Instruction Set Extensions for Dynamic Time Warping

Joseph Tarango, Eamonn Keogh, Philip Brisk
Department of Compuer Science and Engineering
University of California, Riverside
Riverside, CA 92521
{jtarango, eamonn, philip}@cs.ucr.edu

ABSTRACT
Processor specialization through application-specific instruction set customization can significantly improve performance while reducing energy. Due to the costs associated with semiconductor fabrication, specialized processors are only viable for products with high production volumes. The emergence of low-cost sensor-based computing products in recent years has created an urgent need to process time-series data with the utmost efficiency. Although most sensor data is fixed-point, the normalization process—an absolute necessity for highly accurate similarity search of time-series data—converts the data to floating-point in order to avoid a loss in precision. The sensors that collect time-series data are typically connected to low-power microcontrollers or RISC processors sans floating point units. The computational requirements of real-time similarity search would overwhelm such processors. To address this concern, we introduce a specialized instruction set for time-series data mining applications to a 32-bit embedded processor, yielding a 4.87x performance improvement and a 78% reduction in energy consumption compared to a highly optimized software implementation.

Categories and Subject Descriptors
C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems–*real-time and embedded systems.* H.2.8 **[Information Systems]**: Database Application – *Data Mining*.

General Terms
Algorithms, Design, Experimentation, Performance

Keywords
Time-series, Similarity Search, Dynamic Time Warping (DTW), Instruction Set Extension (ISE)

1. INTRODUCTION
Sensor-based embedded systems generate tremendous quantities of *time-series* data. Mining large time-series data sets to extract useful information is one of the foremost challenges falling under the umbrella term "Big Data." Although some time-series data can be collected and stored onto a server, other data *must* be mined in real-time, as the value of the data rapidly degrades otherwise. Real-time time-series data mining applications include speech recognition [10, 19, 20, 22], activity recognition [17, 18], robotics, financial markets, user interfaces, and many others [23].

CODES+ISSS'13, September 29 – October 4, 2013, Montréal, Canada.
978-1-4799-1417-3/13/$31.00 ©2013 IEEE

The Cisco Internet Business Solutions Group (IBSG) predicts that 25 billion devices will connect to the Internet by 2015 [6], and that this number will double by 2020. The vast majority of these devices are projected to be sensor-based embedded systems, i.e., the so-called "Internet of Things." In principle, many of these devices could and should be able to mine the time-series data produced by their sensors in real-time; however, the cost of doing so, both in terms of computation and energy, may be prohibitive.
Similarity search is the computational bottleneck in time-series data mining. One recent empirical study compared twelve widely cited time-series classification algorithms and showed that one in particular, *Dynamic Time Warping (DTW),* performed the best across forty-four data sets, including several based on human physiological data such as gestures and heartbeats [5].

At its core, DTW is a dynamic programming algorithm with a quadratic time complexity; as such, it has historically been too computationally demanding for deployment in a real-time, low-power, cost-constrained embedded system. Optimized software DWT implementations quickly discard unlikely matches, thereby eliminating the need to resort to the full-blown similarity search in the vast majority of queries. The most advanced DTW implementation published to date achieved an amortized constant time complexity per data point [23]. When tested on a data set comprising approximately 8.5 trillion electrocardiogram (ECG) data points, the optimized implementation ran in 18.0 minutes, while the prior state-of-the-art implementation took 49.2 hours.

This was a significant breakthrough, suggesting that DTW, in fact, could perform similarity search on time-series data in real-time. The experiment was performed using one core of a 2x Intel Xeon Quad-Core E5620 2.40GHz with 12GB 1333MHz DDR3 ECC unbuffered RAM, far from a cost-constrained embedded processor with limited battery lifetime. Such a platform is certainly not representative of the semiconductor devices that will provide the computational core of the "Internet of Things."

A typical "Internet of Things" device will run on a system-on-a-chip (SoC), that integrates all of its computational needs. The SoC will include some amount of specialized hardware for common functions that are performance and/or energy critical, while all other applications run in software. Prior work has shown that application-specific processors with specialized instruction sets (ASIPs) can almost equal the energy efficiency of application-specific integrated circuits (ASICs) [8], without sacrificing general-purpose functionality to support other applications.

Our opinion is that DTW falls into the category of functions that justify hardware acceleration: it is computationally demanding and applicable to a wide variety of SoC-based products, thereby justifying the cost of dedicated silicon; an ASIP specialized for DTW is therefore an attractive component for SoC integration.

This paper presents the design and evaluation of an ASIP specialized for DTW. Our objective in designing this ASIP was to

make it as low cost as possible, so that it could be deployed across a wide variety of SoC products, from low-end to high-end, in order to maximize market penetration. The simplest such product would be a simple sensor+ASIP, and should not cost more than a low-end microcontroller or RISC processor. On the high-end, the processor should be capable enough for integration into the semiconductor industry's most advanced SoC products, such as smart phones, tablet PCs, and SSD controllers.

We started with an extensible 32-bit RISC processor and source code for a highly tuned DTW implementation written in C [23]. We identified four computational bottlenecks, each of which could be replaced with an instruction set extension (ISE). Empirically, we observed that introduction of further ISEs did little to improve the performance or energy efficiency of the ASIP, so we stopped with four. Compared to the baseline, the ISEs increased performance by a factor of 4.87x while reducing energy consumption by 78%, when prototyped on an FPGA.

2. TIME-SERIES SIMILARITY SEARCH

2.1 Definition

Let $T = t_1, t_2, \ldots, t_m$ be a time-series, i.e., an ordered list of numbers; in principle these numbers can be integer or real-valued, fixed- or floating-point. In many cases, we consider only a shorter region of T called a subsequence: $T_{i,k} = t_i, t_{i+1}, \ldots, t_{i+k}$. If there is no ambiguity, we refer to a subsequence $T_{i,k}$ as a candidate, C, which eliminates unnecessary subscripting. The general objective is to match a candidate C against a query Q.

In a real-time context, similarity search is performed periodically (e.g., after a fixed number of sensor readings), over a relatively recent window of activity. For example, when determining if a patient is experiencing a heart attack right now, then the query should focus on a recent window of candidate points; yesterday's data points will provide minimal information regarding today's heart attack, and querying them would drain the battery and potentially divert computational resources that could be used for a more prescient purpose.

2.2 Euclidean Distance

Let $|C| = |Q| = n$, the *Euclidean Distance (ED)* between C and Q, denoted ED(Q, C), is

$$ED(Q,C) = \sqrt{\sum_{i=1}^{n}(q_i - c_i)^2}. \quad (1)$$

If $|C| = n$, $|Q| = m$, and $m < n$, then ED can be computed over a sliding window, as discussed above; requiring ED to be computed $n - m$ times to be exhaustive. The smallest ED computed over the sliding window would be returned as the result.

As shown in Fig. 1(a), ED considers a one-to-one mapping in time between the points on the two curves. Fig. 1(b) highlights ED's biggest weakness; it may miss time series that are similar to one another, but with minor offsets in time. DTW, introduced next, corrects this weakness.

2.3 Dynamic Time Warping

DTW is a quadratic-time dynamic programming algorithm: If $|C|=n$ and $|Q|=m$, then its time complexity is $O(mn)$. In practice, $m << n$, as the query is a relatively short pattern (e.g., a motif representing a heart attack), while n is a much longer time series (e.g., a human heartbeat over time). By considering multiple alignments in context, DTW effectively generalizes ED; however, this generalization comes at a significant cost in terms of runtime.

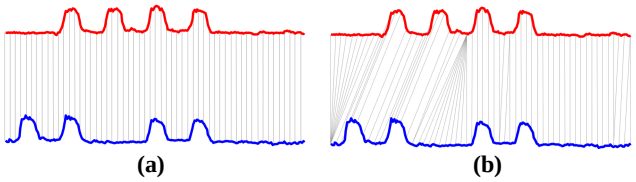


Figure 1. (a) ED (b) and DTW are similarity search metrics; DTW enables realignment in time, while ED does not.

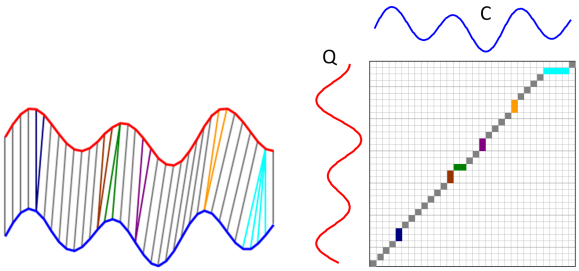


Figure 2. (a) Query (Q, red) and Candidate (C, blue) (b) DTW warping path for alignment between Q and C.

DTW creates an n-by-n matrix M, where $M_{i,j} = d(q_i, c_j) = |q_i - c_j|$ is the distance between points q_i and c_j. DTW permits a one-to-many alignment, which allows query point q_i to align with one or more candidate points, $c_{i-k}, c_{i-k+1}, \ldots c_{i+k-1}, c_{i+k}$, as shown in Fig. 2.A *warping path* P is a contiguous set of matrix elements that establishes a mapping between Q and C. The k^{th} element of P is denoted $P_k = (i, j)_k$; a warping path has the form

$$P = p_1, p_2, \ldots, p_k, \ldots, p_K \quad (2)$$

$$max(m,n) \leq K \leq m+n-1.$$

A warping path is subject to the following constraints:

- **Boundary conditions**: $p_1 = (1,1)$ and $p_K = (m,n)$; W starts and finishes at diagonally opposite corners of the matrix.
- **Continuity and Monotonocity**: If $p_k = (x,y)$, then $p_{k-1} = (x',y')$ where $0 \leq x - x' \leq 1$ and $0 \leq y - y' \leq 1$; this restricts W to (diagonally) adjacent cells, monotonically spaced in time.

The optimal warping path minimizes the warping cost

$$DTW(Q,C) = min\left\{\frac{\sqrt{\sum_{k=1}^{K} M(p_k)}}{K}\right\}; \quad (3)$$

the denominator compensates for paths having different lengths.

Let $\gamma(i,j)$ denote the cumulative distance w_1 to matrix element (i,j), computed recursively, as follows:

$$\gamma(i,j) = d(q_i, c_j) \quad (4)$$
$$+ min\{\gamma(i-1,j-1), \gamma(i-1,j), \gamma(i,j-1)\}.$$

$\gamma(i,j)$ is computed in terms of the cumulative distances of its adjacent neighbors, preceding it on the warping path. The optimal warping path is thereby computed using dynamic programming.

In many practical applications, the issue goes beyond finding the best DTW alignment. For example, finding the best match of an ECG motif representing a heart attack does not mean that a heart attack is occurring; the quality of the match must be assessed.

2.4 Z-Normalization

The time series being compared need to be normalized; this addresses the situation where the time-series are similar, but differ in terms of amplitude, as shown in Fig. 2. The candidate and query have near-identical shapes, but differ by approximately a constant, i.e., $q_i = c_i + k$. If k is large then the ED metric will suggest a large distance (and this will manifest itself in the DTW calculation as well), even though the series have otherwise-identical shapes. Z-Normalization addresses this issue, and is applied to the time series prior to classification.

Without loss of generality, we will Z-normalize C. The arithmetic mean and standard deviation of C are:

$$\mu_C = \frac{1}{n}\sum_{i=1}^{n} c_i, \text{ and} \quad (5)$$

$$\sigma_C = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(c_i - \mu_C)^2}. \quad (6)$$

Each data point c_i is then replaced with a normalized counterpart

$$c_i' = \frac{c_i - \mu_C}{\sigma_C} \quad (7)$$

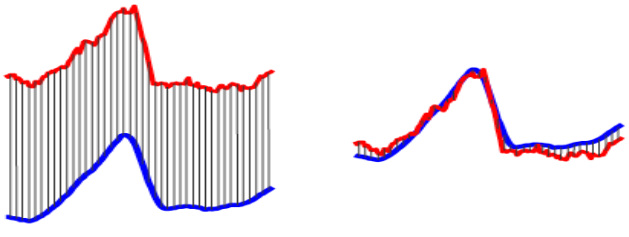
Prior work has shown that normalization is necessary to make meaningful comparisons between time series, and failure to do so can lead to erroneous results (i.e., false negatives) [11]. Each subsequence must be normalized before making a comparison; it is insufficient to normalize an entire dataset. When a sliding window is used, the window itself must be re-normalized as as it slides! We revisit this issue later in Section 3.1.

Even if the time series data are integers, Z-normalization yields real values because of the division and square root operations in Eqs. (5)-(7). Embedded microcontrollers and RISC processors without floating-point units must execute Z-normalization and subsequent DTW operations in software, as they generally do not have integrated floating-point hardware; thus, the performance and energy overheads become significant. Even with floating-point units in-place, specializing the hardware implementation of these calculations can result in significant improvements.

3. OPTIMIZED DTW IMPLEMENTATION

We assume a candidate window of size W, meaning that the query is matched against the W most recent time series data points. Each new sensor reading shifts the window by 1. Let R be the similarity search interval, meaning that DTW is performed after every R sensor readings, meaning we discard the R oldest data points to make room in the window for the newly read data point.

In practice, most queries do not yield positive matches; numerous enhancements have been published that quickly abandon DTW before or during the dynamic programming phase. Our ASIP is based on a recent software implementation of DTW that performs many of these optimizations, which are described next.



(a) (b)

Figure 3. (a) Un-normalized and (a) Z-normalized time series. Z-normalization noticeably reduces the ED and DTW metrics.

3.1 Using the Squared Distance

ED and DTW compute square roots; however, this computation does not alter the rankings of nearest neighbors relative to one another because the functions are monotonic and concave. Eliminating it improves performance and enables subsequent optimizations; the square root from Z-Normalization, Eq. (6), unfortunately, is still required. The *Squared Distance (SD)* refers to the ED (Eq. (1)), with the square root removed.

3.2 Lower Bounds

Computationally efficient lower bounds that recognize non-matching queries can preempt the full DTW computation. At least 18 lower bounds have been reported in past literature [23], three of which are relevant to our implementation:

LB_{Kim} [14] shown in Fig. 4(a), computes the SD between the two sequences first (A), minimum (B), maximum (C), and last (D) points as the lower bound. The time complexity of LB_{Kim} is $O(n)$ because the query and candidate must be searched to find the minimum and maximum values.

LB_{KimFL} [14, 23] is a variant of LB_{Kim} that computes the SD between the first (A) and last (D) points as a lower bound, omitting the minimum and maximum points (B and C). This reduces the time complexity to $O(1)$. Z-normalization tends to reduce the SD of the minimum and maximum points, so the loss in terms of the tightness of the bound is minimal.

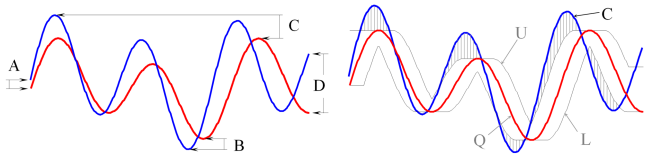
LB_{Keogh} [5, 7, 12], shown in Fig. 4(b) and below in Eq. (8), takes a query Q and creates two other curves, L and U, that envelope Q.

$$LB_{Keogh(Q,C)} = \sum_{i=1}^{n} \begin{cases} (c_i - U_i)^2 & if\ c_i > U_i \\ (c_i - L_i)^2 & if\ c_i < L_i \\ 0 & otherwise \end{cases} \quad (8)$$

The ED between C and the closest part of the envelope, rather than the corresponding point of Q, is taken as a tight lower bound. If the current sum of the SDs exceeds a threshold, indicating that a match will not occur, then the similarity search between C and Q can be abandoned early. The time complexity of LB_{Keogh} is $O(n)$.

3.3 Early Abandoning ED and LB_{Keogh}

When computing the ED or LB_{Keogh}, the current sum of the squared differences can be compared to the best DTW value computed thus far. If the computation exceeds the lower bound, then the calculation can be stopped, knowing that there is no possible way that the current match could. improve upon it, as shown in Fig. 5.



(a) (b)

Figure 4. (a) LB_{Kim} [14] and (b) LB_{Keogh} [5, 7, 12].

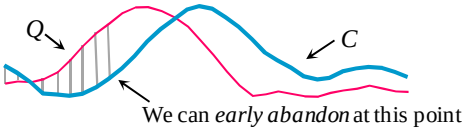
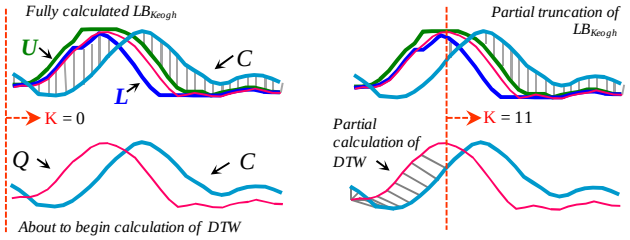


Figure 5. ED early abandoning; in this example, we abandon after summing the SD of 9 out of 32 pairs of data points [23].



(a) (b)

Figure 6. (a) The DTW computation starts with LB_{Keogh} computed up-front. (b) As DTW completes from from K-1 to K, the dashed line moves right; combining the contribution of LB_{Keogh} from the right of the dashed line (top) with the partial DTW from the left of the dashed line (bottom), tightens the bound as exact DTW values replace LB_{Keogh} estimates [23].

3.4 Early Abandoning DTW

A full LB_{Keogh} lower bound that does not prune the query does not guarantee that the full DTW guarantees a match; however, the LB_{Keogh} result does not need to be discarded. As shown in Fig. 6, the basic idea here is to incrementally compute DTW in the range *1…K*, and add the partial accumulation to the LB_{Keogh} contribution from *K+1* to *n*. In other words

$$DTW(Q_{1:K}C_{1:K}) + LB_{Keogh}(Q_{K+1:n}C_{K+1:n}) \leq DTW(Q_{1:n}C_{1:n}), \quad (9)$$

and the bound becomes tighter as *K* increases. If this lower bound exceeds the best DTW value computed thus far, the calculation is pruned and DTW can be abandoned [23].

3.5 Early Abandoning Z-Normalization

Early abandoning Z-normalization [23] incrementally computes a lower bound such as ED or LB_{Keogh} for each data point while that point is being normalized. This enables the normalization step to benefit from early abandonment as well. Since every subsequence of a candidate must be normalized before comparing it to query, it is possible to quickly compute the mean by keeping two running sums with a lag of *m* values; the variance is computed similarly:

$$\mu_C = \frac{1}{m}\left(\sum_{i=1}^{k} c_i - \sum_{i=1}^{k-m} c_i\right), \text{ and} \quad (10)$$

$$\sigma_c^2 = \frac{1}{m}\left(\sum_{i=1}^{k} c_i^2 - \sum_{i=1}^{k-m} c_i^2\right) - \mu_c^2. \quad (11)$$

Fig. 7 shows pseudocode that describes the early abandoning process. A circular buffer *X* stores the candidate, *C*, which is compared with the query, *Q*. The current window is an *m*-element contiguous sub-array of the circular buffer. Each new sensor reading (line 5) overwrites a value in the buffer and shifts the window by one. In the previous iteration (lines 15, 16), the running sums for the average (*ex*) and standard deviation (*ex2*) were updated to remove the value that will be shifted out of the window during the current iteration. *ex* and *ex2* are then updated to account for the new sensor value (line 6), and the mean and standard deviation are updated incrementally (line 8).

Next, the ED for the current window is computed (lines 10-12) including the possibility of early abandonment. Each data point in the current window is normalized on-the-fly (line 11), but the normalized value is not retained; since each new sensor reading changes the average and standard deviation, the normalized value of each data point changes as the window shifts. The process then repeats as the next time series data point is read.

Algorithm	Similarity Search
Procedure	[*L*] = *similaritySearch* (T,*Q*)
1	*best-so-far* ← ∞,*count*← 0
2	*Q* ← *z-normalize*(*Q*)
3	**while** !*next*(T)
4	*i*← *mod*(*count*,*m*)
5	X[*i*] ←*next*(T)
6	*ex*← *ex*+ X[*i*], *ex2*← *ex2*+X[*i*]²
7	**if** *count* ≥ *m*-1
8	μ← *mean*(*ex*,*ex2*,*m*),σ← *stdv*(*ex*,*ex2*,*m*)
9	*j*← 0, *dist*← 0
10	**while** *j* < *m* **and** *dist* < *bsf*
11	*dist*← *dist*+*Q*[*j*]-(X[*mod*(*i*+1+*j*,*m*)]-μ)/σ
12	*j*← *j*+1
13	*ex*← *ex*-X[*mod*(*i*+1,*m*)]
14	*ex2*← *ex2*-X[*mod*(*i*+1,*m*)]²
15	*count*← *count*+1

Figure 7. Pseudocode for similarity search with early abandoning Z-normalization [23].

One concern regarding this approach is the accumulation of floating-point error, as time-series values are added and subtracted from *ex* and *ex2*; to mitigate this factor, the *Z*-normalization resets after one million subsequences, and renormalizes the window. It is important to note that Eqs. (5) and (6) are rarely computed outright, but are updated, for the current window, on-the-fly.

3.6 Reordering Early Abandoning

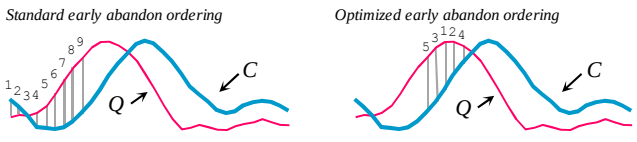
In Figs. 5 and 6, ED and/or SD are computed from left to right; in principle, the distances between pairs of points can be computed in any order. Empirically, it has been shown that sorting the data points based on the absolute values of the Z-normalized query values q_i', and processing them in that order, tends to reduce the number of data points processed when early abandonment occurs. This is not an absolute rule, and offers no theoretical guarantees of improvement. Fig. 8 shows an example.

3.7 Reversing the Query/Data Role in LB_{Keogh}

In Section 3.2 and Eq. (7), $LB_{Keogh}(Q,C)$ builds an envelope every around the query. This is done once, as the query never changes. In principle, we can build an envelope around the candidate, rather than the query; however, the candidate is not fixed, as new data points are added and removed with each new sensor reading. The overhead of building the envelope, however, is quite high. If all other lower bounds fail, we can build the envelope around the sliding window and compute a lower bound $LB_{Keogh}(C,Q)$ as well, as shown in Fig. 9, possibly pruning the query.

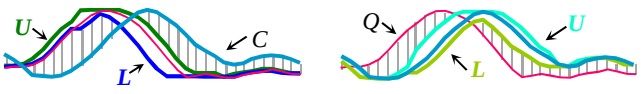
3.8 Cascading Lower Bounds

Cascading lower bounds, shown in Fig. 10, refers to the process of trying all known lower bounds, one after another, sorted by time complexity, hoping to achieve early abandonment; a pruning rate of 99.9999% is reported using this approach [23].



(a) (b)

Figure 8. (a) Standard early abandoning starts from the leftmost data point and works to the right; (b) changing the order in which data points are processed can lead to faster and more effective pruning [23].



(a) (b)

Figure 9. (a) $LB_{Keogh}(Q, C)$ and (b) $LB_{Keogh}(C, Q)$.

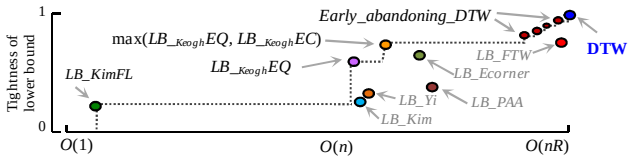


Figure 10. Cascading lower bounds; the tightness of each bound is plotted as a function of its time complexity. Note that $LB_{Keogh}EQ$ refers to Eq. (8), and $max(LB_{Keogh}EQ, LB_{Keogh}EC)$ refers to reversing the query role (Section 3.7) [23].

The overhead of applying all known lower bounds for DTW is significant; Fig. 10 shows a representation of the lower bounds that the authors of ref. [23] considered to be Pareto optimal based on a ranking of their tightness as a function of time complexity; only four lower bounds plus early-abandoning techniques are considered to be Pareto optimal.

3.9 Sakoe-Chiba Band

The Sakoe-Chiba Band [24] (Fig. 11) limits the space of the DTW computation to a narrow band in the matrix, guaranteeing that the warping path deviates by at most R cells from the diagonal:

$$Sakoe\_Chiba \begin{cases} U_i = max(q_{i-r} : q_{i+r}) \\ L_i = min(q_{i-r} : q_{i+r}) \end{cases}. \quad (12)$$

It reduces the computational cost of the computation, but eliminates many warping paths from consideration, thereby sacrificing the theoretical optimality of the DTW computation.

4. APPLICATION ANALYSIS

We started with a publicly available implementation of DTW, as described in Section 3, written in C [23], and a publicly available data set [13]. Figs. 12(a) and (b) show a call graph of the DTW application, and its workflow; modules shown in color represent subroutines that we accelerated with instruction set extensions.

4.1 Data Precision Analysis

Both the DTW source code and publicly available data set were in double-precision (64-bit) floating-point format. Looking at the data values, we realized that it could be converted to fixed-point and/or lower-precision floating-point formats with no loss in precision. We converted the data set and rewrote the source code to use several different fixed-point (Q8.8, Q16.16, Q32.32) and floating-point (half, single, and double precision) formats and ran the application on a modern desktop PC; we observed erroneous operation (e.g., no query matches) at the reduced precision levels.

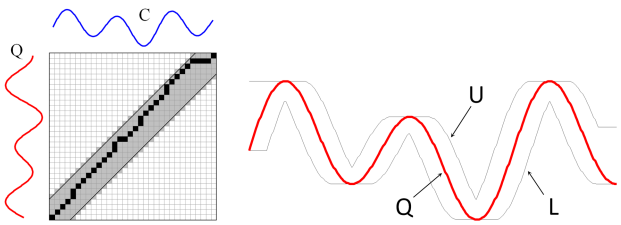
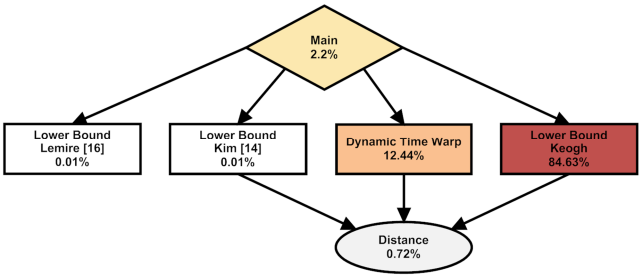
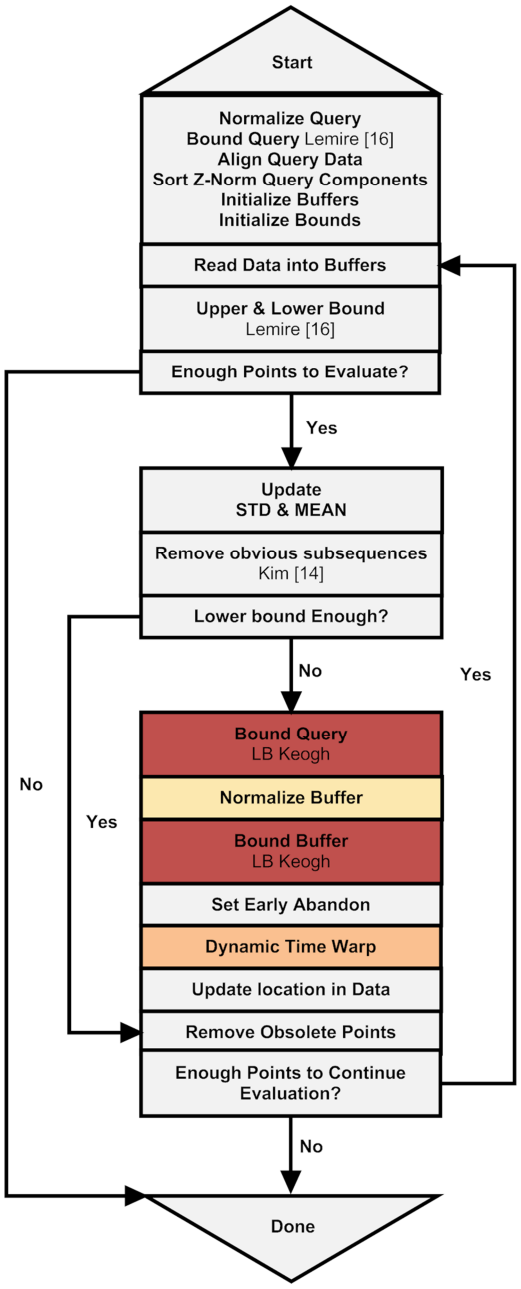


Figure 11. The Sakoe-Chiba band [24] creates a tighter bound around the query and prevents pathological warp patterns.



(a)



(b)

Figure. 12. (a) A call graph (b) and high-level control flow graph illustrate the basic workflow of the DTW application. Modules shown in color are computationally intensive, as determined by profiling.

We modified the source code to track the arithmetic precision of the data on an operator-by-operator basis. We observed that the Z-normalized values of the candidate and query quickly approached zero, within one hundred iterations. The primary culprits were the floating-point multiplication, division, and square root operations.

To further understand the loss in precision, we implemented our own floating-point operators and traced the intermediate values after each sub-step. The normalization and rounding steps at the end of the multiplication and division operators were the primary loss cause for the loss in precision; the precision loss during mantissa alignment for addition and subtraction was negligible. Our square root operator used the Newton-Raphson method, which repeatedly performs division, so all of the aforementioned issues affected this operation as well. Reducing the precision of the floating-point operations from double to single and half yielded similar results, but with different rates of decay.

With a more restrictive dynamic range, the fixed-point operators were thoroughly erroneous. Z-normalization rapidly zeroed out most values leading to incorrect query matches.

For these reasons, we reverted back to the original C language implementation with double-precision operations. At this point, we were confident that fixed-point and lower-precision floating-point operators introduce excessive error into the calculations, and that double precision was the most appropriate choice.

4.2 Supporting Floating-point in Hardware

As discussed in Section 5, our target processor is an extensible 5-stage RISC pipeline that does not natively support double-precision floating-point instructions; however, we can introduce them by extending the instruction set and integrating a double-precision *floating-point unit (FPU)* into the architecture.

We started with a software floating-point library in C. In software, floating-point operations are variable-latency; to estimate the average latency, we executed each operation 100 times in a loop, based on input data obtained from our DTW application, and averaged the latencies. We did this twice: first, with low compiler optimization levels (e.g., gcc -O0 and -O1), and then with higher optimization levels (e.g., gcc -O2 and –O3); in the latter case, the compiler fully unrolled the loops and software-pipelined the operations, in order to achieve higher overall performance.

We also designed and implemented our own double-precision floating-point operators in VHDL and synthesized them on our target platform (a Xilinx EK-V6-ML605-G Virtex 6 FPGA) with a target frequency of 100 MHz.

Table I reports the latencies of the operators.

4.3 Application Profiling

We profiled the software using one million data points of the data suite and a query size of 128. Since we are optimizing the DTW implementation for a specific query length, as determined by our medical application, we know that loops will be unrolled by a factor that does not exceed the query size (128 in our case). If we are running a multi-query instance of DTW, the unroll factor will be the least common multiple (LCM) of the query sizes.

We identified several critical code regions, which are highlighted in Fig. 12(b). Based on our estimates, LB_{Keogh} consumes 84.63% of computation cycles; the dynamic programming DTW computation consumes 12.44%; and Z-normalization consumes 2.2%. The exact percentages vary with the loop unroll factor, but the general trend remains consistent.

Table I. Latencies of double-precision floating-point operators in software (with and without pipelining) and hardware.

	Software		
Operator	Non-pipelined (gcc -O0/O1)	Pipelined (gcc -O2/O3)	FPU
Add	433	285	6
Sub	488	345	6
Mul	516	394	7
Div	498	360	19

4.4 Instruction Set Extension (ISEs)

We tried to run a standard ISE identification and selection algorithm [21] on the critical kernels of the DTW code; unfortunately, it does not support floating-point ISEs. Instead, we identified and selected ISEs manually, rather than automatically; automating the identification and selection process for floating-point ISEs is left open for future work.

The four ISE candidates that we selected manually are:

ISE-Norm: *Main* and LB_{Keogh} perform Z-normalization, e.g., Eq. (7), accounting for 81.3% of total execution time. In Fig. 7, the average and standard deviation are updated on-the-fly for each new time-series datum, so the sums shown in Eqs. (5) and (6) are not computed outright, and are not part of this ISE.

ISE-DTW: The recurrence relation at the core of the DTW dynamic programming algorithm (Eq. (4)) accounts for 12.44% of total computation cycles.

ISE-Accum: In LB_{Keogh} floating-point accumulation ($a = a + b$) accounts for 5.52% of the total computation cycles. The overhead of transmitting data to an ISE can impact performance. Compared to standard floating-point adder ($a = b + c$), the accumulator transmits one value (b) rather than two (b and c).

ISE-SD: The SD function (Eq. (1) without the square root) accounts for 0.72% of computation cycles.

4.5 ISE Synthesis

We used the FloPoCo arithmetic core generator to convert each ISE to VHDL [4], which we then synthesized on our target FPGA platform. After synthesizing each operator, we can estimate the speedup attained through the ISEs. FloPoCo performs many optimizations on compound operators, for example, by eliminating internally redundant normalizations between operators [15, 26]; as well as other FPGA-specific optimizations [15].

We enhanced FloPoCo in several respects. We introduced *operator matching* in floating-point pipelines; for example, consider *(a+b)/(c*d)*; since the latencies of the floating-point adder and multiplier differ, registers are inserted to ensure that the resulting operations arrive at the divider at the same time. We also introduced *'valid' bit propagation*, which allows unused stages of arithmetic operators to switch into a low-power mode, conserving energy. Lastly, we introduced a new floating-point comparator that selects the minimum of three values in ISE-DTW (Eq. (4)).

Table II reports the latencies of the ISEs in software (pipelined and non-pipelined), with floating-point operators, and ISEs. In two cases (ISE-Norm and ISE-Accum), the ISE logic has a higher latency than the FPU-based implementation; this is because the ISE logic must wait for all data to arrive before the pipeline can start. The ISE logic is designed to enable software pipelining, where the objective is to maximize throughput, rather than to minimize latency.

Table II. Latencies of ISEs in software (with and without pipelining), using floating-point operators, and specialized hardware ISE logic generated by FloPoCo.

	Software			
ISE	Non-Pipelined (gcc -O0/O1)	Pipelined (gcc -O2/O3)	FPU	Custom ISE Logic
ISE-Norm	802	613	27	31
ISE-DTW	1851	1575	40	26
ISE-Accum	433	285	9	12
ISE-SD	889	712	18	16

4.6 Performance Estimation

We used standard profiling techniques to identify the performance bottlenecks in the DTW software [1]; profiling weights are assigned to control flow edges, and memory access latencies are based on coarse-grained estimates.

We compiled the DTW software using a query length of 128 using gcc 4.1.2 at optimization level –O3 (loops fully unrolled and software pipelined). Fig. 13 reports the percentage of execution time of different phases of the application as ISEs are introduced one-by-one. With no ISEs, normalization is critical; after speeding up normalization with ISE-Norm, DTW becomes critical. Introducing the three subsequent ISEs changes the critical code section. After introducing all four ISEs, normalization is once again critical, accounting for more than 50% of total execution time. Since ISE-Norm has already accelerated normalization, we stop here, as further acceleration would yield diminishing returns.

5. PROCESSOR DESIGN

To support double-precision floating-point operations and ISEs, we chose a Xilinx MicroBlaze, a 32-bit extensible RISC processor as the baseline. The MicroBlaze is a Harvard architecture with an in-order 5-stage pipeline and 32-bit general-purpose registers. It supports a single-precision floating-point ALU as a configuration option [28]. We added double-precision floating-point ALU operations (Table I) and application-specific accelerators (Table II) as ISEs, as shown in Fig. 14.

Our evaluation platform is a Xilinx EK-V6-ML605-G Virtex 6 development board; on this platform, the MicroBlaze operates at 100 MHz [29]; we estimate that it would operate at around 1 GHz if it was synthesized using a standard cell design flow.

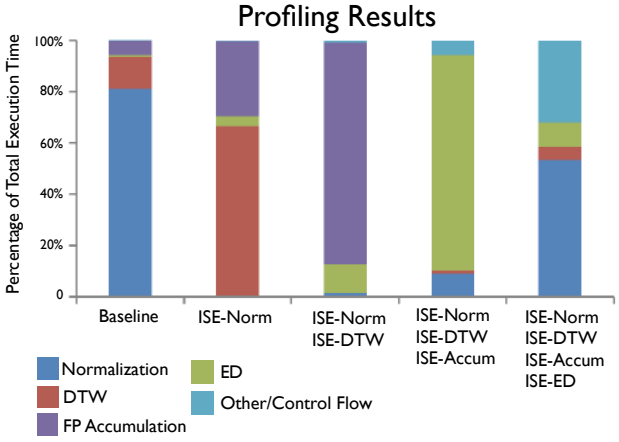


Figure 13. Introducing ISEs significantly alters which computational kernels dominate application performance.

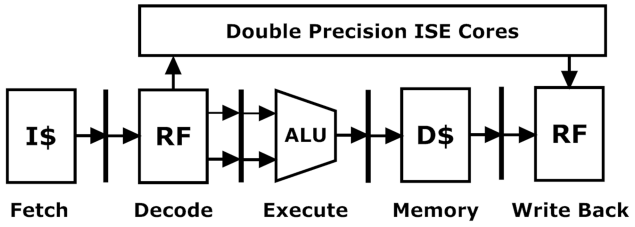


Figure. 14. The MicroBlaze does not support double-precision floating-point operations as part of its instruction set, but they can be added as custom instruction set extensions.

5.1 Custom ISE Operators and Interface

The MicroBlaze has been optimized for performance and includes several interfaces including: Cache Link (CL), Fast Simple Link (FSL), Instruction/Data Local Memory Buses (I/D-LMB), and Instruction/Data OnChip Peripheral Buses (I-OPB) [30]. The point-to-point FLS provides a tightly coupled interface to the register file, allowing seamless integration of ISEs into the processor, without requiring pipeline modifications or affecting the critical path, as shown in Fig. 15.

The FSL uses a single 32-bit Master/Slave interface with optional FIFO data buffers. Since we are using 64-bit double-precision floating-point operations, we separated data into 32-bit words. The interface includes 16-entry FIFO buffers. As the Microblaze operates on 32-bit words of data per cycle, we developed our own ISE operator model that uses *finite state machines (FSMs)* to manage data transfers. Different FSMs were created, depending on the number of data elements to transfer.

Table III lists the critical path delay and area overhead the FSMs. Pop-*k* (Push-*k*) refers to an FSM that transfers *k* data elements, requiring *k* cycles. Tables I and II *do not include* these latencies, which affect both FPU and custom ISE logic.

All ISEs were pipelined to match the system clock. The latency of executing each ISE includes its pipeline depth, plus the number of FSM cycles for pushing and popping, which transfer the data to and from the processor's register file. Since ISEs operate on 64-bit values, two cycles are required to read and write each value. For example, the total latency of ISE-DTW is (10 + 14 + 2 = 26 cycles), noting that Eq. (4) has five inputs.

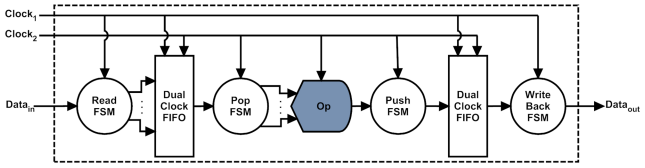


Figure. 15. ISE interface, with dual-clock FIFOs and finite state machine (FSM) control.

Table III. FSM I/O Interface (Fig. 15): critical path delay and area overhead.

FSM	Clock (ns)	Slice Regs	Slice LUTs	LUT FF
Pop-2	1.254	99	130	98
Pop-4	1.045	228	290	291
Pop-6	1.688	226	328	334
Pop-10	1.703	494	520	525
Push-1	1.031	33	50	50
Push-2	1.316	68	68	68

Table IV. Synthesis summary of the double-precision floating-point arithmetic operators.

Combinational					
Operator	Cycles	Clock (ns)	Slice Regs.	Slice LUTs	LUT FF
Add/Sub	1	22.3	203	1627	1734
Mul	1	22.7	12	761	761
Div	1	24.2	128	523	572
Compare	1	3.79	0	121	121
Pipelined					
Operator	Cycles	Clock (ns)	Slice Regs.	Slice LUTs	LUT FF
Add/Sub	6	5.61	659	910	950
Mul	7	6.28	513	1017	413
Div	19	7.42	2841	4637	1307

Table V. Synthesis summary of the four ISEs introduced to accelerate the DTW application.

Combinational					
Operator	Cycles	Clock (ns)	Slice Regs.	Slice LUTs	LUT FF
ISE-Norm	1	156	283	10672	10758
ISE-DTW	1	34.9	214	1978	2114
ISE-Accum	1	22.3	203	1627	1734
ISE-SD	1	35.3	206	2090	2011
Pipelined					
Operator	Cycles	Clock (ns)	Slice Regs.	Slice LUTs	LUT FF
ISE-Norm	23	7.42	3436	5515	6257
ISE-DTW	14	8.33	2270	2501	2970
ISE-Accum	6	5.61	659	910	950
ISE-SD	10	6.17	1151	1263	1325

5.2 Floating-point Operators and ISEs

All double-precision floating-point operators and ISEs were synthesized as combinational operators, and then pipelined to meet a target frequency (delay) of 100 MHz (10ns). Tables IV and V report the synthesis results, including cycle count, delay, and FPGA resource usage. Among the basic operators, the comparator executes in a single-cycle, and does not require pipelining.

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

We used a Xilinx EK-V6-ML605-G Virtex 6 development board for prototyping. It was connected to a desktop PC running Red Hat Enterprise Linux Workstation release 6.3 (Santiago) x86_64. We compiled the DTW application using gcc 4.1.2, and synthesized all custom hardware using Xilinx Embedded System Design Software version 12.4; power and energy estimates were obtained using Xilinx XPower Analyzer [31].

The MicroBlaze configuration was set for single core running at 100MHz with the PLB peripheral bus. Table VI lists the instruction and data cache configurations that we used. We also enabled 64-bit fixed-point multiplication, a hardware divider, and a branch target cache with 2048 entries.

We used a query length of 128. Our input data set consisted of one million double-precision floating-point time-series data values, which exceeded FPGA's block RAM capacity. We modified the linker script to map data sections to the 512MB external DDR3 memory. As part of the configuration process, a PLB peripheral populates the DDR3 memory prior to executing the application.

Table VI. MicroBlaze cache configurations.

Parameter/Policy	I-Cache	D-Cache
Capacity	64KB	64 KB
Cache Line Length	32 bytes	32 bytes
Allocation Policy	Read/Write Allocate	Read/Write Allocate
Associativity	Direct-mapped	Direct-mapped
Victim Buffer Size	8 Victims	8 Victims
Other	1 Stream	Write-back

We introduced parameterized function calls into the source code to invoke the double-precision FPUs and ISEs; inline assembly separated data into 32-bit words and populated the FSL. For each invocation, the MicroBlaze was configured to execute a sequence of NOPs equal in length to the pipeline depth of the operator or ISE. At optimization levels -O2 and -O3 gcc fully unrolled loops and applied software pipelining to best utilize the FPU and ISEs.

6.2 Results

Fig. 16 reports the execution time of the DTW application running on several architectural configurations at varying gcc optimization levels. At -O3, four ISEs yield a 4.87x speedup compared to software execution, and 1.42x compared to the integrated FPU. Multiple ISEs are required before a speedup can be obtained over the FPU because each ISE accelerates one or two specific computations, while the FPU can execute any floating-point operation in the program. With few ISEs and no FPU, all other floating-point operations execute in software. Collectively, the four ISEs execute the vast majority of floating-point operations.

Fig. 17 reports the peak power and energy consumption of different MicroBlaze configurations when running the application compiled with gcc at the -O3 optimization level. The ISEs compare 78% less energy than the baseline processor and 35% less than the baseline processor augmented with a pipelined FPU. The ISEs save power by executing floating-point computations more efficiently than the processor or the FPU, and by avoiding redundant data transfers. For example, consider ISE-Norm (Eq. (7)), which has six inputs and two outputs: 256 bits are transferred. When using the FPU, both the adder and multiplier require four inputs and two outputs, so 384 bits are transferred.

Peak power is an upper bound over application execution, and is relatively flat across all processor configurations. The PLB bus, FPU, and ISEs do consume power when executing or idle, which accounts for the relatively small variations reported in Fig. 17. Table VII reports the peak power consumed by the operators.

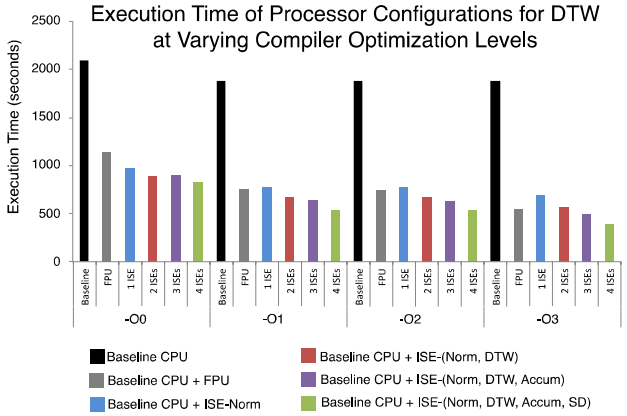


Figure. 16. Execution time of DTW processor configurations at varying compiler optimization levels.

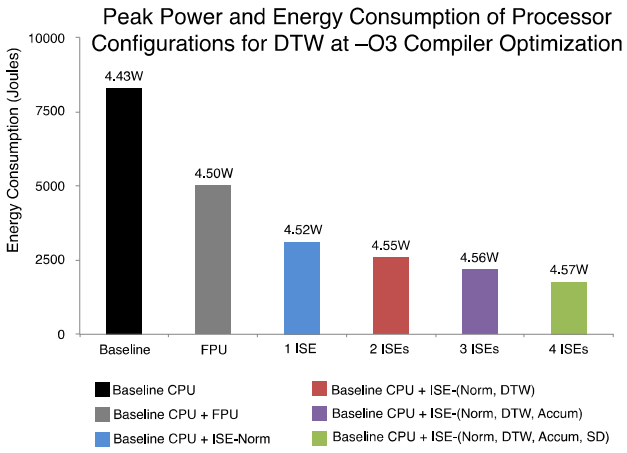


Figure. 17. Energy and peak power consumption of different processor configurations. The DTW application was compiled using gcc at the -O3 optimization level.

Table VII. Peak power consumption of FPU operators and custom ISE logic units.

FPU Op.	Peak Power (W)	ISE Logic	Peak Power (W)
Add/Sub	0.001	ISE-Norm	0.107
Mul	0.007	ISE-DTW	0.043
Div	0.129	ISE-Accum	0.001
		ISE-SD	0.02

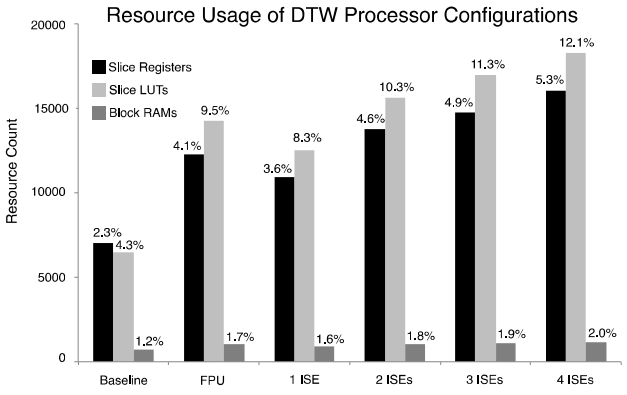


Figure. 18. FPGA resource usage (absolute and percentages) of different DTW processor configurations.

Fig. 18 reports the resource usage of different DTW processor configurations running on the FPGA. Considering Slice LUTs, adding the FPU unit increases the area compared to the baseline by 2.1x, and all four ISEs increases the area by 2.8x. As DTW is not memory-intensive, and time-series data is naturally streaming, the FPU and ISEs make limited use of block RAMs.

7. RELATED WORK

7.1 DTW Accelerators

Early processors introduced in the 1980s featuring custom DTW accelerators for speech recognition [10, 19, 20, 22]. More recently, low-power DTW accelerators were developed for body-area sensor networks to accelerate activity recognition [17, 18]. These designs accelerate the dynamic programming kernel, but do not perform normalization, which becomes a bottleneck when the software is highly tuned.

GPUs and FPGAs can accelerate DTW, improving performance over software by 2 and 4 orders of magnitude respectively [25]; these implementations perform normalization, but do not consider early abandoning through lower bounds. A more recent FPGA implementation adds lower bounding capabilities, including the reversal of the query and data role in LB_{Keogh} (section 3.7) [27], thereby achieving higher throughput than prior work.

The DTW-accelerator processor introduced here is aimed for the embedded market, where minimizing cost and power are the most important objectives. GPUs and FPGAs can easily achieve higher throughput than our proposed processor, but are several orders of magnitude more expensive in terms of both cost and power. The aforementioned DTW accelerators for body-area sensor networks are similar in cost to the DTW processor proposed here.

7.2 ISEs and Floating-Point

Most of the energy consumption in a 5-stage RISC pipeline is due to instruction fetch-and-decode and transferring data to and from the register file [3]. If an *n*-cycle software routine is replaced with an *m*-cycle ISE ($m \leq n$), then *n* fetch-and-decode operations are replaced with *one* fetch-and-decode (the ISE itself) followed by *m-1* state machine transitions in the ISE control logic. Amortized over the entire execution of a program, this can lead to significant energy savings, as is the case here.

One recent study showed that ISEs could eliminate the energy and performance gaps between software and ASIC implementations of an H.264 encoding, while automating the architectural design and verification process [8]. We believe that our approach achieves principally similar results for matching one 128-element query.

Most work on ISE identification, selection, and synthesis has focused on fixed-point operators [21], and these algorithms have not been extended to handle floating-point ISEs; models to estimate the speedup and energy reduction attainable via floating-point ISEs are needed. Possible improvements include operators with customized precision, merging operators to reduce area [2], and removing redundant internal normalizations [4, 15, 26].

Fractured floating-point units (FFPUs) [9] are fixed-point ISEs that accelerate software floating-point operations. FFPUs have a lower cost than floating-point operators and ISEs, but exhibit lower performance and higher energy consumption as a result.

8. Conclusion and Future Work

Customizable embedded processors can provide real-time, DTW in power-constrained environments, such as body area networks that perform physiological monitoring. The ISEs introduced in this paper achieve a speedup of 4.87x and a 78% reduction in energy consumption over software floating-point, and a 1.42x speedup and 35% reduction in energy compared to an FPU.

The ISE identification and selection algorithm that we used [21] did not choose good floating-point ISE candidates. We believe that new floating-point aware ISE identification and synthesis methods are needed, and we plan to investigate them in the future. We also plan to study the use of fixed-point representations for DTW, and dynamic conversion between fixed- and floating-point in response to time series data characteristics. Based on our analysis, the normalization step necessitates the use of the double-precision floating-point representation, regardless of the precision of the input data. This restriction could be lifted if an alternative implementation of normalization could be developed that is friendly to fixed-point numerical representations, even if some precision is lost during the process.

Sensor data in real-world applications are generally low precision fixed point, e.g., 8- or 12-bit. If we can reduce the precision of the intermediate normalized data, then we could ideally switch from a 32-bit RISC processor to an 8- or 16-bit microcontroller. This would further reduce the cost of the system.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grants CNS-1035603 and IIS-1161997. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

REFERENCES

[1] Ball, T., and Larus, J. R., "Optimally profiling and tracing programs," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 4, July, 1994, pp. 1319-1360. DOI= http://dx.doi.org/10.1145/183432.183527

[2] Chong, Y. J. and Parameswaran, S., "Custom floating-point unit generation for embedded systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, May, 2009, pp. 638-650. DOI= http://dx.doi.org/10.1109/TCAD.2009.2013999

[3] Dally, W. J., et al., "Efficient embedded computing," *Computer*, vol. 41, no. 7, July 2008, pp. 27-32. DOI= http://dx.doi.org/10.1109/MC.2008.224

[4] de Dinechin, F., and Pasca, B., "Designing custom arithmetic data paths with FloPoCo," *IEEE Design and Test of Computers*, vol. 28, no. 4, July-Aug. 2011, pp. 18-27. DOI= http://dx.doi.org/10.1109/MDT.2011.44

[5] Ding, H., et al., "Querying and mining of time series data: experimental comparison of representations and distance measures," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, Aug. 2008, pp. 1542-1552. URL= http://dl.acm.org/ citation.cfm?id=1454226

[6] Evans, D., "The Internet of Things: How the Next Evolution of The Internet Is Changing Everything," Cisco Internet Business Solutions Group. White Paper. April, 2011. URL= http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

[7] Fu, A. W-C., et al., "Scaling and time warping in time series querying," *International Journal on Very Large Data Bases*, vol. 17, no. 4, July, 2008, pp. 899-921. DOI= http://dx.doi.org/10.1007/s00778-006-0040-z

[8] Hameed, R., et al., "Understanding sources of inefficiency in general-purpose chips," *Comm. ACM*, vol. 54, no. 10, Oct. 2011, pp. 85-93. DOI= http://dx.doi.org/10.1145/2001269. 2001291

[9] Hockert, N., and Compton, K., "Improving floating-point performance in less area: fractured floating point units (FFPUs)," *Journal of Signal Processing Systems*, vol. 67, no. 1, April, 2012, pp. 31-46. DOI= http://dx.doi.org/10.1007/ s11265-010-0561-y

[10] Iwata, T., et al., "A speech recognition processor," IEEE Int. Solid-State Circuits Conf. (ISSCC '83), pp. 120-121, Feb. 23-25, 1983, DOI= http://dx.doi.org/10.1109/ISSCC.1983. 1156535

[11] Keogh, E., and Kasetty, S., "On the need for time series data mining benchmarks: a survey and empirical demonstration," *Journal of Data Mining and Knowledge Discovery*, vol. 7, no. 4, Oct. 2003, pp. 349-371. DOI= http://dx.doi.org/ 10.1023/A:1024988512476

[12] Keogh, E., et al., "Supporting exact indexing of arbitrarily rotated shapes and periodic time series under Euclidean and warping distance measures," *International Journal on Very Large Data Bases*, vol. 18, no. 3, June, 2009, pp. 611-630. DOI= http://dx.doi.org/10.1007/s00778-008-0111-4

[13] E. Keogh, et al., "The UCR Time Series Classification/Clustering Homepage," URL=http://www.cs.ucr.edu/~eamonn/time_series_data/

[14] Kim, S-W., Park, S., and Chu, W. W., "An index-based approach for similarity search supporting time warping in large sequence databases," 17th Int. Conf. Data Engr. (ICDE '01), pp. 607-614, Apr. 2-6, 2001, DOI= http://dx.doi.org/10.1109/ICDE.2001.914875

[15] Langhammer, M., "Floating-point datapath synthesis for FPGAs," Int. Conf. Field Programmable Logic and Applications (FPL '08), pp. 355-360, Sep. 8-10, 2008, DOI= http://dx.doi.org/10.1109/FPL.2008.4629963

[16] Lemire, D., "Faster retrieval with a two-pass dynamic-time-warping lower bound," *Pattern Recognition*, vol. 42, no. 9, Sep. 2009, pp. 2169-2180. DOI= http://dx.doi.org/10.1016/j.patcog.2008.11.030

[17] Loftian, R., and Jafari, R., "An ultra-low power hardware accelerator architecture for wearable computers using dynamic time warping," Design Automation and Test in Europe (DATE '13), pp. 913-916, Mar. 18-22, 2013, DOI= http://dx.doi.org/10.7873/DATE.2013.192

[18] Loftian, R., and Jafari, R., "A low power wake-up circuitry based on dynamic time warping for body sensor networks," Int. Conf. Body Sensor Networks (BSN '11), pp. 83-88, May 23-25, 2011, DOI= http://dx.doi.org/10.1109/BSN.2011.43

[19] Lowy, M., et al., "An architecture for a speech recognition system," IEEE Int. Solid-State Circuits Conf. (ISSCC '83), pp. 118-119, Feb. 23-25, 1983, DOI= http://dx.doi.org/10.1109/ISSCC.1983.1156528

[20] Owen, R. E., "A VLSI dynamic time warp processor for connected and isolated word speech recognition," IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP '85), pp. 985-988, Apr. 26-29, 1985, DOI= http://dx.doi.org/10.1109/ICASSP.1985.1168159

[21] Pozzi, L., Atasu, K., and Ienne, P., "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. CAD of Integrated Circuits and Systems*, vol. 25, no. 7, July 2006, pp. 1209-1229. DOI= http://dx.doi.org/10.1109/TCAD.2005.855950

[22] Quenot, G., et al., "A dynamic time warp VLSI processor for continuous speech recognition," IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP '86), pp. 1549-1552, Apr. 7-11, 1988, DOI= http://dx.doi.org/10.1109/ ICASSP.1986.1168945

[23] Rakthanmanon, T., et al., "Searching and mining trillions of time series subsequences under dynamic time warping," 18th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD '12), pp. 262-270, Aug. 12-16, 2012 DOI= http://dx.doi.org/10.1145/2339530.2339576

[24] Sakoe, H., and Chiba, S. "Dynamic programming optimization for spoken word recognition," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, Feb. 1978, pp. 43-49, DOI= http://dx.doi.org/10.1109/ TASSP.1978.1163055

[25] Sart, D., et al., "Accelerating dynamic time warping subsequence search with GPUs and FPGAs," 10th IEEE Int. Conf. Data Mining (ICDM '10), pp. 1001-1006, Dec. 13-17, 2010, DOI= http://dx.doi.org/10.1109/ICDM.2010.21

[26] Sethia, A., et al., "A customized processor for energy efficient scientific computing," *IEEE Trans. Computers*, vol. 61, no. 12, Dec. 2012, pp. 1711-1723. DOI= http://dx.doi.org/10.1109/TC.2012.144

[27] Wang, Z., et al., "Accelerating subsequence similarity search based on dynamic time warping distance with FPGA," ACM/SIGDA Int. Symp. Field Programmable Gate Arrays (FPGA '13), pp. 53-62, Feb. 11-13, 2013, DOI= http://dx.doi.org/10.1145/2435264.2435277

[28] Xilinx, "Virtex-6 User Guide Xilinx Corporation," 2010.

[29] Xilinx, "MicroBlaze Processor Reference Guide," October 2010.

[30] Xilinx, "LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)," April 2010.

[31] Xilinx, "Xilinx Power Tools Tutorial UG733," July 25 2010.