A Framework for Fast and Fair Evaluation of Automata Processing Hardware

Xiaodong Yu, Kaixi Hou, Hao Wang, Wu-chun Feng

Department of Computer Science, Virginia Tech

synergy.cs.vt.edu

Objective:

propose a framework for the fast and fair evaluation of Automata Processor devices

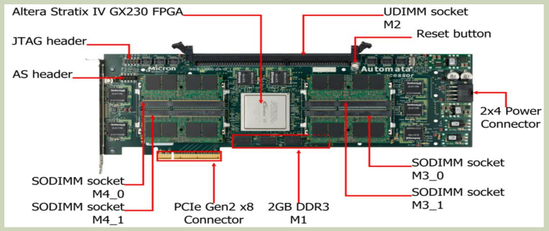
Introduction

•The Automata Processor (AP), introduced by Micron for non-deterministic finite automata (NFA) simulations, can perform parallel automata processing within memory arrays by leveraging memory cells to store trigger symbols and simulate NFA state transitions.

•Previous AP-related designs report thousands-of-fold speedups over their CPU counterparts. However, these remarkable speedups are based on limited-size datasets within a single AP board capacity, thus treating the compilation and configuration overhead as a one-time cost and excluding it from performance evaluation. We argue that such computation-only comparisons become unfair when the dataset size exceeds a single AP board capacity, thus requiring multi-round reconfigurations.

•The AP software development kit (SDK) provides pre-compiled macros to reduce the reconfiguration cost. However, each macro only works for a fixed-shape automaton; any automaton structural change forces the construction to start from scratch, which is a complicated and error-prone process since it requires expertise in both automata theory and AP architecture.

First generation - Micron D480 Automata Processor



Contributions

•We highlight the importance of counting the reconfiguration time in the overall AP performance, which in turn, can provide better insight for researchers to identify essential hardware architecture features.

•We provide a framework for users to fully and easily explore AP device capacity and conduct fair comparisons to counterpart hardware. It includes a hierarchical approach to automatically generate AP automata and cascadable macros to ultimately minimize the reconfiguration cost.

•We evaluate our framework using real-world datasets and conduct an end-to-end performance comparison (i.e., configuration+computation) between the AP and CPU.

Automata Processor

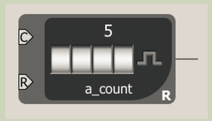
•THREE programmable components: state transition elements (STEs), counters, and boolean gates

• STEs simulate NFA states with trigger symbols. Connections between STEs simulate NFA state transitions.

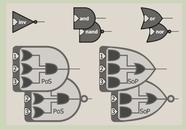
GUI icons

STE

counter



Boolean



Automata Processor (cont'd)

• Developers can define their own AP automata using Automata Network Markup Language (ANML), a XML-based language, by describing the layout of STEs and transitions. It is complicated even for a very simple automaton.

•AP compilation is the most time-consuming step in AP workflow due to the complex place-&-route calculations. To mitigate the overhead of compilation, the AP SDK supports the pre-compiling of automata to be *macros* to enable automata reuse.

Sample ANML code for a simple automaton

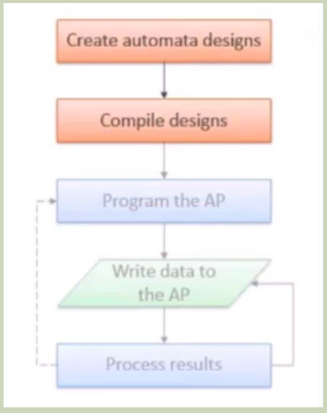
```
ap_anml_t anml = 0;
ap_anml_network_t anmlNet;
struct ap_anml_element element;
ap_anml_element_ref_t element[5];
ap_automaton_t a;
ap_element_map_t elementMap;

// Create the ANML object
anml = AP_CreateAnml();

// Create the automata network in the ANML object
AP_CreateAutomataNetwork(anml, &anmlNet, "anl");
// Create the elements that match "a" and start the search
element.ree_type = RT_STE;
element.start = START_OF_DATA;
element.symbols = "a";
element.match = 0;
AP_AddAnmlElement(anmlNet, &element[0], &element);
// Create the elements that match "b" report the match
element.ree_type = RT_STE;
element.start = NO_START;
element.symbols = "b";
element.match = 1;
AP_AddAnmlElement(anmlNet, &element[1], &element);
// The rest four STEs are created in the same manner
// with different symbols attribute
  .....
// Connect the STEs together to search "abc"
// and allow one Levenshtein distance
// Match
AP_AddAnmlEdge(anmlNet, element[0], element[1], 0);
AP_AddAnmlEdge(anmlNet, element[4], element[5], 0);
// Insertion
AP_AddAnmlEdge(anmlNet, element[0], element[2], 0);
AP_AddAnmlEdge(anmlNet, element[2], element[4], 0);
AP_AddAnmlEdge(anmlNet, element[1], element[3], 0);
AP_AddAnmlEdge(anmlNet, element[3], element[5], 0);
// Substitution
AP_AddAnmlEdge(anmlNet, element[2], element[5], 0);
// Deletion
AP_AddAnmlEdge(anmlNet, element[0], element[5], 0);
// Compile the ANML object to create the automaton
AP_CompileAnml(anml, &a, &elementMap, 0, 0, options, 0);
return 0;
```

Figure 1: AP automaton

AP system workflow

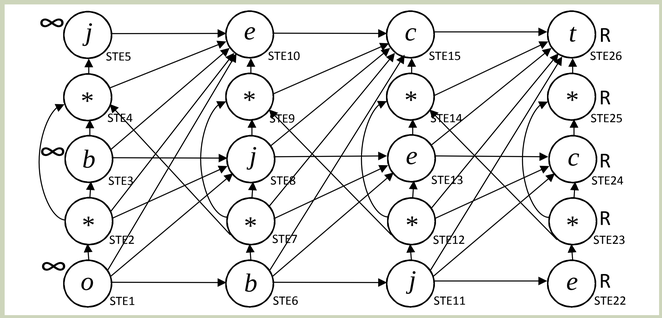


Framework Design

I. Approximate Pattern Matching (APM) on AP

• APM finds strings that match a pattern with a limited number of errors (e.g., insertion, deletion, and substitution). The Levenshtein distance is one of the most common distance types that allow all three kinds of APM errors.

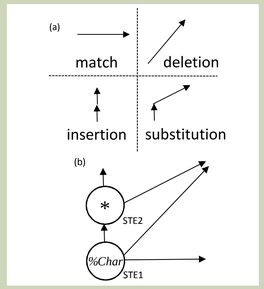
Levenshtein automaton for pattern "object" allowing two errors



II. Paradigm-based AP Automata Construction

***Paradigms and Building Blocks*:**

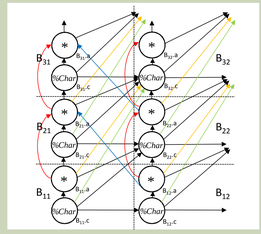
• Three types of errors: *insertion* (I), *deletion* (D), and *substitution* (S), along with *match* (M), can be treated as the paradigms of any APM. For the Levenshtein automata, a building block including two STEs and four types of transitions.



Framework Design (cont'd)

***Block Matrix*:**

• With the building block, pattern length *n* and the maximum number of errors *m*, building an AP automaton can be implemented by duplicating the building blocks and organizing them into a $(m + 1) * (n - m)$ block matrix.



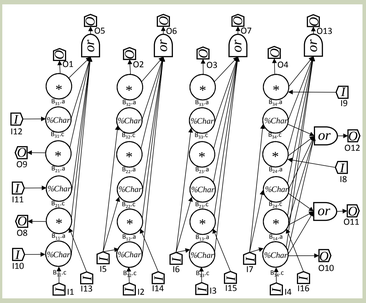
***From Building Blocks to Automata*:**

• We provide an algorithm to automatically fabricates complex AP automata. It creates the building blocks, builds up the automaton with the block matrix determined by maximum error (m) and target pattern length (n), and finally sets all STE labels.

III. Cascadable Macros Design

• Although the pre-compiling technique can build macros to reduce the recompilation time, it is restricted to same automata structure reuse.

We propose *cascadable macros* to support extended reuse of macros. With the cascadable macros, only the connections between instances need to be placed-&-routed.

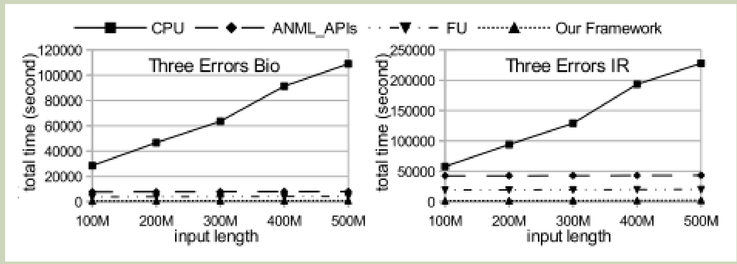


Experiment Results

• We evaluate the framework with two real-world datasets: the BLAST dataset ("Bio") and the NHTSA information retrieval dataset ("IR").

• We compare our performance to other two AP automata construction approaches: Functional Units (FU) and basic ANML APIs (ANML), as well as CPU implementation PatMaN.

• The figures show our framework can achieve 2x to 461x (runtime+compiling time) speedup over CPU PatMaN and up to 33.1x and 14.8x speedup over ANML APIs and FU, respectively.



Conclusions

• In this work, we provide a framework allowing users to easily conduct fair end-to-end performance comparison between AP and its counterparts, especially for large-scale problem sizes leading to high reconfiguration costs.

Acknowledgements

This work was supported in part by NSF I/UCRC IIP-1266245 via NSF CHREC and the SEEC Center via the Institute for Critical Technology and Applied Science (ICTAS).