Fast Support for Unstructured Data Processing: the Unified Automata Processor

Yuanwei Fang, Tung T. Hoang
Dept of Computer Science
Univ of Chicago
{fywkevin,hoangt}@
cs.uchicago.edu

Michela Becchi
Dept of Electrical &
Computer Eng.
Univ of Missouri
becchim@missouri.edu

Andrew A. Chien
Computer Science,
Univ of Chicago
MCS, Argonne Natl Lab
achien@cs.uchicago.edu

ABSTRACT

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. Evaluation on realistic workloads shows that UAP implements all models efficiently, achieving line rates 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.

The UAP design and 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.

Categories and Subject Descriptors

A.1 [Computer Science]: Computer Systems Organization

Keywords

Computer Architecture, Processor, Finite Automata, Data Processing, Pattern-matching, Energy-efficient, Custom Architecture

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MICRO-48, December 05-09, 2015, Waikiki, HI, USA

Copyright 2015 ACM ISBN 978-1-4503-4034-2/15/12

DOI: <http://dx.doi.org/10.1145/2830772.2830809> ...\$15.00.

1. INTRODUCTION

General-purpose computing architectures incorporate extensions of instruction set and micro-architecture to meet the needs of new workloads. Notable examples include floating point (IBM [1], Intel 80486 [2], MIPS R4000 [3]), multi-media (Intel's MMX [4] and SSE [5], AMD's NEON [6]), and recent growth of these into full-blown vector instructions sets (AVX2 [7,8]). Other recent extensions integrate specialized units for cryptography (Intel's AES-NI [9], ARM's Cryptographic extensions [10]). In each case, the new instructions benefit a small, but important class of computations, providing dramatic performance improvements. In each case, the framework of that functionality is significantly different (and thus adds value that single instructions in the traditional framework could not).

We focus on an growing, important class of computations that employ finite automata-based algorithms in order 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 [11], question answering systems such as Watson, Siri, Cortana, and Google Now [12–15], and compression in widely-used NoSQL systems [16–18]. Many genomics applications include pattern matching as a performance-critical phase, followed by a more general-purpose computation. For example, shotgun assembly [19] 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 [20]. Computationally expensive biological applications that include a pattern matching phase include: de novo assemblers [21], orthology inference [22, 23], and motif discovery [24]. Other areas where finite automata are essential include text data mining [25, 26], network monitoring [27], content-based routing [28], and application-level filtering [29]. Finite automata-based applications continue to grow in number and importance, inspiring development of new extended finite automata models [30–38]. In the networking domain alone, the last ten years have seen the publication of more than 400 papers on efficient automata processing, including new models, algorithms, and data structure design.

Despite its foundational importance, the efficient implementation of finite automata processing remains a challenging open research problem and the subject of extensive research. For traditional computers, finite automata processing is challenging because it requires unpredictable memory references and control transfers that reduce the effectiveness of branch prediction and instruction level parallelism - staples of high performance CPUs [39] and GPUs [40–42]. As a result, high performance FA applications have historically resorted to custom hardware approaches (ASICs [43,44] and FPGAs [45–47]) to meet demanding real-time requirements.

We propose the Unified Automata Processor (UAP), a new architecture (software-hardware interface) that enables flexible software innovation in finite automata design, and supports it with efficient, programmable engines that can outperform custom hardware. We analyze requirements for the finite automata computation space, and design the UAP lane – a building block that achieves flexible programmability and broad efficiency. The UAP system integrates UAP lane operations into traditional cores with vector instructions, and exploits transition-level parallelism with both multi-step and vector parallelism. We describe the instruction set extensions and the high-level software interface.

The UAP is a part of the 10x10 project [48–50], an exploration of customized architecture that federates heterogeneous micro-engines to achieve both energy efficiency and general-purpose performance. The UAP and its predecessor the GenPM [51] were developed to address indirection-intensive pattern-matching workloads.

In this study, We implement the UAP – a simulator, detailed hardware design, and associated compilation software. Using these tools and implementation, we evaluate both the generality and efficiency of the UAP across a wide range of finite automata models and representative workloads. Our results show that UAP performance outstrips software-only FA implementations on CPUs and GPUs by orders of magnitude. Further, UAP performance is competitive with custom implementations for specific FA models. But UAP is significantly more general, and that programmability enables it to support software (FA model innovation) to deliver superior application performance. Specific contributions include:

- The architecture of the Unified Automata Processor (UAP), specifically the UAP lane design that supports general-purpose, efficient finite automata computing, and its integration into a conventional instruction set architecture.
- Evaluation of the UAP empirically demonstrating 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 gigasymbols per second), 50x faster, and throughputs up to 295 Gbps, 100x faster than CPU and GPU implementations (Section 6).
- Comparison of UAP to a commercial ASIC design (PowerEN RegX) [44] that shows UAP deliv-

ers 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 (Section 5.3).

- Evaluation of UAP implementation (32nm CMOS, 1.2 Ghz), shows small area (2.2 mm^2 for 64 lanes) and low power (563 mW at max rate). Design evaluation shows UAP is efficient, using few memory references per transition (often one), and requiring little execution energy beyond those references (close to maximum achievable efficiency).

The paper is organized as follows. Section 2 summarizes the broad utility of finite automata models, and shows the value of FA diversity to application efficiency. In Section 3, we present the design of the Unified Automata Processor (UAP). We describe workloads and modelling approaches in Section 4. Experimental results are presented in Section 5, and discussed with related work in Section 6. Section 7 closes summarizing and suggesting future research directions.

2. BACKGROUND

We discuss the finite automata computing model, and show the importance of FA model innovation for application performance.

2.1 Finite Automata Models

Above, we have documented the number and breadth of applications that require finite automata for the efficient processing of unstructured data. Finite automata (FA) [20] are an effective algorithmic tool for multi-pattern search where matching is implemented by FA acceptance. Critical to FA computation is a trade-off between automaton size and per-symbol processing. Non-deterministic and deterministic finite automata (NFAs and DFAs) represent extremes. NFAs have limited size but require expensive per-character processing, whereas DFAs offer limited per-character processing at the cost of a larger automata. In Figure 1, we illustrate NFA and DFA accepting three patterns (**a+bc**, **bcd+** and **cde**). The states active after processing **aabc** are gray. The NFA example shows four active states, whereas the DFA only one. If each state traversal requires a memory oper-

Figure 1: (a) NFA and (b) DFA for **a+bc**, **bcd+** and **cde**. Accepting states (bold), Active states after input **aabc** (gray). Σ denotes the entire alphabet.

ation, processing the next input character requires four

memory operations for NFA and one for DFA. In the worst case, an N-state NFA may have N concurrent active states (N state transitions for single input symbol); a DFA has only one active state, and thus is limited to one transition per symbol. The DFA parsimony on transitions comes at a cost of increased memory requirements. In an equivalent DFA, each state must represent a set of NFA states that can be simultaneously active; therefore, potentially requiring 2^N states. Previous work [34, 36, 52] showed that this so-called “state explosion” occurs only in the presence of complex patterns containing bounded and unbounded repetitions of large character sets.

2.2 Importance of Finite Automata Diversity

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 [43, 53], run-length encoding [43], transition redundancy removal [54] and bitmap [55]. 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 [52], delay-input DFAs [31, 32], hybrid-FAs [35], history-based-FAs [36], XFAs [34], counting-FAs [37], JFAs [30], SFAs [38] and dual-FAs [33]. These alternative automata models avoid or mitigate the state explosion problem in the DFA representation. While reducing the resource requirements of NFAs and DFAs, many of these FA models have equivalent expressive power.

(a) DFA Models

(b) NFA Models

Figure 2: # patterns in 256KB, selected representative workloads.

While each automata model gives benefits for several types of FA structure, none is best for all regular expression patterns. As shown in Figure 2, A-DFA, JFA, and c-DFA respectively achieve highest pattern density for some patterns in representative workloads (see Section 4 for full workload description). 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. In some cases, these extended FAs [30, 37] cannot represent all patterns. Thus, each automata model has its set of strengths and weaknesses.

3. THE UNIFIED AUTOMATA PROCESSOR

3.1 Requirements for General-Purpose

We analyze the mechanism requirements for a general-purpose finite automata engine (Figure 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 covering these six elements efficiently enables a system to implement these twelve models and many more published FA models.

Figure 3: A small set of mechanisms can support diverse finite automata models; six models used for evaluation are in green.

3.2 Key Design Elements

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). UAP transitions are implemented by a programmable *UAP lane*, accomplishing in a few cycles work that requires ≈ 24 instructions in a traditional CPU core [39]. Each lane is composed with local memory bank (on-chip),¹ and efficiently accessed via vector operations in a RISC core.

3.3 Transitions and Memory Layout

Transition Primitives and Actions: Each *UAP transition* consists of a *transition primitive* associated with zero or more *actions* (Figure 4a).² The overall FA is a set of UAP transitions laid out in memory, using the EffCLiP algorithm [56] that combines a *signature* for placement flexibility with coupled-linear packing. The resulting benefits are dense memory utilization and a simple hash function – integer addition (see **Base** below) that enables fast clock rate. EffCLiP effectively achieves a “perfect hash”. FA acceptance information is located at the end of the address space (Figure 4b).

(a) Sample UAP transitions (b) Sample UAP Memory Map
Figure 4: UAP Transitions and Memory Layout

Figure 5: UAP Transition Primitives

UAP supports three transition primitives: base, teleport, and conditional (Figure 5). These primitives are sufficient to efficiently implement our six FA exemplars, all finite automata models of which we are aware, and create 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 (Section 5.5) value. The primitives are depicted in Figure 5. We define below how each computes the transition target address:

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

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

²In some cases, there may be a gap between the transition primitive and associated actions. The actions are always contiguous.

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 are shown in Table 1. All transition primitives and actions are 32 bits each. *Sig* is a signature used to check if the computed transition address is the desired transition. This check, *Sig*, uses the input symbol that uniquely identifies the source, enabling placement freedom for EffCLiP [56] to achieve dense layout. *Target* encodes a target state identifier. *Attach* encodes additional information for automata programmability or placement when signature check fails. *Type* dictates usage of *Attach* (as in Table 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 transition. To increase encoding density, we employ *majority transition* optimization [56] using a single representative transition for a set of transitions from the same source state to the same target state (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 the transition. Here also a single memory access/-transition can be achieved.

Sig(8)	Target(12)	Type(4)	Attach(8)
Unused(7)	Opcode(4)	DstReg(2)	SrcReg(2)
		Imm(16)	Last(1)

Table 1: Transition Primitive and Action Formats

3.4 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 FA 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

Instruction	Operation
<code>write_state_regs(config1, config2)</code>	Write UAP states with values from two 2048-bit vector registers, including <initial state, accept base, lane mask, vector sharing> 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.
<code>read_state_regs(config1, config2)</code>	Read out UAP states <current state, accept base, lane mask, vector sharing> into two 2048-bit vector registers.
<code>traverse(start, steps, acceptReg)</code>	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.
<code>complete_traverse()</code>	Block until all traverse steps complete.
<code>read_combine_Q(lane, entry, reg)</code> , <code>write_combine_Q(lane, entry, reg)</code>	lane #, entry #, the value is extracted from (written to) the combining queue and deposited into (sourced from) a 32-bit register.
<code>read_reg(lane, UAPreg, toReg)</code> , <code>write_reg(lane, UAPreg, fromReg)</code>	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.

Table 3: Instructions to access UAP Functionality

Type	Transition Primitive	FA Term	Attach Usage
0	Teleport	Default Transition	Default state address
1	Teleport	Epsilon Transition	Target state address
2	Base	elaborated Transition	1st action's address (9 bit offset to current address)
3	Conditional	Cond. Transition	Re-target state address
4	Teleport	Majority Transition	= low order 9 bits of target state address
5	Teleport	Majority Transition	low order 9 bits of target state identifier (immediate)
6	Teleport	Default Transition	Default state identifier (immediate)
7	Teleport	Epsilon Transition	Target state identifier (immediate)

Table 2: **Type** Field and Transition Primitives

and UAP). The initial states for each FA are loaded into the UAP by writing the state registers. Then a basic loop of load stream data, process FA transitions, then check for completion ensues (see below). These instructions are described in Section 3.5 and Table 3.

```

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(); }

```

3.5 Instruction Interface to UAP

Instructions to program the UAP are listed in Table 3; 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. See Section 5.5 for detailed UAP lane description.

The UAP `traverse` instruction has two dimensions of transition-level parallelism to exceed one FA transition per instruction (Figure 6). First, for a single UAP lane, multi-step execution is used (specified by `traverse`), al-

Figure 6: A 4-lane UAP processing input stream “ab”

lowing a finite automaton to progress up to 256 steps, consuming 256 input symbols in a single instruction. Second, vector execution across UAP lanes (specified by configuration using `write_state_regs`) allows up to 64 lanes running concurrently with multi-step against one to 64 distinct input streams. Together, these two dimensions enable a single `traverse` instruction to process as many as 16,384 transitions.³ The UAP assumes basic vector operations for loading data and computation.

3.6 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.

³24 instructions/transition is typical for conventional architectures [39], so UAP can replace over 300,000 instructions with a single `traverse` instruction.

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 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 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 5.

Figure 7: Novel Automata on UAP

3.7 UAP Architecture

UAP adds 64 UAP lanes to the base architecture (a core, a vector register file, and local scratchpad memory). Each lane is designed to efficiently support FA models, pairing with a local memory bank. Detailed description of the UAP lane’s micro-architecture is in Section 5.5. To support the understanding of our performance evaluation (Section 5), we outline the performance critical aspects of the design:

Figure 8: UAP adds UAP Lanes to Base Infrastructure (core,VRF,and scratchpad local memory)

1. The UAP lane design implements each UAP transition primitive or action in a single cycle. For simple FA models, a UAP lane achieves one transition per cycle. Complex models may require additional cycles per transition (Figure 6).
2. Each UAP lane is associated with one bank of an on-chip local memory [57,58] that provides fast access for transition primitives and actions (32 bits each). With a 16KB local memory bank, each UAP lane runs at 1.2GHz.
3. *traverse* instruction is defined similar to multi-cycle instructions (such as FDIV), accessing configuration defined registers at issue and retirement. After all lanes finish the multi-cycle execution, the

acceptance results are returned in a vector register.

Total local memory capacity is 1MB (64x16KB). Stream data is moved from DRAM to local memory with an efficient on-chip data-mover/DMA [59], and then loaded into vector registers. UAP lane implementation is discussed in detail in Section 5.5.

4. METHODOLOGY

4.1 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 chosen to exploit the best features of at least one FA model, producing a natural breadth of application areas. We use matching workloads for direct comparisons with best published results on CPU, GPU, AP, and RegX [40,44,60]. These workloads are summarized in Table 4. Specifically, we use the Snort synthetic [40,61] and PowerEN [44] 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 [62], 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 [27]. We set the *Scope* and *wildcnt* trace such that 5%,25%,50% of the traffic triggers counting action. Huffman [63] 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 [63] for Huffman decompression.

RLE is a run-length encoding automaton with 26 counting states (see Figure 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 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 [64] in JSON format. The traces for PowerEN datasets were from [44], where each input stream is 1000 symbols. The input streams used for *Snort* workloads were synthetically generated using the tool described in [61]. 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 models is achieved by adapting open-source software [51,61] to generate the required UAP memory layout (see Section 3.3). We also extended the software to generate the three novel FAs (A-JFA, RLE-DFA, flat-JSON-NFA). In cases where a set of patterns is larger than a local

FA Model	Application Area	Pattern Sets	Pattern Feature
DFA [20]	Decompression, Network monitoring	Huffman(Microsoft) [63];EM, range1 [40, 60];simple [44]	No state explosion
NFA [20]	Network monitoring	spyware, EM, range1, dotstar0.9 [40, 60]; simple, complx [44]	Any regular expressions
A-DFA [32]	Network monitoring	EM, range1 [40, 60];simple [44]	No state explosion
counting DFA [37]	Signal triggering	Scope (Keysight) [62]	String counting quantifier
counting NFA [37]	Network monitoring	Snort v2.9 [27]	Wildcard counting quantifier
JFA [30]	Network monitoring	spyware, dotstar0.9 [40, 60]; complx [44]	State explosion
A-JFA	Network monitoring	spyware, dotstar0.9 [40, 60]; complx [44]	State explosion
RLE-DFA	Compression	Run Length Encoding (RLE) Tree	Encoding tree
Flat-JSON-NFA	Query over JSON data	forall nat in nation: if (nat.nationkey < X) collect nat	Context sensitive

Table 4: Workload (Pattern Sets) used for Evaluation

memory bank, we split the set over multiple banks. If the patterns do not fill all 64 banks, the finite automata are replicated to provide for all lanes.

4.2 Performance 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;
- System Throughput (Gbps): aggregate bit rate across several streams. This unified metric considers both size and speed properties of FA models;
- 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 Hardware Metrics

- Area (mm^2): Si area, 32nm TSMC CMOS process;
- Clock Speed (GHz): max speed of the UAP system;
- Power (mW): UAP on-chip power due to UAP activities – see components in Table 5 and full system power including external memory system;

4.4 System Modeling and Configuration

The UAP is designed and integrated with RISC processor, vector register file using high-level description language (LISA) in Synopsys 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 [65] and using DRAMSim2 [66] 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. Average energy per 64-byte reference at peak bandwidth is approximately 34 nJ (66.4 pJ/bit) for DDR3 and 4 nJ (7.9 pJ/bit) for HMC. This matches public data for DDR and HMC of 66 pJ/bit and 8 pJ/bit [67]. 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, a 64x2048-bit vector register file, and a 64-bank, 1MB scratchpad memory.

5. EXPERIMENTAL RESULTS

5.1 Line Rate, Throughput, and Instructions

We evaluate UAP using six exemplar and new FA models on a variety of workloads, reporting line rate performance (Figure 9). We include the 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 trend as c-DFA but at slightly lower line rates 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 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 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.

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. For single stream, transitions per second increases linearly with lanes, growing to up to

Figure 9: Line Rate vs FA Model

Figure 10: Transition Throughput vs # of Lanes

Figure 11: Scaling up with multiple streams on DDR3 and HMC memory systems on DFA models

72.6 giga-transitions per second (Figure 10).

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 11, 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 [68], our results show that UAP performance scales well up to 64 streams, achieving 454 Gbps, close to the full bandwidth of an HMC memory.

Architecture efficiency can also be measured by number of instructions per transition. Typical RISC and x86 CPU DFA implementations average 24 instructions per transition [39]. UAP’s vector and multi-step parallelism reduce this number dramatically to below one ten-thousandth (1/10,000th) of an instruction per transition (Figure 12). 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.

5.2 Memory Width: Performance and Energy

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 consider UAP lane designs that 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 design), 2-wide, and 3-wide respectively.

First, we explore memory width impact on line rate (see Figure 13a). 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 13b). 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 – some-

times quite dramatically. The reason for this is shown clearly in Figure 13c; 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.

5.3 Comparison to PowerEN RegX

We compare to IBM’s RegX engine, a widely-published accelerator for deterministic finite automata. We use identical workloads to those in [44], enabling a direct comparison. Starting from the left in Figure 14, DFA and JFA throughputs on UAP are 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 bandwidth deliver 50Gbps on both *simple* and *complex* patterns. RegX also benefits from its hardware-managed memory hierarchy (Figure 15). As discussed in Section 6, 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 *simple*, 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 (Section 3.6) achieves highest overall throughput, exploiting UAP flexibility and pattern density to support more simultaneous streams, increasing throughput.

Another key difference between UAP and RegX is the memory hierarchy. RegX uses a much wider memory (4-wide [44]), 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 15), 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 2,000, UAP performance declines to RegX’s throughput, as UAP needs to split patterns in multiple banks when single

Figure 14: Comparing Throughput: UAP (1-wide) on several FA models vs RegX (4-wide)

Figure 15: System Throughput vs Number of Patterns

local memory bank cannot fit all of the patterns. For example, for 3,500 patterns, UAP requires nine memory banks, each 16KB, to hold the complete pattern set, limiting stream parallelism to seven.

5.4 Transition Packing (Compression)

Because encoding density is important, we compare UAP [56] 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 [44] uses the BFSM algorithm [54] that exploits aggressive transition redundancy removal for DFAs.

Figure 16 shows results for the simple400 and complex200 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 12: Instructions per Finite Automata Transition vs. Lanes and Multi-step

Figure 16: Density: UAP vs RegX

5.5 UAP Implementation

Figure 13: Line Rate, Line Rate/Power, and Bandwidth Utilization with Local Memory Width Increase

We describe the implementation of the UAP micro-architecture, described in Section 3.7, beginning with a UAP lane, and summarizing overall area and power aspects of the UAP system.

Figure 17: Micro-architecture of a UAP Lane

The UAP lane supports general-purpose FA computation, with four key elements: 1) state sequence, 2) combining queue, 3) prefetcher, and 4) memory bank interface (Figure 17). First, the state sequence component implements transitions and actions, incorporating a set of fixed decode and operation units that support the transition representation in hardware and implementing the specific composition of 24 load, mask, shift, and EX instructions in a single UAP cycle. Co-design of the UAP formats (Table 1) and lane implementation enable high clock rate whilst supporting general-purpose FA processing. Each UAP lane has two architecturally visible registers storing computation variables for UAP actions. Second, the combining queue provides efficient multi-activation management, queueing, and detecting convergence 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 moves the input stream (from a vector register) to a local stream buffer before *traverse*, based on the static mapping (see Table 3). This frees the vector register file for other use. Finally, the memory bank interface provides fast access to UAP words, as each lane uses data only from its local memory bank. Physi-

cal 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 Synopsys Design Compiler, producing timing, area, and power. The synthesized design achieves a delay of 0.57 ns, that combined with 0.24 ns to access the 16KB local memory bank [65] gives overall UAP timing closure with a clock period of 0.81ns (1.2 Ghz – fastest transition = 1.2 giga-transitions/second).

Figure 18: 64-lane Area and Single-Stream Power

Silicon area and power for the UAP design is shown in Figure 18. A 64-lane UAP system is 5.67 mm^2 . This 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). Comparing power, Figure 18 shows that UAP on-chip power is approximately one-half that of a single Cortex-A9 core. A more detailed area breakdown is shown in Table 5. Adding a 64-lane UAP to a large chip (eg. GTX 980M) would incur 0.5% overhead.

	Component	Area (mm^2)	Fraction
UAP Lane	State Sequence	0.003	n.a.
	Prefetcher	0.018	n.a.
	Combining Queue	0.007	n.a.
UAP (64 lanes)	64 Lanes	1.798	31.7%
	VRF	0.256	4.5%
	Gather Engine	0.138	2.4%
Shared Area	Local Memory(1MB)	3.475	61.4%

Table 5: UAP Area Breakdown

Table 6 shows power for a UAP with 64 lanes processing a single input stream, delivering 72.6 giga-transitions per second on a DFA workload. On-chip power alone, as well as system power including DDR and HMC memory systems are shown. Increasing to 64 input streams increases on-chip power to 563 mW.

On-chip	System(DDR)	System(HMC)
507mW	1040mW	616mW

Table 6: UAP System Power

Figure 19: UAP energy use by Component

Figure 19 presents UAP energy use by component. Note that both UAP(DDR) and UAP(HMC) systems are over 1,000x more energy efficient than the CPU. This benefit comes both from higher performance and efficient transition implementation. In UAP, by far the greatest fraction of on-chip power is local memory references. For the UAP systems, external memory accounts for 50% (DDR3) and 20% (HMC) respectively of system power. Note that in the rightmost bar, UAP(HMC), 66% of system power is consumed in local memory references, so UAP is approaching maximum achievable energy efficiency.

6. RELATED WORK AND DISCUSSION

Software on CPU and GPU: Software on CPU and GPU regular expression matching engines focus on minimizing the memory footprint and bandwidth; extended DFA approaches predominate due to their predictable memory bandwidth requirements [30–37, 39, 52] with effort focused on automata design. One study [40] presents an experimental evaluation of several FA models (NFAs, DFAs and A-DFAs) on CPUs with large pattern sets, reporting single-stream performance of 40 Mbps on a single-core Xeon E5620 processor. Others exploit SIMD execution on small pattern sets using efficient gather/scatter extension [69] and bit-parallel algorithm [70, 71]. Recent work on small FSMs and inputs achieved 1.5 Gbps on a single CPU thread, but the challenges of large, complex FAs and data sets remain unaddressed [63].

Exploiting greater memory bandwidth, GPU implementations [40] report throughputs on *spyware*, *EM*, *range1* upto 0.164, 0.23, 0.207 Gbps, and line rates upto 0.09, 0.13, 0.12 Gbps. Using the same workloads, UAP exploits FA models diversity and multiple lanes achieving throughputs 21.5, 28.2, 14.4 Gbps, and line rates

8.55, 9.07, 9.07 Gbps respectively, 100x higher throughput and 50x higher line rate. The difference lies in UAP instruction efficiency, reducing counts by over 10,000-fold, and efficient use of local memory - encoding, latency, and bandwidth. Other recent work [42] addresses small pattern sets (from 16 to 36 Snort rules) with NFA on GPUs, achieving up to 13 Gbps; while most consider such small pattern sets unrealistic, we estimate UAP performance at > 350 Gbps for such small sets. Ultimately on both CPUs and GPUs unpredictable memory accesses and control flow limit performance.

ASIC implementations: Micron’s AP is a DRAM-based NFA Automata Processor (AP) [60]. In AP, each state and associated inbound transitions is represented as a 256-bit vector (8-bit symbols). These vectors are stored in STE element, and used with the routing matrix to implement transitions (including forking and merging). The AP design requires a full 256 bits to be read for each outgoing edge from currently active states for each symbol. Each AP chip achieves 1Gbps line rate, and 8 can be ganged to deliver 8 Gbps throughput (hardware complexity comparable to UAP systems considered). In comparison, UAP reads 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 chip 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 [60] 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 [43, 44]. We compare in detail to PowerEN, an SoC that integrates a regular expression matching accelerator (RegX) with IBM Power processor [44]. The RegX design implements DFA with extensions for scratchpad memory actions such as set/reset and simple computation (supporting DFA and XFA [34] both supported by UAP). However, RegX cannot support many FA models that UAP can including non-consuming transitions and conditional transitions (JFA, A-DFA⁴, c-DFA, History-FA, SFA), multiple activations (NFA, c-NFA, Hybrid-FA, RLE-DFA) as well as context-sensitive transitions (flat-JSON-NFA). Comparing line rate on DFA using a single context: UAP (1.2 GHz) 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

⁴RegX supports single-depth default transition, but not multiple in sequence.

into superior application performance (see Figure 14).

FPGA designs: Most FPGA approaches [45–47] map FA’s directly into the chip’s programmable logic. Such designs aim to logic size while achieving fast operation. Typically, logic-based implementations use NFA designs based on the one-hot encoding scheme [45]. By encoding each FA state in a flip-flop, NFA is easily implemented with parallel logic at clock cycle per symbol. With multi-symbol achieving even higher symbol rates. Further, NFA models allow efficient patterns combination. FPGA designs report line rates of 1 to 15 Gbps. Since the state information is distributed across the FPGA, supporting multiple streams requires expensive logic duplication. At 3 to 5 Gbps line rate for average workloads, UAP achieves performance parity with these FPGA designs even without recourse to multi-symbol and optimized encoding approaches [43, 47] common in FPGA approaches, but expect that such techniques will make UAP even more competitive. While FPGAs are a convenient platform for network interface integration, our UAP can be integrated with traditional CPU architectures and conveniently joined with higher-level software processing. And of course UAP supports a broad, general set of finite automata models.

7. SUMMARY AND FUTURE WORK

We present the Unified Automata Processor (UAP), an architecture designed for general-purpose, high performance FA processing. Evaluation on representative set of FA models and workloads shows that the UAP matches custom hardware performance, and exceeds pure software implementations on CPUs and GPUs by orders of magnitude. The UAP can be easily integrated into a traditional CPUs at small area and power costs. Combined with its generality, these attributes make UAP for inclusion in general purpose architectures. UAP’s ability to support a wide range of finite automata models creates opportunities for future research. Interesting directions include: exploring the processing of multiple symbols per clock period to achieve higher line rates, study of the cost to integrate UAP in a commercial processor design, and broad comparative studies in software using UAP across finite automata models.

8. ACKNOWLEDGEMENTS

This work is supported in part by the Defense Advanced Research Projects Agency under award HR0011-13-2-0014, a gift from Keysight, and the generous Synopsys Academic program. Becchi was supported by NSF awards CCF-1421765 and CNS-1319748. We thank Henry Hoffmann and anonymous reviewers for their valuable suggestions.

9. REFERENCES

- [1] S. Anderson, J. Earle, R. Goldschmidt, and D. Powers, “The ibm system/360 model 91: floating-point execution unit,” *IBM Journal*, vol. 11, no. 1, pp. 34–53, 1967.
- [2] B. Brey, “Intel microprocessors: Architecture, programming and interfacing-8086-8088, 80186, 80286, 80386, and 80486: Prentice hall ptr,” 1993.
- [3] J. Heinrich, “Mips r4000 microprocessor user’s manual: Mips technologies,” 1994.
- [4] A. Peleg *et al.*, “Mmx technology extension to the intel architecture,” *Micro, IEEE*, vol. 16, no. 4, pp. 42–50, 1996.
- [5] S. K. Raman, V. Pentkovski, and J. Keshava, “Implementing streaming simd extensions on the pentium iii processor,” *IEEE micro*, vol. 20, no. 4, pp. 47–57, 2000.
- [6] J. Yiu, “The definitive guide to arm cortex-m3 and cortex-m4 processors: Newnes,” 2013.
- [7] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, “Intel avx: New frontiers in performance improvements and energy efficiency,” 2008.
- [8] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.
- [9] S. Gueron, “Intel advanced encryption standard (aes) instructions set,” 2010.
- [10] J. Goodacre, “Technology preview: The armv8 architecture,” 2011.
- [11] A. Putnam *et al.*, “A reconfigurable fabric for accelerating large-scale datacenter services,” in *Computer Architecture, ACM/IEEE International Symposium on*, pp. 13–24, 2014.
- [12] D. Ferrucci *et al.*, “Building watson: An overview of the deep qa project,” *AI magazine*, vol. 31, no. 3, pp. 59–79, 2010.
- [13] J. Hauswald *et al.*, “Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers,” in *ASPLOS’15*.
- [14] F. Seide, G. Li, and D. Yu, “Conversational speech transcription using context-dependent deep neural networks,” in *Interspeech*, pp. 437–440, 2011.
- [15] M. F. Porter, “An algorithm for suffix stripping,” *Program: electronic library and information systems*, vol. 14, no. 3, pp. 130–137, 1980.
- [16] A. Harris, “Distributed caching via memcached,” in *Pro ASP. NET 4 CMS*, pp. 165–196, Springer, 2010.
- [17] R. Nishtala *et al.*, “Scaling memcache at facebook,” in *NSDI’13*, pp. 385–398, 2013.
- [18] K. Chodorow, *MongoDB: the definitive guide*. O’Reilly Media, Inc., 2013.
- [19] J. C. Venter *et al.*, “The sequence of the human genome,” *science*, vol. 291, no. 5507, pp. 1304–1351, 2001.
- [20] J. E. Hopcroft and J. D. Ullman, “Formal languages and their relation to automata,” 1969.
- [21] P. Flicek and E. Birney, “Sense from sequence reads: methods for alignment and assembly,” *Nature methods*, vol. 6, pp. S6–S12, 2009.
- [22] G. C. Conant, “Neutral evolution on mammalian protein surfaces,” *Trends in genetics*, vol. 25, no. 9, pp. 377–381, 2009.
- [23] H. Li *et al.*, “Treefam: a curated database of phylogenetic trees of animal gene families,” *Nucleic acids research*, vol. 34, no. suppl 1, pp. D572–D580, 2006.
- [24] I. Roy and S. Aluru, “Finding motifs in biological sequences using the micron automata processor,” in *IPDPS*, pp. 415–424, IEEE, 2014.
- [25] A. Chaudhri, R. Zicari, and A. Rashid, *XML Data Management: Native XML and XML Enabled DataBase Systems*. Addison-Wesley Longman, 2003.
- [26] K. Wang, M. Stan, and K. Skadron, “Association rule mining with the micron automata processor,” in *IPDPS’15*, IEEE.
- [27] M. Roesch, “Snort: Lightweight intrusion detection for networks,” in *LISA*, vol. 99, pp. 229–238, 1999.
- [28] A. Carzaniga *et al.*, “A routing scheme for content-based networking,” in *INFOCOM’04*, vol. 2, pp. 918–928, IEEE, 2004.

- [29] D. Guo *et al.*, "A scalable multithreaded l7-filter design for multi-core servers," in *ANCS'08*, pp. 60–68, ACM, 2008.
- [30] X. Yu, B. Lin, and M. Becchi, "Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence," *Selected Areas in Communications, IEEE Journal on*, vol. 32, pp. 1822–1833, 2014.
- [31] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *SIGCOMM'06*, pp. 339–350, ACM, 2006.
- [32] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *ANCS'07*, pp. 145–154, ACM, 2007.
- [33] C. Liu and J. Wu, "Fast deep packet inspection with a dual finite automata," *Computers, IEEE Transactions on*, vol. 62, no. 2, pp. 310–321, 2013.
- [34] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *SIGCOMM'08*, ACM, 2008.
- [35] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *ACM CoNEXT conference*, p. 1, ACM, 2007.
- [36] S. Kumar *et al.*, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *ANCS'07*, ACM, 2007.
- [37] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *ACM CoNEXT Conference*, p. 25, ACM, 2008.
- [38] Y. Yang and V. K. Prasanna, "Space-time tradeoff in regular expression matching with semi-deterministic finite automata," in *INFOCOM'11*, pp. 1853–1861, IEEE, 2011.
- [39] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 30–39, ACM, 2009.
- [40] X. Yu and M. Becchi, "Gpu acceleration of regular expression matching for large datasets: exploring the implementation space," in *International Conference on Computing Frontiers*, p. 18, ACM, 2013.
- [41] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "infant: Nfa pattern matching on gpgpu devices," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 20–26, 2010.
- [42] Y. Zu *et al.*, "Gpu-based nfa implementation for memory efficient high speed regular expression matching," in *PPoPP'12*, ACM, 2012.
- [43] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ISCA'06*, IEEE Computer Society, 2006.
- [44] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, "Designing a programmable wire-speed regular-expression matching accelerator," in *MICRO'12*, IEEE.
- [45] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *FCCM'01*, IEEE.
- [46] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling pcre to fpga for accelerating snort ids," in *ANCS'07*, ACM.
- [47] Y.-H. Yang and V. K. Prasanna, "High-performance and compact architecture for regular expression matching on fpga," *Computers, IEEE Transactions on*, vol. 61, no. 7, pp. 1013–1025, 2012.
- [48] A. Chien, A. Snaveley, and M. Gahagan, "10x10: A general-purpose architectural approach to heterogeneity and energy efficiency," in *The Third Workshop on Emerging Parallel Architectures at the International Conference on Computational Science*, 2011.
- [49] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, May 2011.
- [50] A. A. Chien, D. Vasudevan, T. Hoang, Y. Fan, and A. Shambayati, "10x10: A case study for federated heterogeneous computing," *University of Chicago Technical Report, TR-2015-08*, March 2015.
- [51] Y. Fang, R. ur Rasool, D. Vasudevan, and A. A. Chien, "Generalized pattern matching micro-engine," in *4th Workshop on Architectures and Systems for Big Data (ASBD) held with ISCA'14*.
- [52] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *ANCS'06*, ACM.
- [53] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," in *4th intl conf on Security and privacy in communication networks*, p. 1, ACM, 2008.
- [54] J. Van Lunteren, "High-performance pattern-matching for intrusion detection," in *INFOCOM'06*, vol. 6, pp. 1–13, 2006.
- [55] Y. Qi *et al.*, "Feacan: Front-end acceleration for content-aware network processing," in *INFOCOM'11*, pp. 2114–2122, IEEE, 2011.
- [56] Y. Fang, A. Lehane, and A. A. Chien, "Effclip: Efficient coupled-linear packing for finite automata," *University of Chicago Technical Report, TR-2015-05*, May 2015.
- [57] J. Wang, J. Sohl, O. Kraigher, and D. Liu, "Software programmable data allocation in multi-bank memory of simd processors," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pp. 28–33, IEEE, 2010.
- [58] D. Moloney *et al.*, *Myriad2 Eye of the Computational-Vision Storm*. Hot Chips'14, 2014.
- [59] T. Hoang, A. Shambayati, and A. A. Chien, "A data layout transformation (dlt) accelerator: Architectural support for data movement optimization in accelerated systems," *University of Chicago Technical Report, TR-2015-02*, March 2015.
- [60] P. Dlugosch *et al.*, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE TPDS'14*, 2014.
- [61] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 79–89, IEEE, 2008.
- [62] L. Barford, "Parallelizing small finite state machines, with application to pulsed signal analysis," in *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*, pp. 1957–1962, IEEE, 2012.
- [63] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *ASPLOS'14*, pp. 529–542, ACM, 2014.
- [64] <http://www.tpc.org/tpch/>.
- [65] <http://www.cs.utah.edu/~rajeew/cacti6/>.
- [66] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.
- [67] S. Borkar, "Exascale computing-a fact or a fiction?," http://www.ipdps.org/ipdps2013/SBorkar_IPDPS_May_2013.pdf.
- [68] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *Hot Chips'11*, vol. 23, pp. 1–24, 2011.
- [69] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira, "Accelerating business analytics applications," in *HPCA'12*, pp. 1–10, IEEE, 2012.
- [70] R. D. Cameron and D. Lin, "Architectural support for swar text processing with parallel bit streams: the inductive doubling principle," in *ASPLOS'09*, ACM, 2009.
- [71] R. D. Cameron *et al.*, "Bitwise data parallelism in regular expression matching," in *PACT'14*, pp. 139–150, ACM, 2014.