 8. CABAC arithmetic encoding loop in H.264 reference code looping over every bit of the "RANGE."**

```
word1full = BIARI_ENCODE_PIPE_5();
if (word1full){
  wordsOutstanding = WRITE_OUT_WORD1();
  while(wordsOutstanding){
    wordsOutstanding =
      WRITE_OUT_UNRESOLVED();
  }
}
```

**Figure 9. CABAC arithmetic encoding loop after insertion of "magic" instructions. The loop corresponding to RANGE has been reduced to one single constant time instruction BIARI_ENCODE_PIPE_5.**

ASIC-like efficiency required 2-3 special hardware units for each sub-algorithm, which is significant customization work. After this effort, the processors optimized for data-parallel algorithms have a total speedup of up to 600x and an energy reduction of 60-350x compared to our base CMP. For CABAC total performance gain is 17x and energy gain is 8x. Figure 4 provides the final energy breakdowns.

**Table 6 - Area and area efficiency at various stages of customization**

| | Area ($mm^2$) | | | | | Speedup | Area Efficiency (Speedup/ Area) |
|---|---|---|---|---|---|---|---|
| | IME | FME | IP | CABAC | Total | | |
| RISC | 1.39 | 1.39 | 1.39 | 1.39 | 5.56 | 1 | 0.18 |
| RISC with Memory Cust. | 0.80 | 1.39 | 1.06 | 1.44 | 4.69 | 1 | 0.21 |
| GP Opt. | 1.79 | 4.12 | 1.76 | 1.55 | 9.22 | 9.2 | 1.00 |
| OP Fusion | 1.83 | 3.32 | 1.63 | 1.64 | 8.42 | 15.7 | 1.87 |
| Magic | 2.10 | 2.28 | 1.58 | 1.1 | 7.06 | 256 | 36.25 |
| ASIC @ 100MHz | 2.82 | 3.33 | 1.47 | 0.27 | 7.89 | 243 | 30.81 |
| ASIC @ 435MHz | 2.82 | 3.33 | 1.47 | 0.27 | 7.89 | 1057 | 133.97 |

## 4.4 Area Efficiency

Table 6 shows area in $mm^2$ for the evaluated optimization strategies. The last column shows the area efficiency for each step, which is defined as speedup/area. Customizing cache sizes to

the requirements of each algorithm results in substantial area savings as depicted by "RISC with Mem Cust". General-purpose optimizations increase the area substantially compared to vanilla RISC versions, but they also help improve the area efficiency for data-parallel algorithms. However, control-intensive CABAC does not benefit from such optimizations. Further customization of datapaths not only improves area efficiency tremendously but also results in a smaller area compared to general-purpose optimizations. Customizations not only reduce the number of instructions, but also substantially improve data reuse inside the processor, which in turn reduces cache sizes. This reduction in memory area helps offset area increases due to addition of custom units.

It might seem that the efficiency of our solution is higher than that of an ASIC, but the ASIC is designed to run at 100MHz in 0.18um while our magic version is designed to run at 435MHz. If we assume that the ASIC in 90nm can run at 435MHz without any modifications, it can achieve 4.35x better performance and thus 4.35x better area efficiency, making it substantially more area efficient than our solution.

## 4.5 Other Applications

While H.264 is representative of applications with very simple compute operations, other applications, for example floating point (FP) applications, have higher-energy operations. FP arithmetic consumes 10x the energy of integer arithmetic; FP functional units comprise a larger fraction of total instruction energy. Thus, one might think less parallelization is required to amortize instruction overheads for FP applications.

However FP operations comprise only 20% of the dynamic instruction stream for representative applications [26][27]. For this reason, FP energy will likely be a small fraction of total application energy. To match the most efficient H.264 design points, 35% or more of the total application energy should be in the ALU. Thus, with an instruction overhead of approximately 130pJ, functional unit energy will need to be at least 70 pJ, which is equivalent to 7 FP operations, or approximately 35 instructions (given a 20% FP instruction mix). While this level of parallelism might be possible for some applications with SIMD and operation fusion, it seems likely that customizations will be needed to achieve this number of ops/instruction for most applications. This is especially true if some part of the application is control and not data limited.

Finally, some applications are dominated by memory costs. In truly memory-bound applications, computation is not the bottleneck, so data path customizations will have little effect. For these applications, it is the energy efficiency of bringing application data to the core that fundamentally needs to be improved. Co-optimization of the memory system and the application can yield large savings in these situations [28], but the advantages of application customization over a conventional memory design with a few adjustable parameters still needs to be explored.

# 5. ENERGY EFFICIENT COMPUTERS

It is now easy to see how an ASIC can be 2-3 orders of magnitude lower energy than a processor. For many applications, and most of the ones performed by ASICs, the basic operations being performed are very low energy, using 8-16 bit integers like in H.264. These applications are computation—and not data fetch—limited, so the fundamental energy/operation bound is a couple hundred femtojoules in a 90nm process, which is equivalent to moving one bit less than a mm. All other costs in a processor—instruction fetch, register fetch, data fetch, control, and pipeline registers—are much larger (140pJ) and dominate overall power.

Standard SIMD and simple operation fusion 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. In addition, some of these overhead instructions are just to control or sequence the data (e.g., CABAC).

Thus, the solution is "instructions" that perform hundreds of operations each time they are executed, so the overhead of the instruction is better balanced by the work performed. Unfortunately this is hard to do in a general way, since bandwidth requirements and utilization of a larger SIMD array would be problematic. 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 are used for many different computations. In addition, since the intra-storage and functional unit communications were fixed and local, they could be managed at ASIC-like energy costs. 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 a create a processor which decreases the instruction and data fetch parts of a processor by more than 10x, these solutions will not be as efficient as solutions with "magic" instructions.

Of course including these "magic" instructions requires custom hardware, and some might 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.

For example, once the initial effort in understanding the application and its characteristics was done, the extensible processor allowed us to implement and verify the fully customized "magic" configuration for each algorithm in two to three man-months, which would not have been possible with an ASIC flow. 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. In fact, in this type of design environment, one might be tempted to make the new hardware that supports the "magic" instructions a little more flexible than required, providing some runtime flexibility just to increase the probability of it still being useful if the algorithm changes.

Yet an extensible processor 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 extensible full chip systems. We are currently working on this creating this type of system.

# 6. CONCLUSION

Ideally, we would like ASIC-like energy efficiencies—100x to 1000x more energy efficient than general-purpose CPUs—on our next generation processors. Our data, while not conclusive, indicates that this goal will be hard to achieve. The basic problem is that many applications include extremely simple, low energy operations. 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. 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 data parallel hardware like wide SIMD units will still leave you far from an ASIC.

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 and executed 100s of simple operations per instruction. 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. The key challenge now is to build a design system that lets application designers create these types of customizations with much greater ease.

# 7. 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.

# 8. REFERENCES

[1] Horowitz, M., Alon, E., Patil, D., Naffziger, S., Kumar, R., Bernstein, K., "Scaling, Power and the Future of CMOS," 20th Int'l Conference on VLSI Design, 2007, held jointly with 6th Int'l Conference on Embedded Systems, p. 23, 6-10 Jan. 2007.

[2] Solomatnikov, A, Firoozshahian, A., Qadeer,W., Shacham, O., Kelley, K., Asgar, Z., Wachs, M., Hameed, R., Horowitz, M., "Chip Multi-Processor Generator," Proceedings of the 44th Annual Design Automation Conference, 2007, pp. 262–263.

[3] Iverson, V., McVeigh, J., Reese, B., "Real-time H.264/avc Codec on Intel architectures," IEEE Int. Conf. Image Processing (ICIP'04), 2004.

[4] Chen, T.-C., et al., "Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder," Circuits and Systems for Video Technology, IEEE Transactions on, vol.16, no.6, pp. 673-688, June 2006.

[5] Davis, W.R., Zhang, N., Camera, K., Markovic, D., Smilkstein, T., Ammer, M.J., Yeo, E., Augsburger, S., Nikolic, B., Brodersen, R.W., "A design environment for high-throughput low-power dedicated signal processing systems," Solid-State Circuits, IEEE Journal of , vol.37, no.3, pp.420-431, Mar 2002.

[6] Balfour, J., Dally, W.J., Black-Schaffer, D., Parikh, V., Park, J.S., "An Energy-Efficient Processor Architecture for Embedded Systems," Computer Architecture Letters, vol.7, no.1, pp.29-32, January-June 2007.

[7] McCloud, S., "Catapult C Synthesis-Based Design Flow: Speeding Implementation and Increasing Flexibility," Mentor Graphics Technical Library, http://www.techonline.com/electronics_directory/techpaper/193102520, August 2004.

[8] Kathail,V., "Creating power-efficient application engines for SoC design," Synfora, Inc. SoC Central, Feb 1, 2005.

[9] Rowen, C., Leibson, S., "Flexible architectures for engineering successful SOCs," Design Automation Conf, 2004. Proceedings. 41st , pp. 692-697, 2004.

[10] Woh, M., Seo, S., Mahlke, S., Mudge, T., Chakrabarti, C., and Flautner, K., "AnySP: anytime anywhere anyway signal processing," SIGARCH Comp. Arch. News 37, 3 (Jun. 2009), 128-139.

[11] Intel Corp., "Motion Estimation with Intel® Streaming SIMD Extensions 4 (Intel® SSE4)" [Online]. Available: http://software.intel.com/en-us/articles/motion-estimation-with-intel-streaming-simd-extensions-4-intel-sse4/.

[12] Intel Corp., "Intel SSE4 Programming Reference" [Online]. Available: http://softwarecommunity.intel.com/isn/Downloads/Intel%20SSE4%20Programming%20Reference.pdf.

[13] Clark, N.T., Zhong, H., Mahlke, S.A., "Automated custom instruction generation for domain-specific processor acceleration," Computers, IEEE Transactions on, vol.54, no.10, pp. 1258-1270, Oct. 2005.

[14] Cong, J., Fan, Y., Han, G., and Zhang, Z., "Application-specific instruction generation for configurable processor architectures," Proceedings of the 2004 ACM/SIGDA 12th international Symposium on Field Programmable Gate Arrays (Monterey, California, USA, February 22 - 24, 2004). FPGA '04. ACM, New York, NY, 183-189.

[15] Shojania, H., Sudharsanan, S., "A VLSI Architecture for High-Performance CABAC Encoding," Visual Communications and Image Processing (SPIE), 2005. Proceedings. vol. 5960, June 2005.

[16] Ienne, P., Leupers, R., "Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon)," Morgan Kaufmann Publishers Inc. 2006.

[17] Tensilica Inc., "Xtensa LX2 Benchmarks" [Online]. Available: http://www.tensilica.com/products/xtensa-customizable/xtensa-lx2/benchmarks.htm.

[18] Tensilica Inc., "The What, Why, and How of Configurable Processors" [Online]. Available: http://www.tensilica.com/products/literature-docs/white-papers/configurable-processors.htm.

[19] Tensilica Inc., "Implementing the Advanced Encryption Standard on Xtensa® Processors", Application note. [Online]. Available: http://www.tensilica.com/products/literature-docs/application-notes/tie-application-notes/advanced-encryption-standard.htm.

[20] Tensilica Inc., "Implementing the Fast Fourier Transform (FFT)", Application note. [Online]. Available: http://www.tensilica.com/products/literature-docs/application-notes/tie-application-notes/fast-fourier-transform-fft.htm.

[21] Tensilica Inc., "Xtensa Processor Extensions for Data Encryption Standard (DES)" Application note. [Online]. Available: http://www.tensilica.com/products/literature-docs/application-notes/tie-application-notes/data-encryption-extensions.htm.

[22] Tensilica Inc., "How to Minimize Energy Consumption while Maximizing ASIC and SOC Performance" White Paper. Available: http://www.tensilica.com/uploads/white_papers/Xenergy_Tensilica.pdf.

[23] Weigand, T., Sullivan, G., Bjontegaard, G., Luthra, A., "Overview of the H.264/AVC Coding Standard," IEEE Transactions on Circuits and Systems for Video Technology, vol 13, no 7, July 2003.

[24] Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, Joint Video Team, ITU-T Recommendation H.264 and ISO/IEC 14496-10 AVC, May 2003.

[25] Tensilica Inc., "Xtensa Energy Estimator (Xenergy) – User's Guide".

[26] Cheveresan, R., Ramsay, M., Feucht, C., Sharapov, I., "Characteristics of workloads used in high performance and technical computing," Proceedings of the 21st annual international conference on Supercomputing, June 17-21, 2007.

[27] Rupnow, K., Rodrigues A., Underwood, K., Compton, K., "Scientific applications vs. SPEC-FP: a comparison of program behavior," Proceedings of the 20th annual international conference on Supercomputing, June 28-July 01, 2006.

[28] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K. and Demmel, J., "Optimization of sparse matrix-vector multiplication on emerging multicore platforms", Proceedings of the 2007 ACM/IEEE conference on Supercomputing, ACM New York, NY, USA, 2007.

[29] Lin, Y.-K., Li, D.-W., Lin, C.-C., Kuo, T.-Y., Wu, S.-J., Tai, W.-C., Chang, W.-C. and Chang, T.-S., "A 242mw 10mm$^2$ 1080p H.264/AVC high profile encoder chip," Proceedings of the 45th Design Automation Conference, 2008.

[30] Chang, H.-C., Chen, J.-W., Su, C.-L., Yang, Y.-C., Li, Y., Chang, C.-H., Chen, Z.-M., Yang, W.-S., Lin, C.-C., Chen, C.-W., Wang, J.-S. and Guo, J.-I., "A 7mw-to-183mw dynamic quality-scalable h.264 video encoder chip," Proceedings of 2007 IEEE ISSCC Dig. Tech. Papers.

[31] ISO/IEC MPEG & ITU-T VCEG, "Fast Integer Pel and Fractional Pel Motion Estimation for JVT," JVT-F017, 2002.

[32] Yin, P, Tourapis, H.-Y. C., Tourapis, A. M., and Boyce, J., "Fast mode decision and motion estimation for JVT/H.264," Proceedings of IEEE International Conference on Image Processing, 2003.

[33] Chen, C.-Y., Chien, S.-Y., Huang, Y.-W., Chen, T.-C., Wang, T. C. and Chen, L.-Y., "Analysis and Architecture Design of Variable Block-Size Motion Estimation for H.264/AVC," IEEE Transactions on Circuits and Systems, 2006.

[34] Li, S., Wei, X., Ikenaga, T. and Goto, S., "A VLSI architecture design of an edge based fast intra prediction mode decision algorithm for h.264/avc," Proceedings of the 17th ACM Great Lakes Symposium on VLSI.

[35] Chen, T-C., Huang, Y.-W. and Chen, L.-G., "Fully Utilized And Reusable Architecture For Fractional Motion Estimation Of H.264/Avc," Proceedings of IEEE International Conference On Acoustics Speech And Signal Processing, 2004.

[36] Osorio, R. R., Bruguera, J. D., "High-Throughput Architecture for H.264/AVC CABAC Compression System," IEEE Transactions on Circuits and Systems for Video Technology, 2006.

[37] Chen, Y.-K., Li, E. Q., Zhou, X. and Ge, S., "Implementation of H.264 encoder and decoder on personal computers," Journal of Visual Communication and Image Representation, April 2006.

[38] Joint Video Team Reference Software JM8.6, ITU-T.

[39] Firoozshahian, A., Solomatnikov, A., Shacham, O., Asgar, Z., Richardson, S., Kozyrakis, C., Horowitz, M., "A memory system design framework: creating smart memories," Proceedings of the 36th annual international symposium on Computer architecture, 2009.

[40] Solomatnikov, A., Firoozshahian, A. Shacham, O., Asgar, Z., Wachs, M, Qadeer, W, Richardson, S. and Horowitz, M., "Using a Configurable Processor Generator for Computer Architecture Prototyping," Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009.