EHDSktch: A Generic Low Power Architecture for Sketching in Energy Harvesting Devices

Priyanka Singla
Indian Institute of Technology Delhi
New Delhi, India
priyanka@cse.iitd.ac.in

Chandran Goodchild
University of Freiburg
Freiburg, Germany
ch.goodchild@gmail.com

Smruti R. Sarangi
Indian Institute of Technology Delhi
New Delhi, India
srsarangi@cse.iitd.ac.in

ABSTRACT

Energy harvesting devices (EHDs) are becoming extremely prevalent in remote and hazardous environments. They sense the ambient parameters and compute some statistics on them, which are then sent to a remote server. Due to the resource-constrained nature of EHDs, it is challenging to perform exact computations on streaming data; however, if we are willing to tolerate a slight amount of inaccuracy, we can leverage the power of sketching algorithms to provide quick answers with significantly lower energy consumption.

In this paper, we propose a novel hardware architecture called EHDSktch – a set of IP blocks that can be used to implement most of the popular sketching algorithms. We demonstrate an energy savings of 4-10X and a speedup of more than 10X over state-of-the-art software implementations. Leveraging the temporal locality further provides us a performance gain of 3-20% in energy and time and reduces the on-chip memory requirement by at least 50-75%.

CCS CONCEPTS

• **Hardware** → **Hardware accelerators**; • **Computer systems organization** → **Embedded systems**; *System on a chip.*

KEYWORDS

Streaming algorithms, Hardware for sketching, Approximate computing

ACM Reference Format:
Priyanka Singla, Chandran Goodchild, and Smruti R. Sarangi. 2021. EHDSktch: A Generic Low Power Architecture for Sketching in Energy Harvesting Devices. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431447

1 INTRODUCTION

Energy harvesting devices (EHDs) are low-powered batteryless devices that harness the ambient energy and create an autonomous system where correctness is guaranteed by elaborate checkpoint-restore mechanisms [12]. These devices are used in diverse areas such as smart homes, hazardous environments, and hard-to-reach remote locations. Many novel applications are in the fields of wildlife monitoring and structural health monitoring. Most of these applications primarily deal with a large amount of continuously produced data that needs to be processed in real-time [13]. In particular, the EHDs sense ambient parameters such as the temperature, location of animals, and subsequently, either make local decisions like switching off the air conditioner in the room or analyze the data locally to compute some simple statistics such as the mean, median, L2 norm, unique elements, and frequencies of elements, or detect anomalies. These statistics are used to identify trends and spot peculiar behavior in the incoming data stream, and are periodically transmitted to remote base stations [5]. Efficiently handling these data streams is challenging due to the limited computational and memory resources of such devices – around 2KB of SRAM and 64-128KB of FRAM (e.g., TI's MSP430FR5969).

Typically, computing exact answers for many simple queries is not feasible on such tiny devices. Hence, we propose to use a family of algorithms known as *sketching algorithms* [1, 2, 8] that make a single pass on the data and compute approximate statistics with a slight loss in accuracy. Such algorithms are already commonplace in the theoretical computer science literature and also have seen diverse implementations in big data systems that typically rely on FPGAs or manycore processors [7, 11, 16].

It is not trivial to translate an FPGA implementation into an ASIC implementation [10]. The resource-constrained nature of the EHDs further exacerbates the issue. FPGAs rely on large block RAMs and LUTs that can be used for indexing; in comparison, EHDs have very little memory – a few KBs on-chip. ❶ Hence, we propose a *generic* template for designing such sketching systems on ASICs; it can be further specialized for individual algorithms. This is our first contribution. ❷ Second, we show a minimum 4X reduction in energy and a concomitant 10X decrease in execution time vis-a-vis an optimized software implementation of these sketching algorithms. ❸ Third, we show that it is possible to reduce the on-chip memory requirements by 50-75%, by including a small, specially designed cache. This supplementary structure can effectively help us shrink the on-chip SRAM from 2 KB to 1 KB or 512 bytes depending upon the desired accuracy. ❹ Furthermore, instead of checkpointing the contents of all the volatile memory structures, including the SRAM arrays, it is possible to augment this mechanism to significantly reduce the amount of state that needs to be periodically checkpointed.

2 BACKGROUND

Data Streams and Sketching A *stream* (X) is a sequence of N data values, i.e., $X = (x_1, x_2, \ldots x_N)$. In our setting, the values represent the readings taken by a sensor. We consider the space of approximate computing methods on streams, notably *sketching*. Sketching

ASPDAC '21, January 18–21, 2021, Tokyo, Japan
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431447

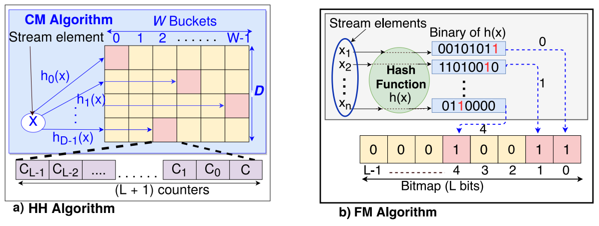

Figure 1: Sketching Algorithms (a) CM and HH and (b) FM

Table 1: Characteristics of different sketching algorithms

Sketch	Time per Update	Time per Query	Space Complexity	Error per Query
CM[1]	$\mathcal{O}(\mathcal{D})$	$\mathcal{O}(\mathcal{D})$	$\mathcal{O}(\mathcal{W}.\mathcal{D})$	$\frac{1}{N} \cdot \lvert N_x - \hat{N}_x \rvert$
N_x: Actual count of x, $\hat{N}_x$: Estimated count, N: Stream size, $\mathcal{W}$: # of buckets, $\mathcal{D}$: # of hashes				
HH[2]	$\mathcal{O}(\mathcal{D}.L)$	$\mathcal{O}(\mathcal{W}.\mathcal{D}.L)$	$\mathcal{O}(\mathcal{W}.\mathcal{D}.L)$	$\frac{1}{N} \cdot \lvert N_a - N_e \rvert$
N_a, N_e : Actual and estimated number of elements with count $\geq N/(k+1)$, k: threshold				
FM[8]	$\mathcal{O}(L)$	$\mathcal{O}(L)$	$\mathcal{O}(L)$	$\frac{1}{N_d} \cdot \lvert N_d - \hat{N}_d \rvert$
L: *BITMAP* size (in bits), N_d,$\hat{N}_d$: Actual and estimated # of distinct elements				

algorithms have the following characteristics: ❶ limited memory usage by the data structure (known as the *sketch* of the algorithm), ❷ generally require only a few passes over the data, and ❸ provide an approximate response to a query very quickly. It is often possible to make theoretical guarantees with regard to the maximum error. In this paper, we consider some of the most common sketching algorithms, namely CountMin (CM)[1], Heavy-Hitters (HH)[2], and Flajolet-Martin (FM)[8] (summarized in Table 1). Though several other sophisticated algorithms to compute complex statistics on sensor-generated streams also exist in the literature, most of these algorithms use CM, HH, or FM as *kernels* [7, 14]. The combinations of these algorithms are sufficient to compute most of the commonly used statistics – top-k elements, unique elements, and anomalies.

2.1 CountMin (CM)

This algorithm[1] maintains an approximate count of the number of occurrences of all the stream elements. In its simplest form, the algorithm uses a hashtable with $\mathcal{W}$ buckets, wherein each bucket stores a counter value. The algorithm maps a stream element, x, to one of the $\mathcal{W}$ buckets. To ensure a uniform mapping, we typically use a hash function of the form $(a \cdot x + b) \ \% \ p \ \% \ \mathcal{W}$, where % is the modulo operation, p is a prime number, a and b are constants.

Furthermore, to reduce the impact of hash collisions, we use $\mathcal{D}$ pairwise independent hash functions $(h_0, h_1, \ldots, h_{\mathcal{D}-1})$, where each hash function has $\mathcal{W}$ buckets of its own. The corresponding data structure can be viewed as a 2D array of counters with $\mathcal{D} \times \mathcal{W}$ elements: count$[0, 0], \ldots,$ count$[\mathcal{D}-1, \mathcal{W}-1]$ (shown in Fig. 1(a) - the blue shaded box). When an element x arrives at an EHD, it initiates an update request. The counters corresponding to all the $\mathcal{D}$ hash functions are incremented; while for a query request, the counter with the lowest value among these counters is returned as an estimate ($\hat{N}_x$) for the count of the element x. These operations can be summarized as follows:

❶ Update(x): count$[i, h_i(x)] \leftarrow$ count$[i, h_i(x)] + 1, \forall i \in [0 \ldots \mathcal{D}-1]$

❷ Query(x): $\hat{N}_x \leftarrow \min(\text{count}[i, h_i(x)]), \forall i \in [0 \ldots \mathcal{D}-1]$

2.2 Heavy-Hitters (HH)

Given a stream of size N, this algorithm[2] determines those elements whose frequency of occurrence is more than a threshold ($\mathcal{T}$), where $\mathcal{T} = N/(k+1)$, k (> 0) being a user-specified parameter. Similar to CM, it uses a $\mathcal{D} \times \mathcal{W}$ matrix of buckets; however, unlike CM, this algorithm maintains *multiple counters* within each bucket. In particular, if an element in the input stream is represented by L **bits**, then the algorithm maintains $L+1$ counters within a bucket: $(C_{L-1}, C_{L-2}, \ldots, C_0)$ – one for each **bit**, and a global counter, C, which keeps track of the total number of updates to the bucket. Thus, the data structure can be represented as a 3D array of counters with $(\mathcal{D} \times \mathcal{W} \times (L+1))$ elements: count$[0, 0, 0], \ldots,$ count$[\mathcal{D}-1, \mathcal{W}-1, L]$ (see Fig. 1(a)). While updating, when a data element x is received, the $\mathcal{D}$ hashes are computed, and for each matching bucket (one per row), the global counter and the counters corresponding to each set bit (= 1) in the element's binary value are incremented. As an example, consider L to be 8. If the input element is 121 (01111001_b), then for a particular bucket, the counters C_6, C_5, C_4, C_3, C_0 and C will be incremented. The corresponding $update(x)$ operation can be formally written as:

❶ Update(x): if $(bit(x, k) = 1 \lor k = L), \forall k \in [0, L]$, then count$[i, h_i(x), k] \leftarrow$ count$[i, h_i(x), k]+1, \forall i \in [0 \ldots \mathcal{D}-1]$

Here $bit(x, k)$ returns the value of the k^{th} bit in x, i.e., 1 or 0 (the LSB is bit 0). In our running example $bit(121, 0) = 1$ and $bit(121, 1) = 0$.

In the query operation, we need to find all the stream elements that have been seen more than $\mathcal{T}$ times. We consider each of the $(\mathcal{D} \times \mathcal{W})$ buckets, one after the other. For each bucket, let's say w_0, in the row belonging to hash function h_0, we initially check if the value of the global counter C exceeds the threshold $\mathcal{T}$. If it does, then we find all the counters from $(C_{L-1}, \ldots C_0)$, that exceed $\mathcal{T}$. The estimate ($\hat{x}$) of the value of the stream element, x, that led to these counts is $\sum_{i=0}^{L-1} 2^i, \forall C_i \geq \mathcal{T}$. For example, assume that only the counters C_0 (LSB) and C_2 exceed the threshold, then $\hat{x} = 3$ (101_b). Please note that if for a counter C_j the following relations hold, $(C_j \geq \mathcal{T})$ and $(C - C_j \geq \mathcal{T})$, we ignore the entire bucket and continue to explore the next bucket. It is because this condition indicates that there are possibly two elements that exceed the threshold, and hence the set of counters that exceed $\mathcal{T}$ do not point to a *unique element* – there is a hash collision here.

Once we have found a candidate $\hat{x}$, we need to perform further checks. For this, we first check if $\hat{x}$ maps to the bucket under consideration by computing its hash. If it matches (i.e., $h_0(3) = w_0$ in our example), then all the remaining $\mathcal{D}-1$ hashes are computed, and the global counters of the corresponding buckets are obtained. If all the counters are greater than the threshold, then this element is returned as one of the results of the query operation.

2.3 Flajolet-Martin (FM)

This algorithm[8] counts the number of distinct elements in a data stream and is based on a hash function that uniformly distributes the incoming elements (similar to CM). The algorithm uses the property that within a set of n distinct and uniformly distributed values, $n/2$ values will have their least significant bit (LSB, i.e., the 0^{th} bit) set. Generalizing this property: $n/2^{k+1}$ values will have their k^{th} bit set (starting from the LSB), with the remaining lower bits $((k-1), \ldots, 0)$ being unset (= 0). The algorithm maintains a bit-vector, *BITMAP*, of length L (L is the number of bits required to represent the hashed value), which is updated upon the arrival of

each stream element as follows:
❶ Update(x): $\text{BITMAP}[\text{LSB}(h(x))] \leftarrow 1$
Here, $LSB(h(x))$ returns the index of the least significant *set* bit in the hashed value of the input element (x). For example, Fig. 1(b) shows a $BITMAP$ with $L = 8$, and the hashed value $(00101011)_b$ of x_1 has its 0^{th} bit set, thus $BITMAP[0]$=1. Similarly, $BITMAP[1]$=1 and $BITMAP[4]$=1, since $h(x_2)$ and $h(x_n)$ respectively have their 1^{st} and 4^{th} bits set. Upon a query, we first find the least index (κ) in $BITMAP$ whose value is 0, and then the distinct count estimate C is computed from κ as follows:
❷ Query(): $C \leftarrow 2^{(\kappa+1)}/\phi$, where $\kappa \leftarrow min\{i|\text{BITMAP}[i] = 0\}$
Here, $\phi \approx 0.7$ (an empirically computed correction factor); this has been theoretically justified by Flajolet et al.[8]. The accuracy of FM can be improved by having multiple hash functions and correspondingly multiple bit-vectors. These hash functions are divided into $\mathcal{W}$ groups, with each group having $\mathcal{D}$ hash functions. When an element arrives, the *BITMAPs* of all the $\mathcal{W} \times \mathcal{D}$ hash functions are updated. Upon a query, the *median* of the indices (κ) of individual hash functions are taken from each group, and then an *average* value is computed from these indices (overall the $\mathcal{W}$ groups); this is used as the final value of κ used to compute C.

3 RELATED WORK

Based on accuracy and efficiency, we classify the existing approaches for handling data streams into two groups:
❶ **Traditional Algorithms** Using a key-value store is the simplest approach to handle data streams. However, the space complexity for a stream with C unique elements is $O(C)$, making key-value stores an inefficient data structure for space-constrained EHDs. The complexity can be reduced to $O(1)$ by using sampling and lossy compression[3], where only a fixed number of elements are stored. However, these techniques can lead to missing/losing some important data samples, resulting in a significant accuracy loss. In contrast, *EHDSktch* achieves the best of both worlds by maintaining information about all the elements using sub-linear space (see Table 1 for the space complexity) and yet providing good accuracy.
❷ **Sketching Algorithms** As described above, sketching-based algorithms provide good accuracy while handling streams in a space-efficient manner. Most sketching algorithms have been implemented in software; however, several FPGA-based hardware implementations have also been proposed [11, 15, 16]. These implementations have different goals in comparison to our architecture. Based on the observation that the stream elements have extremely varying frequencies, Tang et al. [15] propose to use variable-bit counters. Our architecture can also use this idea; however, it would complicate our architecture and reduce its generic nature since all algorithms are not counter-based (e.g., FM). Saavedra et al. [11] present custom hardware for the HH algorithm, and similar to our implementation, exploit parallelism. However, their implementation is not generic. Tong et al. [16] propose a pipelined architecture aiming to increase the system's throughput. Unlike us, they do not emphasize on extending and augmenting the architecture with supplementary structures for memory and performance gains.

The existing accelerators aim to achieve high throughput and are primarily designed for big data systems; they are not compatible with EHDs because their designs are excessively reliant on large indexes stored in power-hungry LUTs and BRAMs. The dynamic power consumption of an LUT is around 500 times higher than that of an ASIC gate, and an FPGA-based implementation consumes, on average, around 14 times more dynamic power than an equivalent ASIC implementation [9].

4 SKETCHING ACCELERATORS FOR EHDS

Most of the major sketching algorithms have a similar structure – they use a set of hashes, perform memory accesses to fetch the data values and have some application-specific computation. They typically perform the update and query operations. Thus, it is possible to create a *generic architecture* (*template*) for them. This template can be realized as a core set of IP blocks that provide specific functionalities. This template can then be suitably modified and integrated into any EHD's SoC (System on Chip) as an accelerator.

4.1 Generic Architecture

To accommodate a diverse set of sketching algorithms, the proposed architecture (see Fig. 2) has three stages: *Hash, Fetch,* and *Evaluation.* Please ignore the IP block '*Request Filter*' for now.

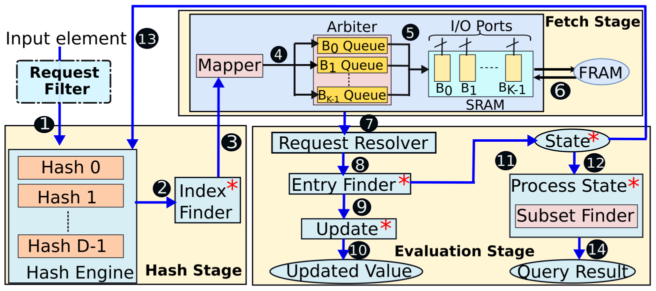

Figure 2: Generic Architecture

4.1.1 *Hash Stage.*

When a sensor input is read from the environment, it is sent to the hash stage, which has a *Hash Engine* that contains a set of $\mathcal{D}$ blocks for computing different hash functions. Each hash function is of the form, $(a.x + b)\%p$, where x is the input, a and b are randomly generated parameters, and p is a prime number. A typical software implementation on an EHD with an in-order pipeline will compute the hashes iteratively. We, however, designed a reconfigurable architecture where the hash computations can be parallelized to achieve a speedup depending upon the available ambient power. The output hash values from the hash engine are then fed into the *index finder* to determine the locations in the sketch that need to be updated/queried. The index finder component acts as a stub, which can be populated with the hardware implementing an application-specific logic. For example, in CM, this component computes a simple modulo with respect to the *width*($\mathcal{W}$) of the sketch, while in the FM algorithm, this unit scans the *BIMTAP* vector and determines the position of the least significant set bit. This can either be done sequentially or can be accelerated using a tree-based structure.

4.1.2 *Fetch Stage.*

Once the set of locations has been determined, we need to access the corresponding data from the on-chip SRAM. However, due to the extremely small size of the SRAM (2 KB), the required data might not be available in the SRAM, and hence should be first fetched from the external FRAM. We observed that the SRAM and FRAM accesses become the bottleneck in the sketching algorithm's performance. So, instead of using a single port SRAM, we use a multi-port SRAM: 1 port per bank. Using multi-port memory increases the per access energy and latency, but we have the advantage of fetching multiple values in parallel, and thus the overall overhead is greatly reduced. The sketch is stored in memory in column-major order, and a column is striped across the banks. Thus, the buckets accessed by multiple hash functions are fetched in parallel. The mapping of the bucket index (from the hash stage) to its bank is done by a *Mapper*, which given a memory address, determines the corresponding bank id. Since multiple requests may map to the same bank, we maintain per-bank request queues (indicated by B_0 Queue, B_1 Queue,. . . in Fig. 2), and an *Arbiter* is used to decide the issuing order. The *Arbiter* can merge requests to the same location.

4.1.3 Evaluation Stage. Once the data is available in the SRAM (upon a hit or from FRAM after a miss), it is brought into a buffer (a small memory or a set of registers), and the data is processed depending upon the request type, i.e., an update or a query, which is determined using a *request resolver*.

Update request: Since the fetched data would contain an entire column of sketch entries, the entry required to be updated (as per the request) is determined using the *entry finder*, and then the application-specific *update* function (implemented in HW) is applied on it to obtain the *updated value*.

Query request: This follows the same path, as the *update request*, till the *entry finder* stage. After this, the value of the entry is read and stored in a *state* buffer since other values required for completely handling the query might not have arrived. E.g., in CM, we compute the minimum of all the values read from the sketch corresponding to different hash functions. Hence, the partial comparison results are stored in the *state* buffer, and subsequently, when all the required data is available, the final output is computed by the *process state* block (via the *subset finder*), and the *query result* is returned.

4.1.4 Subset Finder. This generic IP block can find a subset of values among a set of values stored in an array of registers (or memory), which satisfy a certain property. This is implemented as a binary tree of logic blocks in hardware. The lowest level of the tree consists of all the elements in the array (original value or pre-processed). At each level, an internal node (also referred to as a *choice box*) reads the values computed by its children, computes a function on them, and sends the result to its parent.
CM: In this algorithm, we need to find the minimum value of the counter. We first read the set of counters and store them in a set of registers. Then the values in these registers are sent to the Subset Finder. Each choice box receives values from its children, computes the minimum, and passes on the results to its parent – the root of the tree stores the minimum.
FM: Here, we need to find the least index in an array of bits that contains a 0. The same logic is used for computing this index. Each bit is a leaf in the tree. The bits (and their locations/indexes) are propagated towards the root. If both the inputs to a choice box are 0 then the one with the lower index is chosen, and if one of them is a 0, then that input is chosen. The root holds the location of the lowest index that contains a 0.
HH: In this, we need to iterate through L counters (within a bucket) and collect all those locations whose counters have a value greater than a pre-specified threshold. We can aggregate this information using the Subset Finder that uses an array of comparators.

4.2 Instantiating the Architecture

The described architecture can be used to realize almost all the sketching algorithms. However, the algorithms might skip some stages. E.g., in FM, the sketch's size (*BITMAP*) is extremely small (around 2-16 bytes), so it can be stored in a dedicated register, and the data fetch stage can be avoided. Further, different components in various stages can have different sketch-specific implementations. For example, in FM, the index finder in the hash stage determines the index of the least significant set bit, while in CM, it determines the bucket containing the data item. Similarly, there can be other components that can be instantiated with a sketch-specific implementation. Hence, we provided a generic architecture with different hardware components, which can be reused or extended by various algorithms. These components are marked with a * in Fig. 2. We designed these hardware components in VHDL and used the Cadence Genus tool to estimate the area, power, and time overheads (see Table 2). The designs were fabricated with 28nm technology.
Shared Area: CM and HH share around 7,140μm^2 of hardware, i.e., 75.8% of CM's area and 40.8% of HH's area. All the three algorithms share 3015μm^2 of the total hardware. This corresponds to 32%, 17.2%, and 41.7% of the areas of CM, HH, and FM, respectively.

Table 2: Overheads of components in the Generic Architecture

Block	Area (μm^2)			Power (mW)			Time (ns)		
	CM	HH	FM	CM	HH	FM	CM	HH	FM
Hash Engine	3015	3015	6031	1.8	1.8	3.6	7.8	7.8	7.8
Index Finder	1680	8697	17	0.5	2.5	0.002	5.6	34	0.9
Mapper	176	176	N.A*	0.03	0.03	N.A*	0.8	0.8	N.A*
Arbiter	1167	3022	N.A*	0.11	0.22	N.A*	15.35	16.42	N.A*
Req. Resolver	0.7	0.7	0.7	0.0001	0.0001	0.0001	0.04	0.04	0.04
Entry Finder	82	156	N.A*	0.012	0.024	N.A*	0.22	7.2	N.A*
Update	1019	1019	533	0.12	0.12	0.05	2.3	2.3	4.6
Process State	373	574	1033	0.07	0.09	0.3	5	23	16.8
N.A*-Not Applicable for FM, as there is no memory access									
SRAM Area	CM: 199,756			HH: 199,756			N.A*		
Total Area	CM: 9,412			HH: 17,486			FM:7,221		

4.3 Using a Request Filter

Now, we describe how we can augment our architecture with a supplementary structure for reducing space and gaining performance. We insert a request filter before the hash stage (see Fig. 2). All the incoming requests are performed on this structure with the aim of delaying the updates to the sketch. E.g., in CM, an IP block implementing a constant-sized hash-table can be used as the request filter. It stores the data elements and their counter values, thus avoiding the expensive accesses to the sketch in SRAM and FRAM. The temporal locality in the data stream increases the benefits manifold.

The read requests, however, become slightly expensive. This is because the request filter contains the partial counter value (corresponding to the input stream element) that should be added to the value stored in the sketch; else, the read request will result in a highly inaccurate value. However, since streaming applications have a considerably higher frequency of write (update) requests as compared to read requests, the overall overhead while handling read requests is quite small. But for certain sketching algorithms, such as HH, reads are expensive as they require all the entries in the request filter to be updated in the sketch to avoid an inaccurate output; this negatively impacts the performance. To reduce this overhead, we propose two optimizations: (i) update the sketch in the backend and (ii) update only the most important data. In the former approach, decisions regarding when to update are taken considering the variability of the ambient energy. Particularly, the backend updates can be performed when the amount of harvested energy is high. In the latter approach, the 'most important data' is updated (it is application specific). E.g., in HH, the data with higher counter values is more important.

4.3.1 Conceptual Description of the Request Filter. We use a constant-sized key-value store (N entries) where the key is the input stream element, and the value has several fields. Since there are a fixed number of entries in the key-value store, an older entry might need to be evicted upon a new stream element's arrival. A priority-based eviction is followed, where the priority is computed using a *function* of some of the value fields.

CM: The value is composed of two fields: a counter and a timestamp, which are updated whenever the corresponding entry is accessed in the key-value store. The eviction priority is the timestamp.

HH: The value has three fields: a counter, an *insertion_timestamp*, and an *update_timestamp*. The eviction priority is equal to *(update_timestamp - insertion_timestamp)*. The entry with the maximum difference has the lowest priority, and is evicted. The intuition behind this is as follows: The frequently updated element in the request filter would have a stale counter value in the sketch. Thus, we should refresh the counter value in the sketch with the corresponding value in the request filter. So, we evict the frequently updated entry in the request filter. Such an entry would have a higher value of the *update_timestamp*. However, a newly created entry would also have a high update_timestamp, but we should not evict it because the divergence of its counter value will be lower.

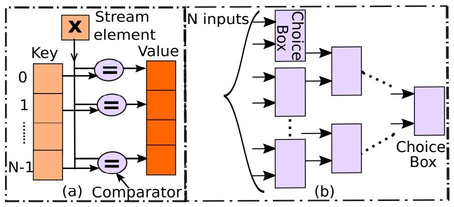

Figure 3: Implementation of the request filter (a) CAM structure for searching, (b) Select logic for selecting an element with a given priority

4.3.2 Implementation of the Request Filter. We primarily need to implement two functionalities: (i) find the key-value pair (entry) whose key matches the input stream element, and (ii) find the entry with minimum priority. The first functionality is implemented with a content addressable memory (CAM) structure (Fig. 3a), where all the keys are compared in parallel with the input stream element. To compute the entry with the minimum priority, we use a select logic with a tree-based structure (similar to the one used in the *Subset Finder*). Given that we have N entries, we need $log_2 N$ levels of choice boxes, with each choice box having two inputs (see Fig. 3(b)). The input is a function of the value field, which can vary across different algorithms. For example, CM uses a timestamp, while HH uses the difference between timestamps. Among the two inputs, the choice box chooses either the left input or the right input according to algorithm dependent custom logic. E.g., in the CM algorithm, the input with the smaller value is chosen, while in HH, a larger-valued input is chosen. The root of the tree computes the minimum priority value, and the corresponding entry is evicted. The time complexity of this select logic is $O(log_2 N)$.

Area Overheads: The area overheads for a 128 and 256-entry request filter for CM are 3,805μm^2 and 7,627μm^2, respectively. Similarly, for HH, the total area of a 128 and 256-entry request filter is 18,966μm^2 and 33,332μm^2, respectively. This corresponds to only $1.8\% - 3.6\%$ of CM's total area, and around $8.7\% - 15\%$ of HH's total area (including 2KB of SRAM).

5 EXPERIMENTS

5.1 Setup

Software System Configuration: We model a 16-bit, 5-stage in-order processor on a cycle-accurate architectural simulator, Tejas [6], which has been rigorously validated against native hardware. The modeled processor is based on 28nm technology, operates at a constant frequency of 16MHz, and is equipped with a 2 KB SRAM and 64 KB FRAM. It has a 32-byte set-associative I-cache consisting of 4 cache lines, 64 bits each. The power and timing values of the processors' components have been computed using McPat. The energy and latency values for the SRAM and FRAM are computed using Cacti 6.0, and are in accordance with the values provided in [12]. We use architectural simulation because our solution proposes changes to the organization of the memory system of conventional hardware. Specifically, we model a 4-port SRAM with 4 banks.

Stream and Sketch Configuration: We consider data streams that follow a Zipfian distribution with a skew of 0.8 (similar to [4]). The data element is represented by 16 bits, i.e., for HH, L=16, and for FM, BITMAP's size is 2 bytes, and (ii) the counter values are of 32 bits. For CM and HH, we used a depth ($\mathcal{D}$) of 8. A width (W) of 1024 is chosen for CM, while HH has a width of 64. Similarly, a set of 8 hash functions ($\mathcal{D}$) has been used for FM. All the input signals in our hardware are 16 bits wide. The memory footprint of *EHDSktch* is less than the amount of memory available in most of the devices used in energy harvesting systems (e.g., TI's MSP430FR5969).

5.2 Performance Evaluation

Fig. 4 compares the energy consumed and the time taken by the sketching algorithms when executed in software to that when they are implemented in hardware. For the software, we considered the efficient state-of-the-art implementations from the C++ standard library. Fig. 4(a,b) shows the energy consumed (EPU: energy per update and EPQ: energy per query) and the time taken (CPU: cycles per update and CPQ: cycles per query) by S/W implementations normalized to the hardware implementations. The hardware implementation of CM consumes around $8-11\times$ (Fig. 4(a)) less energy and takes around $28-34\times$ (Fig. 4(b)) less time than the corresponding software implementation. Similarly, the hardware implementation of HH is around $4-6\times$ more energy-efficient and $10-18\times$ faster than the corresponding software implementation. The updates in FM are around $>10K\times$ more energy efficient because the entire sketch can be stored in a small hardware register, and hence no SRAM or FRAM accesses are involved. It is also $2K-13K\times$ faster than the equivalent software implementation because it uses a fast and parallel tree-structured implementation to compute the index of the least significant set bit, while the software implementation iteratively scans the entire L-bit $BITMAP$.

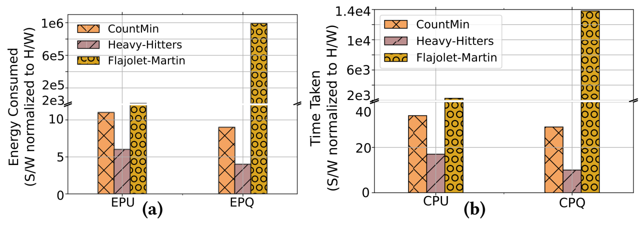

Figure 4: Comparison of S/W sketching to H/W sketching (a) Energy consumed (b) Time taken

5.3 Impact of the Request Filter

Table 3 shows the performance benefits of using the request filter along with the sketch (stored in 512B or 1KB of SRAM). The table shows the time and energy values for CM and HH, normalized to the results obtained by using the straightforward hardware that has a 2 KB SRAM and no request filter.

Space benefit: Using a request filter with only 16 entries, the CM gets a 3-6% performance gain and can reduce 50% of SRAM usage. The SRAM's size can be reduced by 75%, and the performance can be increased by 5-8% with 16 additional entries. A similar pattern is followed by HH. The performance benefit increases with an increase in the number of entries. However, there is an accuracy-performance trade-off. The column marked with a † shows the accuracy obtained in HH by using a request filter normalized to the case when no request filter is used. The results show that as the number of entries increases, the achieved accuracy decreases. This is because the number of evictions is reduced, and the counter values of the frequent stream elements do not get updated in the sketch.

Impact of the eviction policy on accuracy: We show the significance of our eviction policy in HH by comparing it to the least recently used (minimum *update_timestamp*) based eviction. The respective accuracies are shown in the columns marked with † and ‡ in Table 3. The results show that our proposed eviction policy is $4-7\%$ more accurate.

Efficient execution of EHDs: There are two benefits: ❶ Due to space reduction, the checkpoint's size is significantly reduced. Since the checkpointing energy is directly proportional to the checkpoint's size [12], a small checkpoint will have lower checkpoint-restore costs. Though an additional cost will be there due to the entries of the request filter, that overhead is very small. Furthermore, we do not need to checkpoint the timestamps. ❷ Depending upon the available energy, the device can choose a memory configuration considering the accuracy-performance trade-off. If the available energy is high, the device can spend more energy and, thus, execute in a high-accuracy mode, i.e., the device can use a request filter with a small number of entries. In contrast, at times of low available energy, it can use a configuration with a large number of entries: low power and reduced accuracy.

Table 3: Performance evaluation of the request filter

	Normalized Time*				Normalized Energy*				% Relative Accuracy	
	512 Bytes		1024 Bytes		512 Bytes		1024 Bytes			
# Entries	CM	HH	CM	HH	CM	HH	CM	HH	HH†	HH‡
0	0.89	0.95	0.93	0.97	0.89	0.95	0.93	0.97	100	100
8	0.98	0.96	0.99	0.98	0.99	0.96	1.02	0.98	99.9	99.9
16	1.01	0.98	**1.03**	0.99	1.03	0.98	**1.06**	0.99	99.8	99.8
32	**1.05**	1.01	1.07	1.02	**1.08**	1.00	1.10	1.02	99.6	99.6
64	1.09	1.06	1.11	1.07	1.12	1.06	1.14	1.07	99.3	98.4
128	1.18	1.17	1.20	1.88	1.21	1.17	1.23	1.19	98.3	95.7
256	1.30	1.28	1.32	1.29	1.34	1.28	1.36	1.29	98.2	91.1

* Normalized to the results obtained when no request filter was used.
Eviction policy: †*(update_timestamp - insertion_timestamp)* based, ‡LRU based.

6 CONCLUSION

We propose a generic architecture for sketching that can be easily augmented with supplementary structures. These structures lead to a performance gain of 3-20% and reduce the on-chip memory requirement by 50-75%. These savings in performance and space are significantly beneficial for resource-constrained EHDs.

7 ACKNOWLEDGEMENTS

This work has been sponsored in part by the Semiconductor Research Corporation (SRC).

REFERENCES

[1] Graham Cormode and Shan Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms* 55, 1 (2005), 58–75.
[2] Graham Cormode and Shan Muthukrishnan. 2005. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. *ACM TODS* 30, 1 (2005), 249–278.
[3] de Aquino et al. 2007. A Sampling Data Stream Algorithm For Wireless Sensor Networks. In *ICC*. IEEE, 3207–3212.
[4] Fan Deng and Davood Rafiei. 2007. New Estimation Algorithms for Streaming Data: Count-min Can Do More. *Webdocs. Cs. Ualberta. Ca* (2007).
[5] Marjani et al. 2017. Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access* 5 (2017), 5247–5261.
[6] Sarangi et al. 2015. Tejas: A Java based Versatile Micro-architectural Simulator. In *PATMOS*.
[7] Yang et al. 2018. HeavyGuardian: Separate and Guard Hot Items in Data Streams. In *SIGKDD*. 2584–2593.
[8] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic Counting Algorithms for Data Base Applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.
[9] Ian Kuon and Jonathan Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE TCAD* 26, 2 (2007), 203–215.
[10] Ian Kuon, Russell Tessier, and Jonathan Rose. 2008. *FPGA Architecture: Survey and Challenges*. Now Publishers Inc.
[11] Antonio Saavedra, Cecilia Hernández, and Miguel Figueroa. 2018. Heavy-hitter detection using a hardware sketch with the Countmin-CU algorithm. In *IEEE DSD*. 38–45.
[12] Priyanka Singla, Shubhankar Singh, and Smruti R Sarangi. 2019. FlexiCheck: An Adaptive Checkpointing Architecture for Energy Harvesting Devices. In *DATE*.
[13] Knud Erik Skouby and Per Lynggaard. 2014. Smart Home and Smart City Solutions enabled by 5G, IoT, AAI and CoT Services. In *IC3I*.
[14] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. 2018. Sketching Linear Classifiers over Data Streams. In *SIGMOD*. 757–772.
[15] Minjin Tang, Mei Wen, Junzhong Shen, Xiaolei Zhao, and Chunyaun Zhang. 2020. Towards Memory-Efficient Streaming Processing with Counter-Cascading Sketching on FPGA. In *DAC*.
[16] Da Tong and Viktor K Prasanna. 2017. Sketch Acceleration on FPGA and its Applications in Network Anomaly Detection. *TPDS* 29, 4 (2017), 929–942.