However, as the complexity of the dataset increases, as well as the depth of the network topology, such as ResNet [55] on ImageNet [56], the accuracy degradation becomes more important and may not be negligible anymore [34], especially for critical applications such as autonomous driving. Therefore, it is not certain that network compression techniques and approximate computing are suitable for all applications.

B. SPIKE-BY-SPIKE NEURAL NETWORKS ACCELERATORS

Recently, Rotermund et al. demonstrated the feasibility of a neuromorphic SbS IP on a Xilinx Virtex 6 FPGA [16]. It provides a massively parallel architecture, optimized to reduce memory access and suitable for ASIC implementations. Nonetheless, this design is considerably resource-demanding if implemented as a full SbS network in today's embedded technology.

In Ref. [15], we presented a cross-platform accelerator framework for design exploration and testing of fully functional SbS network models in embedded systems. As a hardware/software (HW/SW) co-design solution, this framework offers a comprehensive high level embedded software API that allows the construction of scalable sequential SbS networks with configurable hardware acceleration. However, this design works entirely with standard floating-point arithmetic (IEEE 754). This represents a large memory footprint and inadequate computational cost for error-resilient applications on resource-limited devices. In this article, we will use this design exploration framework to investigate approximate computing for efficient deployment of deep SbS networks on resource-limited devices.

III. BACKGROUND

A. SPIKE-BY-SPIKE NEURAL NETWORKS

Technically, SbS is a spiking neural network approach based on a generative probabilistic model. It iteratively finds an estimate of its input probability distribution $p(s)$ (i.e. the probability of input node s to stochastically send a spike) by its latent variables via $r(s) = \sum_i h(i)W(s|i)$. where $\vec{h}$ is an inference population composed of a group of neurons that compete with each other. An inference population sees only the spikes s_t (i.e. the index identifying the input neuron s which generated that spike at time t) produced by its input neurons, not the underlying input probability distribution $p(s)$ itself. By counting the spikes arriving at a group of SbS neurons, $p(s)$ is estimated by $\hat{p}(s) = 1/T \sum_t \delta_{s,s^t}$ after T spikes have been observed in total. The goal is to generate an internal representation $r(s)$ from the string of incoming spikes s_t such that the negative logarithm of the likelihood $L = C - \sum_\mu \sum_s \hat{p}_\mu(s) log\,(r_\mu(s))$ is minimized. C is a constant which is independent of the internal representation $r_\mu(s)$ and μ denotes one input pattern from an ensemble of input patterns. Applying a multiplicative gradient descent method on L, an algorithm for iteratively updating $h_\mu(i)$ with

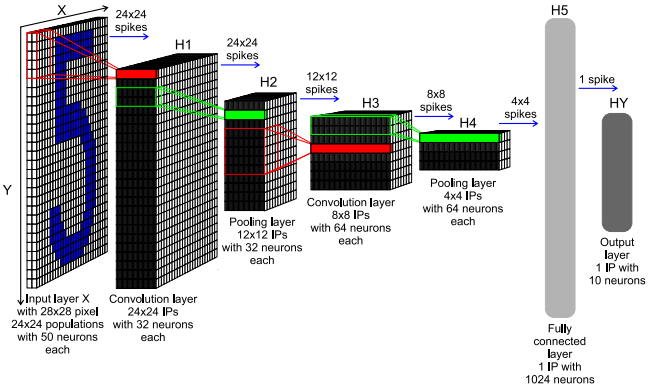

FIGURE 2. SbS network architecture for hand written classification task.

TABLE 1. SbS network architecture for hand written classification task.

	Layer size			Kernel size	
Layer (H^l)	N_X	N_Y	N_H	K_X	K_X
Input (HX)	28	28	2	-	-
Convolution ($H1$)	24	24	32	5	5
Pooling ($H2$)	12	12	32	2	2
Convolution ($H3$)	8	8	64	5	5
Pooling ($H4$)	4	4	64	2	2
Fully connected ($H5$)	1	1	1024	4	4
Output (HY)	1	1	10	1	1

every observed input spike s_t could be derived [5]

$$h_\mu^{new}(i) = \frac{1}{1+\epsilon}\left(h_\mu(i) + \epsilon \frac{h_\mu(i)W(s_t|i)}{\sum_j h_\mu(j)W(s_t|j)}\right) \quad (1)$$

where ϵ is a parameter that also controls the strength of sparseness of the distribution of latent variables $h_\mu(i)$. Furthermore, L can also be used to derive online and batch learning rules for optimizing the weights $W(s|i)$. The interested reader is referred to [5] for a more detailed exposition.

From a practical point of view, SbS provides a mechanism to obtain a sparse representation of input patterns. Given a set of training samples $\{x_\eta\}$, it learns weights (W), that allow to express the input patterns as a linear sparse non-negative combination of features. During inference, it provides a mechanism for expressing each test input x_μ as $x_\mu \approx W\, h_\mu$ where all entries are non-negative.

The inference procedure consists in generating indices s_t distributed according to a categorical distribution of the input pattern $s_t \sim \text{Categorical}(x_\mu(0), x_\mu(1), .., x_\mu(N-1))$. Starting with a random h and executing iteratively **Eq.** (1) the SbS algorithms finds h_μ. The fundamental concept of SbS can be extended from vector to matrix inputs. In this case, the linear operation $W\, h_\mu$ can be replaced by a convolution to obtain a convolutional SbS layer.

B. BASIC NETWORK OVERVIEW

SbS network models can be constructed in sequential layered structures [14]. Each layer consists of many inference populations or IPs (represented by $\vec{h}$), while the communication

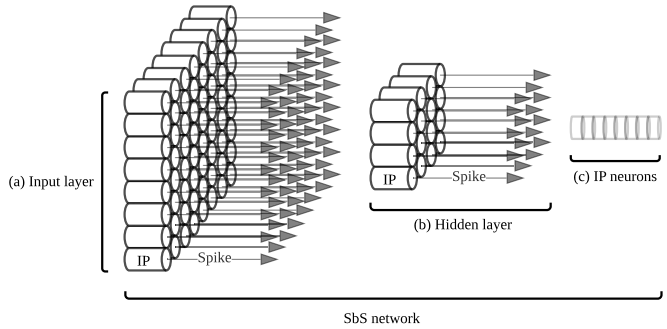

FIGURE 3. SbS IPs as independent computational entities, (a) illustrates an input layer with a massive amount of IPs operating as independent computational entities, (b) shows a hidden layer with an arbitrary amount of IPs as independent computational entities, (c) exhibits a set of neurons grouped in an IP.

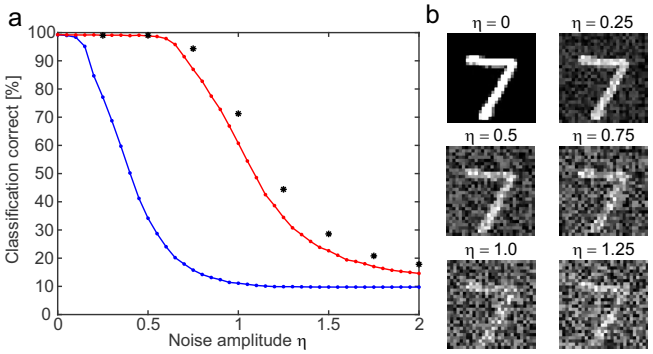

FIGURE 4. (a) Performance classification of SbS NN versus equivalent CNN, and (b) example of the first pattern in the MNIST test data set with different amounts of noise.

between them is organized by a low bandwidth signal – the spikes. A basic SbS network architecture for hand written digit classification is shown in **Fig.** 2 and **Tab.** 1. The SbS network inference is described in Algorithm 1, while spike generation and layer update are described in Algorithm 2 and 3, respectably. Each IP is an independent computational entity, this allows to design specialized hardware architectures that can be massively parallelized (see **Fig.** 3).

C. NOISE ROBUSTNESS COMPARISON

To illustrate the advantages of SbS, an example of the noise tolerance of SbS is presented in **Fig.** 4. It compares the classification performance of a SbS network and a standard convolutional network, with the same amount of neurons per layer as well as the same layer structure. We trained on MNIST dataset [57] without noise (see [14] for details). The figure shows the correctness for the MNIST test set with its 10000 patterns in dependency of the noise level for positive additive uniformly distributed noise. The blue curve shows the performance for the CNN, while the red curve shows the performance for the SbS network with 1200 spikes per inference population. Beginning with a noise level of 0.1, the respective performances are different with a p - level of at least 10^{-6} (tested with the Fisher exact test). Increasing the number of spikes per SbS population to 6000 (performance values shown as black stars), shows that more spikes can improve the performance under noise even more.

Algorithm 1: SbS network inference.

input: Layers of the network as H^l, where l is the layer index.
input: N_L as the number of layers.
input: N_X^l, N_Y^l as the size of layers.
input: N_S as the number of spikes for inference.
output: H^l.

for $t \leftarrow 0$ **to** $N_S - 1$ **do**
  Initialization of $H^l(i_X, i_Y, :)$:
  if $t == 0$ **then**
    for $l \leftarrow 0$ **to** $N_L - 1$ **do**
      for $i_X \leftarrow 0, i_Y \leftarrow 0$ **to** $N_X^l - 1, N_Y^l - 1$ **do**
        for $i_H \leftarrow 0$ **to** $N_H^l - 1$ **do**
          $H^l(i_X, i_Y, i_H) \leftarrow 1/N_H^l$
        end for
      end for
    end for
  end if
  Production of spikes :
  for $l \leftarrow 0$ **to** $N_L - 1$ **do**
    if $l == 0$ **then**
      Draw spikes from Input `// (Algorithm 2)`
    else
      Draw spikes from H^l `// (Algorithm 2)`
    end if
  end for
  Update layers :
  for $l \leftarrow 0$ **to** $N_L - 1$ **do**
    Update H^l `// (Algorithm 3)`
  end for
end for

Algorithm 2: Spike production.

input: Layer as $H_t \in \mathbb{R}^{N_X \times N_Y \times N_H}$, where
  N_X is the layer width,
  N_Y is the layer height
  N_H is the length of $\vec{h}$ (IP vector).
output: Output spikes as $S_t^{out} \in \mathbb{N}^{N_X \times N_Y}$

for $i_X \leftarrow 0, i_Y \leftarrow 0$ **to** $N_X - 1, N_Y - 1$ **do**
  Generate spike :
  $th \leftarrow MT19937PseudoRandom()/(2^{32} - 1)$
  $acu \leftarrow 0$
  for $i_H \leftarrow 0$ **to** $N_H - 1$ **do**
    $acu \leftarrow acu + H_t(i_X, i_Y, i_H)$
    if $th \leq acu$ **or** $i_H == N - 1$ **then**
      $S_t^{out}(i_X, i_Y) \leftarrow i_H$
    end if
  end for
end for

Algorithm 3: SbS layer update.

input: Layer as $H \in \mathbb{R}^{N_X \times N_Y \times N_H}$, where
N_X is the layer width,
N_Y is the layer height
N_H is the length of $\vec{h}$ (IP vector).
input: Synaptic matrix as $W \in \mathbb{R}^{K_X \times K_Y \times M_H \times N_H}$, where
$K_X \times K_Y$ is the size of the convolution/pooling kernel,
M_H is the length of $\vec{h}$ from previous layer,
N_H is the length of $\vec{h}$ from this layer.
input: Input spike matrix from previous layer as $S_t^{in} \in \mathbb{N}^{N_{Xin} \times N_{Yin}}$, where
N_{Xin} is the width of the previous layer,
N_{Yin} is the height of the previous layer.
input: Strides of X and Y as $stride_X$ and $stride_Y$, respectively.
input: Epsilon as $\epsilon \in \mathbb{R}$.
output: Updated layer as $H^{new} \in \mathbb{R}^{N_X \times N_Y \times N_H}$.

Update layer :
$z_X \leftarrow 0$ // X and Y index for S_t^{in}
$z_Y \leftarrow 0$
for $i_Y \leftarrow 0$ **to** $N_Y - 1$ **do**
  for $i_X \leftarrow 0$ **to** $N_X - 1$ **do**
    $\vec{h} \leftarrow H(i_X, i_Y, :)$
    Update IP :
    for $j_X \leftarrow 0, j_Y \leftarrow 0$ **to** $K_X - 1, K_Y - 1$ **do**
      $s_t \leftarrow S_t^{in}(z_X + j_X, z_Y + j_Y)$
      $\vec{w} \leftarrow W(j_X, j_Y, s_t, :)$
      $\vec{p} \leftarrow 0$
      Dot-product :
      $r \leftarrow 0$
      for $j_H \leftarrow 0$ **to** $N_H - 1$ **do**
        $\vec{p}(j_H) \leftarrow \vec{h}_(j_H)\vec{w}(j_H)$
        $r \leftarrow r + \vec{p}(j_H)$
      end for
      if $r \neq 0$ **then**
        Update IP vector :
        for $i_H \leftarrow 0$ **to** $N_H - 1$ **do**
          $h^{new}(i_H) \leftarrow \frac{1}{1+\epsilon}\left(h(i_H) + \epsilon\frac{\vec{p}(i_H)}{r}\right)$
        end for
        Set the new H vector for the layer :
        $H^{new}(i_X, i_Y, :) \leftarrow \vec{h}^{new}$
      end if
    end for
    $z_X \leftarrow z_X + stride_X$
  end for
  $z_Y \leftarrow z_Y + stride_Y$
end for

IV. SYSTEM DESIGN

In this section, we revise the system design of [15]. In Ref. [15], we presented a scalable hardware architecture composed of generic homogeneous accelerator units (AUs). This design works entirely with standard floating-point arithmetic (IEEE 754), which represents an unnecessary overhead for error-resilient applications. Furthermore, this architecture does not implement stationary synaptic weight matrices in the hardware AUs, resulting in heavy data movement and longer computational latency.

In this publication, we present an enhanced hardware architecture composed of specialized heterogeneous processing units (PUs) with hybrid custom floating-point and logarithmic dot-product approximation. This approach represents an advantageous design for error-resilient applications in resource-constrained devices due to the reduced computational costs and memory footprint. Furthermore, the proposed approach allows the implementation of stationary synaptic weight matrices. These novelties result in an improved overall system design.

Regarding the software architecture, this is structured as a layered object-oriented application framework written in the C programming language. This offers a comprehensive high level embedded software application programming interface (API) that allows the construction of scalable sequential SbS networks with configurable hardware acceleration. Conceptually this design is modular, reusable, and extensible. The overall structure is depicted in **Fig.** 5.

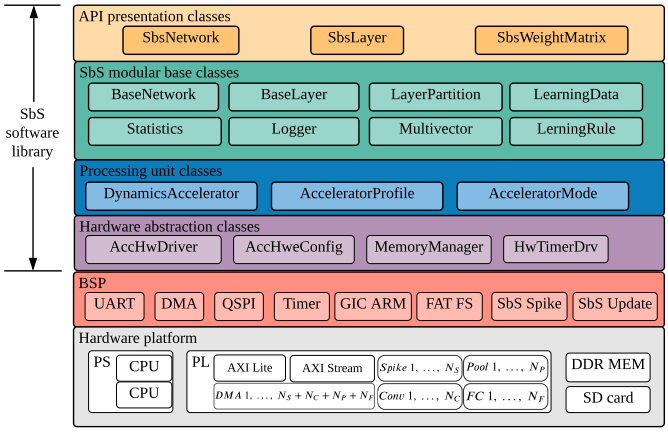

FIGURE 5. System-level overview of the embedded software architecture.

A. HARDWARE ARCHITECTURE

As a hardware/software co-design, the system architecture is an embedded CPU+FPGA-based platform, where the acceleration of SbS network computation is based on asynchronous[1] execution in parallel heterogeneous processing units: *Spike* (input layer), *Conv* (convolution), *Pool* (pooling), and *FC* (fully connected). **Fig.** 6 illustrates the system hardware architecture as a scalable structure. For hyperparameter configuration, each PU uses AXI-Lite interface. For data transfer, each PU uses AXI-Stream interfaces via Direct Memory Access (DMA) allowing data movement with high transfer rate. Each PU asserts an interrupt flag once the job or

[1]The system is synchronous at the circuit level, but the execution is asynchronous in terms of jobs.