An Efficient Data Structure for Sparse Bit-Vectors with Applications in Neuromorphic Computing

Prafull Purohit
Electrical Engineering
Yale University
New Haven, CT, United States
prafull.purohit@yale.edu

Johannes Leugering
Broadband and Broadcastdept
Fraunhofer Institute for Integrated Circuits (IIS)
Erlangen, Germany
johannes.leugering@iis.fraunhofer.de

Rajit Manohar
Electrical Engineering
Yale University
New Haven, CT, United States
rajit.manohar@yale.edu

***Abstract*—Iterating over all the elements of a set is a very common problem in highly parallel systems. In hardware, this is typically realized by either storing the set-membership of each element in a fixed-length bit-vector or by storing just the indices of the members in a dynamically sized queue. However, the former solution is only efficient in terms of memory and runtime if each element is a member of the set with approximately 50% probability, whereas the latter is only efficient if the set is extremely sparse. We propose an alternative asynchronous, concurrent, distributed data structure based on a binary tree topology that is more efficient for sets with sparsity in between these two extremes. The proposed structure allows us to construct a set by adding individual elements one at a time in arbitrary order, and to iterate over all these elements exactly once, clearing the set in the process. We analyzed this data structure, simulated its behavior in CHP, synthesized it into an asynchronous digital circuit, optimized the circuits, and performed SPICE simulation to evaluate our design. The results confirm that our proposed structure offers a low-latency, low-power solution for moderately sparse data, and may thus prove useful for asynchronous and neuromorphic systems.**

***Index Terms*—data structure, neuromorphic systems, asynchronous circuits**

I. Introduction

Iterating over all the elements of a set is a very common problem in highly parallel systems. For example, consider a large number of asynchronous processing elements that need to serially access a shared resource, e.g. a bus. This requires recording (in arbitrary order) which subset of processing elements needs access, and then, once the shared resource becomes available, iterating over just that subset. The most common approaches to implement this rely on one of two different data structures: either they iterate over some form of fixed-length bit-vector, or they use some form of a dynamically sized queue [1]. Bit-vectors tend to perform well for dense subsets and queues tend to perform well for extremely sparse subsets, but neither approach is optimal – particularly not for the case we are most interested in, i.e. when the subset is "moderately sparse". In this paper, we propose a better solution for iterating over such "moderately sparse" subsets based on a binary tree structure, and we present an efficient implementation as a fully asynchronous digital circuit.

The problem we address can be formalized as follows: we want to iterate over all the elements of a sparse subset $S \subseteq \{0, \ldots, N-1\}$, where the sparsity parameter ρ determines the expected fraction of elements belonging to the subset, i.e. $|S| \approx M$ and $\rho = M/N$. We are looking for an efficient data structure that allows us to construct this set by adding individual elements one at a time in arbitrary order, where adding an element that is already in the set has no effect. We then need to be able to iterate over all elements in the set exactly once and subsequently clear the set.[1] In theory, such a data structure for a set of up to N independent elements and sparsity ρ must store at least $B = H_b(\rho)N$bit of information, where $H_b(\rho) = -\rho \log \rho - (1-\rho)\log(1-\rho)$ is the binary entropy function [2]. The minimum amount of energy required to transmit this information grows proportionally. For dense vectors (i.e. $\rho = 0.5$), this simplifies to just $B = N$bit; hence, a hardware implementation would need Nbit to account for the worst-case. Iterating over all elements requires at least $O(M)$ operations (i.e. one per element in the set).

A direct hardware implementation is to use a bit-vector of length N stored in memory, i.e. one 1 bit flag for each element, that can be independently raised to signal whether the corresponding element is part of the set S. This requires a memory of fixed size $O(N)$, and the amount of time and energy required to iterate over it likewise grows with $O(N)$. This solution meets most of our requirements, but it is only efficient if roughly half of the N elements belong to the subset S; for sparser subsets with $M \ll N$ the time and energy required for iteration are far from optimal.

An alternative data structure for this problem is a queue storing just the index of each member of the subset. On average, such a queue would need to store M indices of $\log_2 N$ bits each, resulting in an average case memory use of order $O(M \log N)$. Iterating over all of these entries would require at least on the order of $O(M)$ operations. For

[1] In our particular case, we are only interested in iterating over each set once, and therefore the clearing operation can be performed as part of the iteration. We also don't require the ability to query the set-membership of individual elements or to remove individual elements from the set.

very sparse subsets with $\rho \ll 1/\log N$, this could be much more efficient than the linear bit-vector implementation. In particular, if $\rho = 1/N$, i.e. we only expect to see a set of size $|S| = M = 1$, the queue only actually uses the theoretically optimal memory size of $\log N$ bits. However, if we need to size the queue for the worst-case scenario, i.e. where the entire set needs to fit in the queue and thus $M = N$, then the memory footprint increases to $O(N \log N)$, which is substantially worse than the linear bit-vector. But even for moderately sparse bit-sets with $1/N \ll \rho \ll 1$, a queue is not optimal for our problem: by storing the full index of each element, the queue implicitly holds a lot of redundant information, such as the insertion order and repetitions of the elements. This not only makes the queue less efficient, but it also makes the order of iteration non-deterministic from the perspective of a consumer of the queue. More importantly, to properly implement the semantics of a set, we would need to add a mechanism to prevent duplicates during insertion, which in turn would require additional memory and/or a (costly!) membership-query prior to each insertion.

Other popular data structures used to implement sets are hash tables and linked lists, but neither offers benefits for our specific situation: inserting into and iterating over hash-tables is at least as costly as it is for bit-vectors, and due to the random insertion order a linked-list implementation is equivalent to a queue[2]. In principle, compression algorithms could provide another opportunity to store the set in a memory-efficient representation, but they typically require a sequential stream of data in a predefined order (e.g. for run-length encoding) and/or require comparatively costly computations.

Since none of these approaches is satisfactory for our situation, i.e. for subsets with sparsity $1/N \ll \rho \ll 0.5$, we propose a new asynchronous, concurrent, distributed data structure based on binary tree topology. Our implementation can efficiently represent sparse sets of arbitrary size (ideally powers of two) and supports two operations:

- a `set` operation, which adds a single element to the set (in arbitrary order)
- and a `iterate-and-clear` operation, which iterates over all elements in the set (in a fixed order) and clears each element after reading

Variations of this implementation can produce either a highly compressed bit-stream representation of the set, or sequentially yield the full indices of all elements of the set. We provide two variations of the `set` operation, either of which may be more beneficial for specific applications: one provides serial access, allowing a member to be added to the set by sending index in a bit-serial stream, the other provides parallel access, allowing a member to be added to the set by directly sending the address through a bit-parallel input.

We analyzed this data structure, simulated its behavior in CHP, synthesized it into an optimized asynchronous digital circuit, and performed pre-layout SPICE simulation. The results confirm that our proposed structure is efficient, and may thus prove useful for asynchronous and, in particular, neuromorphic systems.

This paper is organized as follows. In Section II, we discuss a concrete example of the problem that motivated this research, as well as related work on this problem. Section III provides an overview of the proposed solution. Implementation details follow in Section IV. Section V presents simulation results and a comparison of different implementation options. Section VI concludes with a brief summary of the main contributions of this work.

II. Motivation and Prior Work

Asynchronous design excels at scenarios that involve sparse, distributed processing. Their event-driven nature enables fine-grained data-dependent dynamic power management, naturally leading to low power operation. An extreme example of this is the field of neuromorphic computing, where millions to billions of fine-grained computation units ("neurons") operate with extremely sparse, data-dependent activity [3]. While this is a natural application for asynchronous logic, having an explicit circuit for each computational element leads to another issue — the area occupied by such a large number of computational elements quickly becomes prohibitive.

To provide the required computation on a limited area, the standard approach is to time-multiplex computational units and use extremely dense storage to hold the state for each unit in a memory structure. Examples of this design point include larger neuromorphic devices such as IBM's TrueNorth [4] and Intel's Loihi [5] and Loihi2 [6], as well as other deep learning accelerators where multiply-accumulate blocks are time-multiplexed [7]–[9]. While implementing time-multiplexed operation is straightforward for "dense", pipelined and synchronous computations, i.e. where (almost) all compute units are simultaneously busy and operate in lockstep, the situation is more complicated when computations run asynchronously and communication is sparse, i.e. if only a small subset of compute units is active at any given time.

Consider a concrete example from neuromorphic computing, where we have an N-fold time-multiplexed logic implementing a set of N spiking neurons with binary output. The simulation proceeds step-wise, and in any step, only a subset of the neurons produces an output spike. At the end of each step, we need to know which neurons in the set $\{0, 1, \ldots, N-1\}$ just fired a spike in order to perform some operation for each of them (e.g. transmitting an outgoing message or performing an update of synaptic weights).

[2]If elements were inserted (or not inserted) in a predefined order, a linked list could make use of more efficient relative addressing, but this is not possible here due to random insertion order.

Our proposed solution is to use a distributed hardware data structure that implements a bit-vector with `set` and `iterate-and-clear` semantics as described in Section I. Whenever a particular neuron fires, it is added to the set via the `set` operation, and after each simulation step, the `iterate-and-clear` operation is used to retrieve the indices of all neurons that fired in the particular step. Simultaneously, the bit-vector is cleared in preparation for the next simulation step.

Similar problems naturally arise in a number of different neuromorphic architectures. For example, in the TrueNorth architecture, incoming spikes are buffered in an input spike buffer that uses one bit per input axon to record the spike arrival. When processing the input spikes for a particular time step, the TrueNorth design sequentially scans this memory buffer to check if there is a spike for each input axon for a particular time step. This is an implementation of the bit-set using a linear bit-vector as described above, with a time complexity of $O(N)$, where N is the number of input axons. Note that this complexity is independent of the sparsity of spike events because the state of every input axon is scanned at every step to determine if there is an input spike or not. This deviates from the otherwise asynchronous and event-driven principle of operation of a neuromorphic system, where the number of simultaneous spikes should be expected to be very low[3].

As we shall see, our proposed bit-vector design can solve this problem with better energy efficiency. It also recovers some of the event-driven nature of the underlying computation because the energy required for `iterate-and-clear` grows linearly with M rather than N.

III. A Concurrent Bit-vector Data Structure

A direct implementation of the software bit-vector algorithm can be written in CHP as follows[4]:

$$
\begin{array}{l}
BITVEC(N, C, I, O) \equiv \\
\quad *[\,[C?cmd; \\
\quad\quad [cmd = set \longrightarrow I?addr;\ \ bit[addr] := true \\
\quad\quad []cmd = itrclr \longrightarrow \\
\quad\quad\quad\quad i := 0; \\
\quad\quad\quad\quad *[i < N \longrightarrow [\ bit[i] \longrightarrow O!i; bit[i] := false \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad [] \ else \longrightarrow skip \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad]; \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad i := i + 1 \\
\quad\quad\quad\quad];\ \ O!\perp \\
\quad\quad] \\
\quad]
\end{array}
$$

[3]In fact, there is a trade-off between timing accuracy and sparsity: the higher the timing resolution, the shorter each time-step becomes, which in turn increases sparsity. Since the use of timing information is one of the key merits of neuromorphic architectures, this trade-off favors shorter time steps with sparser activation.

[4]A brief summary of the CHP notation is provided in the appendix

Here, C is the command channel that specifies the operation to be performed (*itrclr* is the `iterate-and-clear` operation described earlier); I specifies the bit to be set, for the `set` command; O is used to generate the address of the bits that are set, followed by a final special output ($\perp$) that indicates that there are no further outputs from the bit-vector.

A. Tree-based Implementation

As a first step towards a more efficient implementation, we apply a recursive decomposition. A bit-vector of size 2^k can be designed by combining two bit-vectors of half the size (assuming $k > 0$) as follows:

$$
\begin{array}{l}
TREENODE(k, C, I, O, Cl, Il, Ol, Cr, Ir, Or) \equiv \\
\quad *[\,[C?cmd; \\
\quad\quad [cmd = set \longrightarrow I?addr; \\
\quad\quad\quad\quad [addr < 2^{k-1} \longrightarrow Cl!cmd, Il!addr \\
\quad\quad\quad\quad []else \longrightarrow Cr!cmd, Ir!(addr - 2^{k-1}) \\
\quad\quad\quad\quad] \\
\quad\quad []cmd = itrclr \longrightarrow Cl!cmd,\ \ Cr!cmd; \\
\quad\quad\quad\quad Ol?v;\ \ *[v \neq \perp \longrightarrow O!v; Ol?v]; \\
\quad\quad\quad\quad Or?v;\ \ *[v \neq \perp \longrightarrow O!(v + 2^{k-1}); Or?v]; \\
\quad\quad\quad\quad O!\perp \\
\quad\quad] \\
\quad] \\
\\
BITVEC(2^k, C, I, O) \equiv \\
\quad\quad\quad TREENODE(k, C, I, O, Cl, Il, Ol, Cr, Ir, Or) \\
\quad\quad \| \ \ BITVEC(2^{k-1}, Cl, Il, Ol) \\
\quad\quad \| \ \ BITVEC(2^{k-1}, Cr, Ir, Or)
\end{array}
$$

This decomposition has a tree topology, where two bit-vectors of size 2^{k-1} are combined with a tree node to implement a bit-vector of size 2^k. (Note that the base case where $k = 0$ can be obtained by simplifying the software bit-vector algorithm described earlier.) The *TREENODE* process handles the two commands in the following way. To process the `set` command, it inspects the address; if it is less than 2^{k-1}, then the `set` command is propagated to the "left" sub-tree; otherwise, it is propagated to the "right" sub-tree, after reducing the address by 2^{k-1}. To process the `iterate-and-clear` command, the request is forwarded to both sub-trees. The resulting address streams from the two sub-trees are merged to produce the final address output sequence. Note that the comparisons and addition/subtraction operations can be simplified to simple bit operations (inspecting the most significant bit of *addr* and clearing or setting the most significant bit of a k-bit value respectively).

This tree-structured decomposition can be applied recursively, and we are left with a bit-vector design that has the following properties:

1) **Delay:**

- `Set` operations are accepted in constant time, independent of N;
- The `iterate-and-clear` operation has a response latency of $O(\log N)$, and after that, each output is produced in constant time.

2) **Energy:**

- `Set` operations must traverse the tree to its leaves, and hence require $O(\log N)$ energy since one tree node at each level of the tree is active;
- The `iterate-and-clear` operation requires $O(N)$ energy because every node in the tree is active. In addition, if there are M values stored in the bit-vector, another $O(M \log N)$ energy term is required to propagate the M addresses up the tree.

B. Optimizations

As seen in Section III-A, simply using a tree organization does not eliminate the $O(N)$ term from the energy required by the `iterate-and-clear` operation. We now introduce a sequence of optimizations and refinements that lead to our final optimized architecture.

1) Tracking if a sub-tree is empty: In the bit-vector decomposition from Section III-A, we refer to the CHP process at a particular location in a tree as a *tree node*. Since a tree node is responsible for both `set` and `iterate-and-clear` operations for the entire sub-tree rooted at the node, we can track if a sub-tree has any bits set. We introduce a local bit *has* that is true if the sub-tree has any bit set. If the *has* bit at a tree node is false, then we need not query the sub-tree to determine if there are any bits set. This optimization is shown below:

$TREENODE2(k, C, I, O, Cl, Il, Ol, Cr, Ir, Or) \equiv$

```
has↓;
*[C?cmd;
  [cmd = set ⟶ I?addr; has↑;
      [addr < 2^(k-1) ⟶ Cl!cmd, Il!addr
      []else ⟶ Cr!cmd, Ir!(addr − 2^(k-1))
      ]
  []cmd = itrclr ⟶
      [has ⟶ Cl!cmd,  Cr!cmd;
        Ol?v;  *[v ≠ ⊥ ⟶ O!v; Ol?v];
        Or?v;  *[v ≠ ⊥ ⟶ O!(v + 2^(k-1); Or?v];
      []else ⟶ skip
      ];
      O!⊥; has↓
  ]
]
```

To see the effect of this optimization, consider the case when a single bit is set in the bit-vector. This particular optimization will reduce the energy required to read the bit-vector to $O(\log N)$ from $O(N)$ for the previous solutions.

2) Tracking both sub-trees: The next observation is that although we can use the *has* bit to prune sub-tree operations, the parent node in the tree must make a request to a sub-tree to determine if that sub-tree is empty or not. However, the parent node can in fact track if a specific sub-tree is empty or not during the set phase. To this end, we replace the *has* bit with two bits: *has*0 and *has*1 to track if the left sub-tree or right sub-tree is empty.

$TREENODE3(k, C, I, O, Cl, Il, Ol, Cr, Ir, Or) \equiv$

```
has0↓, has1↓;
*[C?cmd;
  [cmd = set ⟶ I?addr;
    [addr < 2^(k-1) ⟶ Cl!cmd, Il!addr; has0↑
    []else ⟶ Cr!cmd, Ir!(addr − 2^(k-1)); has1↑
    ]
  []cmd = itrclr ⟶
    [has0 ⟶ Cl!cmd  []else ⟶ skip],
    [has1 ⟶ Cr!cmd  []else ⟶ skip];
    [has0 ⟶  Ol?v;
      *[v ≠ ⊥ ⟶ O!v; Ol?v]
    []else ⟶ skip
    ];
    [has1 ⟶ Or?v;
      *[v ≠ ⊥ ⟶ O!(v + 2^(k-1)); Or?v]
    []else ⟶ skip
    ];
    O!⊥; has0↓, has1↓
  ]
]
```

In this version, a sub-tree only receives a request if it has a set bit-position. With this optimization, it is easy to see that the energy for the `iterate-and-clear` operation reduces to $O(M \log N)$; the $O(N)$ term is eliminated. At first it may appear that we have doubled the storage requirements by doubling the number of *has* bits at a tree node. However, note that we can simply delete the *has* bits from the leaves of the tree—i.e. we do not increase the total bits stored. Another way to think about this is that we have relocated the original *has* bits from the two child nodes of a tree node to their parent.

C. Bit-serial Readout

The latency of reading the address during the `iterate-and-clear` operation is $O(\log N)$, since the request for the address must propagate to the leaf of the tree, and the address value propagates up in response. However, note that when we request data from the left (right) sub-tree, we *know* the most-significant bit of the address—it must be zero (one).

To take advantage of this information, we change the output channel into a *bit-serial* output where the bits on the output arrive MSB-first. With this representation, we

need a few additional symbols beyond simply 0 and 1—in particular, we need a symbol to separate successive addresses, and a final symbol to indicate that all the addresses have been read out of the bit-vector data structure ($\perp$ in the description so far). We refer to the symbol which separates successive addresses as *END_ENTRY* and the final symbol which indicates the end of all the addresses of the bit-vector is referred as *END_ALL*. The result is shown in Fig. 1.

It is easy to see that in this version of the bit-vector data structure, the output bits arrive in constant response time—independent of N. Note that the first complete address still requires $O(\log N)$ steps to receive the least significant bit. For the `set` operation, a simple inspection of the MSB of the bit-address is sufficient to determine whether the bit-address corresponds to the left sub-tree or right sub-tree and is used to forward the request down the tree along with the remaining bits of the bit-address. During the `iterate-and-clear` operation, each node sends the request to the sub-trees and then, depending on whether that sub-tree is empty or not, iterates on the left sub-tree and right sub-tree to collect output data and propagate up. The *has*0 (*has*1) tracks if the left (right) sub-tree is empty or not. Therefore, we can start by sending a 0 or 1 representing the MSB of the bit-address and then send the remaining data up the tree as it is received from the sub-tree for each bit-address. Since the left and right sub-tree represent different address ranges, the symbol received from the left sub-tree indicating the end of all addresses does not necessarily mean the end of all the addresses in the tree. The symbol *END_ALL* is replaced with *END_ENTRY* if the right sub-tree is not empty.

Now, considering the advantages of the bit-serial readout, we can use a similar approach for the `set` operation and receive the address value as a serial stream of bits. In this case, we need to add an extra symbol to indicate the end of the address (*END_ALL*). However, we do not need the second symbol to separate successive addresses (*END_ENTRY*) as we send only one address for each command request. Note that the parallel write implementation would need to receive all bits of the address before sending it to the left or right sub-tree whereas the bit-serial implementation can send a serial bit stream to the sub-tree as it arrives. In the version with bit-serial addresses for the `set` operation as well as the `iterate-and-clear` operation, the size of the datapath for the bit-vector design is independent of N.

D. Energy Comparison at Bit-level Granularity

To compare the different versions of the bit-vector design, we re-examine the energy required in terms of bit-operations.

The `set` operation activates a path of length $O(\log N)$ in the tree. If we look at the energy complexity at the bit-level granularity, then there are $\log N$ bits being processed by the root of the tree, $(\log N - 1)$ bits being processed by the next level of the tree, etc. Hence, the total energy-complexity in terms of bit-operations would be $O(\log^2 N)$.

A similar analysis can be used for the bit-parallel `iterate-and-clear` operation. While there are $O(M \log N)$ operations, the energy complexity taking into account individual bit operations would be $O(M \log^2 N)$.

An analysis of the `iterate-and-clear` operation for bit-serial readout also shows that the energy complexity is $O(M \log^2 N)$. Hence, the bit-serial and bit-parallel readout circuits have the same energy complexity.

IV. Circuit Implementation and Evaluation

As shown in Fig. 2(a), the Communicating Hardware Processes (CHP) description from Fig. 1 can be viewed as three separate overlays or logical layers for command, input address, and output serial stream such that each node in the tree contains a member from the three layers, i.e. command, address, and output. The command layer is responsible for generating control signals for the other two layers and synchronizing the data movement. The address and output layers are responsible for moving address data in and out of the tree structure. All units in a node communicate with their parent node and child node through asynchronous channels. Apart from the vertical interconnection paths, all three units in a node communicate with each other and synchronize data movement through asynchronous channels. The division of the request and data circuits into separate logical layers provides flexibility to choose a different implementation for each layer and can be optimized for a given performance metrics such as area, power, or timing. The modification of the `set` operation from bit-parallel to bit-serial is an example of such optimization.

A. `Set` Operation

When an event for a particular computation arrives, as shown in Fig. 2(b), the data structure receives a command (*cmd*) on channel *C* (indicating the `set` operation) and bit-address. After receiving the `set` request, the command unit requests the MSB of the bit-address from the address unit through channel *A*. Depending on the value of returned MSB, the control unit performs two operations. It sets either flag *has*0 or *has*1, to indicate whether the bit-address belongs to the left sub-tree or the right sub-tree. It then forwards the `set` command to one of the child nodes on channel *Cl* or *Cr*.

After receiving the request from the control unit on channel *A*, the address unit receives the index on input address channel *I* and returns the MSB to the control unit. Depending on the MSB of the address, the address unit sends the remaining address bits to either the left sub-tree or the right sub-tree on channel *Il* or *Ir*.

The command and address propagate recursively from the root of the tree structure to the leaf node such that the address value reduces by 1-bit in every node. The distributed

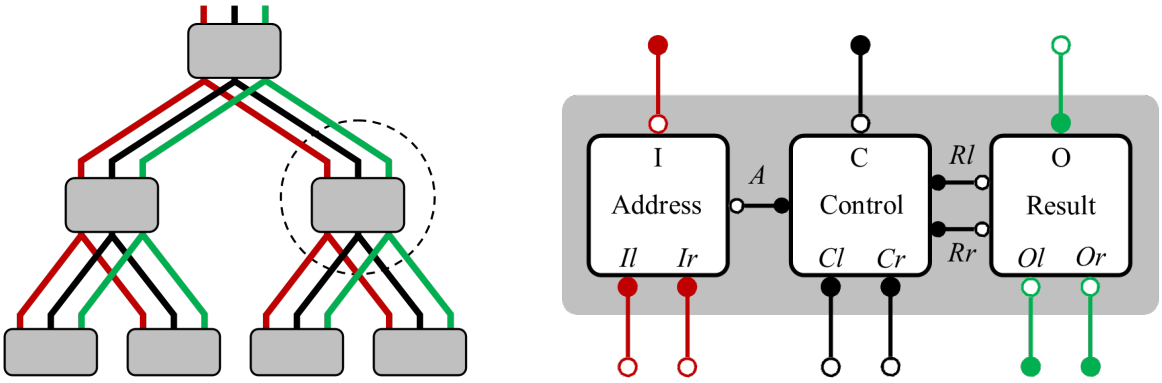
```
has0↓, has1↓;
*[C?cmd;
  [ cmd = set ⟶ I?addr;
    [ addr{MSB} = 0 ⟶ has0↑, Cl!cmd, Il!addr{MSB-1..0}
    [] else         ⟶ has1↑, Cr!cmd, Ir!addr{MSB-1..0}
    ]
  [] cmd = itrclr ⟶
    [ has0 ⟶ Cl!cmd [] else ⟶ skip ],
    [ has1 ⟶ Cr!cmd [] else ⟶ skip ];

    [ has0 ⟶
      *[ O!0;
        *[ Ol?v;
          O!(has1 ∧ v = END_ALL ? END_ENTRY : v) ← v ≠ (END_ALL ∨ END_ENTRY)
        ]
        ← v ≠ END_ALL
      ]
    [] else ⟶ skip
    ];

    [ has1 ⟶
      *[ O!1;
        *[ Or?v; O!v ← v ≠ (END_ALL ∨ END_ENTRY) ]
        ← v ≠ END_ALL
      ]
    [] else ⟶ skip
    ];

    has0↓, has1↓
    ]
]
```

Fig. 1: CHP description of the bit-vector data structure

implementation of the information flow allows pipelined operation for setting a bit-position in the bit-vector, i.e. the root node can accept a new event address while the previous event address propagates through the child nodes.

B. `Iterate-and-clear` Operation

The `iterate-and-clear` operation consists of multiple tasks. When the control unit receives the command to read indexes of all bit-positions which are set, it forwards the `iterate-and-clear` command to the child sub-tree which has any set bit-position (indicated by *has*0 or *has*1). After forwarding the command, the control unit iterates on both sub-trees to read bit-addresses. It starts communication on channel *Rl* and *Rr* sequentially based on the status of the *has*0 and *has*1 flags. The result unit, after receiving the readout request from the control unit, initiates *itrclr* operation by sending out a 0 or 1 depending on the local *has* flag that represents the MSB in the address. For example, if *has*0 corresponds to the left branch of the tree then the MSB of any bit-address from the left sub-tree can be represented by *has*0. Similarly, *has*1 represents the MSB for the right sub-tree. Due to the local storage of the MSB of address, the result unit can start sending out address bits as soon as it receives the command to readout bit addresses, offering a constant response time for the `iterate-and-clear` operation. After sending the MSB, the result unit requests the remaining bits from the child node and sends it up the tree. A special symbol is used to indicate the end of an address entry (END_ENTRY) and the end of all bit-positions (END_ALL) in the output serial stream. While reading out the left sub-tree, the result unit replaces any END_ALL token with END_ENTRY if the right sub-tree has any bit-position set. After reading out addresses of all bit-position that are set in its sub-trees, the result unit clears *has*0 and *has*1 flags throughout the hierarchy to clear the address of bit-positions that were set.



(a) (b)

Fig. 2: Bit-vector tree. (a) Binary tree. (b) Unit block.
⊸ denotes the passive end of the channel and ⊸• denotes the active end of the channel.

We decompose the CHP into three concurrent components: one that handes the control information, one for address processing for the `set` command, and one that handles the address generation for the `iterate-and-clear` command.

- **Control unit:**
  The control unit communicates with the address and result unit to synchronize the data movement based on the command request. The decomposed CHP of the control unit is given in Fig. 3. Active communication channels *Cl* (command-left) and *Cr* (command-right) are used to communicate the request command with the child sub-tree. When the control unit receives the `set` command on channel *C*, it requests the MSB of the bit-address from the address unit on pull-channel *A* to select the sub-tree corresponding to the bit-address. If the received MSB corresponds to an address in the left sub-tree, the control unit sets *has*0 and forwards the command to the left sub-tree on channel *Cl*. Otherwise, it sets *has*1 and forwards the command to the right sub-tree on channel *Cr*. When the control unit receives the `iterate-and-clear` command on channel *C*, it forwards the request to sub-trees if their corresponding *has* bit is set. The control unit then initiates readout from the left sub-tree by communicating with the result unit on channel *Rl*. Since the result unit needs to replace any *END_ALL* value with *END_ENTRY* if any bit-address is set in the right sub-tree, the value of *has*1 is also sent as part of the communication on channel *Rl*. After reading out all the bit-addresses from the left sub-tree, the control unit initiates readout from the right sub-tree through communication on channel *Rr*. Finally, the control unit clears *has* bits after iterating through both sub-trees.

- **Address unit:**

```
Control_Unit ≡
    *[ C?cmd;
       [ cmd = set ⟶ A?x;
           [ x = 0 ⟶ has0↑, Cl!cmd
           [] x = 1 ⟶ has1↑, Cr!cmd
           ]
       [] cmd = itrclr ⟶
           [ has0 ⟶ Cl!cmd [] else ⟶ skip],
           [ has1 ⟶ Cr!cmd [] else ⟶ skip];

           [ has0 ⟶ Rl!has1; Rl![] else ⟶ skip];
           [ has1 ⟶ Rr!; Rr![] else ⟶ skip];
           has0↓, has1↓
       ]
    ]
```

Fig. 3: CHP description of the control unit

The address unit is responsible for receiving bit-address for `set` operation and propagating it down the tree. As shown in Fig. 4, the address unit receives bit-address on channel *I*. The MSB of the bit-address is sent to the control unit on channel *A* for updating the *has* bits and pushing the command request down the tree. Based on the MSB of the bit-address, the address unit then forwards the received address to either the left sub-tree or the right sub-tree. Because we store the MSB as *has* bits and use this information to decide whether the received address is sent to the left sub-tree or the right sub-tree, we do not need to send the MSB. An important thing to note here is that the address unit will always have to receive the complete bit-address before sending it down the tree and the write delay will be something that depends on *N*.
In order to improve the throughput, we implemented

a bit-serial version of the address unit. As shown in the CHP (Fig. 5), the address unit sends the MSB to the control unit on channel *A* and uses this information to forward the rest of the bit stream to either the left sub-tree (channel *Al*) or right sub-tree (channel *Ar*). A special value (*END_ALL*) indicates the end of bit stream.

```
Address_Unit_Parallel ≡
    *[I?addr;
      A!addr{MSB},
      [ addr{MSB} = 0 ⟶ Il!addr{MSB-1..0}
      [] else ⟶ Ir!addr{MSB-1..0}
      ]
    ]
```

Fig. 4: CHP description of the bit-parallel address unit

```
Address_Unit_Serial ≡
    *[I?MSB; A!MSB,
      *[[ MSB = 0 ⟶ I?v; Il!v
        [] else ⟶ I?v; Ir!v
        ] ← v ≠ END_ALL
       ]
    ]
```

Fig. 5: CHP description of the bit-serial address unit

- **Result unit:**
  The result unit, shown in Fig. 6, is implemented as two independent processes for iterating over the left sub-tree and right sub-tree respectively. The process to readout the left sub-tree waits for communication on channel *Rl*. After receiving a request from the control unit, it starts by sending a 0 indicating the MSB of the left sub-tree. The result unit then receives the serial bit stream on passive channel *Ol* and sends it on active channel *O* while the received value does not correspond to the end of an address (END_ENTRY) or the end of all the addresses (END_ALL) in that sub-tree. After receiving an END_ENTRY symbol, the result unit repeats by sending a 0 corresponding to the MSB of a new address. The process terminates after receiving a value indicating the end of all addresses (END_ALL). An END_ALL symbol emitted by a left sub-tree is replaced by an END_ENTRY symbol if the corresponding right sub-tree is not empty.
  Similarly, the process iterates over the right sub-tree after receiving a request on the communication channel *Rr*. Because the control unit iterates on the left sub-tree and then the right sub-tree, we know that an END_ALL symbol from the right sub-tree means end of all addresses in the tree and should not be replaced.

C. Evaluation

We implemented the bit-vector data structure using the Quasi Delay-Insensitive (QDI) circuit family. The decomposed CHP described earlier was translated into production rules and SPICE netlist for CMOS implementation using Martin's synthesis method [10] and the open-source ACT EDA flow for digital asynchronous circuits [11].

The data structure discussed above has been implemented in a standard 65 nm bulk CMOS technology and simulated with a 1V supply at a nominal device temperature of 25 °C to verify the correct functionality of the circuit implementation. In order to account for parasitics and get a more accurate estimate of the circuit performance, we added a small capacitance to the output of every gate in the circuit. The SPICE simulations were performed using Synopsys FineSim, a high-performance circuit simulator.

We simulated the proposed structure with serial and parallel address inputs to evaluate performance for 256 and 1024 bit-positions. A bit-position was randomly selected and latency for writing and reading the address of the selected bit-position was measured. To evaluate the performance of the two implementations, we also measured average power consumption while measuring the latency.

Based on the simulation results listed in Table I, we observe that, for `set` operation:

- bit-serial `set` operation has a constant response time of approx. 1 ns per bit, whereas the delay in the bit-parallel implementation increase from 3.2 ns (0.4 ns/bit) to 5.1 ns (0.51 ns/bit).
- bit-serial `set` operation consumes more energy than a bit-parallel operation due to the increased number of handshake operations.

For the `iterate-and-clear` operation:

- the delay and energy consumption remains similar between the two implementations with bit-serial and bit-parallel `set` operation. We believe the slight differences in the measured numbers are due to differences in the leakage power of both circuits.

The linear bit-vector structure discussed in Section III can be implemented directly by using static random-access memory (SRAM) array for storage and some decoding logic for accessing the memory locations. Such implementations often have average active power consumption in milliwatts (mW) from the memory itself [12], [13]. For comparison, our proposed event-driven and distributed data structure requires less than 1 mW of average power when active, offering a power-efficient solution for managing sparse events in applications such as neuromorphic computing.

Another factor that determines the performance of a neuromorphic system is the speed-up in computation compared to biological time scales. The readout time required to iterate

```
*[ Rl?has1;
     *[O!0;
         *[Ol?v;
           O!(has1 ∧ v = END_ALL ? END_ENTRY : v) ← v ≠ (END_ALL ∨ END_ENTRY)
          ] ← v ≠ END_ALL
      ]; Rl?
 ]
||
*[ Rr?;
     *[O!1;
         *[Or?v; O!v ← v ≠ (END_ALL ∨ END_ENTRY)
          ] ← v ≠ END_ALL
      ]; Rr?
 ]
```

Fig. 6: CHP description of the result unit

TABLE I: Simulation results and area estimate of the bit-vector data structure. (measurements for one randomly-selected bit-position)

Address	`set`		`iterate-and-clear`		Area estimate (# gates)
	Delay	Power	Delay	Power	
8-bit, Parallel	3.2 *ns*	0.57 *mW*	11.8 *ns*	0.75 *mW*	193,529
8-bit, Serial	8.9 *ns*	0.67 *mW*	11.9 *ns*	0.72 *mW*	237,058
10-bit, Parallel	5.1 *ns*	0.55 *mW*	14.7 *ns*	0.94 *mW*	776,470
10-bit, Serial	10.9 *ns*	0.81 *mW*	14.8 *ns*	0.86 *mW*	950,000

through a given data structure decides how fast the computation can advance to the next time step. For a 256-bit vector, assuming the worst case where all bit-positions are set, the proposed implementation would require 3.04 µs to iterate through all the addresses. Comparing this to the TrueNorth where a single time-step is approximately 1 ms, we show that our proposed design can process all the events within the time constraints while using less power consumption. We also remark that the generation of addresses can be overlapped with processing, since a new address is available every 11.8ns (11.9ns for serial read-out).

In order to get a better intuition of the circuit complexity, we also compare the estimated design area. The area is usually measured in μm^2 but this value depends on the fabrication technology, amount of layout optimizations, and the cell library. We compare the design area as *gate equivalents* (GE) where one GE is equal to the area of a 2-input NAND gate designed using gridded cell layout style [14]. Based on the GE, given in Table I, we observe that implementations with bit-serial address are significantly larger than corresponding implementations with bit-parallel address. We believe this difference in GE is due to the fact that bit-parallel implementation contains mainly a completion tree which consists of simple logic gates whereas the bit-serial implementation includes additional control logic for receiving serial bit-stream in a loop.

V. Summary

Iterating over all the elements of a set is a very common problem in highly parallel systems and an optimal data structure for this problem depends on the degree of sparsity in the set. Most common approaches either iterate over some fixed-length bit-vector, which is efficient when the set is densely populated, or they implement some type of dynamic queue, which is efficient for extremely sparsely populated sets. However, neither of these works well in cases where the set is moderately populated. We proposed a better concurrent data structure for iterating over a moderately sparse subset. We started with a simple sequential implementation of the bit-vector and, using different optimization and refinement, obtained an energy- and latency-efficient implementation based on the binary tree structure. In order to take further advantage of the hierarchical request and data movement, we implemented a bit-serial readout where output bits are produced in constant response time. We pipelined the writing logic for the `set` operation and evaluated two implementations using bit-serial and bit-parallel input channels. Finally, we synthesized and evaluated our

implementation using SPICE, which validates our arguments about circuit complexity. The bit-serial implementation offers higher throughput due to pipelined readout, but it consumes more energy due to additional handshakes. The bit-parallel implementation has lower power consumption but suffers from slower throughput due to additional delays arising from the completion circuit. The readout circuit offers the same speed-up in both implementations but draws more power for the bit-parallel case, which we believe is due to the leakage from the extra circuitry.

Appendix

The circuit functionality is described using Communicating Hardware Processes (CHP) language and the key notations of the CHP syntax are summarized below:

- Skip: No operation
- Send: $X!v$ means send the value of v over channel X.
- Receive: $X?v$ means receive a value on channel X and store it in variable v.
- Probe: $\overline{X}$ determines if there is a pending communication on a channel X
- Assignment: $a := b$ means assign the value of b to a.
- Sequential Composition: $S1; S2$ means execute statements $S1$ and $S2$ sequentially
- Parallel Composition: $S1, S2$ means execute statements $S1$ and $S2$ in parallel
- Deterministic Selection: $[G1 \rightarrow S1 \; [] \; ... \; [] \; Gn \rightarrow Sn]$ waits until one of the guards $(G1, G2...Gn)$ is *true* and then execute corresponding statement. Requires that the guards must be mutually exclusive
- Non-Deterministic Selection: $[G1 \rightarrow S1 \mid ... \mid Gn \rightarrow Sn]$ is same as the Deterministic Selection except guards don't have to be mutually exclusive
- Repetition: $*[S]$ infinitely repeats statement S
- Do-while loop: $*[S \leftarrow G]$ executes statement S, and then evaluates the guard G. If G is true, then the loop repeats; otherwise, the loop terminates.

ACKNOWLEDGMENT

This work was supported in part by DARPA IDEA grant FA8650-18-2-7850, and in part by DARPA POSH grant HR001117S0054-FP-042.

References

[1] S. Debasis, *Classic Data Structures, 2nd ed.* Prentice Hall India Pvt., Limited, 2008, ISBN: 9788120337312.

[2] T. Cover and J. Thomas, *Elements of Information Theory*. Wiley, 2012, ISBN: 9781118585771.

[3] S.-C. Liu, T. Delbruck, G. Indiveri, A. Whatley, and R. Douglas, *Event-based neuromorphic systems*. John Wiley & Sons, 2014.

[4] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[5] M. Davies, N. Srinivasa, T.-H. Lin, *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *Ieee Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[6] G. Orchard, E. P. Frady, D. B. D. Rubin, *et al.*, "Efficient neuromorphic signal processing with loihi 2," in *2021 IEEE Workshop on Signal Processing Systems (SiPS)*, IEEE, 2021, pp. 254–259.

[7] A. Ankit, A. Sengupta, P. Panda, and K. Roy, "Resparc: A reconfigurable and energy-efficient architecture with memristive crossbars for deep spiking neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[8] R. Wang, G. Cohen, K. M. Stiefel, T. J. Hamilton, J. Tapson, and A. van Schaik, "An fpga implementation of a polychronous spiking neural network with delay adaptation," *Frontiers in neuroscience*, vol. 7, p. 14, 2013.

[9] C. Frenkel, M. Lefebvre, J.-D. Legat, and D. Bol, "A 0.086-mm^2 12.7-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm cmos," *IEEE transactions on biomedical circuits and systems*, vol. 13, no. 1, pp. 145–158, 2018.

[10] A. J. Martin, "Compiling communicating processes into delay-insensitive VLSI circuits," 1986.

[11] S. Ataei, W. Hua, Y. Yang, *et al.*, "An open-source eda flow for asynchronous logic," *IEEE Design & Test*, vol. 38, no. 2, pp. 27–37, 2021.

[12] S. Ataei and R. Manohar, "Amc: An asynchronous memory compiler," in *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, IEEE, 2019, pp. 1–8.

[13] J. Dama and A. Lines, "Ghz asynchronous sram in 65nm," in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, IEEE, 2009, pp. 85–94.

[14] Y. Yang, J. He, and R. Manohar, "Dali: A gridded cell placement flow," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.