fSEAD: A Composable FPGA-based Streaming Ensemble Anomaly Detection Library

BINGLEI LOU, DAVID BOLAND, and PHILIP LEONG, The University of Sydney, Australia

Machine learning ensembles combine multiple base models to produce a more accurate output. They can be applied to a range of machine learning problems, including anomaly detection. In this article, we investigate how to maximize the composability and scalability of an FPGA-based streaming ensemble anomaly detector (fSEAD). To achieve this, we propose a flexible computing architecture consisting of multiple partially reconfigurable regions, pblocks, which each implement anomaly detectors. Our proof-of-concept design supports three state-of-the-art anomaly detection algorithms: Loda, RS-Hash, and xStream. Each algorithm is scalable, meaning multiple instances can be placed within a pblock to improve performance. Moreover, fSEAD is implemented using High-level synthesis (HLS), meaning further custom anomaly detectors can be supported. Pblocks are interconnected via an AXI-switch, enabling them to be composed in an arbitrary fashion before combining and merging results at runtime to create an ensemble that maximizes the use of FPGA resources and accuracy. Through utilizing reconfigurable Dynamic Function eXchange (DFX), the detector can be modified at runtime to adapt to changing environmental conditions. We compare fSEAD to an equivalent central processing unit (CPU) implementation using four standard datasets, with speedups ranging from 3× to 8×.

CCS Concepts: • **Hardware → Reconfigurable logic applications**; *Hardware accelerators*; • **Computing methodologies → Ensemble methods;**

Additional Key Words and Phrases: FPGA, anomaly detection, partial reconfiguration, composability

ACM Reference format:
Binglei Lou, David Boland, and Philip Leong. 2023. fSEAD: A Composable FPGA-based Streaming Ensemble Anomaly Detection Library. *ACM Trans. Reconfig. Technol. Syst.* 16, 3, Article 42 (June 2023), 27 pages.
https://doi.org/10.1145/3568992

1 INTRODUCTION

Anomaly detection is a key **machine learning (ML)** task and refers to the automatic identification of unforeseen or abnormal samples embedded in normal data [7, 39]. Applications of anomaly detection include fault detection surveillance systems [60], fraud detection in financial transactions [3], intrusion detection for network security [15], monitoring of sensor readings in aircraft [4], and discovery of potential risks or medical problems in health data with predictive maintenance [50].

Ensembles are a class of methods that pool weak detectors to form a more accurate combination [13]. Over a number of decades, they have proven to be an excellent methodology that utilizes

Binglei Lou gratefully acknowledges financial support from the China Scholarship Council.
Authors' address: B. Lou, D. Boland, and P. Leong, The University of Sydney, Camperdown NSW 2006, Sydney, Australia; emails: {binglei.lou, david.boland, philip.leong}@sydney.edu.au.

1936-7406/2023/06-ART42 $15.00
https://doi.org/10.1145/3568992

diversity of weak detectors to reach a better overall decision than the individual ones [14]. Each sub-detector in an ensemble is data-independent with identical structure. This makes it naturally amendable for parallel processing to obtain high throughput. The ensemble-size, i.e., the number of sub-detectors, is determined according to the computational resources and constraints of system performance. Generally, executing a large ensemble on **central processing units (CPUs)** as the sequential nature of program execution is mismatched to the available parallelism. **Field-Programmable Gate Array (FPGA)** is an attractive solution, since spatial parallelism can be employed.

When data is susceptible to concept drift [66], anomaly detectors can be utilized in a streaming fashion, with the detector being updated in an online manner. Streaming techniques store and process a window of recent instances. **Streaming ensemble anomaly detectors (SEADs)** achieve high accuracy under limited memory, processing and time constraints because ensembles contribute high accuracy and robustness, and real-time processing is enabled via streaming algorithms.

Existing anomaly detection libraries have been developed for CPUs. These include unsupervised, supervised, heterogeneous approaches such as SUOD [61] and PyOD [63]. These libraries provide a single, well-documented **application programming interface (API)**, making it easy to compare and compose different algorithms. In particular, PySAD introduces a framework of streaming ADs in Python [59] with a common interface. Using the aforementioned AD libraries, a software-based model combination toolkit, *combo*, was presented in Reference [64], allowing anomaly detectors to be combined in Python. CPU-based SEAD libraries allow programmers to switch detector types or to combine multiple detectors to boost performance for specific scenarios.

Implementing customized anomaly detection algorithms in hardware is desirable to achieve higher performance, lower power, and lower latency [10, 18, 41]. However, most designs do not have comparable flexibility or customizability as software-based approaches. This is one of the main challenges for reconfigurable computing, and while we do not solve the problem, we propose an approach with considerably more flexibility than conventional FPGA designs. To the best of our knowledge, no composable FPGA implementations of ADs have been published to date.

In this article, to enable AD on FPGA with higher flexibility and scalability, we propose fSEAD, a composable and low latency FPGA-based SEAD library. fSEAD has two components. The first is a **high-level synthesis (HLS)**-based module generator that converts three state-of-the-art SEAD algorithms (Loda [45], RS-Hash [51], and xStream [40]) into optimized sub-detector-level paralleled FPGA entities that store all parameters in on-chip memory. The second component is a composable hardware framework that enables online switching and dynamic routing of data between IP cores at runtime through reconfigurable **Dynamic Function eXchange (DFX)** and the arbitrary routable AXI4-Stream Switches. This allows new functionality to be introduced to the design at runtime. Composability is gained through the combination of coarse-grained reconfigurability of sub-detectors and switchability, which facilitates their flexible interconnection. While runtime reconfiguration of the FPGA introduces tens or hundreds of milliseconds in overhead for large FPGAs [6, 34], this is not a concern for fSEAD, as this is only done when fSEAD is idle. The main contributions of this article are:

- The first FPGA-based ML system that allows complex and more powerful ADs to be created from simple blocks without recompilation.
- The creation of hardware implementations for three streaming ensemble anomaly detectors (Loda, RS-Hash, and xStream) and demonstration of how they can be integrated within our framework. These implementations are created from an HLS-based generator for FPGA instances, with a GCC-based alternative that creates multi-threaded CPU versions for

comparison. New detectors can be written in C and Python and are easily integrated in this library.
- A composable framework that utilizes multiple reconfigurable regions connected via two AXI switches. In the implementation, high frequency is achieved through floorplanning of pblocks that surround the fSEAD infrastructure, minimizing routing delay.
- We use PYNQ partial overlays invoked in Python for executing AD functions [30], allowing an easy-to-use interface for composing and comparing different ADs.
- This research uses publicly available datasets, and our design has been made open source to facilitate reproducible research.[1]

The remainder of the article is organized as follows: We start in Section 2 with an introduction of the SEAD background and a formal definition of the framework that we deployed in the FPGA. In Section 3, we describe the design of the proposed fSEAD from four aspects: Module Generator, DFX Tool Flow, Composable Infrastructure, and its FPGA Implementation. Then, the results of experiment and performance are shown in Section 4. Section 5 concludes the article and discusses future work.

2 BACKGROUND AND RELATED WORK

In this section, we first introduce anomaly detectors for streaming data. We then discuss how they can be combined using ensemble-centric methods to achieve greater accuracy. We highlight how this has led to the development of several comprehensive software-based AD libraries; this illustrates the desire for our flexible hardware accelerated library that can achieve better performance and similar accuracy to these software implementations. Finally, we discuss the **Dynamic Function eXchange (DFX)** technique and partial overlays, which we have used to provide the flexibility to choose different ensembles at runtime in our accelerator.

2.1 Streaming Anomaly Detection

Anomaly detection is a key **machine learning (ML)** task, which refers to the automatic identification of unforeseen or abnormal samples embedded in a large amount of normal data [7, 39]. From the perspective of processing data, we distinguish between two anomaly detection types: static and streaming.

Static detectors usually operate on a relatively large batch of data before performing information extraction and feature analysis to identify rare items, events, or observations from the general distribution of a population. Representative methods include **k-Nearest Neighbors (kNN)** [47], **Local Outlier Factor (LOF)** [5], and **Principal Component Analysis (PCA)** [53]; for a more comprehensive collection of static methods, we refer the reader to the SUOD [61]. While batch processing can lead to high throughput and accuracy, it is not suitable for systems that require real-time performance, since the computing resource requirements and latency increase with batch size.

In contrast, streaming methods only store and process a window of recent instances [39, 59]. This is more amenable to achieving accurate anomaly scores under limited memory, processing and time constraints. Moreover, the algorithms are designed to facilitate more light-weight and potentially real-time implementations. A group of algorithms that support streaming anomaly detection processing includes, but is not limited to, ensemble-centric methods [40, 45, 51], tree-based methods [38, 54, 58], kernel-methods [39], as well as Neural Network-based solutions such auto-encoders [49] and adversarial models [52].

In this article, we select a set of three states-of-the-art and representative anomaly detectors for our HLS module generator: **Loda (Light-weight Online Detector of Anomalies)** [45],

[1] fSEAD: https://github.com/bingleilou/fSEAD.

Table 1. Block Diagram of SEAD Methods

Anomaly Detectors	Blocks			
	Projection	***Core***	***Sliding-Window***	***Score***
Loda	prj-loda	histogram	$1 \times W$	$-\log_2(c/W)$
RS-Hash	prj-rshash	CMS	$w \times W$	$-\log_2(1 + \min\{c_1, \ldots, c_W\})$
xStream	prj-xstream	CMS	$w \times W$	$-\log_2(1 + \min\{2^1 c_1, \ldots, 2^W c_W\})$

[1] W: the length of the sliding-window.
[2] w: the number of hash functions in the count-min sketch (CMS).
[3] c: the count of histogram or hash code of CMS.

RS-Hash [51], and xStream (Outlier Detection in Feature-Evolving Data Streams) [40]. These are briefly explained below. We denote $X = \{\vec{x}_i\}$ as the input dataset and dimension d of each sample, then $\vec{x}_i \in \mathbb{R}^d$ is the ith input sample of the data stream, and R is the ensemble size.

Loda is a projection-based histogram detector, RS-Hash uses a randomized subspace hashing algorithm, while xStream is a density-based detector. Although the three techniques are based on different principles, the algorithms can be expressed as a composition of the following standardized blocks: *Projection*, *Core*, *Sliding-window*, and *Score*. Table 1 summarizes the main functional blocks of these three SEAD methods.

The purpose of *projection* is to reduce the dimensionality of a set of points while retaining the essence of the original data. Randomness between sub-detectors to improve diversity in an ensemble is also introduced at this step. It is also the most computationally expensive step. The *core* block is the cornerstone of each method: Loda is based on the histogram; RS-Hash and xStream approaches make use of the **count-min sketch (CMS)** [9], in which w pair-wise independent hash tables are used. These three methods all operate over a *sliding-window* of the input data, which is received in a streaming fashion. The difference is that the histogram-based Loda only uses a 1-row table as a sliding-window, where the CMS-based methods allow the sliding-window with a w-row table ($w \geq 1$). The *score* block calculates the negative log-likelihood so the less likely a sample, the higher the anomaly value.

Algorithms 1 to 3 present the pseudo-code for Loda, RS-Hash, and xStream, respectively. Algorithm 4 introduces the hash function that is applied in RS-Hash and xStream. Clearly, each function has the streaming input x and the streaming output $Score$.

The ensemble can be divided into seven parts: The ❶Windower block uses a shift register to assemble samples $x \in \mathbb{R}^1$ into an entire vector $\vec{x}_i = \{x_1, x_2, \ldots, x_d\} \in \mathbb{R}^d$, ❷Ensemble is a big *for* loop with R independent iterations. Each iteration can be regarded as a sub-detector with a unit of 1, and each sub-detector executes the functional modules of ❸Projection, ❹Core (Histogram for Loda, and Hash Function for RS-Hash and xStream) ❺Sliding-window and ❻Score sequentially. The execution of these modules is time-dependent, that is, the start of the latter one must wait for the execution of the former to complete. Finally, ❼Score Averaging is used for producing the final ensemble anomaly score by averaging the outputs of all sub-detectors.

The serial combination of these four blocks (❸❹❺❻) constitutes a base sub-detector. We use R parallel executions of the base sub-detector with a different starting seed or hash and average their scores to form an *ensemble*. For a CPU implementation, R sub-detectors are processed sequentially. Details of HLS directives used in the pseudo-code are described in Section 3.1.

2.2 Ensembles of Multiple Anomaly Detectors

Ensembles are a class of methods that combine weak detectors to collectively form a more accurate decision by utilizing diversity in the detectors [13, 14]. Each sub-detector in an ensemble is

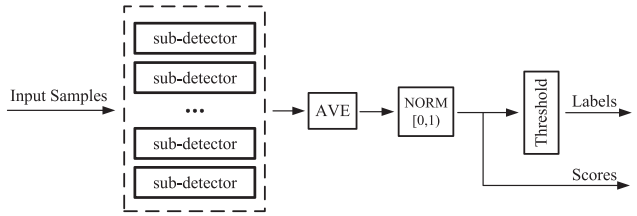


Fig. 1. Sample architecture for SEAD methods.

ALGORITHM 1: Loda

Input: Streaming input signal $X = \vec{x}_i \in \mathbb{R}^d$; ensemble size R; data dimension d; sliding-window length W.
Output: Streaming anomaly value *Score*.

#pragma HLS DATAFLOW
// ❶Windower:
for $dim = 1, 2, \ldots, d$ **do**
  A shift register to produce an entire sample: X with d features.
end
// ❷Ensemble:
for $r = 1, 2, \ldots, R$ **do**
  // ❸Projection:
  for $dim = 1, 2, \ldots, d$ **do**
    #pragma HLS PIPELINE
    $prj\_X \leftarrow prj\_X + X * loda\_prj$ (random projection)
  end
  // ❹Histogram:
  $idx \leftarrow (prj\_X - loda\_\min)/(loda\_\max - loda\_\min)$
  $v \leftarrow sliding\text{-}window[idx]$
  // ❺Sliding-window:
  Update *sliding-window*.
  // ❻Score:
  $score(r) \leftarrow \log_2(v)$
end
// ❼Score Averaging:
$Score \leftarrow \frac{1}{R} \sum_{r=1}^{R} score(r)$
return *Score*

data-independent and structure-identical. The first feature makes it naturally amendable for parallel processing, while the latter gives it the flexibility to assemble arbitrary numbers of sub-detectors into an ensemble according to the computational resources and desired accuracy.

An example architecture for **streaming ensemble anomaly detectors (SEADs)** is illustrated in Figure 1, where each sub-detector inside SEAD takes a stream of input data and produces a stream of transformed outputs that indicate the anomaly scores. Averaging is used to compute the final score from all sub-detectors, or a *threshold* can be applied to translate this averaged score to a *Labels* (anomaly or no anomaly). While a simple ensemble could be multiple instances of the same detector, as described in the previous section, a more powerful ensemble will utilize different detectors.

ALGORITHM 2: RS-Hash

Input: Streaming input signal $X = \vec{x}_i \in \mathbb{R}^d$; ensemble size R; data dimension d; sliding-window length W; hash functions number in CMS: w.
Output: Streaming anomaly value *Score*.

#pragma HLS DATAFLOW
// ❶Windower:
for $dim = 1, 2, \ldots, d$ **do**
  A shift register to produce an entire sample: X with d features.
end
// ❷Ensemble:
for $r = 1, 2, \ldots, R$ **do**
  // ❸Projection:
  for $dim = 1, 2, \ldots, d$ **do**
    #pragma HLS PIPELINE
    $norm\_X \leftarrow$ normalize X[dim] to the range of [0,1]
    $prj\_X \leftarrow (norm\_X + rshash\_alpha[r][\text{dim}])/rshash\_f[r]$
  end
  // ❹Hash-Function:
  for $row = 1, 2, \ldots, w$ **do**
    #pragma HLS UNROLL
    $prj\_hash[row][\text{dim}] \leftarrow prj\_X$
    $hash\_value[row] \leftarrow Jenkins(key = prj\_hash[row], len = d, seed = row)$
    (see **Algorithm 4** for details of *Jenkins*)
    $v_{row} \leftarrow sliding\text{-}window[hash\_value[row]]$
    // ❺Sliding-window:
    Update *sliding-window*.
  end
  $min\_v \leftarrow \min\{v_1, v_2, \ldots, v_w\}$
  // ❻Score:
  $score(r) \leftarrow \log_2(min\_v)$
end
// ❼Score Averaging:
$Score \leftarrow \frac{1}{R}\sum_{r=1}^{R} score(r)$
return *Score*

An easy-to-use scalable library provides the opportunity to explore the performance of ensembles, many of which have been developed. Examples of anomaly detection packages include: ELKI Data Mining [1] and RapidMiner [20] in Java; Outliers [33] in R; SUOD [61], PyOD [63], and PySAD [59] in Python. Aside from their differences in programming languages, different libraries are also tailored to different kinds of anomaly detection, e.g., PySAD focuses in particular on a framework of streaming ADs in Python, whereas only static approaches can be accessed by SUOD and PyOD.

In addition to supporting comprehensive detectors, the libraries provide the ability to combine the output of these models in different ways beyond simply taking the average or maximum across all the base models. A software toolkit, *combo* [64], contains more than 15 model combination methods in Python, including basic generic and global methods (such as Averaging, Maximization, Weighted Averaging, Feature Bagging, etc. [2, 35, 65]) and dynamic selection/combination models

ALGORITHM 3: xStream

Input: Streaming input signal $X = \vec{x}_i \in \mathbb{R}^d$; ensemble size R; data dimension d; sliding-window length W; hash functions number in CMS: w; projection size: K.
Output: Streaming anomaly value *Score*.

#pragma HLS DATAFLOW
// ❶Windower:
for $dim = 1, 2, \ldots, d$ **do**
    A shift register to produce an entire sample: X with d features.
end
// ❷Ensemble:
for $r = 1, 2, \ldots, R$ **do**
    // ❸Projection:
    for $dim = 1, 2, \ldots, d$ **do**
        #pragma HLS PIPELINE
        for $k = 1, 2, \ldots, K$ **do**
            #pragma HLS UNROLL
            $prj\_X[k] \leftarrow prj\_X[k] + X[\text{dim}] * xstream\_prj[\text{dim}][k]$
        end
    end
    // ❹Hash-Function:
    for $row = 1, 2, \ldots, w$ **do**
        #pragma HLS UNROLL
        $prj\_hash[row] \leftarrow perbins(prj\_X)$
        $hash\_value[row] \leftarrow Jenkins(key = prj\_hash[row], len = K, seed = row)$
        (see Reference [40] and **Algorithm 4** for details of *perbins* and *Jenkins*)
        $v_{row} \leftarrow sliding\text{-}window[hash\_value[row]]$
        // ❺Sliding-window:
        Update *sliding-window*.
        $score_{row} \leftarrow \log_2(v_{row}) + row$
    end
    // ❻Score:
    $score(r) \leftarrow \min\{score_1, score_2, \ldots, score_w\}$
end
// ❼Score Averaging:
$Score \leftarrow \frac{1}{R} \sum_{r=1}^{R} score(r)$
return *Score*

(such as DCS [19] and LSCP [62], etc.). While our library currently only performs averaging inside an ensemble, other combination techniques are easily implemented.

2.3 Dynamic Partial Reconfiguration and Partial Overlays

FPGA technology provides the flexibility to modify a hardware implementation without re-fabrication. **Partial reconfiguration (PR)** takes this flexibility a step further, allowing dynamic changes to an active design. This requires implementing static logic, multiple **Reconfigurable Partitions (RPs)** with various **Reconfigurable Modules (RMs)**. The RP is the level of hierarchy within which different RMs can be implemented and an RM is the netlist or HDL description that is implemented within an RP. Generally, Multiple RMs with the same interface exists for a specific

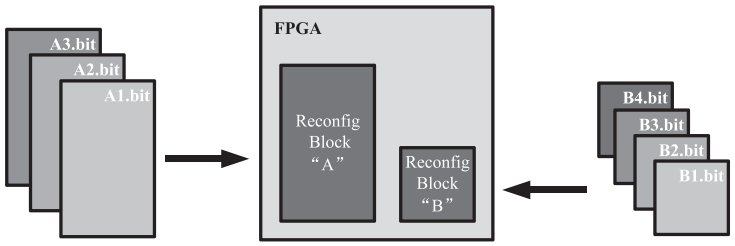


Fig. 2. Basic premise of partial reconfiguration.

ALGORITHM 4: Jenkins Hash Function

Input: The string: *key*, string length: *len* and random seed: *seed*
Output: The hash code of input string.

```
hash ← seed;
for i = 1, 2, ..., len do
    #pragma HLS PIPELINE
    hash ← hash + key[i]
    hash ← hash + (hash << 10)
    hash ← hash ⊕ (hash >> 6)
end
hash ← hash + (hash << 3)
hash ← hash ⊕ (hash >> 11)
hash ← hash + (hash << 15)
hash_code ← hash%MOD
return hash_code
```

RP. Dynamic Function eXchange refers to a Xilinx tool flow that achieves the partial reconfiguration [26, 27].

Figure 2 illustrates the basic premise of partial reconfiguration. The grey area of the FPGA block represents static logic, and the blocks marked as Reconfig Block "A" and Reconfig Block "B" represent the RPs. The functionality implemented in Reconfig Block "A" can be modified by downloading one of several partial BIT files: A1.bit, A2.bit, or A3.bit; similarly, the functions implemented in Reconfig Block "B" can be modified by one of B1.bit to B4.bit.

Using DFX allows the fSEAD library to be able to support improvements or new ADs developed in the future. Moreover, supporting multiple customized partial regions enables fSEAD to support ADs that have different resource requirements. This flexibility is also important, because different applications will have different hardware capabilities and accuracy requirements, while the optimal performance may change, depending on external conditions, such as environmental changes.

2.4 Literature Review

This section reviews other anomaly detection techniques, literature on FPGA-based anomaly detection, and briefly highlights related research in dynamic reconfigurable FPGA implementations for other applications. A powerful class of anomaly detectors utilize tree-based structures, with examples including iForest [38], HS-Tree [54], and RS-Forest [58]. The **Isolation Forest (iForest)** approach builds an ensemble of **"Isolation Trees" (iTrees)** for the dataset, and anomalies are the points that have shorter average path lengths on the iTrees. HS-trees are similar to Isolation

Forest, but decision rules within tree-nodes are generated randomly. RS-Forest takes a further step by using multiple fully randomized space trees to tackle the streaming detection problem from the density estimation aspect, which is more efficient, as it leverages the common operations shared by the prediction and model update processes.

Kernel methods and data-centered models have also been proposed as online anomaly detectors. **Support Vector Regression (SVR)** with Gaussian kernel-based online novelty detection on temporal sequences is presented by Ma et al. [39]. High-performance FPGA implementations of online kernel methods was demonstrated in references [36, 43]. Based on static data-centered LOF methods [5], an incremental LOF algorithm, appropriate for detecting outliers in data streams, is proposed in Reference [46] that provides equivalent detection performance as the iterated static LOF algorithm, while requiring significantly less computational time. Das et al. proposed a system based on feature extraction and **Principal Component Analysis (PCA)** [10]. Their FPGA implementation could support data at over 20 Gbps. Pang et al. [43] present a high-performance FPGA implementation that achieves improvements in execution time, latency, and energy consumption by factors of 5, 5, and 12, respectively, over CPU and **digital signal processor (DSP)** implementations for the online kernel method [39]. Hayashi and Matsutani [18] offload the **Local Outlier Factor (LOF)** calculation to a FPGA-based Network Interface Card for online anomaly filtering. This leads to throughput improvements of up to 10× on the anomaly filtering over a software-based execution.

Neural Network models have also been proposed for streaming anomaly detection. Moss et al. [41] introduce an FPGA-accelerated Neural Network-based anomaly detector based on an auto-encoder for processing of physical-layer **radio-frequency (RF)** signals. This design processes continuous 200 MS/s complex inputs, producing anomaly classifications at the same rate, with a latency of 105 ns, an improvement of at least 4 orders of magnitude over a conventional approach using a software defined radio.

While there are published FPGA implementations of random forests [31, 37, 56], kernel, and neural network-based FPGA-based accelerators, these are all fully customized hardware designs for a specific algorithm. Moreover, they typically utilize most of the available FPGA resources, meaning it is difficult to implement such detectors in a partial region of an FPGA. The focus of this article is on the development of a flexible library. Unlike the many comprehensive and easy-to-use anomaly detection libraries released on different software platforms [1, 20, 33, 59, 61, 63], we believe this is the first flexible FPGA library for anomaly detection in the literature.

Although we are not aware of any publications describing runtime reconfigurable anomaly detection libraries, dynamic reconfiguration on FPGAs has been used for other FPGA applications. The most similar to this work is Wilson, who proposed a real-time video processing pipeline that utilizes the dynamic reconfigurable aspects of FPGA [57]. Their work used 11 reconfigurable regions, allowing for multiple custom runtime configurations, and it adopts the partial bitstreams with PYNQ overlays to ease the software development. Our work is different in that it uses AXI switches to dynamically switch anomaly detectors and model combinations between reconfigurable regions and employ a flexible module generator to create the regions themselves.

3 DESIGN

In this section, we first provide a high-level description of the fSEAD system, followed by the module generator for creating integrated anomaly detection IPs. Next, we introduce the DFX workflow we abstracted for creating partial reconfiguration project in Xilinx Vivado environment. We then describe the interconnection scheme that provides a high degree of flexibility and enables IP modules to be composed at runtime, allowing applications to be efficiently accelerated without regeneration of bitstreams. Finally, the FPGA implementation is discussed.

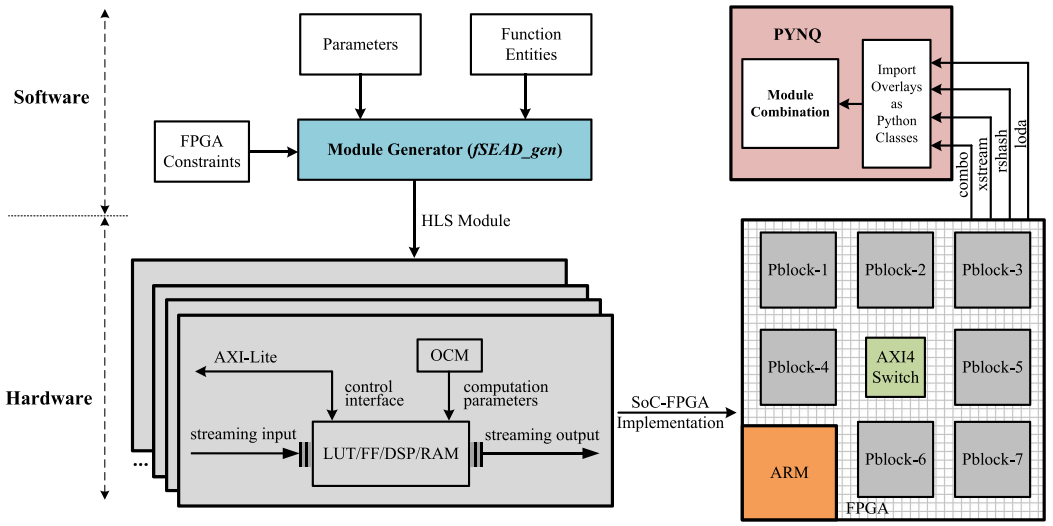


Fig. 3. Overview of the fSEAD framework.

The system framework is illustrated in Figure 3. It consists of four software and hardware components. *fSEAD_gen* in the upper-left corner of Figure 3 is a Python-based module generator. It takes parameterized function entities and produces Vivado HLS modules under. The lower-left corner of Figure 3 shows the interface for pblocks, which take streaming inputs and produce streaming outputs. These then are encapsulated as multiple unique IPs and wrapped within AXI streaming interfaces and an AXI-Lite controller. All parameters required by the IP modules are stored in **on-chip memory (OCM)** for performance. The lower-right sub-figure shows multiple pblock regions. Multiple sub-detector IPs are arranged in a spatially parallel fashion within each pblock. Each pblock is implemented by many **reconfigurable modules (RMs)** and can be customized at runtime.

3.1 Module Generator

The module generator allows customization of the underlying sub-detectors for latency optimization and resource utilization exploration.

The *fSEAD_gen* module generator, written in Python, takes as inputs: the anomaly detector parameters, data-type, precision, function description, target dataset, and a testing set. It produces a standalone C program suitable for synthesis via HLS as output. The parameters, interface, and optimization directives are all embedded in the C program. A compact sub-detector C instance is formed by combining the ❸Projection, ❹Core, ❺Sliding-window, and ❻Score parts of Algorithms 1–3. Arbitrary numbers of sub-detectors, specified by the user to the module generator, are integrated in parallel manner to form an ensemble. A self-verifying test-bench compares the program with golden results from the original Python description. Thus, programming errors introduced by the generated ensemble program can be quickly identified in the simulation step. For synthesis, compiler directives such as DATAFLOW, ARRAY PARTITION, UNROLL, and PIPELINE are used to aid translation of the C description to a spatially parallel RTL circuit.

In Algorithms 1 to 3, we use the DATAFLOW pragma only in the top layer (line 1 of each algorithm) to enable task-level pipelining. This allows functions and loops to execute concurrently and achieve sub-detector parallelism. UNROLL is used to create multiple instances of the loop body in the RS-Hash and xStream algorithms to exploit spatial parallelism. This enables all hash

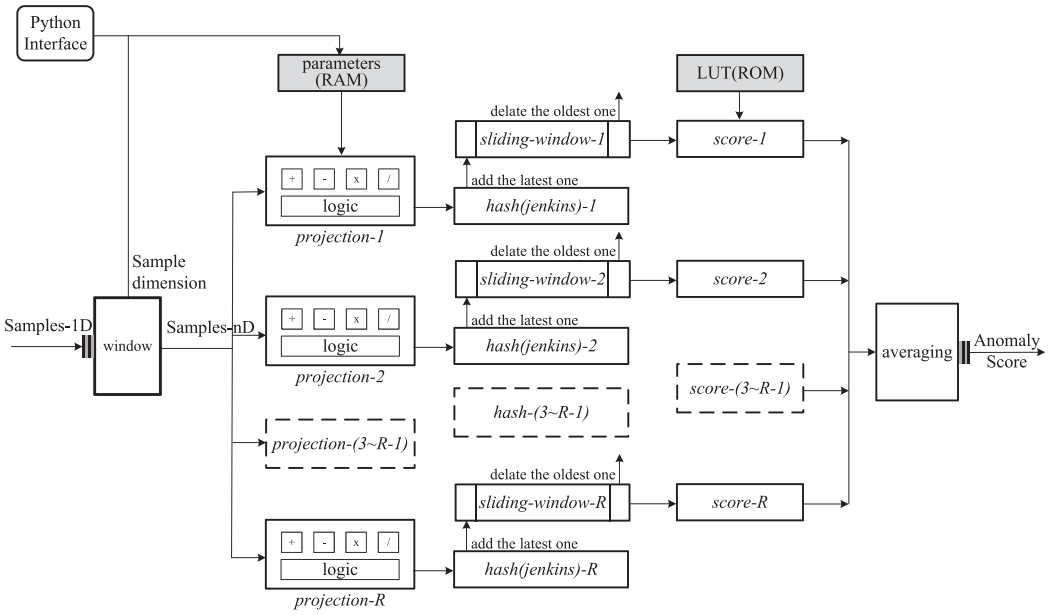


Fig. 4. An example of a hash-based AD hardware structure in fSEAD.

functions inside the CMS table (line 16 in Algorithm 2 and line 18 in Algorithm 3) to execute concurrently, and similarly accelerates the projection of size K for xStream (line 12 in Algorithm 3). PIPELINE reduces the initiation interval (II, the number of clock cycles before the function can accept new input data) by allowing overlapped execution of operations within a loop or function. It is used in the *projection* loop body of all detectors (line 10 of Algorithms 1 to 3) and the for loop in the Jenkins hash function (line 3 of Algorithm 4). $II = 1$ is achieved in all the loop bodies with PIPELINE for our implementation. The result of all these optimizations is a highly parallel design that balances resource utilization and latency.

fSEAD_gen currently supports three types of detectors with arbitrary ensemble size and coefficients (Loda, RS-Hash, and xStream). All HLS directives are first manually designed for the target FPGA and then embedded in *fSEAD_gen*. *fSEAD_gen* can automatically generate optimizations for different ensemble sizes, such as hyper-parameters and other configurations, without modifying the existing HLS directives. However, developers maintain the flexibility to adapt the current HLS directive to find more optimized solutions for a specific area-latency tradeoff requirement. New detectors can be written in C/Python and easily integrated using existing detectors as examples. In this case, specific optimized HLS directives have to be manually re-designed for new members to the fSEAD library.

Figure 4 shows an ensemble of hash-based detectors that have a more complex hardware implementation than Loda. For simplicity, this figure highlights the parallel sub-detectors generated by the DATAFLOW pragma, instead of the detailed circuits from the lower level by PIPELINE and UNROLL pragmas. This represents the main acceleration of fSEAD: task-level parallelism for sub-detectors operating concurrently.

Input data is streamed into the IP on the left interface and outputs the real-time anomaly score on the right side. Input samples are windowed to assemble a batch with target dimension: d (refer to ❶Windower in Algorithms 2 and 3). Computation of all the streams occur concurrently, before a final reduction step (refer to ❷Ensemble, in Figure 4 this is averaging). Sub-detector level

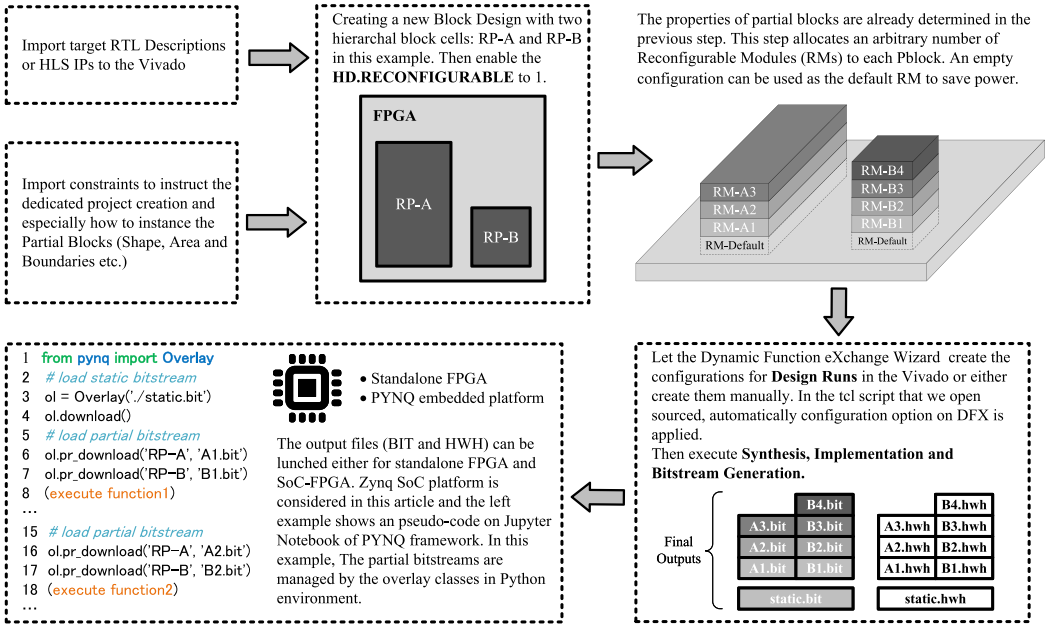


Fig. 5. An example of Xilinx partial reconfiguration tool flow.

parallelism is achieved by applying the HLS DATAFLOW directive on the top function. In this way, the maximum ensemble size is only limited by resources available in the target FPGA. Inside each sub-detector, small area and low latency are the goals. We compute each row of a CMS table using independent hash functions in parallel (refer to ❹Hash-Function). PARTITION directives are used to divide a large RAM into smaller storage units, increasing available parallelism of accesses. This is advantageous, because even with dual-port RAM, only one read and one write operation can be completed in one clock period. Notably, since the usage of PARTITION significantly affects the resource consumption of LUTs or FFs on FPGA, it is only used for a specific dimension of the target array. This guarantees the functions of DATAFLOW, PIPELINE, and UNROLL directives. All HLS directives above enable a balanced optimization inside an ensemble AD instance, which is independent of the partial-block-level model combination scheme introduced in the following Section 3.3.

To avoid high resource cost with little throughput benefit, we constrain the ❸Projection module using PIPELINE instead of UNROLL inside each sub-detector. To compute the logarithm required for the negative log-likelihood score of Table 1, a W-deep lookup table with 32-bit representation is used for window-size of W.

3.2 DFX Tool Flow

DFX is an important tool to endow fSEAD with partial reconfigurability. Its customized workflow for fSEAD is introduced in this section. Continuing on the basis of the example in Figure 2. Figure 5 shows an example of how two reconfigurable areas are mapped to **Reconfigurable Partitions (RPs)** using our tool flow. It should be noted that in addition to the dynamic RMs for each Partial block (e.g., the RM-A1 to RM-A3 for RP-A, and RM-B1 to RM-B4 for RP-B), a default RM can also be assigned for each pblock. The logic of the RM-Default will be first active when the static.bit is downloaded, which brings with a recommended way of setting the default RM to an empty logic to save power before this pblock is configured by the dedicated RM of users.

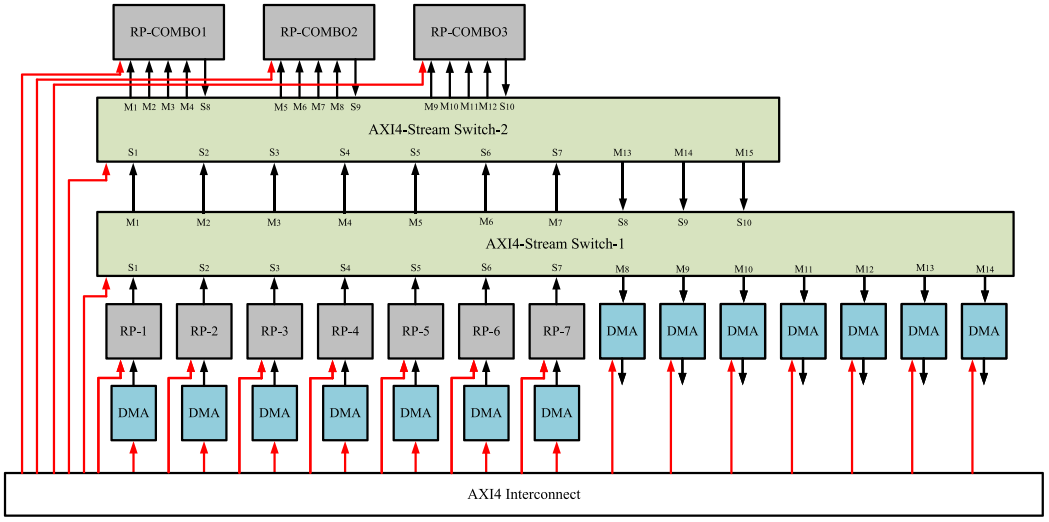


Fig. 6. Composable topology.

In the last step (the lower left sub-figure) in Figure 5, we show that standalone FPGA or Xilinx PYNQ platforms are able to support the fSEAD execution. We select the PYNQ as the final execution platform in this example. The bitstreams and a **hardware hand-off (HWH)** file are used as overlays by PYNQ to automatically identify the Zynq system configuration, IP including static regions and partial blocks.

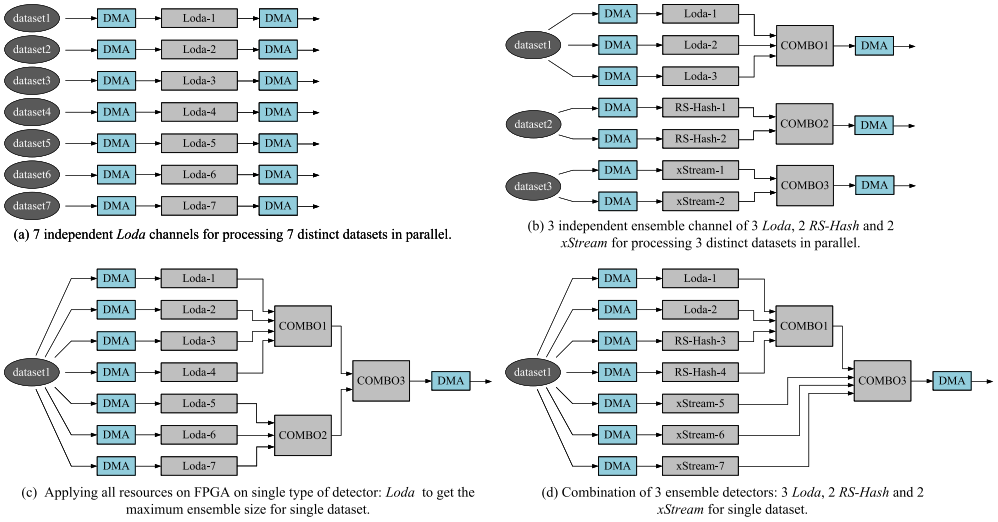
3.3 Composable Infrastructure

The Xilinx DFX tool [27] supports partial reconfiguration of RMs in an FPGA while the rest of the device remains operational. There is no limitation in the number of RMs supported. In fSEAD, each pblock is a unique RM configuration with its own BIT file. By downloading the appropriate one, the hardware functionality can be customized at runtime. In summary, pblocks are either: (1) Ensembles of homogeneous sub-detectors implemented in a pblock; and (2) Combination modules that aggregate heterogeneous pblock output streams.

An AXI4 switch acts as a router and orchestrates data movements between pblocks. Inputs can be routed from multiple input streams to any pblock, with outputs routed to remaining pblocks and back to a host processor.

Figure 6 shows the composable topology proposed in fSEAD. The grey regions are pblocks. Blue blocks symbolize **Direct Memory Access (DMA)** controllers for data transfer. Two AXI4-Stream switches [24] enable dynamic routing from a Slave port to a Master port, e.g., for switch-1, S1 in the lower left is the first Slave interface, and M14 in the lower right symbolizes the 14th Master interface. Master and Slave interfaces are symmetric and point-to-point, so Master output signals can connect directly to Slave input signals. Any number of external modules can be daisy-chained together. The modules can be used for a multitude of different tasks such as buffering, data transform, or routing. Multiple switches are used, since each Xilinx AXI4-Stream Switch IP switch only supports a maximum of 16 Slave ports and 16 Master ports. Cascades of two or more switches allow an arbitrary number of pblocks to be interconnected. The black lines depict AXI4-Stream connections, and the red lines are AXI-Lite bus connections. These are connected and controlled by the AXI4 Interconnect at the bottom.

In our prototype implementation, seven independent pblocks are available for implementing anomaly detectors (shown as RP-1 to RP-7 in Figure 6). Each pblock has one AXI4-Stream input



(a) 7 independent *Loda* channels for processing 7 distinct datasets in parallel.

(b) 3 independent ensemble channel of 3 *Loda*, 2 *RS-Hash* and 2 *xStream* for processing 3 distinct datasets in parallel.

(c) Applying all resources on FPGA on single type of detector: *Loda* to get the maximum ensemble size for single dataset.

(d) Combination of 3 ensemble detectors: 3 *Loda*, 2 *RS-Hash* and 2 *xStream* for single dataset.

Fig. 7. Examples of combination scheme.

connected to a fixed DMA and one output interface connected to a Slave port of AXI4-Stream Switch-1. Multiple sub-detectors can be placed in a single pblock, each with different parameter settings. For example, 35 Loda sub-detectors fit in a single pblock. The combo pblocks at the top of Figure 6 are responsible for aggregation. Each is equipped with four input ports and one output port, all connected to Switch-2.

Table 2 shows the currently supported combination methods implemented in fSEAD for continuous (score) and discrete (label) targets. The label targets are "0" meaning not an anomaly and "1" for anomaly. **General and global (GG)** methods [2, 35, 62, 65] are used by popular machine learning libraries such as scikit-learn [44], XGBoost [8], and LightGBM [32] for aggregation of sub-detector class results. In fSEAD, for sub-detectors with continuous outputs, we implemented three representative GG methods, namely, **Averaging (*GG_A*)**, **Maximum (*GG_M*)**, and **Weighted Average (*GG_WA*)**. To combine label outputs, two commonly used approaches, or (a class is true if any sub-detector outputs true for that label) and voting (the class with most votes is chosen as the output), are applied. In this work, we always use averaging for combining anomaly scores and the or-gate technique to combine labels. Of course, this can be customized by the user.

Routing through the AXI switches is configured via the AXI-Lite interface. A register is used for each master to slave connection. After the registers have been configured, the interconnection is determined. Unused master or slave interfaces are disabled. When a slave interface is connected to multiple masters, only the lowest numbered one is used, e.g., if both Master-1 and Master-3 are configured to connect to Slave-2, then Master-1 wins the arbitration and Master-3 is disabled. Thus, effectively only one connection between each master and slave is made.

Figure 7 shows four example configurations of our composable topology. Figure 7(a) shows the simplest case where seven parallel Loda pblocks (each containing multiple sub-detectors) are used to analyze seven different datasets. Each streaming channel implements a unique and independent anomaly detection application. Switch-1 is configured to directly route RP-1 to RP-7 to seven output DMA channels. This configuration only requires Switch-1 so connections to Switch-2 and the combo pblocks are disabled. The case in Figure 7(b) implements three independent anomaly detection applications. It exploits all pblocks and two of the Switches. RP-1, RP-2, and RP-3 implement a Loda ensemble, and their outputs are routed to the inputs of COMBO1. The output is the final

Table 2. Combination Methods for Scores and Labels

Output	Combination Methods	Equation
score	Averaging	$combo = (score_1 + score_2 + \cdots + score_N)/N$
score	Maximization	$combo = max(score_1, score_2, \ldots, score_N)$
score	Weighted Average	$combo = (w_1 * score_1 + w_2 * score_2 + \cdots + w_N * score_N)/N, (\sum_{i=1}^{N} w_i = 1)$
label	Or	$combo = (label_1 \vert label_2 \vert \cdots \vert label_N)$
label	Voting	$combo = voting(label_1, label_2, \ldots, label_N)$

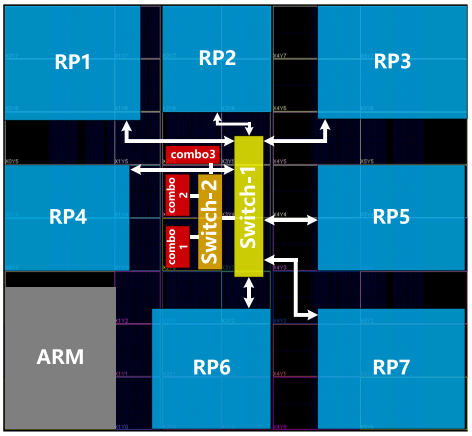


Fig. 8. FPGA layout.

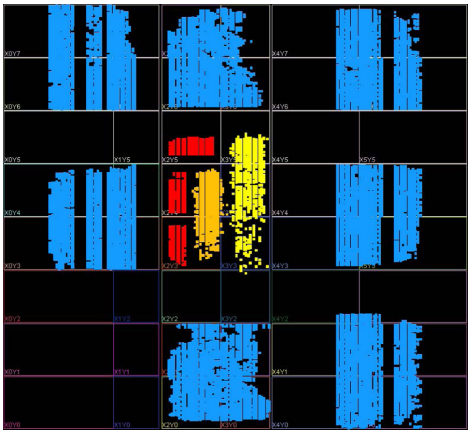


Fig. 9. Placement on FPGA.

score that is sent to the host via DMA. The pblocks (RP-4, RP-5) and (RP-6, RP-7) generate RS-Hash and xStream ensembles for two different datasets. Figure 7(c) is an example that only operates on a single dataset and a single type of anomaly detector, namely, Loda. It uses all the pblocks to implement a maximally parallel, homogeneous ensemble. Finally, Figure 7(d) is similar to Figure 7(c) but incorporates three different types of detectors: Loda, RS-Hash, and xStream.

The configurations are not limited to the four cases just described. By providing different pblocks and switch configurations, a multitude of customized anomaly detection functionalities can be implemented in a runtime-configurable manner.

3.4 FPGA Implementation

This section continues the example of Figure 7(c) as concrete example to describe physical implementation on a Zynq UltraScale+ ZCU111 Evaluation Board with XCZU28DR-2FFVG1517E RFSoC FPGA.

Figures 8 and 9 show the FPGA layout and placement, respectively. The floorplan is based on two considerations: (1) Seven AD-pblocks occupy the main FPGA resources to ensure that the fSEAD has sufficient resources to implement parallel ensembles and complex detectors. In comparison, the combo modules, switches, and DMA units are not resource-intensive and are reflected in the layout; (2) Although long interconnections are needed because of the distributed pblocks, this is alleviated through AXI bus-level pipelining [25]. This technique isolates the path between the master and slave with registers, while maintaining an AXI4 protocol-compliant pipeline stage. The register slice is implemented as a two-deep FIFO buffer that supports bus communication without generating unnecessary idle cycles. It serves to achieve timing closure by reducing the critical path.

We place Switch-1 (yellow) in the center of the FPGA with Switch-2 (orange) adjacent, since the two communicate directly with each other. Switch-1 is assigned a larger area than the Switch-2, as it connects to seven pblocks (in blue), while Switch-2 is smaller to prevent blocking routes from RP-1 and RP-4 to Switch-1. As shown in Figure 8, the nearest routing channel from RP-1 to Switch-1 is only a very narrow programmable slot (white line). The three combo-pblocks (red) only connect to Switch-2. The floorplan for the above-mentioned blocks is compact and sits in the middle layout of FPGA.

The seven AD-pblocks (in blue) occupy the remaining resources. It is important to note that we did not issue placement area constraints for DMAs, AXI4-Interconnect, and the other static components. Instead, we allow these components to be placed by the Vivado tool to minimize routing delay. The spaces between the colored regions are available for any remaining static logic. Furthermore, during the Partial Reconfiguration process, the DFX Decoupler [27] is used to allow users to isolate the logic being configured until it is done and the new logic reset.

4 RESULTS

The aforementioned techniques were implemented and results presented in this section. In Section 4.1, we introduce the development environment and test platform. In Section 4.2, we demonstrate why larger ensembles with more sub-detectors is desirable and how performance is boosted in terms of accuracy from model combination. We then discuss the resource use of the static regions and available resources in each of the FPGA partial regions in Section 4.3. Furthermore, we show the performance gains of our architecture over a CPU in Section 4.4. Finally, in Section 4.5, we analyze the performance characteristics of partial reconfiguration and more general features of the fSEAD architecture.

4.1 Test Platform

We utilize the Xilinx Zynq UltraScale+ ZCU111 (xczu28dr-2ffvg1517e) board for evaluating the fSEAD library. *fSEAD_gen* is used to generate three anomaly detectors (Loda, RS-Hash, and xStream) with HLS and GCC targets used for FPGA and multi-threaded CPU implementation, respectively. The HLS module is synthesized to RTL using the Xilinx Vivado HLS tool (v2020.1) and then passed through Xilinx Vivado Design Suite (v2020.1). We then deploy the architecture on the FPGA using the PYNQ framework. The GCC compiled versions of Loda, RS-Hash, and xStream as CPU benchmarks are tested on desktop with Intel(R) Core(TM) i7-10700F @2.9 GHz and 64 GB memory for performance comparisons. The g++ compiler is used with flags "-Wall –std=c++14 -g -O3," with "-lpthread" applied for multi-threaded optimization. The flag "-ftree-vectorize" is turned on by default under "-O3" to enable automatic vectorization for further optimization.

The anomaly detection performance evaluation was conducted on four publicly available datasets: Cardio, Shuttle, HTTP-3, and SMTP-3, with their main attributes summarized in Table 3. Cardio and Shuttle are also used in SUOD [61]. HTTP-3 and SMTP-3 were derived from the KDD-cup 99 [55] dataset and are 3 feature variants of the 41 feature full datasets, first used in Reference [54]. The sample number n varies from 1,831 (Cardio) to 567,498 (HTTP-3), and the dimensionally ranges from 3 to 21. Cardio proportionally has the largest number of anomalies (9.61%), and only 0.03% of samples in SMTP-3 were in the anomaly category. In addition to the four datasets above, others are available including those in the ODDS Library [48], UCI repository [11], and DAMI Datasets [16] for future research.

Accordingly to the SEAD structure in Figure 1, the output scores of each detector are first normalized to [0,1). A sample with a higher score indicates a higher probability that this sample belongs to the anomaly category. Furthermore, with the anomaly percentage, or named contamination rate that the users know in advance, a threshold can be determined to translate anomaly scores to

Table 3. Datasets

Datasets	Sample Length	Dimension	Outliers	%Outliers
Cardio	1,831	21	176	9.61
Shuttle	49,097	9	3,511	7.15
SMTP-3	95,156	3	30	0.03
HTTP-3	567,498	3	2,211	0.40

Table 4. Hyper-parameter Set for the Three Detectors

Detector	*window size*	*Bins*	*CMS-w*	*CMS-MOD*	*K*
Loda	128	20	1	-	-
RS-Hash	128	-	2	128	-
xStream	128	-	2	128	20

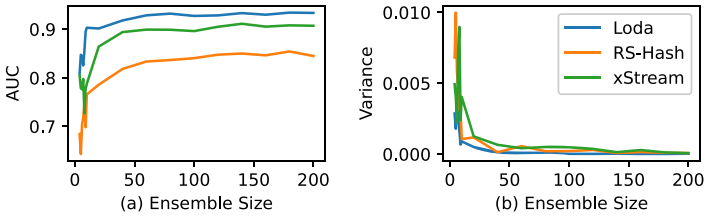


Fig. 10. Ensemble performance measured on dataset: Cardio.

binarized anomaly labels. For example, if the threshold is 0.8, then inputs with anomaly scores of 0.9 and 0.5 would be assigned to labels "1" (anomaly) and "0" (normal), respectively. The standard metric used for evaluating anomaly detectors is **Area Under the Curve (AUC)** of the **Receiver Operating Characteristics (ROC)** curve. It can be used for scores or labels and is described in detail in Reference [12]. We adopt this metric to analyze the accuracy of our implementations.

4.2 Sub-detectors and Ensemble Accuracy

We conducted an experiment to show the effectiveness of increasing the number of sub-detectors for each of the three supported methods in fSEAD.

For different ensemble sizes, the AUC of detectors in fSEAD and the variance of AUC are evaluated over 10 executions with different random seeds. The hyper-parameters were set to the values in Table 4, these being chosen to give an accurate and efficient hardware implementation. We refer the reader to the relevant papers regarding selecting these hyper-parameters [40, 45, 51].

Figure 10 shows ensemble sizes in the range [3, 200] for Loda, RS-Hash, and xStream, respectively. For simplicity, only the Cardio dataset is used to exhibit AUC performance. Figure 10(a) represents the mean AUC, and sub-figure (b) shows the variance.

For all detectors, AUC increases with increasing ensemble size before converging to a maximum; the AUC variance shows a decreasing trend before converging to a minimum. This translates to increasing the number of sub-detectors generally leading to a more reliable result, which follows the underlying principle of ensembles [13, 14].

Furthermore, Table 5 shows results for different heterogeneous combinations of detectors over our four benchmarks. The rows in this table are divided into two main parts, mean and variance of the AUC, with the columns divided into AUC of Score and Label results. A, B, and C in the second row indicate three detectors (Loda, RS-Hash, and xStream, respectively), and the numbers indicate

Table 5. Model Combination Comparison

AUC	Model	Score									Label								
		A7	**B7**	**C7**	**C223**	**C232**	**C322**	**C331**	**C313**	**C133**	**A7**	**B7**	**C7**	**C223**	**C232**	**C322**	**C331**	**C313**	**C133**
Mean	cardio	**0.933**	0.850	0.907	0.897	0.898	0.891	0.899	0.889	0.900	0.659	0.618	0.646	0.711	**0.721**	0.705	0.708	0.706	0.715
	shuttle	0.991	0.990	0.986	0.990	0.991	0.990	**0.992**	0.990	0.991	0.927	0.950	0.933	0.974	0.970	0.970	0.972	0.974	**0.976**
	smtp3	0.847	**0.856**	0.834	0.848	0.849	0.848	0.851	0.851	0.850	0.743	0.717	0.500	0.757	0.755	**0.770**	0.767	0.765	0.737
	http3	0.993	**0.995**	**0.995**	**0.995**	**0.995**	**0.995**	**0.995**	**0.995**	**0.995**	0.595	0.510	0.512	0.595	**0.620**	0.604	0.584	0.600	0.594
Variance ($\times 10^{-3}$)	cardio	**0.04**	0.2	0.05	0.07	0.06	0.09	0.06	0.23	0.05	0.13	0.14	0.09	**0.05**	0.15	0.14	0.15	0.31	0.21
	shuttle	0.005	**0.000**	0.092	0.001	0.002	0.001	**0.000**	0.002	0.001	0.41	0.06	3.04	0.03	0.07	0.06	**0.01**	0.03	0.02
	smtp3	**0.01**	0.05	1.75	0.08	0.03	0.06	0.04	0.04	0.02	0.62	**0.000**	**0.000**	1.96	1.95	0.93	0.89	1.03	0.71
	http3	0.0011	**0.0001**	0.0003	0.0003	**0.0001**	**0.0001**	**0.0001**	**0.0001**	0.0002	5.01	0.13	**0.12**	8.32	7.76	5.46	3.0	2.46	4.82

the number of pblocks used. The numbers of sub-detectors we use for each A, B, and C pblock are 35, 25, 20, respectively; details of this value will be described in Section 4.3. For example, A7, B7, and C7 indicate all seven pblocks are utilized for a single type of detector. The fourth one, C223, represents a combination of two pblocks assigned to Loda, two pblocks for RS-Hash, and the last three pblocks being occupied by xStream. We note that this experiment did not cover all possible combinations. However, the goal of this article is to provide a hardware framework that supports arbitrary model combinations; how to get the best combination scheme for a given benchmark is dataset-dependent and beyond the scope of this work. The best result for each dataset is in bold.

For the Cardio dataset, Table 5 shows that Loda always achieves the best Score AUC mean (0.933) and lower variance (0.00004) than RS-Hash, xStream or any listed combination strategy. However, in the label test, all combined labels can get a higher AUC than any single detector, albeit with higher variance.

In general, it can be seen that Loda, RS-Hash, and xStream perform differently for different datasets and there is no single "best" detector or "ideal" combination. Combined strategies are not always superior to a single detector. However, a combined detector typically yields more reliable performance. For labels, the combined detector always returns a higher AUC. We believe this is because if any detector indicates an anomaly, then the combined result is also an anomaly. This reduces the possibility of missing an anomaly and introducing a higher variance. We believe the strong dataset dependence coupled with the difficulty of finding the best combination of sub-detectors justifies the need to create a runtime reconfigurable framework that can support multiple detectors and multiple instances of each detector.

4.3 Pblock Assignment and FPGA Implementation

Table 6 shows a breakdown of resource utilization for the blocks in Figure 8. RP-1 to RP-7 account for the majority of the FPGA resources and are used for sub-detector implementation. The resource distribution of these blocks is not uniform, since the floorplan was manually created to pass the **Design Rule Check (DRC)** due to the nonuniform resources located on the target FPGA (ZCU111). The resources for blocks to combine the results, noted as combo-pblocks, are minimal, this being LUT (0.63%), DSP (0.75%), BRAM (0.80%), and FF (0.63%). Overall, the partial reconfigurable blocks and two static switch blocks utilize 57.73% LUT, 52.69% DSP, 55.37% BRAM, and 57.74% FF resources. The remaining area is used to implement the remaining interface, including DMAs, AXI-Interconnect, DFX-decoupler, and so on.

An important issue affecting achievable parallelism is how many sub-detectors can be assigned to a pblock. We determine this number by using the smallest pblock (RP-3) as the target and calculate the maximum ensemble size for Loda, RS-Hash, and xStream. This exercise led to ensemble size of $R = 35$ for Loda, $R = 25$ for RS-Hash, and $R = 20$ for xStream in each AD-pblock. The resources required are shown in Table 7. Thus, if we utilize all seven AD-pblocks to implement a homogeneous type of detector, then the current configuration supports a maximum of 245 Loda, 175 RS-Hash, or 140 xStream sub-detectors.

Table 6. Resource Partition of FPGA Blocks

Blocks	LUT	DSP	BRAM	FF
RP-1	6.73%	4.49%	6.67%	6.73%
RP-2	8.57%	7.54%	8.52%	8.57%
RP-3	6.24%	6.46%	6.39%	6.24%
RP-4	6.72%	4.49%	6.67%	6.72%
RP-5	6.24%	6.46%	6.39%	6.24%
RP-6	8.74%	8.24%	8.15%	8.74%
RP-7	7.32%	7.30%	7.22%	7.32%
RP-combo1	0.72%	0.56%	0.74%	0.72%
RP-combo2	0.59%	0.84%	0.83%	0.59%
RP-combo3	0.59%	0.84%	0.83%	0.59%
Switch-1	3.46%	4.49%	2.96%	3.46%
Switch-2	1.81%	0.98%	0%	1.82%
DMAs	2.25%	0%	1.30%	0.48%
DFX-Decouplers	0.04%	0%	0%	0.008%
AXI-InterConnect	0.67%	0%	0%	0.58%
AXI-SmartConnect	2.41%	0%	0%	1.61%
ALL	62.5%	52.69%	56.67%	60.42%

Table 7. Resource of 35 Loda, 25 RS-Hash, and 20 xStream for Cardio

Detector	LUT	DSP	BRAM	FF
Loda-35	16,783 (63.4%)	122 (44.2%)	**54.5 (79.0%)**	11,478 (21.7%)
RS-Hash-25	**23,732 (89.6%)**	68 (24.6%)	50 (72.5%)	14,012 (26.5%)
xStream-20	**23,908 (90.3%)**	80 (29.0%)	60 (87.0%)	12,617 (23.8%)
RP-3	**26,480**	**276**	**69**	**52,960**

4.4 Speed, Accuracy, and Power Comparison

We achieve 80%–90% logic use of all seven partial blocks in fSEAD on the ZCU111 board for homogeneous detect implementations of Loda (245 sub-detectors), RS-Hash (175 sub-detectors), and xStream (140 sub-detectors). This configuration was configured using the PYNQ environment for testing of fSEAD performance. The FPGA was operated at a clock rate of 188 MHz. We also implemented a multi-threaded GCC version of the detectors with the same parameters and scale as fSEAD for comparison.

The detector system in fSEAD is configured with the topology shown in Figure 7(c). An averaging-based combination and an OR-Gate-based label combination techniques were used. The ap_fixed<32,16,AP_TRN,AP_WRAP> type available in Xilinx Vivado HLS [28] was used for all inner non-integer operations. This has a word-length of 32-bit with 16 bits representing the integer part and 16 bits representing the fraction. Convergent rounding (AP_TRN) and saturating arithmetic (AP_WRAP) were used. To facilitate the use of the float32 type in the NUMPY library [17] for streaming data transfer with each module on the FPGA, all fSEAD IP interfaces are converted to float32. This does not cause a significant increase in resource consumption.

Tables 8 through 10 show the performance comparison of Loda, RS-Hash, and xStream detectors in terms of AUC and execution time on the ZCU111 board and CPU, respectively. The execution time test uses the *time()* function in Python to measure the time from the start of the input DMA transfer to when all data is obtained from the output DMA. The GCC implementation was executed on a multi-threaded CPU on the target PC, and the *time* command in *bash* used to measure execution time. The pthread library is used to support the multi-threaded C implementation. Considering that each sub-detector in the ensemble is data-independent, we equally distribute the same

Table 8. Accuracy and Execution Time Comparison between fSEAD and CPU for Detector: Loda

Dataset	AUC-S(CPU)	**AUC-S(FPGA)**	AUC-L(CPU)	**AUC-L(FPGA)**	Ex Time(CPU)	**Ex Time(FPGA)**	Speedup
Cardio	0.9310	0.9311	0.6447	0.6412	13 ms	4.63 ms	2.81×
Shuttle	0.9923	0.9914	0.9490	0.9432	147 ms	34.23 ms	4.29×
SMTP-3	0.8501	0.8506	0.7666	0.7499	222 ms	39.31 ms	5.65×
HTTP-3	0.9937	0.9936	0.6415	0.6336	1,396 ms	228.25 ms	6.12×

Table 9. Accuracy and Execution Time Comparison between fSEAD and CPU for Detector: RS-Hash

Dataset	AUC-S(CPU)	**AUC-S(FPGA)**	AUC-L(CPU)	**AUC-L(FPGA)**	Ex Time(CPU)	**Ex Time(FPGA)**	Speedup
Cardio	0.8546	0.8524	0.6310	0.6274	15 ms	4.87 ms	3.08×
Shuttle	0.9915	0.9910	0.9543	0.9560	168 ms	35.80 ms	4.69×
SMTP-3	0.8525	0.8513	0.7166	0.7166	260 ms	39.63 ms	6.56×
HTTP-3	0.9944	0.9944	0.5065	0.5067	1,490 ms	228.29 ms	6.53×

Table 10. Accuracy and Execution Time Comparison between fSEAD and CPU for Detector: xStream

Dataset	AUC-S(CPU)	**AUC-S(FPGA)**	AUC-L(CPU)	**AUC-L(FPGA)**	Ex Time(CPU)	**Ex Time(FPGA)**	Speedup
Cardio	0.9229	0.9222	0.6467	0.6435	18 ms	4.82 ms	3.73×
Shuttle	0.9914	0.9905	0.9688	0.9680	250 ms	40.62 ms	6.15×
SMTP-3	0.8077	0.8076	0.7167	0.7167	366 ms	50.99 ms	7.18×
HTTP-3	0.9947	0.9948	0.5067	0.5069	2,460 ms	297.85 ms	8.26×

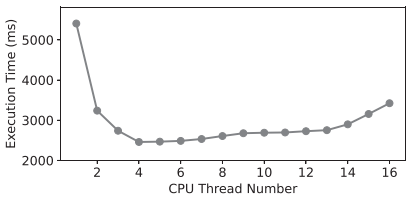


Fig. 11. Multi-threaded CPU implementation for xStream for HTTP-3.

number of sub-detectors to each CPU thread. The sub-score computed by multiple threads requires a synchronization operation at the end of each sample to compute the average score of the ensemble. The *pthread_mutex_lock* and *pthread_mutex_unlock* functions are placed between different threads to guarantee the streaming mode execution. The target CPU has 8 cores and 16 hardware threads; we conduct a test of switching the thread number from 1 to 16. Figure 11 shows the speedup results with different numbers of threads on the longest test-case: xStream for HTTP-3, 4-thread always gets the best speedup. We believe this is due to synchronization overheads introduced by the *mutex* scheme limiting performance when the number of threads is greater than 4 (*mutex* is called in every streaming execution). Based on this result, all the following CPU experiment results are measured from 4-thread configuration.

In terms of accuracy, very similar scores were obtained on CPU and FPGA platforms. This indicates the ap_fixed<32,16> variable type used can provide sufficient accuracy as the float32 variable type of CPU implementation. The minor differences are due to the accumulation of errors during the cumulative calculation of the accuracy differences generated by each ap_fixed and float32 type sub-detector. Future developers may wish to explore minimizing resource consumption further by reducing precision.

On the smallest dataset, Cardio, it can be observed that the FPGA obtains a speedup of 2.81 for Loda, 3.08 for RS-Hash, and 3.73 times for xStream. The speedup ratio increases as the size and

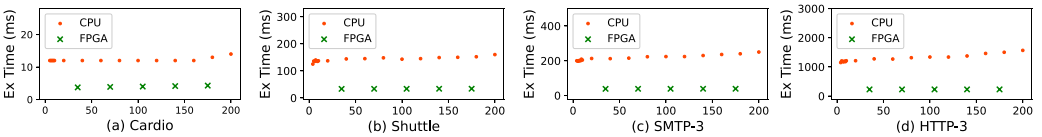


Fig. 12. Execution time comparison of fSEAD and CPU for detector: Loda.

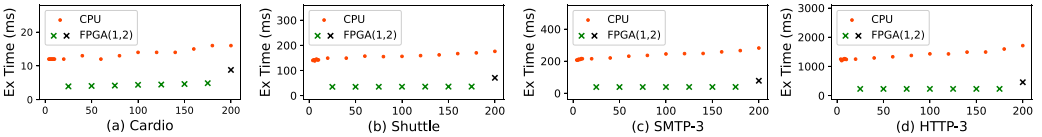


Fig. 13. Execution time comparison of fSEAD and CPU for detector: RS-Hash.

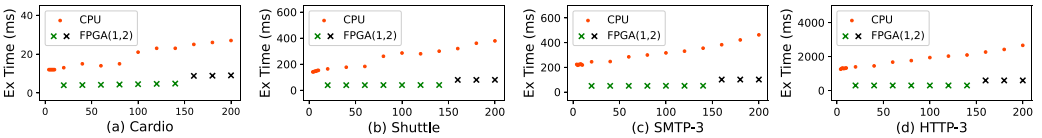


Fig. 14. Execution time comparison of fSEAD and CPU for detector: xStream.

dimension of the dataset increases. On the largest dataset, HTTP-3, the three detectors achieve speedups of 6.12, 6.53, and 8.26, respectively. The highest speedup, 8.26, is achieved by xStream for dataset HTTP-3. For smaller datasets, the transfer time from the Linux OS-based host ARM processor to the FPGA becomes the bottleneck. As fSEAD is optimized for large datasets, this is not a significant issue.

It is worth mentioning that, since the ensemble on fSEAD is based on sub-detector-level parallelism, its latency is not significantly affected by the ensemble size in the case that FPGA resources are sufficient to implement the required ensemble in parallel. In contrast, the ensemble implementation on GCC uses a *for* loop to iterate the computation of each sub-detector, so its execution time increases proportionally with the increase of the ensemble size. Figures 12 to 14 show the execution times of the three detectors on the CPU and FPGA for multiple sets of experiments and different ensemble sizes. The red dots indicating CPU execution time show a linear increase with ensemble size. The green crosses indicate the execution time on FPGA, while the black crosses are the results of two FPGA executions. In all cases, significant speedups are achieved over the CPU and limited only by FPGA resources.

We also use the roofline model to estimate the degree to which our target anomaly detector methods have been optimized. This describes an application's achieved performance and arithmetic intensity against the machine's maximum achievable performance in terms of memory bandwidth and peak computational performance. The CPU roofline model is measured on the Intel Advisor 2022 [22]. **Billions of operations per second (GOPS)** is used as the metric for Performance (y axis in Figure 15) and **operations (OPs)** per byte is used for Arithmetic intensity (x axis in Figure 15). The L1/DRAM bandwidth roofline represents the maximum amount of bytes that can get read or written for a given arithmetic intensity. For FPGA roofline models, we use the method in Reference [42] to calculate the roofline chart in Figure 16. The arithmetic intensity (x axis) is the number of arithmetic operations performed for each byte read or written to off-chip memory (we assume 13.4 GB/s off-chip memory bandwidth [23]). The performance (y axis) is calculated in GOPS.

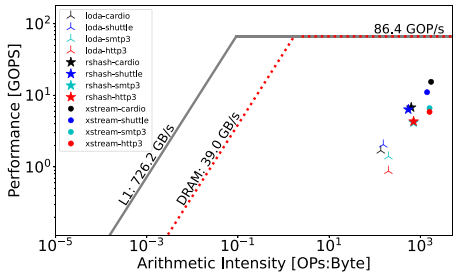


Fig. 15. Roofline model on CPU.

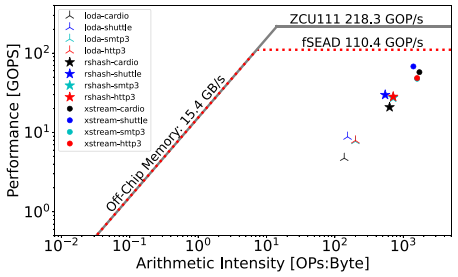


Fig. 16. Roofline model on FPGA.

Table 11. Operation Number of fSEAD

Detectors	Operation Number
Loda	$OP = N * (2Rd + 7R + 2)$
RS-Hash	$OP = N * (5Rdw + 4Rd + 11Rw + R + 2)$
xStream	$OP = N * (2Rdk + 5Rdw + 15Rw + 2R + 2)$

[1] N: the length of the input dataset.
[2] d: the dimension of input dataset.
[3] R: the ensemble size.
[4] w: the hash functions number in CMS.
[5] k: the projection size of xStream.

Table 12. GOPS Comparison of CPU and fSEAD

GOPS	CPU				fSEAD			
	Cardio	**Shuttle**	**SMTP-3**	**HTTP-3**	**Cardio**	**Shuttle**	**SMTP-3**	**HTTP-3**
Loda	1.690	2.049	1.402	0.776	4.748	8.789	7.924	4.748
RS-Hash	6.772	6.353	4.197	4.331	20.858	29.797	27.533	28.282
xStream	15.427	11.050	6.623	5.878	57.544	67.959	47.554	48.551

For the three detectors, Table 11 shows the expressions we used to estimate the total number of operations for the execution of a target dataset. Using this and the execution time from Table 8 to Table 10, we compute the GOPS in Table 12. We use the highest GOPS (67.959 GOPS of xStream for Shuttle) to estimate the compute-bound performance for the target FPGA (ZCU111) and fSEAD structure, respectively. The xStream for Shuttle occupies 132,391 LUTs, 476 DSPs, and 79,485 FFs, which takes up 31.13% of the total FPGA resources and 61.57% of the fSEAD partial blocks resources. From this, we calculate the compute-bound performance for FPGA and fSEAD as 218.3 GOP/s and 110.4 GOP/s, respectively.

While no algorithms reach the boundary of the roofline, xStream is closer to the computational boundary than Loda and RS-Hash, as seen in Figures 15 and 16. We believe this is a function of the algorithms: The current three detectors are not extremely computationally intensive, but xStream has more vector and matrix multiplication operations among the current AD library.

Figure 17 shows scalability of a single pblock, RP-1, using the Cardio dataset with 20% to 80% resource utilization for Loda, RS-Hash, and xStream. Working at the same 188 MHz clock frequency, the throughput of three detectors can be seen to vary linearly with the resource utilization of RP-1. This linear scalability enables one to quickly identify potential ensembles that will fit on a given FPGA and estimate their throughput. One could then perform software experiments, such as those in Table 5, to identify which of these potential ensembles provides the best accuracy for the target dataset.

The power consumption of fSEAD is separated into chip power and system power, Figures 18 and 19 show the measured chip and system power consumption from Xilinx Vivado Tool [29]

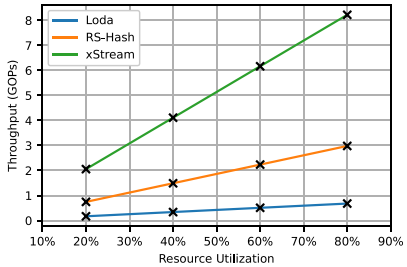


Fig. 17. Example of the scalability inside single partial block: RP-1.

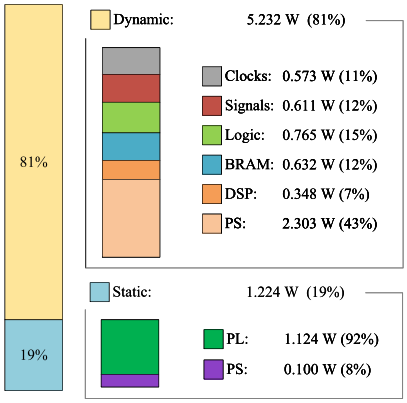


Fig. 18. Chip power consumption.

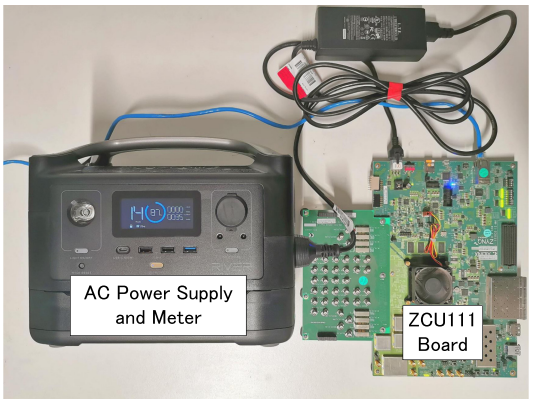


Fig. 19. System power test-bed.

and EcoFlow RIVER Max Portable Power Station [21], respectively. In Figure 19, the idle system power measured on EcoFlow is 30 Watts and the working system power (35 Watts) is gained by configuring the PYNQ to invoke all pblocks for xStream with the biggest dataset HTTP-3, which matches the dynamic power of 5.232 Watts, as demonstrated in Figure 18. Intel(R) Core(TM) i7-10700F CPU power is measured with the powerstat command using **Running Average Power Limit (RAPL)** domains. 120 samples of power measurements with 0.5-second step are averaged over the duration of one minute time. The CPU idle power is 7.90 Watts, and the CPU working power for xStream with the biggest dataset HTTP-3 is 51.23 Watts. The dynamic CPU power (43.33 Watts) is more than 8× higher than the fSEAD dynamic power consumption on ZCU111 FPGA (5.232 Watts).

4.5 Partial Reconfiguration

Although the reconfiguration of each partial region in fSEAD can often be done when the system is idle, we measure the overhead of partial reconfiguration in the PYNQ framework. The horizontal axis of Table 13 shows all pblocks, and the vertical axis shows the direction of reconfiguration for bitstream downloads. For example, *Function* → *Identity* indicates that the configured logic is *Function* that is overwritten with *Identity*. We have chosen a common function module: *Loda_Cardio* and *Averaging* for (RP-1 to RP-7) and (COMBO1 to COMBO3), respectively, as the *Function* module. *Identity* is a design where the input is simply passed to the output. The objective is to evaluate the impact of the hardware logic size of the target bitstream on the reconfiguration time cost. Referring to the size of each pblock provided in Table 6 (all seven AD-pblock resources are larger than those of COMBO1 to COMBO3; among the seven AD-pblocks, RP-3 has the least

Table 13. Partial Reconfiguration Time Cost (Unit is ms) of the Different Pblocks

BIT download directions	RP-1	RP-2	RP-3	RP-4	RP-5	RP-6	RP-7	COMBO1	COMBO2	COMBO3
Function → *Identity*	607.8	606.1	604.5	606.1	608.9	**609.6**	609.5	587.2	582.7	**579.8**
Identity → *Function*	606.3	611.3	607.2	606.0	606.9	608.1	607.5	582.9	580.1	581.9

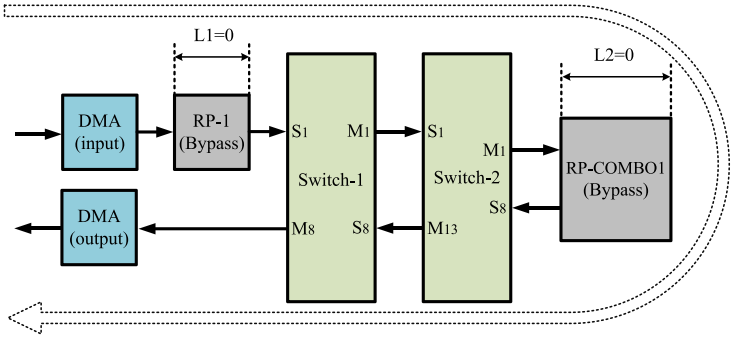


Fig. 20. Example of fSEAD channel with empty logic.

programmable resources and RP-6 the most; while COMBO1 occupies the largest area of the three COMBO blocks).

The latencies reported in Table 13 are the total partial reconfiguration latency. This will be a function of the size and location of the region, as well as the size of the partial bitstream and the chosen FPGA. Since we do not have the ability to modify the chosen FPGA, in this section, we explore how these other factors impact reconfiguration time. The results in Table 13 show that the pblock with larger area takes a little more time to reconfigure, e.g., RP-6 takes 609.6 ms for updating *Loda_Cardio* with *Identity*. The pblock with the smallest area uses the shortest time i.e., COMBO3 used 579.8 ms to reconfigure *Loda_Cardio* to *Identity*. In addition, the complexity of the target bitstream has a slight impact on the reconfiguration time. Apart from RP-2, RP-3, and COMBO3, a general trend indicates that the simpler the logic of the target bitstream (in this case, *Identity*), the lower the reconfiguration time.

While fSEAD is focused on ensemble-centric anomaly detection, different real-life detectors in the literature can be incorporated into this library for higher flexibility and salable combinations. Moreover, it has the potential to be used for more general machine learning applications by adding different modules. To evaluate the default latency of fSEAD interconnections, since this is a key metric that could affect design decisions, Figure 20 illustrates a data-path where each pblock simply copies its input to output (the "Bypass" in Figure 20 is the same as *Identity* in Table 13). The execution time, which is a measure of system latency overhead, is **0.80 ms**. Thus for pblocks with latency L1 on the left and L2 on the right, the maximum latency of the system is ≈ **0.80+L1+L2 (ms)**. We also measure the shorter DMA latency where the data-path include: DMA (input), L1, Switch-1, and DMA (output), the latency is **0.77 ms**. This reflects that the default system latency is dominated by the Linux OS-based PYNQ framework, rather than by the routing latency of switches.

5 CONCLUSIONS AND FUTURE WORK

In this article, we proposed a flexible computing architecture consisting of multiple partially reconfigurable regions, called pblocks, which are interconnected via the AXI-Streaming Switches. The scheme enables parallel constructions of repeating blocks to be easily scaled to fill the FPGA. We also demonstrated a concrete application of this architecture to **streaming anomaly detection**

using ensembles (fSEAD). fSEAD allows complex and more powerful anomaly detectors to be composed from simple pblocks in an arbitrary fashion. Careful floorplanning of the pblocks and switches, together with bus-level pipelining, minimize routing delay and allow timing closure to be achieved. Through a number of experiments involving ensembles of three sub-detectors (Loda, RS-Hash, and xStream), we demonstrate 3 to 8× speedup compared with a CPU.

Future work will extend this architecture in several directions. First, continuing with the anomaly detector application, we will create module generators that can automatically generate optimized HLS directives and detector parameters such as ensemble size. We will also support more anomaly detection methods to enrich the existing library. Finally, we plan to use the proposed composable architecture for a wider range of applications, e.g., classification and regression tasks.

REFERENCES

[1] Elke Achtert, Hans-Peter Kriegel, Lisa Reichert, Erich Schubert, Remigius Wojdanowski, and Arthur Zimek. 2010. Visual evaluation of outlier detection models. In *Proceedings of the International Conference on Database Systems for Advanced Applications*. Springer, 396–399.

[2] Charu C. Aggarwal and Saket Sathe. 2015. Theoretical foundations and algorithms for outlier ensembles. *ACM SIGKDD Explor. Newslett.* 17, 1 (2015), 24–47.

[3] Mohiuddin Ahmed, Abdun Naser Mahmood, and Md Rafiqul Islam. 2016. A survey of anomaly detection techniques in financial domain. *Fut. Gen. Comput. Syst.* 55 (2016), 278–288.

[4] Luis Basora, Xavier Olive, and Thomas Dubot. 2019. Recent advances in anomaly detection methods applied to aviation. *Aerospace* 6, 11 (2019), 117.

[5] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. 2000. LOF: Identifying density-based local outliers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 93–104.

[6] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. 2020. Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq Ultrascale+. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT'20)*. IEEE, 215–220.

[7] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM Comput. Surv.* 41, 3 (2009), 1–58.

[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. ACM, New York, NY, 785–794. DOI: https://doi.org/10.1145/2939672.2939785

[9] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: The count-min sketch and its applications. *J. Algor.* 55, 1 (2005), 58–75.

[10] Abhishek Das, David Nguyen, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. 2008. An FPGA-based network intrusion detection architecture. *IEEE Trans. Inf. Forens. Secur.* 3, 1 (2008), 118–132.

[11] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. Retrieved from http://archive.ics.uci.edu/ml.

[12] David Faraggi and Benjamin Reiser. 2002. Estimation of the area under the ROC curve. *Statist. Med.* 21, 20 (2002), 3093–3106.

[13] Yoav Freund and Robert E. Schapire. 1996. Experiments with a new boosting algorithm. In *Proceedings of the International Conference on Machine Learning*. Citeseer, 148–156.

[14] Yoav Freund and Robert E. Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.* 55, 1 (1997), 119–139.

[15] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. 2009. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Comput. Secur.* 28, 1–2 (2009), 18–28.

[16] G. O. Campos and A. Zimek, et al. 2016. DAMI Datasets. Retrieved from https://www.dbs.ifi.lmu.de/research/outlier-evaluation/DAMI.

[17] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. DOI: https://doi.org/10.1038/s41586-020-2649-2

[18] Ami Hayashi and Hiroki Matsutani. 2017. An FPGA-based In-NIC cache approach for lazy learning outlier filtering. In *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 15–22.

[19] Tin Kam Ho, Jonathan J. Hull, and Sargur N. Srihari. 1994. Decision combination in multiple classifier systems. *IEEE Trans. Pattern Anal. Mach. Intell.* 16, 1 (1994), 66–75.
[20] Markus Hofmann and Ralf Klinkenberg. 2016. *RapidMiner: Data Mining Use Cases and Business Analytics Applications.* CRC Press, New York.
[21] Ecoflow Inc. 2022. River Max Portable Power Station. Retrieved from https://au.ecoflow.com/products/river-max-portable-power-station?variant=40043438211270.
[22] Intel Inc. 2022. Intel Advisor. Retrieved from https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-advisor/top.html.
[23] Xilinx Inc. 2018. ZCU111 Evaluation Board User Guide. Retrieved from https://docs.xilinx.com/v/u/en-US/ug1271-zcu111-eval-bd.
[24] Xilinx Inc. 2020. AXI4-Stream Infrastructure IP Suite v3.0 Product Guide (PG085). Retrieved from https://docs.xilinx.com/v/u/en-US/pg085-axi4stream-infrastructure.
[25] Xilinx Inc. 2020. AXI4-Stream Interconnect v1.1 Product Guide (PG035). Retrieved from https://docs.xilinx.com/v/u/en-US/pg035_axis_interconnect.
[26] Xilinx Inc. 2020. Xilinx. Vivado Design Suite Tutorial: Partial Reconfiguration (UG947). Retrieved from https://docs.xilinx.com/r/en-US/ug947-vivado-partial-reconfiguration-tutorial.
[27] Xilinx Inc. 2020. Xilinx. Vivado Design Suite User Guide: Dynamic Function eXchange (UG909). Retrieved from https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration.
[28] Xilinx Inc. 2020. Xilinx. Vivado Design Suite User Guide: High-Level Synthesis (UG902). Retrieved from https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis.
[29] Xilinx Inc. 2020. Xilinx. Vivado Design Suite User Guide: Power Analysis and Optimization (UG907). Retrieved from https://docs.xilinx.com/v/u/2020.1-English/ug907-vivado-power-analysis-optimization.
[30] Xilinx Inc. 2022. Python Productivity for Zynq. Retrieved from http://www.pynq.io/home.html.
[31] Akira Jinguji, Shimpei Sato, and Hiroki Nakahara. 2018. An FPGA realization of a random forest with k-means clustering using a high-level synthesis design. *IEICE Trans. Inf. Syst.* 101, 2 (2018), 354–362.
[32] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A highly efficient gradient boosting decision tree. *Adv. Neural Inf. Process. Syst.* 30 (2017), 3146–3154.
[33] Lukasz Komsta and Maintainer Lukasz Komsta. 2011. Package outliers. *Med. Univ. Lublin, Lublin* (2011). https://cran.r-project.org/web/packages/outliers/outliers.pdf.
[34] G. Korcyl and P. Korcyl. 2020. Optimized Implementation of the Conjugate Gradient Algorithm for FPGA-based Platforms using the Dirac-Wilson Operator as an Example. Retrieved from https://arxiv.org/abs/2001.05218.
[35] Aleksandar Lazarevic and Vipin Kumar. 2005. Feature bagging for outlier detection. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. Association for Computing Machinery, New York, NY, 157–166.
[36] Quoc Le, Tamás Sarlós, and Alex Smola. 2013. Fastfood-approximating kernel expansions in loglinear time. In *Proceedings of the International Conference on Machine Learning*.
[37] Xiang Lin, R. D. Shawn Blanton, and Donald E. Thomas. 2017. Random forest architectures on FPGA for multiple applications. In *Proceedings of the Great Lakes Symposium on VLSI*. Association for Computing Machinery, New York, NY, 415–418.
[38] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *Proceedings of the 8th IEEE International Conference on Data Mining*. IEEE Computer Society, 413–422.
[39] Junshui Ma and Simon Perkins. 2007. Online novelty detection on temporal sequences. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, New York, NY, 613–618.
[40] Emaad Manzoor, Hemank Lamba, and Leman Akoglu. 2018. xStream: Outlier detection in feature-evolving data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Association for Computing Machinery, New York, NY, 1963–1972.
[41] Duncan J. M. Moss, David Boland, Peyam Pourbeik, and Philip H. W. Leong. 2018. Real-time FPGA-based anomaly detection for radio frequency signals. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'18)*. IEEE, 1–5.
[42] Servesh Muralidharan, Kenneth O'Brien, and Christian Lalanne. 2015. A semi-automated tool flow for roofline anaylsis of OpenCL kernels on accelerators. In *Proceedings of the 1st International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'15)*.
[43] Yeyong Pang, Shaojun Wang, Yu Peng, Nicholas J. Fraser, and Philip H. W. Leong. 2013. A low latency kernel recursive least squares processor using FPGA technology. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'13)*. IEEE, 144–151.

[44] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* 12, Oct. (2011), 2825–2830.
[45] Tomáš Pevný. 2016. Loda: Lightweight on-line detector of anomalies. *Mach. Learn.* 102, 2 (2016), 275–304.
[46] Dragoljub Pokrajac, Aleksandar Lazarevic, and Longin Jan Latecki. 2007. Incremental local outlier detection for data streams. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining*. IEEE, 504–515.
[47] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. 2000. Efficient algorithms for mining outliers from large data sets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, New York, NY, 427–438.
[48] Shebuti Rayana. 2016. ODDS Library. Retrieved from http://odds.cs.stonybrook.edu.
[49] Mayu Sakurada and Takehisa Yairi. 2014. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA 2nd Workshop on Machine Learning for Sensory Data Analysis*. Association for Computing Machinery, New York, NY, 4–11.
[50] Osman Salem, Alexey Guerassimov, Ahmed Mehaoua, Anthony Marcus, and Borko Furht. 2014. Anomaly detection in medical wireless sensor networks using SVM and linear regression models. *Int. J. E-Health Med. Commun.* 5, 1 (2014), 20–45.
[51] Saket Sathe and Charu C. Aggarwal. 2016. Subspace outlier detection in linear time with randomized hashing. In *Proceedings of the IEEE 16th International Conference on Data Mining (ICDM'16)*. IEEE, 459–468.
[52] Thomas Schlegl, Philipp Seeböck, Sebastian M. Waldstein, Georg Langs, and Ursula Schmidt-Erfurth. 2019. f-AnoGAN: Fast unsupervised anomaly detection with generative adversarial networks. *Med. Image Anal.* 54 (2019), 30–44.
[53] Mei-Ling Shyu, Shu-Ching Chen, Kanoksri Sarinnapakorn, and LiWu Chang. 2003. *A Novel Anomaly Detection Scheme Based on Principal Component Classifier*. Report. Miami University, Coral Gables, Florida Department of Electrical and Computer Engineering.
[54] Swee Chuan Tan, Kai Ming Ting, and Tony Fei Liu. 2011. Fast anomaly detection for streaming data. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*. AAAI Press, 1511–1516.
[55] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A. Ghorbani. 2009. A detailed analysis of the KDD CUP 99 data set. In *Proceedings of the IEEE Symposium on Computational Intelligence for Security and Defense Applications*. IEEE, Ottawa, 1–6.
[56] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. 2012. Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA? In *Proceedings of the IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 232–239.
[57] Andrew Elbert Wilson. 2020. *Dynamic Reconfigurable Real-time Video Processing Pipelines on SRAM-Based FPGAs*. Thesis. Brigham Young University.
[58] Ke Wu, Kun Zhang, Wei Fan, Andrea Edwards, and Philip S. Yu. 2014. RS-Forest: A rapid density estimator for streaming anomaly detection. In *Proceedings of the IEEE International Conference on Data Mining*. IEEE Computer Society, 600–609. DOI: https://doi.org/10.1109/ICDM.2014.45
[59] Selim F. Yilmaz and Suleyman Serdar Kozat. 2020. PySAD: A streaming anomaly detection framework in Python. *ArXiv* abs/2009.02572 (2020).
[60] Yuan Yuan, Jianwu Fang, and Qi Wang. 2014. Online anomaly detection in crowd scenes via structure analysis. *IEEE Trans. Cyber.* 45, 3 (2014), 548–561.
[61] Yue Zhao, Xiyang Hu, Cheng Cheng, Cong Wang, Changlin Wan, Wen Wang, Jianing Yang, Haoping Bai, Zheng Li, and Cao Xiao. 2021. SUOD: Accelerating large-scale unsupervised heterogeneous outlier detection. *Proc. Mach. Learn. Syst.* 3 (2021), 463–478.
[62] Yue Zhao, Zain Nasrullah, Maciej K. Hryniewicki, and Zheng Li. 2019. Locally selective combination in parallel outlier ensembles. In *Proceedings of the SIAM International Conference on Data Mining*. SIAM, 585–593. DOI: https://doi.org/10.1137/1.9781611975673.66
[63] Yue Zhao, Zain Nasrullah, and Zheng Li. 2019. PyOD: A Python toolbox for scalable outlier detection. *J. Mach. Learn. Res.* 20, 96 (2019), 1–7. Retrieved from http://jmlr.org/papers/v20/19-011.html.
[64] Yue Zhao, Xuejian Wang, Cheng Cheng, and Xueying Ding. 2020. Combining machine learning models using Combo library. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, New York, 13648–13649.
[65] Arthur Zimek, Ricardo J. G. B. Campello, and Jörg Sander. 2014. Ensembles for unsupervised outlier detection: Challenges and research questions a position paper. *ACM SIGKDD Explor. Newslett.* 15, 1 (2014), 11–22.
[66] Indre Zliobaite, Mykola Pechenizkiy, and Joao Gama. 2016. *An Overview of Concept Drift Applications*. Springer International Publishing AG, Switzerland, 91–114. DOI: https://doi.org/10.1007/978-3-319-26989-4

Received 17 June 2022; revised 22 August 2022; accepted 30 September 2022