Interval Stabbing on the Automata Processor

Indranil Roy[a], Ankit Srivastava[a,*], Matt Grimm[b], Srinivas Aluru[a]

[a] *School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332*
[b] *Micron Technology, Inc., 8000 S Federal Way, Boise, ID 83716*

Abstract

The Automata Processor (AP) was designed for string-pattern matching. In this paper, we showcase its use to execute integer and floating-point comparisons and apply the same to accelerate *interval stabbing queries*. An interval stabbing query determines which of the intervals in a set overlap a query point. Such queries are often used in computational geometry, pattern matching, database management systems, and geographic information systems. The check for each interval is programmed as a single automaton and multiple automata are executed in parallel to provide significant performance gains. While handling 32-bit integers or single-precision floating-point numbers, up to 2.75 trillion comparisons can be executed per second, whereas 0.79 trillion comparisons per second can be completed for 64-bit integers or double-precision floating-point numbers. Additionally, our solution leaves the intervals in the set unordered allowing addition or deletion of an interval in constant time. This is not possible for contemporary solutions wherein the intervals are ordered, making update of intervals complex. Our automata designs are modular allowing them to become constituent parts of larger automata, where the numerical comparisons are part of the overall pattern matching operation. We have validated the designs on the hardware, and the routines to generate the necessary automata and execute them on the AP will be made available as software libraries shortly.

Keywords: finite automata, interval stabbing, Automata Processor

1. Introduction

An *interval stabbing* query identifies the intervals from a list that overlap (or are *stabbed* by) a query point. These queries find usage in a variety of applications such as computational geometry, pattern matching, database management systems, geographic information systems, algorithmic trading, etc. If the intervals are overlapping, state-of-the-art algorithms order the intervals into

*Corresponding author.
Email addresses: iroy@gatech.edu (Indranil Roy), asrivast@gatech.edu (Ankit Srivastava), mgrimm@micron.com (Matt Grimm), aluru@cc.gatech.edu (Srinivas Aluru)

search-trees and use these trees to generate the solution. However, the runtime of any algorithm is output-sensitive, i.e. its run-time complexity is at least dependent on the number of the intervals in the output. Therefore, in the worst case, the runtime may degrade to the order of the number of intervals in the list. Additionally, any modifications to the list may entail significant changes to the search-tree and may incur severe overheads.

The interval stabbing problem falls under a broader category of applications which depend on the execution of one-to-many comparisons quickly. The more such comparisons can be executed in parallel, the less ordered the list needs to be, thereby allowing more expeditious modifications to the list. Towards this end, we present an accelerated solution to the interval stabbing problem, using the Automata Processor (AP) [1, 2]. The AP is a reconfigurable coprocessor which can be programmed to compute a large set of user-defined Nondeterministic Finite Automata (NFAs) in parallel against a single query data-stream. We define one automaton for each interval in the list and execute tens of thousands of such automata in parallel against the query points streamed to the processor.

Our streaming solution has a variety of advantages. First, the fine grained parallelism allows significant acceleration over existing solutions. Second, the addition or deletion of an interval to the unordered list is a constant time operation. And third, the streaming solution is amenable to applications where the query point is embedded in a stream of data. In contemporary solutions, this query point has to be lexically parsed out of the stream before the interval stabbing query can be issued. Our modular designs, on the other hand, can be part of other larger NFA structures which allow the parsing of the data and the answering of the query simultaneously.

Another novelty of our solution is that this is the first known use of the AP for multi-byte integer and floating-point operations in their native formats. Hitherto, the processor has been typically used for string-pattern analysis [3, 4, 5, 6] or for graph analysis [7], by conversion to strings. The solution outlined in this paper extends the scope of use of the processor beyond its design intent. Additionally, the automata designs employed by this paper exemplify design techniques which maximize on-board resource utilization and minimize various compile-time and runtime overheads. These design techniques will be useful to anyone developing solutions on this processor.

In this paper, we begin by outlining the automata designs for fixed-width integers and IEEE floating-point numbers followed by the designs for variable-length numeric-strings. However, the applicability of the method extends to a variety of other datatypes, e.g. multi-dimensional coordinate points, and datestamps or timestamps. Notice that, in the last case, the query point and endpoints may be expressed in different formats. Therefore, by expressing the endpoints as regular expressions, the scope of acceleration on the AP is amplified.

The rest of the paper is organized as follows. First, we explore the state-of-the-art solutions for the interval stabbing problem in Section 2.1. Next, we provide a basic description of the AP architecture in Section 2.2, which is required to understand our AP based solution described in Section 3. We

present results from execution in hardware in Section 4 and describe the scope of future work to handle other data formats including higher-dimensional points, and multiple-format timestamps in Section 5. Finally, we conclude in Section 6.

2. Background

2.1. Interval Stabbing

Identifying the intervals from a set of intervals $I = \{i_1, i_2, \ldots, i_n\}$ which overlap with a query point q is a well studied problem in computational science. If the intervals are non-overlapping, then they can be sorted based on their starting points in $O(n \log n)$ time, the sorted list can be stored in $O(n)$ space, and the queries answered in $O(\log n)$ time. However, if the intervals are arbitrarily overlapping, then the solutions are more involved. We briefly discuss these below.

A brute-force linear search through all the intervals takes $O(n)$ time, which is asymptotically optimal as all the intervals may be stabbed by q. Nonetheless, algorithms have been developed which employ various data-structures to organize the intervals and answer queries in $O(\log n + k)$ or $O(1 + k)$ time, where k is the number of intervals reported by the query. These query times are output-sensitive when the number of outputs are $\Omega(\log n)$ or $\Omega(1)$, respectively.

One commonly used data-structure is called *interval trees* [8, 9, 10, 11]. Each node in an interval tree corresponds to a *center point*. All the intervals whose ending point is smaller than the center point are captured in the subtree rooted at the left child of the node, whereas all the intervals whose starting point is greater than the center point are captured in the subtree rooted at its right child. All the intervals stabbed by the center point are captured within the node, and sorted separately using their starting and ending points. Creation of this tree requires $O(n \log n)$ time, a storage complexity of $O(n)$, and an expected run-time of $O(\log n + k)$ per query.

Segment trees [12] are similar to the interval trees and identical in performance, except in the storage complexity which is $O(n \log n)$ for segment trees. However, the intervals within a node need not be sorted in any order. Schmidt [13] describes a faster data-structure to complete preprocessing in $O(n)$ time and a query in $O(1+k)$ time when the endpoints of the intervals are within a small integer range, e.g. $\{1, 2, \ldots, O(n)\}$.

All the data-structures described above are generally used in a static setting because addition and deletion of intervals is complex with difficult to estimate run-time. Cormen et al. [14] describe a modification to the interval trees called *augmented trees* wherein the insertion or deletion of an interval can be completed with $O(h)$ complexity, where h is the height of the tree. Another search-tree called *priority tree* [15] requires $O(n)$ space and supports insertion and deletion of intervals. However, the priority search-tree is very complex to implement, especially in its balanced form.

Hanson uses *interval skip list* [16] which is simpler to implement than the interval trees and takes an expected time of $O(\log^2 n)$ to insert or delete an

interval. Alstrup [17] describes another method to represent each interval as a coordinate in an $n \times n$ matrix, and then iteratively compute the nearest common ancestors in a *cartesian tree* (as shown by Gabow [18]) in $O(1 + k)$ time. However, this procedure has a significant implementation overhead.

Parallel solutions for variations of the interval stabbing problem have been explored using multiple threads on CPUs and GPUs. Chovanec & Krátkỳ [19] used Streaming SIMD Extensions (SSE) for performing multiple comparisons with the same query point in parallel. The authors then ported their solution for execution on GPUs [20] but didn't observe significant gains over the CPU solution because of the cost of CPU-GPU data transfer and thread-divergence on GPUs due to irregular tree traversals. Kim et al. [21] and Maramreddy & Kothapalli [22] tried to minimize the thread-divergence by proposing methods for transforming irregular tree traversals into sequential access with varying degree of success and is still an area of active research. Further, the solutions described above use parallel implementations of the data-structures discussed earlier which, similar to their sequential counterparts, have a significant cost attached to any modifications in the set of intervals.

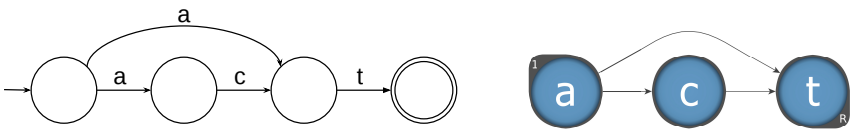
2.2. Automata Processor Basics

The AP is a Multiple Instruction Single Data (MISD) device which can propagate one symbol from the query data stream to every processing element in the chip in every clock-cycle. Each symbol is a 1-byte word. The multiple instructions are user-defined state machines, realized through connecting the processing elements using a reconfigurable routing network. Since all the processing elements receive every data symbol and all the routing lines can be activated in parallel, the state machines emulate NFAs in hardware which execute in parallel. The intrinsic parallelism in the hardware absolves the programmer from dealing with communication delays, race conditions, etc. and concentrate merely on the design of NFAs. Given the novelty of the processor, we now briefly describe the processing elements, routing matrix, programming environment, and programmable resources of the architecture.

2.2.1. Processing Elements

The representation of the automata to be executed on the AP is slightly different from the classical state-diagram representation. We will henceforth refer to the former as ANML-NFA because they are defined using Automata Network Markup Language (ANML, pronounced as animal). The basic processing element is called a State Transition Element (STE) which emulates an edge transition in a traditional state-diagram. The label of the STE is the same as the label of the edge transition and can be any character-class from the 1-byte symbol space. Routing lines between these STEs represent states in a classical state-diagram. All the STEs representing the incoming edges into a state are connected to the STEs representing all the outgoing edges of that state. All the STEs representing the outgoing edges of the start state are represented as *start-STE*s and all the incoming edges of the accept states are marked as

*reporting-STE*s. More details on converting classical NFA to ANML-NFA can be found in [23].



(a) state-diagram (b) ANML-NFA

Figure 1: Automaton to accept words *at* or *act*

The state-diagram and the ANML-NFA representation of an NFA that accepts words *act* and *at* is shown in Fig. 1a and Fig. 1b. The start-STE is shown with an indicator to the top-left with the number 1 in the indicator. The reporting-STE has an indicator to the bottom-right with the symbol R placed inside it.

At the beginning of processing a data-stream, only the start-STEs are active and the first byte of the data-stream is broadcasted. If the broadcasted symbol belongs to the character-class stored in the label of an active STE, then all the STEs connected to its outgoing routing lines are activated for the next clock-cycle. In the next clock-cycle, the subsequent byte in the data-stream is broadcasted and the process continues. If a reporting-STE is matched in a cycle, an output is generated identifying the STE and the offset in the data-stream where the match occurred.

2.2.2. Programming Resources

The programming elements in the AP are arranged hierarchically as follows. 16 STEs are arranged in a *row*, 16 rows in a *block*, 96 blocks in a *half-core*, and 2 half-cores in a chip. Cumulatively, each chip has a total of $49,152$ STEs. All the STEs in a row can be simultaneously connected to each other, while only 24 routing lines are present to connect the STEs in different rows within a block. The STEs within a block can only be connected to the STEs belonging to 8 adjacent blocks. There are no connections between the STEs of the two half-cores.

The AP-compiler completely abstracts the underlying layout while placing the processing elements in the user designs to the physical elements on the chip. Therefore, an automata designer may design automata in a completely layout-agnostic manner. However, experienced designers may consider the hierarchical layout while coming up with designs which can be placed by the compiler with higher resource utilization efficiency.

2.2.3. Processing Rate

An AP-chip functions at 128 MHz, processing a 1-byte symbol per clock-cycle, thus supporting an input data streaming rate of 1 Gbps. There are 32 chips on a single AP-board which can be organized into *logical-cores* of 2, 4,

or 8 chips. All the chips within a logical-core are presented the same data-stream. Separate logical-cores can process different data-streams concurrently. This provides the flexibility of executing a large number of NFAs against a single data-stream using bigger logical-cores, or smaller number of NFAs using smaller logical-cores for a higher combined throughput of up to 16 Gbps.

Output Handling Bottleneck. On the current generation of the chip, the output handling may significantly slow down the overall processing rate. Whenever reporting-STEs generate output in a cycle, an output-vector is created and stored in an output buffer. If the output buffer is sufficiently empty, the vector is stored in the same clock-cycle and the input processing continues unabated from the next cycle. Simultaneously, the output-vector is read out from the output buffer to the main memory of the host processor. This may take between 135 to 494 clock-cycles for each vector. Therefore, if the vectors are generated too frequently, then the output buffer fills up and the input processing has to be stalled till the buffer has been sufficiently emptied for the new output-vector. However, one aspect of the output handling often exploited by many applications [5, 4, 24, 25] is that the output processing rate is not determined by the number of outputs generated, but the number of clock-cycles on which they were generated. Therefore, these applications overcome this bottleneck by batching as many outputs into a single clock-cycle as possible.

2.2.4. Programming Environment

Executing a program on the AP consists of two stages: 1) configuring the processor, and 2) streaming the data to the processor and handling the generated output. The tools to design, evaluate, and compile the user-defined ANML-NFA are part of an AP Software Development Kit (AP-SDK).

In compiling ANML-NFAs, the AP-compiler uses a proprietary algorithm which is hidden from the user. Nevertheless, the place-and-route problem solved by the AP-compiler includes efficiently placing the programmable elements and connections in the user-defined automaton to hardware elements and connections on the board. This is similar to the segmented channel routing on FPGAs [26], known to be NP-hard. The widely variable compilation times for different input ANML-NFAs, between seconds to hours, reflects the same. However, the existence of compiler flags for different levels of optimizations indicates the presence of approximate algorithms. Further, the compiler allows for user-level optimizations when different ANML-NFAs are identical in their structure and differ only in the labels of the STEs. Such an automaton structure can be compiled as a *macro* with the STE labels parameterized. Once the macro has been compiled, replicating it with different labels takes in the order of a few milliseconds.

After the ANML-NFAs have been compiled, loading them into the AP-chips requires about 50 milliseconds for the entire board. A run-time environment can then be used to stream the queries to the AP-board and process the output. The run-time environment is implemented as a set of *C* API calls which have

bindings for other high level programming languages like *Python* and *Java*. The loader and the run-time environment is also part of the AP-SDK.

3. Methodology

For different applications, the query point and the endpoints of the intervals can be represented in various formats. In this paper, we look at the following formats: 4-byte and 8-byte integer formats, single and double precision IEEE floating-point numbers, and *numeric-strings* wherein the digits, sign, and decimal point in the number are represented using their ASCII-equivalent characters. Notice that, while the numbers represented in the integer and floating-point formats have a fixed length, the numbers represented as numeric-strings do not. For example, the numeric-string representation of -45 and 3.89 requires three and four bytes, respectively. Therefore, the automata designs for handling numbers in the integer and floating-point representations are similar to each other while differing from those designed to handle numeric-strings.

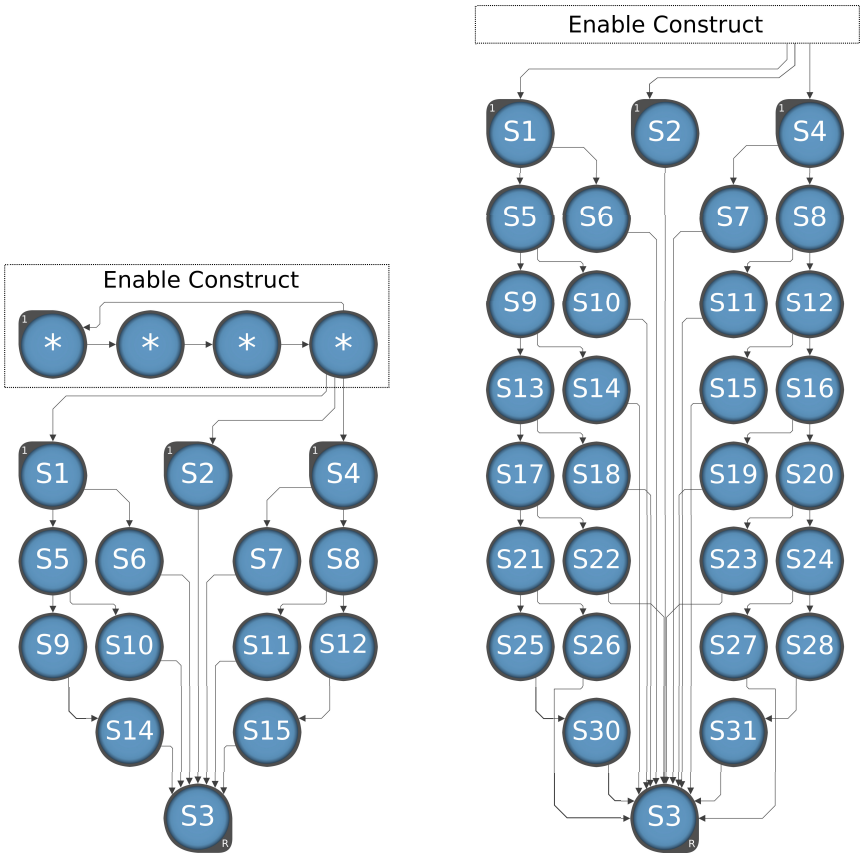
In this section, we present our automata designs for checking if the interval $[x, y]$ is overlapped by a query point z. The points x, y, and z are numbers which can be signed or unsigned, whole or real. For the sake of brevity, our examples only illustrate the case of closed intervals. Semi-open or open intervals can be handled using the same automata structures by modifying the labeling schemes slightly.

The automata designs presented in this section work as follows. Automata for the intervals are loaded into the AP and all the query points are concatenated together to form a single query data-stream. As the data-stream is processed by the AP, all the loaded automata concurrently process one query point at a time and multiple query points in succession. If a query point stabs any intervals, the corresponding automata ids and the offset in the data-stream where the matches occurred are reported to the host processor. Based on these automata ids and the reported offset, the intervals and the corresponding query point are looked-up on the host processor.

3.1. Handling Fixed-Width Format

Comparing multi-byte binary number representations using the AP poses two major challenges. First, the *endianness* of the representation must be taken into account while designing the automata and the state-information stored till all the bytes in the representation have been processed. Second, prevalent methods of representing signed binary numbers, such as the *two's complement* representation, place negative numbers in a higher lexicographical order than their positive counterparts.

We designed the automata as macros, where the labels of the STEs are parameterized. These macros can be pre-compiled and replicated with different labels for multiple intervals quickly. The macros for 4-byte and 8-byte numbers are shown in Fig. 2a and Fig. 2b. A b-byte comparator macro contains $4(b-1)+2$ parameterized STEs, which are tagged from $S1$ to $S < 4(b-1)+3 >$, with



(a) 4-byte comparator macro (b) 8-byte comparator macro

Figure 2: Macros for comparing binary representation of numbers

$S < 4(b-1)+1 >$ missing. This tagging scheme simplifies the labeling algorithm. In our figures, the tags are shown inside the STEs.

Although we limit our discussion to 4-byte and 8-byte numbers, the methodology described here is generic and applicable to handling any b-byte number. Our designs are tailored to handle the little-endian representation, i.e. a number is streamed from its Most Significant Byte (MSB) to its Least Significant Byte (LSB). Simple alterations to this automata design enables reversing the order of checking the bytes and hence handling the big-endian representation. However, for the sake of brevity, we have excluded a detailed explanation of the same from this paper.

The algorithms for assigning labels to the STEs for intervals and queries of unsigned integers, signed integers, and floating-point numbers are described below.

3.1.1. Unsigned Integers

We provide an overview of the 4-byte macro, shown in Fig. 2a, first. The processing starts with the STEs $S1$, $S2$, and $S4$ being active to process the first byte in the data-stream. Thereafter, the *Enable Construct* activates them again on every fourth byte in the data-stream. This ensures that these STEs process the first byte of every query number in the data-stream. The rest of the macro can be visualized as follows. Each row of STEs in the macro processes the same byte of a query number. Since all the unsigned integers are lexicographically ordered, z can be determined to be in the interval $[x, y]$ by comparing the bytes of z to the corresponding bytes of x and y from MSB to LSB. The STEs in the leftmost column are activated successively if the consecutive bytes of z match the corresponding bytes of x. Similarly, the STEs in the rightmost column are activated if the consecutive bytes of z match the bytes of y. If z is determined to be stabbing the interval based on the current byte being processed, then the STE in either the second from the left or the second from the right column, or both, activate $S3$. $S3$ is programmed to generate an output on any input byte, signaling that z overlaps the interval. The same concepts are used to extend the macro to handle 8-byte integers using more rows, and making the Enable Construct activate $S1$, $S2$, and $S4$ after every eight bytes.

Algorithm 1 is used to label the STEs for unsigned integer intervals. In the following discussion, we refer to the j^{th} most significant byte of x, y, and z as x_j, y_j, and z_j. $S2$ is labeled for matching all the bytes in the open interval (x_1, y_1) and, therefore, checks if z_1 is greater than x_1 and less than y_1. If the condition is satisfied, then z overlaps the interval and is reported via $S3$. On the other hand, if $z_1 \notin (x_1, y_1)$, then z_1 must be equal to x_1 or y_1 for z to overlap the interval. These cases are handled by $S1$ and $S4$, respectively. If matched, $S1$ or $S4$ activate the connected STEs in the second row for comparing z_2.

$S6$ and $S7$ handle the case where z can be determined to overlap (x, y) based on z_2. If $x_1 = z_1 = y_1$, then $S6$ and $S7$ are labeled to accept any z_2 in the range (x_2, y_2). Otherwise, if $x_1 = z_1 < y_1$, then z_2 can assume any value larger than x_2 and hence $S6$ is labeled as $(x_2, 255]$. By a similar logic, $S7$ is labeled to match the range $[0, y_2)$. If z_2 is matched by $S6$ or $S7$, then $z \in (x, y)$, and $S3$ is activated to generate an output in the next cycle. Otherwise, $S5$ and $S8$ handle the case wherein $z_2 = x_2$ and $z_2 = y_2$, respectively, and activate the STEs in the next row to process z_3 in an identical fashion. Finally, $S14$ and $S15$ are labeled based on x_4 and y_4 using a logic similar to the one used for the labeling of $S6$ and $S7$. However, since they process the last byte of z, the check for equality with x_4 and y_4 is also rolled into these STEs[1].

Notice that, this automata design requires one extra byte after the streaming of the last byte (LSB) of z to generate the report when $S3$ is activated by $S14$ or $S15$. For every query point in the stream, except for the last, the first byte of the subsequent query point allows $S3$ to report an overlap. However, the input

[1]If semi-open or open intervals are to be handled, the equality with x_4 or y_4, or both, could be left out accordingly.

Algorithm 1 Labeling STEs for unsigned integer intervals

Input:

- Binary representation of unsigned integers x and y. x_j and y_j denote the j^{th} MSB of x and y.
- Number of bytes in the representation of x and y, b.
- Labels of all the STEs in the b-byte comparator, L. L_t denotes the label of the STE tagged $S < t >$.

Ensure:

- $x \leq y$

```
1:  procedure LabelUnsigned(x, y, b, L)
2:      L_2 ← {(x_1, y_1)}
3:      L_3 ← {*}
4:      equalPrefix ← TRUE
5:      for j = 1 to b − 2 do
6:          L_{4(j−1)+1} ← {x_j}
7:          L_{4(j−1)+4} ← {y_j}
8:          if x_j ≠ y_j then
9:              equalPrefix ← FALSE
10:         end if
11:         if equalPrefix == TRUE then
                // x and y are either both
                // non-negative or both negative.
12:             L_{4j+2} ← {(x_{j+1}, y_{j+1})}
13:             L_{4j+3} ← {(x_{j+1}, y_{j+1})}
14:         else
15:             L_{4j+2} ← {(x_{j+1}, 255]}
16:             L_{4j+3} ← {[0, y_{j+1})}
17:         end if
18:     end for
19:     L_{4(b−2)+1} ← {x_{b−1}}
20:     L_{4(b−2)+4} ← {y_{b−1}}
21:     if equalPrefix == TRUE then
22:         L_{4(b−1)+2} ← {[x_b, y_b]}
23:         L_{4(b−1)+3} ← {[x_b, y_b]}
24:     else
25:         L_{4(b−1)+2} ← {[x_b, 255]}
26:         L_{4(b−1)+3} ← {[0, y_b]}
27:     end if
28: end procedure
```

stream needs to be padded at the end with a dummy byte to ensure that all the intervals stabbed by the last query point have a chance to report the overlap to the host processor. It must also be noted that the addition of this extra

byte does not generate any spurious output. This is because although all the automata interpret this extra byte as the first byte of the next query element, none of them have a chance to generate a report after processing only one byte.

3.1.2. Signed Integers

In this section, we describe a technique to handle signed integers represented in the *two's complement* representation. In this representation, the lower lexicographical half of the range is reserved for all the non-negative integers, whereas the upper half is for negative integers. However, the relative ordering between any two negative integers or any two non-negative integers is maintained. We use this property in our labeling algorithm for signed integers described in Algorithm 2.

Algorithm 2 Labeling STEs for signed integer intervals

Input:

- Two's complement binary representation of signed integers x and y. x_j and y_j denote the j^{th} MSB of x and y.
- Number of bytes in the representation of x and y, b.
- Labels of all the STEs in the b-byte comparator, L. L_t denotes the label of the STE tagged $S < t >$.

Ensure:

- $x \leq y$

1: **procedure** LABELSIGNED(x, y, b, L)
2: **if** ($x_1 \leq 127$ **and** $y_1 \leq 127$) **or** ($x_1 > 127$ **and** $y_1 > 127$) **then**
// $x \geq 0$ and $y \geq 0$ or $x < 0$ and $y < 0$.
3: LABELUNSIGNED(x, y, b, L)
4: **else**
// $x < 0$ and $y \geq 0$, since $x \leq y$.
5: $L_2 \leftarrow \{(x_1, 255], [0, y_1)\}$
6: $L_3 \leftarrow \{*\}$
7: **for** $j = 1$ **to** $b - 2$ **do**
8: $L_{4(j-1)+1} \leftarrow \{x_j\}$
9: $L_{4(j-1)+4} \leftarrow \{y_j\}$
10: $L_{4j+2} \leftarrow \{(x_{j+1}, 255]\}$
11: $L_{4j+3} \leftarrow \{[0, y_{j+1})\}$
12: **end for**
13: $L_{4(b-2)+1} \leftarrow \{x_{b-1}\}$
14: $L_{4(b-2)+4} \leftarrow \{y_{b-1}\}$
15: $L_{4(b-1)+2} \leftarrow \{[x_b, 255]\}$
16: $L_{4(b-1)+3} \leftarrow \{[0, y_b]\}$
17: **end if**
18: **end procedure**

If the interval $[x, y]$ is either fully negative or fully non-negative, the comparator for the interval is labeled as unsigned integers, using the procedure described in Algorithm 1. In the only other case, i.e. when the interval $[x, y]$ is part negative and part non-negative, x must be negative and y must be non-negative. Hence $x_1 \neq y_1$. Recall that, if $x_1 \neq y_1$ in the algorithm for labeling unsigned integers, the labels of the STEs in the two columns on the left do not depend on the value of y and the labels of the STEs in the two columns on the right do not depend on the value of x. Therefore, we can again label all the STEs as unsigned integers, except for $S2$. $S2$ activates $S3$ only if processing z_1 ensures that z is in the interval (x, y). In the case of part negative and part non-negative interval, z_1 should either be in the interval $(x_1, 255]$ or in the interval $[0, y_1)$ for z to be reported.

We chose the two's complement representation because of its prevalence over other representations. However, the observations made above, i.e. the segregation of the non-positive and non-negative numbers into two halves and the maintenance of the relative ordering between two non-positive (or non-negative) numbers, also hold true for the *one's complement* representation. Hence, our labeling technique can be adopted with trivial modifications to handle this representation as well. Again, for the sake of brevity, we exclude the details out of this paper.

The relative ordering between non-positive (and non-negative) numbers is not maintained in another frequently used representation, namely *sign magnitude*. However, the technique outlined in the next section to handle IEEE floating-point numbers may be used without any modifications to handle the numbers expressed in the sign magnitude representation.

3.1.3. Floating-Point Numbers

IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008) [27] describes the binary representation of fractional numbers. In this representation, the first bit is the sign bit, followed by a standard-defined number of *exponent* and *trailing significand* bits. This representation of floating-point numbers can be interpreted as *sign magnitude* representation of signed integers for the purpose of comparison [28]. In the sign magnitude representation, the first bit (sign bit) is 1 for negative numbers and 0 for positive numbers. The other bits of the number determine the magnitude of the number. As in the two's complement representation, the positive numbers are represented lexicographically in the lower half of the range, and the negative numbers are shifted to the upper half. Unlike the two's complement representation, the ordering of the negative numbers is reversed.

Algorithm 3 labels the STEs for floating-point intervals as follows. If both x and y are non-negative, i.e. their sign bits are 0, then the labeling is identical to unsigned integers. On the other hand, if the sign bit is 1 for both, i.e. the interval is fully negative, then x and y are interchanged since the lexicographic ordering of negative numbers is reversed. The STEs of the comparator are again labeled using the procedure for labeling unsigned integers. However, if the upper bound and the lower bound have different sign bits, then the interval

Algorithm 3 Labeling STEs for floating-point intervals

Input:

- IEEE standard binary representation of floating-point numbers x and y. x_j and y_j denote the j^{th} MSB of x and y.
- Number of bytes in the representation of x and y, b.
- Labels of all the STEs in the b-byte comparator, L. L_t denotes the label of the STE tagged $S < t >$.

Ensure:

- $x \leq y$

```
1:  procedure LabelFloats(x, y, b, L)
2:      L_3 ← {*}
3:      if (x_1 ≤ 127 and y_1 ≤ 127) then
4:          LabelUnsigned(x, y, b, L)
5:      else if (x_1 > 127 and y_1 > 127) then
            // If x and y are both negative, interchange x and y.
6:          LabelUnsigned(y, x, b, L)
7:      else
            // If x and y have different signs, use two intervals.
8:          L' ← duplicate(L)
9:          LabelFloats(x, −0, b, L)
10:         LabelFloats(+0, y, b, L')
11:     end if
12: end procedure
```

$[x, y]$ is broken into two intervals: $[x, -0]$ and $[+0, y]$, where -0 and $+0$ denote the two representations of zero with sign bit set to 1 and 0, respectively. The two intervals can then be programmed as already discussed and z is reported to overlap the complete interval if it stabs any of the two partial intervals.

3.1.4. Reducing Output Frequency

For a query point, the automata derived from the macro designs shown in Fig. 2 can generate output on different clock-cycles for different intervals. Therefore, when a query data-stream is checked against multiple intervals programmed on the processor, a report can potentially be generated in every cycle after the first. As discussed in Section 2.2.3, this can lead to stalls, thus lowering the overall processing rate. We now present an optimized automata design which ensures that the output from all the automata for a query point is generated in the same clock-cycle. This reduces the output generation frequency to a maximum of once every b cycles when comparing b-byte numbers.

The optimized version of the 4-byte macro in Fig. 2a is shown in Fig. 3. As described in Section 3.1.1, in the original automaton design, $S3$ is activated to generate an output in the subsequent cycle after a query point is determined to overlap an interval. However, for ensuring that the match is reported exactly

after processing the fourth byte, we added two STEs in the central column and labeled them to match all the 1-byte symbols. These STEs process the remaining bytes of the query number, after it is determined to overlap the interval, and activate $S3$ for generating a report after processing the last byte. Further, in the original design, any matches determined by the STEs labeled $S14$ or $S15$ are reported by $S3$ while processing the first byte of the next query in the stream. For handling this case, we made $S14$ and $S15$ reporting-STEs, and removed the connections from these two STEs to $S3$. The 8-byte macro can be optimized in a similar fashion by adding six STEs in the central column.

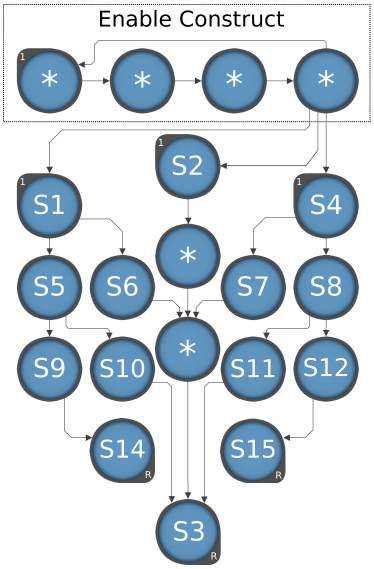


Figure 3: Modified 4-byte macro for reducing output generation frequency

3.1.5. Non-numeric Fixed-Length Formats

The automata designs described earlier in this section compare points denoted by numbers represented as 4-byte or 8-byte binary strings. The alphabet of these strings contains all the symbols in the 1-byte symbol set. These designs can also be used for comparing strings of any fixed-length defined over any alphabet, as long as the symbols in the alphabet can be lexicographically ordered. The labeling algorithm can be modified accordingly for this purpose. Such string based intervals are used to index into entries of large databases, such as those used in the fields of bioinformatics and biometrics.

3.2. Handling Variable-length Formats

Until now, we have only discussed fixed-width formats. In these formats, the representation of the endpoints and the query points uses identical number of bytes. This enables the design of pre-compilable automata structures of fixed shape for all the intervals, irrespective of the values of x and y. However, this is not possible in the case of the numeric-string format, where the representation of different numbers is of variable length. Instead, we have developed a method of defining these automata using instances of four constituent automata substructures of fixed shapes. These are detailed below.

3.2.1. Overview

The numeric-string for a number consists of the ASCII-equivalent character of its digits concatenated to form a string, ordered from left to right. If the number is positive, the digits are preceded by an optional '+' sign. Otherwise, they are preceded by a '−' sign. In case of a real number, the integer and the fractional part are separated by a '.' (decimal point). However, even when the

decimal point is present, the integer part or the fractional part may be missing, but not both.

Similar to the automata designs described in Section 3.1, the STEs are arranged in rows, ordered from top to bottom. The topmost row is used to match the optional sign character in the beginning of the numeric-string of z. The next row checks the first integer digit of z against that of x and y. The second digit is checked by the next row, and so on. An additional row to parse a decimal-point or the end of the numeric-string of z is inserted after the rows for checking the integer digits. Next, rows are inserted to handle the fractional digits in the endpoints, one row per fractional digit. Therefore, the overall number of rows is dependent on the number of integer and fractional digits in x and y. For brevity, we have adopted the following notations in the rest of this paper. For any number a, a_{in} denotes the integer part and a_{fr} denotes the fractional part. The number of digits in the integer part and the fractional part of a are denoted as $d(a_{in})$ and $d(a_{fr})$, respectively.

The checks for the integer and the fractional part of the numbers are different from each other and are therefore handled separately. Since the rows for handling the integer digits are identical to one another, they can be defined as the instances of a macro. However, owing to the architecture and compiler characteristics of the AP, it is sometimes more optimal to combine two such rows into one macro. The topmost two rows are implemented as an instance of the *Leading Integer Digit* macro, whereas every two subsequent rows for checking integer digits are implemented as an instance of the *Subsequent Integer Digits* macro. Similarly, the rows pertaining to the handling of the decimal point and the first fractional digit are implemented as one instance of the *Leading Fractional Digit* macro, whereas each subsequent fractional digit is handled using one instance of the *Subsequent Fractional Digit* macro.

Notice that, the number of integer and fractional digits in the numeric-strings of one of the endpoints may be greater than the other. For example, without loss of generality, if $d(x_{in}) < d(y_{in})$, then $d(y_{in})$ rows are required to complete the checks against all the integer digits of y. While STEs from the top $d(x_{in})$ rows are used to complete the checks with respect to x, and the remaining rows are left unlabeled and unused. We now describe the four constituent macros in detail and illustrate their use in a complete automaton through an example.

3.2.2. Handling the Integer Part

As described above, the number of rows required to parse the integer part equals $\max(d(x_{in}), d(y_{in}))+1$. The top two rows are implemented as an instance of the *Leading Integer Digit* macro shown in Fig. 4. If x or y have more than one integer digit, then the subsequent rows are implemented using instances of the *Subsequent Integer Digits* macro shown in Fig. 5. Since each instance covers two rows, $\lceil (\max(d(x_{in}), d(y_{in})) - 1)/2 \rceil$ such instances need to be created. Notice that, if the total number of rows required to parse the integer digits is odd, then only one of the rows in the last instance of the Subsequent Integer Digits macro is utilized.

The input ports of the Leading Integer Digit macro are connected to a *start-of-processing* logic (not shown in the figure). This logic identifies the beginning of the numeric-string of z in the data-stream. It typically entails the identification of a special delimiter which appears before the numeric-strings in the data-stream. Since the number of integer digits in the query point z is not known a priori, three independent possibilities have to be explored in parallel. 1) $d(z_{in}) = d(x_{in})$, 2) $d(z_{in}) = d(y_{in})$, and 3) $d(x_{in}) \neq d(z_{in}) \neq d(y_{in})$.

The first possibility that $d(z_{in}) = d(x_{in})$, along with the condition that $z \geq x$, is explored by the two leftmost columns in the macro. If x is positive, then the input port marked as *in_x_p* is used. Note that this port allows the numeric-string of z to have an optional leading '+' sign. The output port *out_x_eq* is activated only if the first digit of z equals x. Otherwise, the output port *out_x_gt* is activated only if the first integer digit of z is greater than the first integer digit of x. If x is negative, then only the input port marked as *in_x_n* is used. This port mandates a '$-$' sign be present at the front of the numeric-string of z. The output ports *out_x_eq* and *out_x_gt* are activated only if the first digit of z is equal to the first digit of x or less than that, respectively.

Similarly, the second possibility that $d(z_{in}) = d(y_{in})$ along with the condition $z \leq y$ is explored by the two rightmost columns in the macro. The connections and the labeling logic are very similar to those explained above. The output port *out_y_eq* is activated only if the first digit of z equals y. The output port *out_y_lt* is activated if the first integer digit of z is less than the first integer digit of y, when y is positive, or if the first integer digit of z is greater than the first integer digit of y, when y is negative.

Finally, the last possibility of $d(x_{in}) \neq d(z_{in}) \neq d(y_{in})$ is handled by the STEs in the middle which are part of the *bypass* logic. This logic accepts z based on the number of digits in its integer part. If both x and y are positive, then only the input port *in_pt_p* is utilized, and z is accepted if and only if $z > 0$ and $d(x_{in}) < d(z_{in}) < d(y_{in})$. If both x and y are negative, then only input port *in_pt_n* is utilized, and z is accepted if and only if $z < 0$ and $d(x_{in}) > d(z_{in}) > d(y_{in})$. Finally, if x is negative and y is positive, then both the input ports *in_pt_p* and *in_pt_n* are utilized, and z is accepted if and only if

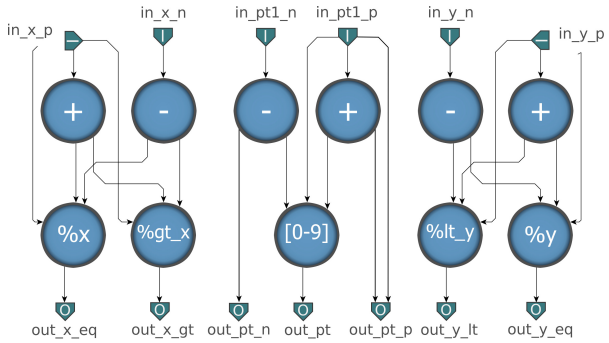


Figure 4: *Leading Integer Digit* macro

$z < 0$ and $d(x_{in}) > d(z_{in})$ or $z > 0$ and $d(y_{in}) > d(z_{in})$. Since these conditions cannot occur when $d(x_{in}) \leq 1$ and $d(y_{in}) \leq 1$, i.e. the automaton has only two rows for handling the integer digits, we defer the discussion of its working to the description of the Subsequent Integer Digits macro.

Instances of the Subsequent Integer Digits macro are chained to one another, with the output ports of one instance connected to the input ports of the next. The input ports of the first instance are driven by the output ports of the instance of the Leading Integer Digit macro; and the output ports of the last instance drive the input ports of the Leading Fractional Digit macro.

The layout of the Subsequent Integer Digits macro is similar to that of the other macro structures described until now. The leftmost three columns correspond to the checks against the digits of the endpoint x, and the rightmost three correspond to the checks against the digits of the endpoint y. Based on the number of remaining integer digits in an endpoint (say x), one or both of the rows in the macro instance may be skipped. This is accomplished by connecting the output port *out_x_eq* of the macro instance placed above the input port *in_x1* (in the case of a one row requirement), and *in_x2* (in the case of a two row requirement) of the instance placed below. If none of the rows of this instance are to be utilized for x, then neither the input port *in_x1* nor the port *in_x2* is used. Instead, the output port *out_x_eq* of the macro instance placed above is directly connected to the instance of the Leading Fractional Digit macro.

We now turn our focus to the central column which is part of the bypass logic. For z to lie between x and y, the number of digits in z must lie within a specific range. In the previous section, we have already explained how the lower and upper bounds of this range can be determined based on $d(x_{in})$, $d(y_{in})$, and the signs of x and y. The bypass logic checks that the $d(z_{in})$ lies within the lower and upper bounds by ensuring that a minimum (lower bound) and maximum (upper bound) number of rows are traversed while processing the integer part of z.

For an instance of the Subsequent Integer Digits macro, if one more row has to be traversed to reach the lower bound, then only the input port *in_pt1* is used, otherwise only *in_pt2* is used. Once the minimum number of rows has

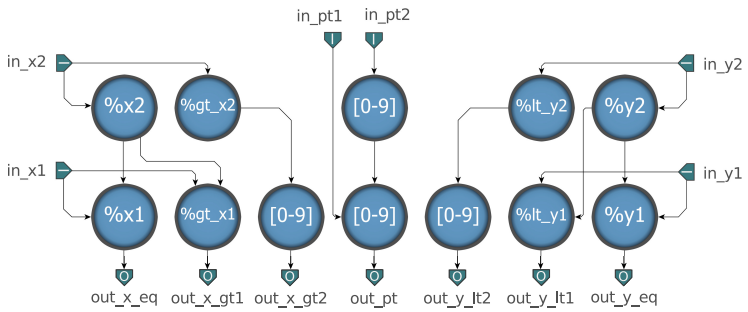


Figure 5: *Subsequent Integer Digits* macro

been traversed, the output port *out_pt* of the bypass logic of the macro instance is connected directly to the input ports of the Leading Fractional Digit macro. It is also connected to both the input ports *in_pt1* and *in_pt2* of the next macro instance. This allows the skipping of one or both rows of the next instance. This pattern of connections is continued till the upper bound of rows is reached ensuring that any number of rows between the lower and upper bound can be skipped. Once the upper bound is reached, the output port *out_pt* of the macro instance where this occurs is connected only to the input ports of the Leading Fractional Digit macro. This disallows any more integer digits in z.

3.2.3. Handling the Fractional Part

There are three distinctions when comparing fractional digits versus the comparison of integer digits: 1) The number of fractional digits does not play a role in determining the magnitude of the fractional part. Therefore, as soon as a place value of z evaluates to greater-than x or less-than y, the following fractional digits of z can not change the result of the evaluation. 2) When x is negative and all of its digits match those of z, down to the last fractional digit of x, then all the following fractional digits of z must be zeros to satisfy the condition $z \geq x$. Similarly, when y is positive and all of its digits match those of z, all the successive digits in z must be zeros for $z \leq y$. 3) Even when x and y are integers, we allow for z to have fractional digits. These paths are available when the integer part of z is greater-than or equal to the integer part of x, or is strictly less-than that of y.

Similar to the methodology for handling the integer part described in the previous section, an instance of the *Leading Fractional Digit* macro (shown in Fig. 6) is used to handle the first two rows of the fractional part. The first row parses the decimal point and the second row is used to check the first

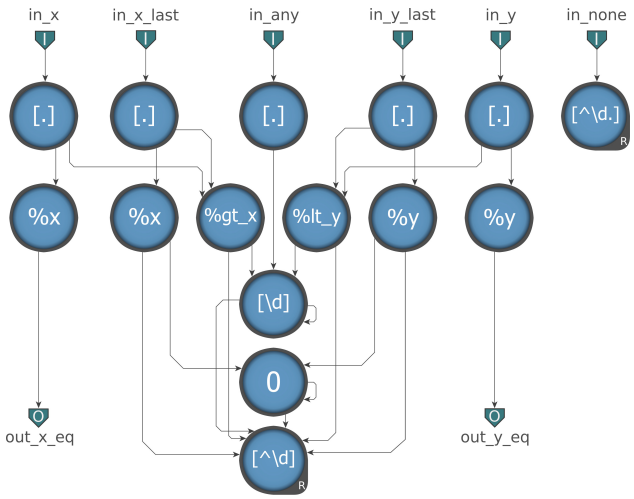


Figure 6: *Leading Fractional Digit* macro

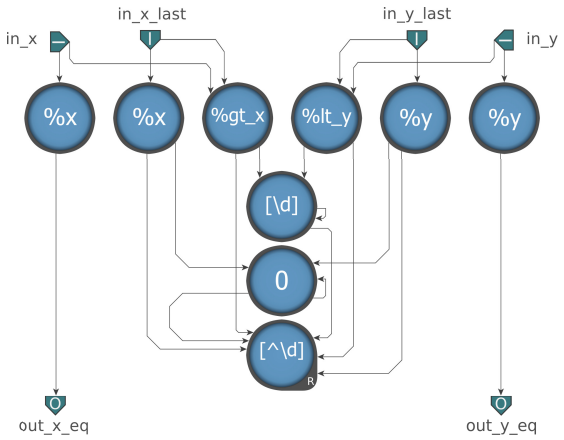


Figure 7: *Subsequent Fractional Digit* macro

fractional digit. Every subsequent row is implemented as an instance of the *Subsequent Fractional Digit* macro shown in Fig. 7. In total, the automaton contains $\max(d(x_{fr}), d(y_{fr})) + 1$ rows to handle the fractional part.

If a determination can be made based on the integer parts of x, y, and z, then the fractional part of z need not be checked. The control is shifted to the bypass logic by means of the *in_any* input port which ignores all the fractional digits, and generates an output after encountering the first non-digit character at the end of the numeric-string of z. The fractional part of z has to be checked if the integer part of z equals x, y, or both. We discuss the case when $z_{in} = x_{in}$ first. If, in any row, the fractional digit of z is determined to be greater than the corresponding fractional digit of x, then the evaluation is shifted to the bypass logic. Otherwise, the next digit of z is compared against the next digit of x in the next row. The same concept is used for comparison with the fractional digits of y, except that the direction of inequality is reversed. When x and y have the same number of digits and also have the same sign, the outputs must be AND-ed together since they will only be active on one symbol cycle and must both be active to satisfy the expression.

3.2.4. Illustrative Example

In this section, we will illustrate the working of our automata designs for handling numeric-strings using an example interval $[x, y] = [-47.82, 361.5]$. Here, $d(x_{in}) = 2$, $d(x_{fr}) = 2$, $d(y_{in}) = 3$, and $d(y_{fr}) = 1$. As discussed in Sec. 3.2.2, one Leading Integer Digit macro is required for handling the sign and the first digit of the integer part while the number of Subsequent Integer Digits macros required for this interval is calculated as $\lceil (\max(d(x_{in}), d(y_{in})) - 1)/2 \rceil = \lceil (\max(2, 3) - 1)/2 \rceil = 1$. Similarly, as discussed in Sec. 3.2.3, one Leading Fractional Digit macro is required for handling the decimal sign and the first fractional digit. The number of Subsequent Fractional Digit macro instances required is $\max(d(x_{fr}), d(y_{fr})) - 1 = \max(2, 1) - 1 = 1$. The resulting automa-

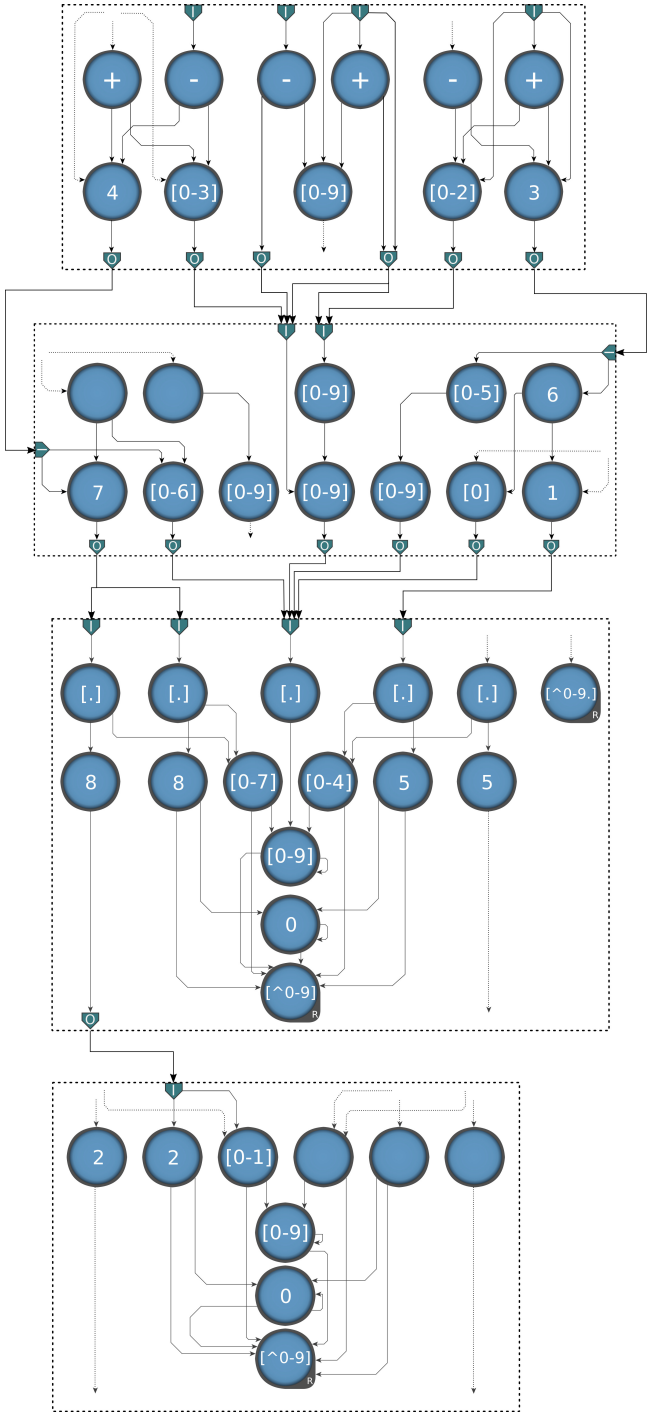


Figure 8: Automaton for reporting all the numeric-strings in the interval $[-47.82, 361.5]$

ton to report a query point z that lies in the interval $[-47.82, 361.5]$ is shown in Fig. 8. The figure shows all the macro instances (depicted as dotted boxes), their labeled STEs, and the communication lines. The unused ports have been faded out and all their incoming and outgoing lines are shown as dotted, indicating that they are present in the instance but are not being used.

The top two boxes show one instance each of the Leading Integer Digit and the Subsequent Integer Digits macros, in that order from top to bottom, for handling the integer part. We discuss the Leading Integer Digit macro first. Since x is negative and y is positive, the input ports *in_x_n* and *in_y_p* are connected to the start-of-processing logic for matching negative numbers using the leftmost two columns and positive numbers using the rightmost two columns. For any negative z with two integer digits to lie in the interval, the first integer digit of z should either be equal to the first integer digit of x, i.e. 4, or be less than that. The STEs in the two leftmost columns of the bottom row of the macro are labeled to match these two cases. Similarly, the STEs in the two rightmost columns are labeled to match the corresponding cases for matching the first integer digit of positive z and the first integer digit of y. The input ports *in_pt1_n* and *in_pt1_p* of the macro are also connected to the start-of-processing logic for matching z using the bypass logic.

Since $d(x_{in})$ is 2, only the bottom row of the Subsequent Integer Digits macro is used and the output from the port *out_x_eq* of the Leading Integer Digit macro is connected to the input port *in_x1*. The two leftmost STEs in the bottom row are labeled to determine if z lies in the interval based on the second digit of x. On the other hand, since $d(y_{in})$ is 3, the output of the port *out_y_eq* is connected to the input port *in_y2* and the rightmost two STEs in the top row and the bottom row of the macro are labeled to handle the cases where z is determined to lie in the interval based on the second and the third digit of y, respectively. Further, the output from the port *out_pt_n* of the Leading Integer Digit macro are connected to the port *in_pt1* for matching negative z with a single integer digit while the output from the port *out_pt_p* is connected to the ports *in_pt1* and *in_pt2* for matching positive z with one and two integer digits, respectively.

The fractional part is handled by the Leading Fractional Digit and the Subsequent Fractional Digit macros shown in the bottom two boxes. If z has been determined to lie in the interval based on the comparison of the integer digits, then the comparison of the fractional digits is not required. Therefore, the output from the ports *out_x_gt1*, *out_y_lt2*, *out_y_lt1*, and *out_pt* from the Subsequent Integer Digits macro is connected to the input port *in_any* of the Leading Fractional Digit macro for reporting on encountering any fractional digits. However, if the integer part of z has been found to exactly match the integer part of x, then the fractional part of the numbers is compared by connecting the port *out_x_eq* of the Subsequent Integer Digit macro to the ports *in_x* and *in_x_last* of the Leading Fractional Digit macro. If the first fractional digit of z also matches the first fractional digit of x, then the check for the trailing digit of z is done by the Subsequent Fractional Digit macro. This is enabled by connecting the port *out_x_eq* of the Leading Fractional Digit macro to the port *in_x* of the

Subsequent Fractional Digit macro. Since the fractional part of y has only a single digit, the output of the port *out_y_eq* of the Subsequent Integer Digits macro is connected to the *in_y_last* of the Leading Fractional Digit macro.

3.3. Analysis

When the numbers are expressed in the integer or the IEEE floating-point format, creating and loading the automata for n intervals requires a preprocessing time of $O(n)$ and a space complexity of $O(n)$. The addition or deletion of an interval to the set can be accomplished with a constant overhead, a significant advantage over contemporary methods. For each query, the presence of overlapping intervals can be ascertained in $O(1)$ time, but identifying all of them requires $O(k)$ time, where k is the number of intervals stabbed by the query.

In the case of the numeric-string representation, the complexity of adding an interval is not constant, but depends on the number of digits in the endpoints. If the average number of digits in all the endpoints in the set is w, then the time as well as space complexity of creating and loading all the required automata is $O(wn)$. The output handling complexity remains $O(k)$, the same as discussed above.

Within the limits of the AP-board, the intervals, in the form of their corresponding automata, can be stored in any arbitrary order. If the number of intervals is larger than what can be fit inside an AP-board, then they can be partitioned into buckets and handled iteratively. The choice of which bucket an interval is placed in can be made using one of the endpoints. The ordering of the intervals within a bucket is arbitrary, and irrelevant to the determination of the query time or the space complexity.

In spite of the theoretical advantages mentioned above, programming a board incurs substantial latency which cannot be amortized through the performance gained using a single query. Therefore, the methodology described here best serves applications where a large number of stabbing queries are serviced with high throughput.

4. Results

The AP is in advanced stages of qualification and production. The authors have access to pre-production prototypes which were used to evaluate the performance reported in this paper. To the best of our knowledge, these are the first hardware results to be published for the processor.

We used the Python API provided with the AP-SDK for creating the macros, combining them into ANML-NFA networks, compilation of ANML-NFA, and substitution of the labels. The designs were validated using the simulator provided with the AP-SDK.

Comparing the AP implementation of the interval stabbing problem to the state-of-the-art CPU-based implementations is unfair because of the following reasons. We are using AP-board hardware and software which are in *bring-up* phase. There are known overheads which slow down the processing by a

factor of 20 times or more than that of the production hardware and software. The current generation of the chip itself is fabricated using *memory process* technology which is a couple of generations behind the state-of-the-art. The next generation of the chip, currently in design, is supposed to be orders of magnitude faster than the current chip. Finally, using the interval stabbing problem as a generic representation of a class of problems requiring comparison of a query element against an unordered list of other elements is not accurate. As discussed in Section 2.1, the list in the interval stabbing problem can be partially ordered using *interval trees*, *segment trees*, etc. The run-time of algorithms using these structures are output-sensitive in $O(log\, n + k)$ time. When k is small, CPU-based algorithms run faster, but as k grows, AP-based solutions become more attractive. This is not the case with all the problems.

Instead, we developed a simple metric similar to the widely adopted FLoating-point Operations Per Second (FLOPS) metric. We simply calculate the number of FLOating-point Comparisons per Second (FLOCS). This can be calculated using the following formula:

$$\texttt{FLOCS} = \texttt{c} \times \frac{\#\texttt{automata per board} \times \texttt{f}}{\#\texttt{cycles per query}}$$

Here, `c` is the number of comparisons per automaton and `f` is the operating frequency. In our case, `c` equals 2, because each automaton compares the query point against the limits of the interval, and `f` is 128 MHz. Using this formula, Table 1 describes the FLOCS for the 4-byte and the 8-byte comparator macros. In the best case, where the output handling does not become a bottleneck, the number of cycles per query are 4 and 8, respectively. However, this may degrade to 494 cycles for both the automata due to the output handling bottleneck in the worst case. The observed FLOCS on the prototype hardware has also been reported. This is in line with the expected behavior as the software and

	#Automata per board	#Cycles per query	giga-FLOCS
Maximum theoretical throughput			
4-byte macro	43008	4	2752.5
8-byte macro	24576	8	786.4
Minimum theoretical throughput (output regulated)			
4-byte macro	43008	494	22.3
8-byte macro	24576	494	12.7
Observed throughput: on prototype hardware			
4-byte macro	43008	-N/A-	0.95
8-byte macro	24576	-N/A-	0.55

Table 1: Theoretical and observed FLOCS for the 4-byte and 8-byte macros

hardware are currently under testing and validation phase. Multiple stages are yet to be pipelined, and some are supposed to be parallelized. Some of the resources are yet to be programmable, and the system software is to be improved with respect to output handling.

5. Future Research

Handling Points in Higher Dimensions. In this paper, we have only demonstrated handling stabbing queries in one-dimensional space. However, the methodology can be extended for handling points in higher dimensions. For example, in the case of two-dimensional space, the region covered by an axis-parallel rectangle can be represented by two endpoints x_1, x_2 and y_1, y_2, where $x_1 \leq y_1$ and $x_2 \leq y_2$. A query point z_1, z_2 lies within this rectangle if and only if $x_1 \leq z_1 \leq y_1$ and $x_2 \leq z_2 \leq y_2$; i.e. by executing two interval queries in series, and declaring a positive result only if both the queries are satisfied. In fact, generating a unified automaton to do so by concatenating two instances of the automaton described above, one for each dimension, is trivial. However, handling more generalized cases, e.g. rectangles without axis-parallel edges, any 2-dimensional polygon, any n-dimensional hyper-rectangles, and finally any n-dimensional polygon is a subject of ongoing research.

Handling Multiple Formats. In some applications, the data may be represented using multiple formats. For example, a datestamp could be represented as *mm/dd/yy*, *mm/dd/yyyy*, *mm-dd-yyyy*, etc. We are currently investigating automata structures which allow the representation of points in one of the many permissible formats. In fact, the performance benefits of using this architecture is amplified in such cases as the conversion of the points to a specific format can be avoided.

6. Conclusion

In this paper, we have provided a streaming solution to the interval stabbing problem using the Automata Processor. To the best of our knowledge, this is the first use of the processor to execute numerical comparisons on multi-byte integers and single and double precision IEEE floating-point numbers. Our implementation is based on defining one automaton for every interval in the list, and executing tens of thousands of such automata in parallel using the resources of a single AP-board. Not only does this provide significant performance benefits in answering the stabbing queries, but also leaves the interval list unordered. This allows expeditious modifications to the list, which is required by many applications. The same ideas are extended to handle variable-length numeric-string representation in this paper; and can be further extended for handing other datatypes such as generic strings, multidimensional coordinates, datestamps, timestamps, etc. The presented automata designs are modular and extendable to larger composite automata. These designs exemplify techniques

to overcome significant resource and performance bottlenecks which may be useful to prospective application designers working on this processor.

Acknowledgments

We acknowledge funding from NSF Exploratory Grant CCF-1448333 and Micron Technology, Inc.

References

[1] The Micron Automata Processor Developer Portal, `http://www.micronautomata.com/`, accessed: Sept, 2016.

[2] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, H. Noyes, An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing, IEEE Transactions on Parallel and Distributed Systems 99 (2014) 1, ISSN 1045-9219.

[3] I. Roy, A. Srivastava, M. Nourian, M. Becchi, A. Srinivas, High Performance Pattern Matching Using the Automata Processor, in: IEEE 30th International Parallel and Distributed Processing Symposium, 1123–1132, 2016.

[4] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, K. Skadron, Brill Tagging on the Micron Automata Processor, in: IEEE 9th International Conference on Semantic Computing (ICSC), 236–239, 2015.

[5] I. Roy, S. Aluru, Finding Motifs in Biological Sequences Using the Micron Automata Processor, in: IEEE 28th International on Parallel and Distributed Processing Symposium, 415–424, 2014.

[6] K. Wang, E. Sadredini, K. Skadron, Sequential Pattern Mining with the Micron Automata Processor, in: Proceedings of the ACM International Conference on Computing Frontiers, CF '16, ISBN 978-1-4503-4128-8, 135–144, 2016.

[7] I. Roy, N. Jammula, S. Aluru, Algorithmic Techniques for Solving Graph Problems on the Automata Processor, in: IEEE 30th International Parallel and Distributed Processing Symposium, ISSN 1530-2075, 283–292, 2016.

[8] H. Edelsbrunner, Dynamic data structures for orthogonal intersection queries, Technische Universität Graz/Forschungszentrum Graz. Institut für Informationsverarbeitung, 1980.

[9] H. Edelsbrunner, A new approach to rectangle intersections, Part I, International Journal of Computer Mathematics 13 (3-4) (1983) 209–219.

[10] H. Edelsbrunner, A new approach to rectangle intersections, Part II, International Journal of Computer Mathematics 13 (3-4) (1983) 221–229.

[11] E. M. McCreight, Efficient algorithms for enumerating intersecting intervals and rectangles, Tech. Rep., 1980.

[12] J. L. Bentley, Solutions to Klee's rectangle problems, Unpublished manuscript (1977) 282–300.

[13] J. M. Schmidt, Interval stabbing problems in small integer ranges, in: Algorithms and Computation, Springer, 163–172, 2009.

[14] T. H. Cormen, Introduction to algorithms, MIT press, 2009.

[15] E. M. McCreight, Priority search trees, SIAM Journal on Computing 14 (2) (1985) 257–276.

[16] E. N. Hanson, The interval skip list: A data structure for finding all intervals that overlap a point, in: Algorithms and Data Structures, Springer, 153–164, 1991.

[17] S. Alstrup, G. S. Brodal, T. Rauhe, New data structures for orthogonal range searching, in: Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on, IEEE, 198–207, 2000.

[18] H. N. Gabow, J. L. Bentley, R. E. Tarjan, Scaling and related techniques for geometry problems, in: Proceedings of the sixteenth annual ACM symposium on Theory of computing, ACM, 135–143, 1984.

[19] P. Chovanec, M. Krátkỳ, Processing of Multidimensional Range Query Using SIMD Instructions, Informatics Engineering and Information Science (2011) 223–237.

[20] P. Bednář, P. Gajdoš, M. Krátkỳ, P. Chovanec, Processing of Range Query Using SIMD and GPU, New Trends in Databases and Information Systems (2013) 13–25.

[21] J. Kim, S.-G. Kim, B. Nam, Parallel multi-dimensional range query processing with R-trees on GPU, Journal of Parallel and Distributed Computing 73 (8) (2013) 1195–1207.

[22] M. K. Maramreddy, K. Kothapalli, GPU Accelerated Range Trees with Applications, in: European Conference on Parallel Processing, Springer, 740–751, 2014.

[23] I. Roy, A. Srivastava, S. Aluru, Programming Techniques for the Automata Processor, in: 45th International Conference on Parallel Processing, 205–210, 2016.

[24] K. Wang, Y. Qi, J. J. Fox, M. R. Stan, K. Skadron, Association rule mining with the micron automata processor, in: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, IEEE, 689–699, 2015.

[25] T. Tracy, Y. Fu, I. Roy, E. Jonas, P. Glendenning, Towards Machine Learning on the Automata Processor, in: Proceedings of the 31st International Conference on High Performance Computing (ISC), Springer International Publishing, 200–218, 2016.

[26] J. Greene, V. Roychowdhury, S. Kaptanoglu, A. E. Gamal, Segmented channel routing, in: Proceedings of the 27th ACM/IEEE Design Automation Conference, ACM, 567–572, 1991.

[27] I. S. Committee, et al., 754-2008 IEEE standard for floating-point arithmetic, IEEE Computer Society Std 2008.

[28] B. Dawson, Comparing floating point numbers, `http://www.cygnus-software.com/papers/comparingfloats/Comparing%20floating%20point%20numbers.htm`, accessed: Sept, 2017.