EFFCLIP + UAP: UNIFIED, EFFICIENT REPRESENTATION AND ARCHITECTURE FOR AUTOMATA PROCESSING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER

DEPARTMENT OF COMPUTER SCIENCE

BY
YUANWEI FANG

CHICAGO, ILLINOIS
2015
ABSTRACT

Finite automata models are a fundamental computing abstraction for algorithms and solutions to a growing range of problems (bioinformatics, text search, ranking, compression, and network monitoring). Consequently there have been two vibrant threads of invention - new finite automata models and novel implementations - but so far there is no accepted software-hardware interface, a computer architecture, to separate and thereby accelerate progress in both software and hardware.

We propose the Unified Automata Processor (UAP), a new architecture that provides general and efficient support for finite automata (FA). We also propose a novel automata transition packing approach used by UAP, “efficient coupled-linear packing” (EffCLiP), that optimizes both finite-automata size and performance. EffCLiP and UAP together enable high performance flexible automata processing. They enable new automata models and applications to be created and deployed as software exploiting integration of high performance automata processing on FA accelerated CPU, instead of requiring a new ASIC/FPGA. This technique elevates novel automata innovation for future hardware platform. EffCLiP employs a novel transition representation that enables a simple addressing operator (integer addition) while providing flexibility to support transition packing. With EffCLiP, the UAP supports a wide range of existing finite automata models (DFAs, NFAs, A-DFAs, JFAs, counting-DFAs, and counting-NFAs), and future novel FA models.

The interface to UAP adds a small number of instructions and registers to a traditional core with vector extensions. The UAP system has multiple parallel lanes pairing with scratchpad memory to allow parallel FA processing. We add lane completion detection feature which allows multi-step FA execution for each UAP lane, and the software-controlled static mapping between input streams and lanes which allows flexible pattern aggregation or aggressive throughput. The UAP lane has three main parts: sequence, combining queue, and stream buffer. The sequence component processes primitives and actions each with one
clock cycle. The combining queue holds multiple active states and ensures no concurrent identical states. The stream buffer stores the input stream and frees the vector register file for other use during UAP execution.

Evaluation across a variety of widely-used finite automata workloads and models show that EffCLiP achieves compression rates as high as 8.4x and packing densities of 77% for DFA, 2.24x and >99% for A-DFA, 3.9x and >99% for NFA while supporting high speed and simple implementation.

Evaluation on realistic workloads shows that UAP implements all models efficiently, achieving line rates of 94.5% of ideal. A single UAP lane delivers line rates 50x greater than software approaches on CPUs and GPUs. Scaling UAP to 64 lanes achieves FA transition throughputs as high as 295 Gbps, more than 100x higher than CPUs and GPUs, and even exceeding ASIC approaches such as IBM’s RegX by 6x. With efficient support for multiple input streams, UAP achieves throughputs that saturate even high speed stacked DRAM memory systems. We implemented the UAP design with EffCLiP, whose implementation is efficient in instructions executed, memory references, and energy. UAP achieves a 1.2 Ghz clock rate (32nm TSMC CMOS), and a 64-lane UAP requires only 2.2 mm$^2$ and 563 mW. At these levels of performance and energy-efficiency, UAP is a promising candidate for integration into general-purpose computing architectures.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ix</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>x</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Current Approaches</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Our Approach: EffCLiP and UAP</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Thesis Work</td>
<td>9</td>
</tr>
<tr>
<td>1.5 Summary of Contributions</td>
<td>10</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Basic Finite Automata</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Transition Compression</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Extended Finite-Automata</td>
<td>14</td>
</tr>
<tr>
<td>2.4 Aho-Corasick Automata</td>
<td>15</td>
</tr>
<tr>
<td>2.5 Recent Hashing Techniques</td>
<td>16</td>
</tr>
<tr>
<td>3 EFFCLiP: EFFICIENT COUPLED-LINEAR PACKING</td>
<td>18</td>
</tr>
<tr>
<td>3.1 Compression Opportunity</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Efficient Coupled Linear Packing</td>
<td>24</td>
</tr>
<tr>
<td>3.2.1 Transition Representation</td>
<td>24</td>
</tr>
<tr>
<td>3.2.2 Linear Packing for Finite Automata</td>
<td>26</td>
</tr>
<tr>
<td>3.2.3 Majority Compression</td>
<td>29</td>
</tr>
<tr>
<td>3.3 Generalization to $D^2$FA, NFA, and String Matching</td>
<td>32</td>
</tr>
<tr>
<td>3.3.1 EffCLiP for $D^2$FA and A-DFA</td>
<td>32</td>
</tr>
<tr>
<td>3.3.2 EffCLiP for NFA</td>
<td>32</td>
</tr>
<tr>
<td>3.3.3 EffCLiP for JFA, XFA, and counting-FA</td>
<td>33</td>
</tr>
<tr>
<td>3.3.4 EffCLiP for Aho-Corasick String Matching</td>
<td>33</td>
</tr>
<tr>
<td>3.4 EffCLiP Evaluation</td>
<td>35</td>
</tr>
<tr>
<td>3.4.1 Metric</td>
<td>35</td>
</tr>
<tr>
<td>3.4.2 Workloads</td>
<td>35</td>
</tr>
<tr>
<td>3.4.3 Concrete EffCLiP Transition Implementation</td>
<td>36</td>
</tr>
<tr>
<td>3.4.4 Workloads Configuration</td>
<td>38</td>
</tr>
<tr>
<td>3.4.5 DFA: Full Table vs EffCLiP</td>
<td>38</td>
</tr>
<tr>
<td>3.4.6 A-DFA: Full Table vs EffCLiP</td>
<td>39</td>
</tr>
<tr>
<td>3.4.7 NFA: Full Table vs EffCLiP</td>
<td>40</td>
</tr>
<tr>
<td>3.4.8 String Matching: Hash vs EffCLiP</td>
<td>41</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Extensible Applications Using Finite Automata ............................................. 2
1.2 Traversal Algorithms on CPUs/GPUs .............................................................. 2
1.3 CPU/GPU exploiting FA models using standard tool chain .............................. 4
1.4 A Simple NFA and its Implementation in Logic Gates ..................................... 5
1.5 FPGA development flow .................................................................................. 5
1.6 Hardware Landscape of Finite Automata Processing Approaches .................... 8
1.7 Easy and fast development on Systems with UAP inside ................................. 8
2.1 NFA and DFA representations for three patterns (a+bc, bcd+ and cde) ............. 12
2.2 Basic DFA Data Structure of Pattern “abca” .................................................... 13
2.3 Uncompressed DFA and DFA using Alphabet Compression Table .................. 14
2.4 Run Length Encoding Example ....................................................................... 14
2.5 Aho-Corasick Automata Matching he,hers,his,she ............................................ 16
3.1 EffCLiP abstract FA transitions creating a simplified transition model .......... 19
3.2 EffCLiP has an abstract memory model ......................................................... 19
3.3 A naive packing scheme VS EffCLiP packing scheme. Note that ’a’ = 97 (ASCII),
    ’b’ = 98, ’e’ = 101. Slot address space is overlayed with state name space, e.g.
    target state of transition (0,e,1) is in slot 101 (0+’e’ = 101). Colored are occupied
    slots holding EffCLiP transitions .................................................................. 20
3.4 EffCLiP second phase: post packing ............................................................... 21
3.5 Majority Tail Phenomenon in ab.*cd ............................................................... 22
3.6 Formulation of Linear Packing ...................................................................... 26
3.7 EffCLiP and Full Table Implementation General View .................................... 37
3.8 EffCLiP and Full Table Transition Formats .................................................... 37
3.9 DFA Compression Rate .................................................................................. 39
3.10 DFA Memory Utilization ............................................................................... 39
3.11 A-DFA Compression Rate ............................................................................. 40
3.12 A-DFA Memory Utilization .......................................................................... 41
3.13 NFA Compression Rate ................................................................................ 41
3.14 NFA Memory Utilization .............................................................................. 42
3.15 Breakdown for Analyzing Majority Compression Impact on A-DFA ............ 43
3.16 EffCLiP Memory Size Progression ................................................................. 44
4.1 UAP using actions to emulate complex FA models’ transition operations ......... 47
4.2 # patterns in 256KB, selected representative workloads .................................. 49
4.3 A small set of mechanisms can support diverse finite automata models; six models
    used for evaluation are in green .................................................................... 50
4.4 UAP Transitions and Memory Layout ............................................................. 52
4.5 UAP Transition Primitives ............................................................................ 52
4.6 2 Dimensions of Transition-level Parallelism ............................................... 61
4.7 Novel Automata on UAP ............................................................................... 63
4.8 UAP Micro-architecture composes vectors, UAP lanes, and local memory banks.  63
4.9 Line Rate vs FA Model .............................................  69
4.10 Instructions per Finite Automata Transition vs. Lanes and Multi-step ....  69
4.11 Line Rate, Line Rate/Power, and Bandwidth Utilization with Local Memory
    Width Increase ..................................................  70
4.12 Density: UAP vs RegX .............................................  71
4.13 Transition Throughput vs # of Lanes ................................  71
4.14 Scaling up with multiple streams on DDR3 and HMC memory systems on DFA
    models .............................................................  72
4.15 Comparing Throughput: UAP (1-wide) on several FA models vs RegX (4-wide)  73
4.16 System Throughput vs Number of Patterns ................................  73
4.17 Micro-architecture of a UAP Lane .............................................................  75
4.18 64-lane Area and Single-Stream Power .............................................  76
4.19 UAP Overall System Energy .............................................  77
LIST OF TABLES

3.1 The Majority Tail Phenomenon in Practical Workloads .......................... 22
3.2 Basic Notation ..................................................................................... 23
3.3 Experiment Configurations and Workload Properties ......................... 38
3.4 Aho-Corasick String Matching ............................................................. 42
4.1 Transition Primitive and Action Formats ............................................. 53
4.2 Type Field and Transition Primitives .................................................... 54
4.3 How to identify transition primitive’s type? ....................................... 55
4.4 Assembly Definition of UAP Words (Primitives and Actions) ............. 56
4.5 Overall Instructions to access UAP Functionality ................................. 57
4.6 Workload (Pattern Sets) used for Evaluation ...................................... 66
4.7 UAP Area Breakdown .......................................................................... 76
CHAPTER 1
INTRODUCTION

1.1 Motivation

Finite automata are an important computational model that are widely used to process large volumes of unstructured data, often under real-time constraints. Example large-scale web applications with real-time requirements include web-search and ranking [40], Question Answering systems such as Watson, Siri, Cortana, and Google Now [17, 21, 39, 45], and compression in widely-used NoSQL systems [13, 20, 35]. Those systems use regular expressions to represent the features for the questions and these queries are highly latency critical. On the other hand, many genomics applications include pattern matching as a performance-critical phase, followed by a more general-purpose computation. For example, shotgun assembly [51] includes an all-to-all comparison of DNA reads followed by a graph-traversal algorithm to recover the chromosome sequences. Pattern-matching computations map naturally onto finite automata abstractions [22]. Computationally expensive biological applications that include a pattern matching phase include: de novo assemblers [18], orthology inference [28], and motif discovery [44]. These applications have huge amount of mining data which need great system throughput to perform automata processing. Other areas where finite automata are essential include text data mining [12], network monitoring [42], content-based routing [11], and application-level filtering [19]. Figure 1.1 shows some of the most important finite automata applications. As a result, there has been a lot of research on novel high performance FA processing, especially the development of new variations of finite automata models [4, 6, 7, 24, 25, 30, 47, 55, 58]. Note that, each FA model requires different hardware support for performance.

However, in general finite automata processing is hard since 1) it requires sequential memory indirections, 2) unpredictable memory access/branch, 3) and short data items (e.g.
Figure 1.1: Extensible Applications Using Finite Automata

8-16 bits) which reduces the speculation and prefetch techniques of modern processors. Since
the next state transition cannot proceed until CPU fetches the current transition, the process
time for a single FA transition is limited by the L1 hit time (typically 4 CPU cycles). Figure
1.2a and Figure 1.2b show a basic DFA and NFA processing algorithm on the CPU/GPU
system. As can be seen, the critical loop highlighted in the red box contains unpredictable
memory indirections and conditional check for acceptance. Even worse, put input streams
and FAs in the same memory hierarchy increase the chance of cache conflict thus further
increase the memory reference latency. Another major reason is that those FA processing
applications require high speed real-time reaction which is often constraint by network and
external device, and multiple layers of FA processing.

(a) DFA traversal algorithm

\[
\begin{align*}
\text{current}_v &= \text{initial}_v \\
\text{while } \neg \text{input}.\text{empty} \text{ do} \\
\quad \text{c} &= \text{input}.\text{first}() \\
\quad \text{input} &= \text{input}.\text{next}() \\
\quad \text{current}_v &= \text{TransitionTable}[\text{current}_v][c] \\
\quad \text{if } \text{current}_v.\text{accept} == 1 \text{ do} \\
\quad \quad \text{report match;}
\end{align*}
\]

(b) NFA traversal algorithm

\[
\begin{align*}
1: & \quad \text{current}_v \leftarrow \text{initial}_v \\
2: & \quad \text{while } \neg \text{input}.\text{empty} \text{ do} \\
3: & \quad c \leftarrow \text{input}.\text{first} \\
4: & \quad \text{input} \leftarrow \text{input}.\text{tail} \\
5: & \quad \text{future}_v \leftarrow \text{current}_v \land \text{persistent}_v \\
6: & \quad \text{while a transition on c is pending do} \\
7: & \quad \quad \text{src} \leftarrow \text{transition source} \\
8: & \quad \quad \text{dst} \leftarrow \text{transition destination} \\
9: & \quad \quad \text{if } \text{current}_v[\text{src}] \text{ is set then} \\
10: & \quad \quad \quad \text{atomicSet}(\text{future}_v, \text{dst}) \\
11: & \quad \quad \text{end if} \\
12: & \quad \text{end while} \\
13: & \quad \text{current}_v \leftarrow \text{future}_v \\
14: & \quad \text{end while} \\
15: & \quad \text{return current}_v
\end{align*}
\]

Figure 1.2: Traversal Algorithms on CPUs/GPUs
1.2 Current Approaches

**CPUs and GPUs** are widely used and have a full breath coverage in modern computing systems. The availability and accessibility is high on these devices. There are many software libraries to use such as python re library, Java JRegex library, Scala Regex, c++ regex, GNU c regex library, etc with a mature software delivery channels. CPU/GPU approach offers great software programmability and software integration into multiple layers of software stack for FA processing. The fast deployment time and simple development phase is shown in Figure 1.3a and Figure 1.3b. Exploiting FA models on CPU’s only uses standard tool chain.

However, there are several key limitations of CPU/GPU approach. The fast deployment/development time and great software programmability comes with the cost of low performance because CPU/GPU architectures don’t perform well in 1) indirection intensive FA traversing, 2) memory hierarchy support for FA and input streams storage, 3) instruction set to execute FA processing. Published results [15,31] show that CPU/GPU [57] can be 50 times slower than a carefully designed ASIC/FPGA accelerator (see Chapter 5) and >100 time less energy efficient (see Figure 1.6). Empirical experiments show that CPU (Xeon, intel x86) delivers ≈ 20Mbps multi-stream FA processing throughput and GPU (GTX480) delivers ≈200Mbps multi-stream throughput. However, both CPUs with SIMD extension and GPUs still have a significant instruction count for each automata transition, which fundamentally limits the performance. Furthermore, state explosion and large counting quantifiers in the expressions also make the CPUs/GPUs approach far away from high performance.

**FPGAs** approach is another candidate for high-performance FA processing. It can achieve high performance [32,46,54] and offers significant throughput improvement compared to general purpose approach. FPGA hard-wires the regular expressions directly into the logic gates constructing an equivalent NFA. The NFA-to-Gate mapping example is shown in Figure 1.4. This approach makes the NFA processing performance much better than the
CPUs/GPUs and avoids the state explosion problem. Our analysis shows FPGA is ≈20x higher line rate performance than CPU/GPU (see Chapter 5) and >10x more energy efficient than CPU/GPU (see Figure 1.6). However, the throughput limitation for FPGA lays on the multi-stream processing. FPGA has to duplicate circuit resources for each independent input stream which is a fundamental limitation on its throughputs. Usually, FPGA is used for single-stream NFA processing achieving ≈13Gbps line rate.

On the other hand, FPGA has major limitations such as FPGAs are not a part of general-purpose computing device in modern computing systems. Even super-computing systems such as BlueGene, Titan, or Tianhe do not have FPGA integrated. Realizing the low FPGA accessibility and availability, Intel has a future plan that may ship embedded FPGA with Xeon processor. However, embedded FPGA approach is still complicated to use. With the help of OpenCL and accelerator abstract layer, software programmers can directly implement the functionality in FPGA from a high-level language. But the challenge is dealing with both a different set of constraints (gates, memory capacities, clock period), type of parallelism (bit level, small bits), and completely different performance structure (what expressions and structures go fast, which go slow, which generate large sets of wires, which generate long clock cycles), and the energy consumption for various implementations. Software programmers are unfamiliar with hardware level performance/energy optimizations due to lack of hardware knowledge. Another major limitation is the FPGA’s poor software integration and
programmability. FA processing that embedded in multiple software layers can not benefit from FPGAs. The time-consuming and complicated FPGA development phase is depicted in Figure 1.5. Incorporation of these FPGA technologies is only now, 30 years after the emergence of fpga’s, happening in large-scale complex computing environments. And, only in the most technologically sophisticated and performance-demanding settings [40]. Even there the primary difficulties of instability in software-hardware interface and the need for cross-layer expertise in expression function, optimization, and as important “operation” remain critical challenges.

![Figure 1.4: A Simple NFA and its Implementation in Logic Gates](image)

Through the analysis of CPU/GPU approach and FPGA approach, we find that FPGA provides 10-100x performance and energy benefit over CPU/GPU approach, which is attracting. However, FPGA deployment into the computing systems has much more limitations

![Figure 1.5: FPGA development flow](image)
such as direct control of the host computing system, expense cost to purchase expensive high-end FPGA, much more human-labor effort on both developing hardware and software, much longer acceleration development cycles, lack of hardware knowledge etc. In this respect, high-performance cheap and general finite automata processing allowing easy deployment is still an unaddressed question. To this end, we propose EffCLiP and UAP that closes the gap between fast deployment/development and high performance general FA processing.

1.3 Our Approach: EffCLiP and UAP

We propose the Unified Automata Processor (UAP), a new architectural approach for efficient finite automata processing. EffCLiP and UAP together enables high performance flexible automata processing. They enable new automata to be created and deployed as software exploiting integration of high performance automata processing on FA accelerated CPU, instead of building specific ASIC/FPGA for high performance/energy-efficiency which is known to be slow development cycle, label-intensive, platform-specific and expensive. With UAP integration, modern CPUs/GPUs will have a boost in extended automata processing and the pervasiveness of CPUs/GPUs also makes UAP easily accessible to everybody enabling cheap and high performance general automata processing everywhere.

On the one hand, this approach has the advantage of CPU/GPU with fast deployment and easy software programmability. On the other hand, EffCLiP and UAP together delivers high-performance general automata processing that approaches ASIC’s high performance and energy efficiency. To inherit high programmability and fast deployment of CPU/GPU, it requires UAP to be integrated into a traditional CPU core and has a set of instructions to program it. Low-power consumption and little area overhead are the key challenges for UAP-CPU integration. We carefully design the UAP execution engine pairing with a shared scratchpad memory on the chip. The power consumption can be as tiny as 7.9mW for a single UAP engine. To inherit high performance and energy efficiency of FPGA/ASIC requires
UAP has intelligent algorithm and specialized design. UAP is specially designed to support the execution of all advance FA models to achieve high performance and efficiency. Several architectural mechanism such as vector parallelism, multi-step execution, delayed acceptance check, state combining are added to further boost UAP performance. UAP borrows EffCLiP’s abstraction transition representation and linear packing algorithm to achieve near optimal memory density. Under these respects, UAP is very competitive in performance, power consumption, and memory utilization with traditional ASIC design.

At the instruction set level, UAP adds a small number of instructions and registers to a traditional core with vector extensions. Thus, UAP operations can be efficiently combined with traditional computing software. At the micro-architecture level, UAP integrates a small set of new functions for finite-automata actions. These functions, combined with a local memory, enable UAP to support a wide range of finite automata models, including traditional non-deterministic and deterministic finite automata (NFAs and DFAs, respectively) and popular models that have been proposed in the networking domain (A-DFAs [7], JFAs [58], counting-DFAs [6], and counting-NFA [6]). We developed EffCLiP, a flexible transition representation to make the special logic of UAP as simple as possible to deploy UAP into a general purpose processor. UAP reuses the vector register files and local memory banks of 10x10 architecture and adds special simple sequence logic. The UAP builds on prior work to enable efficient DFA processing deeply integrated with traditional software [16]. The goal of UAP is to narrow down the gap between high performance with great efficiency and the flexibility to combine with software. As Figure 1.6 shows, ASIC achieves best automata processing energy efficiency through well-designed specialized circuit to exactly fit the application. CPUs/GPUs have better programmability but suffer from insufficient support for automata processing thus have low energy efficiency. Our UAP fits into the big picture of automata processing architectures achieving both energy-efficient automata processing (approaching ASIC) and high programmability (approaching CPU).
UAP is a specialized FA processing execution engine integrated into a traditional CPU core. CPU vendors like Intel, Qualcomm, or AMD ship UAP-integrated CPU chips everywhere in the world. New generation of CPU will have UAP inside and modern computing system from research group cluster, Amazon EC2, or supercomputing centers will have each node carrying CPU with UAP inside. UAP enjoys fast and easy development property of CPU/GPU as shown in Figure 1.7.
1.4 Thesis Work

In this thesis, we present an automata processing architecture, that combines high automata processing performance and automata model flexibility, delivering the capability to software directly. In order to design such a general purpose automata processing architecture, we first identified the applications that require high automata processing performance like network intrusion detection, motif searching, text mining, content-based filtering, NoSQL query, signal triggering, etc. We found different applications need different types of matching patterns (some require many .*, some are string, some need high counting constraint, etc). We next carefully analyzed recently proposed automata and distilled down their unique features. With those automata architecture insights, we created a mapping and compression algorithm called “EffCLiP”, mapping general automata’s topologies to abstract memory storage.

With EffCLiP in mind, we designed a hardware architecture that maps the abstract memory storage to physical memory and has a general purpose set of automata processing primitive. In addition, we employed several additional architecture techniques to improve the performance of automata processing. For example, to reduce the instruction count limitation on performance, we used the vector extension and multi-step execution for automata processing. Together, these architectural features support efficient general purpose automata processing, for the models we studied and many more that can be invented. Finally, we used real workloads to evaluate the implementation of UAP and compare it with state-of-art designs (IBM RegX). The UAP is a potential candidate for unstructured data processing with high efficiency and high performance. It can be used as an IP block which allows flexible integration with current CPU/GPU architectures.
1.5 Summary of Contributions

The thesis lays a foundation for high performance FA processing of efficiency, flexibility, and fast development time. Specific contributions include:

- Analysis of FA models and design a set of mechanisms that support all FA models

- An abstract definition of a novel EffCLiP transition representation and coupled-linear packing algorithm that supports a wide range of finite-automata models (DFA, NFA, A-DFA,...). The transition representation employs a simple fixed addressing operator (addition) and enables fast (linear-time) and effective packing algorithms.

- Evaluation of the EffCLiP transition representation using one concrete instance, and Majority Compression for a range of finite-automata models (DFA, A-DFA, and NFA) that show EffCLiP achieves a compression rate of 8.4x and densities of 77% for DFA; 2.24x and >99% for A-DFA; 3.9x and >99% for NFA.

- The design of the unified automata processor (UAP) including a concrete instruction set interface, transition, action, and word format.

- Evaluation of its architectural efficiency, specifically the UAP lane design that supports general-purpose, efficient finite automata computing, and its integration into a conventional instruction set architecture using few memory references per transition (often one), and requiring little execution energy beyond those references (close to maximum achievable efficiency).

- Evaluation of UAP physical implementation (32nm CMOS, 1.2 Ghz), shows small area (2.2 $mm^2$ for 64 lanes) and low power (563 milliwatts at max rate). The performance evaluation empirically demonstrates its generality across finite automata models, and high performance as measured by line rate and throughput. Line rates up to 9.07 Gbps
(1.13 giga-symbols per second), 50x faster, and 295 Gbps, 100x faster than CPU and GPU implementations.

- Comparison of UAP to a commercial ASIC design (PowerEN RegX) [31] that shows UAP delivers comparable line-rate and higher throughput for simple FA models. And that UAP’s programmability enables it to exploit advanced FA models not supported by others to deliver 6x higher application performance.

The remainder of the thesis is organized as follows. In Chapter 2, we describe the background of a broad range of finite automata models that have been developed, and we then motivate and refine a set of exemplar finite automata models that represent the full range of diversity. In Chapter 3, we present the EffCLiP that enables flexible implementation of different kinds of finite-automata and shows the compression and speed performance of the technique with a set of popular finite-automata. We present the design of the Unified Automata Processor (UAP), including both the software interface and micro-architecture in Chapter 4. Besides, to support a rigorous evaluation, we describe representative workloads and our modeling approaches and experimental results in Chapter 4 and these results are discussed in the context of related work in Chapter 5. We close in Chapter 6 with a summary of our results and a discussion of several promising future research directions.
CHAPTER 2
BACKGROUND

Regular expression matching engines employ finite automata processing as a basic primitive, so high performance must be achieved on a variety of platforms. In general, approaches face a trade-off between the size of the automaton and the worst-case bound processing per-character.

2.1 Basic Finite Automata

Finite automata (FA) [22] are an effective algorithmic tool for multi-pattern search. In automata-based approaches, the matching operation is equivalent to a FA traversal guided by the content of the input stream. Worst-case guarantees on the input processing time can be met by bounding the amount of per character processing. For larger pattern-sets and complex patterns, limiting the size of the automaton becomes challenging. Deterministic finite automata (DFA’s) are fast, with every state presenting a single transition for each input symbol (see Figure 2.1b).

An NFA can have many transitions per symbol at each state, leading to a large number of active states (all of which require processing). In general, there can be multiple transitions

![Figure 2.1: NFA and DFA representations for three patterns (a+bc, bcd+ and cde)]
for each of multiple active states. Figure 2.1a illustrates multiple active states in an NFA. Any NFA can be translated into a DFA, but an NFA with \( N \) states can require as many as \( 2^N \) states in an equivalent DFA.

The basic DFA data structure is a 2D table like figure 2.2. Each row corresponds to a given state and each entry in the row corresponds to the next state for the given input symbol. The structure of NFA is similar as DFA’s but it isn’t an exact 2D table. Each input symbol of a given state of an NFA may correspond to several target states. SRAM-based approaches (except for FPGA) implement the transition functions of automata as table look up.

The basic DFA data structure is a 2D table like figure 2.2. Each row corresponds to a given state and each entry in the row corresponds to the next state for the given input symbol. The structure of NFA is similar as DFA’s but it isn’t an exact 2D table. Each input symbol of a given state of an NFA may correspond to several target states. SRAM-based approaches (except for FPGA) implement the transition functions of automata as table look up.

![Diagram](image)

\( \Sigma = \{ a, b, c \} \)

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( b )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.2: Basic DFA Data Structure of Pattern “abca”

### 2.2 Transition Compression

Researchers have shown \([24, 37, 47, 55]\) that “state explosion” happens only with complex patterns (typically those containing bounded and unbounded repetitions of large character sets). So, one approach is to use the DFA model for speed, but compress the transitions using alphabet reduction \([10, 23, 48]\) or run-length encoding \([10]\). These representations do not alter the number of states in the DFA, but reduce the number of transitions or total size by exploiting the transition redundancy in realistic DFA structures.

For example, multiple Alphabet Compression Tables (ACT) \([23]\) exploits redundancy in
target states, translating the automata to avoid storing same target multiple times for a particular state. The Alphabet Compression Table act as meta data required to achieve highly-compressed implementation of an unstructured DFA. The example of alphabet compression table is shown in Figure 2.3.

![Uncompressed DFA and DFA using Alphabet Compression Table](image)

Figure 2.3: Uncompressed DFA and DFA using Alphabet Compression Table

Figure 2.4 shows the example using run length encoding to shrink the size of the transition table. Each entry the original 2D table (Figure 2.4a) corresponds to the next state of a given input symbol of a state. After exploiting the columnar redundancy using run length encoding as Figure 2.4b, we only need less than half the memory size to store the original transition table.

![Run Length Encoding Example](image)

Figure 2.4: Run Length Encoding Example

### 2.3 Extended Finite-Automata

Many proposals extend DFA models on traditional computing systems to produce smaller size and higher performance, often for particular workloads. These efforts employ (i) transition
compression to reduce DFA memory footprint, and (ii) new automata models that reduce the number of states. Examples of transition compression techniques include: alphabet reduction [10, 23], run-length encoding [10], transition redundancy removal [50] and bitmap [41]. These representations do not alter the number of states in the DFAs, but reduce the number of transitions, and do so by exploiting the transition redundancy present for realistic DFA workloads. Novel finite automata models include: multiple-DFAs [56], delay-input DFAs [5, 25], hybrid-FAs [4], history-based-FAs [24], XFAs [47], counting-FAs [6], JFAs [58], SFAs [55] and dual-FAs [30]. These alternative automata models avoid or mitigate the state explosion problem in the DFA representation. While reducing the resource requirements of NFAs and DFAs, some of these automata models have equivalent expressive power.

While each automata model gives benefits for several types of FA structure, none is best for all regular expression patterns. A-DFA, JFA, and c-DFA respectively achieve highest pattern density for some patterns in representative workloads (see Section 4.1 for detail). Unfortunately some of these extended FA’s [6, 58], cannot represent all patterns. Each automata model has its set of strengths and weaknesses, and the most powerful and flexible systems must support many models.

All of these FA models, including NFA and DFA, have equivalent expressive power, but differ significantly in encoding size (depending upon both FA model and implementation) and per-transition computation. As we show, EffCLiP can be used to densely encode and efficiently implement all of these models. In Section 3.3, we show how it can be applied to half a dozen exemplar FA models.

### 2.4 Aho-Corasick Automata

Beginning with the Aho-Corasick (AC) automaton [3] that has deterministic worst case performance, many multi-string matching variations [26, 36, 49, 53] have been proposed. For a set of string patterns, an AC automaton is constructed by resembling a trie with links
between the various nodes/states. Each node/state on the trie corresponds to a valid prefix of strings and links on a trie are known as \textit{go to} transitions. Another type of links are known as \textit{failure} transitions. They allow fast transitions between failed pattern matches to branches with a common prefix (see example in Figure 2.5 for matching he, hers, his, she).

![Aho-Corasick Automata Matching he, hers, his, she](image)

Figure 2.5: Aho-Corasick Automata Matching he, hers, his, she

### 2.5 Recent Hashing Techniques

Hashing is a popular technique to implement FA transition function. For high transition function speed, the hash function should be fixed and needs to have as few hash conflicts as possible. For generality, the hash function should work for all kinds of FA topologies. For memory size efficiency, the hash function should have a high load factor minimizing the memory size. For energy efficiency, the hash function shouldn’t have additional memory references other than the basic one. A lot of efforts have been made to meet these four challenges. However, these techniques can’t meet all four requirements at the same time. Several researchers have proposed hashing to implement transitions and layouts \cite{26,27,50,53}, requiring either additional translation tables, or that the hash function be programmable to
have the generality matching any kinds of FA transition structure. Those hash techniques can be categorized into two classes: fixed hash function [26, 50, 53], reconfigurable hash function [27]. In the former case, additional memory references are needed. For example, cuckoo hashing [26] is used to make several chaining hash memory access to avoid a conflict. A hash function directly operated on bit mask [50] needs fetching multiple possible result at the same time. In order to bound the number of conflicts, BFSM technique [50] needs to split the transition table into multiple smaller tables. In perfect hash technique [53], it uses a secondary reconfigurable look-up table to rename the states’ name to avoid conflicts. All the hashing techniques have the hash conflict problem and they require additional memory references to avoid the conflict. In the latter, the requirement of programmable hash hardware increases the complexity and energy, and reduces the speed of implementation. As we’ll see, EffCLiP solves all four challenges, combining a fixed simple transition operator (addition) with high load factors without translation table.
CHAPTER 3

EFFCLIP: EFFICIENT COUPLED-LINEAR PACKING

In this chapter, we explain the Efficient Coupled-Linear Packing algorithm. EffCLiP offers perfect hash for automata transitions with single memory access, which allows flexible compression technique for automata. As a result, EffClPiP offers both performance and size optimization for various finite-automata models.

EffCLiP representation works for all automata models in an abstract way. There are many popular FA models such as DFA [22], NFA [22], A-DFA [7], c-DFA [6], c-NFA [6], JFA [58], etc. However, their transitions can be classified into two types after ignoring operations associated with the transitions: 1) transition with a source state and one destination state and an input symbol 2) non-input symbol consuming transition (see Figure 3.1). The size of the FA models strongly depend on the number of transitions they have. In order to solve the dense transition packing problem, EffCLiP has a simplified transition model using two transition primitives that 1) (source state, symbol, target state), 2) (source state, target state) in a FA model ignoring complex behavior associated with the transition primitive. As a result, each FA model has a set of EffCLiP abstract transitions which need to be placed in a memory layout. For the memory model, EffCLiP has a list of linear slots which can hold one EffCLiP transition per slot (see Figure 3.2).

EffCLiP’s packing algorithm aims to efficiently create a memory layout and implement the FA transition function enabling fast and flexible addressing. For example, given a source state S and an input symbol “m”, how to address the EffCLiP transition that contains the target state T? The challenge lays in the implementation of transition function should be fixed, to allow fast hardware implementation but it also should be flexible enough for any FA models’ topologies; the resulted memory layout should be highly dense to have good memory utilization; memory reference number should be minimized to reduce energy cost. Extended FAs can associate complex operations with each transition. EffCLiP acknowledges
such operations, but does not address them explicitly. These complex operations present significant challenges, and are an important part of what is addressed in design choice in UAP architecture (See Chapter 4).

Here is a brief description about what EffCLiP packing does. In general, EffCLiP has two phases given a set of EffCLiP transitions which consist of two transition primitives.

First, EffCLiP packs the symbol-consuming primitives of EffCLiP transitions. EffCLiP chooses the integer addition on the current state identifier and input symbol to compute the next transition. In order to allow packing freedom and make it possible for any automata topology, we rename the states and overlay the automaton’s naming space with the linear address space of slots (the slots’ ID space). Figure 3.3 gives an intuition of the EffCLiP approach. The original FA has 4 states with 4 EffCLiP transitions. How to pack them compactly? A naive packing implementation would use a hash function. Due the the randomness property in hash function, there are some empty slots. But they are needed so as to maintain the linear slot addressing space (slot ID space). With EffCLiP, slot naming space
requires as few as 4 linear slot IDs (98,99,100,101) with appropriate state renaming. Thus only 4 slots are needed. More rigorous packing detail can be found in the following sections. In one word, EffCLiP captures the abstract transition representation which works for every automaton model and manages to minimize the size of linear slots needed to hold the whole automaton model (assume 1 abstract EffCLiP transition per slot) whilst maintaining fast fixed hash function to implement the automaton’s transition function given a list of linear slots.

Figure 3.3: A naive packing scheme VS EffCLiP packing scheme. Note that 'a' = 97 (ASCII), 'b' = 98, 'e' = 101. Slot address space is overlayed with state name space, e.g. target state of transition (0,e,1) is in slot 101 (0+'e' = 101). Colored are occupied slots holding EffCLiP transitions

The Second step is post packing. This is the minor refinement for non-input consum-
ing EffCLiP transitions. Those transitions are packed after the first step and update the “attach” field of the symbol-consuming EffCLiP transition packed already. Figure 3.4 gives an intuition of this step. More detail can be found in the algorithm description of EffCLiP below.

![EffCLiP diagram](image)

**Figure 3.4: EffCLiP second phase: post packing**

### 3.1 Compression Opportunity

Our goal is to combine dense encoding (size) with high throughput (speed). Size is critical because it affects data movement and working set costs, and throughput is critical to meet performance objectives. We address both of these challenges, and produce a combined solution that achieves both small size and low transition complexity. We first document, and then exploit a frequent pattern in finite automata that we term *majority tail*. The idea of majority tail arises from the observation that a common pattern in finite automata is many transitions from a state, 1 share a target state, 2 and most of transitions from state 2 are self-transitions (also go to state 2). In this case, we term (1,2) the majority tail of state 1 (see Section 3.2.3 for rigorous definition).

We illustrate this “majority tail” situation for the regular expression `ab.*cd` in Figure 3.5. Every state with its most frequent transition target state forms a majority tail: (0,0),
Figure 3.5: Majority Tail Phenomenon in ab.*cd

(1,0), (2,2), (3,2), (4,2). This phenomena is frequent; we present measurements from varied finite-automata and their majority tail frequency is over 99% (see Table 3.1).

<table>
<thead>
<tr>
<th>Spyware</th>
<th>#DFAs</th>
<th>#Maximum Possible Majority Tails</th>
<th>#Majority Tails</th>
</tr>
</thead>
<tbody>
<tr>
<td>spyware</td>
<td>23</td>
<td>713,532</td>
<td>70,550</td>
</tr>
<tr>
<td>EM</td>
<td>13</td>
<td>29,307</td>
<td>29,307</td>
</tr>
<tr>
<td>range0.5</td>
<td>13</td>
<td>30,698</td>
<td>30,444</td>
</tr>
<tr>
<td>range1</td>
<td>13</td>
<td>33,214</td>
<td>32,906</td>
</tr>
<tr>
<td>dotstart0.05</td>
<td>23</td>
<td>165,297</td>
<td>163,981</td>
</tr>
<tr>
<td>dotstar0.1</td>
<td>30</td>
<td>260,347</td>
<td>258,437</td>
</tr>
<tr>
<td>dotstar0.2</td>
<td>57</td>
<td>340,414</td>
<td>338,118</td>
</tr>
<tr>
<td>bro217</td>
<td>3</td>
<td>2,945</td>
<td>2,945</td>
</tr>
<tr>
<td>tcp_homenet_externalextenalnet</td>
<td>43</td>
<td>105,568</td>
<td>105,545</td>
</tr>
</tbody>
</table>

Table 3.1: The Majority Tail Phenomenon in Practical Workloads

**Basic Definitions** We define basic notation then introduce an abstract EffCLiP transition data type. That data type, defined by its operations, is sufficient discussion of optimization issues. To enable quantitative evaluation, a concrete instance of the EffCLiP transition is introduced in Section 3.4.1.

A DFA is a 5-tuple \((Q, \Sigma, \delta, q_0, A)\), where \(Q\) is a finite set of states, \(\Sigma\) is the alphabet, \(\delta : Q \times \Sigma \rightarrow Q\) is a transition function, \(q_0\) is a start state, \(A \subseteq Q\) is a set of accepting states. All notations being used in following sections are listed in Table 3.2. Our goal is to store a DFA in a linear name space \(M\), whose transitions are addressed by simple hardware operator of \((\text{current state}, \text{symbol})\), even with different kinds of compression schemes.

**Defn 1.** \(M\) is called an \(n\)-dimension binary vector if \(M \in \{0, 1\}^n\).
Table 3.2: Basic Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q )</td>
<td>a set of DFA states</td>
</tr>
<tr>
<td>( \Sigma )</td>
<td>the alphabet</td>
</tr>
<tr>
<td>( \delta )</td>
<td>a DFA transition function</td>
</tr>
<tr>
<td>( p, q )</td>
<td>a DFA state</td>
</tr>
<tr>
<td>( \text{EL} )</td>
<td>effective length</td>
</tr>
<tr>
<td>( A )</td>
<td>a set of accepting states of a DFA</td>
</tr>
<tr>
<td>( m_q )</td>
<td>the majority of state ( q )</td>
</tr>
<tr>
<td>( s_q )</td>
<td>the binary transition vector of state ( q )</td>
</tr>
<tr>
<td>( t_q )</td>
<td>a state picked for ( q ) to form ( s_q )</td>
</tr>
<tr>
<td>( P )</td>
<td>a post-packing set</td>
</tr>
<tr>
<td>( M )</td>
<td>binary vector</td>
</tr>
<tr>
<td>( \mathbb{M} )</td>
<td>a linear name space, (</td>
</tr>
</tbody>
</table>

**Defn 2.** The effective length of \( \overrightarrow{v} \) is defined as \( \text{EL}(\overrightarrow{v}) = j - i + 1 \), where \( \overrightarrow{v} = (v_1, v_2, \ldots, v_n) \) is an \( n \)-dimension binary vector, \( i \) is the index of its first non-zero coordinate and \( j \) is the last.

**Defn 3.** The load factor \( \alpha \) of a binary vector \( M \) is given as follows: \( \alpha = \frac{K}{\text{EL}(M)} \), where \( K \) is number of non-zero coordinates in \( M \).

**Defn 4.** Define binary transition vector (BTV) of \( q \) as follows:

\[
s_q(n) = \begin{cases} 
0 & \text{if } \delta(q, \varphi(n)) = t_q \\
1 & \text{otherwise}
\end{cases}
\]

where \( \varphi \) is a bijective function: \( \{1, 2, \ldots, |\Sigma|\} \rightarrow \Sigma, n \in \{1, 2, \ldots, |\Sigma|\}, t_q \in Q \) or \( t_q = \emptyset \).

**Defn 5.** \( P \) is called a post-packing set if \( P \) contains all elements that aren’t needed for linear packing but needed to be put in \( M \) after linear packing.

For example, let \( t_q \) be an any state \( \in Q \). For every state \( q \), we arbitrarily pick a state \( t_q \), construct a BTV \( s_q \). Tuple\((q, t_q)\), can be considered as the representative of all transitions originating from state \( q \) going to state \( t_q \). Since we are going to store the representative
transition: \((q, t_q)\) in \(M\) instead of linear packing all transitions from \(q\) to \(t_q\), we put the tuple \((q, t_q)\) in the post-packing set \(P\). After linear packing, we put \(P\) in \(M\).

### 3.2 Efficient Coupled Linear Packing

#### 3.2.1 Transition Representation

We present an abstract transition representation, enabling discussion of linear packing and majority compression. First, we define several abstract basic operations to perform EffCLiP DFA state traversal. We use C++ style pseudo-code such at \(t\).foo() where \(t\) is the current transition, and foo() is the transition class function invoked on \(t\). Since each transition points to the next state, so “target” acts as the pointer to next state. Besides, we may perform actions on a given transition so that “attachment” specifies the action. At the same time, we perform the transition compression so that “info” is needed to specify the compressed transition stored in the “attachment”. In the end, we use the hash function to do state transition so that “source” is needed to verify the transition. In short, EffCLiP representation captures the general topology as well as flexible compression and actions.

- \( t.\text{EffCLiP}_\text{transition}(\text{source, target, info, attachment}) \) constructs an EffCLiP transition. \textit{source} label each transition uniquely, enabling a signature check for match. \textit{target} is transition destination. \textit{info} encodes the transition type for the “non-regular” case, \textit{attachment} is used in those cases.

- \( t.\text{get}_\text{transition}(\text{symbol}) \) fetches a transition computed by \textit{integer_add} of the current state and input symbol. This is the “regular” case.

- \( t.\text{signature}_\text{check}(\text{transition, symbol}) \) returns true if transition matches current state and symbol, otherwise false.

- \( t.\text{get}_\text{transition}_\text{explicit}() \) fetch the transition using the explicit address in \textit{attachment}. 
• \texttt{t.accept()} returns true if current state is an accepting state.

Algorithm 1 DFA Traversal\((q_0, in)\)

\begin{verbatim}
let post-packing set \( P \leftarrow \emptyset \)
for all dfa transitions \( tr : (source, symbol, target) \) and \( t_q, \forall q \) do
    \( tr.EffCLiP\_transition(source, symbol, target, info, attach.) \)
    add \((q, t_q)\) to \( P \)
end for
Put EffCLiP transitions and post-packing set \( P \) in memory

//====== start state traversal
symbol \leftarrow \text{in.head}
initialize \( tr_{cur} \)
while \( symbol \neq \text{null} \) do
    \( tr_{fet} \leftarrow tr_{cur}.get\_transition(symbol) \)
    if \( tr_{cur}.signature\_check(tr_{fet}, symbol) \) then
        if \( tr_{fet}.accept() \) then
            AcceptProcedure\((tr_{fet})\)
        end if
        \( tr_{cur} \leftarrow tr_{fet} \)
    else
        \( tr_{cur} \leftarrow tr_{cur}.get\_transition\_explicit() \)
    end if
    symbol \leftarrow symbol.next()
end while
\end{verbatim}

After picking \( t_q \) for every state \( q \) and construct BTVs associated with \( t_q \), we place the EffCLiP transitions and any post packing set element \((P)\). Optimization of their placement compatible with our simple addition addressing operator in subsequent sections. For FA optimizations that store a single representative transition \((q, t_q)\) to denote a set of transitions from \( q \) to \( t_q \), \textit{attachment} explicitly specifies the transition. This name is used when signature check fails indicating where to find the needed representative transition (depends on the FA model). Algorithm 1 shows the corresponding state traversal.

The core idea of the EffCLiP is to use a simple operation (\textit{integer addition}) on the current state identifier and input symbol to compute the next transition. However, the critical element is to name the transitions so that this simple operation can be used, and a dense encoding of the finite automata is achieved. Together these two elements enable
minimize \( M \in \{0, 1\}^{[M]} \) 
\[ \max \{ \text{EL}(M), \alpha \text{EL}(M) + |P| \} \]
subject to 
\[ \sum_{i=1}^{|Q|} h(s_i, l_i) = M, \ s_i \in S, l_i \in \mathcal{L} \]
\[ l_i \neq l_j, \ \forall i \neq j \]

where 
\( \mathcal{L} = \{0, 1, 2, \ldots, |M| - 1\} \)
\( S \) is a set of BTVs derived from \( \delta \)
\( P \) is the post-packing set
\( \alpha \) is the load factor of \( M \)
\( h : S \times \mathcal{L} \to \{0, 1\}^{[M]}, [h(s_i, l_i)](n) \)
\[ = \begin{cases} 
    s_i(n - l_i) & n \in [l_i, l_i + |\Sigma| - 1] \\
    0 & \text{otherwise}
\end{cases} \]

Figure 3.6: Formulation of Linear Packing

fast implementation in hardware. To begin, for each state \( q \), EffCLiP picks a state \( t_q \) from amongst its transition targets, and defines \( (q, t_q) \) as the representative transition for \( q \).

3.2.2 Linear Packing for Finite Automata

At a high-level, EffCLiP linear packing has three steps.

1. Pick \( t_q \) for every state \( q \), form the post-packing set, and construct a set of binary transition vectors from transition function \( \delta \).

2. Place those binary transition vectors in a linear name space.

3. Add elements of the post-packing set to the name space, filling gaps that correspond to unused names.

Linear Packing Formulation

Given a DFA \( (Q, \Sigma, \delta, q_0, A) \) and a name space \( M \) of size \( |Q| \times |\Sigma| \). The linear packing problem is formulated as shown in Figure 3.6. Where \( M \) is a linear name space with each
name storing a EffCLiP regular transition. The goal is to place the DFA compactly. The transition function $\delta$ is converted to be a set of binary transition vectors (BTVs). The placement problem then formulated as how to place the BTVs in $M$ such that the “1”s in one BTV do not overlap those in other BTVs. Every state has a unique identifier and every name in $M$ at most stores one transition.

**A Greedy Linear Packing (GLP) Algorithm**

The size of the post-packing set $|P|$ is a constant after we pick $t_q$ for each $q \in Q$. Let’s assume $\alpha \leq 1 - \frac{|P|}{EL(M)}$, or $\alpha \to 1$, so the optimization target is as follows:

$$\text{minimize } \max \{EL(M), \alpha EL(M) + |P|\}$$

$$\Leftrightarrow \text{minimize } EL(M)$$

Later we demonstrate how each finite-automata model meets this assumption. To minimize the effective length of $M$, we propose a heuristic greedy linear packing algorithm (GLP). Our GLP places longest BTVs first, the heuristic is based upon the notion these are the hardest to pack whilst short BTVs are relatively easy to pack. Let $n$ be the number of states in the DFA. GLP time complexity is $O(n)$ and memory size is $O(n)$. The inputs of GLP are a set of DFA states $Q$, a list of binary transition vectors $S$, a list of state identifiers $N$ that stores the mapping between BTVs and states. The GLP’s output is a set $R$ of tuples (new identifier, old identifier).

GLP starts packing every BTV from the beginning of $M$, and tries to insert it without overlapping “1”s. For 8-bit symbols, the length of each BTV is 256, so checking for overlap is a constant time operation. Inserting a BTV always succeeds, as it can always be inserted after last “1” in $M$. With the initial size of $|M| = |Q| \times |\Sigma|$, the successful placement of each BTV is guaranteed. With the return tuple set $R$, each BTV’s location is defined in $M$. The new identifier of each state is actually the start position of its BTV in the linear name space $M$, each state is also renamed. After all the states have been renamed, we assign elements
Algorithm 2 GLP \((Q, S, N, R)\)

sort \(S = \{s_1, s_2, \ldots, s_{|Q|}\}\) based on EL in decreasing order

update \(N = \{n_1, n_2, \ldots, n_{|Q|}\}\) correspondingly

\(L \leftarrow \phi\)

let \(M\) be a name space contains all 0, s.t. \(|M| = |Q| \times |\Sigma|\)

for \(i \in (1, 2, \ldots, |Q|)\) do

\(loc \leftarrow 0\)

\(insert \leftarrow false\)

while \(insert = false\) do

\(test \leftarrow false\)

while \(test = false\) do

\(test \leftarrow true\)

for \(j \in (loc, loc + 2, \ldots, loc + |\Sigma| - 1)\) do

if \(M[j] \land s_i[j - loc]\) then

\(test \leftarrow false\)

break

end if

end for

\(loc \leftarrow loc + 1\)

end while

\(l \leftarrow loc - 1\)

if \(l \notin L\) then

\(L.add(l)\)

\(R.add((n_i, l))\)

\(insert \leftarrow true\)

\(M[l, l + 1, \ldots, l + |\Sigma| - 1] \leftarrow s_i\)

end if

end while

end for

return \(R\)
of post-packing set $P$ unused names in $M$.

### 3.2.3 Majority Compression

With the EffCLiP transition representation, we have shown how to create a working finite-automata implementation. Here we present the *majority compression* optimization which exploits the EffCLiP flexibility to do effective compression of majority tails, whilst maintaining the simple next state transition function – integer addition.

**Defn 6.** We say $M_q$ is the majority set of state $q$ of a DFA $(Q, \Sigma, \delta, q_0, A)$, if $M_q = \{p \in Q : |\Gamma_{q,p'}| \leq |\Gamma_{q,p}|, \forall p' \in Q\}$, where $\Gamma_{i,j} = \{\gamma | \delta(i, \gamma) = j, \gamma \in \Sigma\}, i,j \in Q$.

**Defn 7.** State $m_q$ is called the majority of state $q$ if $m_q \in M_q$, where $M_q$ is the majority set of $q$. For every $q$, if $|M_q| > 1$, we fix $m_q$ arbitrarily.

**Defn 8.** Tuple $(p, m_p)$ is called a majority pair, where $m_p$ is the majority of state $p$.

**Defn 9.** We say $(p, q)$ is a majority tail if $q = m_p$ and $q = m_q$, where $m_q, m_p$ are the majorities of state $p, q$ respectively.

A state’s majority set reflects finite-automata redundancy over transitions that share the same source and target. Each state and its majority target form a majority pair and may also form a majority tail. Thus, the total number of majority pairs and maximum number of majority tails in a DFA are both $|Q|$. Our measurements suggest that in real DFA workloads, most majority pairs are also majority tails (more than 99%). With this observation, we are able to shrink DFA size by employing the EffCLiP representation without sacrificing performance.

Picking $m_q$ to be $t_q$ during the binary transition vector $s_q$ construction for state $q$ and storing $m_q$ in the *attachment* of EffCLiP transition representation has 2 benefits: 1) Tuple$(q, m_q)$ represents the most numerous transitions originated from $q$. Making $(q, m_q)$ the representative transition for $q$ avoids storing all transitions from $q$ to $m_q$, achieving
compression. Because \((q, m_q)\) is a majority tail, we can avoid additional memory references if signature check fails as we already have the needed data stored in attachment in current transition: \(m_q\) is the next state, \((m_q, m_q)\) is a Majority Tail, and \(m_q\) is not an accepting state. With this information we create a synthetic transition, avoiding the need to fetch one from memory. (if \(m_q\) is an accepting state, we never pick \(m_q\) as \(t_q\).) By assigning \(t_q = m_q, \forall q \in Q, |P| = |Q| - |N_{norm}|\), where \(P\) is the post-packing set and \(N_{norm} = \{p| (p, m_p)\text{is a majority tail, }p \in Q\}\). Usually, \(|P| \sim 0.01|Q|\), and EL(M) > \(|Q|\). Therefore, \(\alpha \leq 1 - \frac{|P|}{EL(M)}\). For EffCLiP to support Majority Compression, we need to add one more abstract operation.

- \texttt{t.type()} returns the type of the current transition (MAJORITY PAIR, MAJORITY TAIL, DEFAULT TRANSITION, DEACTIVATE) from encodings in the \texttt{info} bits.

The remaining abstract EffCLiP transition operations are as in Section 3.2. Next, we demonstrate a complete example of majority compression algorithm with EffCLiP representation using Greedy Linear Packing (GLP).

The majority compression algorithm first forms a list of BTVs and stores representative transition for each state which cannot be synthetically created because they cannot form a majority tail with their majority. Next, GLP determines the start locations of all BTVs. Knowing the exact location, all transitions (EffCLiP regular transitions) in a binary transition vector (denoted by “1”) are placed as a 6-tuple transition (here we omit accepting information which is actually encoded in \texttt{info}). Finally, updating attachment in all EffCLiP transitions to add the explicit address of tuples (representative transitions) in the post-packing set \(P\).
Algorithm 3 Majority Compression \((Q, \Sigma, \delta, q_0, A, M)\)

\[ S \leftarrow \phi, N \leftarrow \phi, P \leftarrow \phi, R \leftarrow \phi, D \leftarrow \phi \]

initiate every basic block in \(M\) to be 0

\[ //======\]

\[ \text{construct BTV list } S \text{ and post-packing set } P \]

\[ \text{for } q \in Q \text{ do} \]
\[ \quad \text{construct BTV } s_q \text{ from } \delta \]
\[ \quad S.append(s_q), N.append(q) \]
\[ \quad \text{if } (q, m_q) \text{ doesn’t form a majority tail then} \]
\[ \quad \quad P.append(q, m_q) \]
\[ \quad \text{end if} \]
\[ \text{end for} \]

\[ //======\]

\[ \text{place EffCLiP transitions based on } R \]

\[ \mathcal{R} \leftarrow \text{GLP } (Q, S, N, R) \]

\[ \text{for } q \in Q \text{ do} \]
\[ \quad \text{for } \alpha \in \Sigma \text{ do} \]
\[ \quad \quad \text{let } (\delta(q, \alpha), l_{\delta(q,\alpha)}), (q, l_q), (m_{\delta(q,\alpha)}, l_{m_{\delta(q,\alpha)}}) \in \mathcal{R} \]
\[ \quad \quad \text{if } \text{tuple}(\delta(q, \alpha), m_{\delta(q,\alpha)}) \notin P \text{ then} \]
\[ \quad \quad \quad M[l_q + \alpha] = (l_q, \alpha, l_{\delta(q,\alpha)}, \text{MajTail}, l_{m_{\delta(q,\alpha)}}) \]
\[ \quad \quad \text{else} \]
\[ \quad \quad \quad M[l_q + \alpha] = (l_q, \alpha, l_{\delta(q,\alpha)}, \text{notMajTail}, \text{ADDR}) \]
\[ \quad \quad \text{end if} \]
\[ \quad \text{end for} \]
\[ \text{end for} \]

\[ //======\]

\[ \text{put post-packing set } P \text{ in } M \]

\[ \text{for } (p, m_p) \in P \text{ do} \]
\[ \quad \text{let } (m_p, l_{m_p}), (m_{mp}, l_{mp}) \in \mathcal{R} \]
\[ \quad \text{if } \text{tuple}(m_p, m_{mp}) \notin P \text{ then} \]
\[ \quad \quad \text{insert } (\text{null}, \text{null}, l_{mp}, \text{MajTail}, l_{mp}) \text{ at addr if } M[addr] = 0 \]
\[ \quad \quad \text{else} \]
\[ \quad \quad \quad \text{insert } (\text{null}, \text{null}, l_{mp}, \text{notMajTail}, \text{ADDR}) \text{ at addr if } M[addr] = 0 \]
\[ \quad \quad \text{end if} \]
\[ \quad D.add((m_p, addr)) \]
\[ \text{end for} \]

\[ //======\]

\[ \text{update ADDR field in } M \]

\[ \text{for } (p, addr) \in D \text{ do} \]
\[ \quad \text{replace ADDR with } addr \text{ for all } M[i] \text{ if } m_t = p, \text{ where } (t, M[i].target) \in \mathcal{R} \]
\[ \text{end for} \]
3.3 Generalization to D²FA, NFA, and String Matching

3.3.1 EffCLiP for D²FA and A-DFA

Recall A-DFA (an improved version of D²FA) explores transition redundancy for standard DFA by using default transition compression. As part of the implementation process, A-DFA creates irregular structure, making transition addressing by simple operator (state,symbol) difficult. Hashing [26,27] is a commonly approach, but it suffers from inability to guarantee successful placement and unstable achieved packing. EffCLiP solves this problem efficiently for A-DFA. The construction of BTVs for A-DFA is simple: we let $t_q$ be the default state of state $q$. For states using default transitions, the location of the default state is stored in the attachment and the info identifies the attachment type. If the signature check fails, we fetch the transition based on attachment which contains the address of default transition. The post-packing set $P$ is empty, $|P| = 0$, which meets the assumption ($\alpha \leq 1 - \frac{|P|}{EL(M)}$). We extend the EffCLiP t.type() with DEFAULT_TRANSITION. Majority Compression can also be applied to A-DFA using EffCLiP extended abstractions. A-DFA’s have root states with full transition representations which are susceptible to our Majority Compression. Similarly, for non-root A-DFA states, we can compare default transition compression and Majority Compression and pick the one produces least transitions.

3.3.2 EffCLiP for NFA

From an operational point of view, the primary distinction between NFA’s and DFA’s is that an NFA may follow multiple transitions for a single input symbol. We encode this possibility in EffCLiP by chaining concurrent transitions together through the attachment. We pick one transition of NFA if one state has concurrent transitions on the same symbol, treat it as regular transition, and put the rest in the post-packing set $P$. GLP is used to allocate all regular transitions. Then, we put the post-packing set containing the remaining concurrent
transitions in the memory. Because the full names of each concurrent transition is encoded in the attachment, there are no layout implications. Although \(|P|\) increases, NFA has much fewer transitions than DFA, thus making linear packing much easier and the load factor \(\alpha \to 1\). We extend one EffCLiP abstract operation.

- \(\text{t.get.transitions}(\text{symbol})\) returns 1 or more transitions. For an NFA, one state may have several transitions for the same input symbol. One of them is addressed by simple addition of current state and symbol, rest are chained using the attachment, and placed as part of the post-packing set.

In Algorithm 4, we present a generalized state traversal algorithm supporting DFA, A-DFA, NFA, and Majority Compression.

### 3.3.3 EffCLiP for JFA, XFA, and counting-FA

Next we briefly note our support for JFA [58], XFA [47], counting-FA [6], this is achieved by attaching rules and actions associated with the transitions. Each time we verify the fetched transition, an evaluation on the rules is performed. If the evaluation is true, we do the associated actions. For example, JFA has \(\text{jump}\) action, \(\text{variable}\) actions, XFA and counting-FA have \(\text{counter}\) actions; each of the actions is triggered under certain conditions.

### 3.3.4 EffCLiP for Aho-Corasick String Matching

Fixed string matching usually uses AC algorithm which has goto functions and failure transitions. Without changing EffCLiP abstract operations, AC-DFA string matching can be naturally supported by treating the failure transitions as EffCLiP default transitions and requires no change to our abstract operations.
Algorithm 4 Generalized Traversal($q_0, in$)

for all transitions $tr : (source, symbol, target)$ and $t_q, \forall q$ do

$tr$.EffClIP.transition(source, symbol, target, info, attach.)

end for

linearly pack EffClIP transitions

$\text{symbol} \leftarrow in\_head$

initialize $tr_{\text{cur}}$, add $\{tr_{\text{cur}}\}$ to $T$, let $T' \leftarrow \emptyset$, $S \leftarrow \emptyset$

while $\text{symbol} \neq \text{null}$ do

for $\forall tr_{\text{cur}} \in T$ do

$T' \leftarrow tr_{\text{cur}}$.get_transitions($\text{symbol}$)

for $\forall tr_{\text{fet}} \in T'$ do

if $tr_{\text{cur}}$.signature_check($tr_{\text{fet}}$, $\text{symbol}$) then

if $tr_{\text{fet}}$.accept() then

AcceptProcedure($tr_{\text{fet}}$)

end if

$tr_{\text{cur}} \leftarrow tr_{\text{fet}}$

else

switch ($tr_{\text{cur}}$.type())

case MAJORITY_TAIL:

$tr_{\text{cur}} \leftarrow (\text{null, null, } tr_{\text{cur}}.\text{attachment}, \text{MajTail, } tr_{\text{cur}}.\text{attachment})$

break

case MAJORITY_PAIR:

$tr_{\text{cur}} \leftarrow tr_{\text{cur}}$.get_transition_explicit()

break

case DEFAULT_TRANSITION:

while $\neg tr_{\text{cur}}$.signature_check($tr_{\text{fet}}$, $\text{symbol}$) do

$tr_{\text{cur}} \leftarrow tr_{\text{cur}}$.get_transition_explicit()

$tr_{\text{fet}} \leftarrow tr_{\text{cur}}$.get_transition($\text{symbol}$)

end while

$tr_{\text{cur}} \leftarrow tr_{\text{fet}}$

break

case DEACTIVATE:

continue

end switch

end if

end for

end for

$T \leftarrow S, S \leftarrow \emptyset$

$\text{symbol} \leftarrow \text{symbol.next}()$

end while
3.4 EffCLiP Evaluation

3.4.1 Metric

Metrics for Finite Automata Models

- Compression Rate: $\frac{\text{size of baseline}}{\text{total memory}}$, includes unused memory.
- Memory Utilization: $1 - \frac{\text{wasted memory (empty)}}{\text{total memory}}$, this metric is similar to load factor.
- Majority Tail Fraction: $\frac{\text{# of majority tails}}{\text{# of states}}$, this metric reflects the frequency of the majority tail phenomenon.
- Memory Size: Total memory footprint for a whole regular expression dataset.

Metrics for String Matching

- Load Factor: to evaluate the memory utilization for hash-based schemes.
- Memory Size: total memory footprint for a whole string dataset.
- Memory/char: $\frac{\text{total memory}}{\text{total characters}}$ with memory in bytes, shows the effectiveness of the AC-string matching compression scheme.

3.4.2 Workloads

The workload for evaluating EffCLiP on finite automata are taken from synthetic datasets described [8, 57]. We evaluate EffCLiP for string matching on both synthetic (EM) and an actual large dataset (snort). The snort workload only contains string patterns extracted from the Snort 2.9.4.6 version [42]. For the purposes of this evaluation, we have selected 9000 distinct string patterns among over 50K patterns.

Workloads for Finite Automata Models
• spyware (462 patterns) [57]
• EM (1000 patterns) [57]
• dotstart0.05, dotstar0.1, dotstar0.2 (1000 patterns each) [57]
• range0.5, range1 (1000 patterns each) [57]
• bro217 (217 patterns) [8]
• tcp_homenet_externalnet (733 patterns) [8]

Workloads for String Matching

• EM (1000 patterns) [57]
• snort (9000 patterns) [42]

3.4.3 Concrete EffCLiP Transition Implementation

We describe a concrete EffCLiP transition representation, and later use it quantitatively evaluate our approach. We also detail how to implement DFA, A-DFA (an improved version of $D^2FA$), NFA, and Majority Compression using EffCLiP. The general view of memory layout is shown in Figure 4.11. We use a two dimensional Full Table layout for DFA as baseline and a one dimensional linear array layout for EffCLiP. The detail basic transition block is shown in Figure 3.8.

In EffCLiP, the source (17 bits) and input (8 bits) implement signature check – verifying the matches with the current state. The target (17bits) the regular transition target (for this input character). The attachment (17bits) stores an explicit additional transition address or state identifier used for majority pair, majority tail, and default transitions, etc. The info (3bits) describes the type stored in the attachment and acceptance information. For
example, if info encodes a default transition, the attachment has the address of default transition. For NFA’s, attachment is used to chain multiple active states together. Given the basic transition size of $|basic_{eff}|$, then we can write the memory size of an EffCLiP encoded FA as:

$$MEM_{eff} = |basic_{eff}| \cdot EL(M)$$

We use the Full Table implementation of DFA as baseline for both DFA and A-DFA (shown in Figure 3.7b) and NFA implementation in [57] for NFA baseline. We construct a two dimensional table $T$ for the DFA transition function $\delta$, where $|T| = |Q| \times |\Sigma|$. The $i$th entry in row $q$ denotes the target state $p$ of $q$ with the $i$th input symbol in $\Sigma$ and if $p$ is an accepting state. The basic block in Full Table uses 17 bits to denote the state identifier and 1 bit to denote the acceptance. The memory size of Full Table is given by:

$$MEM_{ft} = |basic_{ft}| \cdot |\Sigma| \cdot |Q|$$
### Table 3.3: Experiment Configurations and Workload Properties

#### 3.4.4 Workloads Configuration

Any concrete implementation has address space limits. We partition large workloads into expression sets that match the address sizes in Figure 3.8a. As many patterns as possible are grouped into each address space. In Table 3.3, we show the number of partitions, total states, and baseline memory size for experiments across DFA, A-DFA, and NFA. A-DFA workloads show additional information – # root states, # states with default transition depth to root of 1, and # states with default transition depth to root of 2. We don’t show the number of non-root states with ≥ 3 to-root depth.

In the following sections we show the evaluation result of EffCLiP with DFA and DFA under Majority Compression, A-DFA, and A-DFA under Majority Compression, and NFA.

### 3.4.5 DFA: Full Table vs EffCLiP

Using DFA Full Table implementation as baseline with a basic block size shown in Figure 3.8b, we describe both EffCLiP with a Full Table implementation and EffCLiP DFA with Majority Compression using the basic block size shown in Figure 3.8a.

For a structured DFA implementation using Full Table, first EffCLiP requires more space per transition, increasing the total memory size (see Figure 3.9). After majority compression is employed, the compression rate increases dramatically. Figure 3.9, a geometric mean of 4.65x reduction can be achieved by EffCLiP with Majority Compression across all workloads. It is noted that Bro217 only manages a 1.15 compression rate due to low levels of sparsity.
in the transition table when employing Majority Compression.

Memory utilization is shown in Figure 3.10. Employing Majority Compression Full table wastes a lot of space (96%), but EffCLiP achieves much denser packing, overall > 50% load factors. In short, EffCLiP’s flexibility enables compression that more than makes up for its larger basic transition size.

A-DFA employs default transition compression to exploit the basic DFA transition redundancy. With DFA Full Table as baseline we use default transition compression (also known
as A-DFA, $D^2 FA$) and encode it with EffCLiP transition formats. Then we repeat, adding Majority Compression.

![Figure 3.11: A-DFA Compression Rate](image)

As shown in Figure 3.11, A-DFA on EffCLiP achieves high compression. The reason for this is that default transition compression exploits transition redundancy among states and destroys the regular structure of the DFA, requiring the use of root states. As a result, EffCLiP achieves an overall 29.2x compression rate across workloads over DFA Full Table baseline. Adding Majority compression increases the achieved compression, by additional factors as high as 2.24x (dotstar0.2) and by a geometric mean of 1.55x across all benchmarks. Peak compression rates achieve surpass 40x. With the switch from default to majority compression, benefit arises mainly from shrinking of root states and by reducing transitions for non-root states. The memory utilization is shown in Figure 3.12. A-DFA with EffCLiP and A-DFA with EffCLiP and Majority Compression both achieve high memory utilization.

### 3.4.7 NFA: Full Table vs EffCLiP

Here we use the NFA implementation in [57] as baseline, comparing it to NFA using the EffCLiP transition representation. [57] implements NFA transition table as symbol-first representations: encoding transitions as a $(source, destination)$ list with a metatable of start positions for each symbol. $source$ and $destination$ are each 16 bits.

Our results show that NFA on EffCLiP achieves significantly better compression, includ-
ing 3.9x on spyware workload and over 1.85x for all workloads (see Figure 3.13). The overall compression rate improves 2.5x over [57]. Notably, an NFA has fewer transitions than the corresponding DFA, but more transition references per input symbol. An NFA has little transition redundancy, but often has irregular structure – making structured layout inefficient. As shown in Figure 3.14, NFA on EffCLiP enables dense packing with this irregular structure, achieving a high load factor (> 97%) across all datasets.

### 3.4.8 String Matching: Hash vs EffCLiP

String matching is another branch of pattern matching. Different from regular expression matching, string matching has fixed patterns (no Kleene closures and wildcards), thus hash-based encoded schemes are widely deployed. For this experiment alone, we assume the EffCLiP transition representation is 64 bits instead of the 62 bits shown in Figure 3.8a.
Figure 3.14: NFA Memory Utilization

Table 3.4: Aho-Corasick String Matching

We compare the memory cost of the EffCLiP methods in AC string matching with existing approaches (Table 3.4). For perfect hash schemes [53], EffCLiP achieves higher load factor (97%) than \((2D)^2 - \text{Hash} \ (90.9\%)\) with no need for a separate translation table, which causes additional memory references. EffCLiP is among the most compact AC-DFA/AC implementations for string matching. The AC algorithm provides an upper-bound for the mem/char of EffCLiP, which is 8 bytes per character in worst case where no redundancy in a set of string patterns can be exploited. We also evaluate our EffCLiP scheme with a much larger dataset (9000 snort rules), and it only needs 1 partition compared to CDFA [26] and B-FSM [50] schemes which require several serval partitions on much smaller datasets.

<table>
<thead>
<tr>
<th>AC types</th>
<th>AC types</th>
<th>#Rules</th>
<th># Partitions</th>
<th>Total characters</th>
<th>Load factor</th>
<th>Total memory</th>
<th>Mem/char</th>
</tr>
</thead>
<tbody>
<tr>
<td>EffCLiP</td>
<td>AC-DFA</td>
<td>1K</td>
<td>1</td>
<td>35.4K</td>
<td>0.97</td>
<td>236KB</td>
<td>6.82B</td>
</tr>
<tr>
<td>EffCLiP</td>
<td>AC-DFA</td>
<td>9K</td>
<td>1</td>
<td>312.6K</td>
<td>0.96</td>
<td>2630KB</td>
<td>6.66B</td>
</tr>
<tr>
<td>((2D)^2)-Hash [53]</td>
<td>AC</td>
<td>6.4K</td>
<td>1</td>
<td>105K</td>
<td>0.91</td>
<td>699KB</td>
<td>7.6B</td>
</tr>
<tr>
<td>((2D)^2)-Hash [53]</td>
<td>AC-DFA</td>
<td>6.4K</td>
<td>1</td>
<td>105K</td>
<td>0.91</td>
<td>767KB</td>
<td>8.33B</td>
</tr>
<tr>
<td>CDFA [26]</td>
<td>AC-DFA</td>
<td>1.8K</td>
<td>2</td>
<td>29K</td>
<td>NA</td>
<td>129KB ~ 256KB</td>
<td>4.45B ~ 8.2B</td>
</tr>
<tr>
<td>B-FSM [50]</td>
<td>AC-DFA</td>
<td>1.5K</td>
<td>4</td>
<td>25.2K</td>
<td>NA</td>
<td>188KB</td>
<td>7.4B</td>
</tr>
<tr>
<td>Bitmap [49]</td>
<td>AC</td>
<td>1.5K</td>
<td>1</td>
<td>18.2K</td>
<td>NA</td>
<td>2.8MB</td>
<td>154B</td>
</tr>
<tr>
<td>Path Compression [49]</td>
<td>AC</td>
<td>1.5K</td>
<td>1</td>
<td>18.2K</td>
<td>NA</td>
<td>1.1MB</td>
<td>60B</td>
</tr>
</tbody>
</table>
3.4.9 Majority Compression and Transition References

Implementing finite-automata usually is a memory versus transition references tradeoff (DFA vs. A-DFA vs. NFA). However, by employing Majority Compression and EffCLiP on DFA, we achieve compression (see Figure 3.9) without increasing the number of transition references per input symbol. As mentioned in earlier sections, if a state and its majority forms a majority tail it guarantees 1 transition reference per input symbol. Otherwise, one more transition reference is needed if signature check fails. In practice, the fraction of majority tails in a DFA determines the number of transition references for one input symbol. In Table 3.1 we documented the fraction of majority tails, showing that for realistic datasets over 99% states can form a majority tail with their majority.

![Figure 3.15: Breakdown for Analyzing Majority Compression Impact on A-DFA](image)

Comparing A-DFA and A-DFA EffCLiP with Majority Compression is intricate, depending on the varied performance of A-DFA and its root states as well as majority tail and majority pair. For regular transitions (signature check succeeds), it’s a single transition reference in both systems. When signature check fails, there are four different cases as shown in Figure 3.15. In case 1 (blue), it’s a root state in A-DFA that also forms a majority tail. In this case, both systems require only one transition reference, but A-DFA requires a larger memory representation (full table) for a root state. In case two (red), A-DFA requires three transition references, but A-DFA EffCLiP with Majority Compression can exploit the ma-
majority tail, so it requires only one reference. In case three (green), A-DFA has a root state, so it requires only one reference, A-DFA EffCLiP with Majority Compression will require two references. While slower, A-DFA EffCLiP with Majority Compression is still much smaller. In case four (purple), A-DFA requires three transition references, and A-DFA EffCLiP with Majority Compression only two. So, with comparable size, A-DFA EffCLiP with Majority Compression is faster. So in summary, A-DFA EffCLiP + Majority Compression outperforms A-DFA in red and purple cases, and has the same performance, but a smaller memory size in blue case. In green, A-DFA has the advantage on transition references but has larger memory size.

Notably, the fraction of all A-DFA states that use Majority Compression is $\leq 1\%$. Overall, these results show that majority tail fraction differs from $< 1\%$ ($\text{range0.5,range1}$) to $> 93\%$ (spyware, EM, bro217, tcp) across all workloads, and an average of 55% root states, which probably matter the execution speed most, form a majority tail.

![Figure 3.16: EffCLiP Memory Size Progression](image)

Figure 3.16 is a summary of EffCLiP and Majority Compression for DFA and A-DFA for dotstar0.2 workload. First moving from DFA Full Table to DFA EffCLiP increases the memory size – the cost of EffCLiP transition flexibility. But, when Majority Compression or A-DFA default compression are added, the memory size shrinks dramatically, to far smaller than the original structure. Finally combining the Majority Compression and A-DFA gives the most compact size, at similar performance.
3.5 A Summary of EffCLiP

In this chapter, we have proposed a new approach, “efficient coupled-linear packing” (EffCLiP), that optimizes both finite-automata size and performance. EffCLiP employs a novel transition representation that enables a simple addressing operator (integer addition) while providing flexibility to support transition packing. EffCLiP is a general approach, and we show how it can be used for not only DFA but also a wide range of models such as NFA, A-DFA, and Aho-Corasick multi-string matching. We observe a new structure in finite automata, called “majority tail” and document its frequency.

Based on these, we present a novel algorithm for efficient transition packing (efficient coupled-linear packing). We also present a new finite-automata optimization algorithms “majority compression” that exploits the majority tail phenomena. Evaluation across a variety of widely-used finite automata workloads and models show that EffCLiP achieves compression rates as high as 8.4x and packing densities of 77% for DFA, 2.24x and >99% for A-DFA, 3.9x and >99% for NFA while supporting high speed and simple implementation.
CHAPTER 4

UAP: THE UNIFIED AUTOMATA PROCESSOR

EffCLiP enables high density FA transition packing, fast transition function implementation, flexible renaming suit for any FA models’ topologies, and almost always 1 memory reference doing 1 FA transition (except some corner cases). In general, EffCLiP balances the memory layout and hardware complexity to achieve, uses simple and fixed functionality hardware, keeps as few memory reference as possible, and maintains high utilization of slots address space. However, EffCLiP doesn’t model complex operational behavior in extended-FA models, ignores the physical memory slot size (bits) and how to partition each EffCLiP field in a slot.

The UAP architecture realizes the EffCLiP the FA abstract model. UAP addresses how to represent and implement actions, how to represent and implement transitions of different complexity, how to implement acceptance with associated actions (see Section 4.2.3).

UAP further provides an architecture to address how to implement FA execution, how to interface with the core processor, how to boost performance of throughput and line rate, how to integrate with large software system, how to mitigate the impact of multiple same concurrent active states (see Section 4.2.4).

To realizes the EffCLiP abstract model, we had a careful analysis of extended-FA models, and a set of actions (detail see Section 4.2.3) are proposed to associate with the EffCLiP transition to emulate the complex operational transitions in an extended-FA model with high efficiency. Figure 4.1 simply reflects this intuition. Actions are further packed in the EffCLiP memory model and are pointed by the EffCLiP transition primitives. UAP adopts EffCLiP transition representation to design UAP transition primitive and inheriting EffCLiP’s linear packing algorithm and further set the field size and memory slot (or memory word, in UAP term) size in bits. This helps a physical implementation of the EffCLiP memory model and transition model. For example, in UAP representation, each slot in EffCLiP becomes
a concrete memory word (32 bits). UAP representation inherits “signature” and “target” fields in EffCLiP, and has “type” and “attach” field. The “attach” field can act as an offset pointing to the associated actions and also serves other various purposes. UAP also creates a memory layout integrating UAP actions and UAP transition primitives by expanding the usage of an EffCLiP transition field and adjusting the packing algorithm. One thing worths pointing out is UAP inherits EffCLiP’s two transition primitives: symbol-consuming and non-symbol consuming (base and teleport in UAP terms), and combines these two primitives creating a third transition primitive: conditional. The conditional primitive is mainly for the performance of FA models that have conditional transitions, such as counting-DFA, counting-NFA, JFA, JSON-NFA, etc.

![Diagram of FA model with actions on transitions](image)

**Figure 4.1:** UAP using actions to emulate complex FA models’ transition operations

To boost performance, a bunch of architectural techniques are used in the design. UAP architecture applies vector-parallelism, multi-step execution, smart completion detection, acceptance detection, stream sharing and combing features. UAP architecture adds delayed acceptance detection feature which greatly reduces the overhead of acceptance check for each input symbol, combining feature ensuring no concurrent identical active states, lane’s completion detection feature which allowing multiple lanes operated asynchronously, and input stream sharing which allows flexible pattern aggregation or producing aggressive throughput. These features either improve UAP’s performance, flexibility, or scalability.
UAP provides a set of instructions with vector extensions to allow easy interface with the core processor. UAP directly operates on a vector register file and a bunch of local memory banks, which are very popular in current chip design. Thus, UAP operations can be efficiently combined with traditional computing software. At the micro-architecture level, UAP integrates a small set of new functions for finite-automata actions. These functions, combined with a local memory, enable UAP to support a wide range of finite automata models, including traditional non-deterministic and deterministic finite automata (NFAs and DFAs, respectively) and popular models that have been proposed in the networking domain (A-DFAs [7], JFAs [58], counting-DFAs [6], and counting-NFA [6]). UAP reuses the vector register files and local memory banks of 10x10 architecture and adds special simple sequence logic. In section 4.4 and 4.5.3, we describe the detail UAP implementation addresses the detail hardware design choice of UAP which includes memory bank size, how many UAP engines per memory bank, how many contiguous UAP words to fetch at a time, and combining queue implementation. These implementation choices helps UAP achieving high clock rate and low power consumption.

4.1 Importance of FA Diversity

While each automata model gives benefits for a range of FA structures, none is best for all regular expression patterns. As shown in Figure 4.2, A-DFA, JFA, and c-DFA respectively achieve highest pattern density for some patterns in representative workloads (see Section 4.3 for full workload descriptions). For example, if no patterns cause state explosion, large pattern sets can be encoded in a moderate size DFA (comparable to the equivalent NFA). In such cases, the DFA can be the best model. On the other hand, if patterns have wildcard repetitions, DFAs may suffer from exponential state explosion, making even a single DFA infeasibly large. In such cases, extended models that mitigate the state explosion (e.g. hybrid-FA, history-based FA, XFA and JFA) can produce an automaton that is compact
with better properties than the equivalent NFA. Unfortunately in some cases, these extended FA’s [6,58], cannot represent all patterns. Thus, each automata model has its set of strengths and weaknesses.

(a) DFA Models
(b) NFA Models

Figure 4.2: # patterns in 256KB, selected representative workloads.

4.2 UAP Design

4.2.1 Requirements for General-Purpose

In the previous chapter, we introduced “EffCLiP” which maps the high level automata topology to abstract memory with high space and performance efficiency. Next we are going to further turn the abstract memory representation into concrete instruction set and hardware architecture design. We analyze the mechanism requirements for a general-purpose finite automata engine (see Figure 4.3). First, conditional transitions are common to counting-FAs and history-based-FAs. Second, conditional actions are required for XFAs and JFAs. Third, the immediate state shift functionality is common to $D^2$FAs, A-DFAs, JFAs, SFAs, and dual-FAs. NFAs, counting-NFAs, hybrid-FAs, SFAs and dual-FAs allow multiple state activations. Our analysis shows that by covering these five elements efficiently, a system
might implement these twelve models and many more published FA models.

Figure 4.3: A small set of mechanisms can support diverse finite automata models; six models used for evaluation are in green.

4.2.2 Key Design Elements

The key design elements include how to implement UAP transitions (transition primitives and actions), how to interface with the base core processor, and what is the execution model, and how to keep the UAP state. With these insights, the UAP implements transitions by composing two parts: 1) a transition primitive and 2) a set of finite-automata actions into each UAP transition. The transition primitives cover the full space of existing models, and the actions support the varied transition work (e.g. counting). In terms of the execution
model, the UAP executes one transition primitive or one action per cycle for each UAP lane. UAP transitions are implemented by a programmable UAP lane, accomplishing in a few cycles work that requires 20-50 instructions in a traditional CPU core. To integrate with the CPU core, local memory banks and a vector register file are shared. UAP lanes are composed with banks of an on-chip local memory,\textsuperscript{1} and efficiently mated to a SIMD register file in a RISC core.

4.2.3 Transitions and Memory Layout

Transition Primitives and Actions: EffCLiP enables dense memory utilization and a simple hash function – integer addition (see Base below) that enables fast clock rate. EffCLiP effectively achieves a “perfect hash”. UAP further realizes EffCLiP abstract model. The overall FA is a set of UAP transitions laid out in memory, using the EffCLiP algorithm that combines a signature for placement flexibility with coupled-linear packing. Each UAP transition consists of a transition primitive associated with zero or more actions (Figure 4.4a).\textsuperscript{2} FA acceptance information is located at the end of the address space (Figure 4.4b). Detail action set is listed in the end of this section.

UAP supports three transition primitives: base, teleport, and conditional (Figure 4.5). These primitives are sufficient to efficiently implement our six FA exemplars, all finite automata models of which we are aware, and can also allow creating new FA models. In short, they support flexible programmability. Each transition primitive chooses the next state based on the input symbol, transition primitive data, and UAP register state. The primitives are depicted in Figure 4.5, and define below how each computes the transition target address:

- **Base** target address (TA) is the sum of current state address and input symbol (perfect

\textsuperscript{1} The local memory can be shared with other compute units on chip, as on many systems-on-chip (SoC’s).

\textsuperscript{2} In some cases, there may be a gap between the transition primitive and associated actions. The actions are always contiguous.
Transitions with distinct source state and symbol can be addressed with **Base** transition.

- **Teleport** target address (TA) is the data stored in “Attach” field or “Action” (see **Encodings**). Non-consuming transitions like epsilon transitions in NFA, default transitions in A-DFA, and jump transitions in JFA are implemented with **Teleport**.

- **Conditional** uses a value in UAP Reg 1 as a flag to select a target address (TA). If the flag is TRUE, then the target address is computed as in **Base**; if FALSE, the “Attach” field of the computed **Base** primitive is used as the target address. For example, counting -DFA and counting-NFA employ **Conditional** transition primitives.

UAP transitions can add actions to a transition primitive, thereby implementing the **hash function**. (a) Sample UAP transitions (b) Sample UAP Memory Map

**Figure 4.4: UAP Transitions and Memory Layout**

**Figure 4.5: UAP Transition Primitives**

TA = (s0.target + ‘c’)  TA = (s0.attach)  TA = r1?(s0.target+’c’):(((s0.target+’c’).attach)

hash function). Transitions with distinct source state and symbol can be addressed with **Base** transition.

- **Teleport** target address (TA) is the data stored in “Attach” field or “Action” (see **Encodings**). Non-consuming transitions like epsilon transitions in NFA, default transitions in A-DFA, and jump transitions in JFA are implemented with **Teleport**.

- **Conditional** uses a value in UAP Reg 1 as a flag to select a target address (TA). If the flag is TRUE, then the target address is computed as in **Base**; if FALSE, the “Attach” field of the computed **Base** primitive is used as the target address. For example, counting -DFA and counting-NFA employ **Conditional** transition primitives.

UAP transitions can add actions to a transition primitive, thereby implementing the
additional functionality required in more complex finite automata models. UAP actions implement a basic set of operations including simple arithmetic and logical operations, predicate evaluation, and data movement amongst input register, UAP registers, and local memory.

**Encodings:** Detailed encoding of transition primitives and actions is shown in Table 4.1. All transition primitives and actions are 32 bits each. *Input* is a signature used to check if the transition is correct (hash match). *Target* encodes a target state address. *Attach* encodes additional information for programmability across automata. *Type* dictates usage of *Attach* (as in Table 4.2). *Type* values of 0-3 encode behaviors sufficient for the six exemplar FA models and *Type* 4-7 encode optimization to increase transition performance.

In the normal cases, transitions are explicitly stored, so the signature check verifies the fetched state. To increase encoding density, we employ *majority transition* optimization using a single representative transition for a set of transitions from a given state to the same target (the *majority transition*). For such a grouped transition the signature check must fail, so the *Attach* field supplies the address of the majority transition (Type 4). The *Types* 5, 6, 7 optimize (single memory access/transition) for majority transition, default transition, and epsilon transition. For these, *Attach* serves as the state identifier instead of the address of that state. Here also a single memory access per transition can be achieved.

<table>
<thead>
<tr>
<th>Input(8)</th>
<th>Target(12)</th>
<th>Type(4)</th>
<th>Attach(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused(7)</td>
<td>Opcode(4)</td>
<td>DstReg(2)</td>
<td>SrcReg(2)</td>
</tr>
</tbody>
</table>

Table 4.1: Transition Primitive and Action Formats

The architecture state of a single UAP lane consists of a input register pointer (Riptr) holding the offset of input stream (8bit), 4 UAP General-Purpose registers (r0,r1,r2,r3, 32 bit each), 1 UAP transition register (TranReg, 32 bit) holding the fetched transition that relates to the current state, a 16-entry combining queue (32 bits each entry) holding multiple current active states and multiple next states, a accept information region register (offset the region in local memory, 16 bits), a configuration register (32 bits), and a single R/W
<table>
<thead>
<tr>
<th>Type</th>
<th>Transition Primitive</th>
<th>FA Term</th>
<th>Attach Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Teleport</td>
<td>Default Transition</td>
<td>Default state address</td>
</tr>
<tr>
<td>1</td>
<td>Teleport</td>
<td>Epsilon Transition</td>
<td>Target state address</td>
</tr>
<tr>
<td>2</td>
<td>All three</td>
<td>Transition with Action</td>
<td>1st Action offset</td>
</tr>
<tr>
<td>3</td>
<td>Conditional</td>
<td>Cond. Transition</td>
<td>Re-target state address</td>
</tr>
<tr>
<td>4</td>
<td>Teleport</td>
<td>Majority Transition</td>
<td>= low order 9 bits of target state address</td>
</tr>
<tr>
<td>5</td>
<td>Teleport</td>
<td>Majority Transition</td>
<td>low order 9 bits of target state identifier (immediate)</td>
</tr>
<tr>
<td>6</td>
<td>Teleport</td>
<td>Default Transition</td>
<td>Default state identifier (immediate)</td>
</tr>
<tr>
<td>7</td>
<td>Teleport</td>
<td>Epsilon Transition</td>
<td>Target state identifier (immediate)</td>
</tr>
</tbody>
</table>

Table 4.2: **Type** Field and Transition Primitives

Port local memory bank which stores the automaton transition primitives, actions, variables, and acceptance information. We use an "assembly like language" to describe UAP words including transition primitive and actions in Table 4.4. Table 4.3 describe the way UAP engine identifies three types of transition primitives. Assume the current transition is Tc and fetched transition is Tf. For the teleport transition primitive, if the transition is directly through Tc.attach rather than referencing the local memory, the current state needs to be terminated if a signature check fails again due to there is no more attach field available for the primitive. Any primitive fetched that doesn’t meet the requirements below must cause termination of the current state. Only primitives that explicitly stored in the local memory can have actions. For example the accept is represented as an “addition” action, a “movIn2Reg” action, and a “pack” action.
### Table 4.3: How to identify transition primitive’s type?

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>if $T_f$.input = Stream[Riptr] and $T_f$.type $\neq$ 3 then $T_f$ is Base</td>
</tr>
<tr>
<td>Teleport</td>
<td>$T_f$.input $\neq$ Stream[Riptr], then LM[$T_c$.attach] (Tc.type=0,4) or Tc.attach (Tc.type=5,6) is Teleport; $T_f$.input = Stream[Riptr]. $T_f$.type = 1 or 7 then LM[$T_f$.attach] or $T_f$.attach is Teleport</td>
</tr>
<tr>
<td>Conditional</td>
<td>$T_f$.input = Stream[Riptr] and $T_f$.type = 3, then $T_f$ is Conditional</td>
</tr>
</tbody>
</table>

**4.2.4 Integrating UAP Computation**

Finite automata computation performed on the UAP can be easily integrated into a larger software program in a fashion similar to current finite automata or regular expression libraries. These systems typically require expressions/patterns to be compiled, and then higher-level software invokes the matching function. In similar fashion, finite automata are compiled by the UAP compiler, and then loaded in the local memory (accessible by both the CPU and UAP). The starting states for each FA are loaded into the UAP by writing the state registers. Then, as shown below, a basic loop of load stream data, process FA transitions, then check for completion ensues. These operations are described in greater detail in Section 4.2.5.

```c
write_state_regs(initial states, memory offsets, lane mask, sharing mode);
vec64 = 0;
while (all_zeros(vec64)){
    ...load stream data into vector registers...
    traverse(0, 256, vec64);
    complete_traverse(); }
```

55
<table>
<thead>
<tr>
<th>UAP word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base $T_b$</td>
<td>TranReg $\leftarrow$ Tb, Riptr $\leftarrow$&lt;$Riptr$&gt; + 1 if Tb.attach $\neq$ 0; else finish the actions with offset attach</td>
</tr>
<tr>
<td>Teleport $T_t$</td>
<td>TranReg $\leftarrow$ Tt, Riptr $\leftarrow$&lt;$Riptr$&gt; + 1 if Tt.type $\neq$ 0; else finish the actions with offset attach</td>
</tr>
<tr>
<td>Condition $T_o$</td>
<td>TranReg $\leftarrow$ To if r0 = 1, else TranReg $\leftarrow$ LM[Tc.target + Stream[Riptr] +1] (the UAP word right after To),Riptr $\leftarrow$&lt;$Riptr$&gt; + 1 if TranReg.attach $\neq$ 0; else, finish the actions</td>
</tr>
</tbody>
</table>

add Rd, Rs $\quad$ Rd $\leftarrow$ <Rd> + <Rs>  
addi Rd, imm $\quad$ Rd $\leftarrow$ <Rd> + imm  
sub Rd, Rs $\quad$ Rd $\leftarrow$ <Rd> - <Rs>  
subi Rd, imm $\quad$ Rd $\leftarrow$ <Rd> - imm  
and Rd, imm $\quad$ Rd $\leftarrow$ <Rd> and imm  
or Rd, imm $\quad$ Rd $\leftarrow$ <Rd> or imm  
bani Rd, imm $\quad$ Rd $\leftarrow$ <Rd> bitwise and imm  
bori Rd, imm $\quad$ Rd $\leftarrow$ <Rd> bitwise or imm  
lt Rd, Rs, imm $\quad$ Rd $\leftarrow$ (<Rs> < imm) ?  
eq Rd, Rs, imm $\quad$ Rd $\leftarrow$ (<Rs> == imm) ?  
gt Rd, Rs, imm $\quad$ Rd $\leftarrow$ (<Rs> > imm) ?  
mov Rd, Rs $\quad$ Rd $\leftarrow$ Rs  
movi Rd, imm $\quad$ Rd $\leftarrow$ imm  
movIn2Reg Rd $\quad$ Rd $\leftarrow$ Stream[Riptr]  
movLM2Reg Rd, Rs $\quad$ Rd $\leftarrow$ LM[Rs]  
movReg2LM Rd, Rs $\quad$ LM[Rd] $\leftarrow$ <Rs>  
movLM2TranReg imm $\quad$ TranReg $\leftarrow$ LM[imm]  
pack Rd, Rs $\quad$ Rd $\leftarrow$ (<Rd> right shift 16 + <Rs>)  
accept imm $\quad$ LM[acceptInfo+Riptr] $\leftarrow$ imm right shift 16 + <Riptr> |

| Table 4.4: Assembly Definition of UAP Words (Primitives and Actions) |

### 4.2.5 Instruction Interface to UAP

Instructions to program the UAP are listed below; each is a single instruction that can be conveniently accessed by the indicated intrinsic function. `read/write_state_regs` instructions use vector registers to configure UAP lanes and move states to (launch) and from (complete) the UAP. The `traverse` instruction performs FA transitions. `complete_traverse` synchronizes lane completion. `read/write_combine.Q` and `read/write.reg` instructions expose combining queues and UAP Registers to programmers, enabling UAP context-switch. The overall
description is in the table and more detail can be found below.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>write_state_regs(config1, config2)</code></td>
<td>Write UAP states with values from two 2048-bit vector registers, including <code>&lt;initial state, accept base, lane mask, vector sharing&gt;</code> for the 64 lanes. Initial state specifies the first state for the FA’s mapped to the lane; accept base is the base addresses for acceptance rules, lane mask indicates the lanes to activate, vector sharing specifies one of the four static associations between vector registers and UAP lanes.</td>
</tr>
<tr>
<td><code>read_state_regs(config1, config2)</code></td>
<td>Read out UAP states <code>&lt;current state, accept base, lane mask, vector sharing&gt;</code> into two 2048-bit vector registers.</td>
</tr>
<tr>
<td><code>traverse(start, steps, acceptReg)</code></td>
<td>Traverse FA for all activating lanes, Start: offset within the vector register; Steps: 1-256 (how many traversal steps to execute before stop); AcceptReg: #accepts for each lane, 64x32bit vector.</td>
</tr>
<tr>
<td><code>complete_traverse()</code></td>
<td>Block until all traverse steps complete.</td>
</tr>
<tr>
<td><code>read_combine_Q(lane, entry, reg), write_combine_Q(lane, entry, reg)</code></td>
<td>lane #, entry #, the value is extracted from (written to) the combining queue and deposited into (sourced from) a 32-bit register.</td>
</tr>
<tr>
<td><code>read_reg(lane, UAPreg, toReg), write_reg(lane, UAPreg, fromReg)</code></td>
<td>lane #, the value is extracted from (written to) the UAP lane register, 16 bits, so it is deposited into (sourced from) a 32-bit general purpose register.</td>
</tr>
</tbody>
</table>

Table 4.5: Overall Instructions to access UAP Functionality

**Instruction: write_state_reg**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>write_state_reg VecReg1</code></td>
<td>Configure UAP based on the vector register VecReg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14 bits</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>0</td>
<td>VecReg1</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description:**

Write UAP states with values from one 2048-bit vector register (VecReg1), including `<initial state, accept base, lane mask, vector sharing>` for the 64 lanes. Initial state (12 bits for each lane) specifies the first state for the FA’s mapped to the lane; accept base is the base addresses for acceptance rules (12s bit for each lane), lane mask (1 bit for each
lane) indicates the lanes to activate, vector sharing (total 2 bits) defines the association of a stream with (1, 16, 32, or 64 lanes). Lane mask is 64 bits, initial state is 12x64 bits, accept is 12x64 bits, vector sharing is 2 bit. They are packed in a vector register.

**Instruction: read_state_reg**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>read_state_reg VecReg1</td>
<td>Get UAP configuration information from VecReg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 bits</td>
<td>6 bits</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VecReg1</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description:**

Return UAP status information in VecReg1: lane mask, initial states, vector sharing, and accept base into a vector register

**Instruction: traverse**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>traverse GPReg1 GPReg2 VecReg3</td>
<td>process UAP transition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 bits</td>
<td>6 bits</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>GPReg1</td>
<td>GPReg2</td>
<td>VecReg3</td>
</tr>
</tbody>
</table>

**Description:**

Traverse FA for all activating lanes, GPReg1: offset within the vector register, the beginning position to process the input stream; GPReg2: 1-256 (how many traversal steps
to execute before stop for all active UAP lanes); VecReg1 for each lane, 64x32bit vector, returning the brief accept information (more detail matching information is stored in the local memory indicated by the accept base). This instruction is major function for automata processing.

**Instruction: complete_traverse**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>complete_traverse GPReg1</td>
<td>UAP lanes completion detection return to GPReg1</td>
</tr>
</tbody>
</table>

**Description:**

For a single `traverse` instruction, UAP lanes run concurrently. Block until all traverse steps complete for all lanes. In this way, though each lane running different automata has different completion time, UAP lanes synchronize every `traverse` instruction. Final result returns to GPReg1, 0 means at least 1 lane not complete, 1 means all lanes are completed.

**Instruction: read_combine_Q**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>read_combine_Q GPReg1 GPReg2 GPReg3</td>
<td>Get data stored in the UAP queue</td>
</tr>
</tbody>
</table>

**Description:**

For a single `traverse` instruction, UAP lanes run concurrently. Block until all traverse steps complete for all lanes. In this way, though each lane running different automata has different completion time, UAP lanes synchronize every `traverse` instruction. Final result returns to GPReg1, 0 means at least 1 lane not complete, 1 means all lanes are completed.
Description:

GPReg1 specifies the lane ID, and GPReg2 specifies the entry ID of the combining queue of the lane. The value is extracted from the combining queue and deposited into a 32-bit general purpose register indicated by GPReg3.

**Instruction: write_combine_Q**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>write_combine_Q GPReg1 GPReg2 GPReg3</td>
<td>Write data in the UAP queue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 bits</td>
<td>6 bits</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>GPReg1</td>
<td>GPReg2</td>
<td>GPReg3</td>
</tr>
</tbody>
</table>

Description:

GPReg1 specifies the lane ID, and GPReg2 specifies the entry ID of the combining queue of the lane. The value is written into the combining queue entry from a 32-bit general purpose register indicated by GPReg3.

**Instruction: reade_uap_reg**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>read_uap_reg GPReg1 GPReg2 GPReg3</td>
<td>Get UAP Reg state return to GPReg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 bits</td>
<td>6 bits</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>GPReg1</td>
<td>GPReg2</td>
<td>GPReg3</td>
</tr>
</tbody>
</table>

Description: UAP return the UAP register state in a general-purpose register GPReg1. UAP lane is specified in GPReg2, and UAP register ID is specified in GPReg3.

**Instruction: write_uap_reg**
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>write_uap_reg GPReg1 GPReg2 GPReg3</td>
<td>set UAP Reg state from GPReg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 bits</td>
<td>6 bits</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>7</td>
<td>GPReg1</td>
<td>GPReg2</td>
<td>GPReg3</td>
</tr>
</tbody>
</table>

**Description:** UAP writes the UAP register state from a general-purpose register GPReg1. UAP lane is specified in GPReg2, and UAP register ID is specified in GPReg3.

The UAP *traverse* operation employs efficient transition logic to achieve rates up to one transition per clock cycle (DFA). UAP adds two dimensions of transition-level parallelism to exceed one transition per cycle (see Figure 4.6). First, UAP supports multi-step operations, allowing a finite automaton to progress up to 256 steps, consuming 256 input symbols in a single instruction. Second, UAP supports vector operations, allowing up to 64 transitions per clock cycle against one to 64 distinct input streams. The number of traversal steps, lanes and streams are configured upon invocation of *traverse*. Together, these two dimensions enable a single *traverse* instruction to process as many as 16,384 transitions.\(^3\) The UAP assumes

---

3. 24 instructions/transition is typical for conventional architectures \[9\], so UAP can replace over 300,000 transitions
basic vector operations for loading data and computation.

\textbf{4.2.6 UAP Flexibility and Programmability}

UAP’s single transition format unifies the functionality of all six exemplar finite-automata models, the twelve examples, and, we believe, a “general-purpose” range of future potential finite-automata models. Thus, UAP can support FA model innovation at the software level with high efficiency – obviating the need for new custom hardware implementation for each model.

The UAP’s design not only supports the full range of FA models, but its unified format allows hybridization on a per-transition basis. That is, a new “unified FA” could be defined and used that includes unique transitions from a dozen different FA models – where most useful to implement an expression or structure – all integrated into a single UAP FA. Directly, the UAP enables unification of all of the different FA models at the level of individual transitions and states.

As a demonstration of UAP’s programmability, we show that the UAP design can support newly invented automata models with high efficiency. For example, actions such as “move symbol to Reg” can be used to construct a JSON-query like finite automaton (see Figure 4.7a) that not only matches keywords but also evaluates predicates along the matching process. Another example is extending c-DFA to implement a novel RLE-DFA (Figure 4.7b) which has 26 letters as alphabet (‘a’-'z’). Yet another example we implemented is A-JFA, which adopts A-DFA’s default state property and JFA’s jumping property achieving both density and state explosion elimination. The excellent performance of all three novel automata is documented in Section 4.4.

\begin{itemize}
\item Instructions with a single \texttt{traverse} instruction.
\end{itemize}

62
(a) A flat-JSON-NFA matches pattern: \{"AGE":x,...\}, where x \geq 45 (red are conditional)

(b) An 'a-z' RLE-DFA

Figure 4.7: Novel Automata on UAP

4.2.7 UAP Micro-architecture

The key elements of the UAP micro-architecture include the design of the UAP lane to efficiently support FA models, the pairing of parallel UAP lanes with banks of local memory, and the use of vectors to efficiently manage parallel UAP lanes. The critical performance aspects of the design include:

1. The UAP lane design achieves single cycle operation for each UAP transition primitive or action. For simple automata models, a UAP lane achieves one transition per cycle.
Complex models may require additional cycles per transition. The UAP lane design is described in detail in Section 4.5.3.

2. Each UAP lane is associated with one bank of an on-chip local memory [33, 52] that provides fast access for transition primitives and actions (32 bits each). The UAP lane and memory bank access latency determine the UAP’s cycle time.

3. UAP uses vectors to manage 64 UAP lanes (see Figure 4.8). Vector registers are used to initialize lanes and test for acceptance in parallel. They are also used for input/stream data and can share input data flexibly across lanes.

The 64 local memory banks, each 32-bit wide, delivers 256 bytes of memory bandwidth per cycle – the same required for a regular, single-cycle vector register load. Total local memory capacity is 1 megabyte. Stream data is moved from DRAM to local memory with an efficient on-chip data-mover/DMA [14], and then loaded into vector registers. UAP implementation power and size is discussed in Section 4.5.3.

4.3 Methodology

4.3.1 Workloads and Software

Each workload consists of a set of patterns and input streams of symbols. We compile the patterns to UAP format and use either single or multiple input streams as appropriate. Workloads are selected to stress the unique features of specific finite automata models, and whenever possible, we use workloads that allow direct comparisons with published results [31,34,42,57]. These workloads are summarized in Table 4.6.

Specifically, we use the Snort synthetic [8, 57] and PowerEN [31] dataset to evaluate UAP’s performance on regular expression patterns. In Snort workloads, all patterns in range1 contain character class, 90% patterns in dotstar0.9 contain unbounded repetitions
of wildcards (aka "dot-star" terms), *spyware* is a real Snort dataset with 462 patterns and *EM* contains string patterns. Each synthetic dataset consists of 1,000 regular expressions (except for *spyware*). *Scope* is a proprietary dataset consisting of trace data from analog-to-digital converters, in oscilloscopes, stressing the string counting constraint feature. Wildcard followed by counters is often used to express the string distance. We pick 64 patterns containing large wildcard counting constraint (*wildcnt*) from Snort v2.9 [42]. We set the *Scope* and *wildcnt* trace such that 5%, 25%, 50% of the traffic triggers counting action. Huffman [34] consists of 34 Huffman decoding trees derived from the 34 most downloaded books (as of July 4th, 2013) from the Project Gutenberg. We used the same traces as [34] for Huffman decompression.

RLE is a run-length encoding automata with a 26 counting states (see Figure 4.7b). The synthetic trace used for RLE has an average contiguous repeated symbol length of 4, 8, 16. For flat-JSON-NFA, we set value X in the select query to let the record matching rate as 5%, 20%, and 50%. The trace used in the query is the *nation* table from TPC-H [2] in JSON format. The traces for PowerEN datasets were from [31], where each input stream is 1000 symbols. The input streams used for Snort workloads were synthetically generated using the tool described in [8]. In the generation, we set the probability of in-depth transition in the set as 0.35. Each input stream (trace) has a 65,536 symbols.

Compilation for the six exemplar finite automata model is achieved by adapting open-source software [8, 16] to generate the required UAP memory layout (see Section 4.2.3). We also extended the software to generate the three novel FAs (A-JFA, RLE-NFA, flat-JSON-NFA). In all cases where a set of patterns is too large to fit in one local memory bank, the set is split over multiple banks. If there are not enough patterns to fill all 64 banks, we replicate the finite automata, providing work for all 64 lanes.
<table>
<thead>
<tr>
<th>FA Model</th>
<th>Application Area</th>
<th>Pattern Sets</th>
<th>Pattern Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA [22]</td>
<td>Decompression, Network monitoring</td>
<td>Huffman(Microsoft), EM, range1 [57]; simple [31]</td>
<td>No state explosion</td>
</tr>
<tr>
<td>NFA [22]</td>
<td>Network monitoring</td>
<td>spyware, EM, range1.dotstar0.9 [57]; simple,complx [31]</td>
<td>Any regular expressions</td>
</tr>
<tr>
<td>JFA [58]</td>
<td>Network monitoring</td>
<td>spyware, dotstar0.9 [57]; complx [31]</td>
<td>State explosion</td>
</tr>
<tr>
<td>A-JFA</td>
<td>Network monitoring</td>
<td>spyware, dotstar0.9 [57]; complx [31]</td>
<td>State explosion</td>
</tr>
<tr>
<td>RLE-DFA</td>
<td>Compression</td>
<td>Run Length Encoding (RLE) Tree</td>
<td>Encoding tree</td>
</tr>
<tr>
<td>Flat-JSON-NFA</td>
<td>Database query</td>
<td>SELECT * FROM nation WHERE n_nationkey &lt; X</td>
<td>Context sensitive</td>
</tr>
</tbody>
</table>

Table 4.6: Workload (Pattern Sets) used for Evaluation

### 4.3.2 Architecture Metrics

We use stream and system metrics as defined below:

- **Line Rate (symbol/cycle):** the input symbol rate that can be achieved on a single stream;
- **Patterns / Kilobyte:** the number of expressions divided by their size in a finite automata model in local memory (a measure of encoding density);
- **Instructions/Transition:** number of instructions executed, divided by the number of transitions implemented (a measure of architecture efficiency);

### 4.3.3 Implementation Metrics

- **System Throughput (Gbps):** aggregate bit rate across several streams. This unified metric considers both size and speed properties of FA models;
- **Area (mm²):** Si area, 32nm TSMC CMOS process;
- **Clock Speed (Ghz):** max speed of the UAP system;
• System Power (mW): all power on-chip due to UAP activities – see components in Table 4.7;

4.3.4 System Modeling and Configuration

The UAP is designed and integrated with RISC processor, vector register file using high-level description language (LISA) in Synopsis Processor Design (PD) flow. We extended PD flow and processor simulation to incorporate integrated simulation of 2-level cache hierarchy, multi-bank local memory and off-chip memory systems (either DDR3 or HMC). For system modeling, we estimate cache power and timing using Cacti 6.5 [1] and using DRAMSim2 [43] to model DDR3 memory system. In addition, we developed in-house model for HMC memory stacks, validating its timing and energy efficiency against the HMC specification. All the system components including UAP are integrated into a cycle-accurate performance and detailed energy model. System configuration includes the RISC Core, the UAP, and a 64-bank, 1MB scratchpad memory.

4.4 Architecture Efficiency

4.4.1 Line rate and Instruction count

We evaluate UAP on our six exemplar and new FA models on a variety of workloads, reporting line rate performance (see Figure 4.9). We include maximum achievable rate of one symbol/cycle for comparison. Starting from top-left with DFA, UAP consistently achieves a line rate of 0.945 symbols/cycle, losing only a few percent to UAP setup and completion detection overhead. A-DFA is a more complicated model, so each transition requires several sequential references (non-symbol consuming default transition), producing a lower line rate, ranging from 0.47 to 0.75 symbols/cycle for different patterns. We see that UAP achieves high line rates $> 0.9$ symbols/cycle consistently for JFA; efficiently delivering its benefits.
in mitigating state explosion. A-JFA, a new model, adds default states, and as a result, achieves lower line rate. Counting-DFA achieves 0.49 to 0.87 symbols/cycle with a range of 50% down to 5% of input symbols requiring counting actions. As the frequency of counting actions decreases, line rate increases. c-NFA exhibits a similar same trend as c-DFA but at slightly lower values due to c-NFA’s concurrent active states. UAP achieves NFA line rates from 0.27 to 0.75 symbols/cycle for varied patterns and traffic.

For our novel RLE-DFA and flat-JSON-NFA models (see bottom and right), UAP shows its programmability and high efficiency. For these new models, the delivered performance is dramatically higher than software on CPU/GPU (100x or more), and comparable to that for known FA models. For RLE-DFA, line rates of nearly 0.47 symbols/cycle can be achieved. There are many actions in the RLE benchmark, the processing of those explains the distance from maximum achievable. The data for the novel flat-JSON-NFA is shown in the last subfigure, where flat-JSON-NFA delivers over 0.6 symbols/cycle line rate on a select query on the nation table. The lower the matching fraction, the more likely that only 1 state is active, improving line rate. So, as matching rate is decreased, line rate increases. As with RLE-DFA, performance on flat-JSON-NFA is orders of magnitude faster than software on traditional CPU/GPU.

Architecture efficiency can also be measured by number of instructions per transition. Typical RISC and x86 CPU DFA implementations average 24 instructions per transition [9]. UAP’s vector and multi-step parallelism reduce this number dramatically to below one ten-thousandth (1/10000th) of an instruction per transition (see Figure 4.10). This improvement is broad, reflecting the general-purpose capability of UAP; it is achieved for all of the exemplar FA models, and the three new FA models.

Due to their unpredictable memory references, a key limit for efficient FA processing is on-chip memory bandwidth. To evaluate the UAP lane design, we consider increasing memory width and evaluate performance, power efficiency, and memory efficiency. We con-
sider UAP lane designs that can fetch a single word (transition primitive or action), two words (transition primitive + action, two actions), and three words (transition primitive + 2 actions, three actions). We call these designs 1-wide (the base UAP design), 2-wide, and 3-wide respectively.

Figure 4.9: Line Rate vs FA Model

Figure 4.10: Instructions per Finite Automata Transition vs. Lanes and Multi-step
4.4.2 Memory Width and Performance

First, we explore memory width impact on line rate (see Figure 4.11a). Wider memory only benefits finite automata models that use many actions, accruing benefits only for c-DFA, c-NFA, RLE-DFA. Second, we consider the power efficiency (line rate/power) of wider memory UAP designs (Figure 4.11b). Aggressively fetching from a wider memory significantly increases power without any performance benefit for most of the FA models, so power efficiency decreases with memory width – sometimes quite dramatically. The reason for this is shown clearly in Figure 4.11c; most of the additional memory fetches are wasted. UAP effectively encodes the data most likely to be needed into a single word, so the 1-wide memory delivers both high performance and good power efficiency. In all other parts of the paper, we discuss only the 1-wide UAP design.

![Figure 4.11: Line Rate, Line Rate/Power, and Bandwidth Utilization with Local Memory Width Increase](image)

4.4.3 Transition Packing (Compression)

Because encoding density is important, we compare UAP [29] and RegX transition packing. UAP is a general-purpose FA engine, effective density (patterns/kilobyte) can exploit both choice of FA model as well as UAP transition packing. PowerEN’s RegX accelerator [31] uses the BFSM algorithm [50] that exploits aggressive transition redundancy removal for DFAs.

Figure 4.12 shows results for the simple400 and complx200 pattern sets. For DFA (far left), UAP’s pattern density is low, with RegX 2 to 3x greater. However, for the same
patterns, UAP’s ability to support advanced FA models (A-DFA, A-JFA, and NFA) nets a greater than 8x better pattern density.

![Figure 4.12: Density: UAP vs RegX](image)

4.5 Implementation Efficiency

4.5.1 System Throughput

![Figure 4.13: Transition Throughput vs # of Lanes](image)

The UAP efficiently exploits automata parallelism, scaling up by vector lanes. These parallel finite automata workloads can be homogeneous or employ multiple different finite automata models, exploiting the sequence flexibility in UAP that provides independent execution in each lane. As shown in Figure 4.13, for single stream, transitions per second
increases linearly with lanes, growing to up to 72.6 giga-transitions per second; more than 100x higher than traditional CPUs and GPUs.

Many workloads require the ability to support multiple streams efficiently. UAP supports up to 64 streams, flexibly associating them with UAP lanes. As shown in Figure 4.14, UAP throughput increases with parallel streams until limited by the external memory system bandwidth. Performance increases linearly for DDR3, saturating the 667Mhz DDR3 interface at 80 Gbps. For advanced memory systems such as the Hybrid Memory Cube (HMC) system [38], our results show that UAP performance scales well up to 64 streams, achieving 454 Gbps, close to the full bandwidth of an HMC memory.

![Figure 4.14: Scaling up with multiple streams on DDR3 and HMC memory systems on DFA models](image)

4.5.2 Comparison to PowerEN RegX

We provide a detailed comparison to IBM’s RegX engine, a widely-published accelerator for deterministic finite automata. We use identical workloads to those in [31], enabling a direct comparison. Starting from the left in Figure 4.15, DFA and JFA line-rate performance on UAP is comparable to RegX (at right). For complex patterns, the DFA and A-DFA models suffer a state explosion, decreasing their size efficiency and throughput (as fewer patterns can fit in the local memory). The RegX design is 4-wide, and uses that much higher memory
Figure 4.15: Comparing Throughput: UAP (1-wide) on several FA models vs RegX (4-wide) bandwidth deliver 50Gbps on both simple and complex patterns. RegX also benefits from its hardware-managed memory hierarchy (see Figure 4.16). As discussed in Section ??, RegX cannot fully implement the A-DFA and NFA models. The UAP implementations of A-DFA, A-JFA, and NFA show the power of UAP’s programmability and the importance of FA model innovation. These three FA models outperform RegX dramatically on the simple workload, achieving 283Gbps (A-DFA), 283Gbps (A-JFA), and 281Gbps (NFA).

Both A-JFA and NFA outperform RegX dramatically on complex, achieving system throughputs of 295Gbps and 184Gbps respectively. The novel A-JFA model (see Section 4.2.6) achieves highest overall throughput, exploiting UAP flexibility and pattern density to support more simultaneous streams, increasing throughput.

Figure 4.16: System Throughput vs Number of Patterns
Another key difference between UAP and RegX is the memory hierarchy. RegX uses a much wider memory (4-wide [31]), and cached management, whereas UAP employs a 1-wide memory, and scales both lanes and streams to 64. We plot the system throughput versus the number of patterns (Figure 4.16), showing that with < 800 patterns, UAP delivers higher throughput than RegX (as much as 5x higher and superior for all pattern sets). When the number of pattern exceeds 2000, UAP performance declines to match RegX’s throughput, as UAP needs to split patterns in multiple banks when single local memory bank cannot fit all of the patterns. For example, for 3500 patterns, UAP requires nine memory banks, each 16KB, to hold the complete pattern set, limiting stream parallelism to seven.

4.5.3 Hardware Design

We describe the implementation of the UAP micro-architecture, described in Section 4.2.7, beginning with the design of a UAP lane, and then summarizing overall area and power aspects of the UAP system.

UAP lanes are designed to support general-purpose FA computation, and have four key elements: 1) state sequence, 2) combining queue, 3) prefetcher, and 4) memory bank interface (Figure 4.17). First, the state sequence component implements transitions and actions, supporting flexible programmability and generality across FA models. Second, the combining queue is critical for efficient multi-activation management, queueing, and detecting convergence efficiently as needed for NFAs. The combining function is implemented with 12-bit comparators on state identifiers, and we implemented an 8-entry combining queue that support 8 concurrent activations per lane. Third, the prefetcher exploits the linear access for stream data and the static sharing map (connecting streams to UAP lanes, see Table 4.5) to reduce energy and avoid vector register file contention. Finally, the memory bank interface provides efficient access to UAP words, as each lane uses data only from its local memory bank. Physical proximity can then be optimized to minimize access latency and energy.
We designed and implemented the UAP lane in VHDL RTL and synthesized it for 32-nm TSMC process with the Synopsis Design Compiler, producing timing, area and power information. The synthesized design achieves UAP lane delay of 0.57 ns, that combined with 0.24 ns to access the 16KB local memory bank [1] gives overall UAP timing closure at an 0.81 ns clock period (1.2 Ghz). Thus, the UAP lane’s fastest transition rate is 1.2 giga-transitions/second.

Silicon area and power for the UAP design in 32nm TSMC process is shown in Figure 4.18. Area for a 64-lane UAP system is 5.67 mm$^2$. This area includes 64 UAP lanes (31.7%) and a supporting infrastructure that includes 1MB of local memory organized as 64 banks (61.4%), a vector register file (4.5%), and a gather engine (2.4%). The 64 UAP lanes require 1.8 mm$^2$ – less than half of ARM Cortex A9 core’s logic which fabricated in the same 32nm process (4.65 mm$^2$ without L1 cache). Area breakdown detail is shown in Table 4.7. Adding a 64-lane UAP to a large chip (eg.GTX 980M) would incur 0.5% overhead.

Figure 4.18 shows the UAP system on-chip power for 64 lanes processing DFA patterns (see right). For a single stream UAP system power is 507 mW, and 563 mW for 64 streams. In both case, the UAP system is delivering 72.6 giga-transitions/second, with over 80% of the power spent in the local memory references. Adding a UAP system to a large chip (eg. GTX980M) incurs only 0.3% power increase.
<table>
<thead>
<tr>
<th>Component</th>
<th>Area ($mm^2$)</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAP Lane</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State Sequence</td>
<td>0.003</td>
<td>n.a.</td>
</tr>
<tr>
<td>Prefetcher</td>
<td>0.018</td>
<td>n.a.</td>
</tr>
<tr>
<td>Combining Queue</td>
<td>0.007</td>
<td>n.a.</td>
</tr>
<tr>
<td>UAP (64 lanes)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 Lanes</td>
<td>1.798</td>
<td>31.7%</td>
</tr>
<tr>
<td>VRF</td>
<td>0.256</td>
<td>4.5%</td>
</tr>
<tr>
<td>Gather Engine</td>
<td>0.138</td>
<td>2.4%</td>
</tr>
<tr>
<td>Shared Area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Memory (1MB)</td>
<td>3.475</td>
<td>61.4%</td>
</tr>
</tbody>
</table>

Table 4.7: UAP Area Breakdown

Figure 4.18: 64-lane Area and Single-Stream Power

To demonstrate energy efficiency, we compare to a 1Ghz 32-bit RISC processor, and compute the relative energy efficiency (see Figure 4.19a). UAP achieves over 1000x (DDR3) and 2500x (HMC) increase. This benefit comes both from higher performance and efficient transition implementation (local memory reference, a fraction of an instruction). Detailed analysis of where the rather small remaining power goes (see Figure 4.19b) reveals that UAP spends most of its energy local memory reference to access the FA transitions. The large fraction (38% with DDR, 66% with HMC) show that UAP is close the fundamental efficiency limits.
In this chapter, we propose the Unified Automata Processor (UAP), a new architecture that provides general and efficient support for finite automata (FA). The UAP supports a wide range of existing finite automata models (DFAs, NFAs, A-DFAs, JFAs, counting-DFAs, and counting-NFAs), and future novel FA models.

UAP architecture provides a set of UAP actions to support complex extended-FA behavior and realizes the physical memory layout integrated with UAP actions. UAP architecture makes it possible for EffCLiP abstract transition and packing algorithm to be practically implemented and a general support all FA models.

Evaluation on realistic workloads shows that UAP implements all models efficiently, achieving line rates of 94.5% of ideal. The UAP design is efficient in instructions executed, memory references, and dense-packing using EffCLiP. With efficient support for multiple input streams, UAP achieves throughputs that saturate even high speed stacked DRAM memory systems.

UAP achieves a 1.2 Ghz clock rate (32nm TSMC CMOS), and a 64-lane UAP requires only 2.2 mm$^2$ and 563 mW. A single UAP lane delivers line rates 50x greater than software
approaches on CPUs and GPUs. Scaling UAP to 64 lanes achieves FA transition throughputs as high as 295 Gbps, more than 100x higher than CPUs and GPUs, and even exceeding ASIC approaches such as IBM’s RegX by 6x. At these levels of performance and energy-efficiency, UAP is a promising candidate for integration into general-purpose computing architectures.
CHAPTER 5
RELATED WORK AND DISCUSSION

With growing network link bandwidths and unstructured data sizes, the appetite for higher performance finite-automata processing continues to grow rapidly. Research and commercial efforts for efficient implementation of finite automata can be organized into three basic categories: algorithm and data structure approaches for software implementation on traditional CPU's and GPU's, application-specific integrated circuits (ASICs), and field-programmable gate array's (FPGA's). We will discuss those parts one by one in this chapter.

5.1 Algorithms and Data structures

Algorithm and data structure techniques all seek to create small tables for finite-automata, and simplify the computation required to do each transition. Small tables increase the effectiveness of caches, simplifying the computation reduces instruction count.

For example, multiple Alphabet Compression Tables (ACT) [23] exploits redundancy in target states, translating the automata to avoid storing same target multiple times for a particular state. Alphabet Compression Tables act as meta data required to achieve highly-compressed implementation of an unstructured DFA. However, the MACT approach requires additional memory references to access the ACTs for one input symbol and the ACT sizes often large ($#ACT \times |\Sigma| \times \log_2 |Q| \text{bits}$). So, the benefits of automata size reduction are mitigated by the need to maintain ACT tables in the memory hierarchy. Majority Compression captures the redundancies targeted by MACT. In contrast, Majority Compression combined with the EffCLiP transition representation exploits them without the penalty of ACT tables. The EffCLiP transition structure enables a DFA to be densely packed in memory, and no additional serialized memory references are required for transitions.

Another approach is to exploit default transitions for more efficient encoding. $D^2FA$ and
A-DFA [7,25] use this approach to exploit transition redundancy. However, doing so creates an unstructured DFA, raising the question of how to pack the DFA yet maintain a simple addressing function. \( CD^2FA \) [27] solves this problem by using a hash function to address the non-root states. To achieve a perfect hash with addresses generated using the hash function \( CD^2FA \) treats \( D^2FA \) states with more than five transitions as root states, adding a discriminator for each state label. The probability of finding a perfect match depends on actual constructed bipartite graph and the complexity of finding a perfect matching is \( O(n^{3/2}) \). For each root state, a full transition representation that stores a row of a “full table”, and employs direct addressing. These root states wind up relatively large, usually occupying half of total memory used. The flexibility provided by the EffCLiP transition representation and our simple addition operator allow EffCLiP to efficiently pack transitions with \( O(n) \) running time in an unstructured FA together without penalty of root states. Thus EffCLiP achieves similar high load factors (> 97%) as \( CD^2FA \) (100%) with less construction time and guarantees the perfect matching.

Another approach is to construct an alphabet translation table for a given hash function for the specific DFA, thus enabling both dense packing and use of a hashing function. \((2D)^P^2\)-Hash [53] takes this approach for multi-string matching, it progressively construct a translation table achieving the perfect hash for the AC automata. However, the success of translation table construction for a given hash function isn’t guaranteed. Therefore, there is the need to customize the hash function for the specific DFA. Compared to AC-DFA with \((2D)^P^2\)-Hash, AC-DFA with EffCLiP achieves lower memory/character (6.65 bytes/character) and a higher load factor (97%) without the need for a translation table. Further, EffCLiP construction running time complexity is the same as \((2D)^P^2\)-Hash \( (O(n)) \) whilst guarantees packing success. Even better EffCLiP is not limited to multi-string matching, but can also be used to implement other finite-automata models.

Despite these many optimizations for software implementations of FA’s on CPU’s and
GPU’s, realized performance is relatively low. Specifically, the best published performance is 40 Mbps (DFA) and 4.9 Mbps (NFA) on CPU’s [57] and 235 Mbps (DFA) and 40 Mbps (NFA) on GPU’s [57]. To achieve higher speeds customized hardware approaches are required.

EffCLiP is unique in its use of a larger, but more flexible transition representation. That representation allows it to achieve greater packing density (load factor) than hash-based approaches and traditional full table representations employed by CPU and GPU systems. And this compression is achieved with fast, linear-time compression algorithms. But beyond that, the flexibility of the EffCLiP transition also enables simple implementation of addressing, supporting high speed implementation in software and hardware. EffCLiP complements existing hardware approaches, providing a potential avenue both to increase their flexibility and breadth of applicability. And, given the simple addressing function (integer add) employed by EffCLiP, speed may be enhanced as well.

5.2 Application-specific Integrated Circuits (ASICs)

Micron has introduced a DRAM-based NFA Automata Processor (AP) [15]. In AP, state transition elements, counters and boolean operators are interconnected through a hierarchical routing matrix, and integrated in a DRAM array achieving efficient NFA processing with counters (c-NFA). Each chip achieves a line rate of 1 Gbps for NFA, and up to eight can be combined (hardware complexity comparable to UAP systems considered) to deliver 8 Gbps throughput for eight streams. AP’s architecture reads the entire state transition list parallel for each state/input symbol, at each transition. Thus, for a $n$-state NFA, AP reads $n$ bits per symbol up to 48K bits (max states on a chip). In comparison, UAP reads only one or several 32-bit words per transition. While the memory technology differences are significant (AP uses DRAM, UAP uses SRAM), the resulting power differences are large. For example, the worst performance UAP NFA line rate (2.6 Gbps on dotstar0.9) is higher than AP’s 1Gbps. At a system power of 507 mW for worst case NFA (dotstar0.9), UAP achieves 5.3
Gbps NFA throughput, a power efficiency of 10.5 Gbits/watt. AP’s 4 watts per chip [15] is only 0.25 Gbits/watt. Adding more AP chips does not change this figure of merit.

Several ASIC designs specialize for a small set of models (DFA and few variations), employing customized transition hardware, replicating it for throughput, and automatic memory hierarchy management for large FA’s [10, 31]. We compare in detail to PowerEN, an SoC that integrates a regular expression matching accelerator (RegX) with IBM Power processor [31]. The RegX design implements DFA with extensions for scratchpad memory actions such as set/reset and simple computation (supporting DFA and XFA [47] both supported by UAP). However, RegX cannot support many FA models that UAP can including non-consuming transitions and conditional transitions (JFA, A-DFA\(^1\), c-DFA, History-FA, SFA), multiple activations (NFA, c-NFA, Hybrid-FA, RLE-NFA) as well as context-sensitive transitions (flat-JSON-NFA). Comparing line rate on DFA using a single context: UAP (1.2Ghz) achieves 9.07 Gbps, and RegX (2.3Ghz) achieves 9.2 Gbps, similar performance. So, UAP lane performance is similar to a RegX context (BFSM). RegX uses pipelining and multi-context, to increase throughput to 20-40 Gbps. On the same workloads UAP scales to higher throughputs - 80 Gbps (DDR3) and 295 Gbps (HMC). The more important difference is that UAP provides greater model flexibility and programmability, and that flexibility translates into superior application performance (see Figure 4.15).

5.3 FPGA’s: Encoding the FA directly into Logic

FPGA’s have a special advantage over ASIC approaches when it comes to finite-automata. The FPGA approaches encode the FA structure directly in combinational and sequential logic [32, 46, 54], avoiding the interpretive overhead required for both ASIC and software designs. As a result, despite slower base logic, the FPGA designs achieve higher performance on single input streams (1 to 15 Gbps). However, if they are to be used in general processing system,
FPGA designs require complex integration of programmable hardware and tool chains for creating the FPGA designs. And further, the resulting FA implementation in FPGA, while high speed, is typically not easily integrated with higher level software processing.

5.4 Summary

CPU/GPU approach has the properties of great programmability, fast development, and easy accessibility. Modern computing systems carrying CPU/GPU make future integrated UAP possible to let normal people use high performance FA processing everywhere in the world. However, the dark side of this approach is its relative low-performance extended-FA processing and can’t meet the needs of many emerging applications. On the other hand, ASIC delivers best performance and energy efficiency because its direct map and hardware support for FA processing. A lot of optimization techniques are used to improve the memory efficiency. However, ASIC chip development cycle and fabrication cost becomes the major barrier for wide acceptance. A better hardware solution would be using FPGA device. Excellent NFA processing line rate is its fantastic advantage compared to CPU/GPU. However, huge resource waste for multi-stream matching, thus results poor overall throughput and still relative complex and slow development cycle are FPGA approach’s major limitations.

The best case would be ASIC/FPGA performance and energy efficiency with CPU/GPU programmability and development time. That requires a fundamentally new computer architecture: UAP. With EffCLiP, UAP achieves high memory density and fast clock cycle. UAP delivers cheap high-performance general FA processing. It has great programmability and can be easily integrated into a traditional CPU/GPU allowing fast deployment of FA processing.
CHAPTER 6

SUMMARY AND FUTURE WORK

6.1 Summary

EffCLiP and UAP together enables high performance flexible automata processing. They enable new automata to be created and deployed as software exploiting integration of high performance automata processing on FA accelerated CPU, instead building specific ASIC/FPGA for high performance/energy-efficiency which is known to be slow development cycle, label-intensive, platform-specific and expensive. The traditional approach for high performance automata processing involves mapping the automata directly to hardware circuit (ASIC/FPGA). General-purpose architectures like CPU/GPU has great software programmability but low performance. With the our novel EffCLiP and UAP approach, applications fully utilize the powerful high performance automata processing which has direct hardware acceleration support on traditional CPU/GPU with UAP extension. Future, they can invent new models that suit their applications and also exploit these changes rapidly with high performance. FA models flexibility brings superior memory size and performance advantage. Easy integration with traditional CPU/GPU enables UAP has great software programmability and fast development time. These properties are very critical in current FA processing applications.

Our research efforts have produced the design of a unified automata processor (UAP), and demonstrated both the architecture’s generality and an efficient implementation on a representative set of finite automata models and workloads. Our results show that UAP achieves line rates up to 9.07 Gbps (1.13 giga-symbols per second), 50x faster, and 295 Gbps , 100x faster than CPU and GPU implementations. As a result, UAP can match the performance of many highly customized ASIC and FPGA implementations, and exceed the performance of pure software implementations on conventional CPU and GPU platforms. In
addition, UAP can be easily integrated into a traditional CPU architecture with small area and power costs. Combined with demonstrated generality and high efficiency, we believe that these attributes make UAP a promising computer architecture design to be included in general purpose architectures.

6.2 Future Work

There are still several open research problems. We discuss several of them below:

We demonstrated that in the UAP can achieve high line rate for the DFA model, but the line rate is lower for NFA. An interesting problem is how to accelerate NFA. Exploration on how to manage multiple activations, avoid conflicts in their layout, and exploit multiple UAP lanes are all promising.

UAP delivers excellent throughput by exploiting parallel lanes. But each lane is limited to FA’s that will fit in a single 16KB memory bank. Can we add mechanisms to enable resource sharing across UAP Lanes (combining queues, local memory banks, or control) to achieve greater flexibility without reducing energy efficiency or throughput in simple cases.

While UAP can fit even the largest regular expression in our workloads into the local memory, some workloads and FA’s require even larger representations. One approach would be to add a memory hierarchy. What are the performance, power, and cost consequences?

Our work on UAP demonstrates it delivers high-performance FA processing. However, we did not consider multi-tasking, and its implications for performance and protection. What approaches are suitable, and what are their implications for performance and energy?

We learned that under most of the cases, UAP provides sufficient hardware resources complete FA processing successfully. However, it is possible that NFA’s that create many actives states may overflow the combining queue. How should these cases be handled?

While we have performed extensive studies on a range of network intrusion detection, compression, signal triggering, and database workloads, broader application studies are of
interest. For example, exploration of bioinformatics, text search, big data, and even web question-and-answer applications are likely to benefit.

UAP’s ambition is to support flexible FA-model innovation at high-performance. Study of any range of domains, to develop new, novel extended finite automata models is promising.
REFERENCES


