Programming Techniques for the Automata Processor

Indranil Roy, Ankit Srivastava, and Srinivas Aluru
Email:iroy@gatech.edu, asrivast@gatech.edu, and aluru@cc.gatech.edu
School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA

***Abstract*—The Micron Automata Processor (AP) is a novel co-processor accelerator that supports the parallel execution of multiple Nondeterministic Finite Automata (NFA) programmed directly into hardware over a single data-stream. In this paper, we present a number of programming techniques to develop automata that execute efficiently on this processor. First, we present general techniques to transform NFAs defined in their classical representation to the representation used by the AP, and optimize the same. Then, we present automata development techniques using simple but powerful generic building blocks. All the above techniques are generic in nature and can be useful to application developers working on this new upcoming co-processor architecture.**

1. Introduction

The study of *finite automata* (or *state machines*) has received considerable attention in literature. However, easy implementation of automata, especially Nondeterministic Finite Automata (NFA), which is both compact in representation and expeditious in execution remains a challenge to this day [1]. On general-purpose processors like CPUs and GPUs, the execution of NFAs can only be simulated, which takes considerable time. On the other hand, although hardware implementation of NFA using reconfigurable Field Programmable Gate Arrays (FPGAs) is both possible and fast, two primary challenges persist. First, programming FPGAs is often non-trivial. A programmer needs in-depth knowledge of the underlying architecture and experience in logic-design to use these processors correctly and efficiently. Second, for some problems, such as network intrusion detection, the capacity of even the largest FPGAs is not enough to accommodate all the required automata at any given time [2], [3], [4]. This leads to higher complexity and runtime.

The Micron Automata Processor (AP) [1] is a new co-processor accelerator which supports the execution of multiple NFA over a single data-stream. As this processor is specifically designed for implementing NFA in hardware, it overcomes the challenges outlined above for implementing the same on general-purpose FPGAs. Firstly, its programmable elements closely resemble the *states* and *transitions* in the classical representation of an NFA. This makes it easier to program for NFA designers. Secondly, its capacity and speed of execution is superior to the largest of FPGAs [1]. Therefore, the use of this processor to accelerate various pattern-matching applications follows naturally.

In this paper, we present several programming techniques that we designed and discovered to be useful in solving problems using the AP. These techniques promote modular automata designs which require minimal on-board resources and maximize performance by taking into consideration the various auxiliary capabilities of the processing elements, routing limitations of the architecture, and configuration and run-time overheads.

2. Automata Processor Basics

The AP is based on the Multiple Instruction Single Data (MISD) paradigm, where all the programmable elements process a single data-stream, called a *data-flow*. In each cycle, an 8-bit *symbol* from the data-flow is broadcast to all the programmable elements, and the processing of this symbol by all the elements is carried out in parallel.

2.1. Programmable elements

The primary programmable element in the AP is a *State Transition Element* (STE). The label of an STE can be set to any character-set from the 8-bit symbol-set. If *active* in a cycle, the STE checks if the symbol broadcast in that cycle matches its label and, if so, the elements connected to its outgoing *routing lines* are activated for the next symbol-cycle. These routing lines are part of a programmable *interconnection network*.

Apart from the STEs, the AP contains *Counter* and *Boolean* elements which allow additional expressive capability of the automata programmable into the AP, beyond traditional NFAs. A Counter element is used to count the number of occurrences of a particular pattern or sub-pattern, and activate elements connected to its outgoing lines on reaching a programmatically set target count. The Boolean elements are used to emulate *AND*, *OR*, *NOT*, *NAND*, *NOR*, *sum-of-product*, and *product-of-sum* operations. The output of the element is activated if the incoming lines are triggered such that the combinational logic of the element is satisfied.

2.2. Programmable resources

The STEs on the AP-chip are hierarchically organized into *half-cores*, *blocks*, and *rows*. Each chip contains two half-cores, each half-core contains 96 blocks, each block contains 16 rows, and each row contains 16 STEs. In a

block, 4 of the rows contain a Counter element each, and the remaining 12 contain a Boolean element each. Cumulatively, a single AP-chip contains $49,152$ STEs, 768 Counter elements, and $2,304$ Boolean elements.

The physical routing capability decreases as one goes higher up the hierarchy. Each element in a row can be connected to every other element in that row simultaneously using *row-routing* (RR) lines. However, only 24 *block-routing* (BR) lines are present to connect elements from different rows in a block. The connectivity between elements from different blocks is rarer still. There are no physical connections between elements from the two half-cores of a chip. However, this underlying routing structure is abstracted by the compiler, presenting a flat structure to the user.

A single AP-board consists of 32 AP-chips which connect to the host processor using the *PCIExpress* interconnect. The AP-chips are arranged in 4 *ranks* of 8 chips each. The chips on a rank can be grouped into logical cores of 2, 4, or 8 chips. Chips within a single logical core process the same data-flow, and all the logical cores can process independent data-flows in parallel.

2.3. Processing throughput

Once the automata have been programmed into the AP-chip(s), the processing rate is defined only by the input and output handling rate. There are no hidden bottlenecks or non-determinism in the run-time. Each AP-chip operates at 128 MHz, processing one input symbol per cycle (henceforth called *symbol-cycle*) giving rise to an input processing rate of 1 Gbps per chip/logical-core.

From the perspective of output handling, the chip is divided into 6 *output-regions*. During a cycle, if the processing elements within a region produce an output, an *output-vector* is generated. This fixed-length vector identifies the element(s) which produced the output and the current offset in the data-flow. The output-handler tries to store this vector in an *output-buffer*. If the necessary space is available, the operation is completed within the current cycle, and the input processing continues unabated from the next cycle. Asynchronously, the output-vector is transmitted to the host application so that it can be processed. However, if the required buffer-space for the transfer is unavailable, the entire processing pipeline is stalled till the buffer can be sufficiently emptied. This takes 40 symbol cycles per output-vector. Therefore, handling a single output-event may require up to 240 cycles, if all six output-regions are involved. Additionally, a 16 cycle initial transfer latency may be incurred per output event.

3. Automata Conversion and Optimization

NFAs to be executed on the AP are defined using a proprietary language, called *Automata Network Markup Language* (ANML, pronounced as "animal"). Henceforth, in this paper, these NFAs are referred to as *ANML-NFA*s. Since automata designers are used to defining NFAs as *state-diagrams*, we propose techniques to convert the same into equivalent ANML-NFAs. We further extend these techniques to handle ϵ-transitions. Finally, we propose methods to optimize ANML-NFAs.

3.1. Converting classical NFA to ANML-NFA

Classically, an NFA is defined using a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

Σ is the input alphabet,

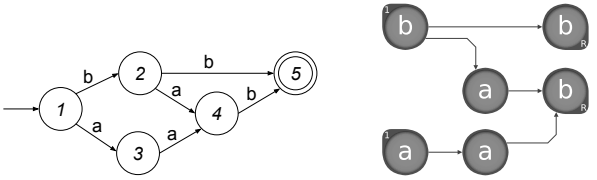
S is a finite, non-empty set of states,

s_0 is the start state, $s_0 \in S$,

δ is the state-transition function, $\delta : S \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathrm{P}(S)$, where P($S$) is the power set of S,

F is the set of accept states, $F \subseteq S$.

Figure 1a shows the classical state-diagram of an NFA which accepts the strings belonging to the *language*: $\{aab, bab, bb\}$.

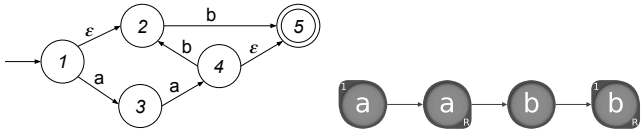


(a) State-diagram (b) ANML-NFA

Figure 1: Automaton to accept strings from the language $\{aab, bab, bb\}$ over the alphabet $\{a, b\}$.

For converting a state-diagram into its equivalent ANML-NFA, we start by defining an STE for each transition in the state-diagram and assigning it the same label. An STE is represented using a circle with the label placed inside it. Every STE representing an outgoing transition of the *start-state* is marked as a *start-STE*, shown with an indicator on the top-left corner. Similarly, STEs representing incoming transitions to *accept state*s are marked as *reporting-STE*s, shown with an indicator at the bottom-right with the symbol R placed inside it. All the states are captured using connections between the STEs. Each STE representing an incoming transition into a state is connected to all the STEs representing outgoing transitions from that state. The equivalent ANML-NFA so obtained, for the state-diagram in Figure 1a, is shown in Figure 1b.

The technique described above does not take ϵ-transitions into account. Handling ϵ-transitions out of a start-state was introduced in Dlugosch et al. [1]. We now propose a general technique to handle NFAs with ϵ-transitions, using *ϵ-closure*. The *ϵ-closure* $E(s)$ of state $s \in S$ is defined as the set of states which can be reached from s using only



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

Figure 2: Handling ϵ-transitions using ϵ-closures.

Algorithm 1 Converting state-diagrams to ANML-NFA

1: Compute $E(s_i)$, $\forall s_i \in S$
2: Create $STE(i, j, l)$ where $l \neq [\epsilon]$ and $\forall s_j = \delta(s_i, l)$
3: Connect all $STE(i, j, l)$ to $STE(p, q, l')$ $\forall s_p \in E(s_j)$
4: Mark all $STE(i, j, l)$ as start-STEs, $\forall s_i \in E(s_0)$
5: If $\exists s_k \in E(s_j)$, where $s_k \in F$, then mark all $STE(i, j, l)$ as reporting-STEs

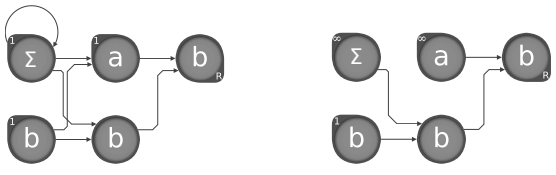
ϵ-transitions. For example, in Figure 2a, $E(1) = \{1, 2\}$, $E(2) = \{2\}$, $E(4) = \{4, 5\}$, etc. Note that $s \in E(s)$.

Our algorithm for converting a state-diagram with ϵ-transitions to an equivalent ANML-NFA is shown in Algorithm 1. Its working can be described as follows. An STE corresponding to a transition, $s_j = \delta(s_i, l)$ where l is the label of the transition, is represented as $STE(i, j, l)$. In steps $1-2$, the ϵ-closure of each state is computed and STEs corresponding to each non-ϵ-transition in the state-diagram are drawn in the ANML-NFA. In step 3, for any two states s_j and s_p, each STE corresponding to an incoming edge of s_j is connected to all the STEs representing outgoing edges of s_p, where $s_p \in E(s)$. In step 4, all the STEs corresponding to any state s_i in the state-diagram, which is in the ϵ-cover of the start-state, is marked as a start-STE. Finally, in step 5, any STE corresponding to an incoming transition of s_j whose ϵ-cover contains at least one final state, is marked as a reporting-STE. The equivalent ANML-NFA for the state-diagram in Figure 2a is shown in Figure 2b.

3.2. Optimizing ANML-NFA

3.2.1. Using all-input-start-STEs. A start-STE is qualified as a *start-of-data* STE unless it is configured as an *all-input-start* STE, which is active in every cycle. The number 1 placed in the indicator at the top-left denotes a start-of-data STE. An all-input-start STE has the symbol ∞ in the indicator instead. By identifying STEs which are active on every symbol-cycle, representing them as all-input-start STEs and removing all incoming lines to the same, an ANML-NFA can be optimized. Next, all start-of-data STEs connected to the outgoing lines of these all-input-start-STEs can be also be converted to start-of-data. This process is continued iteratively till no new all-input-start STEs can be formed. Finally, all incoming edges into all-input-start STEs are removed. Figure 3b shows the optimized ANML-NFA shown in Figure 3a by following this method.

3.2.2. Removing redundant STEs. Any given ANML-NFA may contain redundant STEs. For example, even though the classical NFA shown in Figure 1a is optimal,



(a) (b)

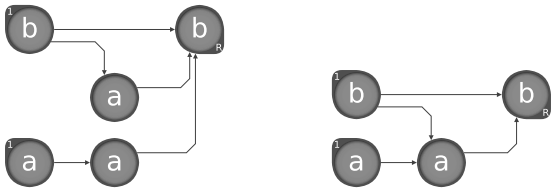
Figure 3: Modifying ANML-NFA using *all-input-start* STE.

Algorithm 2 Removing redundant STEs in ANML-NFA

1: $leaves \leftarrow$ list of all the STEs with no outgoing lines
2: Create an empty list $reportingLeaves$
3: **for** ste **in** $leaves$ **do**
4: **if** reporting(ste) **then**
5: Add ste to $reportingLeaves$
6: **else**
7: Remove ste from $leaves$
8: Remove ste from ANML-NFA with all its incoming lines
9: $newLeaves \leftarrow$ list of STEs which had only one outgoing line connected to r
10: Extend $leaves$ with $newLeaves$
11: **end if**
12: **end for**
13: **for** r_1 **in** $reportingLeaves$ **do**
14: **for** r_2 **in** $reportingLeaves$ **do**
15: **if** $r_1 \neq r_2$ **and** label(r_1) = label(r_2) **then**
16: Merge r_2 with r_1
17: Remove r_2 from $reportingLeaves$
18: **end if**
19: **end for**
20: **end for**
21: **for** r **in** $reportingLeaves$ **do**
22: $thisLevel \leftarrow [r]$
23: **while** size($thisLevel$) > 0 **do**
24: Create an empty list $nextLevel$
25: **for** ste **in** $thisLevel$ **do**
26: $incoming \leftarrow$ list of STEs with outgoing line coming into ste
27: Extend $nextLevel$ with $incoming$
28: **end for**
29: $thisLevel \leftarrow nextLevel$
30: **for** ste_1 **in** $thisLevel$ **do**
31: **for** ste_2 **in** $thisLevel$ **do**
32: **if** ($ste_1 \neq ste_2$ **and** label(ste_1) = label(ste_2) **and** count(outgoing lines of ste_1) = 1 **and** count(outgoing lines of ste_2) = 1) **then**
33: Merge ste_2 with ste_1
34: Remove ste_2 from $thisLevel$
35: **end if**
36: **end for**
37: **end for**
38: **end while**
39: **end for**

its equivalent ANML-NFA shown in Figure 1b is not. Interestingly, even though the original optimized NFA has 6 edge-transitions, the corresponding optimized ANML-NFA has only 4 STEs! This is because one STE may be used to represent multiple state-transitions.

Algorithm 2 identifies redundant STEs which may be removed or merged to produce a more optimized ANML-NFA. It works as follows. First, an *intermediate transformation* is created by removing all the *dead-STE*s, i.e. non-reporting-STEs with no outgoing edges. In the lines $1-12$, all the dead-STEs in the unoptimized ANML-NFA are removed



(a) Intermediate transformation (b) Optimized NFA

Figure 5: Steps in obtaining an optimized ANML-NFA.

along with their incoming lines. If chains of STEs lead to a dead-STE, new dead-STEs are formed. These STEs are removed iteratively, if possible, in the lines $13-20$. Figure 5a shows the intermediate transform of the ANML-NFA shown in Figure 1b.

Finally, in the lines $21-39$, identical chains terminating in the same reporting-STE in the intermediate transform are merged. Starting with a reporting-STE, all the STEs connected to incoming lines with identical labels are merged. Then, the process continues iteratively with all the STEs connected to these merged STEs, until no more STEs can be merged in an iteration. For example, the two STEs with label a connected to the reporting-STE in Figure 5a are merged into a single STE in the final transformation shown in Figure 5b.

3.2.3. Using Counter elements. A Counter element does not consume any symbol from the data-flow and functions as follows. It is configured to activate its outgoing line on reaching a 12-bit *target-value* programmed by the user. The *counter-value* is set to 0 at the beginning of the data-flow, and incremented or reset by activating its *count* and *reset*-lines respectively. The outgoing-line can be configured to generate a *pulsed* or *latched* output, i.e., be active for a single symbol-cycle or stay activated for the rest of the data-flow or until the next reset signal, whichever occurs earlier.

The ANML-NFA shown in Figure 6 checks for substrings in the data-flow wherein the sequence ab repeats 50 times followed by the sequence aa. The equivalent ANML-NFA using a Counter element is shown in Figure 4a. The target-value is shown inside the counter element. The count-line and the reset-line are shown as pentagons on the boundary, and inscribed with the characters C and R respectively. The outgoing-line on the right is accompanied with a ⎍ or ⎍ symbol, denoting a pulsed-output or a latched-output respectively. Similarly, Figure 4b and Figure 4c show ANML-NFAs which count for repetitions above a certain threshold and within a range, respectively. A detailed and generic analysis of when Counter elements may be used to compress NFA implemented in hardware can be found in the works of Becchi et al. [5], [6].

Figure 6: ANML-NFA to identify substrings accepted by the regular expression $(ab)\{50\}aa$.

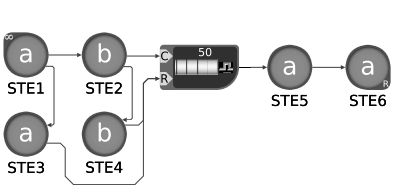
4. Selective Enabling of ANML-NFA at Run-time

Some applications require scanning the data-flow over a user-defined selection of patterns from a fixed but large set. Generation of instantly loadable images for all possible combinations of patterns is intractable. On the other hand, compilation of individual ANML-NFA for each pattern, and then combining the images of only the selected patterns at run-time is inefficient, both in space and time utilization. This is because the images of the individual ANML-NFAs are aligned to row-boundaries, leaving many rows in the combined final image underutilized.

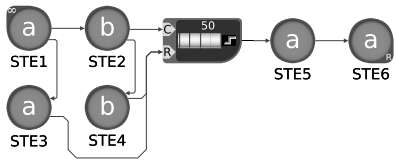
Our proposed technique programs the entire set at all times, thereby requiring a single common image for all user choices. The ANML-NFA for each pattern is designed to enable the search for that pattern only if its *id* is present in a *preamble sequence* streamed at the beginning of the data-flow.

Figure 7 shows *Selective-enable* automaton for a single pattern. Each constituent macro is shown as a rectangular box with pentagonal input and output ports. The pattern itself is encoded as a *Pattern macro*. For a generic succinct representation of any pattern, a shorthand notation for the Pattern macro showing only two STEs is adopted. The first STE, connected to the input port $i2$, symbolizes all the STEs in the Pattern macro from which the matching of the pattern starts. We call them pseudo-start-STEs. In the Selective-enable automaton, they are activated by an *Enable macro*. The other STE represents all the reporting-STEs in the Pattern macro. The dashed lines connecting these two STEs denote the network of intermediate elements and routing lines connecting the two.

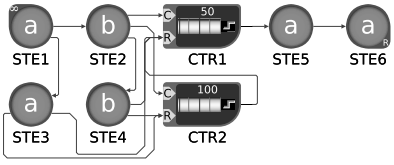
The notation used for the labels of STEs is as follows. The '•' symbol represents the entire 8-bit symbol-set. Σ represents the input alphabet for all the patterns. d represents a 1-byte end-delimiter of the preamble sequence, where $d \notin \Sigma$. The *id* of a pattern is denoted by a 2-byte integer, whose most significant byte cannot take the value of d. In our implementations, we take $d = 255$ which allows us to



(a) Match exactly 50 repetitions.



(b) Match repetitions above the threshold of 50.



(c) Match repetitions within a range of 50 to 100.

Figure 4: Using Counter elements to generate compact ANML-NFA with repeating sub-patterns.

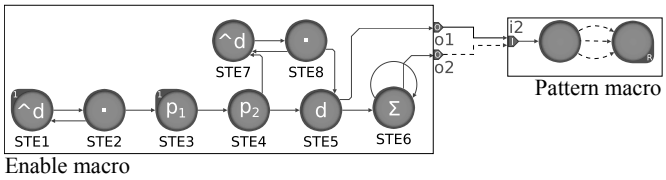


Figure 7: Selective-enable automaton

assign pattern ids from a continuous range $0-65,025$. Note that this range is greater than the number of STEs in a chip and hence sufficient for the number of patterns that can be fit inside a chip. p_1 and p_2 represent the first and the second byte of different pattern ids.

The Enable macro works as follows. *STE*s 1 and 3 process the first symbol of the first id in the preamble sequence. Thereafter, *STE*s 1 and 2 ensure that *STE* 3 is active to process the first symbol of every subsequent id in the preamble sequence. If *STE*s 3 and 4 match the id, then *STE* 5 is activated signifying that the search for the pattern is enabled. *STE*s 7 and 8 are used to keep *STE* 5 activated to match the delimiter at the end of the preamble sequence. On encountering this delimiter, *STE* 5 activates the processing elements connected through the outgoing port $o1$ as well as *STE* 6. *STE* 6 then drives an activation signal to the processing elements connected to the output port $o2$ for every subsequent symbol in the data-flow.

For patterns anchored to the beginning of the data-flow, only the output port $o1$ of the Enable macro is connected to the input port $i2$ of the Pattern macro. For patterns that can match anywhere in the data-flow, the output port $o2$ is also connected to $i2$. This optional connection is shown by a dashed line in Figure 7.

5. Repetition Handling Techniques

We now discuss the handling of applications where the data-flow is logically segmented into multiple partitions. For a pattern to be reported as matched, it must occur in a fixed combination of these logical partitions. A simple approach to finding such patterns can first find all the occurrences of the enabled pattern in the entire data-flow. Then, for each pattern, the host-application can identify all the partitions in which the pattern occurred, and whether these partitions satisfy the overall criteria for matching.

However, such an approach is sub-optimal. The cumulative number of occurrences for all the patterns is typically very high, and mostly belongs to patterns which do not fulfill the overall criteria. This leads to high output generation frequency on the AP, potentially lowering performance. In this section, we describe automata design techniques to overcome this problem and generate reports for only those patterns which satisfy the overall criteria.

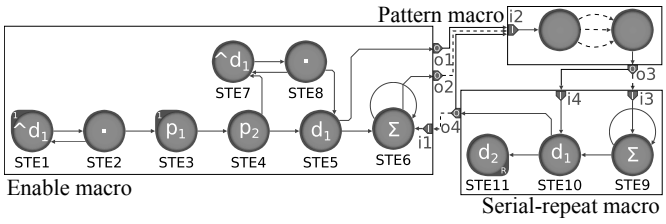


Figure 8: All-repeat-check automaton

5.1. Patterns Occurring in *All* partitions

A pattern that occurs in all the partitions of a data-flow can be identified by its serial occurrence in every partition. The *All-repeat-check* automaton shown in Figure 8 extends the Selective-enable automaton by using the *Serial-repeat* macro to implement this check. The labels of all the STEs in this macro are parameterized. Its working can be described as follows. The end of the preamble sequence and each partition in the data-flow is earmarked by the delimiter d_1. The end of the data-flow is delimited using the symbol d_2, where $d_1, d_2 \notin \Sigma$. The reporting capability of all the reporting-STEs in the Pattern macro is removed and they are instead connected to the output port $o3$.

On finding the id of the pattern in the preamble sequence, the Enable macro activates the pseudo-start-STEs in the Pattern macro for the first partition, as described in Section 4. On matching the pattern in the first partition, *STE* 10 in the Serial-repeat macro is activated. If the occurrence of the pattern needs to be anchored to the end of the partition, then *STE* 10 tries to match the end-of-partition delimiter, which must follow immediately. If this is successful, the pseudo-start-STEs in the Pattern macro are reactivated to search for the occurrence of the pattern in the next partition. If the pattern need not be anchored to the beginning of the segment, port $o4$ is connected to port $i1$, otherwise not. Similarly, if a pattern need not be anchored to the end of every segment, then *STE* 9 is also activated on matching the pattern in a partition, through the connection between $o3$ and $i3$. *STE* 9 ensures that *STE* 10 is active to process the end-of-partition delimiter. If a pattern is not matched in a partition, then *STE* 10 is never activated in that partition, and all the STEs in the automaton become inactive at the end of that partition.

Due to the serial nature of the checks, if *STE* 10 matches the end-delimiter of the last partition, then the pattern must be present in all the previous partitions. In this case, *STE* 11 is activated to match the end-of-data-flow delimiter d_2, which follows immediately afterwards, and report an output. Notice that the output for all the reported patterns are generated on this last symbol of the data-flow.

5.2. Patterns Occurring in At least q partitions

In some applications, a pattern should be reported if it occurs in a significant number of partitions defined by a user-specified threshold. We developed the *Minimum-repeat-check* automaton shown in Figure 9 to find such patterns. It works similarly to the All-repeat-check automaton, but replaces *STE* 11 in the Serial-repeat macro with a Counter element programmed to generate an output on reaching the threshold value of q. Also, it employs a *Bypass* macro to overlook a partition wherein a pattern does not occur and

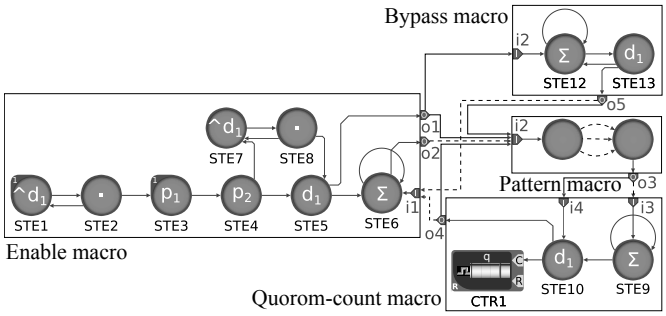


Figure 9: Minimum-repeat-check automaton

continue the search in the next partition. This automaton creates a single output event for every reported pattern.

5.3. Patterns Occurring in Hierarchical Partitions

We now showcase an example where the partitions in the data-flow are hierarchically arranged into three levels l_1, l_2, and l_3 (arranged in ascending order). The data-flow consists of the sequence of l_3-partitions, each containing l_2-partitions, and l_2-partitions, each containing their l_1-partitions. The l_1, l_2, and l_3-partitions are end-delimited by the symbols d_1, d_2, and d_3 respectively. The end of the preamble sequence and the data-flow are earmarked by the symbol d_4, where $d_1, d_2, d_3, d_4 \notin \Sigma$. The order and number of l_3-partitions in the data-flow, of l_2-partitions within an l_3-partition, and of l_1-partitions within an l_2-partition, is irrelevant. However, every l_3-partition must have at least one l_2-partition, and every l_2-partition must have at least one l_1-partition.

For a pattern to be matched in an l_2-partition, it must occur in all its l_1-partitions. Moving higher, a pattern is said to match an l_3-partition, if it matches at least one l_2-partition contained in it. Finally, a pattern is reported if it is matched by every l_3-partition in the data-flow. This example is derived from the solution presented in our previous work [7]. A simple automaton to achieve this complex pattern-matching is shown in Figure 10. It uses the same basic building blocks described in the previous sections.

Enable macro and Pattern macro work identically as described in previous sections. The only difference between the *Serial-repeat* macro defined earlier and the *Non-reporting Serial-repeat* macro is that *STE* 11 is non-reporting. Instead, it is connected to the outgoing port $o5$ in the latter. Collectively, they check if the pattern is matched in all the l_1-partitions of an l_2-partition. If so, *STE* 11 activates *STE* 14 to remember that at least one l_2-partition has been matched in an l_3-partition. The Bypass macro ensures that the pattern matching starts afresh at the beginning of each l_2-partition within the current l_3-partition. Finally, the Serial-repeat macro checks if all the l_3-partitions have been matched in the data-flow, and if so, generates a single output event through *STE* 16 at the end of the data-flow.

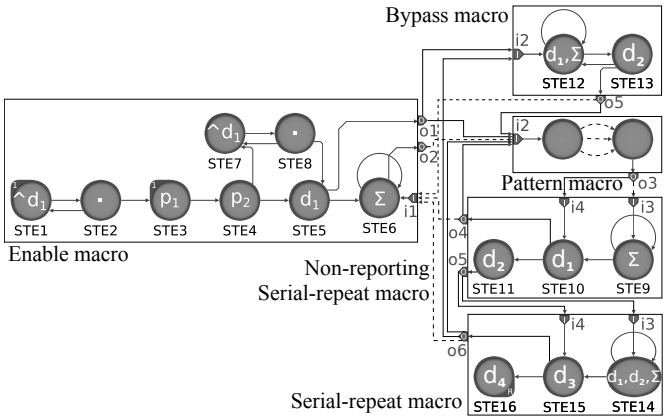


Figure 10: Hierarchical-repeat-check automaton

6. Conclusion

The AP is a reconfigurable co-processor which is specifically built for doing large-scale pattern-matching operations in hardware. This gives it unparalleled simplicity of programming, large capacity, and very high throughput. In this paper, we have presented a number of automata design techniques to make effective utilization of this emerging co-processor. These include techniques for the conversion of NFAs from their classical state-diagram representation to ANML-NFA, their subsequent optimization, and addition of capability to optionally enable or disable their search at runtime. We have also outlined techniques to define complex pattern matching operations, wherein the patterns can occur in combinatorial ways in logical partitions of the input data. We expect that developers can derive performance benefits from this upcoming processor through these automata design and programming techniques.

Acknowledgments

This research is supported in part by the National Science Foundation Exploratory Grant CCF-1448333.

References

[1] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 12, Dec 2014.

[2] M. Becchi, "Data structures, algorithms and architectures for efficient regular expression evaluation," Ph.D. dissertation, Washington University. Department of Computer Science and Engineering., 2009.

[3] H. Nakahara, T. Sasao, and M. Matsuura, "A regular expression matching circuit based on a decomposed automaton," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2011.

[4] Y.-H. Yang and V. K. Prasanna, "High-performance and compact architecture for regular expression matching on FPGA," *Computers, IEEE Transactions on*, vol. 61, no. 7, 2012.

[5] M. Becchi and P. Crowley, "A Hybrid Finite Automaton for Practical Deep Packet Inspection," in *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2007.

[6] M. Becchi and P. Crowley, "Extending Finite Automata to Efficiently

Match Perl-compatible Regular Expressions," in *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2008.

[7] I. Roy and S. Aluru, "Finding Motifs in Biological Sequences Using the Micron Automata Processor," in *IEEE 28th International on Parallel and Distributed Processing Symposium*, May 2014.