Fast and Energy-Efficient Digital Filters for Signal Conditioning in Low-Power Microcontrollers

Carlos Moreno
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
cmoreno@uwaterloo.ca

Sebastian Fischmeister
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
sfischme@uwaterloo.ca

ABSTRACT

Embedded systems often use digital filtering after analog-to-digital conversion when signal conditioning is required. However, digital filters are computationally demanding, making them unsuitable for low-power microcontrollers.

In this paper, we propose a fast and energy-efficient digital filtering technique based on look-up tables. The novelty in our technique is the use of multiple small LUTs, with a novel way to combine them to reduce the quantization error. As an additional novelty element, we propose a new *input-centric* efficient implementation, specific to LUT-based software filters. Experimental results confirm the technique's remarkable potential for speed as well as its energy efficiency.

1 Introduction

Embedded systems applications such as control and monitoring may benefit from signal conditioning to remove unwanted components. To this end, digital filters may be used after capturing analog signals into a discrete-time quantized form (a *digital signal*) [9]. A typical application is signal smoothing — removing fast-changing components of the signal, usually corresponding to noise, while maintaining the underlying information in the signal [7].

The Savitzky-Golay method [10] formulates this smoothing technique in terms of polynomial approximation in the least-square sense; the digital signal processing (DSP) community formulates the technique in terms of Fourier (spectral) analysis and processing of signals. For the case of finite impulse response (FIR) filters (which is the focus in this work), there is no difference in the implementation between Savitzky-Golay filters and FIR filters in the DSP sense.

FIR filters require a multiply-and-accumulate structure involving real values, making them computationally demanding and thus unsuitable for software implementation on low-power microcontrollers (MCUs). Since the signals are in general sequences of integer values, the use of look-up tables (LUTs) can reduce the computational requirements by replacing multiplications by a real value with a LUT access. However, the size of this LUT grows exponentially with the bitlength of the signal samples; for example, for a 16-bit digital signal, these LUTs would have 2^{16} elements of 16 bits each, requiring 128 kbytes per LUT.

This basic idea of replacing each multiplication by a LUT access has been used for simple cases such as data of 8-bit or less (LUT with 256 elements or less), mostly for hardware applications [8]. Distributed arithmetic has also been proposed [11], making use of LUTs. However, this technique is unsuitable for software implementations (in low-power processors) and is typically used in hardware implementations, as it relies on parallelism to achieve high performance.

Our Contributions

In this paper, we propose a fast and energy-efficient digital filter software implementation technique based on LUTs. The technique is practical and suitable for low-power microcontrollers. The novelty in our technique is the use of multiple small LUTs; in particular, a novel way to use these multiple LUTs to reduce the quantization error in the result. Moreover, our technique can be used for either signed or unsigned data and coefficients. As an additional novelty element, we propose a new *input-centric* efficient implementation strategy, specific to LUT-based filtering in software.

We include experimental data confirming the remarkable potential for speed of this technique. We also include energy consumption measurements that confirm the energy-efficiency of our technique.

Organization of the Paper

The remainder of the paper proceeds as follows: Section 2 describes FIR digital filters and related assumptions for this work. Our proposed technique is presented in Section 3; Section 4 describes our input-centric implementation strategy. Experimental results are shown in Section 5, including a case-study in Section 6, and Section 7 concludes the paper.

2 Background – Digital Filters

Let $x(n)$ be a discrete-time signal, where $n \in \mathbb{Z}$ is the time index, and let $h(n)$ be a digital filter defined as the sequence of filter coefficients, corresponding to the filter's impulse response. In the most general case, h is an infinite sequence. However, in this work we focus on FIR filters, for which h is a finite-length sequence. Furthermore, in practical embedded systems applications, $h(n) = 0 \; \forall \; n < 0$, since values at negative indices correspond to a system responding before

DAC '17, June 18-22, 2017, Austin, TX, USA
© 2017 ACM. ISBN 978-1-4503-4927-7/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062245

the input has been applied. Let $y(n)$ be the output signal (the result of applying the filter h to the signal x), and let L be the length of the filter. Then:

$$y(n) = \sum_{k=0}^{L-1} h(k)x(n-k) \tag{1}$$

In embedded systems applications, $x(n)$ and $y(n)$ are typically represented as integer values. However, in most cases the values of $h(n)$ are real values. Thus, implementation of Equation (1) in principle requires floating-point operations,[1] making the task computationally demanding and unsuitable for implementations on low-power MCUs.

Assumptions

This work relies on the following assumptions, which are reasonable in the context of signal conditioning for embedded applications:

- **Filters are known at compile-time:** for a given filter, the set of coefficients $h(n)$ is fixed.
- **Coefficients' magnitude:** Without loss of generality, we will assume that for all the filter coefficients, $|\,h(n)\,| < 1$ — we can always scale the coefficients and apply the inverse scaling factor to the final result. Moreover, for practical signal conditioning applications, assuming amplification or attenuation of the signal is not required, it must hold that $\sum_k h(k) = 1$. Though this does not strictly imply that $|\,h(n)\,| < 1$ for each n, this is virtually always the case.

3 Our Proposed Technique

Our proposed technique is centered around the idea of replacing multiplications by a real value with LUT accesses. Since the filter coefficients are fixed, in the most basic form of the technique we define one LUT for each distinct coefficient; the value of $x(n)$ is used as subscript and the output is the precomputed value of the subscript multiplied by the coefficient.

With this mechanism, we trade storage space — an increasingly accessible commodity even in low-power MCUs — for reduced computations. Thus, we also trade space for energy consumption, since reduced computations translate into reduced energy consumption.

The novelty in our proposed technique is to extend the principle to larger bit-widths such as 12-bit or 16-bit without having to resort to larger LUTs (8 kbytes or 128 kbytes, respectively), and in a way that minimizes the quantization error. This is achieved through several novel ways of splitting the operation through multiple small LUTs. Though [6] suggests the basic idea of splitting the operations through multiple LUT accesses to reduce the size of the LUT, they do so with a single LUT by splitting the operand into blocks of contiguous bits. This approach has the disadvantage that it disregards the issue of quantization error, making their idea and implementation unsuitable for practical applications.

The next sections show the details of our proposed techniques for the various relevant scenarios.

[1] Fixed-point arithmetic can also be used, but it is still too computationally demanding for low-power MCUs.

3.1 Split-LUT Multiplication – Unsigned Values

Let us consider the general case of an N-bit input value x. We first consider the unsigned case ($0 \leqslant x < 2^N$). This value can be written in terms of two $N/2$-bit values, $x_{_L}$ and $x_{_H}$. For notational convenience, we define

$$B \triangleq 2^{\frac{N}{2}} \tag{2}$$

Thus,

$$x \;=\; B\,x_{_H} \;+\; x_{_L} \tag{3}$$

Multiplication by a fixed value a (with $0 < a < 1$; we will consider the case $a < 0$ in Section 3.3) results in the N-bit value ax as shown below:

$$ax \;=\; B\,ax_{_H} \;+\; ax_{_L} \tag{4}$$

Since $x_{_H}$ and $x_{_L}$ are both $N/2$-bit values, then $ax_{_H}$ and $ax_{_L}$ can be obtained through LUT-based multiplication with a $2^{N/2}$ elements LUT. Since $a < 1$, then the LUT elements are $N/2$-bit values. In this case, recombining these two values to obtain the result as per Equation (4) is essentially a free operation, as it simply means concatenating the two $N/2$-bit values to form the N-bit result.

Reducing the Quantization Error

The above describes the underlying idea in its most basic form, which is not practical given the large quantization error it exhibits. Let $\widehat{ax_{_H}}$ and $\widehat{ax_{_L}}$ be the approximated values that result from using the LUT (which contains the precomputed results rounded to the nearest integer). Then:

$$|\,\widehat{ax_{_H}} - ax_{_H}\,| < 0.5 \tag{5}$$
$$|\,\widehat{ax_{_L}} - ax_{_L}\,| < 0.5 \tag{6}$$

Since $\widehat{ax} = B\,|\,\widehat{ax_{_H}}\,| + |\,\widehat{ax_{_L}}\,|$, the error in the final result is

$$|\,\widehat{ax} - ax\,| \;<\; B\,|\,\widehat{ax_{_H}} - ax_{_H}\,| \;+\; |\,\widehat{ax_{_L}} - ax_{_L}\,| \tag{7}$$

From Equations (5) and (6), we obtain

$$|\,\widehat{ax} - ax\,| \;<\; \frac{B}{2} + 0.5 \tag{8}$$

As an example, for $N = 16$ bits, the error could be as large as 128.5.

To address this issue, we use two independent LUTs — a Low LUT and a High LUT: instead of computing the high part of the result, $B\,ax_{_H}$, as the $N/2$-bit output of the LUT shifted $N/2$ positions to the left, we setup a LUT for the N-bit value of $B\,ax_{_H}$. Notice that the size of this additional LUT is still $2^{N/2}$ elements, since $x_{_H}$ (an $N/2$-bit value) is used as its input. With this, we limit the overall error in the result to ± 1, since $B\,ax_{_H}$ is computed through a LUT and thus its maximum error magnitude is 0.5:

$$|\,\widehat{ax} - ax\,| < \left|\,\widehat{B\,ax_{_H}} - B\,ax_{_H}\,\right| + |\,\widehat{ax_{_L}} - ax_{_L}\,| \;<\; 1 \tag{9}$$

The space requirements are small compared to a straightforward single-LUT implementation: for 16-bit results, we require $256 \,+\, 2 \times 256 = 768$ bytes, instead of the 128 kbytes that a LUT for 16-bit inputs and outputs would require.

3.2 Split-LUT Multiplication – Signed Values

Signed values introduce a difficulty in that a value $-x$ (with $0 < x \leqslant 2^{N-1}$, where N is the number of bits) is represented by the unsigned N-bit value $x' = 2^N - x$. The decomposition of x' into two components x'_H and x'_L of $N/2$ bits each is not consistent with the decomposition of x into x_H and x_L.

The key observation is that we can perform arithmetic with the unsigned value x' to obtain ax' (in this scenario as well, we first consider the case $a > 0$) and adjust the result to obtain the signed representation of $-ax$:

$$ax' = a(2^N - x) = a2^N - ax \; = \; 2^N - ax + (a-1)2^N$$

$$\Rightarrow \quad 2^N - ax \; = \; ax' + (1-a)2^N \qquad (10)$$

The term $2^N - ax$ represents the negative value $-ax$, which corresponds to the result. Since $(1-a)2^N \equiv -a2^N$ in modulo 2^N arithmetic, it suffices to subtract $a2^N$ from the result using unsigned integer arithmetic of N bits. For example, in the case of 16-bit values in C, this happens automatically if we convert to `uint16_t` to enable modulo 2^{16} arithmetic. $a2^N$ can be precomputed and conditionally subtracted from the result for negative input values. This adjustment can be made part of the N-bit LUT (the one used for x_H) by conditionally subtracting during construction of the LUT, thus avoiding conditional statements at run time.

3.3 Split-LUT Multiplication – Negative Coefficients

Negative values of a introduce an additional difficulty. For a non-negative input value $x = B\,x_H + x_L$, we have:

$$ax \; = \; -\,|\,a\,|\,x \; = \; -\,|\,a\,|\,B\,x_H - |\,a\,|\,x_L \qquad (11)$$

Precomputing the N-bit value $-\,|\,a\,|\,B\,x_H$ is not a problem: we just obtain the actual value of this expression and add 2^N to obtain the signed representation of that negative value. However, the term $-\,|\,a\,|\,x_L$ is a problem if we want to use a LUT with $N/2$-bit elements: representation of negative values at $N/2$ bits is not directly compatible and would require a conditional sign extension at run time.

The key observation in this case is that the multiplication procedure may be different for the cases where $a < 0$: the LUT corresponding to the low part of the input value outputs the (non-negative) value of $|\,a\,|\,x_L$, and the result is obtained as the difference between the two LUTs (High LUT − Low LUT). We observe that the choice (adding the outputs of the two LUTs for $a > 0$ vs. subtracting when $a < 0$) is coded in the operation before compiling the program, since the value of a is known; at run time, no conditional statements are required.

Negative input values are not a problem, as they are handled in the exact same manner as they are handled with $a > 0$: the value $-x$ is represented as $x' = 2^N - x$; we compute the term ax' through the LUT, considering that $a < 0$, and apply the same adjustment of conditionally subtracting $a2^N$ for negative input values.

In all cases, we can reduce the rounding error by setting up LUTs of a size larger than the operand size that include a scaling factor. For example, 12-bit operations can be implemented with a scaling factor of $2^4 = 16$ such that the result is a 16-bit value. The error will be within ± 1 for the 16-bit value, and "disappears" (for most output samples) when scaling back down to 12-bits — an inexpensive operation, since we only have to right-shift.

3.4 Higher Number of Look-Up Tables

Splitting into two LUTs is perhaps the more reasonable design, given the typical operand sizes used in low-power microcontroller applications: additional LUTs involve additional computations, which sacrifices both speed and energy efficiency. Moreover, with three LUTs the error can be up to ± 1.5, instead of ± 1 with two LUTs.

However, the principles outlined for two LUTs are easily extended to three or more LUTs. For example, with 12-bit signals, we could have three LUTs with 4-bit inputs; if x_L, x_M, x_H denote the low, middle, and high 4-bit chunks of the operand (respectively), then the low LUT would have 4-bit elements and outputs the value of ax_L, the middle LUT, with 8-bit elements, outputs the value of $16 \cdot ax_M$, and the high LUT would have 12-bit elements with the value of $256 \cdot ax_H$ including adjustments to account for signed operands and negative coefficients, as described in the previous sections. This configuration requires only 64 bytes, instead of 192 bytes for a dual LUT with 6-bit inputs.

Though the experimental evaluation and the case-study focus on dual LUT setups, the distributed source code and demos [4] include an example of 12-bit input values implemented as three LUTs with 4-bit inputs.

4 Input-centric vs. Output-centric Computations

Equation (1) derives from the linear time-invariance (LTI) of the system: the output is a superposition of impulse responses, h, weighted and positioned according to each input sample, as illustrated in Figure 1.

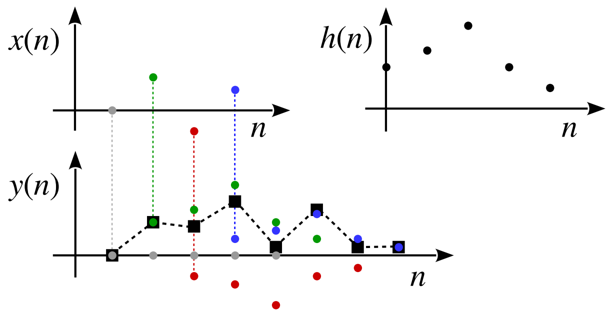

Figure 1: Superposition of Impulse Responses.

This view of the digital filter's behavior can lead to a substantially more efficient implementation when using our LUT-based approach. The straightforward approach of using Equation (1) to compute output samples when needed requires accessing values from different LUTs; this involves re-loading addresses for each input sample.

Instead of setting up a separate LUT for each distinct coefficient value, we can store complete sequences of weighted impulse responses: the sequence $1 \cdot h(0), 1 \cdot h(1), 1 \cdot h(2) \cdots 1 \cdot h(L-1)$ is stored in contiguous memory locations; followed by the sequence $2 \cdot h(0), 2 \cdot h(1) \cdots 2 \cdot h(L-1)$, and so on, until $255 \cdot h(0), 255 \cdot h(1) \cdots 255 \cdot h(L-1)$. In a sense, we can view this as a single LUT where each element is a weighted impulse response and the subscript is the value of the input (representing the weighting for the impulse response).[2]

The computational advantage is that with virtually all architectures and ISAs, we can efficiently access all the elements of h in sequence. Thus, we make the processing *input-centric*: instead of computing each output as a linear combination of the past L input samples, we process each input sample by superposing the weighted impulse response to the future L output samples. At time index k, after processing input sample $x(k)$, the value of $y(k)$ is already the correct result, since future input samples no longer have an effect on that output sample. The output can be efficiently represented by a circular buffer with length given by the smallest $2^m \ : \ 2^m \geqslant L$. Thus, instead of computing each output sample as follows (example in C-like pseudocode with 16-bit data),

```
y[n] = LUT_h0_high[high(x[n])] + LUT_h0_low[low(x[n])]
  + LUT_h1_high[high(x[n-1])] + LUT_h1_low[low(x[n-1])]
  + LUT_h2_high[high(x[n-2])] + LUT_h2_low[low(x[n-2])]
  + ...
```

we process each input sample by doing the following:

```
uint16_t * out = &y[n];
uint16_t * h_high = &LUT_h_high[L * high(x[n])];
uint8_t  * h_low = &LUT_h_low[L * low(x[n])];

*out   += *h_high++;
*out++ += *h_low++;
*out   += *h_high++;
*out++ += *h_low++;
.... (repeated L times)
```

For the negative coefficients, the `+=` operator is replaced by a `-=` operator for the `h_low` pointer. The above (pseudo)code assumes that future values of `y` have been initialized to zero. It also assumes a linear output array — a circular buffer would not allow the simple `*out++` expression, though it can still be coded efficiently. As an example, in AVR 8-bit assembly code [2], using the `avr-gcc` tool chain, we can implement each operation (each of the L pairs above) as follows, with an output circular buffer of size 16 (32 bytes):

```
; Output buffer declared in the main program (in C) as:
;     uint16_t y[16] __attribute__((aligned(256)));

; Based on received parameters:
;   Load X (registers r27:r26) with the value of h_low
;   Load Y (registers r29:r28) with the value of h_high
;   Load Z (registers r31:r30) with the value of out

    ld r24, Z
    ldd r25, Z+1
    ld r18, Y+
    ld r19, Y+
    add r24, r18
    adc r25, r19
    ld r18, X+
    sub r24, r18
    sbc r25, __zero_reg__
    st Z+, r24
    st Z+, r25
    andi r30, 0x1F  ; output buffer is 256-byte aligned
```

[2] To simplify the explanation, we describe the scheme with a single LUT; in practice, the layout would apply to both low and high LUTs.

The above assumes that aligning to 256-byte multiples does not sacrifice storage. If this was not the case and storage space was critical, we could always align to 32-byte multiples, avoid the auto-increment of `Z` and code the increment as:

```
r30 ← (r30 & 0xE0) | ((r30 + 1) & 0x1F)
```

FIR filters are often designed with a symmetric impulse response, since this produces filters with linear phase characteristics [9]. In these cases, the input-centric method can benefit from further space savings by storing only half the impulse response; when processing an input sample, the pointer to the output signal is incremented L times, whereas the pointers to the LUTs are incremented until reaching the symmetry point of the impulse response, and from that point they reverse direction, being decremented to traverse the impulse response samples back to the beginning, resulting in a symmetric sequence of values corresponding to the complete impulse response. Notice that this maintains the computational and energy efficiency while offering savings in storage.

5 Experimental Results

Our experimental setup was designed to demonstrate the two main advantages of our proposed technique; namely, execution speed and energy efficiency.

5.1 Execution Speed

For this evaluation, we implemented what we believe is a dramatic demonstration, even if it is not an application that one would do in practice: we implemented *CD-quality audio* filtering on an 8-bit MCU — an AVR Atxmega 128A1U running at 32 MHz [3]. That is, we were able to process stereo audio data (2 × 16-bits at 44100 samples per second — 88200 filtering operations per second) in real-time; with the most efficient implementation, this took about 73% of the available CPU time. We implemented an 11-tap ($L = 11$) low-pass filter with a 40 dB attenuation for the stop-band. Table 1 shows the execution times to process one second of audio. Since execution takes a fixed amount of clock cycles, (it has no conditional statements or any input data dependencies), and the resolution in the pin-toggling measurement is in the order of one clock period ($1/32\,\mu$s), we do not report confidence intervals or accuracy figures.

Implementation	**Execution Time (ms)**
Existing Approaches:	
Non-LUT Floating-point	8858.2
Non-LUT Integer Arithmetic	1536.1
This Work:	
Standard (output-centric) LUT	926.76
Input-centric – C	1130.6
Input-centric – ASM	730.85

Table 1: Execution Time to Process One Second of Audio.

By inspecting the assembly code generated by the compiler (`avr-gcc`), we observed several inefficiencies for the input-centric implementation; especially in the way the circular buffer was handled. For this reason, we implemented the assembler version, as briefly sketched in Section 4. This was not the case for the standard LUT implementation: the compiler-generated code was efficient, which is the reason why it is faster than the input-centric C implementation. Moreover, as we did not find any important opportunities for optimization, we did not include an assembler implementation for the standard LUT technique in our experimental evaluation.

The integer arithmetic implementation replaces the multiplications by a coefficient $a < 1$ with an integer multiplication by $2^{16}\,a$; the intermediate results are all done in 32-bit arithmetic, and the final result is scaled back to the $[0, 2^{16})$ range.

We emphasize the aspect that in practice one would choose a specialized architecture for processing audio data; however, being able to filter audio data with a low-power MCU is quite remarkable — digital filters for real-time audio data are usually left not only to processors with far more computing power and faster clock speeds, but typically for specialized DSP architectures. For embedded systems applications, often with 12-bit data and sampling rate no more than a few kilohertz, the results allow us to conclude that signal conditioning through digital filtering can be done with low computational cost.

In terms of comparing our technique against prior work, we observe that the non-LUT implementations constitute the state-of-the-art, since our technique applies in the context of low-power microcontrollers. We do not compare our technique against implementations on DSP chips or architectures with advanced hardware-assisted arithmetic. Also, we do not compare our technique against the straightforward single-LUT approach; with that technique, processing 16-bit signals would require a prohibitively high amount of memory for the LUT, making any comparison meaningless.

5.2 Energy Efficiency

We also determined the total amount of energy consumed by the MCU to perform filtering. To this end, we used a small current-sensing shunt resistor at the Power-In pin of the MCU. We fed the voltage difference (proportional to power consumption) to an analog integrator to determine the total amount of energy. To integrate the signal at the analog level, we simply charge a capacitor with a current proportional to the input and measure the voltage increase in the capacitor over the interval of execution. This has the advantage that it does not require sampling the power consumption signal at an extremely high rate (which would be necessary if we numerically integrate the digital signal).

Table 2 shows the measurements, along with 99.9% confidence intervals (each experiment consisting of 1000 measurements), for the AVR Atxmega 128A1U running at 32 MHz. These measurements confirm the energy-efficiency of our technique, as they closely correlate to the execution time measurements. This is reasonable and should be expected for a low-power MCU architecture: it means that the operations involved for each of the cases have similar power consumption, so that energy consumption is roughly proportional to execution time.

We also wanted the results to demonstrate the aspect that the LUT-based implementation is intrinsically more energy-efficient — as opposed to this aspect being a consequence of the smaller execution time. To this end, we compared the standard floating-point filter against our C input-centric implementation using a Raspberry Pi with a 700 MHz ARM1176JZF-S [1]. The rationale is that this processor has a floating-point unit, and thus the processing speed of the straightforward implementation is comparable to that of the LUT-based implementation. Table 3 shows the measurements for this setup, also with each measurement repeated 1000 times and ± figures indicating 99.9% confidence intervals.

Implementation	**Energy (mJ)**
Existing Approaches:	
Non-LUT Floating-point	680.19 ± 0.36
Non-LUT Integer Arithmetic	170.17 ± 0.08
This Work:	
Standard (output-centric) LUT	101.79 ± 0.06
Input-centric – C	133.43 ± 0.04
Input-centric – ASM	73.68 ± 0.04

Table 2: Energy Consumed to Process One Second of Audio – AVR MCU.

Implementation	**Energy (mJ)**
Non-LUT Floating-point	28.70 ± 0.1
LUT – Input-centric in C	25.83 ± 0.03

Table 3: Energy Consumed to Process One Second of Audio – CPU with Hardware Floating-Point Support.

The values in Table 3 indicate the total board consumption, including hardware peripherals and software overhead. Though we did not attempt to isolate the (common) background energy consumption, the lower figure for the LUT implementation confirms the intuition that accessing values in memory is bound to exhibit lower power consumption, compared to having the hardware execute the operations.

6 Case-Study: Cruise Control Application

We implemented the Cruise Control sample application included with SCADE 6 [5] on an AVR ATmega2560 running at 8 MHz. SCADE is a model-based design tool that produces C code that implements the semantics of the (graphical) model.

In the Cruise Control application, the system's inputs include accelerator and brake pedals position and the measured speed of the vehicle. In our implementation, we used a bench-top voltage source to act as the speed sensor's output. This signal is sampled by an analog-to-digital converter (ADC) at a frequency of 1 kHz, which corresponds to the processing period (the frequency of the "periodic tick"). For the purpose of the demonstration, the speed signal is assumed to be noisy (e.g., noisy or quantized speed sensor). Thus, we included a low-pass FIR filter for signal conditioning, using the standard (output-centric) LUT implementation. The output, representing the throttle to be applied, was sent to a digital-to-analog (DAC) converter, which allowed us to observe the smoothed transitions, given the smoothing applied to the input signal.

We coded all the processing in the service routine of the periodic interrupt; there, we read a sample from the ADC and feed it to the digital filter, assign the inputs to the SCADE model, and execute the main evaluation function. The C code for the processing function is shown below:

```
void process() // Invoked from periodic interrupt ISR
{
    filter_signal (read_adc());
    set_inputs();       // sets global variables
    Cruise_control_main_function();
    set_outputs();      // reads global variables
}
```

A fragment with the relevant sections of the function `filter_signal` is shown below. The filter's impulse response is $h = \{-0.05001, -0.084077, -0.018439, 0.143041, 0.315074, 0.38882, 0.315074, 0.143041, -0.018439, -0.084077, -0.05001\}$. Notice the difference, LUT_high + LUT_low for positive filter coefficients, and LUT_high − LUT_low for negative coefficients:

```
    uint16_t x[256] = {0};
    uint16_t y[256] = {0};  // Global variable
    uint8_t pos = 0;        // Global variable
        // uint8_t subscript directly cycles through
        // 256-element buffers

    #define lo8(x) ((x) & 0xFF)
    #define hi8(x) ((x) >> 8)

    void filter_signal (uint16_t input)
    {
        uint8_t k = ++pos;
        x[k] = input;

        uint16_t out = LUT_m0p05001_high[hi8(x[k])];
        out -= LUT_m0p05001_low[lo8(x[k++])];
        out += LUT_m0p084077_high[hi8(x[k])];
        out -= LUT_m0p084077_low[lo8(x[k++])];
        out += LUT_m0p018439_high[hi8(x[k])];
        out -= LUT_m0p018439_low[lo8(x[k++])];
        out += LUT_0p143041_high[hi8(x[k])];
        out += LUT_0p143041_low[lo8(x[k++])];
        out += LUT_0p315074_high[hi8(x[k])];
        out += LUT_0p315074_low[lo8(x[k++])];
        // ... etc.

        y[pos] = out;
    }
```

The `set_inputs` function is shown below:[3]

```
void set_inputs()
{
    // Assign variables specifying the state of the
    // device (on, quick accel, engage, etc.)
    accelerator pedal ← 32768;   // half-pressed
    brake pedal ← 0;             // No brakes applied
    vehicle speed ← y[pos];      // filtered speed
}
```

Table 4 shows the CPU usage, with the processing invoked every millisecond. The figures confirm the computational efficiency of the method, even when implemented as output-centric in C.

[3] Due to copyright issues, we only show pseudocode, with "blinded" variable names.

With Filtering	No Filtering
11.0%	5.9%

Table 4: CPU Usage – Cruise Control Application

A full demo and source code of the filters is accessible through a BSD-style license as a public bitbucket.org repository [4]. This includes standalone C++ programs to generate the `#include` files defining the LUTs. Due to copyright issues, both the paper and the distributed source code omit the Cruise Control complete implementation.

7 Conclusions

In this paper, we have presented a LUT-based approach to digital filtering providing a practical, fast, and energy-efficient mechanism for signal conditioning in MCU applications. Experimental results demonstrated a remarkable speed, showing that the technique can provide effective signal conditioning at low computational cost for typical embedded applications. Experimental results also confirmed the energy-efficient aspect of our proposed technique.

8 References

[1] ARM Limited. ARM1176JZF-S Technical Reference Manual, 2004–2009. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf.

[2] Atmel Corp. Atmel AVR 8-bit Instruction Set. http://www.atmel.com/images/Atmel-0856-AVR-Instruction-Set-Manual.pdf.

[3] Atmel Corp. Atxmega 128A1U, 2016. http://www.atmel.com/devices/ATXMEGA128A1U.aspx.

[4] Carlos Moreno. Source Code and Demo for LUT-based FIR Digital Filters, 2017. https://bitbucket.org/cmoreno_uw/lut-based-filters-demo-and-source-code.

[5] F.-X. Dormoy. SCADE 6: A Model Based Solution for Safety Critical Software Development. In *European Congress on Embedded Real Time Software*, 2008.

[6] R. Gee. Table Lookup Multiplier with Digital Filter, 1992. US Patent US 5117385 A.

[7] K. George and P. L. Tang. System and Method for Conditioning Noisy Signals, April 2016. US Patent EP 2818672 B1.

[8] P. K. Meher. New Approach to Look-Up-Table Design and Memory-Based Realization of FIR Digital Filter. *IEEE Transactions on Circuits and Systems*, 2010.

[9] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications.* Prentice Hall, Fourth edition, 2006.

[10] A. Savitzky and M. J. Golay. Smoothing and Differentiation of Data by Simplified Least Squares Procedures. *Analytical Chemistry*, 1964.

[11] H. Yoo and D. V. Anderson. Hardware-Efficient Distributed Arithmetic Architecture for High-Order Digital Filters. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2005.