CmpctArch: A Generic Low Power Architecture for Compact Data Structures in Energy Harvesting Devices

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

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

***Abstract*—Small sub-mW sensor devices that rely on energy harvesting often have limited computation capability to locally process, query, and update data. Such energy harvesting devices (EHDs) need to operate under a strict power constraint and are also extremely cost-sensitive. Based on current costs and estimated near-term trends, we observe that the price of the nonvolatile memory component dominates (in $\approx$ 20 USD devices). Hence, there is a pressing need to reduce the overall memory footprint in a data-intensive setting.**

This paper is the first to propose a generic hardware architecture, *CmpctArch*, for implementing compact data structures (CDSs) on such devices. They reduce the memory footprint by up to 3.5$\times$ without significantly increasing the overall energy consumption or time taken (max. additional energy 1.04$\times$ and time 1.18$\times$). The hardware implementations are 160-1200$\times$ more energy-efficient and 280-620$\times$ faster than the corresponding software implementations of CDSs. Our generic template can be used to instantiate a wide variety of data structures commonly used in EHD applications.

***Index Terms*—Compact data structures, Hardware accelerators, Memory footprint, Resource-constrained, System on chip**

I. Introduction

Energy harvesting devices (EHDs) that rely on ambient power sources such as vibrations, solar, and RF radiations [1; 2] are increasingly gaining prominence. As of today, their market size is estimated to be 468 million dollars, with a growth rate of 8.4% per year [3]. These devices are being used in a wide variety of applications ranging from smart homes to environmental and structural health monitoring in hazardous and remote locations. They sense large amounts of environmental parameters and perform local computations (e.g., determine the frequency of different values or find anomalies). The data is then stored locally or transmitted to remote base stations [4]. Given that communication requires a lot of energy, it makes sense to do some local computation.

Different data structures such as hash tables [4], tries [5], and lists [6] are used to locally store data. An issue with these data structures is that they use a lot of memory, a critical resource for EHDs. A typical commercial EHD compute node is equipped with only 2-8KB of SRAM and 64 KB of FRAM (e.g., TI MSP430FR5969). Such a device costs around 20 USD [7]. Increasing the amount of memory in these devices is not a viable solution due to the high prices of nonvolatile memories (NVMs) (see Fig. 1(a)); they account for more than 50% of the overall cost. Moreover, the large-scale deployment of these devices aggravates the problem even further because of the net increase in the overall cost. Thus, there is a pressing need to devise data structures that efficiently use the available memory in these devices. Additionally, with efficient memory utilization, we can also have devices with reduced memory sizes, thus, making the devices cheaper and reducing the total cost. Even if NVMs were to reduce in price by 50%, which is unlikely to happen immediately, this problem will still remain because NVMs already account for a large fraction of the overall cost ($\approx$ 50%).

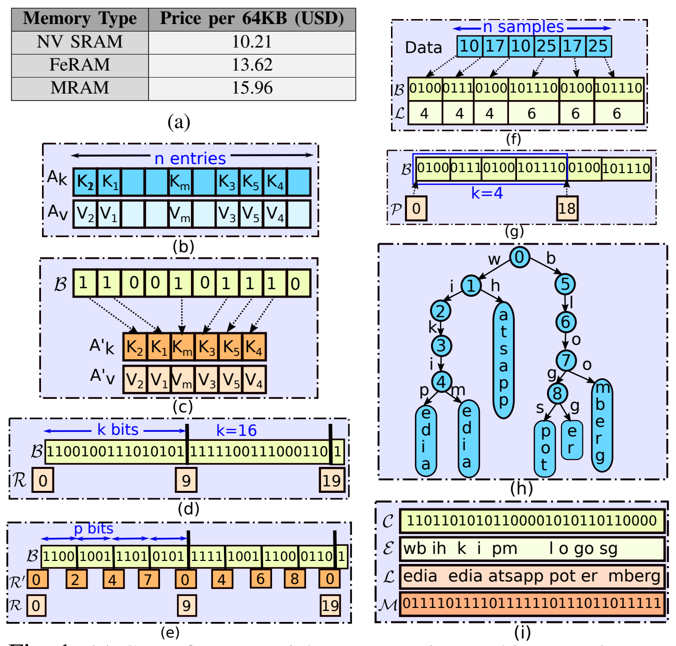
Please note that we considered this class of devices, whose total cost is between 15-25 USD (at current prices), because they are very popular choices in the area of EHDs, and have been used in numerous prior works [2; 4]. A key constraint with these devices is that we have a power budget of 500-750 μW. Additionally, we can assume that programs run correctly in such processors where the ambient power is variable, and crashes can happen (please refer to Singla et al. [1] for a survey of checkpointing techniques for EHD devices). We thus consider this a solved problem.

Our *aim* is to minimize the memory footprint while staying within the power constraints and not significantly increasing the processing time. ❶ We propose to use compact implementations [8] of various data structures that primarily rely on bitwise operations. ❷ Second, our proposed implementations allow accessing and updating the data in the compact form itself, without any decompression. ❸ Our third contribution is to propose a generic template for designing such compact data structures on ASICs. This generic template idea has been motivated by existing literature that targets other data structures [4; 9]. A generic template provides us with the flexibility of instantiating different kinds of data structures in a broad family. ❹ Fourth, with an instantiation of our generic template, we show the feasibility of reducing the on-chip memory requirements by up to 3.5$\times$, indicating that we can have devices with much smaller memory size, thus reducing the price of the devices while obeying all power constraints.

The paper is organized as follows: we provide the necessary background in §II followed by our proposed generic architecture in §III. We evaluate our implementation in Section §IV, discuss related work in §V, and finally, conclude in §VI.

II. Background

We chose three of the most popular data structures used in EHD applications [4; 5; 6]: hash tables (HT), lists, and tries. We consider the compact representations of these structures. In terms of implementation complexity, power, and perfor-



Memory Type	Price per 64KB (USD)
NV SRAM	10.21
FeRAM	13.62
MRAM	15.96

Fig. 1: **(a)** Cost of commercial NV memories used in EHDs/IoT [7], **(b)** Regular HT, **(c)** Compact HT, **(d)** Efficient rank computation, **(e)** Two-level rank computation, **(f)** Compact list, **(g)** Accessing the list, **(h)** Regular trie, **(i)** Compact trie

mance overheads, they represent three different extrema of the spectrum of compact data structures (CDSs) [8]. These compact representations support reads and limited updates, i.e., the updates that do not change the underlying structure. For example, in a hash table with a fixed key set, only the values of the keys can be updated. This notion of read mostly and limited updates is well accepted in the CDS community [8], as changing the underlying structure would result in a lot of complexities and overheads. Furthermore, most EHD applications fit this model as their data distributions are generally fixed. For example, a temperature sensor deployed in a smart home will have a bounded set of temperature values with a median of roughly $25°C$ (room temperature). In the following descriptions the calligraphic font (e.g. $\mathcal{B}$) refers to structures that will be implemented in hardware.

A. Hash Tables (HT)

Fig. 1(b) shows the implementation of a hash table using two arrays – A_k and A_v – to store the keys and values, respectively. A typical hash table implementation reserves extra space to handle collisions. For example, in Fig. 1(b), a table with m keys is realized by having n entries ($n > m$) for each array. This wastes $2 * (n - m)$ entries. This wastage can be eliminated using a compact representation, as shown in Fig. 1(c). The occupied key-value entries in the original arrays (A_k and A_v) are stored in two new arrays, A'_k and A'_v (of size m), in the same order as they appear in A_k and A_v, respectively. A bit-vector $\mathcal{B}[0 : n-1]$ captures the occupied and free indices in A_k (or A_v). $\mathcal{B}$ can be formally defined as:

$$\mathcal{B}[i] = \begin{cases} 1, & \text{if } A_k[i] \neq NULL \qquad \forall i \in [0, n-1] \\ 0, & \text{otherwise} \end{cases}$$

A function $rank_1(\mathcal{B}, i)$ that returns the number of 1 bits in $\mathcal{B}[0 : i]$ is formally defined as:

$$rank_1(\mathcal{B}, i) = \sum_{j=0}^{i} \mathcal{B}[j] \quad \forall i \in [0, n-1]$$

An entry at a random index, i, in A_k (or A_v) can now be accessed as follows:

$$A_k[i] = \begin{cases} NULL, & \text{if } \mathcal{B}[i] = 0 \\ A'_k[rank_1(\mathcal{B}, i) - 1], & \text{otherwise} \end{cases}$$

This representation considerably reduces the space used; however, it comes at the cost of the sequential scanning of $\mathcal{B}$ for computing the rank. Sequential scanning can be avoided by storing the rank for each index in a separate array $\mathcal{R}$. But, it would add a lot of space overhead. A better way is to store the rank after every k bits ($k = 16$ in Fig. 1(d)). Let us call this set of k bits a *superblock*. Thus, the rank is given by,

$$rank_1(\mathcal{B}, i) = \mathcal{R}[\lfloor i/k \rfloor] + cntOnes(\mathcal{B}, i, k)$$

Here, $\mathcal{R}[0] = 0$ and $\mathcal{R}[l]$ denotes the total number of 1s in the superblocks $[0 \dots (l-1)]$. *cntOnes* counts the number of 1s in the current superblock of size k (i.e., from $\mathcal{B}[\lfloor i/k \rfloor \times k]$ to $\mathcal{B}[i]$).

We can further extend the design to a 2-level structure (Fig. 1(e)), where we divide a superblock into equal-sized sub-blocks of p bits ($p = 4$ in our example in Fig. 1(e)), and have an additional array $\mathcal{R}'$ that stores the rank values for these smaller blocks. These values are relative to the superblock containing them. For example, for $i = 21, k = 16, p = 4$, $\mathcal{R}'[5]$ contains the number of 1s in only the 4^{th} sub-block (sub-blocks start from 0). The counts in sub-blocks $0 \dots 3$ are accounted for in $\mathcal{R}$. Using $\mathcal{R}'$ reduces the number of bits that need to be stored. $cntOnes(\mathcal{B}, i, k)$ can be defined as:

$$cntOnes(\mathcal{B}, i, k) = \mathcal{R}'[\lfloor i/p \rfloor] + cntOnes(\mathcal{B}, i, p)$$

Here, $cntOnes(\mathcal{B}, i, p)$ is computed by linearly scanning p. Using our formalisms, the *update* and *query* operations are as follows.

❶ Update(k,v): $A'_v[rank_1(\mathcal{B}, h(k)\%n) - 1] \leftarrow v$
❷ Query(k): $v \leftarrow A'_v[rank_1(\mathcal{B}, h(k)\%n) - 1]$

Here, $h()$ is a hash function. We use three hash functions (akin to double hashing), where if there is a collision with one hash function we try the next one.

Observation: Note that accessing the compact HTs comprises three key steps. First, an *index* is computed using a hash function, which is then used to access the bit-vector $\mathcal{B}$. Then, using $\mathcal{B}$, a *rank* is computed, which is finally used to *access an array* to retrieve the data. In case the retrieved data does not *match* the required data (due to a hash collision), these three steps are repeated using a different hash function. These steps are summarized as the following regular expression:

$$((Index)(Rank)(Array\ \ Access)(Match))^{+}$$

B. Lists

This is a compressed list of values that follow a certain distribution. Instead of assigning the same number of bits to every value, we can encode them more efficiently using Huffman encoding. We divide a hexadecimal output of a sensor into its corresponding digits. Since the distribution is fixed, we learn it and generate Huffman codes for the *digits*. Huffman codes form a prefix tree, where the codeword is the path from the root to a leaf node. We store the *digit* $\rightarrow$ *codeword* mapping as a dictionary in memory. Fig. 1(f) shows a list, $Data$ (with n sensor samples), which is stored in compressed form as a bit-vector $\mathcal{B}$, and the code lengths of the samples are stored in an array $\mathcal{L}[0:n-1]$. The code lengths are stored in a compact form by concatenating the binary representations of all the length values. Though different samples would have different code lengths, we made them of the same length (maximum length, depending upon the application) by prefixing 0 bits to the smaller code lengths.

To access the value at an index, i i.e., $Data[i]$ ($\forall i \in [0, n-1]$), we first need to get the corresponding code (denoted by $code_i$) from $\mathcal{B}$, which can be retrieved as follows (the notation $x:y$ refers to the subarray from index x to y (both included)).

$$code_i = \mathcal{B}[j : j + \mathcal{L}[i] - 1], \text{where } j = \textstyle\sum_{k=0}^{i-1} \mathcal{L}[k]$$

Here, the computation of j requires the sequential scanning of $\mathcal{L}$ to compute the sum of the code lengths. This scanning can be avoided by logically dividing $\mathcal{B}$ into blocks of k values (k=4 in Fig. 1(g)) and storing the sum for each block in array $\mathcal{P}$ (akin to the previous algorithm). j can now be computed as follows.

$$j = \mathcal{P}[\lfloor i/k \rfloor] + \textstyle\sum_{w=\lfloor i/k \rfloor \times k}^{i-1} \mathcal{L}[w]$$

Here, $\mathcal{P}[0] = 0$, and $\mathcal{P}[m]$ indicates the sum of code lengths for blocks $[0 \ldots (m-1)]$. Once we have $code_i$, we can access the i^{th} sample in $Data$ as follows.

❶ $Query(i)$: $v \leftarrow get\_code(code_i)$

get_code needs a traversal of the Huffman tree, which is a slow operation. To optimize this, we maintain a reverse dictionary (*codeword* $\rightarrow$ *digit*) in a CAM structure. This still requires a sequential search. Instead of searching one bit at a time, we set the minimum code size to α bits and maximum to β bits. Now, we consider the first α bits of $code_i$, then $\alpha+1$ if there is no match, and so on (until β). After the digits are retrieved we concatenate them to generate the value. Using α and β reduces the searching overheads.

Observation: Similar to the HT, lists follow a sequence of steps. Given an index i (from the application) a *rank* like functionality (comprising summation of values at various array indices) is performed to compute j. Then, an *array is traversed* to retrieve $code_i$. Finally, either a *tree or a CAM traversal* is used to retrieve the *matching* data sample. The corresponding regular expression is:

$$(Rank)(Array\ Traversal)((Tree|CAM\ Traversal)(Match))^{+}$$

C. Tries

1) Basic structures: A trie is a prefix tree data structure that stores a set of strings. The strings are stored as *edge labels*: an edge label is a character from the string, labeling the edge that connects a node to a particular child. A string is represented by the edge labels starting from the root to a leaf. Fig. 1(h) shows a trie data structure where a chain of nodes without branches have been coalesced into a single node (only done at the leaves). For example, in the string 'wikipedia', the characters 'w','i','k','i','p' form the edge labels, while the remaining characters are merged into a leaf. The compact representation of this trie is shown in Fig. 1(i). It comprises four arrays: a bit-vector $\mathcal{C}$ (for children), two character arrays $\mathcal{E}$ (edges) and $\mathcal{L}$ (leaves), and another bit-vector $\mathcal{M}$ (used to iterate through $\mathcal{L}$).

$\mathcal{C}$ encodes the structure of the tree. We generate it by performing a preorder traversal of the tree. At every node, we count its children (cnt), and append cnt 1s followed by a 0 to the bit-vector $\mathcal{C}$. For example, in Fig. 1(i), the root node ⓪ has two children, ① and ⑤, so we append 110 to $\mathcal{C}$. The edge labels corresponding to the children are stored in array $\mathcal{E}$. The leaf labels (strings in the leaves) are concatenated, left to right, and stored in array $\mathcal{L}$. The bit-vector $\mathcal{M}$ stores their corresponding sizes (of the leaf labels). For a leaf label with 'l' characters, we add a 0 followed by l 1s. Note that the order of entries in $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{M}$ is exactly the same as that enumerated by the preorder traversal (refer to Fig. 1(i)).

2) Index of the j^{th} child: Now, the problem is, given a node's list of children (let us call it, the node's *description*) at index u in $\mathcal{C}$, can we find the index to the description of its j^{th} child (i.e., $child(u,j)$)? Let c be the number of children of the node whose description is stored at u. We need to compute $child(u,j)$. To start with, note that $child(u,1) = u + c + 1$.

To compute $child(u,2)$ we need to find the sum of the sizes of the descriptions of the sub-tree rooted at $child(u,1)$. The basic insight is that for any sub-tree, the number of 0s exceeds the number of 1s by 1 (in the description of the sub-tree); this is because for every node in the sub-tree, there is a 1 for it in its parent's description and a 0 in its description. For the root, we are only counting a 0, not a 1. To find $child(u,2)$ we have to start from $child(u,1)$, and find the size of the prefix of $\mathcal{C}$ for which this property holds.

Let us now generalize this. We start traversing $\mathcal{C}$ from $child(u,1)$ and maintain two counters: cnt_0 (number of 0s) and cnt_1 (number of 1s). The moment we find $cnt_0 = cnt_1 + 1$, we note that we have reached $child(u,2)$. We reset the counters and continue in this manner until we reach $child(u,j)$.

3) Matching an edge label: Let us now discuss the $Query(P)$ operation for checking the existence of a pattern P in the trie. Beginning from the root, we check if P's first character ($P[0]$) is an outgoing edge label of the root and continue so on and so forth traversing the tree.

To check if a character p (from the pattern P) is an outgoing edge label of a node (whose description begins at u), we compute the following.

```
1.  $pos \leftarrow rank_1(\mathcal{C}, u) - 1$
2.  $idx \leftarrow find\_index(\mathcal{E}, pos, p)$
3.  $j \leftarrow idx - pos + 1$      /* then visit $child(u,j)$ */
```

Note that entries in $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{M}$ are stored in the same order as that generated by the preorder traversal. We thus have a one-to-one correspondence between 1s in $\mathcal{C}$ and entries in $\mathcal{E}$. In Line 1, we set pos to the rank of u minus 1 (number of entries before u in the preorder traversal, or its position in $\mathcal{E}$ counting from 0). We then use the $find\_index$ function in Line 2 to traverse the array $\mathcal{E}$ from position pos till we find the character p, i.e., $\mathcal{E}[idx] = p$. The caveat is that we will only look at the first c entries because the node at u has only c children. We then move to the j^{th} child using the algorithm in Section II-C2.

4) The full algorithm: Let us now describe the full tree traversal algorithm. We start at the root. Then we keep on matching consecutive characters in P and traversing outgoing edges with the corresponding edge labels as described in the previous paragraph. This process can either terminate at an internal node (entire pattern matches), or we may find that there are no outgoing edges with the given edge label at an internal node. The latter case would indicate a failure (no pattern match). The interesting case is when we reach a leaf node. Consider the pattern "wikipedia" in Fig. 1(h). The remaining characters ("edia") are within the leaf node (because of the node coalescing optimization). We then need to traverse the characters in the leaf node (left to right) and search for a match.

Finding the location of the leaf node within $\mathcal{M}$ and $\mathcal{L}$ is complicated. The edge traversal will give us the index of the description of the leaf node in $\mathcal{C}$. Let this be index u, we now need to find the corresponding indexes in $\mathcal{M}$ and $\mathcal{L}$. We need to know how many leaf nodes preceded the leaf node at u in the preorder traversal. Any leaf node has a 0 in its description, and is also preceded by a 0 in $\mathcal{C}$. We thus need to find the number of 00 patterns that precede it in $\mathcal{C}$. This can be found out using a generalized $rank$ function referred to as $rank_{00}$. It will give the leaf's number (u_n) in the preorder traversal. Now, we need to find the index of the u_n^{th} 0 in $\mathcal{M}$. By traversing $\mathcal{M}$ we can find the corresponding index u_m. Now, the required location (index) in $\mathcal{L}$ is equal to $u_l = u_m - u_n$. We need to start searching $\mathcal{L}$ from this index u_l for matches and declare a success or failure accordingly.

Observation: First, starting from the root (i.e., $index = 0$), the *rank* function is computed to find pos. Then, *array* $\mathcal{E}$ *is traversed* to find the location (idx) of the character *matching* p. Finally, idx is used to compute $child(u, j)$, i.e., the bit-vector C is *traversed* till the property $cnt_0 = cnt_1 + 1$ is *matched*. These steps are repeated till we reach a leaf node. Subsequently, the *rank* (rank$_{00}$) is computed, then the arrays $\mathcal{M}$ and $\mathcal{L}$ are *traversed* one after the other to *match* the leaf labels. Its regular expression is:

$$((Index^*)(Rank)((Array\ Traversal)(Match))^+)^+$$

III. CDS Accelerators for EHDs

The data retrieval from most of the compact representations follows a similar sequence of steps, which can be summarized as the following general regular expression.

$$((Index^*)(Rank)(Array\ Traversal)((Tree|CAM|Array\ Traversal)^*(Match))^+)^+$$

We can leverage this common pattern to create a *generic architecture*. This generic template is primarily a core set of IP blocks with specific functionalities. This template can be suitably modified and integrated into any EHD's SoC (System on Chip) as an accelerator. Such templates are commonly used to implement a family of similar algorithms [4; 9].

A. Generic Architecture

Based on the general regular expression, we propose our 3-stage architecture (see Fig. 2). The stages are: *Index, Rank,* and *Evaluate*. The required functionality of a compact data structure can be realized by traversing these stages (single-pass or in a loop iteratively).

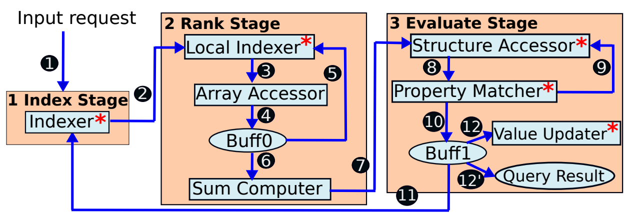


Fig. 2: **Generic architecture**

1) Index Stage: This stage has two inputs: (i) the sensor value(s) or the application-specific input and (ii) an internal state. The output is an index that will be used to access the stored arrays. Let us explain with examples. In HT, it computes $h(k)\%n$: it essentially implements the hashing function and the logic for computing a logical AND (to compute $\%n$, when n is a power of 2). For lists, it is an empty IP block that outputs the input element itself. For tries, it is a $2-to-1$ mux that outputs 0 in the first iteration and the input element in the successive iterations. The *Indexer* module can be instantiated to either be empty or implement a hashing function or a mux.

2) Rank Stage: This is a generic *rank* computer that uses an array, a pattern, and an index. The rank is defined as the number of occurrences of the pattern (in the array) up till the index. The pattern can also be a wildcard '*'. In this case, the function computes the sum of the array entries.

Since computing the rank by sequentially scanning an array is expensive, precomputed values are stored in various arrays and are used for the rank computation (discussed in §II). Each of these arrays would have their own indexes, relative to the index received from the previous stage. For example, lists use $\mathcal{P}$ and $\mathcal{L}$ arrays with indexes $\lfloor i/k \rfloor$ and $\lfloor i/k \rfloor \times k$, respectively (here i is the input received from the previous stage). The indexes for different arrays are computed by the *local indexer*. Then, an *array fetcher* computes the corresponding memory addresses, and fetches the data at those addresses. The fetched data is stored in a temporary buffer (*buff0*) as the data from other arrays might not have arrived. Subsequently, when all the required data is available, the *sum computer* adds them all and sends the corresponding result to the *evaluate* stage. Please note that instead of storing all the data and then computing the summation, we could also compute the summation iteratively, thus, storing only the partial sums. This would keep the temporary buffer small.

3) Evaluate Stage: This stage reads the output from the *rank* stage and uses it to fetch data from the stored arrays. Then a *property matcher* checks if the fetched data satisfies some data structure-specific property. Now, three cases are possible: (i) the property does not match, (ii) the property matches partially, i.e., for some part of the fetched data the property holds while not for the remaining (we will describe this later with an example), and (iii) the property matches completely. For the first case, we either access the array again until we fetch data with the matching property, or we store the *mismatch* information in a temporary buffer (*buff1*). In the second case, we store the *partially matched* information in buff1 and continue to *access the array or CAM (in lists)* till we entirely match the data. In the third case, we store the *match* information. Subsequent to matching the property, the state information is either used to handle the input request (query or update) or sent back to the *index* stage to perform another pass of the entire structure. In the former case, depending upon the request type, the state information is either returned as the *query result* or a *value updater* uses it to update a stored value.

In the above discussion, depending upon the data structure, different property matcher implementations are possible.

❶ **HT:** This uses a *comparator* to check if the fetched data is the same as the required key. A mismatch indicates a hash collision, and double hash probing is performed by iterating over the entire set of stages (using a different hash function in the *index* stage).

❷ **Lists:** Here, the property matcher is the content addressable memory (CAM) structure, which checks if the data (i.e., the codeword retrieved from the bit-vector $\mathcal{B}$) matches the codes stored as keys in the reverse dictionary. This is an example of *partial matching*. The first α bits of the data are checked in the CAM. If a match is found, the corresponding digit (from the *code* $\rightarrow$ *digit* mapping) is stored in the *state* buff1, and then the next α bits are matched. In the case of a mismatch, $\alpha + 1$ bits are matched. The property is iteratively matched until we reach the end of the data. Please refer to the get_code functionality in §II-B for reference.

❸ **Tries:** Tries need to satisfy two types of properties during the entire query operation. The first property, used in $find\_index$, is to match a character with a given array entry, and is realized via a comparator. The second property is used in the $child(u, j)$ function: we traverse the array ($\mathcal{C}$) while maintaining the counts of $0s$ (cnt_0) and $1s$ (cnt_1). Upon each array access, the property checked is $cnt_0 = cnt_1 + 1$.

B. Instantiating the Architecture

The described architecture can be used to realize most of the CDSs [8] and data structures used in EHD applications. However, the data structures might skip some stages. For example, in lists, the index stage outputs the input element itself; thus, it can be skipped. Further, different components in various stages can have different data structure-specific implementations (such components are marked with a * in Fig. 2). For example, the *local indexer* in the HT and tries is realized by a left-shift and an adder component, while lists require an additional right-shift component. Different components of the generic architecture are designed in VHDL and synthesized using the Cadence Genus tool in $28nm$ technology. Table I summarizes the estimated area, power, and time overheads of these components.

Shared Area: HT and tries share around $3,925\mu m^2$ of hardware, i.e., 89% of HT's area and 82% of trie's area. All the CDSs share $3153\mu m^2$ of the total hardware. This corresponds to 71%, 37%, and 66% of the areas of HT, lists, and tries, respectively.

TABLE I: Overheads of components in the generic architecture

Block	Area (μm^2)			Power (mW)			Time (ns)		
	HT	Lists	Tries	HT	Lists	Tries	HT	Lists	Tries
Indexer	1396	N.A†	11	0.17	N.A†	0.001	33	N.A†	0.06
Local Indexer	100	670	100	0.01	0.15	0.01	3	25	3
Array Accessor	97	63	97	0.01	0.01	0.01	3.5	2.3	3.5
Sum Computer	25	25	25	0.003	0.003	0.003	1.6	1.6	1.6
Structure Accessor	63	87	1084	0.008	0.01	0.07	2.3	3.9	31.2
Property Matcher	13	207	38	0.001	0.02	0.004	0.2	0.2	1.8
Value Updater	50	N.A‡	N.A‡	0.004	N.A‡	N.A‡	1.7	N.A‡	N.A‡
N.A-Not Applicable, †-No such component, ‡-data structures do not support update requests									
Total Area (μm^2)	HT: 4395			Lists: 8389			Tries: 4767		

C. Extending the Architecture

Our proposed architecture can be easily extended to support various other functionalities. For example, the $find\_index$ function (in *tries*) requires the traversal of array $\mathcal{E}$ till we find a matching character. This could also be done by using a hashtable, where the characters in $\mathcal{E}$ are the keys, and the list of indexes of the characters are the values. Similarly, a leaf's location can be computed using a hashtable instead of traversing $\mathcal{M}$. Here, the key is the number of the leaf, and the index is the position in $\mathcal{M}$.

Let us now see how well our architecture generalizes. We consider a set of popular data structures used in sensor devices [6]. For each data structure, Table II specifies the corresponding regular expression and the primitive data structures (HT, lists, or tries) whose implementation closely resembles the implementation of the data structure under consideration. Details are omitted due to a lack of space.

Apart from the sensor application-based data structures, our architecture also supports implementing most of the other compact data structures [8].

TABLE II: Realizing various sensor application-based data structures [6] using our generic architecture

Data structure	Regular expression	Closest primitive data structures
Tree	$((Index^*)(Rank)(Array\ Traversal)(Match))^*$	Tries
Key-value store	$(Index)((Index^*)(Rank)(Array\ Traversal)$ $((Tree \mid CAM \mid Array\ Traversal^*)(Match))^+)^+$	HT+(HT/Lists/Tries)
Queue	$((Array\ Traversal)(Match))^+$	Lists
Positional Index	$(Index)((Array\ Traversal)(Match))^*$	HT + Lists

IV. Experiments

A. Setup

System Configuration: We modeled a 16-bit, 5-stage in-order processor using 28nm technology on a cycle-accurate architectural simulator, Tejas [10]. Tejas has been previously used to simulate EHDs using the same processor configuration [2; 4]. The modeled processor operates at a constant frequency of 16 MHz, and the simulated device is equipped with a 32B set-associative cache, 2 KB of SRAM, and 64 KB FRAM. Tejas includes power and cache timing models, which have been validated with native hardware. In our hardware, all the input signals are 16 bits wide.

Datasets: Similar to prior works on sensor applications [11], we used the Intel Lab Dataset [12], comprising samples of different ambient parameters (e.g., temperature and humidity) collected from various sensors. We considered only the temperature samples; the other parameters follow similar trends.

B. Results

Memory Savings: Fig. 3(a) presents the compaction ratio ($mem\_footprint_{regular}/mem\_footprint_{compact}$) of the three data structures. The ratios are between 3.44X (Tries) and 1.3X (HT). The term *regular* refers to an implementation of a traditional, non-compact data structure.

Energy Consumed and Time Taken: In this case we compute the ratio of compact to regular (reverse). Fig. 3(b) shows the relative energy consumed (EPA: energy per access) and the relative time taken (CPA: cycles per access) for software implementations. The results show that the software implementation of the compact representations of HT and tries consume around 2.34-37.17$\times$ more energy and take 2.41-34.75$\times$ more time. We observed that certain operations such as $cntOnes$ and $child$ (in tries) involve linear scanning of the bit vectors, resulting in large overheads.

The hardware results are however significantly better (shown in Fig. 3(c)) owing to parallelism and reduced memory/register usage. The results show that the compact representation of tries is much more energy-efficient (0.53$\times$) and took almost similar time. Similarly, for HT, the hardware implementation of the compact representation consumed similar energy (1.04$\times$) with a time overhead of 1.18$\times$.

In contrast to HT and tries, the compact software implementation of lists is very inefficient (see Fig. 3b), with the energy and time overheads being 320-334$\times$. The reason is the multiple memory accesses involved while performing the linear scanning in the get_code function. However, this scanning could be easily done by implementing the dictionary as a CAM structure in hardware. Fig. 3(c) shows the corresponding performance improvement, with compact lists consuming only 0.31$\times$ the energy and taking a similar amount of time.

We observed that, in general, the hardware implementations of CDSs are 160-1200$\times$ more energy-efficient and 280-620$\times$ faster than the corresponding software implementations.

Preferable Ambient Source: We observed that our hardware implementations of the considered data structures (HT, lists, and tries) consume power in the range of 180µW-700µW, which is much lower than that provided by various ambient sources. Commercially available solar harvesters (e.g., MAX20361) and piezo transducers (e.g., S118-J1SS-1808YB) can harvest the required power [7], making vibrational and solar sources preferable ambient sources.

V. Related Work

We are not aware of any other approach (bespoke or generic) that implements compact data structures in hardware for EHD devices (operate using $<$ 1 mW power). There is some work in processing sensor data using 10-100$\times$ more power.

The existing literature achieves space efficiency either by reducing the number of samples being sensed (based on the correlation between the samples) [13] or by compressing the sampled data using lossy or lossless compression techniques [14; 15]. Generic compression and decompression are not feasible with such low power budgets (see [16] that uses $\approx$ 100 mW of power at 16 MHz), and it is not possible

	HT	Lists	Tries
Compaction Ratio $\left(\frac{MF_R}{MF_C}\right)$	1.31	1.36	3.44
MF_R, MF_C: Memory Footprints of *regular* and *compact* representations.			

(a)

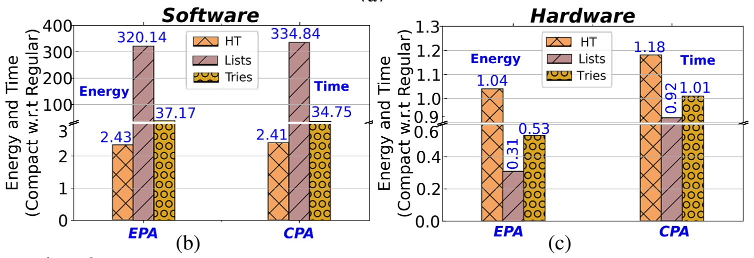


Fig. 3: Comparison between compact and regular data structures **(a)** Memory gain, **(b-c)** Energy and time results for **(b)** the software implementations and **(c)** the hardware implementations

to query these structures without maintaining uncompressed versions in memory. Second, it is hard to find and leverage the temporal relationships in a data stream [15] in EHD devices. Our approaches are agnostic to the data, are lossless, and allow efficient querying.

VI. Conclusion

We propose a generic architecture for implementing different compact data structures used in sensor devices, notably EHDs. The results show that all of them can operate within a power budget of 700 μW and, in many cases, are more time and energy-efficient than their regular data structure counterparts. The generic template can be used to instantiate a large number of popular CDSs and data structures that are used in a wide variety of sensor devices.

References

[1] P. Singla and S. R. Sarangi, “A Survey and Experimental Analysis of Checkpointing Techniques for Energy Harvesting Devices,” *Journal of Systems Architecture*, 2022.

[2] P. Singla, S. Singh, and S. Sarangi, “FlexiCheck: An Adaptive Checkpointing Architecture for Energy Harvesting Devices,” in *DATE*, 2019.

[3] Market Research Report. (2020) Energy Harvesting System Market. https://www.marketsandmarkets.com/Market-Reports/energy-harvesting-market-734.html. MarketsandMarkets Research Private Ltd.

[4] P. Singla, C. Goodchild, and S. R. Sarangi, “EHDSktch: A Generic Low Power Architecture for Sketching in EHDs,” in *ASPDAC*, 2021.

[5] Z. Han, Y. Li, and J. Li, “A novel routing algorithm for iot cloud based on hash offset tree,” *Future Generation Computer Systems*, vol. 86, pp. 456–463, 2018.

[6] G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy, “Ultra-low power data storage for sensor networks,” *ACM Transactions on Sensor Networks*, vol. 5, no. 4, pp. 1–34, 2009.

[7] Mouser Electronics. (2022) . https://www.mouser.com.

[8] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[9] D. Tong and V. K. Prasanna, “Sketch acceleration on fpga and its applications in network anomaly detection,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 929–942, 2017.

[10] S. R. Sarangi *et al.*, “Tejas: A Java based Versatile Micro-architectural Simulator,” in *PATMOS*, 2015.

[11] S. A. Abdulzahra, A. K. M. Al-Qurabat, and A. K. Idrees, “Data reduction based on compression technique for big data in iot,” in *ESCI*. IEEE, 2020.

[12] Samuel Madden, “Intel Lab Data,” http://db.csail.mit.edu/labdata/labdata.html, MIT Comp. Sc. and Artificial Intelligence Lab, 2004.

[13] S. Silvestri, R. Urgaonkar, M. Zafer, and B. J. Ko, “A Framework for the Inference of Sensing Measurements Based on Correlation,” *ACM Transactions on Sensor Networks*, vol. 15, no. 1, pp. 1–28, 2018.

[14] J. Azar, A. Makhoul, M. Barhamgi, and R. Couturier, “An energy efficient IoT data compression approach for edge machine learning,” *Future Generation Computer Systems*, vol. 96, pp. 168–175, 2019.

[15] D. Zordan, T. Melodia, and M. Rossi, “On the Design of Temporal Compression Strategies for Energy Harvesting Sensor Networks,” *IEEE Trans. on Wireless Communications*, vol. 15, no. 2, pp. 1336–1352, 2015.

[16] C. P. Antonopoulos and N. S. Voros, “A Data Compression Hardware Accelerator Enabling Long-Term Biosignal Monitoring Based on Ultra-Low Power IoT Platforms,” *Electronics*, vol. 6, no. 3, p. 54, 2017.