Accelerated Clock Drift Estimation for High-Precision Wireless Time-Synchronization

Andreas Engel and Andreas Koch
TU Darmstadt
Embedded Systems and Applications Group
Hochschulstraße 10, 64289 Darmstadt, Germany
{engel, koch}@esa.cs.tu-darmstadt.de

***Abstract*—Time synchronization is essential for most Wireless Sensor Network (WSN) applications and the required precision increases with the sampling- and communication rates. These algorithms require a significant amount of computational effort, especially for the clock drift estimation. In this paper, an improved Rolling Linear Regression (RLR) is proposed. By performing a coordinate transformation on each update of the regression table, the required arithmetic operates on smaller absolute numbers without loss of accuracy. On an 8 bit microcontroller (MCU), the improved algorithm is executed at least 22 % faster than the conventional Linear Regression (LR) algorithm. In addition, a hardware-accelerated implementation of the improved RLR is proposed, which further reduces the time spent for LR by another 66 % on the heterogeneous Hardware-Accelerated Low Power Mote (HaLoMote). Finally, the influence of the regression table size and the synchronization period on the accuracy of a multi-hop time synchronization protocol is investigated. For a 10 s synchronization period, the synchronization inaccuracy can be kept below 1 µs even at the fifth hop.**

I. Introduction

Time synchronization is required in WSN applications mainly for two reasons. First, data sampled from spatially distributed sensors cannot be properly interpreted without knowledge of the exact sampling time. The acceptable uncertainty is typically a small percentage of the application's sampling period, which may range from several seconds in environmental monitoring [11] down to several milliseconds in vibration-based structural health monitoring [9] or even below in acoustic localization applications [3]. Thus, synchronization protocols with an accuracy of a few microseconds are required for the more demanding applications.

A second reason for precise synchronization derives from the large power consumption of the radio transceiver idly waiting for incoming messages. If sender and receiver are synchronized, the radio protocol can define short periods of time in which transmissions can be initiated and received, thus limiting idle listening.

The local time at a sensor node a at a certain absolute time t is represented as a timestamp $a(t) \coloneqq f_a \cdot (t - t_{a,0})$, which is the value of an oscillator driven counter with oscillation frequency f_a and start-up time $t_{a,0}$. If two nodes a and b exchanged their timestamps $(a(t_e), b(t_e))$ at a time t_e, then node b can calculate the timestamp of node a at any subsequent time $t_s > t_e$ as

$$\begin{aligned} a(t_s) &= a(t_e) + f_a \cdot (t_s - t_e) \\ &= a(t_e) + \frac{f_a}{f_b}(b(t_s) - b(t_e)) \\ &= b(t_s) + a(t_e) - b(t_e) + \frac{f_a - f_b}{f_b}(b(t_s) - b(t_e)) \quad (1) \\ &= b(t_s) + \underbrace{a(t_e) - b(t_e)}_{\text{offset compensation}} + \underbrace{(f_a - f_b) \cdot (t_s - t_e)}_{\text{drift compensation}} \end{aligned}$$

Even if both nodes run at the same nominal frequency, the temperature and voltage dependency of the oscillators result in small relative frequency deviations $\frac{f_a - f_b}{f_b}$, typically in the range of a few part per million (ppm), as shown in Figure 3. Manufacturers specify the maximum frequency uncertainty over the entire operating temperature range at even higher values (e.g., ±40 ppm for the TI CC2530). Without explicit drift compensation, the timestamps used for offset compensation have to be exchanged at a rate of at least $(f_a - f_b)/(\Delta t \cdot f_b)$ to keep the synchronization uncertainty below Δt. Even if the timestamp exchange could be piggybacked onto regular application payload, the resulting additional communication demand is undesirable for small Δt.

To actually perform the drift compensation, node b has to estimate its clock drift relative to node a. This can be achieved by calculating the LR on the last $n \geqslant 2$ exchanged timestamp pairs $(a(t_{e,i}), b(t_{e,i}))_{i=1}^{n} =: (a_i, b_i)_{i=1}^{n}$ as

$$\frac{f_a}{f_b} = \frac{\Delta a}{\Delta b} = \frac{\sum_{i=1}^{n}(n \cdot a_i - \mathrm{sa})(n \cdot b_i - \mathrm{sb})}{\sum_{i=1}^{n}(n \cdot b_i - \mathrm{sb})^2} \quad (2)$$

with $\mathrm{sa} \coloneqq \sum_{i=1}^{n} a_i$ and $\mathrm{sb} \coloneqq \sum_{i=1}^{n} b_i$. Instead of performing the whole regression in $\mathcal{O}(n)$ for every newly exchanged timestamp pair, the RLR [13]

$$\frac{f_a}{f_b} = \frac{\Delta a}{\Delta b} = \frac{n \cdot \mathrm{sab} - \mathrm{sa} \cdot \mathrm{sb}}{n \cdot \mathrm{sbb} - \mathrm{sb} \cdot \mathrm{sb}} \quad (3)$$

can be calculated in $\mathcal{O}(1)$ by updating the rolling sums sa, sb, $\mathrm{sbb} \coloneqq \sum_{i=1}^{n} b_i b_i$, and $\mathrm{sab} \coloneqq \sum_{i=1}^{n} a_i b_i$.

The dominating arithmetic operations required in Equations 2 and 3 are the multiplications, which are applied to full timestamps (e.g., $a_i \cdot b_i$) or even sum of timestamps (e.g., $\mathrm{sa} \cdot \mathrm{sb}$). As high precision synchronization typically requires large timestamps (e.g., 48 bit to represent 9 years with 1 µs

resolution), considerable computational effort has to be spent even for the RLR algorithm.

Therefore, an improved RLR algorithm and its implementation on a software processor and a hardware-accelerator are the main contribution of this work. Furthermore, reasonable synchronization intervals and regression table sizes are investigated in a multi-hop network topology.

The remainder of this work is structured as follows. Section II summarizes current research related to clock drift compensation and efficient LR implementations. Afterwards, the improved RLR algorithm is proposed in Section III-A and its integration into a multi-hop radio stack is described in Section III-B. Various software- and hardware implementation are detailed in Sections III-C and III-D. Their resource requirements are compared against each other in Section IV-B after deriving reasonable regression settings from the achievable synchronization accuracy in Section IV-A. The work is concluded in Section V.

II. Related work

A large variety of wireless time synchronization protocols have been proposed in the last decade [8], and only a few of the described clock drift estimators are not based on LR. The R^4Syn protocol [7] uses a Maximum-Likelihood-Estimator that introduces the same computational complexity as Equation 3. In contrast, the Kalman filter used in [2] can be executed 10 % faster than a LR with a table size of two. However, for synchronization periods up to 10 s, the average synchronization error of the Kalman filter proved to be larger than for the LR based clock drift estimator.

The Flooding Time Synchronization Procotol (FTSP) [12] has become one of the most popular reference implementations for high precision synchronization protocols. It performs the clock drift compensation with a LR over the last 8 synchronization points without justifying the selected regression table size. At a 30 s synchronization period, FTSP achieves an average accuracy of 1.5 µs. While no details about the regression implementation or its resource requirements are given in [12], publicly available FTSP implementations (e.g., provided as TinyOS module) are based on the simple LR described in Equation 2.

Several improvements of the FTSP have been suggested. In [4], a temperature dependent correction factor is introduced to respond faster to temperature induced clock drift variations. The authors report an average accuracy of 1.1 µs after a 20 K temperature variation at a single node. The Recursive Time Synchronization Procotol (RTSP) [1] is also based on FTSP, but it uses automatic Medium Access Control (MAC) timestamping, i.e., the timestamps corresponding to the transmission or reception of the IEEE 802.15.4 start of frame delimiter (SFD) are captured by the radio transceiver without software intervention. Furthermore, RTSP dynamically adjusts the synchronization period depending on the current clock drift and accuracy at each node. The average accuracy was improved to 0.3 µs, although restricting the regression table to two entries to simplify the computational effort for the clock drift estimation.

The slope calculated by Equation 2 and 3 can be described as the ratio between the covariance of a and b and the variance of b. Thus, efficient formulations of the rolling (co-)variance directly result in an efficient RLR algorithm. The invariance of the (co-)variance against translations (i.e., $\mathrm{var}(\boldsymbol{b} + \Delta b) = \mathrm{var}(\boldsymbol{b})$) has been used to improve the numerical stability of the algorithms in closed form (i.e., without sliding windows) [5]. To the best of our knowledge, the application of a translation to the rolling (co-)variance and its impact on the computational requirements of the algorithms has not been described before.

While the LR calculation is the most compute-intense part of the drift compensation, in context of the entire WSN operation, the time-synchronization procedure requires only little computational effort, as it is executed only infrequently. However, LR has also been suggested for use in more demanding applications, such as RSSI-based node localization [14], or predictive data reduction [6]. It is thus worthwhile to examine the acceleration of the LR algorithm on resource-constrained embedded systems.

III. Rolling Linear Regression with Coordinate Transformation

A. Description of the Coordinate Transformation

As stated in Section I, the computational complexity of the RLR algorithm (Equation 3) is dominated by the multiplications that have to be applied to full timestamps and sum of timestamps. Let $(a_0, b_0), \ldots (a_p, b_p)$ be the timestamp pairs exchanged between node a and b at a certain point in time. The coordinate transformation translates these synchronization points into

$$(\underline{a}_{p,k}, \underline{b}_{p,k}) \coloneqq (a_p - a_k, b_p - b_k) \qquad \forall k \leqslant p \tag{4}$$

such that the latest exchanged timestamp pair is mapped to the origin of the coordinate system. This translation does not effect the slope calculated by the LR, but it limits the size of the numerical data to be processed to $n \cdot \Delta c$, where n is the regression table size and Δc is the maximum number of oscillator cycles between two timestamp exchanges. The arithmetic operations for the Rolling Linear Regression with Coordinate Transformation (RLRCT) can thus be implemented more efficiently than without applying the coordinate transformation.

The following formulas are stated only for the a values, but the same statements hold for the b values. To apply the proposed transformation to the rolling sums of the RLR with minimal effort, the regression table buffers the translation steps

$$\mathrm{da}_k \coloneqq a_k - a_{k-1} \qquad \forall p - n < k \leqslant p \tag{5}$$

between the last n synchronization points. The absolute coordinates $\underline{a}_{p,k}$ can be recovered for $p - n \leqslant k \leqslant p$ by accumulating

the translation steps:

$$\sum_{i=k+1}^{p} \mathrm{da}_i \overset{(5)}{=} \sum_{i=k+1}^{p} a_i - \sum_{i=k+1}^{p} a_{i-1}$$
$$= a_p + \sum_{i=k+1}^{p-1} a_i - \sum_{i=k}^{p-1} a_i = a_p - a_k \overset{(4)}{=} \underline{a}_{p,k} \quad (6)$$

At the next synchronization point (i.e., reception of a_{p+1}), the coordinate transformation (i.e., translation by da_{p+1}) must be applied to all coordinates, as

$$\underline{a}_{p+1,k} \overset{(6)}{=} \sum_{i=k+1}^{p+1} \mathrm{da}_i = \mathrm{da}_{p+1} + \sum_{i=k+1}^{p} \mathrm{da}_i \overset{(6)}{=} \mathrm{da}_{p+1} + \underline{a}_{p,k} \quad (7)$$

As the RLR does in Equation 3, the RLRCT also relies on updating the rolling sum

$$\mathrm{sa}_p \coloneqq \sum_{k=p-n+1}^{p} \underline{a}_{p,k} \quad (8)$$
$$= \underbrace{\underline{a}_{p,p}}_{0} - \underline{a}_{p,p-n} + \sum_{k=p-1-n+1}^{p-1} \underline{a}_{p,k}$$
$$\overset{(7)}{=} -\underline{a}_{p,p-n} + \sum_{k=p-1-n+1}^{p-1} \mathrm{da}_p + \underline{a}_{p-1,k}$$
$$\overset{(8)}{=} n \cdot \mathrm{da}_p - \underline{a}_{p,p-n} + \mathrm{sa}_{p-1} \quad (9)$$

Instead of explicitly calculating $\underline{a}_{p,p-n}$, another rolling sum is introduced as

$$\mathrm{sda}_p \coloneqq \underline{a}_{p,p-n} \overset{(6)}{=} \sum_{i=p-n+1}^{p} \mathrm{da}_i$$
$$= \mathrm{da}_p - \mathrm{da}_{p-n} + \mathrm{sda}_{p-1} \quad (10)$$

The sum of coordinate products

$$\mathrm{sab}_p \coloneqq \sum_{k=p-n+1}^{p} \underline{a}_{p,k} \cdot \underline{b}_{p,k}$$
$$= \underbrace{\underline{a}_{p,p} \cdot \underline{b}_{p,p}}_{0} - \underline{a}_{p,p-n} \cdot \underline{b}_{p,p-n} + \sum_{k=p-n}^{p-1} \underline{a}_{p,k} \cdot \underline{b}_{p,k}$$
$$\overset{(7,10)}{=} -\mathrm{sda}_p \cdot \mathrm{sdb}_p + \sum_{k=p-n}^{p-1} (\mathrm{da}_p + \underline{a}_{p-1,k})(\mathrm{db}_p + \underline{b}_{p-1,k})$$
$$\overset{(8)}{=} \mathrm{sab}_{p-1} + \mathrm{da}_p \cdot \mathrm{sb}_{p-1} + \mathrm{db}_p \cdot \mathrm{sa}_{p-1}$$
$$+ n \cdot \mathrm{da}_p \cdot \mathrm{db}_p - \mathrm{sda}_p \cdot \mathrm{sdb}_p \quad (11)$$

is not actually handled as rolling sum in RLRCT. Instead, the whole numerator of Equation 3 is accumulated as

$$\mathrm{num}_p \coloneqq n \cdot \mathrm{sab}_p - \mathrm{sa}_p \cdot \mathrm{sb}_p$$
$$\overset{(9)}{=} n \cdot \mathrm{sab}_p - (n \cdot \mathrm{da}_p - \mathrm{sda}_p + \mathrm{sa}_{p-1})$$
$$\cdot (n \cdot \mathrm{db}_p - \mathrm{sdb}_p + \mathrm{sb}_{p-1})$$
$$\overset{(11)}{=} n \cdot \mathrm{sab}_{p-1} - \mathrm{sa}_{p-1} \cdot \mathrm{sb}_{p-1} - (n+1) \cdot \mathrm{sda}_p \cdot \mathrm{sdb}_p$$
$$+ \mathrm{sda}_p(n \cdot \mathrm{db}_p + \mathrm{sb}_{p-1}) + \mathrm{sdb}_p(n \cdot \mathrm{da}_p + \mathrm{sa}_{p-1})$$
$$\overset{(9)}{=} \mathrm{num}_{p-1} - (n+1) \cdot \mathrm{sda}_p \cdot \mathrm{sdb}_p$$
$$+ \mathrm{sda}_p(\mathrm{sdb}_p + \mathrm{sb}_p) + \mathrm{sdb}_p(\mathrm{sda}_p + \mathrm{sa}_p)$$
$$= \mathrm{num}_{p-1} - (n-1) \cdot \mathrm{sda}_p \cdot \mathrm{sdb}_p$$
$$+ \mathrm{sda}_p \cdot \mathrm{sb}_p + \mathrm{sdb}_p \cdot \mathrm{sa}_p \quad (12)$$

Algorithm 1: Update procedure for Rolling Linear Regression with Coordinate Transformation

Input: Received synchronization point (a, b)
Input: Previously received synchronization point $(\hat{a}, \hat{b})$
Input: Accumulators sda, sdb, sa, sb, num, den
Input: Regression table T
Output: Slope represented as fraction of num and den

$\mathrm{da} \coloneqq a - \hat{a}$;
$\mathrm{db} \coloneqq b - \hat{b}$;
$\hat{a} \coloneqq a$;
$\hat{b} \coloneqq b$;
$(\hat{\mathrm{da}}, \hat{\mathrm{db}}) \leftarrow T \leftarrow (\mathrm{da}, \mathrm{db})$; // en/dequeue
$\mathrm{sda} \mathrel{+}= \mathrm{da} - \hat{\mathrm{da}}$; // Equ. 10
$\mathrm{sdb} \mathrel{+}= \mathrm{db} - \hat{\mathrm{db}}$;
$\mathrm{sa} \mathrel{+}= n \cdot \mathrm{da} - \mathrm{sda}$; // Equ. 9
$\mathrm{sb} \mathrel{+}= n \cdot \mathrm{db} - \mathrm{sdb}$;
$t \coloneqq (n-1) \cdot \mathrm{sdb}$;
$\mathrm{num} \mathrel{+}= \mathrm{sdb} \cdot \mathrm{sa} + \mathrm{sda} \cdot \mathrm{sb} - t \cdot \mathrm{sda}$; // Equ. 12
$\mathrm{den} \mathrel{+}= \mathrm{sdb} \cdot \mathrm{sb} \cdot 2 - t \cdot \mathrm{sdb}$; // Equ. 13

Algorithm 2: Update procedure for Rolling Linear Regression (without Coordinate Transformation)

Input: Received synchronization point (a, b)
Input: Accumulators sa, sb, sab, sbb
Input: Regression table T
Output: Slope represented as fraction of num and den

$\mathrm{ab} \coloneqq a \cdot b$;
$\mathrm{bb} \coloneqq b \cdot b$;
$(\hat{a}, \hat{b}, \hat{\mathrm{ab}}, \hat{\mathrm{bb}}) \leftarrow T \leftarrow (a, b, \mathrm{ab}, \mathrm{bb})$; // en/dequeue
$\mathrm{sa} \mathrel{+}= a - \hat{a}$;
$\mathrm{sb} \mathrel{+}= b - \hat{b}$;
$\mathrm{sab} \mathrel{+}= \mathrm{ab} - \hat{\mathrm{ab}}$;
$\mathrm{sbb} \mathrel{+}= \mathrm{bb} - \hat{\mathrm{bb}}$;
$\mathrm{num} \coloneqq n \cdot \mathrm{sab} - \mathrm{sa} \cdot \mathrm{sb}$; // Equ. 3
$\mathrm{den} \coloneqq n \cdot \mathrm{sbb} - \mathrm{sb} \cdot \mathrm{sb}$

The denominator of Equation 3 is accumulated accordingly:

$$\mathrm{den}_p = \mathrm{den}_{p-1} - (n-1) \cdot \mathrm{sdb}_p^2 + 2 \cdot \mathrm{sdb}_p \cdot \mathrm{sb}_p \qquad (13)$$

Algorithm 1 summarizes the RLRCT update procedure. It consists of 15 additions and nine multiplications (four of them with a small constant). As shown in Algorithm 2, it is assumed that the RLR buffers the additional values

$$(\mathrm{ab}_k, \mathrm{bb}_k)_{k=p-n+1}^{p} \coloneqq (a_k \cdot b_k, b_k \cdot b_k)_{k=p-n+1}^{p}$$

in its regression table to avoid the recomputation of ab_{p-n} and bb_{p-n} required in line 6 and 7. The RLR requires fewer operations (10 additions, six multiplications) than the RLRCT, but under certain circumstances, the latter can be implemented more efficiently, as will be shown in Section IV-B. Furthermore, the RLRCT requires less memory to represent the regression table.

B. Integration into a Wireless Synchronization Protocol

The heterogeneous HaLoMote [10] was chosen as the target platform for the RLRCT-based synchronization protocol. It incorporates an 8 bit MCU (TI CC2531 Radio System-On-Chip) that provides a 40 bit timer driven by a 32 MHz oscillator and supports automatic MAC timestamping.

As in the FTSP, a single node is selected as time reference, i.e., its local time represents the global time all other nodes try to synchronize with. As this paper focuses on the LR-based drift compensation, dynamic reselection of the reference node is not considered here further.

We propose to insert the synchronization layer between the MAC and the network layer. While violating the compliance with standardized protocols like Zigbee, this design ensures that all nodes get synchronized, even if they are just routing packets. The synchronization layer adds up to 6 B to the radio packet. The first Byte indicates whether or not the sender has already been synchronized to the reference, i.e., it is the reference or it received a sufficient number of timestamps to perform the offset and drift compensation between its local clock and the reference clock. If the sender is synchronized, a 5 B timestamp $a(t_{\mathrm{TX}})$ is appended to represent the senders assumption of the global time of the SFD transmission.

To calculate this timestamp, the sender first inserts the PHY and the MAC header and the first byte of the synchronization header into the radio buffer and initiates the transmission. After the SFD was transmitted at time t_{TX}, the sender calculates $a(t_{\mathrm{TX}})$ from the captured local time $b(t_{\mathrm{TX}})$. At the reference node, the local time is used as global time, while all other nodes apply Algorithm 3 to the local timestamp. This algorithm realizes Equation 1 based on the outcome of the clock drift compensation described in the previous section. More specifically, the ratio between num and den equals the ratio between the oscillator frequencies of the reference node and the local node, i.e., $\mathrm{num}/\mathrm{den} = f_a/f_b$. While Algorithm 3 could be simplified (i.e., $a \coloneqq \hat{a} + (t \cdot \mathrm{num})/\mathrm{den}$), the proposed formulation avoids large intermediate results by not multiplying the entire local timestamp difference with the large numerator. To avoid a radio buffer underrun, Algorithm 3 must be finished before the previous header data was transmitted (i.e., $8\,\mathrm{B}/250\,\mathrm{kbit/s} = 256\,\mu\mathrm{s}$).

Upon reception of the SFD of a radio packet at time t_{RX}, the receiver timer captures the local timestamp $b(t_{\mathrm{RX}})$. The difference between t_{RX} and t_{TX} results from the propagation delay between sender and receiver (i.e., 3.3 ns/m) and should not be significant for node distances of a few meters. In practice, however, $t_{\mathrm{RX}} - t_{\mathrm{TX}} = (3.51 \pm 0.04)\,\mu\mathrm{s}$ was measured for 210 transmissions by observing the automatically generatable SFD sampling events at five different receiver nodes located less than 10 cm away from the transmitter. A similar systemic offset was observed by [1]. It is compensated for by subtracting $\Delta_{\mathrm{SFD}} \coloneqq 3.51\,\mu\mathrm{s} \cdot 32\,\mathrm{MHz} = 112$ from the local timestamp before updating the LR parameters with Algorithm 1.

In this manner, the synchronization information is flooded over multiple hops into the network. To perform a certain action (e.g., sensor sampling) simultaneously, all synchronized nodes must agree on a certain global time, convert this time into their local time by applying Algorithm 4, and configure their timers to generate an interrupt at the calculated local time.

C. MCU Implementation

To analyze the benefits of the proposed RLRCT, Algorithm 1 and 2 as well as Equation 2, and an optimized version for $n = 2$ was implemented on the CC2531 MCU. Furthermore, Algorithm 3 and 4 were implemented as required by the synchronization layer.

The 8051-ISA compliant compilers (e.g., SDCC, IAR or Keil) support up to 64 bit integer arithmetic operations. However, this is not sufficient for the LR implementation with 40 bit timestamps. Furthermore, the default operator granularities of 8 bit, 16 bit, 32 bit and 64 bit are not efficient, as the main ben-

Algorithm 3: Conversion from local to global time

Input: Local timestamp b
Input: Last exchanged synchronization point $(\hat{a}, \hat{b})$
Input: Clock drift represented as num and den
Output: Global timestamp a corresponding to b

1 $t \coloneqq b - \hat{b}$;
2 $a \coloneqq \hat{a} + t + (t \cdot (\mathrm{num} - \mathrm{den}))/\mathrm{den}$; // Equ. 1

Algorithm 4: Conversion from global to local time

Input: Global timestamp a
Input: Last exchanged synchronization point $(\hat{a}, \hat{b})$
Input: Clock drift represented as num and den
Output: Local timestamp b corresponding to a

1 $t \coloneqq a - \hat{a}$;
2 $b \coloneqq \hat{b} + t - (t \cdot (\mathrm{num} - \mathrm{den}))/\mathrm{num}$; // Equ. 1

efits of the RLRCT algorithm arise from optimized narrowed sizes.

Thus, the required operators where implemented in 8051 assembler with up to 96 bit wide representations at a granularity of 8 bit. All intermediate variables and accumulators are located in the directly addressable memory and the assembler routines operate on that memory without accessing the stack. The implementation is thus optimized for execution speed at the expense of a large code memory footprint.

D. FPGA Implementation

To further speed-up the LR and the timestamp conversion, a dedicated hardware-accelerator was implemented on the target platform's Field Programmable Gate Array (FPGA). As described in [10], the HaLoMote supports hardware (HW) kernels that can be controlled by the MCU, i.e., providing inputs, starting and observing the execution and reading back the results. On the FPGA, arithmetic operations of arbitrary width can be implemented with a granularity of 1 bit and multiple operations can be executed in parallel. Thus, the RLRCT is well suited for hardware-acceleration.

Three HW kernels were implemented for Algorithm 1, 3 and 4. As the synchronization protocol does not require parallel execution of these, they share some control and operator logic to reduce the required FPGA resources.

Three different architectures were implemented to trade off execution speed against resource requirements. All architectures buffer the regression table in Block RAM (BRAM) and perform the slope division (i.e., line 2 of Algorithm 3 and 4) as sequential shift-subtract steps, while the additions and multiplications are executed as single-cycle combinatorial logic.

In the *Full Parallel* architecture, every arithmetic operation of Algorithm 1 is implemented with dedicated logic. The data dependencies result in a seven cycle datapath for the LR. In the *Single MAC* architecture, a single multiply-accumulate operator is implemented and the registers buffering the accumulators and intermediate results are sequentially multiplexed to this operator, resulting in a 21 cycle computation for the LR.

Finally, the *µArchitecture* shown in Figure 1 buffers all accumulators next to the regression table in a dual-port BRAM. To efficiently utilize the BRAM, the larger accumulators (i.e., num and den) occupy two memory locations. Three registers ($T0$ to $T2$) can be fed with data from the BRAM, inputs of the HW kernels (i.e., timestamps), results from previous operations, or some shifted values required for the slope division. Two out of these registers are selected as inputs (A and B) for the arithmetic unit, which can calculate an addition, a subtraction, a multiplication, an absolute difference, or a conditional operation. The latter is either an addition or a subtraction depending on a flag in one of the input registers. To simplify the control logic of the *µArchitecture*, the pipeline delay resulting from BRAM and register access is handled by the instructions coded in dedicated logic. Furthermore, predication logic is used to implement the sequential divisions without branches.

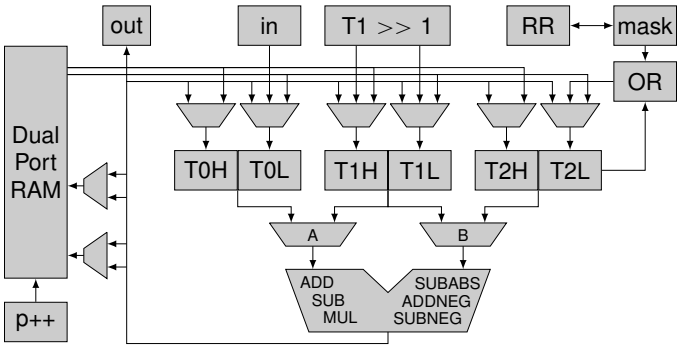

Fig. 1. HW Kernel (µArchitecture) for RLRCT

IV. Evaluation

In this section the synchronization accuracy of the protocol described in Section III-B is analyzed for different configurations of the synchronization period and the regression table size. These results are not effected by the proposed RLRCT implementation, but by the actually utilized WSN hardware (e.g., oscillator stability, timestamp accuracy and radio-triggered timestamp capturing) as well as the network topology and other environmental conditions (e.g., temperature stability). After restricting the design space to reasonable configurations, the resource requirements of the various LR implementations are compared against each other.

A. Achievable synchronization accuracy

The network setup shown in Figure 2 was used to evaluate the achievable synchronization accuracy. Five receiver nodes are located in the broadcast range of a gateway node. The latter dumps measurement results to the PC over a serial connection and acts as the time reference in the synchronization protocol. All nodes are not more than 1 m apart from each other, so the propagation delay is smaller than 3 ns and can be ignored. Note that the synchronized nodes are required to be within the broadcast range of the gateway only to simplify the test of the synchronization accuracy. The synchronization protocol itself relies on exchanging timestamps along the linear multi-hop chain and does not require any broadcasts.

Once per second, the gateway initiates a new measurement consisting of two phases. The *test* phase is started by a broadcast from the gateway to all other nodes (message 1 in Figure 2). The gateway captures the broadcast transmission time $a(t_{\mathrm{TX}})$. Each receiving node i captures its local broadcast reception

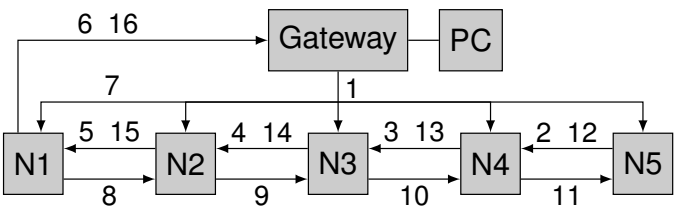

Fig. 2. Network setup for timestamp capturing: *test* messages (1-6) and *update* messages (7-16)

time $b_i(t_{\mathrm{RX}})$, derives the local broadcast transmission time as $b_i(t_{\mathrm{TX}}) \approx b_i(t_{\mathrm{RX}}) - \Delta_{\mathrm{SFD}}$, and calculates the corresponding assumed global time $a_i(t_{\mathrm{TX}})$ using offset and drift compensation. These timestamps are reported back to the gateway in a linear chain starting at node 5 (messages 2 to 6). The actual synchronization accuracy A_i and the actual clock drift D_i of node i relative to the reference are derived from the received timestamps as

$$A_i(t_{\mathrm{TX}}) \coloneqq a_i(t_{\mathrm{TX}}) - a(t_{\mathrm{TX}})$$
$$D_i(t_{\mathrm{TX}}) \coloneqq \frac{a(t_{\mathrm{TX}}) - a(t_{\mathrm{TX}} - 1\,\mathrm{s})}{b_i(t_{\mathrm{TX}}) - b_i(t_{\mathrm{TX}} - 1\,\mathrm{s})} - 1$$

where $t_{\mathrm{TX}} - 1\,\mathrm{s}$ denotes the broadcast event of the previous *test* phase.

Immediately after the *test* phase, the *update* phase is started by the gateway transmitting a unicast request (7) to node 1, which is forwarded in a linear chain (message 8 to 11) to node 5. Each node captures its local request reception time and the subsequent local request transmission time and reports them back to the gateway (message 12 to 16). The actual synchronization update (i.e., insertion of the exchanged timestamps into the regression table and update of the slope parameters) is only performed in every tenth *update* phase. Thus, the effective synchronization period of the whole network is 10 s. However, the timestamps captured during the *update* phase allow for an offline simulation of the whole synchronization protocol. This simplifies the analysis of the impact of the regression table size and the synchronization period on the synchronization accuracy. All data presented below for a regression table size other than two, or a synchronization period other than 10 s, result from these simulations.

Figure 3 shows the clock drift of the receiver nodes relative to the reference clock. For the first 55 min, all nodes were kept at a fixed temperature of about 21 °C. Under this condition, the standard deviation of the drift amounts to 0.1 ppm at all nodes.

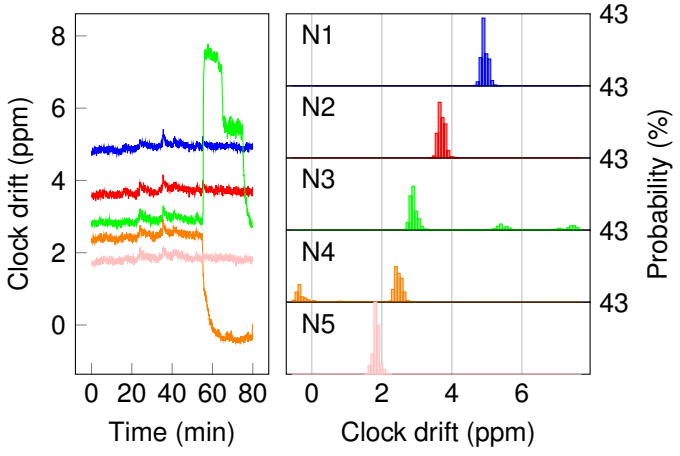

Fig. 3. Observed clock drift relative to gateway under temperature variation (0 min to 55 min: 21 °C, 55 min to 80 min: 7 °C at N4, 55 min to 65 min: 52 °C at N3, 65 min to 75 min: 40 °C at N3)

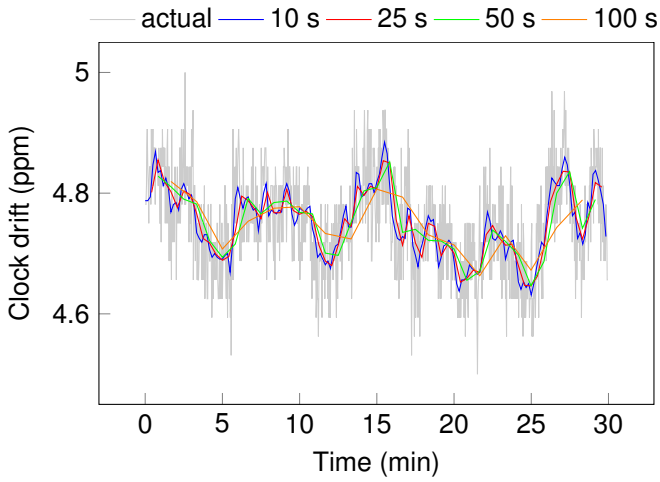

Fig. 4. Impact of the synchronization period on estimated clock drift at node 1 (regression table size = 2)

Afterwards, node 3 was cooled down to 7 °C resulting in a drift drop of about 3 ppm. In the same time, node 4 was heated up to 52 °C before cooling down to 40 °C again. This resulted in a drift step of 4.5 ppm and 2.5 ppm respectively. The insights gained from this measurement are twofold. First, without clock drift compensation, node 1 would have to exchange timestamps with the reference node at least every 200 ms to keep the synchronization inaccuracy below 1 µs. Thus, high precision time synchronization has to provide drift compensation to keep the communication overhead manageable. Second, even if the sensor nodes in typical WSN applications will not be exposed to sudden large temperature changes, a significant variation of the clock drift can be expected if the subset of nodes that are exposed to full sunlight is changing.

The influence of the node's supply voltage on the clock drift was also investigated as the supply of battery-powered sensor

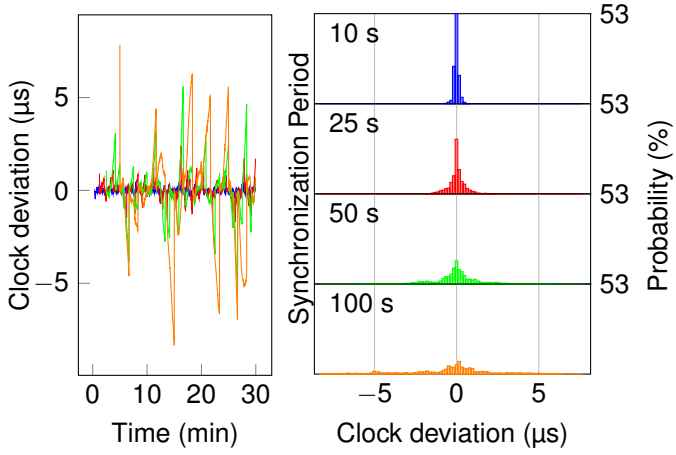

Fig. 5. Impact of the synchronization period on the synchronization accuracy at node 1 (regression table size = 2)

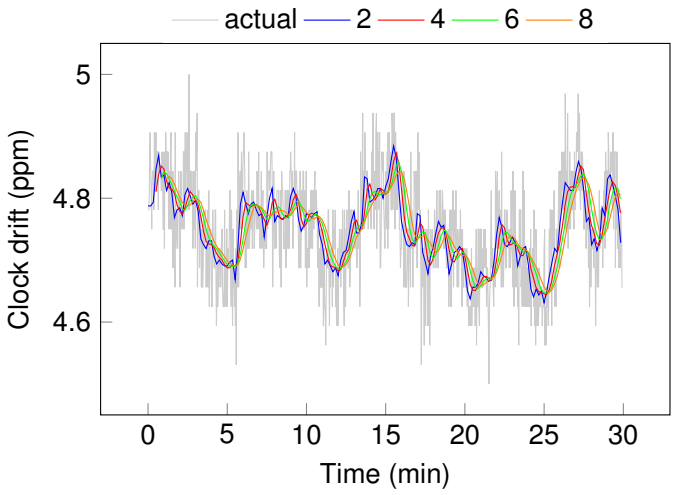

Fig. 6. Impact of the regression table size on estimated clock drift at node 1 (synchronization period = 10 s)

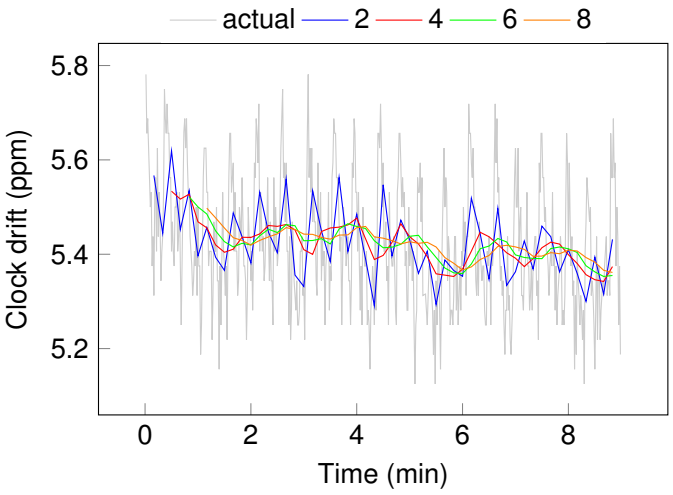

Fig. 8. Impact of the regression table size on estimated clock drift at node 3 (synchronization period = 10 s)

nodes can not be assumed to be stable. However, due to the voltage converters inside the MCU, the reduction of the supply voltage from 3 V to 2 V did not significantly influence the clock drift at the corresponding node.

Figure 4 shows the actual clock drift of node 1 relative to the reference (captured at 1 Hz as described above) and its assumption about its clock drift derived by LR with a synchronization period ranging from 10 s to 100 s and a fixed regression table size of two. As expected, the estimation becomes more inaccurate with longer synchronization periods. Figure 5 shows how this inaccuracy in drift estimation translates into synchronization inaccuracy. The maximum absolute clock deviations for the different synchronization periods are 0.7 µs, 2.4 µs, 5.7 µs and 8.3 µs respectively. In general, the appropriate synchronization period depends on the concrete accuracy demands of the application.

Figure 6 shows the actual clock drift of node 1 relative to the reference and its assumption about its clock drift derived by LR with a regression table size ranging from 2 to 8 elements at a fixed synchronization period of 10 s. Larger regression tables have an effect similar to the smoothing of the drift estimation occurring for larger synchronization periods. Indeed, larger regression tables actually do not improve the average synchronization accuracy, as shown in Figure 7. However, even for a table size of 8, the maximum absolute clock deviation does not exceed 1 µs.

The real benefit of larger regression tables becomes obvious only if the actual clock drift is more spiky, e.g., due to temporary temperature fluctuations, as shown in Figure 8. While the average clock deviation is not improved by the larger regression tables as shown in Figure 9, the maximum absolute error is reduced from 2.8 µs to 1.7 µs when choosing a table size of 8

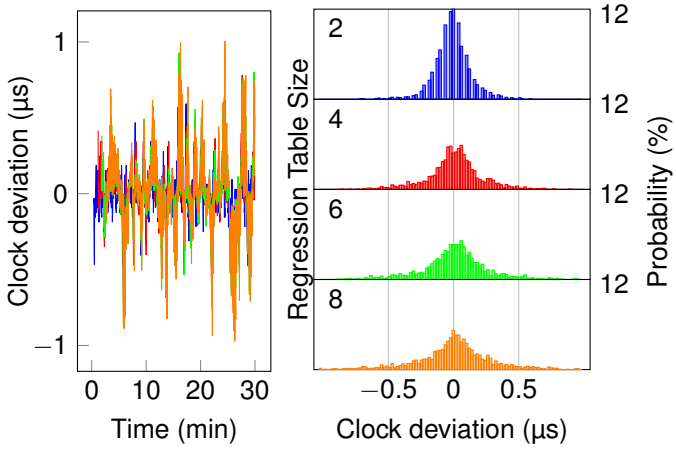

Fig. 7. Impact of the regression table size on the synchronization accuracy at node 1 (synchronization period = 10 s)

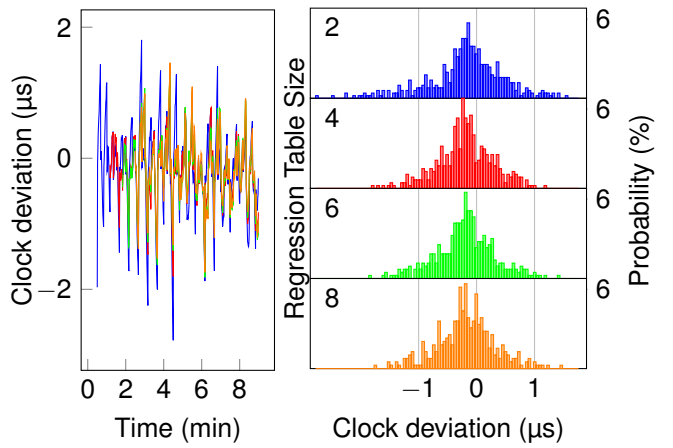

Fig. 9. Impact of the regression table size on the synchronization accuracy at node 3 (synchronization period = 10 s)

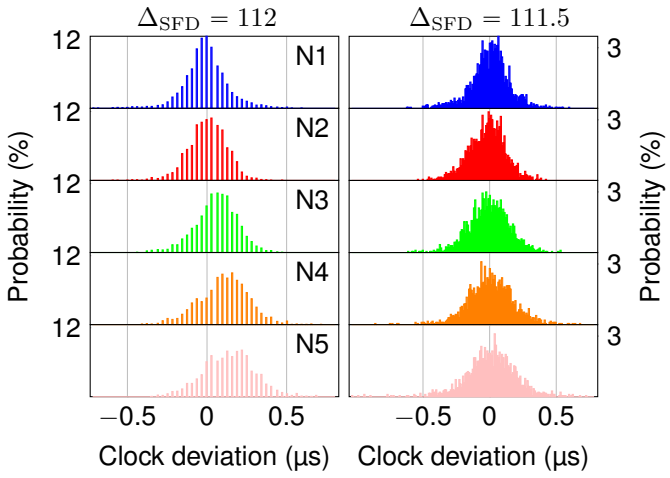

Fig. 10. Impact of the hop distance to the reference node on synchronization accuracy (synchronization period = 10 s, table size = 2)

instead of 2.

Finally, the synchronization accuracy over multiple hops is shown in Figure 10 (left). Even at the fifth hop, the maximum absolute clock deviation can be kept below 1 µs. However, the average deviation is increased by about 30 ns per hop. This might be caused by the fixed compensation time for the SFD delay as described in Section III-B. The accuracy of this compensation is limited by the timestamp resolution of $1/32\,\mathrm{MHz} = 31\,\mathrm{ns}$. When simulating the synchronization with $\Delta_{SFD} = 111.5$, the mean synchronization error is kept stable over multiple hops as shown in Figure 10 (right).

B. Resources required for Linear Regression

As shown in Figure 5, the synchronization period should not exceed 10 s to achieve a synchronization accuracy of 1 µs. At a timestamp resolution of $1/32\,\mathrm{MHz}$, the differences between successive timestamps inserted into the RLRCT algorithm (i.e., da and db in Algorithm 1) can thus be represented with 29 bit. Furthermore, regression tables with more than 8 entries do not improve the synchronization accuracy, as shown in the Figures 7 and 9.

The arithmetic operators required for the various LR implementations described in Section III-C and III-D were optimized based on these assumptions and compiled with the SDCC compiler (Version 3.4). The resulting execution times were measured with an oscilloscope and are shown in Figure 11. All $\mathcal{O}(1)$ implementations outperform the $\mathcal{O}(n)$ implementation even for $n = 2$. Furthermore, the proposed RLRCT is 22 % faster than the RLR implementation. For $n > 2$, the hardware accelerator outperforms the best software implementation by 66 %. Note that 95 % of the time required by the hardware accelerator is spent transferring the 80 bit timestamp pair from the MCU to the FPGA. Future versions of the HaLoMote will improve the inter-processor bandwidth. Nevertheless, if a regression table of size 2 satisfies the application accuracy requirements, the optimized software implementation (referred to as OPT in Figures 11 to 13) actually computes the fastest regression.

All software implementations of the time conversion require 118 µs for the offset and drift compensation. The hardware accelerator requires only 72 µs, including the inter-processor communication. As the time conversion has to be performed more frequently than the LR (e.g., once per sampling cycle), the sensor node benefits from the hardware accelerator even for the smallest regression tables.

Beyond execution time, memory is another limited resource on embedded systems. As shown in Figure 12, the proposed RLRCT clearly outperforms the RLR in terms of required Random Access Memory (RAM), as its regression table requires only $2n \cdot 29$ bit, while the RLR buffers $2n \cdot (40 + 80)$ bit. For regression tables larger than 12 entries, the RLRCT also requires less RAM than the LR implementation and is thus

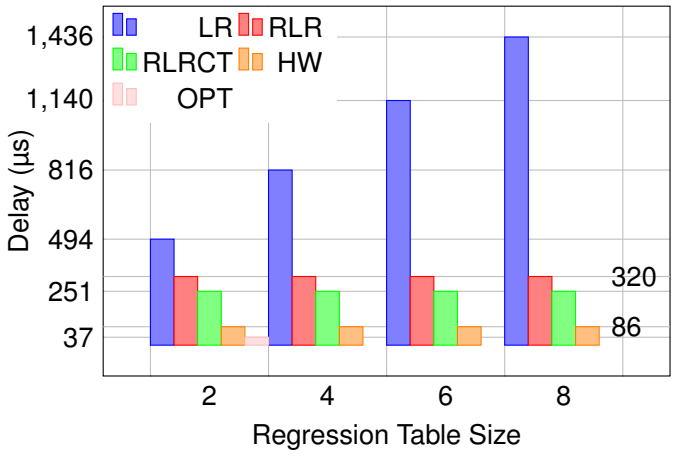

Fig. 11. Overall execution time for the LR implementations (LR = Equation 2, RLR = Algorithm 2, RLRCT = Algorithm 1, HW = Accelerated µArchitecture, OPT = Optimized for $n = 2$)

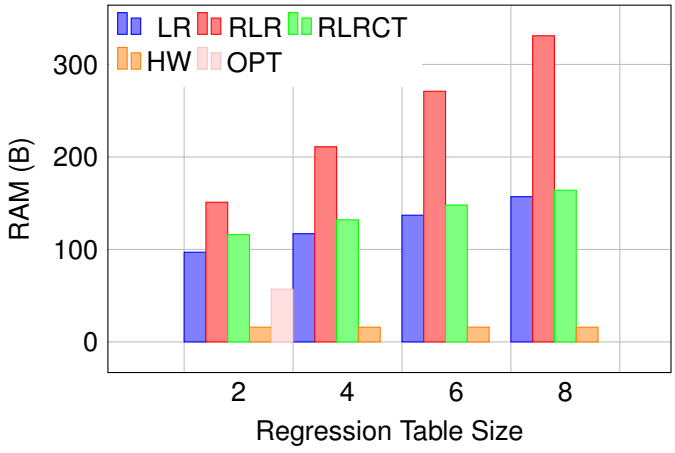

Fig. 12. MCU-RAM requirement of the LR implementations (LR = Equation 2, RLR = Algorithm 2, RLRCT = Algorithm 1, HW = Accelerated µArchitecture, OPT = Optimized for $n = 2$)

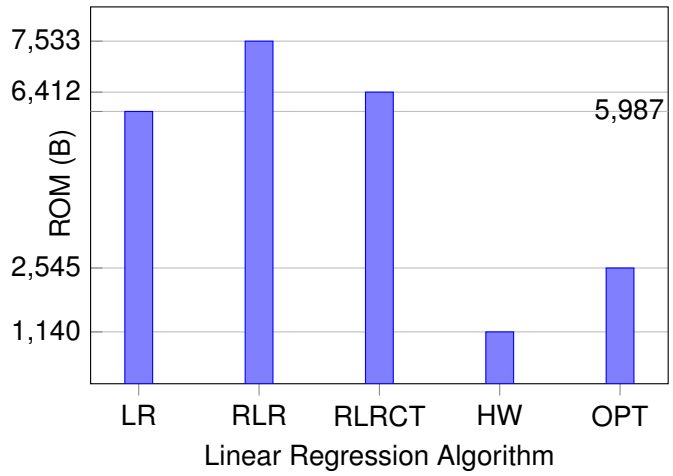

Fig. 13. MCU-ROM requirement of the LR implementations (LR = Equation 2, RLR = Algorithm 2, RLRCT = Algorithm 1, HW = Accelerated μArchitecture, OPT = Optimized for $n = 2$)

favourable in memory constrained systems.

As the assembler-based implementation of the arithmetic operations are optimized for execution speed, the instruction memory required by the different LR algorithms is relatively large. As shown in Figure 13, RLRCT requires 15 % less Read Only Memory (ROM) for storing instructions than RLR, although the RLRCT has to perform more arithmetic operations.

Finally, the three different hardware-accelerator architectures were synthesized for the Microsemi Igloo M1AGL1000 FPGA with Synplify Pro (ME I-2014.03M-SP1). The resulting resource requirements are summarized in Table I. The *Full Parallel* architecture does not fit on the target device, so its usage is restricted to larger FPGAs. Both other architectures occupy slightly more than half of the FPGA logic resources, mainly due to the usage of combinatorial multipliers.

V. Conclusion and Future Work

In this work, an improved Rolling Linear Regression (RLR) algorithm was proposed to reduce the resources required for time drift compensation, which is a key part of state-of-the-art high precision wireless synchronization protocols such as FTSP. By applying a coordinate transformation, the RLRCT algorithm operates on smaller absolute numbers and can thus be executed faster. For the specific requirements derived from a multi-hop synchronization protocol with less than 1 μs clock deviation at the fifth hop, RLRCT is 22 % faster than RLR while requiring less RAM and ROM. Compared to the LR implementation with linear runtime-complexity as used by publicly available FTSP implementations, RLRCT is executed 83 % faster. Furthermore, an RLRCT hardware accelerator was proposed for the heterogeneous HaLoMote, which reduces the computation time by *another* 66 %.

Some improvements are envisioned for the proposed system. First, the SFD delay compensation should be performed with subtick precision by some kind of pulse-width modulation (e.g., alternating Δ_{SFD} and $\Delta_{\mathrm{SFD}}-1$ at an appropriate ratio). Second, the hardware accelerator could be improved to spend less time for inter-processor communication. The time thus gained could be used to implement sequential multiplications, which would reduce the required FPGA resources significantly. Furthermore, the benefits of the proposed implementation for other LR-based WSN applications such as RSSI-based localization should be investigated.

TABLE I
Impact of hardware accelerator architecture on area and performance on an M1AGL1000 FPGA

Implementation	Full Parallel	Single MAC	μArchitecture
Core Cells [%]	141	63	50
BRAM [%]	12	6	15
$f_{\max}$ [MHz]	8.7	7.1	8.5
Regression [cycles (μs)]	7 (0.8)	21 (3.0)	29 (3.4)
Conversion [cycles (μs)]	27 (3.1)	29 (4.1)	32 (3.8)

References

[1] M. Akhlaq and T. Sheltami, "Rtsp: An accurate and energy-efficient protocol for clock synchronization in wsns," *IEEE Trans. on Instrumentation and Measurement*, vol. 62, pp. 578–589, March 2013, 5.

[2] M. Aoun, A. Schoofs, and P. van der Stok, "Efficient time synchronization for wireless sensor networks in an industrial setting," in *Proc. of the 6th ACM Conf. on Embedded Network Sensor Systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008, pp. 419–420.

[3] S. Astapov, J. Ehala, and J.-S. Preden, "Collective acoustic localization in a network of dual channel low power devices," in *Proc. of the 21st Int. Conf. on Mixed Design of Integrated Circuits Systems (MIXDES),*, June 2014, pp. 430–435.

[4] J. Castillo-Secilla, J. Palomares, and J. Olivares, "Temperature-aware methodology for time synchronisation protocols in wireless sensor networks," *Electronics Letters*, vol. 49, no. 7, pp. 506–508, 2013.

[5] T. F. Chan, G. H. Golub, and R. J. Leveque, "Algorithms for computing the sample variance: Analysis and recommendations," *The American Statistician*, vol. 37, no. 3, pp. 242–247, 1983.

[6] C. de Carvalho, D. Gomes, J. de Souza, and N. Agoulmine, "Multiple linear regression to improve prediction accuracy in wsn data reduction," in *7th Latin American Network Operations and Management Symposium (LANOMS)*, Oct 2011, pp. 1–8.

[7] D. Djenouri, "r^4syn : Relative referenceless receiver/receiver time synchronization in wireless sensor networks," *IEEE Signal Processing Letters*, vol. 19, no. 4, pp. 175 –178, April 2012.

[8] D. Djenouri and M. Bagaa, "Synchronization protocols and implementation issues in wireless sensor networks: A review," *IEEE Systems Journal*, vol. PP, no. 99, pp. 1–11, 2014.

[9] A. Engel, A. Friedmann, M. Koch, J. Rohlfing, T. Siebel, D. Mayer, and A. Koch, "Hardware-accelerated wireless sensor network for distributed structural health monitoring," in *Proceedings of the 2nd International Conference on System-Integrated Intelligence: Challenges for Product and Production Engineering*, 2014.

[10] A. Engel, B. Liebig, and A. Koch, "Energy-efficient heterogeneous reconfigurable sensor node for distributed structural health monitoring," in *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, October 2012, pp. 1–8.

[11] M. Lazarescu, "Design of a wsn platform for long-term environmental monitoring for iot applications," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, pp. 45–54, March 2013.

[12] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *Proc. of the 2nd Int. Conf. on Embedded networked sensor systems*, ser. SenSys '04. ACM, 2004, pp. 39–49.

[13] P. M. Neely, "Comparison of several algorithms for computation of means, standard deviations and correlation coefficients," *Commun. ACM*, vol. 9, no. 7, pp. 496–499, July 1966.

[14] F. Vanheel, J. Verhaevert, E. Laermans, I. Moerman, and P. Demeester, "Automated linear regression tools improve rssi wsn localization in multipath indoor environment," *EURASIP Journal on Wireless Communications and Networking*, no. 1, 2011.