ASPEN: A Scalable In-SRAM Architecture for Pushdown Automata

Kevin Angstadt,* Arun Subramaniyan,* Elaheh Sadredini,† Reza Rahimi,†
Kevin Skadron,† Westley Weimer,* Reetuparna Das,*
*Computer Science and Engineering
University of Michigan
Ann Arbor, MI USA
{angstadt, arunsub, weimerw, reetudas}@umich.edu
†Department of Computer Science
University of Virginia
Charlottesville, VA USA
{elaheh, rahimi, skadron}@virginia.edu

***Abstract*—Many applications process some form of tree-structured or recursively-nested data, such as parsing XML or JSON web content as well as various data mining tasks. Typical CPU processing solutions are hindered by branch misprediction penalties while attempting to reconstruct nested structures and also by irregular memory access patterns. Recent work has demonstrated improved performance for many data processing applications through memory-centric automata processing engines. Unfortunately, these architectures do not support a computational model rich enough for tasks such as XML parsing.**

In this paper, we present ASPEN, a general-purpose, scalable, and reconfigurable memory-centric architecture for processing of tree-like data. We take inspiration from previous automata processing architectures, but support the richer deterministic pushdown automata computational model. We propose a custom datapath capable of performing the state matching, stack manipulation, and transition routing operations of pushdown automata, all efficiently stored and computed in memory arrays. Further, we present compilation algorithms for transforming large classes of existing grammars to pushdown automata executable on ASPEN, and demonstrate their effectiveness on four different languages: Cool (object oriented programming), DOT (graph visualization), JSON, and XML.

Finally, we present an empirical evaluation of two application scenarios for ASPEN: XML parsing, and frequent subtree mining. The proposed architecture achieves an average 704.5 ns per KB parsing XML compared to 9983 ns per KB in a state-of-the-art XML parser across 23 benchmarks. We also demonstrate a 37.2x and 6x better end-to-end speedup over CPU and GPU implementations of subtree mining.

***Index Terms*—pushdown automata, emerging technologies (memory and computing), accelerators**

I. Introduction

Processing of tree-structured or recursively-nested data is intrinsic to many computational applications. Data serialization formats such as XML and JSON are inherently nested (with opening and closing tags or braces, respectively), and structures in programming languages, such as arithmetic expressions, form trees of operations. Further, the grammatical structure of English text is tree-like in nature [1]. Reconstructing and validating tree-like data is often referred to as *parsing*.

Studies on data processing and analytics in industry demonstrate both increased rates of data collection and also increased demand for real-time analyses [2], [3]. Therefore, scalable and high-performance techniques for parsing and processing data are needed to keep up with industrial demand. Unfortunately, parsing is an extremely challenging task to accelerate and falls within the "thirteenth dwarf" in the Berkeley parallel computation taxonomy [4]. Software parsing solutions often exhibit irregular data access patterns and branch mispredictions, resulting in poor performance. Custom accelerators exist for particular parsing applications (e.g., for parsing XML [5]), but do not generalize to multiple important problems.

We observe that *deterministic pushdown automata* (DPDA) provide a general-purpose computational model for processing tree-structured data. Pushdown automata extend basic finite automata with a stack. State transitions are determined by both the next input symbol and also the top of stack value. Determinism precludes stack divergence (i.e., simultaneous transitions never result in different stack values) and admits efficient hardware implementation. While somewhat restrictive, we demonstrate that DPDAs are powerful enough to parse most programming languages and serialization formats as well as mine for frequent subtrees within a dataset.

In this paper, we present ASPEN, the Accelerated in-SRAM Pushdown ENgine, a realization of deterministic pushdown automata in Last Level Cache (LLC). Our design is based on the insight that much of the DPDA processing can be architected as LLC SRAM array lookups without involving the CPU. By performing DPDA computation in-cache, ASPEN avoids conventional CPU overheads such as random memory accesses and branch mispredictions. Execution of a DPDA with ASPEN is divided into five stages: (1) input symbol match, (2) stack symbol match, (3) state transition, (4) stack action lookup, and (5) stack update, with each stage making use of SRAM arrays to encode matching and transition operations.

To scale to large DPDAs with thousands of states, ASPEN adopts a hierarchical architecture while still processing one input symbol in one cycle. Further, ASPEN supports processing of hundreds of different DPDAs in parallel as any number of LLC SRAM arrays can be re-purposed for DPDA processing. This feature is critical for applications such as frequent subtree mining which require parsing several trees in parallel.

To support direct adaptation of a large class of legacy parsing applications, we implement a compiler for converting existing grammars for common parser generators to DPDAs executable by ASPEN. We propose two key optimizations for improving the runtime of parsers on ASPEN. First, the architecture supports popping a reconfigurable number of values from the stack in a single cycle, a feature we call

multipop. Second, our compiler implements a state merging algorithm that reduces chains containing ε-transitions. Both of these optimizations reduce stalls in input symbol processing.

To summarize, this work makes the following contributions:

- We propose ASPEN, a scalable execution engine which re-purposes LLC slices for DPDA acceleration.
- We develop a custom data path for DPDA processing using SRAM array lookups. ASPEN implements state matches, state transition, stack updates, includes efficient multipop support, and can parse one token per cycle.
- We develop an optimizing compiler for transforming existing language grammars into DPDAs. Our compiler optimizations reduce the number of stalled cycles during execution. We demonstrate this compilation on four different languages: Cool (object oriented programming), DOT (graph visualization), JSON, and XML.
- We empirically evaluate ASPEN on two application scenarios: a tightly coupled XML tokenizer and parser pipeline and also a highly parallelized subtree miner. First, results demonstrate an average of 704.5 ns per KB parsing XML compared to 9983 ns per KB in a state-of-the-art XML parser across 23 XML benchmarks. Second, we demonstrate 37.2× and 6× better end-to-end speedup over CPU and GPU implementations of subtree mining.

II. Background and Motivation

In this section, we review automata theory relevant to ASPEN. We then introduce two real-world applications that motivate accelerated pushdown automata execution.

A. Automata Primer

A *non-deterministic finite automaton* (NFA) is a state machine represented by a 5-tuple, $(Q, \Sigma, \delta, S, F)$, where Q is a finite set of states, Σ is a finite set of symbols, δ is a transition function, $S \subseteq Q$ are initial states, and $F \subseteq Q$ is a set of final or accepting states. The transition function determines the next states using the current active states and the next input symbol. If the automaton enters into an accept state, the current input position is reported. In a *homogeneous* NFA, all transitions entering a state must occur on the same input symbol [6]. Homogeneous NFAs (and traditional NFAs) are equivalent in representative power to regular expressions [7], [8].

Pushdown automata (PDAs) extend basic finite automata by including a stack memory structure. A PDA is represented by a 6-tuple, $(Q, \Sigma, \Gamma, \delta, S, F)$, where Γ is the finite alphabet of the stack, which need *not* be the same as the input symbol alphabet. The transition function, δ, is extended to consider stack operations. The transition function for a PDA considers the current state, the input symbol, and the top of the stack and returns a new state along with a stack operation (one of: push a specified symbol, pop the top of the stack, or no operation).

B. Deterministic Pushdown Automata

In this paper, we restrict attention to *deterministic pushdown automata* (DPDAs), which limit the transition function to only allow a single transition for any valid configuration of the DPDA and an input symbol. This restriction prevents stack divergence, a property we leverage for efficient implementation in hardware. Some transitions perform stack operations without considering the next input symbol, and we refer to these transitions as *epsilon- or ε-transitions*. To maintain the determinism, all ε-transitions take place before transitions considering the next input symbol.

Unlike basic finite automata, where non-deterministic and deterministic machines have the same representative power (any NFA has an equivalent DFA and vice versa), DPDAs are strictly *weaker* than PDAs [7]. DPDAs, however, are still powerful enough to parse most programming languages and serialization formats as well as mine for frequent subtrees within a dataset. We leave the exploration of hardware implementations of PDAs for future work.

For hardware efficiency, we extend the definition of homogeneous finite automata to DPDA. In a homogeneous DPDA (hDPDA), all transitions to a state occur on the same input character, stack comparison, and stack operation. Concretely, for any $q, q', p, p' \in Q$, $\sigma, \sigma' \in \Sigma$, $\gamma, \gamma' \in \Gamma$, and op, op' that are operations on the stack, if $\delta(q, \sigma, \gamma) = (p, op)$ and $\delta(q', \sigma', \gamma') = (p', op')$, then

$$p = p' \Rightarrow \sigma = \sigma' \wedge \gamma = \gamma' \wedge op = op'.$$

This restriction on the transitions function does not limit computational power, but may increase the number of states needed to represent a particular computation.

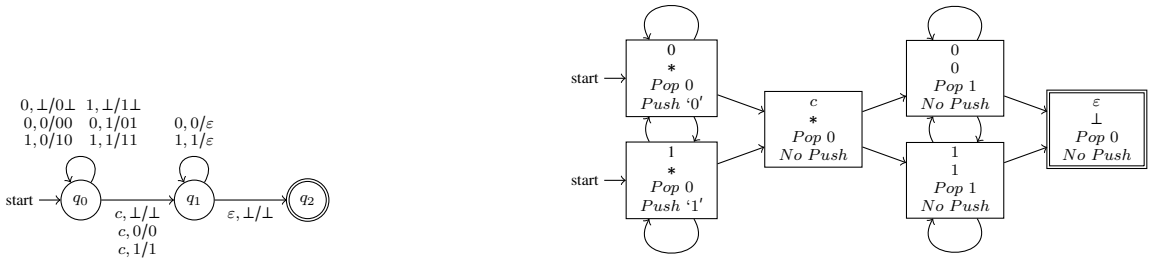
Claim 1. *Given any DPDA $A = (Q, \Sigma, \Gamma, \delta, S, F)$, the number of states in an equivalent hDPDA is bounded by $O(|\Sigma||Q|^2)$.*

Proof. We consider the worst case: A is fully-connected with $|\Sigma| \cdot |Q|$ incident edges to each state and each of these incoming edges performs a different set of input/stack matches and stack operations. Therefore, we must duplicate each node $|\Sigma|(|Q| - 1)$ times to ensure the homogeneity property. For any node $q \in Q$, we add $|\Sigma| \cdot |Q|$ copies of q to the equivalent hDPDA, one node for each of the different input/stack operations on incident edges. Therefore, there are at most $|\Sigma| \cdot |Q| \cdot |Q| = |\Sigma||Q|^2$ vertices in the equivalent hDPDA. □

In practice, DPDAs tend not to be fully-connected and have a fixed alphabet, resulting in less than quadratic growth. Even in the worst case, hDPDAs do not significantly increase the number of states (cf. the exponential NFA to DFA transformation). Figure 1 provides an example DPDA and hDPDA for odd-length palindromes with a known middle character.

C. Parsing of XML Files

A common data processing task that makes use of PDAs is parsing. *Parsing*, or syntactic analysis, is the process of validating and reconstructing tree (nested) data structures from a sequence of input *tokens*. In natural language, this process relates to validating that a sequence of words forms a valid sentence structure, and for a programming language, a parser will verify that a statement has the correct form (e.g., a conditional in C contains the correct keywords, expressions,



(a) (b)

Fig. 1: Equivalent DPDA (a) and hDPDA (b) for recognizing odd-length palindromes with a given center character. For simplicity, we consider strings formed from $\Sigma = \{0, 1\}$ with center character c. Transition rules for the DPDA (a) are written as $a, b/c$, where a is the matched input symbol, b is the matched stack symbol, and c is the top of the stack after a push or ε for a pop. Note that $\perp$ is a special symbol to represent the bottom of the stack. The hDPDA (b) lists (in order) the input symbol match (ε for no match), stack symbol match ($*$ is a wildcard match), number of symbols to pop, and symbol to push.

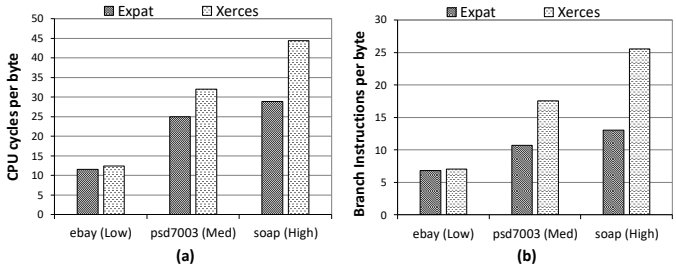


Fig. 2: Conventional parser performance. (a) CPU cycles per byte. (b) Branch instructions per byte

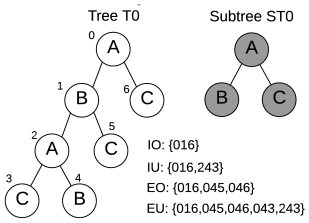


Fig. 3: An example of subtrees (I = Induced, E = Embedded, O = Ordered, U = Unordered)

and statements in the correct order). In this paper, we focus on the task of parsing XML files, which is common to many applications. Parsing XML produces a special tree data structure called the Document Object Model (DOM).

Parsers are typically implemented as the second stage of a larger pipeline. In the first stage, a lexer, tokenizer, or scanner reads raw data and produces a list of tokens (i.e., a lexer converts a stream of characters into a stream of words), which are passed to the parser. The parser produces a tree from these input tokens, which can be further validated and processed by later pipeline stages. For example, an XML parser will validate that tags are properly nested, but a later stage in the pipeline performs semantic checks, such as verifying that text in opening and closing tags match.

Parsing performance on CPUs: Conventional software-based parsers exhibit complex input-dependent data and control flow patterns resulting in poor performance when executed on CPUs. Figure 2 (b) shows two state-of-the-art open-source XML parsers, Expat [9] and Xerces [10], which can require ~6–25 branch instructions to process a single byte-of-input depending on the *markup density* of the input XML file (i.e., ratio of syntactic markup to document size). These overheads result from nested *switch-case* statements that determine the next parsing state. Furthermore, as the parser alternates between markup processing and processing of variable-length content, there is little data reuse, leading to high cache miss rates (~100 L1 caches misses per kB for Xerces). As a result of both high branch misprediction and cache miss rates, software parsers take ~12–45 CPU cycles to parse a single input byte (see Figure 2 (a)). In contrast, ASPEN, by virtue of performing DPDA computation in-cache, does not incur these overheads.

D. Frequent Subtree Mining

Another application that may make use of the DPDA computational model is *frequent subtree mining*. This analysis is used in natural language processing, recommendation systems, improving network packet routing, and querying text databases [11], [12]. The core kernel of this task is subtree inclusion, which we consider next.

Subtree inclusion problem: Assume S and T are two rooted, labeled, and ordered trees. Define $t_1, t_2, ..., t_n$ to be the nodes in T and $s_1, s_2, ..., s_m$ be the nodes in S. Then, S is an *embedded subtree* of T if there are matching labels of the nodes $t_{i_1}, t_{i_2}, ..., t_{i_m}$ such that (1) $label(s_k) = label(t_{i_k})$ for all $k = 1, 2, ..., m$; and (2) for every branch (s_j, s_k) in S, t_{i_j} should be an ancestor of t_{i_k} in T. The latter condition preserves the structure of S in T. We also consider *induced subtrees*, which occur when restricting the ancestor-descendant relationship to parent-child relationships in T for the second condition. Figure 3 shows examples of subtrees. Sadredini et al. [13] proposed an approximate subtree inclusion checking kernel for an NFA hardware accelerator that can lead to false positives. We focus on *exact* subtree inclusion checking, which can use a deterministic pushdown automaton to count the

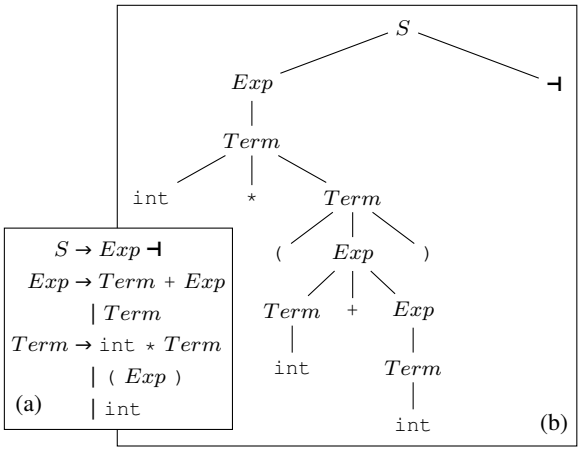


Fig. 4: An example CFG (a) and parse tree (b). The grammar represents a subset of arithmetic expressions. We use ⊣ to signify the endmarker for a given token stream, which is needed for transformation to a DPDA. The parse tree given in (b) is for the expression $3 * (4 + 5)$. Note that integer numbers are transformed to `int` tokens prior to deriving the parse tree.

length of a possible branch when searching for a subtree in the input tree.

III. Compiling Grammars to Pushdown Automata

In this section, we describe context-free grammars, our algorithms to compile such grammars to pushdown automata, and our prototype implementation.

A. Context-Free Grammars

While DPDAs provide a functional definition of computation, it can often be helpful to use a higher-level representation that generates the underlying machine. Just as regular expressions can be used to generate finite automata, *context-free grammars* (CFGs) can be used to generate pushdown automata. We briefly review relevant properties of these grammars (the interested reader is referred to references such as [7], [14]–[16] for additional details).

CFGs allow for the definition of recursive, tree-like structures using a collection of *substitution rules* or *productions*. A production defines how a symbol in the input may be legally rewritten as another sequence of symbols (i.e., the right-hand side of a production may be *substituted* for the symbol given in the left-hand side). Symbols that appear on the left-hand side of productions are referred to as *non-terminals* while symbols that do not are referred to as *terminals*. The *language* of a CFG is the set of all strings produced by recursively applying the productions to a starting symbol until only terminal symbols remain. The sequential application of these productions to an input produces a *derivation* or *parse tree*, where all internal nodes are non-terminals and all leaf nodes are terminals.

An example CFG for a subset of arithmetic operations is given in Figure 4 (a). This particular grammar demonstrates recursive nesting (balanced parentheses), operator precedence (multiplication is more tightly bound than addition), and associativity (multiplication and addition are left-associative in this grammar). Figure 4 (b) depicts the parse tree given by the grammar for the equation $3 * (4 + 5)$.

B. Compiling Grammars to DPDAs

Next, we consider the process of compiling an input CFG to a DPDA. As noted in Section II-A, PDAs and DPDAs do not have equal representative power. Therefore, there are CFGs that cannot be recognized by a DPDA. We focus on support for a strict subset of CFGs known as $LR(1)$ grammars, which are of practical importance and supported by DPDAs. Most programming language grammars have a deterministic representation [7], and many common parser generator tools focus on supporting $LR(1)$ grammars [17]–[19]. By targeting this class of grammars, we can therefore support parsing common languages such as XML, JSON, and ANSI C.

Existing parser generators (e.g., `YACC` or `PLY`) are unsuitable for compiling to ASPEN because these tools do not produce hDPDAs (or even DPDAs!). Instead, they generate source code that makes use of the richer set of operations supported by CPUs. We do, however, demonstrate how existing tools may be leveraged for a portion of our compilation process.

This transformation from grammar to hDPDA is broken down into three stages: (1) parsing automaton generation, (2) hDPDA generation, and (3) optimization.

Parsing Automaton Generation: Parsing of input according to an $LR(1)$ grammar makes use of a DFA known as a *parsing automaton*,[1] a state machine that processes input symbols and determines the next production to apply. This machine encodes *shift* and *reduce* operations. Shifts occur when another input token is needed to determine the next production and are encoded as transitions between states in the parsing automaton. Reduce operations (the reverse applications of productions) occur when the machine has seen enough input to determine which substitution rule in the grammar to apply and are encoded as accepting states in the DFA. Each accepting state represents a different production. Determining the correct shift or reduce operation may require inspecting the current input symbol and also a subsequent *lookahead* symbol.

We leverage off-the-shelf tools to generate parsing automata. Concretely, we support parsing automata generated by the `GNU Bison`[2] and `PLY`[3] parser generator tools. These two tools produce CPU-based parsers and generate parsing automata as an intermediate output.

Conceptually, parsing proceeds by processing input symbols using the parsing automaton and pushing symbols to the stack until an accepting state is reached. The input string is rewritten by popping symbols from the stack. The most recently-pushed symbols are replaced by the left-hand-side of the discovered substitution rule. Processing is then restarted from the beginning of the rewritten input, repeating until only the starting non-terminal symbol remains. With this classical approach,

[1] Also referred to as DK in the literature after its creator, Donald Knuth [7].
[2] https://www.gnu.org/software/bison/
[3] http://www.dabeaz.com/ply/

parsing requires multiple iterations over (and transformations to) the input symbols.

hDPDA Generation: To improve the efficiency of parsing, we simulate the execution of the parsing automaton using a DPDA [7, Lemmas 2.58, 2.67] to process input tokens in a single pass with no transformations to the input. With this approach, input symbols are *not* pushed to the stack. Instead, the stack of the hDPDA is used to track the sequence of states visited in the parsing automaton. Shift operations push the destination parsing automaton state to the stack (shifts are transitions to other states in the parsing automaton). When a reduce operation rewriting n symbols to a single non-terminal symbol is performed by the parsing automaton, the hDPDA pops n symbols off the stack. The symbol at the top of the hDPDA stack is the state of the parsing automaton that immediately preceded the shift of the first token from the reduced rule. In other words, popping the stack for a reduction "runs the parsing automaton in reverse" to undo shifting the symbols from the matched rule. The hDPDA then continues simulation of the parsing automaton from this restored state.

Our prototype compiler generates an hDPDA by first reading in the textual description of the parsing automaton generated by `Bison` or `PLY`. Next, for each state in the parsing automaton, we generate hDPDA states for each terminal and non-terminal in the grammar. A separate state is needed for each terminal and non-terminal symbol because the homogeneity property only supports a single pushdown automata operation per state (see Section II-A):

- For each terminal symbol, we generate two states: one state matches the lookahead symbol (i.e., lookahead symbols are stored in "positional" memory) and one state encodes the relevant shift or reduce operation. A shift operation pushes parsing automaton states on the stack, while a reduce operation pops a symbol from the stack and generates an output signal.
- For each non-terminal symbol, only one state is generated: the state performing the shift/reduce operation. In addition, this state must also match the top of the stack to validate undoing shift operations.

Then, we add additional states to perform stack pop operations for the reduce operations, one pop for each symbol reduced from the right-hand side of a production. Finally, we connect the states with transitions according to transition rules from the parsing automaton.

The final hDPDA is emitted in the MNRL file format. MNRL is an open-source JSON-based state machine serialization format that is used within the MNCaRT automata processing and research ecosystem [20]. We extend the MNRL schema to support hDPDA states, encoding the stack operations with each state. Using MNRL admits the reuse of many analyses from MNCaRT with minimal modification.

Optimization: While our algorithm to transform the parsing automaton to a DPDA is direct, the resulting DPDA contains a large number of ε-transitions and extraneous states. First, we remove all unreachable states (states with no incoming transitions). Then, we perform optimizations to reduce the total number of ε-transitions within the hDPDA. Recall that ε-transitions occur when stack operations take place without reading additional input (e.g., when popping the stack during a reduce operation and transitioning to another state). We make two observations about the hDPDA produced by our compilation algorithm.

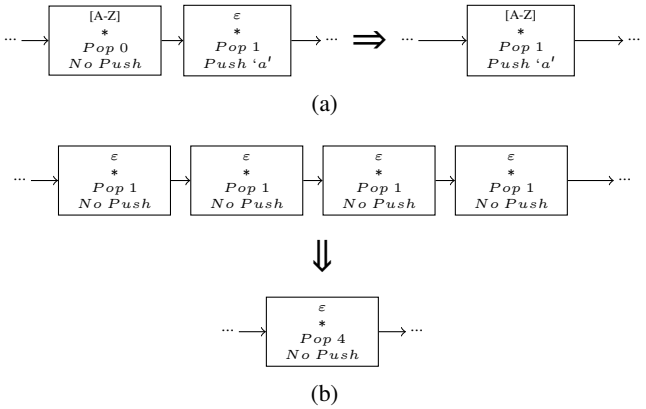


Fig. 5: Two compiler optimizations for reducing the number stalls incurred by ε-transitions. Epsilon merging (a) attempts to combine states to perform non-overlapping operations. Multipop (b) allows for the stack pointer to be moved a configurable distance in one operation.

First, the algorithm produces separate states to "read in" input symbols and to perform stack operations. In many cases, these states may be combined, or merged, to match the input and perform stack operations simultaneously. After producing the initial hDPDA, we perform a post-order depth-first traversal of the machine and merge such connected states when possible. We call this optimization *epsilon merging* and apply it conservatively: only states that occur on a linear chain are merged. Figure 5 (a) shows an example in which a state performing input matching on capital letters and a state (with no input comparison) performing a pop and a push are merged.

Second, our basic algorithm assumes a computational model that only supports popping one symbol at a time. On reduction operations for productions containing several symbols on the right-hand side, this results in long-duration stalls. Note, however, that no comparisons are made with these intermediate stack symbols. If our architecture can support moving the stack pointer by a variable amount, then a reduction may be performed in one step. We refer to this as *multipop*. Figure 5 (b) demonstrates a reduction of four states to one state with multipop.

C. Compilation Summary

We presented an overview of CFGs, a high-level language representation that may be used to generate pushdown automata. Then, we described an algorithm for compiling an important subset of CFGs ($LR(1)$ grammars) to hDPDAs. We leverage existing tools to produce an intermediate parser representation (the parsing automaton), which we then encode in an hDPDA for execution with ASPEN. We also introduce

two optimizations, epsilon merging and multipop, to reduce stalls while processing input. Our approach supports and accelerates existing parser specifications without modification. This means that parsers do not have to be redesigned to take advantage of ASPEN's increased parsing performance.

IV. Architectural Design

In this section, we describe the ASPEN architecture that augments LLC slices with support for DPDA processing. We also discuss the design of a DPDA processing pipeline based on ASPEN and the tradeoffs involved.

A. Cache Slice Design

The proposed ASPEN architecture augments the last level cache slices of a general purpose processor to support in-situ DPDA processing. Figure 6 (a) shows an 8-core enterprise Xeon-E5 processor with LLC slices connected using a ring interconnect (not shown in figure). Typically, the Intel Xeon family includes 8-16 such slices [21]–[23]. Each last-level cache slice macro is 2.5 MB and consists of a centralized cache control box (CBOX). A slice is organized into 20 ways, with each way further organized as five 32 kB *banks*, four of which constitute data arrays, while the fifth one is used to store the tag, valid and LRU state (Figure 6 (b)). All the ways of the cache are interconnected using a hierarchical bus supporting a bandwidth of 32 bytes per cycle. Internally, each bank consists of four 8 kB SRAM arrays (256×256).

A bank can accommodate up to 256 states and a DPDA can span several banks. We repurpose two of the four arrays in each bank to perform the different stages of DPDA processing. The remaining two arrays (addressed by the `PA[16]` bit) can be used to store regular cache data. State-transitions are encoded in a hierarchical memory-based interconnect, consisting of local and global crossbar switches (L-switch, G-switch). A 256-bit register is used to track the active states in each cycle (Active State Vector in Figure 6 (c)). We provision input buffers in the C-BOX to broadcast input symbols or tokens to different banks. Output buffers are also provided to track the report events generated every processing cycle.

B. Operation

This subsection provides the details of DPDA processing. Recall that, in a DPDA, only a single state is active in every processing cycle, and initially, only the start state is active. Each input symbol from the DPDA input buffer is processed in five phases. In the *input match* and *stack match* phases, we identify the active DPDA state which has the same label as that of the input symbol and the top of stack (TOS) symbol respectively. In the *stack action lookup* phase, the stack action defined for that state is determined (i.e., push symbol or number of symbols to pop from the stack). The stack is updated in the following phase (*stack update*). Finally, in the *state-transition phase*, a hierarchical transition interconnect matrix determines the next active state.

Cycles in which states with an ε-transition are active require special handling. These states do not consume an input symbol but perform a stack action in that cycle (i.e., push or pop). A 256-bit ε-mask register tracks the ε-states in each bank. A logical AND of the ε-mask register and Active State Vector is used to determine if an ε-state is active in the next processing cycle. If an ε-state is active, a 1-bit ε-stall signal is sent to the C-BOX to stall the input for the next processing cycle.

While a single stack action per cycle is sufficient to support DPDA functionality, reducing stalls to the input stream can significantly improve performance. The *multipop* optimization, discussed in Section III-B, reduces stalls due to ε-transitions and is supported in hardware by manipulating the stack pointer and encoding the number of popped symbols in the stack action lookup phase. We now proceed to discuss the different stages involved in DPDA processing.

(1) Input-Match (IM): We adapt the state-match design of memory-centric automata processing models [24], [25] for the input-match phase. Each state is mapped to a column of an SRAM array as shown in Figure 6 (c). A state is given a 256-bit input symbol label which is the one-hot encoding of the ASCII symbol that it matches against. The homogeneous representation of DPDA states ensures that each state matches a single input symbol and each state can be represented using a single SRAM column. The input symbol is broadcast as the row address to the SRAM arrays using 8-bits of global wires. By reading out the contents of the row into the *Input Match Vector*, the set of states with the same label as the input symbol can be determined in parallel.

(2) Stack-Match (SM): In contrast to NFAs, where all active states that match the input symbol are candidates for state-transition, DPDA states have valid transitions defined only for those states that match *both* the input symbol and the symbol on the top of the stack (8-bit TOS in Figure 6). We re-purpose an SRAM array in each bank to determine the set of DPDA states that match the top of stack (TOS) symbol. Similar to Input-Match, we provision 8 bits of global wires to broadcast the TOS symbol as the row address to SRAM arrays. By reading out the contents of the row into the *TOS Match Vector* and performing a logical AND with the *Input Match Vector* and the *Active State Vector*, the candidate states for state-transition are determined. We refer to these candidate states simply as *active states*.

We leverage sense-amplifier cycling techniques [25] to accelerate the IM and SM stages.

(3) Stack Action Lookup (AL): Each DPDA state is also associated with a corresponding stack action. The supported stack actions are `push`, `pop` and `multipop`. The stack action is encoded with 16 bits. Each `push` action uses 8 bits to indicate the symbol to be pushed onto the stack. The remaining 8 bits are used by the `pop` action to indicate the number of symbols to be popped from the stack (> 1 for `multipop`).

The stack action corresponding to each state is packed along with the IM SRAM array in each bank. However, in the AL stage, we lookup this SRAM array using the 256-bit result vector obtained after logical AND in the previous step (see Figure 6). This removes the decoding overhead from the array access time. We reserve 16 bits of global wires to communicate

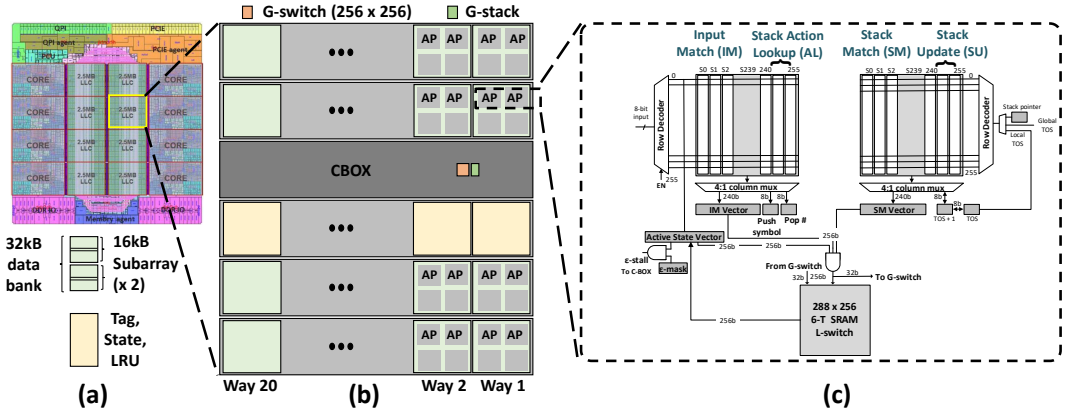


Fig. 6: The figure shows (a) 8-core Xeon processor, (b) one 2.5MB Last-Level Cache (LLC) slice and (c) Internal organization of one 32kB bank with two 8kB SRAM arrays repurposed for DPDA processing.

the stack action results from each bank to the stack control logic in the C-BOX.

(4) State Transition (ST): The state-transition phase determines the set of states to be activated in the next cycle. We observe that the state transition function can be compactly encoded using a hierarchy of local and global memory-based crossbar switches. The state transition interconnect is designed to be flexible and scales to several thousand states. The L-switches provide dense connectivity between states mapped to the same bank while the G-switch provides sparse connectivity between states mapped to multiple banks. A graph partitioning based algorithm [26] is used to satisfy the local and global connectivity constraints while maximizing space utilization.

The crossbar switches consisting of N input and output ports and N×N cross-points are implemented using regular 6-T SRAM arrays (e.g., L-switch in Figure 6 (c)). The 6-T bitcell holds the state of each cross-point. A flip-flop or register can also be used for this purpose but these are typically implemented using 24 transistors making them area inefficient. A '1' is stored in bitcell (i, j) if there is a valid transition defined from state i to state j. All the cross-points are programmed once during initialization and used for processing several MBs to GBs of input symbols. The set of *active* states from the previous phase serve as inputs to the crossbar switch. For DPDAs, only a single state can be active every cycle and we can use 6-T SRAM arrays for state transition, since only a single row is activated.

(5) Stack Update (SU): To allow for parallel processing of small DPDAs, (e.g., in subtree mining), we provide a local stack in each bank. We repurpose 8 columns of the SM array to accommodate the local stack. Larger DPDAs (e.g., in XML parsing) make use of a global stack to keep track of parsing state. The global stack is implemented in the C-BOX using a 256×8 register file and is shared by all the DPDAs mapped to two adjacent ways. Providing a stack depth of 256 is sufficient for our parsing applications (see Section VI). Note that only one sort of stack (local or global) is enabled at configuration time based on the DPDA size. The stack pointer is stored in an 8-bit register and is used to address the stack. We also store

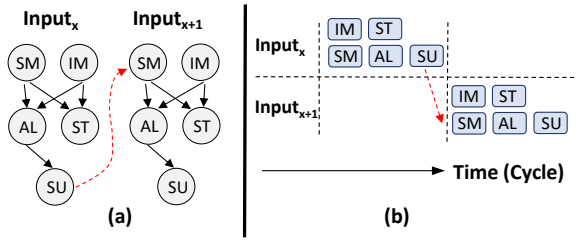


Fig. 7: DPDA processing on ASPEN. (a) Dependency graph between stages. (b) Serial processing of input symbols.

the symbols at stack positions TOS and TOS+1 in separate 8-bit registers. This optimization saves a write and read access to the larger stack register file and ensures early availability of the top-of-stack symbol for the next processing cycle. The `push` operation writes the stack symbol to TOS+1. A lazy mechanism is used to update the stack with the contents of TOS. Similarly, the `pop` operation copies TOS to TOS+1, while lazily reading the stack register file to update TOS.

C. Critical Path

ASPEN's performance depends on two critical factors: (1) the time taken to process each symbol in the input stream (i.e., clock period) and (2) the time spent stalling due to ε-transitions. The multipop optimization reduces stalls due to ε-transitions. We now consider the clock period.

In a naïve approach, each input symbol would be processed sequentially in five phases, leading to a significant increase in the clock period. However, not all phases are dependent on each other and need to be performed sequentially. Figure 7 (a) shows the dependency graph for the DPDA processing stages. The intra-symbol dependencies are shown in black, while the inter-symbol dependencies are marked in red. Using the dependency graph, each of the five stages can be scheduled as shown in Figure 7 (b), where the propagation through the interconnect (wire and switches) for state-transition is overlapped with stack action lookup and stack update. Since the top of stack cannot be determined until the stack has been updated based on the previous input symbol, DPDA processing is serial. We contrast this to NFA processing, which has two

independent stages (input-match and state-transition) which can be overlapped to design a two-stage pipeline [25]. We find that the critical path delay (clock period) of ASPEN is the time spent for input/stack-match and the time taken for stack action lookup and update. The time spent in state-transition is fully overlapped with stack related operations. Section V-B discusses the pipeline stage delays and operating frequency.

D. Support for Lexical Analysis

Two critical steps in parsing are *lexical analysis*, which partitions the input character stream to generate a token stream, and *parsing*, where different grammar rules are applied to verify the well-formedness of the input tokens (see Section II-C). ASPEN can accelerate both these phases. We leverage the NFA-computing capabilities of the Cache Automaton architecture [25] for lexical analysis. To identify the longest matching token, we run each NFA until there are no active states. When the Active State Vector is zero, a *state exhaustion* signal is sent to the lexer control logic in the C-BOX. The symbol cycle and reporting state ID of the most recent report are tracked in a 64-bit report register in the C-BOX. A 256-bit reporting mask register is used to mask out certain reports based on lexer state. On receiving the *state exhaustion* signal from all banks, the lexer control logic resets the reporting mask, reloads the NFA input buffer for the next token and generates a token stream to be written into the DPDA input buffer (using a lookup table to convert report codes to tokens).

E. System Integration

ASPEN shares the last level cache with other CPU processes. By restricting DPDA computation to only 8 ways of an LLC slice, we allow for regular operation in other ways. Furthermore, the cache ways dedicated to ASPEN may be used as regular cache ways for non-parsing workloads. Cache access latency is unaffected since DPDA-related routing logic uses additional wires in the global metal layers.

DPDAs are (1) placed and routed for ASPENs hardware resources, and (2) stored as a bitmap containing states and stack actions. At runtime, the driver loads these binaries into cache arrays and memory mapped switches using standard load instructions and Intel Cache Allocation Technology [27]. The input/output buffers for ASPEN are also memory-mapped to facilitate input streaming and output reporting, and ISA extensions are used to start/stop DPDA functions. We disable LLC slice hashing at configuration time. The configuration overheads are small, especially when processing MBs or GBs of input, but are included in our reported results. To support automata-based applications that require counting, we provision four 16-bit counters per way of the LLC.

Post-processing of output reports takes place on the CPU. For XML parsing pipelines, a DOM tree representation (see Section II-C) can be constructed by performing a linear pass over the DPDA reports. Richer analyses (such as verifying opening and closing tags match for XML parsing supporting arbitrary tags) may be implemented as part of tree construction. Although the CPU-ASPEN pipeline can support this, we leave evaluation of DOM tree construction for future work.

TABLE I: Subtree Mining Datasets

Dataset	#Trees	Ave_Node	#Items	Max_Depth	#Subtrees
T1M	1M	5.5	500	13	9825
T2M	2M	2.95	100	13	3711
TREEBANK	52581	68.03	1387266	38	5280

Ave_Node = Average number of nodes per tree
#Items = Frequent label set size
Max_Depth = Maximum tree depth in the dataset

V. Experimental Methodology

A. Experimental Setup

We describe our XML parsing workload, followed by frequent subtree mining. All CPU-based evaluations use a 2.6 GHz dual-socket Intel Xeon E5-2697-v3 with 28 cores in total, GPU-based evaluations use NVIDIAs's TITAN Xp. We used PAPI [28] and Intel's RAPL tool [29] to obtain performance and power measurements and NVIDIA's *nvprof* utility [30] to profile the GPU. We utilize the METIS graph partitioning framework [26] to map DPDA states to cache arrays.

XML Parsing: We evaluate ASPEN against the widely-used open-source XML tools Expat (v.2.0.1) [9], a non-validating parser, and Xerces-C (v.3.1.1) [10], a validating parser and part of the Apache project. The validation application used is *SAXCount*, which verifies the syntactic correctness of the input XML document and returns a count of the number of elements, attributes and content bytes. We restrict our analysis to the SAX interface and WFXML scanner of Xerces-C and filter out all non-ASCII characters in the input document. We do not include DOM tree generation in our evaluation. This is consistent with prior work and evaluations (e.g., Parabix, Xerces SAX, and Expat). We assume that input data is already loaded into main memory. Our XML benchmark dataset is derived from Parabix [31], Ximpleware [32] and the UW XML repository [33]. We only evaluate XML files larger than 512 kB in size, as we were unable to obtain reliable energy estimations when baseline benchmark execution time was under 1 ms. To evaluate the lexing-parsing pipeline, we extend the open-source, cycle-accurate virtual automata simulator, *VASim* [34], to support DPDA computation and derive per-cycle statistics. The tight integration of the lexer and parser in the LLC enables ASPEN to largely overlap the parsing time. Each lexing report can be processed and used to generate the token stream for the DPDA in 2 cycles.

Frequent Subtree Mining: We compare ASPEN against TreeMatcher [35], a single-threaded CPU implementation, and GPUTreeMiner [13], a GPU implementation. Both employ a breadth-first iterative search to find frequent subtrees. We evaluate using three different datasets, one real-world (TREEBANK[4]), and two synthetically generated by the tree generation program provided by Zaki[5] (T1M and T2M). Table I shows the details of the datasets. TREEBANK is widely used

[4] http:// www.cs.washington.edu/research/xmldatasets/

[5] http://www.cs.rpi.edu/~zaki/software/

TABLE II: Stage Delays and Operating Frequencies

Design	IM/SM	ST	AL	SU	Max Freq.	Freq Oper.
ASPEN	438 ps	573 ps	349 ps	349 ps	880 MHz	850 MHz
CA	250 ps	250 ps	-	-	4 GHz	3.4 GHz

in computational linguistics and consists of XML documents. It provides a syntactic structure for English text and uses part-of-speech tags to represent the hierarchical structure of the sentences. T1M and T2M are generated based on a mother tree with a maximal depth and fan-out of 10. The total number of nodes in T1M and T2M are 1,000,000 and 100,000, respectively. The datasets are then generated by creating subtrees of the mother tree. First, the database is converted to a preorder traversal labelled sequence representation. Then, for each subtree node, depending on its label and position, a set of predefined rules determines the corresponding DPDA. Detailed information on these rules can be found in Iváncsy and Vajk [36]. The total number of subtrees summed over all the iterations of the frequent subtree mining problem is given in the *#Subtrees* column.

B. ASPEN parameters

Each 256×256 6-T SRAM array in the Xeon LLC can operate at 4 GHz [22], [23]. In the absence of publicly-available data on array area and energy, we use the standard foundry memory compiler at 0.9 V in the 28nm technology node to estimate the power and area of a 256×256 6-T SRAM array. The energy to read out all 256 bits was calculated as 22 pJ. Since ASPEN is based on a Xeon-E5 processor modeled at 22nm, we scale down the energy per access to 13.6 pJ. The area of each array and 6-T crossbar switch were estimated to be 0.015 mm^2 and 0.017 mm^2 respectively. Each LLC slice contains 32 L-switches and 4 G-switches to support DPDA computation in up to 8 ways. These switches can leverage standard 6-T SRAM push-rules to achieve a compact layout and have low area overhead (~6.4% of LLC slice area). Being 6-T SRAM based, these switches can also be used to store regular data when not performing DPDA computation. Similar to the Cache Automaton [25], we use global wires to broadcast input/stack symbols and propagate state transition signals. These global wires with repeaters have a 66ps/mm delay and an energy consumption of 0.07pJ/mm/bit.

Table II shows the stage delays for DPDA processing on ASPEN. The IM/TM phases leverage sense-amplifier cycling [25] and take 438 ps. The ST stage requires 573 ps, composed of 198 ps wire delay and 375 ps due to local and global switch traversal. AL and SU each take 349 ps, composed of 99 ps wire delay and 250 ps for array access.

VI. Evaluation

In this section, we evaluate ASPEN on real-world applications with indicative workloads. First we evaluate the generality of ASPEN and our proposed optimizations by compiling several parsers for the architecture. Second, we evaluate runtime and energy for our two motivating applications.

TABLE III: Description of Grammars

Language	Description	Token Types	Grammar Productions	Parsing Aut. States
Cool	Programming language	42	61	147
DOT	Graph visualization	22	53	81
JSON	Data interchange	13	19	29
XML	Data interchange	13	31	64

TABLE IV: Compilation Results. Our optimizations reduce the number of epsilon states by an average of 65%.

Language	Optimizations	hDPDA States	Epsilon States	Average Compilation Time (sec)
Cool	None	3505	2733	0.88
	Mutlipop + Eps	1666	894	2.75
DOT	None	1690	1494	0.34
	Mutlipop + Eps	1062	866	0.98
JSON	None	764	619	0.16
	Mutlipop + Eps	461	316	0.5
XML	None	2068	1653	0.36
	Mutlipop + Eps	865	450	0.88

A. Parsing Generality

We first demonstrate compilation of four different languages: *Cool*, an object oriented programming language[6]; *DOT*, the language used by the GraphViz graph visualization tool [37]; *JSON*; and *XML*. We selected these benchmarks because grammar specifications (for either `PLY` or `Bison`) were readily available. Importantly, no modification to existing legacy grammars was necessary to support compilation to ASPEN. The architecture is general-purpose enough to support these diverse applications, and our prototype compiler supports a large class of existing parsers. Details for each of these languages, including number of token types, number of grammar rules, and the size of the parsing automaton, are provided in Table III. Higher numbers of parsing automata states (see Section III) indicate more complex computation for determining which grammar production rule to apply. This complexity is related both to the number of token types as well as the total number of productions in the grammar.

In Table IV, we present compilation statistics using our prototype compiler. We report the average time across ten runs of our compiler and optimizations. Compilation of all grammars, including optimization, is well below 5 seconds, meaning that compilation of grammars is not a significant bottleneck with ASPEN. With both our multipop and epsilon reduction optimizations enabled, we observe a 47%, on average, decrease in the number of states. The number of epsilon states is reduced by 65% on average. As noted in Section III, reducing epsilon states reduces input stalls. Note that the numbers reported here are prior to placement and routing of the design for ASPEN. The final hDPDA may contain more states to reduce fan-in or fan-out complexity; however, the length of epsilon chains will neither increase nor decrease.

Next, we evaluate the performance of XML parsing using our compiled XML grammar. While we expect performance

[6] https://en.wikipedia.org/wiki/Cool_(programming_language)

results to generalize, for space considerations, we do not evaluate the other parsers in detail.

B. XML Parsing

Using the graph partitioning framework METIS, we find that the XML parser hDPDA (with optimizations) maps to 8 cache arrays and results in an LLC cache occupancy of 128KB.

Figure 8 compares ASPEN's performance and energy against Expat and Xerces on the SAXCount application (lower is better). We evaluate two DPDA configurations: (1) ASPEN-MP has both multipop and epsilon merging optimizations enabled and (2) ASPEN, which only enables epsilon merging. We group our XML datasets based on markup density which is an indirect measure of XML document complexity. Performance of Expat and Xerces drops as the markup density of the input XML document increases, because complex documents tend to produce a large number of tokens for verification. ASPEN also sees a slight increase in runtime with increase in markup density, but the dependence is less pronounced. There is a noticeable trend in performance and energy benefits of ASPEN-MP over ASPEN as markup density increases. As the density increases, tokens are generated more frequently, and ε-transition stalls are less likely to be masked by the tokenization stage of the pipeline. ASPEN-MP reduces the number of stalling cycles during parsing, thus improving performance with high markup density. ASPEN-MP achieves 30% improvement in both performance and energy over ASPEN.

Overall, averaged across the datasets evaluated, ASPEN-MP takes 704.5 ns/kB and consumes 20.9 μJ/kB energy. When compared with Expat, a 14.1× speedup and 13.7× energy saving is achieved. ASPEN-MP also achieves 18.5× speedup and consumes 16.9× lower energy than Xerces for SAXCount. Even after considering the idle power of the CPU core, XML parsing on ASPEN takes 20.15 W, which is well within the TDP of the Xeon-E5 processor core (160 W). The low power consumed can be attributed to: (1) removal of data movement and instruction processing overheads present in a conventional core, and (2) only a single bank of the cache being active in any processing cycle, due to the deterministic nature of the automaton, resulting in energy savings.

C. Subtree Inclusion

To evaluate the benefits of DPDA hardware acceleration for the subtree inclusion kernel, we consider the frequent subtree mining (FTM) problem, where the major computation is subtree inclusion checking. FTM is composed of two steps. In the first step, the subtree candidates of size $k+1$ (*(k+1)-candidates*) are generated from the frequent candidates of size k (*k-frequent-candidates*), where k is the number of nodes in a subtree. Candidate-generation details and a proof of correctness are provided by Zaki [35]. In the second stage, for each candidate subtree, we count the number of occurrences (inclusions) of that subtree in the dataset. If the count exceeds a specified support threshold, we report the candidate as frequent and use it as a seed in the next generation step.

TABLE V: Architectural Parameters for Subtree Inclusion

Dataset	Automata Alphabets	Stack Alphabets	Stack-Size
T1M	16	17	29
T2M	38	39	49
TREEBANK	100	101	110

Table V lists the architectural parameters for the FTM application on different datasets. This application is compatible with the hardware restrictions, including maximum stack depth and supported alphabet size. In contrast to XML parsing, there are no ε-transitions in the subtree inclusion DPDAs, which means that runtime is linear in the length of the input data. The homogeneous DPDAs designed for FTM have an average node fan-out of 2.2 (maximum of 4).

Figure 9 shows the kernel and total speedup of ASPEN over CPU and GPU baselines. For ASPEN, we include timing for pre-processing, intermediate processing (between iterations) on the CPU, loading time (transferring data from DRAM to LLC), and reporting time (moving report vectors back to DRAM), in addition to the kernel time.

ASPEN shows 67.2× and 6× end-to-end performance improvement over CPU and GPU (Figure 9). TREEBANK consists of larger trees with higher average node out-degree, which makes its processing difficult on the CPU and GPU. In particular, TREEBANK has an uneven distribution of trees with different sizes in the database, which causes the synchronization overhead between the threads in a warp to increase. In addition, larger trees also increase the thread divergence in a warp, because the possibility of checking a subtree node against different labels in the input tree of the same warp increases. Therefore, GPUs are not an attractive solution for larger trees. On the other hand, GPUs show 2× speedup over ASPEN on T1M. This is because the T1M dataset consists of small and evenly sized trees. Unlike CPUs or GPUs, the complexity of subtree inclusion checking in ASPEN is independent of the input dataset.

Figure 10 shows the total energy for ASPEN, CPU, and GPU implementations. The trends in energy are similar to that of performance. The unevenly-sized large trees in TREEBANK increase the runtime of CPU and GPU, leading to an increase in total energy. On average, ASPEN achieves 3070× and 6279× improvements in total energy when compared to CPU- and GPU-based implementations, respectively.

VII. Related Work

To the best of our knowledge, this is the first work that demonstrates and evaluates pushdown automata processing implemented in last-level cache. We position our work in context with respect to related efforts and approaches.

Finite Automata Accelerators: A recent body of work studies the acceleration of finite automata (NFA and DFA) processing across multiple architectures. Becchi et al. have developed a set of tools and algorithms for efficient CPU-based automata processing [38]. Several regular-expression-matching and DFA-processing ASIC designs have also been proposed [39]–[42]. Some (e.g., [43]) incorporate regular expression matching into an extract-transform-load pipeline, sup-

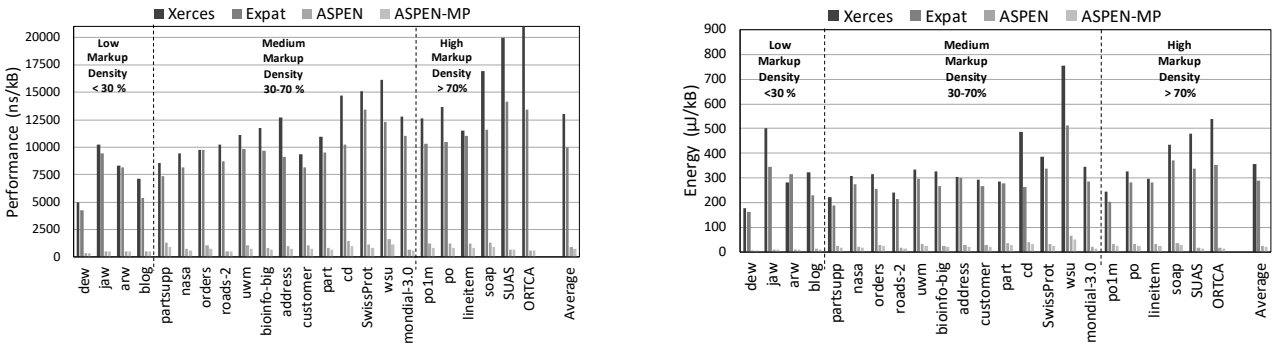


Fig. 8: (Left): ASPEN performance (in ns/kB) and (Right): ASPEN energy on SAXCount compared to Expat and Xerces.

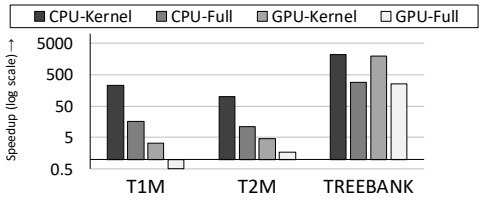


Fig. 9: Speedup of ASPEN over CPU and GPU.

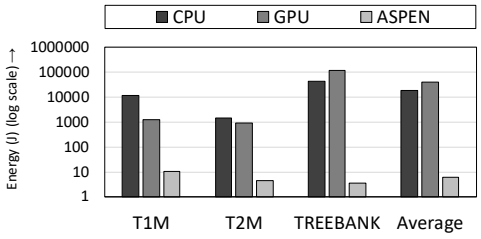


Fig. 10: Total Energy of ASPEN compared to CPU and GPU.

porting a richer set of applications. Most closely-related to ASPEN are memory-centric architectures for automata processing, such as Micron's D480 Automata Processor (AP) [24], Parallel Automata Processor (PAP) [44], Subramaniyan et al.'s Cache Automaton (CA) [25], and Xie et al.'s REAPR [45]. While the performance of these finite automata processing accelerators is quite promising, the underlying computational model is not rich enough to support the parsing or mining applications discussed in this paper. Finite automata processors can perform regular expression processing, but lack the stack memory needed for nested data structures.

Software Parser Generators: CPU algorithms for parsing input data according to a grammar have remained largely unchanged over the years. These algorithms generate a parsing automaton (as described in Section III), which is encoded as a lookup table. Early optimization efforts focused on compression algorithms to allow the lookup tables to fit in the limited memory of period systems [46]–[48]. There have also been efforts to support different classes of grammars, such as Generalized LR, and $LL(k)$ [49], [50]. These parser generators, however, typically require grammars to be rewritten and redesigned, precluding legacy support.

XML-Parsing Acceleration: Our evaluation demonstrates that ASPEN is competitive with custom XML accelerators [5], [51], [52] which achieve at best 4096 ns/KB. Moreover, ASPEN supports more applications than just XML parsing, which we demonstrate in this paper by compiling parsers for several languages and also evaluating subtree inclusion. Generality is preferable in a datacenter setting, where compute resources are rented to clients and more than one parsing application is likely to be performed. ASPEN is derived by re-purposing cache arrays and can be used as additional cache capacity for applications that do not use pushdown automata. Parabix [31] is a programming framework that also supports acceleration of XML parsing and achieves 1063 ns/KB (with 2.6 GHz CPU), while ASPEN achieves 709.5 ns/KB. Parabix also often requires a redesign of grammar specifications for use with parsing. Orthogonal to the proposed work, Ogden et al. [53] propose an enumerative parallelization approach for XML stream processing that can also benefit ASPEN.

Subtree Inclusion: Recently, Sadredini et al. [13] proposed an approximate subtree inclusion checking kernel on the AP. The authors convert the tree structure to a set of simpler sequence representations and use the AP to prune the huge candidate search space. This approach may introduce a small percentage of false positives. ASPEN performs exact subtree mining and therefore produces no false positives.

VIII. Conclusion

We present ASPEN, a general-purpose, scalable, and reconfigurable memory-centric architecture that supports rich push-down automata processing for tree-like data. We design a custom datapath that performs state matching, stack update, and transition routing using memory arrays. We also develop a compiler for transforming large classes of existing grammars to pushdown automata executable on ASPEN.

Our evaluation against state-of-the-art CPU and GPU tools shows that our approach is general (supporting multiple languages and kernel tasks), highly performant (up to 18.5× faster for parsing and 37.2× faster for subtree inclusion), and energy efficient (up to 16.9× lower for parsing and 3070× lower for subtree inclusion). By providing hardware support for DPDA, ASPEN brings the efficiency of recent automata acceleration approaches to a new class of applications.

Acknowledgments

This work is funded, in part, by the NSF (1763674, 1619098, CAREER-1652294 and CCF-1629450); Air Force (FA8750-17-2-0079); and CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] N. Chomsky and G. A. Miller, "Introduction to the formal analysis of natural languages," in *Handbook of Mathematical Psychology*, 1963, vol. 2, ch. 11, pp. 269–322.

[2] Computer Sciences Corporation, "Big data universe beginning to explode," http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode, 2012.

[3] DNV GL, "Are you able to leverage big data to boost your productivity and value creation?" https://www.dnvgl.com/assurance/viewpoint/viewpoint-surveys/big-data.html, 2016.

[4] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[5] Z. Dai, N. Ni, and J. Zhu, "A 1 cycle-per-byte xml parsing accelerator," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010.

[6] V. M. Glushkov, "The abstract theory of automata," *Russian Mathematical Surveys*, 1961.

[7] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. Cengage Learning, 2013.

[8] P. Caron and D. Ziadi, "Characterization of Glushkov automata," *Theoretical Computer Science*, vol. 233, 2000.

[9] J. Clark, "The Expat XML parser," http://expat.sourceforge.net.

[10] A. S. Foundation, "Xerces C++ XML parser," http://xerces.apache.org/xerces-c/.

[11] P. Kilpeläinen *et al.*, "Tree matching problems with applications to structured text databases," 1992.

[12] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok, "Frequent subtree mining—an overview," *Fundamenta Informaticae*, vol. 66, 2005.

[13] E. Sadredini, R. Rahimi, K. Wang, and K. Skadron, "Frequent subtree mining on the automata processor: challenges and opportunities," in *International Conference on Supercomputing*, 2017.

[14] S. A. Greibach, "A new normal-form theorem for context-free phrase structure grammars," *J. ACM*, vol. 12, Jan. 1965.

[15] M. M. Geller, M. A. Harrison, and I. M. Havel, "Normal forms of deterministic grammars," *Discrete Mathematics*, vol. 16, 1976.

[16] M. A. Harrison and I. M. Havel, "Real-time strict deterministic languages," *SIAM Journal on Computing*, vol. 1, 1972.

[17] J. Levine and L. John, *Flex & Bison*, 1st ed. O'Reilly Media, Inc., 2009.

[18] D. Beazley, "PLY (python lex-yacc)," http://www.dabeaz.com/ply/index.html.

[19] INRIA, "Lexer and parser generators (ocamllex, ocamlyacc)," http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html.

[20] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron, "MNCaRT: An open-source, multi-architecture automata-processing research and execution ecosystem," *IEEE Computer Architecture Letters*, vol. 17, Jan 2018.

[21] W. J. Bowhill, B. A. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, O. Mendoza, C. Morganti, C. Houghton, D. Krueger, O. Franza, J. Desai, J. Crop, B. Brock, D. Bradley, C. Bostak, S. Bhimji, and M. Becker, "The Xeon® processor E5-2600 v3: a 22 nm 18-core product family," *J. Solid-State Circuits*, vol. 51, 2016.

[22] W. Chen, S.-L. Chen, S. Chiu, R. Ganesan, V. Lukka, W. W. Mar, and S. Rusu, "A 22nm 2.5 mb slice on-die l3 cache for the next generation Xeon® processor," in *Symposium on VLSI Technology*, 2013.

[23] M. Huang, M. Mehalel, R. Arvapalli, and S. He, "An energy efficient 32-nm 20-mb shared on-die L3 cache for Intel® Xeon® processor E5 family," *J. Solid-State Circuits*, vol. 48, 2013.

[24] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, 2014.

[25] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *International Symposium on Microarchitecture*, 2017.

[26] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Scientific Computing*, vol. 20, 1998.

[27] Intel, "Cache Allocation Technology," 2017. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology

[28] "Performance Application Programming Interface." http://icl.cs.utk.edu/papi/.

[29] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *International Symposium on Low-Power Electronics and Design*, 2010.

[30] "nvprof profiling tool," http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview.

[31] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. D. Cameron, "Parabix: Boosting the efficiency of text processing on commodity processors," in *International Symposium on High Performance Computer Architecture*, 2012.

[32] "Ximpleware XML dataset," http://www.ximpleware.com/xmls.zip.

[33] "XML Data Repository," http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html.

[34] J. Wadden and K. Skadron, "VASim: An open virtual automata simulator for automata processing application and architecture research," University of Virginia, Tech. Rep. CS2016-03, 2016.

[35] M. J. Zaki, "Efficiently mining frequent trees in a forest," in *knowledge Discovery and Data Mining*, 2002.

[36] R. Iváncsy and I. Vajk, "Automata theory approach for solving frequent pattern discovery problems," *Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 1, 2007.

[37] J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, S. Description, and L. Technologies, "Graphviz— open source graph drawing tools," in *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[38] M. Becchi, "Regular expression processor," http://regex.wustl.edu, 2011, accessed 2017-04-06.

[39] J. van Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, "Designing a programmable wire-speed regular-expression matching accelerator," in *International Symposium on Microarchitecture*, 2012.

[40] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch, "HAWK: hardware support for unstructured log processing," in *International Conference on Data Engineering*, 2016.

[41] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "HARE: hardware accelerator for regular expressions," in *International Symposium on Microarchitecture*, 2016.

[42] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: the unified automata processor," in *International Symposium on Microarchitecture*, 2015.

[43] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "UDP: a programmable accelerator for extract-transform-load workloads and more," in *International Symposium on Microarchitecture*. ACM, 2017.

[44] A. Subramaniyan and R. Das, "Parallel automata processor," in *International Symposium on Computer Architecture*, New York, NY, USA, 2017.

[45] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. R. Stan, "REAPR: Reconfigurable engine for automata processing," in *International Conference on Field-Programmable Logic and Applications*, 2017.

[46] V. B. Schneider and M. D. Mickunas, "Optimal compression of parsing tables in a parsergenerating system," Purdue University, Tech. Rep. 75-150, 1975.

[47] P. Dencker, K. Dürre, and J. Heuft, "Optimization of parser tables for portable compilers," *ACM Trans. Program. Lang. Syst.*, vol. 6, Oct. 1984.

[48] E. Klein and M. Martin, "The parser generating system PGS," *Software: Practice and Experience*, vol. 19, 1989.

[49] S. McPeak and G. C. Necula, "Elkhound: A fast, practical GLR parser generator," in *Compiler Construction*, 2004.

[50] T. Parr and K. Fisher, "LL(*): The foundation of the ANTLR parser generator," in *Programming Language Design and Implementation*, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993548

[51] J. Van Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson, "Xml accelerator engine," in *The First International Workshop on High Performance XML Processing*, 2004.

[52] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel, "Hardware acceleration in the IBM PowerEN processor: Architecture and performance," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012.

[53] P. Ogden, D. Thomas, and P. Pietzuch, "Scalable XML query processing using parallel pushdown transducers," *Proceedings of the VLDB Endowment*, vol. 6, 2013.