MicroSpec: Speculation-Centric Fine-Grained Parallelization for FSM Computations

Junqiao Qiu, Zhijia Zhao
Department of Computer Science & Engineering
University of California, Riverside
jqiu004@ucr.edu, zhijia@cs.ucr.edu

Bin Ren
Department of Computer Science
The College of William and Mary
bren@wm.edu

ABSTRACT

Finite state machines (FSMs) are basic computation models that play essential roles in many applications. Enabling efficient parallel FSM execution is critical to the performance of these applications. However, they are very challenging to parallelize due to their inherent data dependencies that occur at each step of computations.

Existing efforts on FSM parallelization either explore coarse-grained speculative parallelism or leverage parallel prefix-sum. The former ignores prevalent fine-grained hardware parallelism on modern processors (such as ILP or SIMD parallelism) while the latter limits the benefits of fine-grained parallelism mainly to state enumeration.

This work presents *MicroSpec*, a set of parallelization techniques that, for the first time, expose fine-grained speculative parallelism to FSM computations. Based on a rigorous analysis of three types of parallelism at fine-grained level, *MicroSpec* consists of a list of four fine-grained speculative parallelization approaches along with a speculation-oriented data transformation. Experiments on a large set of real-world FSM benchmarks show that *MicroSpec* achieves substantial performance improvement over the state-of-the-art.

Keywords

Program Parallelization, Finite State Machine, FSM, SIMD, Speculative Parallelization, Vectorization

1. INTRODUCTION

Exposing parallelism is key to computing efficiency and scalability of software applications. Modern microprocessors feature a variety of hardware parallelism from instruction level to on-chip multiprocessors. Effectively leveraging such rich hardware parallelism critically affects the performance.

This work focuses on exposing effective fine-grained parallelism to Finite State Machine (FSM)-based computations, a class of computations that are frequently used in a wide range of applications, including *deep packet inspection* in network intrusion detection system (NIDS) [60, 15, 34, 39], *motif searching* in biological databases (e.g., Genebank and Prosite) [53, 11, 59, 8], *Huffman decoding* in image and video compression [3, 54, 24, 32], *path queries and validation* in semi-structural data stream [44, 9, 19, 58], and *model checking* in software engineering [45, 49, 10, 4], among others.

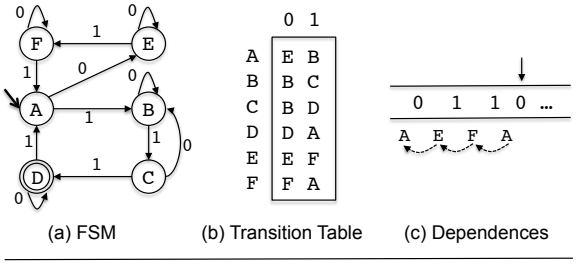

Figure 1: An FSM Example

FSMs are extremely challenging to parallelize, formerly known as "embarrassingly sequential" computations [5]. The challenges lie in the inherent data dependencies among state transitions. Figure 1 shows an example FSM with six states. The valid transitions in an FSM can be represented as a table, called *transition table*. Given an input string, the execution of an FSM starts from a predefined state (called *initial state*). Each time it reads one symbol from the input string. The FSM looks up the transition table based on the current state and the read symbol to find and transition to the next state. The dependence between the current state and next state exists at every transition step. These dependencies form a tight dependence chain, inherently preventing any parallelism from being exposed.

State of The Art. Given their fundamental importance in computing theory and their broad range of real-world applications, recent years have seen growing interests in parallelizing FSM computations, leading to some significant advancements. In general, they fall into two groups based on the types of parallelism that they explore: (1) speculative parallelization [61] and (2) parallel prefix-sum [40] . The former breaks the dependencies by predicting future states. In the cases that some predictions fail, reprocessing might be needed to ensure correctness. In comparison, the latter needs no prediction at all. Instead, it enumerates all the possible states, which always cover the actual one. With either of the two ways, an input string now is allowed to be partitioned into segments and processed in parallel.

Even though they break the barrier of making FSM computations run parallel, none of them has released the full po-

PACT '16, September 11-15, 2016, Haifa, Israel
 ISBN 978-1-4503-4121-9/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967965

tential of processing power in today's processors. The former relies on sophisticated prediction and only works at thread level; while the latter only exposes fine-grained parallelism to state enumeration – making transitions for each possible state. None of them can further shorten the critical execution path of an individual input segment (see Section 3.2).

To address these concerns, this work introduces two new dimensions of parallelism, *multi-state speculation* and *multi-level speculation*. The former extends the speculation from commonly used single-value prediction to multi-value prediction, while the latter expends the speculative parallelism across different layers of hardware parallelism. Based on a rigorous analysis on parallel prefix-sum and the two new types of parallelism, this work presents *MicroSpec*, a set of speculation-centric parallelization methods that maximize the efficiency of FSM computations by effectively exploiting fine-grained speculative parallelism. Specifically, *MicroSpec* consists of a list of four fine-grained speculation techniques as well as a speculation-oriented data layout optimization. Together, they are able to effectively exploit both Instruction-Level Parallelism (ILP) and Single Instruction Multiple Data (SIMD) parallelism [1].

Our evaluation of *MicroSpec* on a set of 17 FSM benchmarks from four application domains demonstrates its effectiveness in accelerating FSM computations, yielding about 14X speedup on 13 benchmarks, boosting the state-of-the-art by up to a factor of four.

In sum, this work makes the following contributions.

- It proposes two new dimensions to explore the fine-grained parallelism for FSM computations: *multi-state speculation* and *multi-level speculation*, which makes the parallelization design more flexible.
- Through a rigorous analysis on three types of parallelism for fine-grained FSM parallelization, it theoretically reveals the efficiency issue in the state-of-the-art and offers guidelines for the design of efficient FSM parallelization techniques.
- It designs and implements four speculation-centric fine-grained parallelization techniques which, for the first time, enable fine-grained speculative parallelization.
- It evaluates the proposed techniques on a large group of real-world benchmarks, demonstrating significant advancement over the state-of-the-art.

2. BACKGROUND AND PROBLEM

2.1 FSM and Its Dependences

FSMs form the backbone of a variety of applications, ranging from intrusion detection and data decompression to compilation and pattern searching. The core computation of these applications can be formulated as an abstract machine with a finite number of possible states. Transitions are allowed among certain states when satisfying given conditions. FSMs can be deterministic or non-deterministic depending on if a condition can lead to a unique following state. This work focus on deterministic ones for their better efficiency [2].

[1]This work focuses on vectorization on CPUs, but general ideas are applicable to SIMD parallelism on GPUs as well.

[2]Non-deterministic FSM can be converted to deterministic ones through subset construction.

Parallelizing FSM computations are extremely difficult due to their inherent sequential characteristics — dependencies exist between every consecutive state transitions, as illustrated by Figure 1 (c). A natural way to parallelize its execution is to partition the input string into to segments, and let thread process segments concurrently, one segment per thread. However, the starting states are unknown except the first thread (which starts from initial state 'A'). A starting state for a segment is essentially the ending state of the previous segment. These dependencies form a chain structure, preventing any concurrent execution among threads.

Existing work to solve this problem mainly follow two directions: speculative parallelization and parallel prefix-sum. Zhao and others [61] followed the first direction and proposed a coarse-grained speculative parallelization approach to circumvent the dependencies. Instead of speculation, Todd and others [40]'s approach enumerates all the possible cases to leverage classic parallel prefix-sum. They implemented with both coarse-grained and fine-grained parallelism to take advantage of different levels of hardware parallelism. However, each of them has its own limitations. The former is only able to explore coarse-grained thread-level parallelism, leaving widely available fine-grained hardware parallelism (such ILP and SIMD) unused. The latter uses fine-grained hardware parallelism only for enumerating different cases. None of them fully take advantage of the computing power of today's microprocessors.

Hence, the goal of this work is to maximize the parallel efficiency on modern processors by exposing more effective fine-grained parallelism to FSM computations. However, challenges exist at several levels. First, fine-grained parallelism is notoriously more difficult to expose comparing to coarse-grained thread-level parallelism due to the lack of friendly programming models. For example, programming with Intel SSE instruction set requires knowledge about microarchitecture and is more error-prone. Second, different types of parallelism exist for FSM computations, it is non-trivial to find out which ones are more effective at fine-grained levels. Third, fine-grained hardware parallelism varies across different architectures. For example, some microarchitectures may not support `gather` instruction, which is critical for fine-grained FSM parallelization (see Section 4.2).

2.2 Coarse-Grained Speculative Parallelization

As this work mainly follows the first direction – speculation-based parallelization, we briefly summarize its ideas for self-containedness. At high-level, there are four major steps in coarse-grained speculative FSM parallelization. To make it easier to follow, we use Algorithm 1 to illustrate its basic ideas, followed by a step-by-step explanation.

Algorithm 1 Coarse-Grained Speculative Parallelization

```
1: Π = coarse_grained_partition(N_core); /* Step 1 */
2: for thread 1 ··· N_core do
3:     S_start(i) = predict (suffix of Π(i − 1)); /* Step 2 */
4:     process(Π(i), S_start(i)); /* Step 3 */
5: thread_join();
6: for partition 1 ··· N_core do /* Step 4 */
7:     if validate(S_start(i)) == FALSE then
8:         reprocess(Π(i));
```

1. **Partitioning.** Given an input string of length L, it first cuts it evenly into N_{core} segments, where N_{core} is the number of available cores.

2. **Predicting Starting States.** For each segment (except the first one), it predicts its starting state with a technique called *lookback*. For segment i, lookback examines the suffix of its prior segment $i-1$ and uses it as conditions to rule out impossible states or states with low chances to be the correct starting state (more details in [61]). Later, a single state is selected as the predicted starting state.
3. **Parallel Execution.** With predicted starting states, it then executes each segment of length $L_{seg} = L/N_{core}$ in parallel. For each individual segment, this execution is the same as a sequential FSM execution.
4. **Validation and Reprocessing.** At last, it validates the correctness of the predicted starting states after the parallel execution. The validation compares the predicted starting state of segment i with the ending state of segment $i-1$, if they are different (i.e., prediction fails), segment i would be reprocessed.

Three things are important to note. First, According to prior results [61], the prediction accuracy highly depends on segment suffix, rather than how far it is away from the input beginning. Second, in Step 4, validations among different segments need to be in sequential order to ensure the correctness; Third, the reprocessing of a segment may stop earlier thanks to the *state convergence* property that widely exists in many FSMs. We elaborate this property using the example in Figure 2.

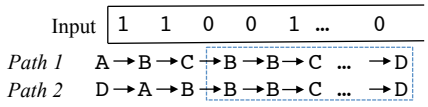

Figure 2: Example of State Convergence.

Consider processing a piece of an input string, starting with two different states A and D. There are two paths of state sequence, each for a different starting state. After processing the first three symbols `110`, both paths get into the same state B. Since then, these two paths would keep producing the same state sequence as they will observe the same symbols. This phenomenon is referred to as *State Convergence* [61, 40].

In the context of reprocessing, as long as the predicted (wrong) state converges with the actual starting state before reaching the end of the segment, the reprocessing can safely stop since the remaining states would be the same as the correct ones. In fact, state convergence is not only useful for speculative FSM parallelization, but also for parallel prefix-sum, where paths from different starting states may also converge and hence maintaining one of them is sufficient. We elaborate the details shortly in Section 3.2.

3. FINE-GRAINED PARALLELISM

Fine-grained parallelism is becoming increasingly prevalent in mainstream microprocessors, in a variety of forms, such as deep pipelining, multi-instruction issue, and SIMD vector units. For example, Intel's recent microarchitectures, `Haswell`, supports Advanced Vector Extensions 2 (AVX2) which features 256-bit vector units that can process 8 integer-typed data in parallel.

Effectively utilizing such fine-grained hardware parallelism is critical to maximizing the efficiency of various applications. In this section, we first discuss three types of parallelism that can be used in fine-grained level, two of which are proposed by this work. Then, we compare their effectiveness with a rigorous analysis, which in turn guides the design of FSM parallelization techniques.

3.1 Three Dimensions

The only fine-grained parallelism that has been seen in prior work comes from *associative parallelism* [35, 40]. We propose two other types of parallelism that are applicable to fine-grained levels, namely, *multi-state speculation* and *multi-level speculation*. We next elaborate each of them. For convenience, we refer to them as P1, P2, and P3.

P1: Parallelism in Associative Operations

Computations with associative operations can be trivially parallelized, such as multiplying a sequence of matrices. In fact, an FSM execution on an input sequence $c_1 c_2 \cdots c_L$ can also be associative. This is achieved by enumerating all the states in the FSM and making transitions for each of them, referred as *prefix-sum parallelism* by Ladner and Fischer [35].

In practice, as described in [40], it first cuts the input into T segments, then it enumerates all the n states for each segment except the first segment (which starts from *initial* state) to start transitions. After a segment has been processed, a mapping between each starting state and its ending state would be available. With the known *initial* state, it finally goes through every resulted mapping in order and selects the correct path. Clearly, it brings in $n-1$ times extra computations, where n is the number of states. It may be beneficial when the available hardware parallelism is more than n. However, with *state convergence* optimization, the extra cost can be dramatically reduced [40] (see Section 2.2).

P2: Parallelism in Multi-State Speculation

Existing work on speculative parallelization of FSMs partition the input based on the number of CPU cores and predict a single starting state for each segment, the one with the highest potential to minimize the misspeculation penalty. A straightforward extension to this approach is *speculating multiple starting states for each segment, instead of one.* The intuition is that the more candidates are used for prediction, the more likely the correct starting state gets covered and the more likely the misspeculation penalty gets reduced. Such extension enables new parallelism as each one of the speculated starting states can start its own path independently. We refer to it as *multi-state speculative parallelism.* The difference between single-state and multi-state speculation is significant because most previous work was based on single-value prediction, such as the BOP system [29].

Essentially, multi-state speculative parallelism provides a tradeoff between single-state speculative parallelization and parallel prefix-sum. It offers more flexibility to deal with FSMs that are hard to speculate and FSMs that are hard to enumerate due to a large number of states.

P3: Parallelism in Multi-Level Speculation

The third way to expose parallelism is further partitioning the N_{core} input segments into $N_{core} * W^{l-1}$ finer-grained segments recursively, assuming that W is the degree of parallelism at fine-grained levels (l is the number of levels). We refer to this type of parallelism as *Multi-Level Speculation.*

Since hardware parallelism is also hierarchical – a CPU has multiple computing cores, each with its own SIMD units

– multi-level speculation offers a natural mapping from software parallelism to hardware parallelism. For example, the first level speculative parallelism can be mapped to coarse-grained thread-level hardware parallelism (i.e., multicores), while the second level can be mapped to fine-grained ILP or SIMD parallelism. Note that such parallelism is not free; it may bring more overhead as it involves more speculation. We will shortly prove that it is still more efficient than the first two types of parallelism when used properly.

3.2 Efficiency Analysis

We next analyze the efficiency of the three types of parallelism theoretically. To facilitate our analysis, we bring two commonly used metrics into the context of FSM execution.

- *Expected Critical Path Length (ECPL).* This is the expected number of state transitions on the longest transition path of an FSM execution.
- *Degree of Parallelism (DoP).* This is the number of processing units that can be effectively used by an FSM execution.

For example, in a sequential execution, an FSM proceeds on a single transition path. Hence, $ECPL(seq) = L$, where L is the input length. As only one processing unit is used for all the transitions, we have $DoP(seq) = 1$.

Since state convergence is used by recent work [61, 40] for its large efficiency boost, we assume that it is applied in our discussion. Without loss of generality, we also assume that the input is partitioned into two segments at a coarse-grained level and the following analysis is on the second segment.

To analyze the effects of state convergence, we introduce two concepts: *convergence length* and *convergence matrix*.

Definition 1. *Given an input string I and two different starting states s_i and s_j. The convergence length between s_i and s_j on I is the least number of transitions for each of them to take in order to transition to the same state, denoted as $L^I(s_i, s_j)$. If by end of I, they end at different states, set $L^I(s_i, s_j) = \infty$.*

Consider the example in Figure 2, we have $L^I(A, D) = 3$. Based on this, we define *convergence matrix* as follows.

Definition 2. *Given an FSM with n states, the convergence matrix over an input I is an $n \times n$ matrix, where each element is the convergence length between states s_i and s_j on input I (i.e., $L^I(s_i, s_j)$), denoted as M_L.*

$$M_L = \begin{bmatrix} L^I(s_1,s_1) & L^I(s_1,s_2) & \dots & L^I(s_1,s_n) \\ L^I(s_2,s_1) & L^I(s_2,s_2) & \dots & L^I(s_2,s_n) \\ \vdots & \vdots & \ddots & \vdots \\ L^I(s_n,s_1) & L^I(s_n,s_2) & \dots & L^I(s_n,s_n) \end{bmatrix} \quad (1)$$

M_L has some properties: (i) It is symmetric as $L^I(s_i, s_j) = L^I(s_j, s_i)$; (ii) $L^I(s_i, s_i) = 0$; (iii) If $L^I(s_i, s_j) = l_1$, $l_1 \leq \|I\|$ and $L^I(s_j, s_k) = l_2$, $l_2 \leq \|I\|$, then $L^I(s_i, s_k) = max\{l_1, l_2\}$, where $\|\cdot\|$ means the length or number of transitions.

Convergence matrix embodies information about how states converge at each step during an FSM execution, it hence can help us reason about the reprocessing cost for P2 and P3.

In P1, each state starts its own transition path (denoted as $Path(s_i)$). Once a path finds that it converges with another path, one of the two paths would be *killed* (stopped), the other one would be kept *live*. Hence, the length of $Path(s_i)$ is the shortest convergence length between s_i and any other states, supposing that s_i converges with at least one of other states. Otherwise, its length would equal to the length of the input. Formally, we have

$$\|Path(s_i)\| = \min\{L^I(s_i, S - \{s_i\}), \|I\|\} \quad (2)$$

where s_i converges with $S - \{s_i\}$ when s_i converges with at least one state from $S - \{s_i\}$. Correspondingly, $L^I(s_i, S - \{s_i\}) = \min\{L^I(s_i, s_j) | s_j \in S - \{s_i\}\}$.

By definition, it is possible that $\|Path(s_i)\| < \|I\|$ for every s_i. To finish the whole input, one of the transition paths $Path(s_i)$, $s_i \in S$, has to continue $\|I\|$ - $\max_{1 \leq i \leq n}\{\|Path(s_i)\|\}$ transitions. Hence, the $ECPL$ of P1 is simply the input length.

Lemma 1. *Given an input I, the $ECPL$ of P1 is*

$$ECPL(P1) = \|I\| \quad (3)$$

On the other hand, the DoP of P1 may vary as the FSM executes depending on state convergence. Starting from all states S, when the number of live paths at the j-th input symbol, $live(S, j)$, exceeds the number of processing units, PU, the DoP equals to PU; Otherwise, the DoP drops to $live(S, j)$.

$$DoP(P1) = \min\{live(S, j), PU\}, \text{ where } 1 \leq j \leq \|I\| \quad (4)$$

In P2, suppose K states, denoted as S_K, are selected as the prediction. Since the selection does not change the path length of any state, if S_K covers the correct state, then $ECPL$ equals to the input length. Otherwise, it needs to reprocess the input until the correct state converges with one of selected K states. The reprocessing length is

$$\|redo\| = min\{L^I(s_i, s^*) | s_i \in S_K, s^* \text{ is the true state}\} \quad (5)$$

Assuming that the reprocessing in P2 runs sequentially, we have Lemma 2 holds for P2.

Lemma 2. *Given an input I, the $ECPL$ of P2 is*

$$ECPL(P2) = \|I\| + (1 - P_k) \cdot \|redo\| \quad (6)$$

where P_k is the probability that S_K covers the true state s^*.

Before reprocessing, the DoP of P2 is similar to P1; During reprocessing, the $DoP(P2)$ drops to one.

$$DoP(P2) = \begin{cases} \min\{live(S_k, j), PU\} & 1 \leq j \leq \|I\| \\ 1 & \text{redo} \end{cases} \quad (7)$$

In P3, the input segment is further cut into PU finer-grained chunks, each of them is processed with a predicted starting state $\hat{s}_i$, $1 < i \leq PU$. Suppose the probability of each predicted starting state is $p(\hat{s}_i)$ and the corresponding reprocessing length is $redo(\hat{s}_i)$, then the expected amount of reprocessing is

$$\|\overline{redo}\| = \sum_{i=2}^{PU} (1 - p(\hat{s}_i)) \cdot \|redo(\hat{s}_i)\| \quad (8)$$

Note that the reprocessing of different chunks runs sequentially, since the correctness validation of chunk i depends on the validation of chunk $i - 1$. This is also true at coarse-grained level. Hence, the expected reprocessing length for the whole input should include the reprocessing at both coarse-grained and fine-grained levels, that is, replacing PU in Equation 8 with $PU \cdot (T - 1)$, where T is the number of threads at coarse-grained level.

Putting them together, we have Lemma 3 for P3.

LEMMA 3. *Given an input I, the ECPL of P3 is*

$$ECPL(P3) = \|I\| / PU + \left\|\overline{redo}\right\| \quad (9)$$

According to Lemma 3, any misspeculation has the potential to lengthen the critical path, compromising the benefits of speculative parallelization. In the worst case, when all prediction fails, $ECPL(P3)$ would equal to the input length, the same as a sequential execution.

As each processing unit processes a different input chunk, no state convergence would happen. Hence, the DoP of P3 equals to PU before reprocessing and drops to one during reprocessing.

$$DoP(P3) = \begin{cases} PU & 1 \leq j \leq \|I\| \\ 1 & \text{redo} \end{cases} \quad (10)$$

Discussion. Based on the above analysis, we compare the three types of parallelism in terms of both $ECPL$ and DoP.

First, $ECPL$ captures the expected execution length. For P1 and P2, since enumerating all states or a set of states do not shorten the critical path, $ECPL(P1)$ and $ECPL(P2)$ at least equals to the segment length. In comparison, by cutting the segment into finer-grained chunks, $P3$ have the chances to further shorten the critical path length. However, due to the dependence in reprocessing, the $ECPL$ of P3 could be as long as the whole input length, which happens when all prediction fails.

Second, DoP captures the utilization of fine-grained hardware parallelism. $DoP(P1)$ and $DoP(P2)$ start dropping when the number of live paths goes below the number of processing units PU. In another word, some of the processing units become idle. Unfortunately, $DoP(P3)$ cannot guarantee full utilization all the time neither, due to possibility of sequential reprocessing.

Overall, the efficiency of a type of parallelism depends on the properties of FSMs and hardware architecture (e.g., PU). In this work, we choose P3, mainly based on our observation that the reprocessing lengths are usually short thanks to the quick state convergence. This has two positive consequences. First, it ensures that $ECPL(P3)$ is usually much shorter than segment length (see Section 5). Second, it guarantees high hardware utilization by keeping $DoP(P3)$ mostly as high as PU.

4. MICROSPEC

Guided by the analysis in Section 3, we design and implement *MicroSpec*, a library that leverages multi-level speculation to maximize the efficiency of parallel FSM execution on modern processors. We first describes its major techniques, then introduces an optimization to facilitate its use.

4.1 Overview

At high-level, *MicroSpec* consists of four speculation-centric parallelization techniques (denoted as S1 - S4) plus a speculation-oriented data transformation. The parallelization techniques are able to expose fine-grained speculative parallelism to FSM computations while the data transformation automatically re-layouts the input for better locality.

Predicting Starting States. Since starting states prediction is not the focus of this work, we simply choose a relatively straightforward prediction, named *simple lookback*, which has been used by prior work [61, 1]. Basically, it starts from the suffix of a prior segment with a random state, then uses its ending state after processing the suffix as the predicted starting state. More advanced predictions can be ported to *MicroSpec*. However, there will be a tradeoff between accuracy and overhead, which remains to be investigated. In the following, we elaborate these four major techniques and the optimization in details.

4.2 Techniques

In multi-level speculation, each level follows a speculative parallelization scheme that is similar to the one in Algorithm 1. The key differences lie in the implementations. In the following, we consider two cases: two-level speculation and three-level speculation. For the first level, that is, the coarse-grained level, we simply follow the coarse-grained speculative parallelization in Algorithm 1. For the second and third levels, we mainly focus on ILP and SIMD parallelism, both of which are common features owned by modern processors. As the first level is given in Section 2, in the following, we only show the algorithms in the second and third levels. Next, we first present two two-level speculations, followed by two three-level ones.

S1: Speculative SIMD Gather

We first consider SIMD parallelism only for the second-level speculation. Algorithm 2 shows the pseudo-code of this approach. As this approach mainly relies on SIMD operation `gather`, we refer to it as *Speculative Gather*.

Algorithm 2 Speculative SIMD Gather

```
1: π = fine_grained_partition(W);
2: S = predictInitStates(π);
3: for (i=0; i < L_seg/W, i++) do
4:     I = readInputVec(i);
5:     F = S × N_sym + I;
6:     S = gather(T, F);
7: end
```

Basically, given an input segment of length L_{seg} from the first-level speculation, speculative gather partitions it based on the SIMD width W (e.g., $W = 8$ for 256-bit integer operations) (Line 1). Then, it predicts the starting states for the W smaller segments with simple lookback (Line 2). Since there are no dependencies among predictions, they can be vectorized with SIMD operations as well.

With the predicted starting states, stored in a vector S, it goes through W smaller segments in parallel with SIMD operations, as shown through Lines 3 to 6 in Algorithm 2. The `readInputVec()` can be implemented either in SIMD operation or a sequence of non-SIMD `read` operations. To find next states, it accesses the transition table T, which is stored in a state-major one-dimensional array. This is finished in two steps. First, it calculates the address of next states and stores them in the offset vector F. Then it leverages a single `gather` operation to load W next states to vector S.

To illustrate the functionality of `gather`, consider the running example. Suppose the SIMD width $W = 8$, current state vector S = [D, C, A, C, F, A, E, B] (i.e., [3, 2, 0, 2, 5, 0, 4, 1]), input vector I = [1, 0, 0, 1, 1, 0, 1, 0], then offset vector $F = S \times 2 + I$ = [7, 6, 0, 5, 11, 0, 9, 2]. The next state vector would be $S = gather(base, F)$ = [A, B, E, D, A, E, F, B].

S2: Speculative Unrolling

Alternatively, we can also consider unrolling for the second-level speculation. Unrolling is one of the major ways to expose ILP. However exposing such low-level parallelism is not

straightforward. In fact, by default, due to the tight dependencies across state transitions, unrolling does not provide any benefits. As shown in Figure 3, the performance of after unrolling is almost the same as the default version. This is mainly because the state transition dependencies turn into instruction dependencies, making most unrolled instructions incapable of executing in parallel.

To overcome the above difficulty, we apply the idea of speculation to unrolling, aiming to break the most dependencies among the unrolled instructions. We refer to it as *Speculative Unrolling*, illustrated by Algorithm 3.

Algorithm 3 Speculative Unrolling

```
1: π = fine_grained_partition(R);
2: s[0 ··· R − 1] = predictInitStates(π);
3: B = L_seg/R;
4: for (i=0; i < B, i++) do
5:     c[0] = readInput(i);
6:     s[0] = T[s[0]][c[0]];
7:     c[1] = readInput(i + B);
8:     s[1] = T[s[1]][c[0]];
9:     ··· ···
10:    c[R-1] = readInput(i + B ∗ (R − 1));
11:    s[R-1] = T[s[R-1]][c[R-1]];
12: end
```

The basic idea of speculative unrolling is as follows. At first, it takes a coarse-grained input segment and partitions it into finer-grained segments according to the unrolling factor, R. Then it predicts the starting state for each fine-grained segment. So far, it is the same as S1, speculative SIMD gather. The difference is in the next. Instead of using some SIMD operations, it unrolls the loop body R times, with a goal to bring in artificial ILPs. Note that, with starting state prediction, the unrolled loop iterations do not have any dependencies, hence, can be executed in parallel and optimized by microprocessors.

A key question in speculative unrolling is the selection of unrolling factor R. If choosing R too high, it takes more risks of bringing in misspeculated segments; If choosing R too low, it may not fully utilize the potential of ILPs in microprocessors. In Section 5, we will examine this with experiments.

Discussion. Note that both of the above approaches rely on speculation to expose fine-grained parallelism. The former exposes SIMD parallelism while the latter exposes ILP. They are essentially orthogonal, hence, might be combined to expose even richer parallelism, the third-level speculative parallelism, pushing the utilization of microprocessor to the extreme. Depending on the order that they are combined, we refer to the combined approaches as *Speculative SIMD Gather+* and *Speculative Unrolling+*, respectively. We elaborate them next, namely, S3 and S4.

S3: Speculative SIMD Gather+

Intuitively, this approach applies speculative unrolling to speculative SIMD gather. This essentially requires more speculation, in particular, $W \times R$ times of speculation for a coarse-grained input segment, where W is the SIMD width and R is the unrolling factor. Algorithm 4 describes this approach. Basically, the loop body in Algorithm 2 is unrolled R times as that in Algorithm 3. Note that the number of loop iterations drops to $L_{seg}/W/R$.

Similarly to speculative unrolling, it also needs to select the loop unrolling factor R. An interesting question is whether it has a smaller optimal R comparing to that of speculative unrolling. We show our findings to this question in Section 5.

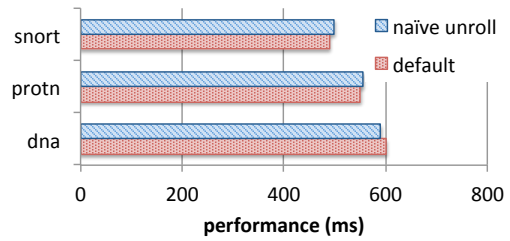

Figure 3: Performance of Naive Unrolling

Algorithm 4 Speculative SIMD Gather+

```
1: π = fine_grained_partition(W × R);
2: S[0 ··· R − 1] = predictInitStates(π);
3: B = L_seg/W/R;
4: for (i=0; i < B, i++) do
5:     I[0] = readInputVec(i);
6:     F[0] = S[0] × N_sym + I[0];
7:     S[0] = gather(T, F[0]);
8:     I[1] = readInputVec(i + B);
9:     F[1] = S[1] × N_sym + I[1];
10:    S[1] = gather(T, F[1]);
11:    ··· ···
12:    I[R − 1] = readInputVec(i + B ∗ (R − 1));
13:    F[R − 1] = S[R − 1] × N_sym + I[R − 1];
14:    S[R − 1] = gather(T, F[R − 1]);
15: end
```

S4: Speculative Unrolling+

Different from S3, *speculative unrolling+* first applies speculative unrolling to the second level of speculation, then applies speculative gather to the third level. The pseudo-code of this approach is illustrated as in Algorithm 5. Each for-loop corresponds to the unrolling as in S2. Within each for-loop, a segment is further partitioned into W segments to initiate speculative gather. Similarly to S3, S4 also aims to realize the maximal utilization of the processing power by aggressively increasing the amount of speculation.

In sum, S1 and S2 are based on two-level speculation, while S3 and S4 are based on three-level speculation. The total number of partitions increases from W and R in the former cases to $W \times R$ in the latter cases.

4.3 Optimization

For coarse-grained speculative parallelization, the input is partitioned evenly into coarse-grained segments based on the number of cores. Each thread sequentially accesses its own segment which is stored in a piece of continuous memory (since inputs are arrays). In this case, the locality is ideal. However, when multi-level speculation is used, the accessing pattern is not sequential any more, instead, it becomes stride-based. Even worse, the width of stride is typically large (i.e., the length of a fine-grained segment). This non-coalesced memory accessing pattern could drag the performance benefits down.

To overcome this, we propose a *speculation-oriented data transformation* that re-layouts the data according to the accessing pattern in multi-level speculation scheme to minimize the memory accessing delays. Basically, it transforms

Algorithm 5 Speculative Unrolling+

π = fine_grained_partition($W \times R$);
$S[0 \ldots R-1]$ = predictInitStates(π);
$B = L_{seg}/R$;
for (i=0; $i < B/W$; i++) **do**
    $I[0]$ = readInputVec(i);
    $F[0] = S[0] \times N_{sym} + I[0]$;
    $S[0]$ = gather(T, $F[0]$);
end
for (i=B; $i < B + B/W$; i++) **do**
    $I[1]$ = readInputVec(i);
    $F[1] = S[1] \times N_{sym} + I[1]$;
    $S[1]$ = gather(T, $F[1]$);
end
... ...
for (i=$B*(R-1)$; $i < B*(R-1) + B/W$; i++) **do**
    $I[R-1]$ = readInputVec(i);
    $F[R-1] = S[R-1] \times N_{sym} + I[R-1]$;
    $S[R-1]$ = gather(T, $F[R-1]$);
end

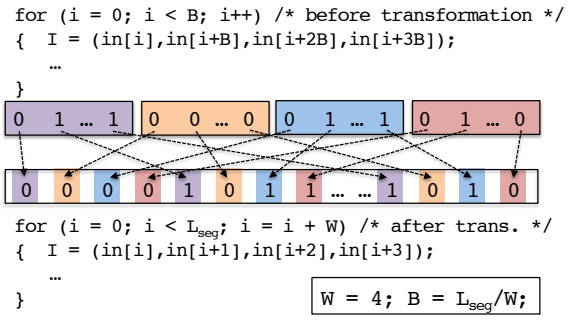

Figure 4: Spec.-Oriented Data Transformation

the big stride-based accessing to simple sequential accessing. It does this by moving each group of stride-based accessed data next to each other, as shown in Figure 4.

Consider S1, speculative SIMD gather [3]. Suppose the SIMD width $W = 4$, a coarse-grained input segment with length of L_{seg} is further partitioned into four fine-grained segments, each with a length of $B = L_{seg}/W$. To get an input vector I (as in Algorithm 2), the original memory accessing has a stride width of B. After the data transformation, the memory accessing becomes strictly sequential.

Speculation-oriented data transformation can either work offline (pre-layout) or online (on-the-fly re-layout). In many scenarios, the whole dataset is available and stable and different FSMs are executed over the same dataset many times. A typical example is biological sequence analysis, which may search different patterns on the same sequence database multiple times. Though the database may be updated sometimes, it is expected that updating rate is much lower than accessing rate. For scenarios like this, it is reasonable to do offline data transformation as the cost of data transformation will be amortized across different FSM executions.

4.4 Implementation

We prototyped *MicroSpec* as a C library using `Pthread` and Intel's `AVX2` instruction set. The library provides a uniform interface to various FSMs through a set of APIs, which implement both the four speculative parallelization methods and the data transformation. The major arguments to the APIs include the FSM `FSM*` and input `char*`. Other parameters such as the number of threads are automatically configured. In terms of FSM formats, it supports both transition table and `dot` file (a graphical FSM representation). It can also take regular expressions as arguments with the help of some off-the-shelf regular expression processors.

The compilation of *MicroSpec* depends on the use of the APIs. Since S1 does not include any SIMD instructions, it can be compiled even on machines without `AVX2` using standard C compilers, such as GCC or ICC. In comparison, the implementations of S2-S4 use `_mm256_i32gather_epi32` instruction from `AVX2`, hence need to be compiled on recent Intel microarchitectures, such as `Haswell` and its successors. We implement the data transformation in two versions: an API call that can be invoked by S1-S4 at runtime and a standalone tool that runs the transformation offline.

[3]Similar analysis is applicable to other three fine-grained speculation techniques in *MicroSpec*.

5. EVALUATION

In this section, we evaluate the effectiveness of *MicroSpec* using a set of real-world FSM applications that are manually collected from different domains, including *motif searching* in Bioinformatics, *rule matching* in NIDS, and *Huffman decoding* in data decompression, among others.

5.1 Methodology

The evaluation of *MicroSpec* includes all four speculation-based fine-grained parallelization techniques as well as the speculation-oriented data transformation. Table 1 summarizes them and lists their abbreviation used in the evaluation.

Table 1: *MicroSpec* Framework

Techniques in *MicroSpec*	Abbreviation
S1: Speculative SIMD Gather	SpecGather
S2: Speculative Unrolling	SpecUnroll
S3: Speculative SIMD Gather+	SpecGather+
S4: Speculative Unrolling+	SpecUnroll+
Spec.-Oriented Data Trans.	SpecTrans

We compare *MicroSpec* with prior techniques, the coarse-grained speculative parallelization [61] and parallel prefix-sum [40], including both state convergence and range coalescing optimizations. For convenience, we refer to them as *coarseSpec* and *prefixSum*, respectively. Our implementations are based on our best understanding of their papers.

Our major experiments run on a quad-core machine equipped with Intel 2.8GHz Xeon E5-1603 v3 processor with AVX2. The machine runs CentOS Linux 7.2.1511 and has GCC 4.8.5. For comparison, we also tested a machine without AVX2 supports. It is a quad-core machine equipped with Intel 3GHz Xeon CPU E5-1607 v2 processor with SSE 4.2. It runs Ubuntu 14.04.4 LTS and has GCC 4.9.3.

All programs are compiled with "-O3" optimization flag. The timing results reported are the average of 10 repetitive runs with all runtime cost included. We do not report 95% confidence interval of the average when the variation is not significant. In fact, we found that the measurements are usually stable since FSM executions involve a large amount of repetitive but similar computations.

5.2 Benchmarks

The benchmarks are selected to cover a wide range of FSM applications with different levels of complexities. We first

Table 2: Protein Motifs. [·] means alternative symbols; 'x' means any symbol; (·) is the number of repetition.

Bench	Description and Regular Expression
protn1	IQ calmodulin-binding motif. `[FILV]Qx(3)[RK]Gx(3)[RK]x(2)[FILVWY]`
protn2	Hemopexin domain signature. `[LIFAT]ILx(2)Wx(2,3)[PE]xVF[LIVMFY][DENQS][STA][AV][LIVMFY]`
protn3	P-type 'Trefoil' domain signature. `[KRH]x(2)Cx[FYPSTV]x(3,4)[ST]x(3)Cx(4)CC[FYWH]`

Table 3: Snort Rules.

Bench	Description and Regular Expression
snort1	(\xff{32})|([0-9A-F]{22})|(Color|Motion)
snort2	(\xFF\x41)|(Start)|(\/999)
snort3	(admin|axis2)|(\x3d?\x3d\r\n)|([rs]{4})
snort4	(\x2F\d{10})|(L\d\d\x00)|(POST\s)
snort5	(asp\x5C)|(2x\/.*php)|(htr\x5C)
snort6	snort1 | snort2 | snort3 | snort4 | snort5

elaborate them by groups, then summarize their statistics.

Biological Sequence Analysis. Pattern searching is a basic way to analyze biological sequences, such as DNA sequences or protein sequences. For example, a DNA motif is a short pattern of nucleic acid, while a protein motif is a pattern of amino acids. Usually, protein patterns are represented as regular expressions. Table 2 lists three protein patterns randomly selected from a widely used protein database PROSITE [13]. For DNA motifs, they are more commonly represented with Hamming distances. In our benchmarks, *dna1* is a DNA motif `ATCGGTCC(8,3)`, which means three of the eight preceding symbols can be different as specified. Similarly, *dna2* and *dna3* are two other DNA motifs `TCGAGGACCA(10,4)` and `AGGGTAAA(8,1)`, respectively. We converted the above protein and DNA motifs to FSMs using standard regular expression transformation algorithms.

Intrusion Detection Rule Matching. Network Intrusion Detection Systems (NIDSs) use regular expressions (called *signatures*) to detect malicious activities on the internet traffic. Among various NIDSs, Snort [52] is arguably the most widely used open source NIDS. It has a rich body of signatures/rules, most of them have a `pcre` field, where a Perl-compatible regular expression is used to specify the pattern interested.

In our evaluation, we randomly chose a set of 15 PCRE patterns from 15 signatures in Snort version 2.9.8.0 as our benchmarks. They are then randomly put into 5 groups, each with 3 patterns. We created the 6th group by putting all the 15 PCRE patterns together. Each group then is compiled to a single FSM using off-of-shelf PCRE to FSM tools. Table 3 lists the 6 groups with their PCRE patterns. The inputs to the Snort FSMs are network traffic trace collected from a Linux server and a laptop via `tcpdump`.

Mixed FSM benchmarks. This group contains a mixed set of FSM benchmarks, including Huffman decoding, mathematical testing and a couple of searching patterns.

For its optimality, Huffman algorithm has been widely used for encoding and decoding digital data (e.g., text, JPEG and MPEG). During the decoding stage, an FSM is employed to automate the decoding process. Basically, a Huffman decoding FSM contains a set of accept states, each of them corresponding to a code. It runs over an encoded (binary) file. Each time it reaches an accept state, a code is recognized. Note that this work targets the decoding phase, as the encoding phase is embarrassingly parallel [23].

Our Huffman FSM benchmark *huff* is built based on a collection of e-books downloaded from Project Gutenberg (as of Dec 15th, 2015). To make the decoding FSM more applicable, we created a single Huffman tree and a single decoding FSM that are capable of encoding any text files with ASCII symbols and decoding them, respectively. Since extended ASCII contains 256 symbols, there are 256 accept states (i.e., leaf nodes of Huffman tree). Together with 255 non-accept states, *huff* consists of 511 states. The inputs to *huff* are binary files that encode a large collection of e-books.

Mathematical testing benchmarks include *div* and *evenodd*. The former tests if a binary sequence is divisible by seven while the latter tests if a text file of $\{a, b, c, d\}$ satisfies that $|a|+|b|$ is even and $|c|+|d|$ is odd, where $|\cdot|$ means the number of appearances in the file.

We also include two searching patterns that are more challenging to speculate, namely, *commadot* and *likeapple*. Their patterns are $((.+,.+\backslash.)\{4\}|(.+\backslash,)\{4\}|(.+\backslash.)\{4\})\{3\}$ and $(.*l.*i.*k.*e)\{6\} \mid (.*.a.*.p.*.p.*.l.*.e)\{5\}$.

Table 4 summarizes the benchmarks used in our evaluation, including the total number of states, the number of accept states, state visiting frequency range and the state range after range coalescing optimization [40].

Table 4: Summary of FSM Benchmarks.

Bench	#States	#Accept	FRange	CRange
dna1	371	76	0 - 3.6%	133
dna2	2871	583	0 - 2.1%	953
dna3	40	5	0 - 32.9%	15
protn1	69	6	0 - 73.4%	31
protn2	281	14	0 - 24.7%	99
protn3	832	48	0 - 61.9%	509
snort1	86	4	0 - 56.0%	32
snort2	10	1	0 - 99.4%	4
snort3	15	2	0 - 91.3%	5
snort4	19	1	0 - 98.7%	13
snort5	20	3	0 - 91.3%	5
snort6	299	22	0 - 44.7%	72
huff	511	256	0 - 11.3%	255
div	7	1	14.28%	7
evenodd	4	1	25%	4
commadot	130	7	0 - 97.1%	81
likeapple	495	1	0 - 88.3%	494

5.3 Results

Unrolling Factor. Since the selection of the unrolling factor R may affect the performance of *MicroSpec*, we first discuss it. Table 5 shows the execution time of *dna4* on a small testing input using different unrolling factor values. The results answer the question in Section 4.2 – the best R varies across methods, 6 or 8 for *SpecUnroll*, 2 for *SpecGather+* and *SpecUnroll+*. This implies that the ILP for SIMD operations is less effective than the one for non-SIMD operations. Since we found that the best Rs are stable across different benchmarks, we empirically set $R = 8$ for *SpecUnroll* and $R = 2$ for *SpecGather+* and *SpecUnroll+* in the following.

Group A: Motif Searching. Figure 5 shows the perfor-

Table 5: Unrolling Factor Selection

exec. time(ms)	unrolling factor				
method	1	2	4	6	8
SpecUnroll	397.3	199	100.7	73.3	75.6
SpecGather+	145.6	97.6	188	484.1	251.7
SpecUnroll+	147.9	94.6	129.7	299.6	210

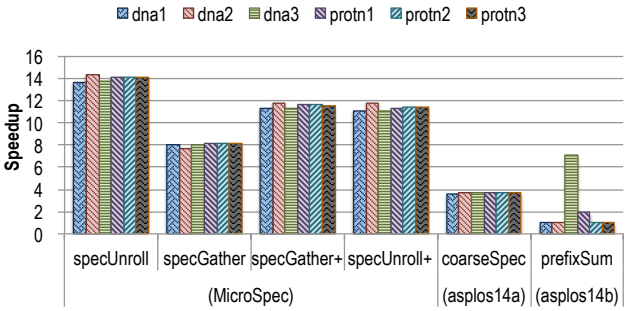

Figure 5: Speedups for Biological Benchmarks

mance results for *motif searching* benchmark group. Overall, the performance of four speculation-based methods in *MicroSpec* outperform previous two methods substantially, achieving about 14X speedups among all six benchmarks.

More specifically, *specUnroll* yields the best speedups among all tested methods. This implies that even though modern processors come with highly optimized ILP, they can be barely utilized by the default version. *specGather* yields about 8X speedup on average, also exceeding prior methods. It demonstrates the benefits of utilizing `gather` intrinsic from Intel AVX2 for FSM computations. However, on the other hand, it barely reaches around 60% performance of *specUnroll*, which indicates that the limitation of current `gather` compromises the speculation benefits. Methods *specGather+* and *specUnroll+*, yield similar speedups, higher than *specGather* but lower than *specUnroll*.

Note that *prefixSum* yields inconsistent speedups across different benchmarks. The reason is that its performance depends on the properties of FSMs. For FSMs with fast convergence length and less number of states, it tends to perform much better. For example, it gets about 7X speedup on benchmark `dna3`, which has only 40 states. These states converge quickly to a single state within 50 transitions. In comparison *coarseSpec* shows consist but limited speedups due to its unawareness of fine-grained parallelism.

Group B: Snort Rules Matching. Figure 6 shows the performance results for *Snort rules* benchmarks. In general, the results are similar to those in the first group. The main differences come from *prefixSum*, which achieves the best speedups for two benchmarks `snort2` and `snort3`. The reason is that both benchmarks have less than 16 states, smaller than the maximal number of states that a single SIMD shuffle (`_mm_shuffle_epi8`) can handle. This means it only needs a single `shuffle` instruction for each transition. Hence, this shows the optimal speedup of *prefixSum*. Comparing with *specGather*, this also validates that *shuffle* is much more efficient than *gather* on current processors.

Group C: Mixed FSM benchmarks. Figure 7 shows the performance results of the last benchmark group, which are mixed with Huffman decoding (*huff*) and some hard-

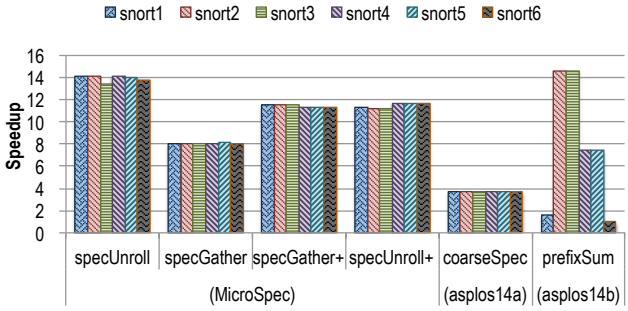

Figure 6: Speedups for Snort Benchmarks

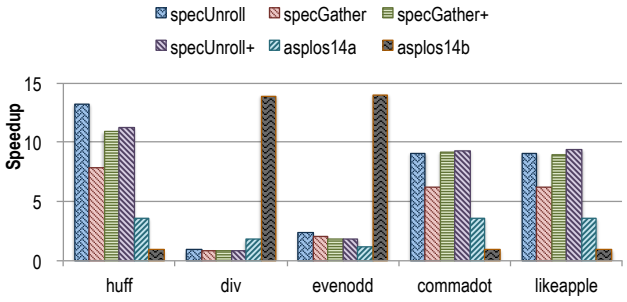

Figure 7: Speedups for Mixed FSM Benchmarks

to-speculate FSM benchmarks *div*, *evenodd*, *commadot*, and *likeapple*. After range coalescing, *huff* has a state range of 255. Though it can be executed by *prefixSum* using a mix of `shuffle` and `blend` operations, it hardly gets any benefits due to the large number of SIMD operations involved. In comparison, the four methods from *MicroSpec* show similar speedups on `huff` as those in the previous two groups.

The other four benchmarks in this group are more difficult to speculate due to their special structures. For *div* and *evenodd*, no states converge no matter what input sequences they are given. In this case, *MicroSpec* either shows limited improvement, about 2X speedup on *evenodd* or even performance degradation, about 10% slowdown on *div*. In comparison, *prefixSum* reaches 1.39X and 14X speedups, respectively, thanks to its speculation-free property and the small number of states in these two benchmarks (7 and 4). The other two benchmarks, *commadot* and *likeapple*, have relatively large number of states, meanwhile most states take long distances to converge (often exceeding 10K transitions). In this situation, *MicroSpec* gets about 8-9X speedup on average. Note that *specUnroll+* and *specGather+* all get similar or better performance than their counterparts, demonstrating the potential of combining SIMD *gather* and with speculation unrolling. In comparison, *prefixSum* could not get any speedups due to a large number of states in these two benchmarks (130 and 495).

Optimization *specTrans*. Table 6 shows the cost of *specTrans* optimization. In fact, the cost is quite comparable to the FSM execution time, about 1/3 of the sequential FSM execution time for input size of 100MB. Hence, it is recommended to used only offline, where the same datasets are reused across different FSM executions, such as different mo-

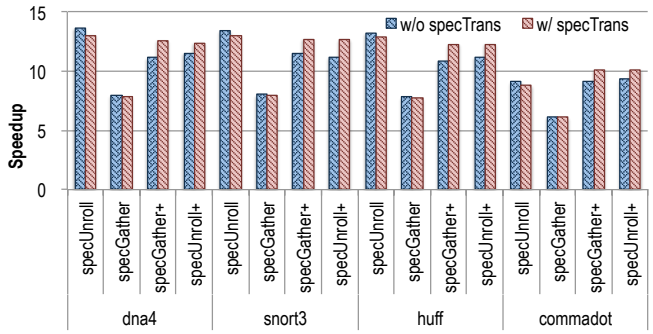

Figure 8: Performance Improvements of *specTrans*

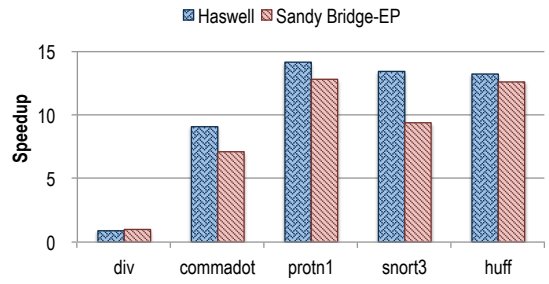

Figure 9: Performance on Different Machines

tif queries to the same DNA or protein sequence database. Figure 8 shows the improvements of *specTrans* optimization. On average, it brings about 8.5% extra speedup.

Table 6: Cost of *specTrans* (ms)

	num. of chunks			
input size	2	4	8	16
10MB	15	13	13	12
100MB	122	105	93	99
1GB	63K	63K	83K	64K

Comparison on Different Architectures. Finally, we also tested *MicroSpec* on an architecture without SIMD gather. In this case, only S2, *specUnroll* is experimented. Figure 9 summarizes the results. `Haswell` has AVX2, which supports an 8-way integer SIMD gather (`_mm256_i32gather_epi32`). In comparison, `Sandy Bridge EP` only comes with an earlier version of instruction set AVX. Overall, the performance on `Sandy Bridge EP` is slightly less than `Haswell`; but both follow a similar pattern. This demonstrates the potential of *MicroSpec* in a larger scope, across different architectures.

6. RELATED WORK

Program parallelization in general has received many efforts from various aspects, including but not limited to language design (e.g. Cilk [17], X10 [7]), hardware support (e.g., TLS [55, 21]) and programming models (e.g., STM [2, 6]). This section focuses the studies that are closed to FSM and speculative parallelization.

FSM and Speculative Parallelization. Traditional ways to parallelize FSM are through parallel prefix-sum or its variations [35]. Todd and others [40] implement this method on machines with vector units with a couple of optimizations. Some other FSM parallelization work focus on a few specific FSM applications, such as browser front-end [27] and JPEG decoder [31]. The basic ideas in these work were later formalized by Zhao and others [61] by introducing a concept called *principled speculation*. Other examples include hot state prediction for FSMs in intrusion detection [38] and speculative parsing [28]. For non-FSM applications, speculative parallelization has been studied for many years, including designing new language constructs [47] and parallelization frameworks [50, 12, 48, 56, 14]. Some of these studies have explored parallelism in irregular programs [33, 20, 46], which provide useful insights for exploiting parallelism in FSM computations, given that FSMs essentially run on an irregular data structure (a graph). They are mainly based on coarse-grained speculative parallelism. Some other prior work have explored bit-parallel fine-grained parallelism for FSMs by converting FSM computations into a sequence of bit operations [41, 36]. In comparison, this work uses both fine-grained and coarse-grained speculative parallelism. Integrating such bit-level parallelism to *MicroSpec* would be an interesting research topic that remains to be studied.

Vectorization and GPU. Vector extensions, such as SSE, have been widely used in many applications such as graphics [25], scientific applications [18], and signal processing [16]. Based on such vector extensions, many efforts have been put into auto-vectorization [42, 43, 57, 22]. However, there are still computations that are difficult to vectorize, due to irregular data structures, heavy branch operations, and noncontinuous memory access. To address these, prior work have vectorized binary tree search [30], tree traversal [26], irregular tree forest and multiple graphs traversal [51], and sparse matrix-vector multiplication [37]. The basic principles of vectorizing irregular applications include both improving the cache performance by careful data layout and resolving the branch operations. FSM parallelization has also been studied on GPU platform in the context of NFAs [62] which naturally run in parallel. Since FSMs are also irregular structure, the results of this work would provide valuable insights to the parallelization of other irregular computations.

7. CONCLUSION

This paper provides a rigorous analysis among three types of parallelism that can be exposed at fine-grained levels for FSM computations. It deepens the understanding to the efficiency of different FSM parallelization schemes. Guided by the analysis, it presents *MicroSpec*, a set of speculation-centric parallelization techniques that expose fine-grained speculative parallelism into FSM computations, along with a data transformation optimization. *MicroSpec* extends the available parallelism in FSM computations to a new level. Experiments show that *MicroSpec* outperforms the state-of-the-art by up to a factor of four, demonstrating the benefits of fine-grained speculative parallelism.

8. ACKNOWLEDGMENTS

We thank all anonymous reviewers for their constructive comments and our paper shepherd Keshav Pingali for his great help with the final version preparation.

9. REFERENCES

[1] Browsing web 3.0 on 3.0 watts: Why browsers will be parallel and implications for education. invited talk at The 3rd Workshop on Software Tools for MultiCore Systems, April, 2008.

[2] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2008.

[3] T. Algra. Fast and efficient variable-to-fixed-length coding algorithm. *Electronics Letters*, 28(15):1399–1401, 1992.

[4] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, May 2001.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[6] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming langauges. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[8] Y. Chen, D. Che, and K. Aberer. On the efficient evaluation of relaxed queries in biological databases. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, CIKM '02, pages 227–236, New York, NY, USA, 2002. ACM.

[9] C. Chitic and D. Rosu. On validation of XML streams using finite state machines. In *Proceedings of the Seventh International Workshop on the Web and Databases, WebDB 2004, June 17-18, 2004, Maison de la Chimie, Paris, France, Colocated with ACM SIGMOD/PODS 2004*, pages 85–90, 2004.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[11] S. Datta and S. Mukhopadhyay. A grammar inference approach for predicting kinase specific phosphorylation sites. *PLoS ONE*, 10(4):e0122294, 2015.

[12] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior-oriented parallelization. In *PLDI*, 2007.

[13] L. Falquet, M. Pagni, P. Bucher, N. Hulo, C. J. Sigrist, K. Hofmann, and A. Bairoch. The prosite database, its status in 2002. *Nucleic acids research*, 30(1):235–238, 2002.

[14] M. Feng, R. Gupta, and Y. Hu. Spicec: Scalable parallelism via implicit copying and explicit commit. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, 2011.

[15] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. D. Pietro. An improved DFA for fast regular expression matching. *Computer Communication Review*, 38(5):29–40, 2008.

[16] F. Franchetti and M. Puschel. A SIMD vectorizing compiler for digital signal processing algorithms. In *IPDPS*, pages 7–pp, 2002.

[17] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[18] C. García, R. Lario, M. Prieto, L. Piñuel, and F. Tirado. Vectorization of multigrid codes using SIMD ISA extensions. In *IPDPS*, pages 8–pp, 2003.

[19] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, pages 173–189, 2003.

[20] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, 2008.

[21] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 1993.

[22] K. Hou, H. Wang, and W.-c. Feng. Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS)*, pages 383–392. ACM, 2015.

[23] P. G. Howard and J. S. Vitte. Parallel lossless image compression using huffman and arithmetic coding. In *Data Compression Conference, 1992. DCC'92.*, pages 299–308. IEEE, 1992.

[24] D. A. Huffman. Notes on information-lossless finite-state automata. *Il Nuovo Cimento (1955-1965)*, 13:397–405, 1959.

[25] N. Ide, M. Hirano, Y. Endo, S. Yoshioka, H. Murakami, A. Kunimatsu, T. Sato, T. Kamei, T. Okada, and M. Suzuoki. 2.44-GFLOPS 300-MHz floating-point vector-processing unit for high-performance 3D graphics computing. volume 35, pages 1025–1033, 2000.

[26] Y. Jo, M. Goldfarb, and M. Kulkarni. Automatic vectorization of tree traversals. In *PACT*, pages 363–374, 2013.

[27] C. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the web browser. In *HotPar*, 2009.

[28] B. Kaplan. Speculative parsing path. `http://bugzilla.mozilla.org`.

[29] C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding. Safe parallel programming using dynamic dependence hints. In *ACM SIGPLAN Notices*, volume 46, pages 243–258. ACM, 2011.

[30] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *ACM SIGMOD*

International Conference on Management of data, pages 339–350, 2010.

[31] S. Klein and Y. Wiseman. Parallel huffman decoding with applications to jpeg files. *Jounal of Computing*, 46(5), 2003.

[32] S. T. Klein and Y. Wiseman. Parallel huffman decoding with applications to JPEG files. *Comput. J.*, 46(5):487–497, 2003.

[33] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 2007.

[34] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. S. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Pisa, Italy, September 11-15, 2006*, pages 339–350, 2006.

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

[36] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[37] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *ACM conference on International conference on supercomputing*, pages 273–282, 2013.

[38] D. Luchaup, R. Smith, C. Estan, and S. Jha. Multi-byte regular expression matching with speculation. In *RAID*, 2009.

[39] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OoarticloOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 309–328, 2014.

[40] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *ASPLOS '14: Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*. ACM Press, 2014.

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

[42] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, volume 41, pages 132–143. ACM, 2006.

[43] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *PACT*, pages 2–11, 2008.

[44] Y. Pan, Y. Zhang, K. Chiu, and W. Lu. Parallel XML parsing using meta-dfas. In *Third International Conference on e-Science and Grid Computing, e-Science 2007, 10-13 December 2007, Bangalore, India*, pages 237–244, 2007.

[45] A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *Modeling and verification of parallel processes*, pages 196–205. Springer, 2001.

[46] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[47] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2010.

[48] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, 2010.

[49] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley. Efficient bdd algorithms for fsm synthesis and verification. *IWLS95, Lake Tahoe, CA*, 253:254, 1995.

[50] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, pages 218–232, 1995.

[51] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD parallelization of applications that traverse irregular data structures. In *CGO*, pages 1–10, 2013.

[52] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[53] I. Roy and S. Aluru. Finding motifs in biological sequences using the micron automata processor. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 415–424, 2014.

[54] P. Shankar, A. Dasgupta, K. Deshmukh, and B. S. Rajan. On viewing block codes as finite automata. *Theoretical Computer Science*, 290(3):1775–1797, 2003.

[55] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[56] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the International Symposium on Microarchitecture*, 2008.

[57] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT*, pages 327–337, 2009.

[58] R. van Engelen. Constructing finite state automata for high-performance XML web services. In *Proceedings of*

the International Conference on Internet Computing, IC '04, Volume 2 & Proceedings of the International Symposium on Web Services & Applications, ISWS '04, Las Vegas, Nevada, USA, June 21-24, 2004, pages 975–981, 2004.

[59] Z. G. Wang, J. Elbaz, F. Remacle, R. D. Levine, and I. Willner. All-DNA finite-state automata with finite memory. *Proc. Natl. Acad. Sci. U.S.A.*, 107(51):21996–22001, Dec 2010.

[60] 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 *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 93–102, 2006.

[61] Z. Zhao, B. Wu, and X. Shen. Challenging the "embarrassingly sequential": Parallelizing finite state machine-based computations through principled speculation. In *ASPLOS '14: Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*. ACM Press, 2014.

[62] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong. Gpu-based nfa implementation for memory efficient high speed regular expression matching. In *PPoPP '12: Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–140, 2009.