Data-Parallel Finite-State Machines

Todd Mytkowicz
Microsoft Research
toddm@microsoft.com

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Wolfram Schulte
Microsoft
schulte@microsoft.com

Abstract

A finite-state machine (FSM) is an important abstraction for solving several problems, including regular-expression matching, tokenizing text, and Huffman decoding. FSM computations typically involve data-dependent iterations with unpredictable memory-access patterns making them difficult to parallelize. This paper describes a parallel algorithm for FSMs that breaks dependences across iterations by efficiently enumerating transitions from all possible states on each input symbol. This allows the algorithm to utilize various sources of data parallelism available on modern hardware, including vector instructions and multiple processors/cores. For instance, on benchmarks from three FSM applications: regular expressions, Huffman decoding, and HTML tokenization, the parallel algorithm achieves up to a 3× speedup over optimized sequential baselines on a single core, and linear speedups up to 21× on 8 cores.

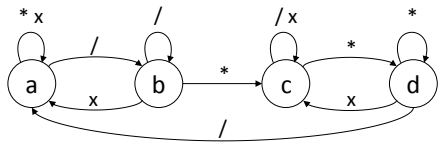
Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; F.1.2 [*Modes of Computation*]: Parallelism and concurrency

Keywords Parallelism, Finite State Machines, Regular Expression Matching, Huffman Decoding

1. Introduction

Hardware is parallel. While desktops, tablets, and even phones have vector instructions, multiple cores, and GPUs, many important algorithms are unable to exploit all of this parallelism. This paper focuses on finite-state machines, a class of applications that has previously been hard to parallelize.

A finite-state machine (FSM) is a fundamental model of computation [10] that is at the core of many practical applications ranging from regular-expression matching, tokenizing text, dictionary-based decoding, network intrusion detection, and feature extraction from web pages. Figure 1(a) shows an example FSM that identifies C-style comments. The table in Figure 1(b) determines how the FSM states transition on each input symbol, and Figure 1(c) is a straightforward implementation of this FSM that iteratively accesses the transition table to obtain the state after each input symbol.



(a)

T:	/	*	x
a	b	a	a
b	b	c	a
c	c	d	c
d	a	d	c

(b)

```
1  state = a;
2  foreach (input in)
3    state = T[in][state];
```

(c)

Figure 1. An FSM containing four states that accepts C-style comments in source code delineated by `/*` and `*/`. The input `x` represents all characters other than `/` and `*`.

FSMs are difficult to parallelize for two reasons. First, there is a tight dependence between successive loop-iterations making it nontrivial to distribute loops across multiple processors. Second, FSMs perform little computation in each iteration with memory-access patterns that are input-dependent and unpredictable.[1] This makes it difficult for FSM implementations to use parallelism *within* a processor, namely instruction-level parallelism, vector (SIMD) capabilities and memory-level parallelism.

This paper presents an efficient parallel algorithm for FSM computations that is able to utilize various kinds of data parallelism available on modern hardware, both paral-

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541988

[1] Alternately, FSM implementations can encode transition tables using a large `switch` statement — trading unpredictable data accesses for unpredictable control flow.

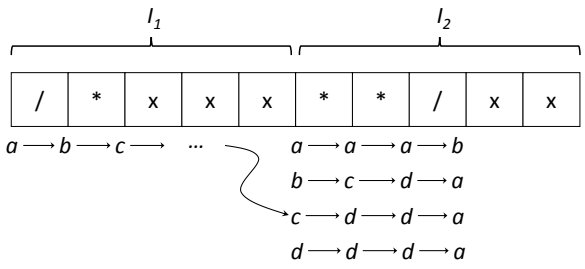


Figure 2. Enumerative computation for parallelizing FSMs.

lelism across multiple processors and parallelism within a single processor. The algorithm is up to 3× faster than optimized sequential implementations on a single processor and achieves subsequent multiplicative speedups with multiple processors.

Figure 2 shows the key idea behind our parallel algorithm. Consider a run of the FSM in Figure 1(a) on the input shown in Figure 2. We can parallelize this computation by splitting the input into two chunks I_1 and I_2 and performing the two subcomputations in parallel. Obviously, this is not possible unless we know the start state for the second subcomputation, which is the final state of the first subcomputation.

One way to break this dependency is to fork a version of the second subcomputation starting from *every* state of the FSM as shown in Figure 2. We call this an *enumerative* computation as it enumerates all possible start states. Once the first subcomputation has finished, we pick the version of the enumerative computation that started from the correct state, which in this case is state c. It is possible to generalize this approach by splitting the input many times where all but the first computation is done enumeratively in parallel (Section 3).

The obvious disadvantage of this approach is the overhead of the enumerative computation. For an FSM with n states, an enumerative computation performs n times more work than the sequential version. In other words, a naïve implementation requires a linear number of processors to achieve constant speedups and does not scale for all but small FSMs.

The main contributions of this paper are optimizations that make this enumerative computation efficient. The first optimization relies on the observation that different enumerations perform redundant work when multiple states *converge* to the same state on some input symbol (Section 5.2). For instance, after reading a *, both states c and d transition to state d in Figure 1(a). This causes the three of the four enumerations to converge after reading the first two input symbols in Figure 2. We call the states of such non-redundant computations *active* states. By dynamically utilizing convergence, the overhead of the enumerative computation is proportional to the number of active states and not to the total number of state in the FSM.

Using realistic examples, this paper demonstrates that most FSMs, even those with many states, often converge to 16 or less active states for *any* input. This makes the convergence optimization worthwhile. Our experiments also show that while convergence is common, convergence to a single state is rare. As a result, the enumerative computation is still more expensive than the sequential computation.

A major component of this overhead is the transition-table lookup required for each active state. To optimize this cost, we use a *range-coalesced* representation of the transition table (Section 5.3). In this representation, states get different identifiers based on the most recently seen input. By appropriately assigning these identifiers, we ensure that accesses made by active states are coalesced into a smaller range of memory, reducing memory pressure.

Finally, we observe that different enumerations are *independent* and thus can be performed simultaneously using different lanes of a single-instruction multiple-data (SIMD) processor. One challenge in using SIMD instructions is the need for a *gather* operation that accesses the transition table at different offsets from each lane. Unfortunately, current desktop processors do not *yet* support generic gather operations. We use an existing SIMD primitive, `shuffle`, to implement this operation.

Our benchmarks consist of thousands of FSMs from three case studies: regular-expression matching, Huffman decoding, and HTML tokenization (Section 6). For more than 80% of these FSMs, our implementation performs one or two `shuffle` operations per input symbol. The associativity of these operations [26] allows the hardware to process multiple input symbols simultaneously, increasing instruction-level parallelism. As a result, our parallel implementation is up to 3× faster than optimized sequential implementations on a single processor core. We further demonstrate multiplicative speedups on multiple processor cores, up to 21× on 8 cores.

2. Background

This section provides a brief primer on FSMs and introduces terminology used in the rest of the paper. It then describes a general approach for parallelizing FSM computations.

2.1 Primer on Finite-State Machines

We assume that the reader is familiar with classic automata theory, as described in Hopcroft and Ullmann [10]. A deterministic *finite-state machine* D is a tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a fixed alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and δ is the transition function with type $Q \times \Sigma \to Q$. This paper focuses on deterministic FSMs. A nondeterministic FSM can be converted to a deterministic FSM using standard techniques [10], albeit with a worst-case exponential blowup in the number of states. Applying the techniques discussed in

this paper to parallelize nondeterministic FSMs is left for future work.

To determine if a string $s = s_1, s_2, \ldots, s_m$ is accepted by an FSM D, the FSM starts in state q_0 and iteratively applies the transition function to determine the state q_i on reading s_i using the recurrence $q_i = \delta(q_{i-1}, s_i)$. If the final state q_m is in F, then the string s is *accepted* by the FSM D. When discussing our algorithm, we will also refer to a *transition function* T_a that maps states to states such that $T_a(q) = r$ iff $r = \delta(q, a)$. In implementations, the transition function is implemented as an array or arrays, in which case, we will refer to T_a simply as `T[a]`.

Applications sometimes perform actions on each state transition (e.g. tokenization or decoding) apart from accepting or rejecting the input string. This paper uses the Mealy formalism [20] and assumes the existence of a ϕ function that is invoked with the current input symbol s_i and the state q_i reached after reading that symbol. In the context of parallelizing FSMs, we will make a simplistic assumption that the ϕ function can be invoked out of order. Modifying sequential clients of FSMs to satisfy this assumption is not discussed in this paper.

2.2 Identifying Associative Operations in FSM

The relationship between FSM computations and semirings is well known [2]. A semiring is an algebraic structure with a generalized additive and multiplicative operations defined on a domain. The properties of these operations ensure that the generalized matrix multiplication in semirings is associative. In theory, this allows one to parallelize problems that can be cast as matrix multiplications in semirings.

Consider the semiring on the Boolean domain with disjunction as the additive operation and conjunction as the multiplicative operation. For an FSM with n states, let M_s be the $n \times n$ Boolean matrix, such that $M_s[i, j] = true$ iff state i transitions to state j on symbol s. For a given input string $s = s_1, s_2, ...s_m$, the matrix product

$$M = M_{s_m} \cdot \ldots \cdot M_{s_2} \cdot M_{s_1}$$

has the property that $M[i, j]$ is true iff the FSM reaches state q_j on reading the input s starting from state q_i. Thus, the FSM accepts the input iff $M[0, j]$ is true for some state $q_j \in F$.

Ladner and Fischer [15] exploit this formulation to parallelize FSM computation using parallel prefix-sums. Their algorithm generalizes to nondeterministic FSMs and executes in $O(log(m) \times n^3)$ time with m processors, using the standard cubic algorithm for matrix multiplication.

Hillis and Steele [8] describe an improved parallel prefix computation for evaluating deterministic FSMs that reduces the overhead to $O(log(m) \times n)$. The basic idea, which forms the basis for the enumerative computation described in Section 1, is to succinctly encode FSM computations as *composition* of transition functions. If T_{s_i} represents the transition function on input s_i and $\otimes$ represents function composition, then the function

$$T = T_{s_m} \otimes \ldots \otimes T_{s_2} \otimes T_{s_1}$$

determines the final state of the FSM starting from some state on input $s_1, s_2, \ldots, s_m$. Parallelization follows from the fact that function composition is associative.

This paper builds on these theoretical insights. The key contribution of this paper is in optimizing away the dependence on n, the number of states in the FSM, using properties of FSMs seen in practice, and on demonstrating a scalable implementation that exploits both fine-grained and coarse-grained parallelism available in modern hardware.

3. Parallel Algorithm

This section describes the data-parallel FSM algorithm but first introduces a primitive used to describe the algorithm.

3.1 Gather Primitive

Let S and T be arrays of length m and n respectively. A gather operation, represented by $S \otimes_{m,n} T$, is an array of length m such that

$$(S \otimes_{m,n} T)[i] = T[S[i]]$$

In essence, the array S contains indices (or addresses) that are used to lookup the array T. Obviously, the lookup is well-formed only when the indices are within the bounds of T. For convenience, we will assume that the index modulo n is used for such out-of-bound indices. Unless when necessary, we will drop the subscripts and simply refer to the gather of S and T as $S \otimes T$. Gather is associative.

$$((S \otimes T) \otimes U)[i] = U[(S \otimes T)[i]] = U[T[S[i]]]$$

$$(S \otimes (T \otimes U))[i] = (T \otimes U)[S[i]] = U[T[S[i]]]$$

When S is a set of FSM states and T is the transition function for some input symbol, the elements of $S \otimes T$ are the respective successor states for the states in S. In other words, gather implements the functional composition of the transition functions.

There is a trivial sequential implementation of $\otimes_{m,n}$ that performs m memory lookups, assuming the contents of S are already available in registers. More efficient implementations are possible with appropriate hardware support. For instance, Intel SSE3 architectures provide a `shuffle` instruction that performs $\otimes_{16,16}$ for byte arrays. Section 4 describes a way to implement a general $\otimes_{m,n}$ using $m/16 * n/16$ invocations of $\otimes_{16,16}$.

3.2 Base Enumerative Algorithm

Figure 3 provides the base enumerative algorithm for a given start state `st` and an input sequence `in`. When compared to the sequential algorithm in Figure 1(c), the enumerative

```
1  Base(State st, Input in){
2    States S = Id;
3    for (i=0; i<in.len; i++) {
4      a = in[i];
5      S = S ⊗ T[a];
6      φ(a, S[st]);            }}
```

Figure 3. Base Enumerative Algorithm.

```
1  Base_ILP(State st, Input in){
2    States S = Id;
3    for (i=0; i<in.len/3; i+=3) {
4      (a,b,c) = (in[i], in[i+1], in[i+2]);
5      Sa = S ⊗ T[a];  Tbc = T[b] ⊗ T[c];
6      Sb = Sa ⊗ T[b]; S = Sa ⊗ Tbc;
7      φ(a, Sa[st]); φ(b, Sb[st]); φ(c, S[st]); }}
```

Figure 4. Unrolling the loop in Figure 3 exposes ILP. Instructions in the same line can be executed in parallel.

algorithm maintains an array of states, `S`, at each step rather than a single state. At each step, element i of `S` is the state reached at that step if the FSM had started from state i. `S` is initialized to the identity array `Id` whose element i is i.

On an input symbol `a`, the algorithm obtains the transition function `T[a]`. The gather operation provides `S` for the next iteration. In addition, the algorithm calls the output ϕ function on the actual FSM state `S[st]`. Of course, invoking the ϕ function at each step is unnecessary when the FSM is only determining an accept/reject decision on the input (say, when performing regular-expression matching). In such cases, we will assume that invoking the ϕ function after processing all of the input provides the accept or reject decision.

3.3 Using Fine-Grained Parallelism

When compared to the sequential algorithm in Figure 1(c), the base enumerative algorithm performs more work. However, the associativity of gather alleviates this overhead by exposing instruction-level parallelism (ILP). For instance, Figure 4 unrolls the loop three times (and assumes that the input length is divisible by three). The instructions on the same line do not depend on each other and thus can execute in parallel. Assuming an efficient implementation of gather, this additional ILP can make the base enumerative algorithm run faster than the sequential version, despite performing more work. Optimizations described in Section 5 enable this possibility.

3.4 Using Coarse-Grained Parallelism

Figure 5 shows the parallelization of the base enumerative algorithm on multiple cores/machines, using an implementation of parallel-prefix sum [1, 15]. The parallel algorithm distributes the input equally among available processors and each processor processes its chunk in three phases. In the first phase, each processor runs the base enumerative algorithm on its chunk of the input in parallel. The end goal of the first phase is to compute for each processor `S[p]`, an array that determines the final state of the FSM when processing the input chunk from every starting state.

```
1  Base_Multicore(State st, Input in){
2    chunk = in.len/NumProc;
3
4    // Run chunks in parallel
5    States S[NumProc];
6    parallel.for (proc p in [0...NumProc-1]){
7      my_in = in[p*chunk .. (p+1)*chunk-1];
8      S[p] = T[my_in[0]];
9      for (i = 1; i<chunk; i++){
10       S[p] = S[p] ⊗ T[my_in[i]];          }}
11
12   // Compute start states for each chunk
13   State st[NumProc];
14   st[0] = st;
15   for (p in [1..NumProc-1]){
16     st[p] = S[p-1][st[p-1]];}
17
18   // Compute output in parallel
19   parallel.for (proc p in [0...NumProc-1]){
20     my_in = in[p*chunk .. (p+1)*chunk-1];
21     Base(st[p], my_in);                   }}
```

Figure 5. Using multiple cores to parallelize the base enumerative algorithm.

The sequential second phase (which can be parallelized, if necessary), computes the start states for each processor using these arrays. With the correct start states known, the third phase simply invokes the base enumerative algorithm for each input chunk in parallel. Note, the ϕ function is only called in the third phase, making the first two phases extremely fast. The implementations of the first and the third phase can additionally benefit from fine-grained parallelism as shown in Figure 4.

There are a variety of ways to implement a parallel-prefix sum on modern parallel hardware. For example, prior work has implemented these primitives on GPUs [4, 7, 29], on a cluster [6, 36], or using threads [31]. In contrast, the algorithm in Figure 5 is designed to minimize communication when the number processors is much smaller than the amount of parallelism available (proportional to the size of the input), which is the case for our applications.

4. Implementation of Gather

The performance of the parallel algorithms in Section 3 depend on an efficient gather. This sections provides such an implementation.

4.1 Non-SIMD Gather

The straightforward implementation of $S \otimes_{m,n} T$ is with m memory lookups. Figure 6 illustrates the performance of this implementation on a *gather microkernel*. The microkernel emulates the inner-loop of the base algorithm in Figure 3 on random inputs and random transition funtions. In particular,

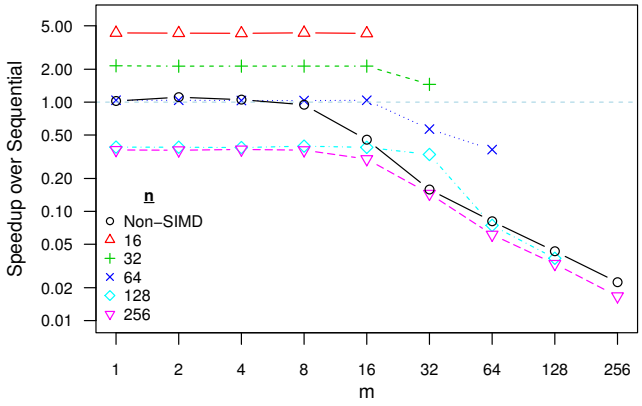


Figure 6. Performance of $\otimes_{m,n}$ on a machine with SIMD width $W = 16$.

the microkernel allocates 1024 arrays of size $n = 1024$, each initialized with random values between 0 and $n - 1$. Then for different values of m, it enters a tight loop that computes $S = S \otimes_{m,n} T_r$ for a table T_r randomly picked from the allocated arrays. Figure 6 compares the performance of the microkernel with a sequential baseline of Figure 1(c).

The "Non-SIMD" line in Figure 6 shows that the compiler and the hardware are able to hide the cost of multiple lookups (i.e., performance is at 1.0) for m up to 8. This suggests that for small FSMs with less than or equal to 8 states, the enumerative algorithm incurs no additional overhead. However, the performance degrades for larger values of m.

4.2 SIMD Gather

Modern microprocessors are equipped with Single Instruction Multiply Data (SIMD) registers that apply an operation on W data items in one instruction. In particular, `x86` has support for *byte* level `shuffle` of an array of $W = 16$ characters that essentially implements $\otimes_{16,16}$ for two SIMD registers. Future architectures are expected to increase the width up to 1024 bytes along with a support for general purpose gather [12].

Now, we will describe how to implement a general $\otimes_{m,n}$ using multiple invocations of $\otimes_{16,16}$. In this paper, we will only need instances where $m \leq n$. The implementation is best understood through an example. Suppose the SIMD width is $W = 4$ and we desire to implement $S \otimes_{m,n} T$ when $n = m = 8$ for the following arrays

$$S = [3, 5, 0, 1, 5, 4, 6, 2], T = [A, B, C, D, E, F, G, H]$$

The desired answer is $[D, F, A, B, F, E, G, C]$. Let us store S and T into two SIMD registers S_{03} and S_{47}, and T_{03} and T_{47} respectively. Current implementations of `shuffle` use the index modulo W when an index exceeds W. The first four values of $S \otimes T$ can be obtained by performing

$$S_{03} \otimes T_{03} = [D, B, A, B], \quad S_{03} \otimes T_{47} = [H, F, E, F]$$

and blending the results based on whether the index exceeds W or not. In fact, `x86` supports a `blend` instruction for this purpose. Thus,

$$(S \otimes T)_{03} = \texttt{blend}(S_{03} \otimes T_{03}, S_{03} \otimes T_{47}, S_{03} < 4)$$

$$(S \otimes T)_{47} = \texttt{blend}(S_{47} \otimes T_{03}, S_{47} \otimes T_{47}, S_{47} < 4)$$

Generalizing this example, $\otimes_{m,n}$ can be implemented using $(m * n)/W$ invocations of $\otimes_{W,W}$.

SIMD gather provides two benefits. First, it utilizes the entire memory bandwidth between the processor and its L1 data cache. A single SIMD load fetches 128 bits while a non-SIMD load can only fetch 64 bits. Second, SIMD gather exploits the fact that FSMs are memory bound computations with underutilized functional units. The `shuffle` and `blend` instructions use the functional units that would have otherwise been idle.

Figure 6 illustrates the performance of SIMD gather for various values of m and n for the gather microkernel running on a machine with $W = 16$. When $n = W$, the SIMD gather is $4.4\times$ faster than the sequential baseline. The optimizations discussed in Section 5 enable many of the FSMs from our benchmarks to exploit this performance. For n up to 64, the SIMD gather still outperforms the non-SIMD version. For larger values of m, the overhead of performing $(m * n)/16$ `shuffle` operations becomes a bottleneck. However, this overhead can be hidden by parallelizing the enumerative computation on multiple processor cores.

4.3 Implementation Details

The implementation uses C++ template specialization to instantiate gather for appropriate values of m and n. We hand-coded specialized implementations for all pairs of m and n that are multiples of 16 in the range $[16, 256]$. Peculiarities of `x86` make the implementation nontrivial. For instance, the `shuffle` instruction in SSE 4.2 (`_mm_shuffle_epi8`) treats indices as signed bytes and treats negative indices as index 0. Supporting FSMs with greater than 127 states requires additional bit manipulation to handle the sign bit correctly. Similarly, byte level `shuffle` is not implemented on AVX and so our current implementation cannot utilize 256 bit wide registers despite running on AVX hardware.

5. Optimizations

As discussed in the previous section, the cost of $\otimes_{m,n}$ depends both on m and n. This section describes two optimizations for the parallel FSM algorithm, namely *convergence* that reduces m and *range coalescing* that reduces n. We start with a primitive necessary to explain these optimizations.

5.1 Factor Primitive

Given an array S, $Factor(S)$ is a pair (L, U) such that $S = L \otimes U$ and U contains only the unique elements of S. (Here L stands for "lookup" and U stands for "unique".)

```
1  Convergence( State st, Input in ){
2    States S = Id;
3    States Acc = Id;
4
5    for (i=0; i<in.len; i++){
6      S = S ⊗ T[in[i]];
7      if (conv_check()){
8        (L, U) = Factor(S);
9        S = U;
10       Acc = Acc ⊗ L; }
11     S_base = Acc ⊗ S;
12     φ(i, S_base[st]);        }}
```

Figure 7. Convergence Algorithm

An example is given below:

$$[s,t,u,t,t,u,s] = [0,1,2,1,1,2,0] \otimes [s,t,u]$$

Note, $|U| \leq |S|$. When performing $S \otimes T$, identical elements in S result in redundant lookups of T. We can eliminate this redundancy by factoring S into (L, U) and only performing $U \otimes T$. This is particularly useful if the result of the gather is used to perform subsequent gathers as in the base enumerative algorithm in Figure 3.

Unfortunately, there is no direct support for the factor operation in architectures today. Therefore, we implement the straightforward sequential algorithm for this operation that takes linear time. Accordingly, it is important to use the factor operation sparingly.

5.2 Convergence Optimization

The key observation behind the convergence optimization is that transition functions in most FSMs are not permutations—many states transition to the same state on a symbol. In Figure 1(a) for instance, both states c and d transition to d on reading a $*$. Accordingly, many elements of the array S in the base algorithm in Figure 3 are likely to be the same. The convergence optimization eliminates this redundancy by factoring.

Convergence Algorithm. Figure 7 shows the algorithm with the convergence optimization. Periodically at line 8, the algorithm factors S into L and U, uses U for subsequent iterations. It also accumulates L into Acc (line 10). The correctness of the algorithm follows from the loop invariant that S_{base} at line 11 is the same as S at line 5 in Figure 3. This invariant follows from the fact that S = L ⊗ U and the associativity of ⊗.

We define *active* states as the states in S at the end of a particular iteration. The number of active states is non-increasing during the execution of this algorithm. As the number of active states become smaller, gathers in subsequent loop iterations are faster. In essence, convergence allows us to move left toward smaller values of m (number of active states) along a performance curve in Figure 6 determined by n (the number of states in the FSM).

The algorithm in Figure 7 can be implemented to use both the fine-grained parallelism within a core and the coarse-grained parallelism across cores by appropriately modifying the respective algorithms in Figure 4 and Figure 5. Also, it is not necessary to compute all elements S_{base} in an implementation as only the entry corresponding to the start state st is required for the output ϕ function. We chose to present the algorithm this way to make its correctness evident.

Frequency of Convergence Checks The conv_check() predicate at line 7 determines how often the algorithm checks for convergence. Since factoring is not natively supported in the hardware, it is important to only check for convergence when we expect the number of active states to dramatically decrease since the last convergence check. For our SIMD-based implementation, this decrease has to be larger than the number of states that fit in a SIMD register (which in many cases is 16) to reduce the cost of subsequent gathers

We use two heuristics in our implementation. First, we statically analyze the convergence characteristics (Section 5.2 below) of the given FSM to determine the frequency of convergence checks. Second, for every input symbol, we pre-calculate the size of the range of the transition function for that symbol. For instance, if the FSM can only go to one of three states on an input symbol, then we know that the number of active states after reading that symbol is less than or equal to three.

Convergence in Practice Convergence arises when multiple states transition to the same state on an input symbol—that is, the transition function is many-to-one. There are n^n possible functions from n states to n states, but there are only $n!$ permutations. From Stirling's approximation for the factorial, it follows that there are exponentially more (e^n) many-to-one functions than permutations. Thus, randomly chosen FSMs are not likely to have transition functions that are permutations, for reasonably large n.

FSMs, in practice, have more structure than randomly generated FSMs and are likely to converge faster. For instance, the transition function for a symbol might only be defined for some of the states with the rest transitioning to an error state or a reset state. To evaluate the rate convergence in practice, we performed two experiments on FSMs generated from 2711 regular expressions in the Snort suite [27]. The number of states in these FSMs range from 1 to 4020, with a median of 25.

Adversarial Inputs The first experiment studies convergence under *adversarial* worst-case inputs. We emulated the enumerative computation for *all* inputs of length k, and picked the input that resulted in the most number of active states. For these FSMs, the inputs are arbitrary character strings, and thus there are 256^k possible inputs of length k. To perform the emulation efficiently, we systematically explore the state space of all configurations reached during an

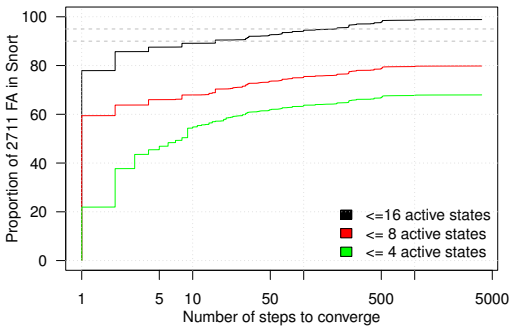


Figure 8. Almost all FSMs converge to 16 active states or less for adversarial inputs.

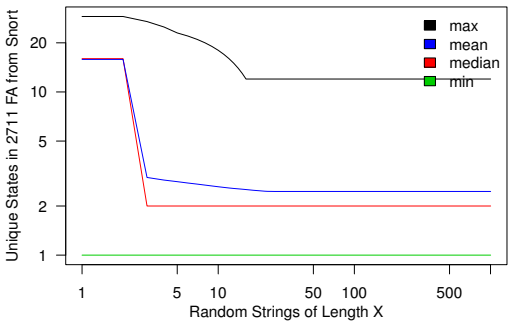


Figure 9. FSMs converge on randomly chosen inputs. All FSMs converge to less than 16 active states after 20 steps.

enumerative computation, where a configuration is the set of active states. There are 2^n possible configurations. The very fact that we are able to explore such a large state space is a testament to convergence — many of the possible configurations are not reachable from the initial configuration that contains the set of all states.

Figure 8 shows the worst-case convergence for inputs of length k shown on the logarithmic x-axis. The y-axis shows the proportion of FSMs for which the number of active states is respectively less than or equal to 16, 8, and 4 for a given k. The figure shows that around 90% of FSMs converge to 16 active states or less after a mere 10 steps, and 95% converge after 200 steps. For inputs longer than this, the convergence algorithm is guaranteed to use the fastest SIMD-gather possible for a given n, the number of FSM states, as shown in Figure 6.

Figure 8 also shows that only 80% ever converge to 8 active states, and less than 70% converge to 4 active states. This has two important implications. First, an adversary can always make the enumerative computation asymptotically more expensive than the sequential computation. Second, speculative approaches [13, 14, 24] that rely on predicting the likely state of an FSM after reading some input will fail on carefully designed inputs.

Random Inputs Our second experiment studies convergence on non-adversarial inputs. Unfortunately, the Snort regular expressions are designed for matching against (adversarial) network traffic, and obtaining expected input distribution for these FSMs is not possible. We approximated this by using inputs at *random offsets* in a large dump of Wikipedia pages. Figure 9 shows the average number of active states (y-axis) after running an FSM on 10 randomly chosen inputs of a particular length (x-axis). The various lines show respectively, the max, min, median, and the mean number of active states for the 2711 Snort regular expressions. As expected, we see better convergence than for adversarial inputs. All of the FSMs converged to 16 active states or less. But, more than half of the FSMs did not converge to one active state.

5.3 Range Coalescing

Like convergence, range coalescing relies on the observation that transition functions of FSMs are likely to be many-to-one. In particular, the range of a transition function is likely to be smaller than the number of states in the FSM. Range coalescing uses this observation to reduce the range of memory addresses accessed by an enumerative computation for each input symbol.

Running Example Consider the FSM in Figure 10 with five states, p through t, and two input symbols a and b. The range of a is $\{p, q, t\}$ and the FSM is guaranteed to be in one of these three states after reading a. This means that in the next step, the FSM will only access three of the five rows of the transition table T. The goal of range coalescing is to ensure that these three rows are contiguous in memory. Similarly, the range of b is $\{p, r, s\}$ and we desire to make these three rows contiguous after reading a b.

To achieve this, range coalescing maintains different *names* for each state, one for each input symbol whose range the state belongs to. In Figure 10, p_a and p_b are the two names for p corresponding to a and b respectively. All other states, get one name as they belong to the range of only one symbol. We will refer to state names corresponding to a symbol a simply as *names of* a.

Range coalescing generates a transition table for each input symbol indexed by the names of that symbol. For instance, the table T_a is indexed by p_a, q_a, t_a. The FSM will use T_a (instead of T) for the lookup after reading an a and T_b after reading a b. To ensure correct lookups, range coalescing maintains the invariant that the current name always corresponds to the last input symbol read by the FSM. This requires that the transition tables use the names of a particular symbol when representing the destination of a transition on that symbol. For instance, the columns of the transitions tables T_a and T_b for a use the names of a.

Generation of Transition Tables The generation of range-coalesced transition tables can be elegantly described using the factor primitive. Let $(L_a, U_a) = Factor(T[a])$ be the factorization of $T[a] = [q, t, t, q, p]$, the transition function

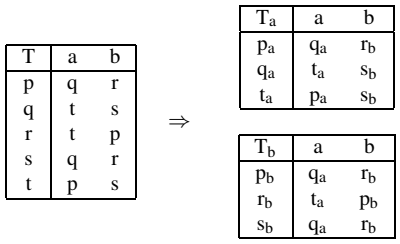


Figure 10. Range-Coalescing Example.

```
1  RangeCoalescing( State st, Input in ){
2    a = in[0];
3    S = La;
4    T = Ta;
5    for (i=1; i<in.len; i++){
6      b = in[i];
7      S = S ⊗ T[b];
8      T = Tb;
9      Sbase = S ⊗ Ub;
10     φ (i, Sbase[st]);        }}
```

Figure 11. Range-Coalescing Algorithm.

of a in Figure 10. We have

$$T[a] = (L_a = [1, 2, 2, 1, 0]) \otimes (U_a = [p, q, t])$$

We will use the indices of states in U_a as its names of a: $p_a = 0, q_a = 1, t_a = 2$. Then, L_a maps how states transition on a when using the names of a for the destination states. U_a provides a map from names of a to states. Similarly, we will use $L_b = [1, 2, 0, 1, 2]$ and $U_b = [p, r, s]$ to describe corresponding maps for b.

The transition function $T_a[b]$ describes how names of a transition to names of b on reading b. This is given by

$$T_a[b] = U_a \otimes L_b$$

where U_a maps from names of a to states and L_b performs the lookup but uses the names of b for the destination states. Similarly, $T_a[a] = U_a \otimes L_a$. Such pairwise combination of two symbols generates all the entries in the range-coalesced transition tables, as shown in Figure 10.

Range-Coalescing Algorithm Figure 11 describes the range-coalescing algorithm. The algorithm precomputes the transition tables for every input symbol as described above. The algorithm initializes the set of states S with L_a for the first symbol a at line 3. S now represents the states, represented as names of a, reached after reading a for all initial states. Then, the algorithm sets T to T_a to be used for the lookup in the next iteration.

For each input symbol b, the algorithm performs a gather with the transition function $T[b]$ at line 7. If the previous symbol was a, then T points to T_a and $T_a[b]$ performs the transition lookup from names of a to names of b. The algorithm then switches T to T_b to be used in the next step.

The correctness of the algorithm follows from the invariant that S_{base} at line 9 is the the same as S at line 5 in Figure 3. This invariant holds because $T_a[b] = U_a \otimes L_b$ and gather is associative. This algorithm can be parallelized using both fine-grained and coarse-grained parallelism by appropriately modifying the algorithms in Figure 4 and Figure 5.

Performance Discussion The key performance gain comes from using smaller transition tables at each step. Range-coalescing allows the use of $\otimes_{m,n}$ operations where n is reduced from the number of states in the FSM to the range size of the last seen input symbol. In effect, range coalescing allows us to move to a higher performance curve in Figure 6. In contrast, the convergence optimization discussed in Section 5.2 allows us to move left along a performance curve (by reducing m). In practice, however, we avoid dynamically switching between code that uses $\otimes_{m,n}$ for different values of n. Thus, we set n to the maximum of the range size for all input symbols.

Another benefit of range-coalescing is that the algorithm only requires state names when processing input. Mapping back to states is only required when calling the output ϕ function. This can lead to efficient state encodings. For instance, if an FSM has more than 256 states but the maximum range size is less than 256, then the range-coalescing algorithm can encode state names with a byte and use byte-level SIMD gathers. In contrast, encoding states directly will otherwise require the use of much-slower word-level gathers.

On the downside, a state gets as many names as its incoming edges in the FSM. For an FSM with n states, e edges, and k symbols in its input language, the original transition table will have $n \times k$ entries (one for each state and input symbol), while the range-coalesced tables will together have $e \times k$ entries. In the worst case, the FSM is total and a transition is defined for every symbol and every state. In this case, the range tables can have as many as $n \times k^2$ entries. An implementation should ensure that this blow up does not impact the runtime performance, say when the working set of the range-coalesced tables no longer fit in the first-level cache.

Finally, in addition to the static overhead above, the algorithm requires an additional memory access at line 8 when compared to the base enumerative algorithm in Figure 3. As discussed in Section 6, this additional lookup in the tight-loop can deteriorate performance.

6. Case Studies

This section demonstrates the efficacy of our approach by instantiating three real-world implementations: Snort regular expressions, Huffman decoding, and HTML Tokenization.

Platform We conducted all experiments on an unloaded Intel 2.67GHz Xeon (X5650) workstation with 16 cores and 16GB RAM, running Windows 8. We use the Intel C++ compiler, version 12.1. For generality, we also repeated our

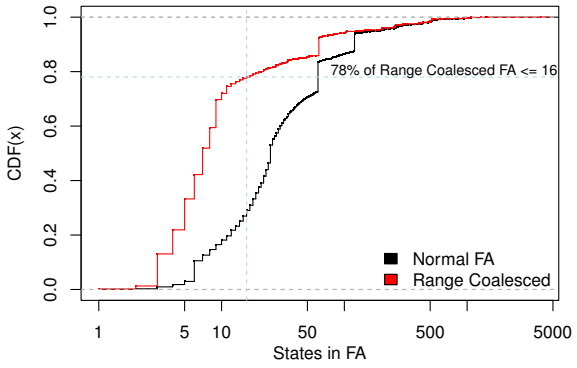


Figure 12. Distribution of the number of states and the maximum range sizes for 2711 Snort regular expressions.

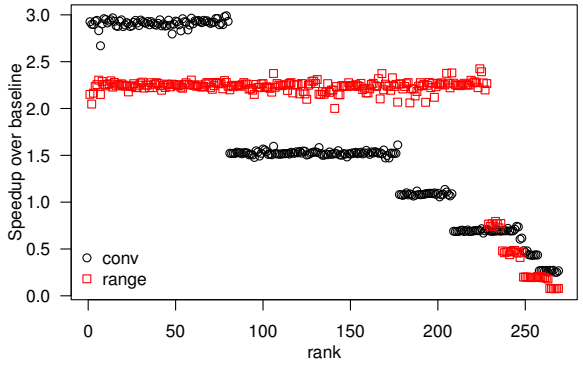


Figure 13. Single-core performance for Snort regular expressions.

experiments (not shown) on an older 8-core Intel 2GHz (L5420) workstation and found slightly slower (e.g. due to fewer cores and slower clock) but qualitatively similar results.

Measurement We measure *time* from within each process using `clock_t` to record ticks. The experiments do not measure setup costs—only the time taken for one invocation of each program. When we compare against a baseline, we modify that code to provide comparable measurements. Our benchmarks read all data into memory and then operate on that data. The speed of commodity disks are in the order of 100 MByte/Sec and many of our implementations are an order of magnitude faster.

Finally, to get statistically significant results, we run each experiment 30 times and report the mean. We do not report the 95% confidence interval of the mean when there is no significant variation from the mean. We found that our performance numbers are usually predictable as gather, our key primitive, is dominated by regular computations.

6.1 Snort Regular Expressions

Snort [27] is a network intrusion prevention and detection system. Snort comes with a large body of rules and signatures that are used to match against network packets to detect suspicious activity or attacks.

Benchmarks Regular expressions studied in this paper are obtained from version 2.9.4.0 snapshot of the Snort rules. We extracted the `pcre:` attribute from the rules and obtained 2711 regular expressions that our Perl-regular-expression compatible front-end could parse. The remaining 2828 regular expressions contain Snort-specific extensions that our front-end is not able to parse.

Figure 12 shows the distribution of the number of states in the FSMs for the parsed regular expressions. More than 95% of these regular expressions have less than 256 states, but the maximum number of states is 4020. The median state size is 25. Figure 12 also shows the opportunity for range coalescing and reports the maximum range size of the transition functions for these FSMs. 78% of these FSMs have a maximum range less than or equal to 16. The convergence properties of these FSMs are already discussed in Section 5.2.

Single-Core Performance Figure 13 shows the performance of enumerative computation on a single processor core using the SIMD-gather (Section 4.2). To reduce experiment times, we randomly sampled 269 regular expressions and the figure shows the speedup for both optimizations, over the sequential baseline of Figure 1(c) with optimal loop unrolling. For our convergence experiments (conv), we sort along the x axis by the number of states in the FSM while for our range-coalescing experiments (range), we sort by the maximum range size of the transition functions. The graph shows a series of plateaus. For convergence, each plateau denotes $16 * \lceil n/16 \rceil$, where n is the number of states in the FSM. For range coalescing each plateau is for $16 * \lceil m/16 \rceil$, where m is the maximum range size of the transition function across all input symbols.

For FSMs with a maximum of 16 states, our implementation with either of the two optimization performs one `shuffle` operation per input character. We observe up to a $3\times$ speedup with convergence (first plateau of figure). Due to the increase in the size of the range-coalesced transition tables and the additional memory lookup required to chose the transition table at line 8 in Figure 11, we only see a $2.2\times$ speedup. However, many of the FSMs with more than 16 states, have a maximum range size that is less than 16. Range coalescing benefits these FSMs. Since the maximum range of the transition function is statically known, it is possible for an FSM compiler to predetermine if the range coalescing optimization will perform well or not.

Multi-Core Performance The benefit of an enumerative computation with good single-core performance implies multiplicative multi-core performance. Figure 14 demonstrates the speedup obtained with multiple cores for both convergence and range coalescing, where the baseline is the respective single-core performance for each FSM. Both implementations achieve near linear strong-scaling up to 8

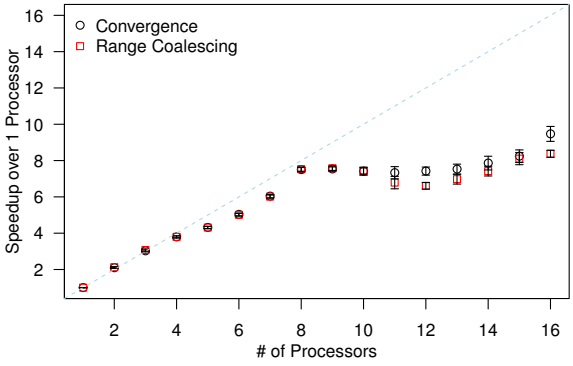


Figure 14. Multi-core performance for Snort regular expressions.

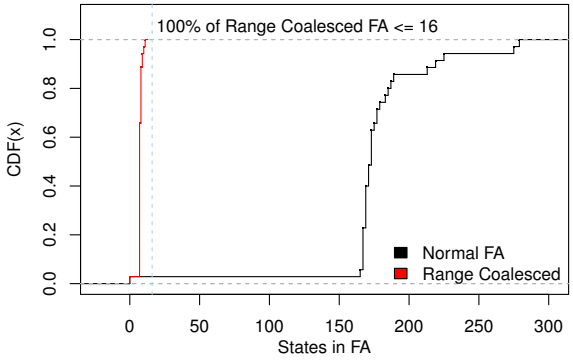


Figure 15. Distribution of states in Huffman trees, before and after range coalescing.

cores, after which the size of the input chunks per core is not sufficient to justify the cost invoking more threads. Also, the multi-core performance is mostly independent of the technique used to achieve single-core performance.

Larger FSMs Our goal in using Snort benchmark is to obtain a large class of useful regular expressions. While we generated FSMs from individual regular expressions, the typical use of these Snort rules is to match an incoming packet with *all* of the regular expressions at once. One way to obtain much larger FSM ([19], for instance), is to create a disjunction of all these expressions into a single regular expression. The resulting orders-of-magnitude blowup up in the number is well known [35]. It is unclear if our enumerative approach scales to such large FSM. On the other hand, creating a disjunction of regular expressions sequentializes a problem that is originally embarrassingly parallel — matching an input against many independent regular expressions.

6.2 Huffman Decoding

Another interesting application of FSMs is Huffman decoding. Huffman coding is a variable length encoding/decoding scheme, where each input character is encoded by a binary string whose length is inversely proportional to its occurrence probability. These encodings are represented as paths in a binary Huffman tree. Since the tree is finite, one can consider Huffman decoding as an FSM that traverses the Huffman tree on each input bit and emits the decoded character whenever it reaches a leaf. Huffman encoding is an embarrassingly parallel problem [11]. Instead, this paper focuses on the decoding phase.

Baseline We initially used `libhuffman` [16], an open source `C` library for Huffman encoding/decoding, which is used in PHP, as the baseline. We soon realized that `libhuffman` sequentially walks through each bit of an input and updates the position of a pointer in a Huffman binary tree. It spends most of its time performing bit operations and pointer chasing and can decode at most 5 MB/s.

We implemented a sequential baseline that is two orders of magnitude faster. Our baseline processes bytes rather than bits. The idea is to *unroll* the FSM such that each transition is a composition of 8 original transitions. Such an unrolling increases the number of edges in the FSM but not in the number of states. The resulting sequential baseline can decode at speeds greater than 300 MB/s.

Due to unrolling, each transition can potentially output more than one character. This complicates the output ϕ function. We encode the output as a sequence of (statically predetermined) strings, rather than a sequence of characters. This allows the ϕ function to be performed out of order. This requires an additional pass to process the output into appropriate form. We account for this pass in our evaluation.

Benchmarks We obtained 34 Huffman trees, each from the 34 most downloaded books (as of July 4th, 2013) from Project Gutenberg [25]. Each tree is slightly different based on the distribution of characters in these texts. Similar to the sequential baseline, we use an unrolled FSM whose inputs are 8-bit characters.

Figure 15 shows the distribution of number of states and the maximum range sizes of these FSM. The interesting observation is that while these Huffman trees can have as many as 300 states, the maximum range size is at most 16 states. Range coalescing provides two crucial advantages. First, we can encode state names with bytes allowing us to use Intel's byte-level `shuffle` instruction. Second, since the lookup tables are not more than 16, the implementation requires a single `shuffle` instruction for every input character. With such efficient encoding, there is no advantage to use the convergence optimization. The downside is that we create 256 range tables each of size 16*256. Accessing the 1MB of range tables reduces the performance due to L1-cache limits.

Single-Core Performance Figure 16 shows the single-core performance of our implementation using the SIMD-gather (Section 4.2). A bar on this plot provides the (i) sequential baseline and (ii) the range-coalesced FSM for the 34 Huffman trees when decoding a 1GB file. We observe a 2×

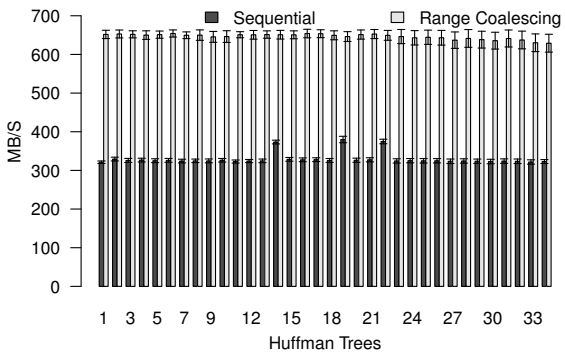


Figure 16. Single-core performance for Huffman decoding

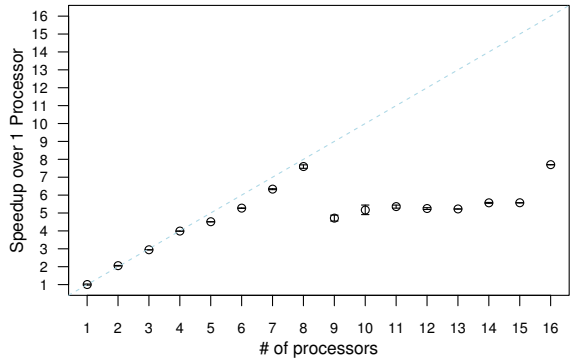


Figure 17. Multi-core performance for Huffman decoding

speedup over the sequential baseline for almost all Huffman trees (three give 1.75×).

Multi-Core Performance Figure 17 demonstrates the strong scaling of our approach with multiple cores. A point on this graph (x,y) gives the runtime in seconds (y) as a function of the number of processor cores used (x). We see near linear speedups until 8 processors after which the scaling stops. We expect our approach to scale for more processors with larger inputs.

6.3 HTML Tokenization

The third case study uses another important application of FSMs — lexing or tokenizing text. A tokenizer reads a sequence of bytes, character by character, and categorizes subsequences of those characters into *tokens*. In this section, we implement a data-parallel tokenizer for HTML that is a binary drop-in replacement for the one used in the crawler of the web-search engine `bing` .

Baseline `bing` uses an optimized hand-written tokenizer that uses switch statements to encode the FSM. For our experiments, we tokenize a 6MB dump of HTML from Wikipedia.

Benchmark We reverse engineered the `bing` implementation manually into an FSM with 27 states. The output ϕ function produces exactly the same output as `bing` allowing our

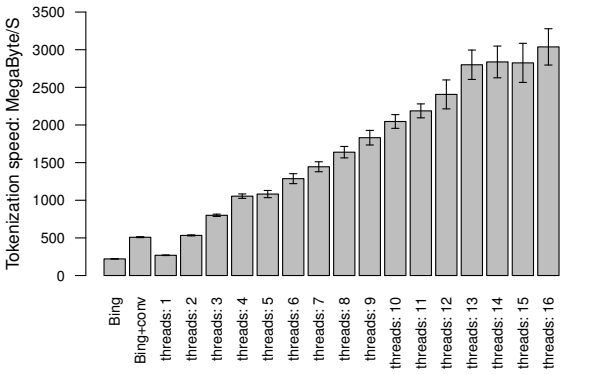


Figure 18. Multi-core performance of HTML tokenization.

implementation to be integrated with existing clients. Additionally, this enables us to verify that our manually generated FSM provides the same functionality as the hand-optimized version in `bing` . As the FSM has less than 32 total states, range-coalescing did not provide performance beyond what the convergence optimization provided. We only report numbers with the convergence optimization.

The manual effort of extracting an FSM prevents us from evaluating other tokenizers as part of this case study. To the best of our knowledge, modern compilers use similar hand-optimized lexers. Currently, we are not able to extract FSMs from automatic lexer generators, such as flex [23], that account for the complex acceptance and backtracking semantics implemented in these lexers.

HTML-Tokenization Performance Figure 18 compares the throughput of our baseline (`Bing`) to our implementation with a single-core implementation of convergence (`Bing+conv`) and our multi-core implementations (from `threads:1` to `threads:16`). The y-axis is the tokenization throughput in megabytes per second.

Our single-core implementation is 2.3× faster than the baseline. Our multi-core implementations require two passes over the input (Figure 5), once to determine the start state for each processor core and a second time to invoke the ϕ function. This explains the loss in performance between `Bing+conv` and `threads:1`. With 16-cores, our implementation tokenizes at 3025 MByte/Sec; faster than the line-rate of many networks (1000 MByte/Sec). This is 14× faster than the baseline.

7. Related Work

Due to its importance, optimizing FSM computations has received considerable attention in prior literature. This section describes closely related work.

Parallel FSMs As described in Section 2, parallel algorithms for FSMs have long been known [8, 15]. This paper builds on these ideas and demonstrates an efficient implementation that uses both fine-grained and coarse-grained parallelism. The most related prior works are optimized im-

plementations [9, 22] of the Hillis and Steele algorithm [8]. Holub et al. [9] rely on the input FSM being *k-local*, which requires that any two states converge to the same state on any input of length k. As our experiments show (Figure 8), most FSMs seen in practice are not k-local. Pan et al. [22] use a powerset construction, similar to the one used to determinize nondeterministic FSMs, to build an FSM that emulates the enumerative computation of the input FSM. This construction indirectly captures the convergence properties observed in this paper. However, the power set construction can result in exponential blow up in the number of states. Our algorithm can be considered as a dynamic variant of theirs which performs enumerative computation only on the given input, thereby avoiding this exponential blowup.

Speculative Parallelization One way to parallelize across dependencies is to speculate [13, 14, 19, 24]. When applied to FSMs, this amounts to guessing, rather than enumerating, the start state for all but the first chunk of the input (See Figure 2). The key observation behind these approaches is that two distinct states, the guessed state and the true start state, are likely to converge to the same state after reading some input. There are two major issues with a speculative approach. First, the efficacy of a speculative approach is difficult to predict. As seen in our convergence studies (Figure 8 and Figure 9), convergence is not always guaranteed. Further, if a processor does not converge on its chunk, then the next processor is forced to restart from a new state. The probability of such cascading misspeculations increases with the number of processors, thereby limiting scalability. Second, a speculative approach is still limited by the sequential implementation on a single core. In contrast, enumerative computation can use the fine-grained parallelism of a single processor.

Bit-Parallel FSMs Another closely related approach uses vector instructions to speed up single-core FSM performance. In contrast, our goal is to achieve multi-core parallelism through enumerative computation and use vector instructions to optimize the enumerative computation.

Parabix [18] converts an FSM computation into a sequence of bit operations where a bit is assigned for each input symbol. This allows Parabix to process W characters simultaneously on a SIMD machine of width W. Parabix uses bitstream addition [3] to preserve dependences across characters that occur in common text processing applications, such as XML parsing. While providing significant single-core speedups, Parabix achieves limited parallelization across multiple processors/cores. Moreover, their approach requires nontrivial processing to transform the input string into a sequence of bitstreams. For tasks such as XML processing, the resulting FSM is small enough that our implementation requires a single `shuffle` instruction per input symbol and we expect the single-core performance of our implementation to be competitive with Parabix. However, Parabix will provide better performance as future hardware architectures support larger SIMD widths. In contrast, our approach can scale the enumerative computation to larger FSMs with larger SIMD widths.

NR-grep [21] uses bit-parallelism to simulate a nondeterministic FSM for the purpose of pattern matching. By appropriately assigning the bits of a machine word (or a SIMD register), it transforms the FSM computation into a sequence of bit operations. While this technique can be optimized for special classes of patterns, matching regular expression in their generality requires an input-dependent memory lookup similar to the sequential algorithm in Figure 1(c).

Regular Expression Engines There is a large body of work on parallel regular expression matching on FPGAs [30, 34], GPUs [32], and CELL [28]. These approaches all parallelize regular expression matching by running multiple inputs in parallel on each hardware context. For example, Scarpazza et al. uses builds a $Flex$ like tokenizer to tokenize SGML. In contrast, our approach is data parallel *within* a single input and is complementary to these approaches.

Parallelizing Huffman Decoding Like the speculative approaches mentioned above, researchers have used speculation to speed up Huffman decoding in software [14] and in hardware [17, 33]. Klein et al. [14] parallelize Huffman decoding using the observation that decoding at different offsets are likely to *synchronize* after reading a few symbols. Such synchronization is not always guaranteed for certain codes—fixed length encodings, for instance. Convergence of finite state machines strictly generalizes synchronization and uniformly works for all codes. Others [5] have observed that if the compression algorithm is modified to generate independent compressed blocks, each of these blocks can be decompressed in parallel.

8. Conclusion

This paper demonstrates an efficient parallel algorithm finite state machines that uses enumerative computations. While an enumerative computation strictly performs more work than a sequential one, we demonstrate optimizations and implementation strategies, which in concert allow many of the FSMs we evaluate in this paper to take advantage of fine-grained parallelism (e.g., ILP/SIMD) and coarse-grained parallelism (e.g., multi-core).

We show significant multi-factor performance improvements on three real world problems, regular expressions from Snort, Huffman decoding, and HTML tokenization. This paper mainly focuses on parallelism found in a modern workstation; however, we believe our approach is suitable for *any* modern data parallel architecture, from GPUs to large clusters running MapReduce like frameworks. We also believe that future FSM compilers will be able to automatically explore the various tradeoffs described in the paper to obtain fast implementations for a given hardware architecture.

References

[1] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[2] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical Theory of Automata*, 12:529–561, 1962.

[3] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. Popowich. Parallel scanning with bitstream addition: An XML case study. In *European Conference on Parallel and Distributed Computing, Part II*, pages 2–13, 2011.

[4] B. Catanzaro, N. Sundaram, and K. Keutzer. A map reduce framework for programming graphics processors. In *In Workshop on Software Tools for MultiCore Systems*, 2008.

[5] R. L. Cloud, M. L. Curry, H. L. Ward, A. Skjellum, and P. Bangalore. Accelerating lossless data compression with GPUs. *Computing Research Repository (CoRR)*, abs/1107.1525, 2011.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce framework on graphics processors. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, 2008.

[8] W. D. Hillis and G. L. Steele. Data parallel algorithms. In *Commun. ACM*, volume 29, pages 1170–1183, Dec 1986.

[9] J. Holub and S. Štekr. On parallel implementations of deterministic finite automata. In *Implementation and Application of Automata*, CIAA '09, pages 54–64, 2009.

[10] J. E. Hopcroft and J. D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

[11] P. Howard and J. Vitter. Parallel lossless image compression using Huffman and arithmetic coding. In *Data Compression Conference, 1992. DCC '92.*, pages 299 –308, March 1992.

[12] Intel Haswell Microarchitecture, 2013. URL `http://software.intel.com/en-us/haswell`.

[13] C. G. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodík. Parallelizing the web browser. In *Hot Topics in Parallelism (HotPar)*, pages 7–7, 2009.

[14] S. T. Klein, Y. Wiseman, S. T. Klein, and Y. Wiseman. Parallel Huffman decoding with applications to JPEG files. *The Computer Journal*, 46:487–497, 2003.

[15] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[16] libhuffman. URL `http://huffman.sourceforge.net/`.

[17] C.-H. Lin and C.-W. Jen. Low power parallel Huffman decoding. *Electronics Letters*, 34(3):240 –241, Feb 1998.

[18] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. D. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *High Performance Computer Architecture (HPCA)*, pages 373–384, 2012.

[19] D. Luchaup, R. Smith, C. Estan, and S. Jha. Speculative parallel pattern matching. *IEEE Transactions on Information Forensics and Security*, 6(2):438–451, June 2011.

[20] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[21] G. Navarro. NR-grep: A fast and flexible pattern matching tool. *Software: Practice and Experience*, 31(13):1265–1312, 2001.

[22] Y. Pan, Y. Zhang, and K. Chiu. Simultaneous transducers for data-parallel XML parsing. *International Symposium on Parallel and Distributed Processing*, pages 1–12, 2008.

[23] V. Paxson. flex - fast lexical analyzer generator, 1988.

[24] G. R. Prakash Prabhu and K. Vaswani. Safe programmable speculative parallelism. In *Programming Languages Design and Implementation (PLDI)*, June 2010.

[25] Project Gutenberg. URL `http://www.gutenberg.org/`.

[26] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *Programming Languages Design and Implementation (PLDI)*, pages 118–131, 2006.

[27] M. Roesch. Snort - Lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, LISA '99, pages 229–238, 1999.

[28] D. P. Scarpazza and G. F. Russell. High-performance regular expression scanning on the Cell/B.E. processor. In *International Conf. on Supercomputing*, ICS '09, pages 14–25, 2009.

[29] S. Sengupta, M. Harris, Y. Zhang, and J. Owens. Scan primitives for GPU computing. In *SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 97–106. Eurographics Association, 2007.

[30] P. Sutton. Partial character decoding for improved regular expression matching in FPGAs. In *Field-Programmable Technology*, pages 25 – 32, Dec. 2004.

[31] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for shared-memory systems. In *Workshop on MapReduce and its Applications*, pages 9–16, 2011.

[32] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. Markatos, and S. Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Recent Advances in Intrusion Detection*, volume 5758, pages 265–283. 2009.

[33] B. Wei and T. Meng. A parallel decoder of programmable Huffman codes. *Circuits and Systems for Video Technology*, 5 (2):175 –178, Apr 1995.

[34] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *Architectures for Networking and Communications Systems*, ANCS '08, pages 30–39, 2008.

[35] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications Systems*, ANCS '06, pages 93–102, 2006.

[36] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Operating Systems Design and Implementation*, OSDI'08, pages 1–14, 2008.