UNIVERSITY OF CALIFORNIA
RIVERSIDE

FSM-Centric Speculative Parallelization for Scalable Data Processing

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Junqiao Qiu

September 2020

Dissertation Committee:

Dr. Zhijia Zhao, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Rajiv Gupta
Dr. Zizhong Chen

The Dissertation of Junqiao Qiu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First and foremost, I am extremely grateful to my PhD. advisor, Dr. Zhijia Zhao, for his guidance, patience, and support. I would like to thank Dr. Zhijia Zhao for leading me into the research in the field of Parallel Computing. Without his great support and guidance, it is impossible for me to get the Ph.D. degree in Computer Science in US.

I would like to thank Dr. Nael Abu-Ghazaleh, Dr. Rajiv Gupta, and Dr. Zizhong Chen for serving on my Ph.D dissertation committee. They provided me insightful comments and valuable suggestions to help me finish this dissertation and the defense.

I would like to thank my lab-mates and friends for helping me in various ways: Lin Jiang, Umar Farooq, Amir Hossein Nodehi Sabet, Xiaolin Jiang, Chengshuo Xu, Xiaofan Sun, Xizhe Yin. I am grateful for the joyful chats we had, encouraging words you gave, and the time we fought hard together.

I would like to thank a special friend, Wenmei Jiang, for encouraging me not to give up even though the life might be so difficult. I really hope you can be happy forever.

Words cannot express my gratitude to my family. Without their unconditional love and unflagging support, I could not have gone this far.

Publication Acknowledgement

I acknowledge part of this dissertation has been published previously in the following conferences.

- Chapter 2 was previously published [99] in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT). 2016*

- Chapter 3 was published [100] in *Proceedings of the International Conference on Supercomputing (ICS). 2017.*

- Chapter 5 was published [98] in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2020.*

To my parents, and my lost child.

ABSTRACT OF THE DISSERTATION

FSM-Centric Speculative Parallelization for Scalable Data Processing

by

Junqiao Qiu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2020
Dr. Zhijia Zhao, Chairperson

Parallelism is key for designing and implementing high-performance data analytics on modern processors. However, many data processing routines cannot be executed in parallel, due to the sequential nature of their underlying computation models. This dissertation focuses on an important class of sequential data processing routines that are driven by or can be modeled as finite-state machines (FSMs). It proposes a series of speculation-based parallelization and modeling techniques to improve the parallelism and scalability of FSM-based computations. Moreover, it successfully applies the FSM-based speculative parallelization to non-FSM computations, significantly expanding the applicability of the proposed techniques.

More specifically, we first introduce multi-level speculation by integrating the instruction-level and SIMD-level parallelism into the existing multicore-level speculative parallelization. We then systematically model the scalability of speculative FSM parallelization and point out its limitations. To address them, we design two novel optimizations: path fusion and higher-order speculation, which together bring the scalability to another level for

FSMs that are conventionally hard to parallelize effectively. Finally, we demonstrate that, with rigorous static analysis, we can precisely model bitstream computations with FSMs, hence solve their parallelization with existing FSM parallelization techniques.

Contents

List of Figures

List of Tables

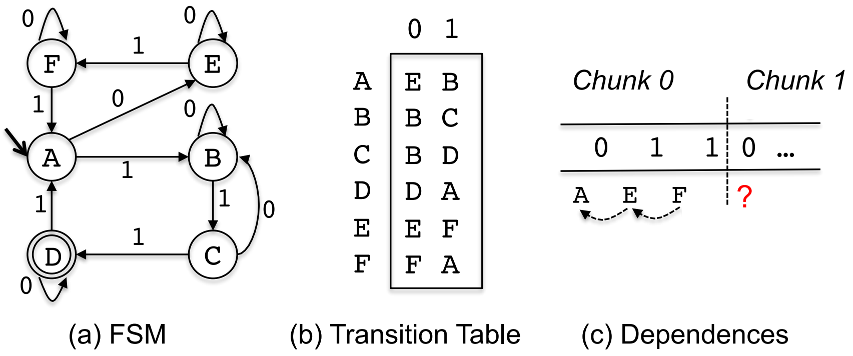
Chapter 1

Introduction

As a basic computation model, finite-state machine (FSM) embodies a variety of important applications, ranging from intrusion detection [134, 71, 114, 9] and data decoding [67, 113] to motif searching [109, 29], rule mining [129], and textual data analytics [92, 42, 30]. For its fundamental role in many performance-critical applications, it is anticipated that emerging architectures will feature hardware supports for FSM computations, including accelerators such as automata processor [32].

Unfortunately, the execution of an FSM is known to be "embarrassingly sequential" [8, 138] due to the inherent dependences among state transitions – in each state transition, the current state always depends on the prior state [1]. Figure 1.1 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.

[1] Here, it refers to a deterministic FSM; Non-deterministic FSMs can be converted to deterministic ones via a classic conversion algorithm [2].



	0	1
A	E	B
B	B	C
C	B	D
D	D	A
E	E	F
F	F	A

(a) FSM (b) Transition Table (c) Dependences

Figure 1.1: An FSM Example

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 dependences form a tight dependence chain, inherently preventing any parallelism from being exposed. Assume the input to the FSM is partitioned into two chunks, as shown in Figure 1.1, each of which is then assigned to one thread to process. Due to the dependences among state transitions, the starting state for the second thread would be unknown, until the first chunk has been processed – the ending state of the first chunk is the starting state of the second chunk.

State of The Art. To address the inherent state dependences in FSM computations, existing work [50, 62, 138, 83, 137, 99, 100] has followed two basic parallelization schemes: (1) speculative parallelization [138] and (2) parallel prefix-sum [83] . The former breaks the dependences by predicting the states at the beginning of a divided input chunk. In the cases that some predictions fail, reprocessing would be needed to ensure correctness. Instead of making prediction, the latter scheme needs no prediction at the beginning of an input chunk. Instead, it enumerates all the possible states, which always cover the actual one. With either of the two ways, different input chunks can be processed in parallel.

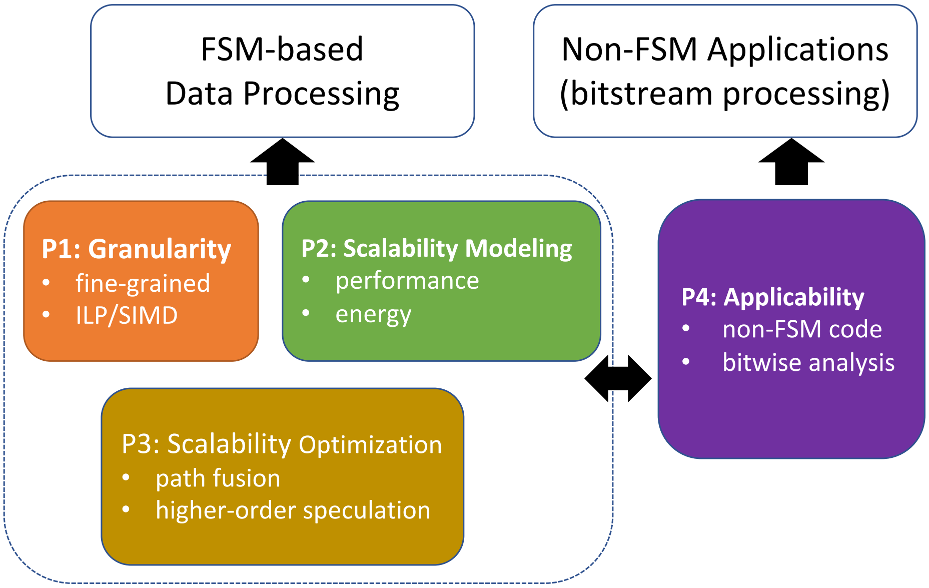


Figure 1.2: Scalable FSM-Driven Data Processing

1.1 Dissertation Overview

The benefits of existing FSM parallelization solutions are restricted in several dimensions. First, the speculative parallelism is limited to the coarse granularity – they are unaware of the more fine-grained instruction level and vector level parallelism prevalent on modern CPUs. Second, the existing FSM speculation frameworks are scalability insensitive – they aggressively employ all the available cores on the CPUs, which might not only waste energy, but also suffer from suboptimal performance. Third, despite promising results of speculative parallelization and parallel prefix-sum, the efficiency of both parallelization schemes highly depends on the properties of the FSM. For FSMs exhibiting limited state convergence, the former is bottlenecked by the serial reprocessing among the misspeculation cases, while the latter suffers from the overhead of maintaining multiple execution paths. Last but not least, the existing solutions are designed for computations that explicitly use FSMs as the underlying models, limiting their applicability. Figure 1.2 summarizes the above limitations as the granularity, scalability, and applicability issues.

In this dissertation, we tackle all issues discussed above with practical parallel programming techniques, systematic performance modeling, and rigorous program analysis and optimizations.

1.2 Contributions

1.2.1 Fine-Grained FSM Parallelization

First, this dissertation presents *MicroSpec*, a set of parallelization techniques that expose fine-grained speculative parallelism to FSM computations. 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.

1.2.2 Scalability-Sensitive Speculative Parallelization

In this dissertation, we conduct a systematic scalability analysis for speculative FSM parallelization. Unlike many other parallelizations which can be modeled by the classic Amdahl's law or its simple extensions, the scalability of speculative FSM parallelization is challenging to analyze due to the non-deterministic nature of speculation and the cost variations of misspeculation. This work makes the following four-fold contributions.

- This work, for the first time, points out the suboptimality of the state-of-the-art speculative FSM parallelization when moving to larger-scale parallel platforms.
- It provides a series of rigorous scalability models, including a *sample-based conditional regression* technique, that enables the characterization of complex scaling behaviors in speculative FSM parallelization.
- To facilitate the use of the proposed models, this work designs `S3`, a scalability-sensitive speculative parallelization framework that can automatically characterize the FSM properties and guide the speculative parallelization with the optimal configuration at runtime.
- It evaluates `S3` on both many-core processor (i.e., the latest Xeon Phi processor) and

multi-socket multi-core architectures, and demonstrates large ratio of performance improvements and significant energy reduction.

1.2.3 Scalable FSM Parallelization

In this dissertation, we introduce two techniques: *path fusion* and *higher-order speculation*, to address the scalability limitation in each basic FSM parallelization scheme, respectively. This work makes a three-fold contribution.

- First, it introduces the *path fusion* technique to address the high cost of maintaining multiple execution paths in state enumeration-based FSM parallelization.

- Second, it proposes *higher-order speculation* for FSM parallelization and designs an *iterative speculation* scheme to address the serial validation bottleneck in the existing speculation-based FSM parallelization.

- Finally, this work designs a heuristic-based method to select the better parallelization scheme for the given FSM and confirms the effectiveness of the proposed techniques with a systematic evaluation.

1.2.4 Non-FSM Applications Parallelization

In the dissertation, we look beyond the FSM computations and find an important class of non-FSM computations, that is, bitstream processing, that may also benefit from speculative FSM parallelization. This work makes the following major contributions to bitstream processing.

- First, it offers a new perspective to the sequential bitstream processing, bringing FSM-based dependence modeling to bitstream programs.

- Second, it introduces a static analysis to rigorously find out the dependent bits in bitstream programs.

- Third, it adopts FSM speculation to bitstream processing with customized mis-speculation handling.

- Finally, it integrates ideas from static analysis, automata theory, and speculation, into *principled bitwise speculation*, and confirms its effectiveness in accelerating real-world bitstream applications.

Chapter 2

Fine-Grained Speculative Parallelization for FSM Computations

2.1 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 chapter focuses on exposing effective fine-grained parallelism to Finite State Machine (FSM)-based computations. Even though speculative parallelization and parallel prefix-sum developed in recent years break the barrier of making FSM computations run parallel, none of them has released the full potential 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 2.3.2).

To address these concerns, this chapter 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 chapter 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.

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

2.2 Background and Problem

2.2.1 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 chapter focus on deterministic ones for their better efficiency [2].

Parallelizing FSM computations are extremely difficult due to their inherent sequential characteristics — dependences exist between every consecutive state transitions, as illustrated by Figure 1.1 (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 dependences 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 [138] followed the first direction and proposed a coarse-grained speculative parallelization approach to circumvent the dependences. Instead of speculation, Todd and others [83]'s approach enumerates all the possible cases to leverage classic parallel prefix-sum. They implemented with both coarse-grained

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

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 chapter 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 2.4.2).

2.2.2 Coarse-Grained Speculative Parallelization

As this chapter 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

```
$\Pi$ = coarse_grained_partition($N_{core}$); /* Step 1 */
for thread $1 \cdots N_{core}$ do
    $S_{start}(i)$ = predict (suffix of $\Pi(i-1)$); /* Step 2 */
    process($\Pi(i)$, $S_{start}(i)$); /* Step 3 */
thread_join();
for partition $1 \cdots N_{core}$ do /* Step 4 */
    if validate($S_{start}(i)$) == FALSE then
        reprocess($\Pi(i)$);
```

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 [138]). 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 [138], 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 2.1.

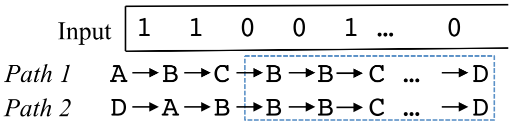


Figure 2.1: Example of State Convergence.

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* [138, 83].

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 2.3.2.

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 chapter. Then, we compare their effectiveness with a rigorous analysis, which in turn guides the design of FSM parallelization techniques.

2.3.1 Three Dimensions

The only fine-grained parallelism that has been seen in prior work comes from *associative parallelism* [73, 83]. 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 $c_1c_2\cdots c_L$ 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 [73].

In practice, as described in [83], 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 [83] (see Section 2.2.2).

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 [64]. 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.

2.3.2 Efficiency Analysis

We next analyze the efficiency of 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 [138, 83] 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 2.1, 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.*

$$M_L = \begin{bmatrix} L^I(s_1, s_1) & L^I(s_1, s_2) & \dots & L^I(s_1, s_n) \\ L^I(s_2, s_1) & L^I(s_2, s_2) & \dots & L^I(s_2, s_n) \\ \vdots & \vdots & \ddots & \vdots \\ L^I(s_n, s_1) & L^I(s_n, s_2) & \dots & L^I(s_n, s_n) \end{bmatrix} \tag{2.1}$$

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

$$\|Path(s_i)\| = \min\{L^I(s_i, S - \{s_i\}), \|I\|\} \tag{2.2}$$

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 3 *Given an input I, the $ECPL$ of P1 is*

$$ECPL(P1) = \|I\| \tag{2.3}$$

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 $live(S, j)$.

$$DoP(P1) = \min\{live(S, j), PU\}, \text{ where } 1 \leq j \leq \|I\| \tag{2.4}$$

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

$$\|redo\| = min\{L^I(s_i, s^*)|s_i \in S_K, s^* \text{ is the true state}\} \tag{2.5}$$

Assuming that the reprocessing in P2 runs sequentially, we have Lemma 4 holds for P2.

Lemma 4 *Given an input I, the $ECPL$ of P2 is*

$$ECPL(P2) = \|I\| + (1 - P_k) \cdot \|redo\| \tag{2.6}$$

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.

$$DoP(P2) = \begin{cases} \min\{live(S_k, j), PU\} & 1 \leq j \leq \|I\| \\ 1 & \text{redo} \end{cases} \tag{2.7}$$

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

$$\left\|\overline{redo}\right\| = \sum_{i=2}^{PU}(1 - p(\hat{s}_i)) \cdot \|redo(\hat{s}_i)\| \tag{2.8}$$

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 5 for P3.

Lemma 5 *Given an input I, the $ECPL$ of P3 is*

$$ECPL(P3) = \|I\| / PU + \left\|\overline{redo}\right\| \tag{2.9}$$

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.

$$DoP(P3) = \begin{cases} PU & 1 \leq j \leq \|I\| \\ 1 & \text{redo} \end{cases} \tag{2.10}$$

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 chapter, 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 2.5). Second, it guarantees high hardware utilization by keeping $DoP(P3)$ mostly as high as PU.

2.4 MicroSpec

Guided by the analysis in Section 2.3, 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.

2.4.1 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 chapter, we simply choose a relatively straightforward prediction, named *simple lookback*, which has been used by prior work [138, 12].

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.

2.4.2 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 2.2, 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

```
$\pi$ = fine_grained_partition($W$);
$S$ = predictInitStates($\pi$);
**for** (i=0; $i < L_{seg}/W$, i++) **do**
    $I$ = readInputVec($i$);
    $F = S \times N_{sym} + I$;
    $S$ = gather($T$, $F$);
**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 dependences 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 dependences across state transitions, unrolling does not provide any benefits. As shown in Figure 2.2, the performance of after unrolling is almost the same as the default version. This is mainly because the state transition dependences turn into instruction dependences, 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 dependences among the unrolled instructions. We refer to it as *Speculative Unrolling*, illustrated by Algorithm 3.

Algorithm 3 Speculative Unrolling

```
$\pi$ = fine_grained_partition($R$);
$s[0 \cdots R-1]$ = predictInitStates($\pi$);
B = $L_{seg}/R$;
for (i=0; $i < B$, i++) do
    c[0] = readInput($i$);
    s[0] = T[s[0]][c[0]];
    c[1] = readInput($i + B$);
    s[1] = T[s[1]][c[0]];
    $\cdots$ $\cdots$
    c[R-1] = readInput($i + B * (R-1)$);
    s[R-1] = T[s[R-1]][c[R-1]];
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

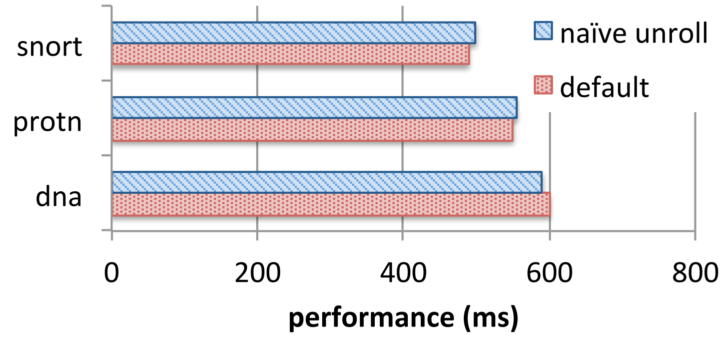


Figure 2.2: Performance of Naive Unrolling

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 dependences, 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 2.5, 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 2.5.

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.

Algorithm 4 Speculative SIMD Gather+

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

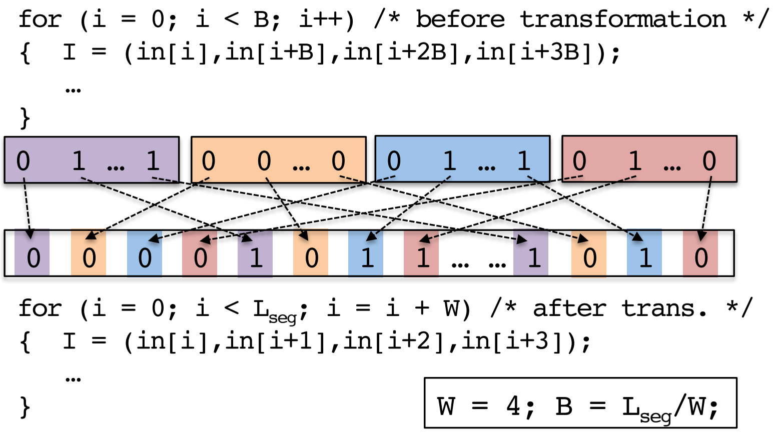
2.4.3 Optimization

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.

Algorithm 5 Speculative Unrolling+

```
$\pi$ = fine_grained_partition($W \times R$);
$S[0 \ldots R-1]$ = predictInitStates($\pi$);
$B = L_{seg}/R$ ;
**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**
```

To overcome this, we propose a *speculation-oriented data transformation* that relayouts the data according to the accessing pattern in multi-level speculation scheme to minimize the memory accessing delays. Basically, it transforms the big stride-based accessing to simple sequential accessing. It does this by moving each group of stride-based accessed



```
for (i = 0; i < B; i++) /* before transformation */
{  I = (in[i],in[i+B],in[i+2B],in[i+3B]);
   …
}
```

```
for (i = 0; i < Lseg; i = i + W) /* after trans. */
{  I = (in[i],in[i+1],in[i+2],in[i+3]);
   …
}                                  W = 4; B = Lseg/W;
```

Figure 2.3: Spec.-Oriented Data Transformation

data next to each other, as shown in Figure 2.3. 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.

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

2.4.4 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.

2.5 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.

2.5.1 Methodology

The evaluation of *MicroSpec* includes all four speculation-based fine-grained parallelization techniques as well as the speculation-oriented data transformation. Table 2.1 summarizes them and lists their abbreviation used in the evaluation.

Table 2.1: *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 [138] and parallel prefix-sum [83], 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.

2.5.2 Benchmarks

The benchmarks are selected to cover a wide range of FSM applications with different levels of complexities. We first 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 2.2 lists three protein patterns randomly selected from a widely used protein database PROSITE [34]. In Table 2.2, $[\cdot]$ means alternative symbols and 'x' means any symbol; while $(\cdot)$ is the number of repetition. 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.

Table 2.2: Protein Motifs

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]`

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 [108] 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 2.3 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`.

Table 2.3: Snort Rules.

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

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 chapter targets the decoding phase, as the encoding phase is embarrassingly parallel [53].

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 $((.+, .+n.)\{4\}|(.+n,)\{4\}|(.+n.)\{4\})\{3\}$ and $(. * l. * i. * k. * e)\{6\} \mid (. * .a. * .p. * .p. * .l. * .e)\{5\}$.

Table 2.4 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 [83].

Table 2.4: 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

2.5.3 Results

Unrolling Factor. Since the selection of the unrolling factor R may affect the performance of *MicroSpec*, we first discuss it. Table 2.5 shows the execution time of *dna4* on a small testing input using different unrolling factor values. The results answer the question in Section 2.4.2 – 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 2.4 shows the performance results for *motif searching* benchmark group. Overall, the performance of four speculation-based methods in *MicroSpec*

Table 2.5: 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

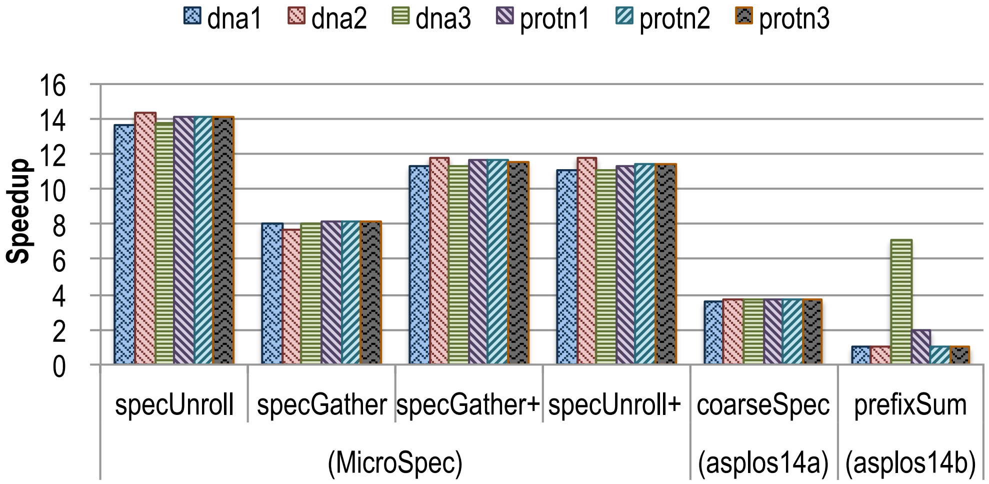


Figure 2.4: Speedups for Biological Benchmarks

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 2.5 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 2.6 shows the performance results of the last benchmark group, which are mixed with Huffman decoding (*huff*) and some hard-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

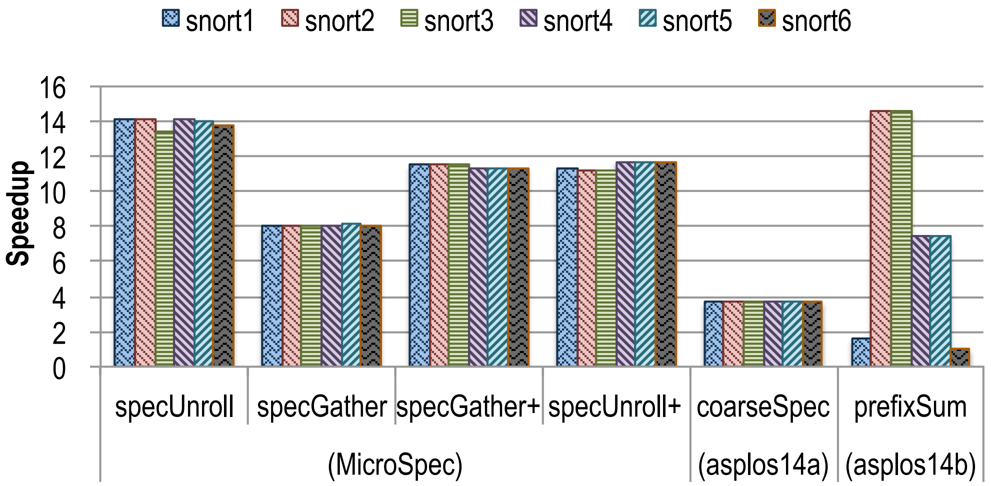


Figure 2.5: Speedups for Snort Benchmarks

`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

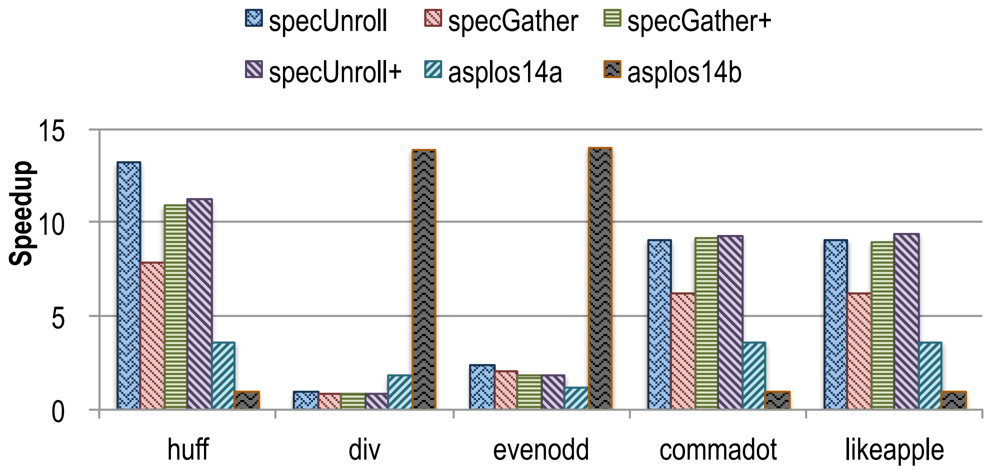


Figure 2.6: Speedups for Mixed FSM Benchmarks

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 2.6 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 motif queries to the same DNA or protein sequence database. Figure 2.7 shows the improvements of *specTrans* optimization. On average, it brings about 8.5% extra speedup.

Comparison on Different Architectures. Finally, we also tested *MicroSpec* on an architecture without SIMD gather. In this case, only S2, *specUnroll* is experimented. Figure **??**

Table 2.6: 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

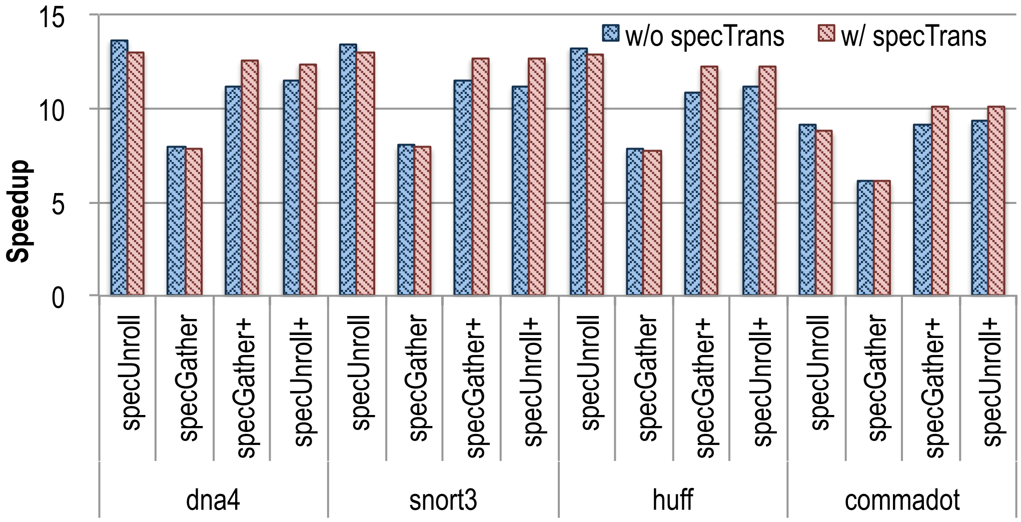


Figure 2.7: Performance Improvements of *specTrans*

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.

2.6 Summary

This chapter 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.

Chapter 3

Scalability-Sensitive Speculative Parallelization for FSM Computations

3.1 Introduction

Scalability is fundamental to the high-performance applications. An accurate scalability analysis not only helps realize the optimal performance, but also avoid unnecessary use of additional computing resources. In this chapter, we aim to provide an accurate scalability analysis for the speculative parallelization of finite state machine (FSM) computations.

For the fundamental role of FSMs in many performance-critical applications, it is anticipated that emerging architectures will feature hardware supports for FSM computations, including accelerators such as automata processor [32]. However, due to the tight dependences among state transitions, FSM computations are extremely difficult to parallelize. As shown in the code snippet below, at each transition, the current state `state` not only depends on the input symbol `c` but also the prior state `prior`. Such dependences essentially form a dependence chain, inherently prevent any FSM computations from running in parallel.

```
prior = init;
while c!=EOF do {
  c = read();
  state = trans(prior, c); // dependence
  ϕ(state, c); // action at a state
  prior = state;
}
```

State of the Art. To overcome the dependences, existing methods often rely on speculative parallelization [97, 138, 83]. Basically, they first partition the input sequence evenly into N_{core} chunks where N_{core} is the number of available cores, then process the chunks in parallel, each with a predicted starting state, except the first chunk. In the case where a prediction fails, they need to reprocess the wrong part to ensure the correctness (more details in Section 3.2). This strategy has shown promise on small-scale multicore processors (up to eight cores). However, it is still poorly understood how well it can scale to larger

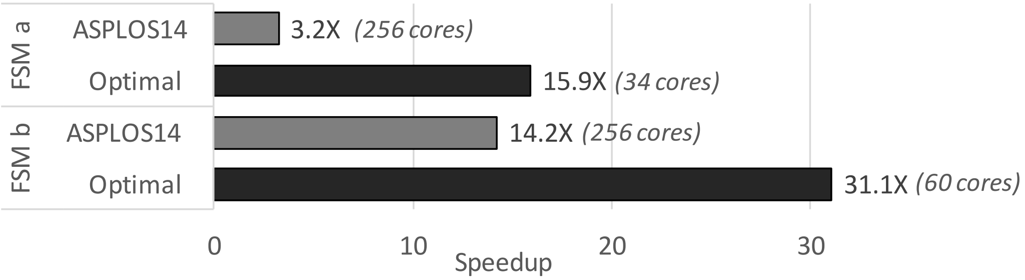


Figure 3.1: Suboptimal Speedup of Existing Methods [138] on Larger-scale Platforms (256-core Xeon Phi).

parallel platforms with tens of or even hundreds of processing cores [1] (such as Xeon Phi processors).

In particular, *is it always the best practice to use all available cores (i.e., N_{core}) to achieve the best performance? If not, what is the optimal number of cores to employ to maximize the benefits of speculative FSM parallelization?*

Without answering these questions, existing methods may not only suffer from suboptimal performance, but also waste precious computing resources that would be otherwise used for other computations. As illustrated by Figure 3.1, when executed on a Xeon Phi processor with 256 logical cores, existing methods [138, 137] result in suboptimal speedup on two FSM benchmarks, up to nearly 5X performance degradation, comparing to the optimal ones. It is also important to note that the optimal number of cores varies across different FSMs.

Unlike prior work that focus on either designing sophisticated speculation techniques [138] or reducing the cost of profiling [137], this chapter aims to achieve the optimal

[1] By default, this chapter refers to logical cores as cores, unless noted otherwise.

performance gain for speculative parallelization of FSM computations by offering an accurate scalability analysis.

However, accurately analyzing the scalability for speculative FSM parallelization is challenging for four-fold reasons. First, by nature, speculative parallelization is non-deterministic. Its overall performance highly depends on the accuracy of the speculation. Second, when a speculation fails (*misspeculation*), it is required to reprocess the incorrect part. But the processing cost may vary across different chunks, depending on the convergence. Consequently, the total cost of misspeculation correlates with the number of cores non- linearly, making existing models fail to capture its scalability. Finally, the actual scalabilities of FSM computations are also constrained by the machines where they are executed via resource contention and relative execution speed.

To address the above complexities, this work introduces a series of scalability models for speculative FSM parallelization. The models integrate a probabilistic analysis to capture the non-deterministic behaviors of speculation and an offline *sample-based conditional regression* (SCR) technique to characterize the cost variation of misspeculation. Unlike existing FSM characterization [138, 137] that requires to profile the convergence property for every pair of states, SCR only profiles state pairs that are more likely to appear in the actual speculative execution. Based on the probabilistic models and SCR, this work designs both architecture-independent scalability analysis and architecture-aware scalability analysis. The former analyzes the scalability solely based on the design of speculative parallelization and the properties of an FSM. It guides the designers to tune speculative parallelization scheme and helps developers compare the scalabilities of various FSMs. In comparison, the latter further characterizes the architecture factors that may affect the actual scalability, making the scalability analysis practical in real- world computing environments.

To effectively leverage the above scalability analyses, this chapter develops `S3` – a scalability-sensitive speculative parallelization framework for FSM computations. At high level, `S3` works in three steps: (1) it first characterizes the FSM's properties and measures the architectural factors of the machine; (2) With the measurements, `S3` next automatically reasons about the scalability and infers the optimal number of cores n^* to use; (3) Finally, it feeds n^* into the speculative parallelization to maximize its performance gain.

Experiments on a set of real-world FSM benchmarks demonstrate the accuracies of the proposed models and show that `S3` can boost the performance of existing techniques up to 5X, with significant energy savings in most cases (up to 77%).

3.2 Motivation

In this section, we first illustrate the basic approach of speculative parallelization used by existing work for FSM computations, then point out the suboptimality of performance in existing solutions due to their unawareness of scalability, hence the necessity to enable scalability-sensitive speculative parallelization.

Speculative Parallelization. To address the tight dependence among state transitions in FSM computations, existing solutions often rely on speculative parallelization techniques, which is based on a predict-validate-reprocess strategy. Next, we describe the high-level ideas of speculative FSM parallelization, which consists of three major phases.

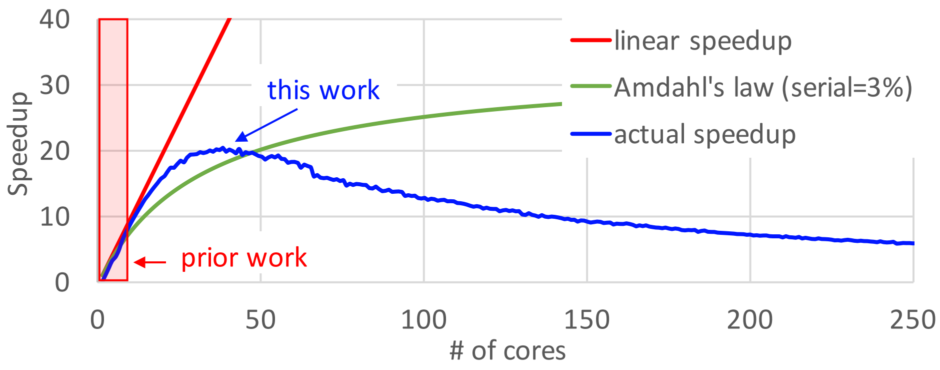


Figure 3.2: Projected Scalability v.s. Actual Scalability

1. **Partition**: Divide the input sequence into equal-sized chunks according to the total number of CPU cores N_{core}.

2. **Predict & Process**: For each chunk i, predict its starting state s^i_{pred} and assign a thread to process [2].

3. **Validate & Reprocess**: Once every thread has finished its chunk, check if the predicted state equals to the true state (i.e., $s^i_{true} = s^i_{pred}$) one by one. If a prediction fails, reprocess its corresponding chunk before validating the next starting state. Note that the reprocessing may stop earlier when states s^i_{true} and s^i_{pred} both transition to the same state (known as *state convergence*, see Section 3.4.1).

According to the first phase, the basic speculative parallelization approach assumes that it can scale up to the total number of CPU cores N_{core}. While this might be true for small-scale multicore processors (e.g., quad/oct-core processors), but may not hold for larger-scale platforms with tens of or hundreds of CPU cores.

[2] More details about the design of the prediction can be found in prior work [138, 137].

Performance Suboptimality. Figure 3.2 shows the speedup curve for an FSM benchmark on a Xeon Phi machine with 256 cores. As the blue line shows, the actual speedup increases linearly at the beginning before reaching about 10 cores, which is confirmed by prior work [138, 137]. However, the increase becomes non-linear thereafter and even starts dropping after about 30 cores. Finally, the speedup drops to merely 5.7X when all 256 cores are used. This result clearly demonstrates that using all available cores may not lead to the optimal performance.

On the other hand, unlike many parallel applications, the speedup curve of speculative FSM parallelization is difficult to model using the classic Amdahl's law and its simple extensions. As Figure 3.2 shows, the speedup curve predicted by Amdahl's law, with 3% of serial execution (green line), follows an obviously different trend comparing with actual speedup curve.

The principle reason to such a discrepancy is due to the inherent complexities of speculative FSM parallelization. First, during a speculative FSM execution, not only the parallel part (i.e., Phase 2) depends on the number of cores, but also the sequential execution part (i.e., Phase 3). In comparison, the serial part in Amdahl's law, by default, is assumed to be a constant. Furthermore, the relation between the sequential execution performance and the number of cores in parallel part is non-linear, due to the variation of state convergence. The two complexities make existing scalability models fail to faithfully capture the scalability of this advanced parallelization technique.

To address the challenges and seek for the optimal performance, we propose `S3`, a speculative parallelization framework that can automatically characterize the scalability of a given FSM and calculates the best configuration to maximize the performance. We next give an overview of S3 before presenting its details.

3.3 Overview

At high level, S3 includes three layers. From bottom to top, they are *characterization*, *modeling* and *guidance*, as shown in Figure 3.3. We next briefly present each of the three layers in order.

Characterization. As both FSM computations and the underlying architecture can affect the scalability of speculative parallelization, but from completely different perspectives, it is natural to separate the characterization on two orthogonal dimensions: application dimension and architecture dimension. Symbolically, we refer to the characterization results on the application side and architecture side as *FSM.properties* and *Arch.properties*, respectively.

① On one side,FSMs exhibit dramatically different behaviors when executed speculatively. Some FSMs are easier to speculate while others may be much more challenging (such as div in [138]), depending on their transition structures and the characteristics of their input domains. Furthermore, when a misspeculation happens, the penalty not only varies across different FSMs, but also varies across different speculatively processed chunks of the same FSM, depending on how fast the predicted (wrong) state s_{pred} converges with the correct starting state s_{true}. The faster they converge, the less penalty the misspeculation incurs. Prior work [138, 137] introduce a couple of metrics, namely *state feasibility* and *expected convergence length*, to quantify some of the above characteristics. While being

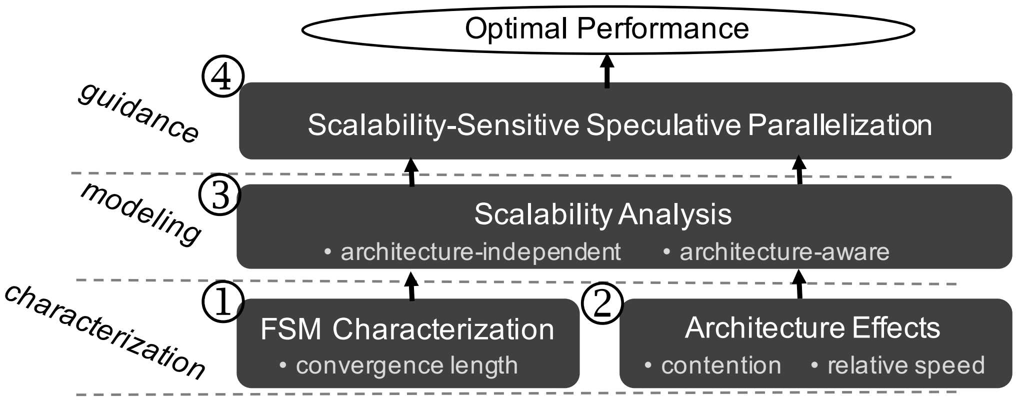


Figure 3.3: Overview of S3

useful to tune the design of the predictor, these metrics are inadequate to accurately model the details of the non-deterministic behaviors of speculation, such as the distribution of misspeculation penalty across different chunks.

Rather than computing the expected convergence length by averaging the samples, S3 maintains a short list of raw convergence length samples (typically < 150) for each state pair. The list of samples encapsulates not only the average convergence length, but also its distribution, which is the key to accurately model the variation of misspeculation penalty (Section 3.4.2). To reduce the overhead of characterization, unlike existing methods which require to profile the convergence property for every pair of states, S3 only profiles the state pairs that are more likely to appear in actual speculative executions. We will elaborate FSM characterization and its uses in Section 3.4.

② On the other side, the characteristics of the architecture also directly affect the scalability of speculative FSM parallelization in various ways, depending on the specific design of the architecture. In this chapter, we focus on two main factors that play critical roles in the scalability analysis: *resource contention* and *relative execution speed*. Resource contention happens when different threads share the same computing resources, such as last level cache (LLC) and memory bandwidth. Depending on the design of the architecture and the number of concurrent threads, such contention could vary significantly. Note that resource contention only happens in the parallel phase of speculative FSM execution. When moving into the reprocessing phase, only a single thread is left due to dependences, the contention hence reduces to zero. However, due to the tracking of state convergence, the execution speed in the reprocessing phase might be slightly slower than the parallel phase. This difference directly influences the scalability, but may vary across architectures. Therefore, it is necessary to capture the relative execution speed between the two phases, in order to precisely quantify the scalability. We will present architecture characterizations in Section 3.5.1.

Scalability Modeling. In this chapter, the scalability is defined as the capability of speculative parallelization to scale up to larger amount of computing units (i.e., CPU cores). In particular, given an FSM with a fixed-size input, the scalability concerns how the execution time varies with the number of CPU cores used [3].

③ With the characterization results, `S3` can automatically reason about the scalability using a series of scalability models that are derived based on the design of speculative FSM parallelization.

In specific, the models define the speedup of speculative FSM parallelization S as a non-linear function of the number of cores employed n, along with other parameters, such

[3] This is commonly referred to as *strong scaling*.

as the properties of FSM computations $FSM.properties$ and the architecture properties $Arch.properties$ [4]. Depending on if $Arch.properties$ is considered, the models fall into two types:

architecture-independent scalability model:

$$S = f(n,\, FSM.properties) \tag{3.1}$$

architecture-aware scalability model:

$$S = f'(n,\, FSM.properties,\, Arch.properties) \tag{3.2}$$

where n is the number of cores used in speculative execution, $FSM.properties$ represents convergence properties of the given FSM, and $Arch.properties$ contains architecture characteristics such as resource contention among threads and relative execution speed of different FSM operations. The architecture-independent models can be used to compare the scalabilities of different FSMs and guide the design of speculative parallelization; The architecture-aware models provide more accurate scalability analysis results that are customized for a specific architecture.

At high level, the modeling breaks down the entire speculative FSM execution time T_{spec} into two parts: the *parallel processing time* T_{para} and the *sequential reprocessing time* T_{repr}. Let T_{seq} be the sequential execution time, then the speedup of speculative parallelization can be defined as follows:

$$S = \frac{T_{seq}}{T_{spec}} = \frac{T_{seq}}{T_{para} + T_{repr}} \tag{3.3}$$

[4] Note that some architecture properties, such as resource contention, also depend on the number of cores used n.

Unlike the classic Amdahl's law that assumes a constant ratio for the sequential part, in Equation 3.3, both the parallel part T_{para} and the sequential part T_{seq} primarily depend on the number of cores n. Moreover, the relation between T_{repr} and n follows a non-linear pattern, making standard scalability models fail to faithfully capture its scalability. We address the challenges with a novel *sample-based conditional regression* (SCR) technique. Different from traditional regression models, SCR conditionally accept convergence length samples (i.e., *FSM.properties*) based on the parameters of speculative parallelization. With such fine-grained customization, SCR can precisely model the above non-linear relation.

We will describe the basic scalability analysis in Section 3.4.2 and the integration of architecture properties in Section 3.5.2.

Scalability-Sensitive Speculative Parallelization. The goal of `S3` is to maximize the efficiency of speculative parallelization by reasoning about its scalability and discovering the optimal number of cores to use (i.e., n^*).

④ With the scalability models, this problem can be formalized as the following discrete optimization problem.

$$\begin{aligned} &\max S \\ &\text{s.t. } 1 \leq n \leq N_{core} \end{aligned} \tag{3.4}$$

where the number of cores used by speculative FSM parallelization is bounded by the total number of cores on the machine. When $n = 1$, the FSM execution becomes sequential.

To solve the optimization problem, depending on the models, `S3` either simply enumerates each configuration and chooses the one with the highest speedup, or directly computes the optimal configuration from a closed-form expression. By setting n^* in the speculative parallelization, S3 can maximize the benefits of the objective. We refer to this scheme as *scalability-sensitive speculative parallelization.*

In sum, the three layers closely depend on each other from top to bottom. They together enable a new speculative parallelization scheme for FSM computations on larger-scale parallel platforms. In the following, we elaborate architecture-independent scalability analysis and architecture-aware scalability analysis, respectively.

3.4 Architecture-Independent Scalability Analysis

This section presents the scalability analysis that does not assume any particular architecture, but solely based on the properties of the FSM computations and the design of speculative parallelization.

3.4.1 FSM Characterization

As a basic computation model, FSMs feature many properties. In this work, we focus on a type of characteristics that has a significant influence on the penalty of misspeculation – the *convergence length.*

As mentioned in Section 3.2, when a misspeculation happens, the speculative parallelization framework may not have to reprocess the whole chunk, thanks to the fact that the predicted (wrong) state s_{pred} may converge with the actual state s_{true}. The shorter it takes for them to converge, the less penalty of the misspeculation incurs. To effectively model such behaviors, we leverage the concept of state convergence length [138], defined as follows.

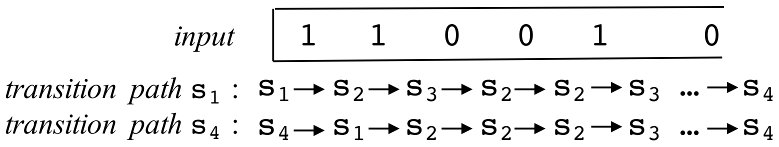

input
1 1 0 0 1 0
transition path s1 : s1 → s2 → s3 → s2 → s2 → s3 ... → s4
transition path s4 : s4 → s1 → s2 → s2 → s2 → s3 ... → s4

Figure 3.4: Example of State Convergence in FSM Transitions

Definition 6 *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 $L^I(s_i, s_j) = \infty$.*

In Figure 3.4, though starting from two different states, transition paths s_1 and s_4 reach the same state s_2 after consuming the third input symbol, hence the convergence length between s_1 and s_4 on this piece of input is $L^I(s_1, s_4) = 3$.

Given an FSM, its convergence length properties can be profiled either offline using a set of training inputs [138], or online using the testing inputs [137]. Note that prior work require to profile the average state convergence length for every pair of states, in order to guide the design of the starting state predictor [138, 137]. In comparison, S3 maintains a pool of raw convergence length samples for state pairs that are more likely to appear in actual speculative executions, in order to facilitate a high-precision scalability analysis, as explained in the next subsection.

3.4.2 Scalability Analysis

The goal of scalability analysis is to examine how speedup S varies as the number of cores used n changes. Based on the definition of S in Equation 3.3, this requires to model the ratio between sequential execution time and its corresponding speculative execution time. In architecture-independent scalability analysis, we use the number of state transitions to quantify the relative execution time, instead of the concrete execution time which may vary across different architectures. For example, given an input of length $\|I\|$, when processed sequentially, the execution time is $T_{seq} = \|I\|$.

For speculative execution, the total time T_{spec} mainly consists of the parallel processing time T_{para} and the sequential reprocessing time T_{repr} (Phases 2 and 3 in Section 3.2) [5].

$$T_{spec} = T_{para} + T_{repr} \tag{3.5}$$

In parallel processing phase, each thread first predicts the starting state, then processes its corresponding chunk with the predicted starting state. In general, there is a tradeoff between the prediction accuracy and prediction cost. However, after the design of predictor is fixed, the prediction cost becomes a constant (more details in [138, 137]). We use C_{pred} to represent the prediction cost and T_{proc} to represent the processing time of chunks. Since the input is evenly partitioned based on the number of cores n, we have T_{proc} $= \|I\| / n$. Hence, the parallel phase execution time T_{para} is

$$T_{para} = C_{pred} + T_{proc} = C_{pred} + \frac{\|I\|}{n} \tag{3.6}$$

[5] The partitioning time in Phase 1 is typically negligible.

Next, we analyze the execution time of the reprocessing phase, which is more challenging due to two complexities inherited in the design of speculative FSM parallelization.

Complexitiy I: Undeterministic behaviors of speculation. By its nature, speculation is non-deterministic. If a speculation succeeds, there would be no cost of reprocessing; otherwise, the speculation framework has to initiate reprocessing to correct the mistakenly processed parts. As only the latter case degrades the scalability, an effective scalability model needs to distinguish the two cases. However, since the speculation happens during the actual runs, such a distinction is as hard as the speculation itself.

Complexity II: Variation of reprocessing costs. To reduce the penalty of misspeculation, existing methods [138, 137] leverage the convergence property of FSMs (see Section 3.4.1) by tracking if the misspeculated state s_{pred} converges with the actual state s_{true}. Once they converge, the reprocessing can safely stop. On one hand, this design helps reduce the reprocessing costs of misspeculation. On the other hand, it also complicates the modeling of speculative parallelization, as the reprocessing costs for different chunks may vary significantly, depending on their convergence lengths.

In the following, we present an analytical model that address both complexities together, referred to as *sample-based conditional regression model.* Before introducing the model, we first formalize the total execution time of reprocessing.

In reprocessing phase, due to dependences, all chunks, except the first one, have to be validated and reprocessed sequentially. Therefore, the total reprocessing time T_{repr} is composed of the reprocessing time of each chunk T^i_{repr}, where $1 < i \leq n$. Let $L^i(s^i_{pred}, s^i_{true})$ be the convergence length between the misspeculated state s^i_{pred} and the actual state s^i_{true}

for chunk i. Then the total reprocessing time can be represented as

$$T_{repr} = \sum_{i=2}^{n} T_{repr}^{i} = \sum_{i=2}^{n} L^{i}(s_{pred}^{i}, s_{true}^{i}) \tag{3.7}$$

Two points worth to mention here. First, to address the above two complexities together, we unify the representations by referring to a successful speculation as a "mis-speculation" with reprocessing length of zero, that is,

$$L^{i}(s_{pred}^{i}, s_{true}^{i}) = 0, \text{ if } s_{pred}^{i} = s_{true}^{i} \tag{3.8}$$

Second, the reprocessing of a chunk cannot go beyond the size of the chunk, hence the following constraint holds:

$$L^{i}(s_{pred}^{i}, s_{true}^{i}) \leq \frac{\|I\|}{n} \tag{3.9}$$

Sample-based Conditional Regression. A key challenge in the scalability analysis of speculative FSM parallelization is precisely estimating Equation 3.7 in practice. We address this challenge with *sample-based conditional regression* (SCR). Different from a classic regression analysis, SCR considers samples conditionally – only if they satisfy the given constraint.

In the context of reprocessing time modeling, a sample in SCR is the convergence length for a pair of states $L^{k}(s_i, s_j)$ on a piece of training input k. The constraint for the samples is the chunk size $\|I\| / n$. For a given state pair (s_i, s_j), SCR maintains a short list of K samples [6]. However, during the regression analysis, SCR only chooses samples with convergence length shorter than the constraint (i.e., $L^{k}(s_i, s_j) < \|I\| / n$). Note that

[6] K is tunable to balance the accuracy and cost. In our evaluation, K is set to 120.

the constraint can vary, depending on the input size $\|I\|$ and the number of cores n. This flexibility allows the customization of SCR based on the needs of scalability analysis. On the other hand, with a pool of samples, the differences among samples resemble the variation of reprocessing costs among different chunks.

Note that convergence length profiling for different state pairs is already required by existing speculative FSM parallelization [138, 137], in order to improve the starting state prediction. In these cases, SCR does not require any extra profiling.

However, maintaining a list of samples for every pair of states could be expensive in terms of both space cost and the cost of regression analysis, especially when the number of states is large. To reduce the total amount of samples, SCR maintains samples only for state pairs that are more likely to appear in actual runs. To find out these state pairs, SCR performs a lightweight state pair frequency profiling offline, by invoking a speculative execution with a large number of parallel threads [7]. Let the set of high-frequency state pairs be S_f^2, then

$$S_f^2 = \{(s, s')|\text{frequency}(s, s') > H_f\} \tag{3.10}$$

where H_f is a predefined frequency threshold.

With the samples of high-frequency state pairs, SCR computes the total reprocessing

[7] In our experiments, this number is set to 1000.

cessing cost estimate T'_{repr} for a configuration n with the following equation:

$$\begin{aligned} T'_{repr}(n) &= \frac{n-1}{\left\|S_f^2\right\| \cdot K} \sum_{(s,s')\in S_f^2} \sum_{k=1}^{K} \mathcal{L}^k(s,s'), \text{ where} \\ \mathcal{L}^k(s,s') &= \begin{cases} L^k(s,s'), & L^k(s,s') < \|I\|/n \\ \|I\|/n, & \text{otherwise} \end{cases} \end{aligned} \tag{3.11}$$

Note that each sample $L^k(s,s')$ is considered only if it satisfies the constraint, that is, $L^k(s,s') < \|I\|/n$. Statistically speaking, if set $H_f = 0$, then we have

$$\lim_{K\to\infty, n\to\infty} T'_{repr}(n) = T_{repr}(n) \tag{3.12}$$

Model M1. Putting all together, we have the estimated speculative execution time

$$T_{spec} = C_{pred} + \frac{\|I\|}{n} + T'_{repr}(n) \tag{3.13}$$

As the sequential execution time $T_{seq} = \|I\|$, we have the first scalability model M1:

$$S_{M1} = \frac{T_{seq}}{T_{spec}} = \frac{\|I\|}{C_{pred} + \|I\|/n + T'_{repr}(n)} \tag{3.14}$$

Based on Equation 3.14, for a given FSM and an input size $\|I\|$, Model M1 can compute the speedup of speculative FSM parallelization for any configuration n, with the help of SCR (Equation 3.11), hence, find out the optimal configuration n^*, such that,

$$S_{M1}(n^*) = max\{S_{M1}(n) | 1 \leq n \leq N_{core}\} \tag{3.15}$$

Depending on the number of state pairs $\left\|S_f^2\right\|$ and the number of samples for each state pair K, the calculation of Equation 3.11 may introduce some runtime cost. One way to reduce the cost is by tuning the state pair frequency threshold H_f and the number of samples K, which in turn may compromise the accuracy. Next, we will discuss another way to balance the accuracy and modeling cost, by simplifying the SCR model. The simplification will lead to a closed-form representation of the optimal configuration.

Model M2. Considering the SCR model in Equation 3.11, there are two scenarios in which the model can be simplified by eliminating the constraint. First, when the input size is large enough or the convergence lengths between state pairs are relatively short, such that $L^k(s, s')$ is often smaller than $\|I\|/n$, then we can assume that $\mathcal{L}^k(s, s') = L^k(s, s')$. Second, when the speculative parallelization does not adopt the state convergence optimization (e.g., because few states can converge, like FSM `div` in [138]), or the convergence length is so long, such that $L^k(s, s')$ is often larger than the chunk size, then we assume $\mathcal{L}^k(s, s') = \|I\|/n$. Putting two scenarios together, we have a new model for reprocessing time estimation:

$$T''_{repr}(n) = \begin{cases} \bar{L} \cdot (n-1), & \bar{L} < \|I\|/n \\ (1-P_s) \cdot (n-1) \cdot \|I\|/n, & \text{otherwise} \end{cases} \tag{3.16}$$

where $\bar{L}$ is the average convergence length among all samples and P_s is the probability of successful speculation, that is, $P(s = s')$. Depending on the ratio between input size and number of cores, the new model $T''_{repr}(n)$ switches between two equations.

By substituting the corresponding term in Equation 3.14 with the new model, we get the second scalability model M2:

$$S_{M2} = \frac{T_{seq}}{T_{spec}} = \frac{\|I\|}{C_{pred} + \|I\|/n + T''_{repr}(n)} \tag{3.17}$$

One advantage of Model M2 is that the optimal number of cores n^* can be represented in a closed-form expression, hence calculated directly without going through the pool of samples (required by Model M1). Considering Equation 3.16 and Equation 3.17 together, we can solve speedup maximization problem in Equation 3.3 and get the following optimal configuration:

$$n^* = \begin{cases} \sqrt{\|I\| / \bar{L}}, & \bar{L} < \|I\| / N_{core} \\ \sqrt{1/(1-P_s)}, & \text{otherwise} \end{cases} \tag{3.18}$$

where $\bar{L}$ and P_s capture the convergence properties of the FSM and the speculation accuracy, respectively.

Equation 3.18 quantitatively reflects two basic intuitions behind scalability analysis. First, as the convergence length $\bar{L}$ increases, the optimal configuration n^* should be reduced. Second, when the speculation accuracy P_s increases, the speedup tends to be better when choosing to use more available cores.

Discussion. Comparing models M1 and M2, there is a tradeoff between the accuracy and the modeling cost. On one hand, with the SCR, Model M1 captures more details of misspeculation cost variation, hence tends to be more accurate in most cases. Meanwhile, M1 incurs more overhead as it needs to go through the pool of samples to calculate the speedup for each configuration. On the other hand, though Model M2 directly computes the optimal configuration, it may lose some accuracy, especially when the average convergence length $\bar{L}$ is close to the chunk size $\|I\| / n$.

Both models M1 and M2 are solely based on FSMs' properties, and can be used for comparing the scalability of different FSMs when being executed speculatively. In practice, the actual scalability also depends on the characteristics of underlying architecture. Next, we will discuss how to extend the FSM properties-based scalability models to *architecture-aware scalability models*.

3.5 Towards Architecture-Aware Scalability Analysis

On different architectures, the scalability of a type of computations may vary significantly, not only depending on the characteristics of the architecture, but also depending on their interaction with the computations. To enable accurate and practical scalability analysis for a given computing platform, this section presents architecture characterizations and discusses how to integrate them into the scalability models introduced in Section 3.4.

3.5.1 Architecture Effects

Considering the complexity of modern architectures, the interplay between an architecture and an application can be quite involved. Here, we focus on the end-to-end architecture effects that are closely relevant to the performance of speculative FSM parallelization. In another word, our architecture characterizations are customized for speculative FSM parallelization.

Since the execution time of speculative FSM execution mainly consists of two phases: parallel processing phase and sequential reprocessing phase (Section 3.4.2), we separate our discussion on the two phases. In specific, for each phase, we identify the major factor(s) that directly influences the performance.

Resource Contention in Parallel Phase. During the parallel phase, a group of n threads are created, each of them occupying a separate (logical) core. Based on their predicted starting states, these threads proceed with their own input chunks individually, and do not need to communicate either other. Thus, they do not suffer from any lock contention that is often caused by concurrent access of the shared data structures [8]. However, different cores physically share hardware resources, such as last level cache (LLC) and memory bandwidth, and even more resources among logical cores in a hyper-threaded core. The sharing of resources leads to contentions that directly influence the performance of this phase.

As the number of cores used increases, the resource contention tends to increase as well. However, the contention may not increase linearly or even monotonically, depending on the design of the architecture as well as the mapping between threads and logical cores. Without loss of generality, this chapter assumes that `S3` uses the default mapping that is chosen by the operations system (defined in `/proc/cpuinfo`).

To quantitatively measure the resource contention, we introduce the metric *contention factor*, denoted as $\alpha(n)$.

$$\alpha(n) = T(n)/T(1) \tag{3.19}$$

where $T(i)$ is the execution time of processing i input chunks of the same length with i cores. Contention factor $\alpha(n)$ captures the degree of resource contention when executing with n parallel threads, comparing with a single thread execution. For commonly used architectures, it is expected that $\alpha(n) > 1$. For a given architecture, the contention factor $\alpha(n)$ can be easily measured by running a micro benchmark N_{core} times.

[8] Threads do share the same FSM transition table, but only perform read operations.

Relative Execution Speed of Reprocessing. After entering into the reprocessing phase, only one thread is left, responsible for validating the correctness of each speculation and correcting the mistakenly processed parts caused by misspeculation. This implies there is no resource contention in this phase (i.e., $\alpha(1) = 1$). However, due to the tracking of state convergence (between s_{pred} and s_{true}), the execution speed (i.e., processing time per symbol) in reprocessing phase might be relatively slower than regular state transitions. The actual difference depends on the architecture, meanwhile, affects the scalability: the slower the reprocessing is, the less scalability the speculative parallelization can achieve.

To capture the relative execution speed, we introduce the *relative speed* factor, denoted as γ.

$$\gamma = T^0_{repr}/T^0_{seq} \tag{3.20}$$

where T^0_{repr} and T^0_{seq} are the processing time of a single symbol during reprocessing and a sequential execution, respectively. It is also expected that $\gamma > 1$. Similar to contention factor, γ can be measured with a micro FSM benchmark, but just running twice.

For a given architecture, $\alpha(n)$ and γ only need to be profiled once. Next, we discuss how to integrate these two architecture factors into the scalability models presented in Section 3.4. The integration will lead to a pair of architecture-aware scalability models that are more accurate and practical than their counterparts.

3.5.2 Integration of Architecture Factors

We first consider the resource contention factor $\alpha(n)$ in the parallel phase, then the relative speed factor γ in the reprocessing phase, and finally put them together.

The parallel phase execution time model in Equation 3.6 assumes that the parallel processing time T_{proc} equals sequential processing time (modeled as $\|I\|$) divided by the number of cores n. When considering the resource contention factor (Equation 3.19), that is, $\alpha(n) = T_{proc}/(T_{seq}/n)$, we can easily infer

$$T_{proc} = \alpha(n) \cdot \frac{T_{seq}}{n} = \alpha(n) \cdot \frac{\|I\|}{n} \tag{3.21}$$

Equation 3.21 implies that the higher the resource contention is, the longer the parallel processing time would become.

Similarly, we can extend to the reprocessing phase model in Equation 3.7 by integrating relative speed factor γ

$$T_{repr} = \gamma \cdot \sum_{i=2}^{n} L^i(s^i_{pred}, s^i_{true}) \tag{3.22}$$

Putting all together, we have two enhanced scalability models M1+ and M2+, corresponding to models M1 and M2, respectively.

Model M1+. Based on Model M1 in Equation 3.14, we have the following extended Model M1+ with architecture factors.

$$S_{M1+} = \frac{T_{seq}}{T_{spec}} = \frac{\|I\|}{C_{pred} + \alpha(n) \cdot \|I\|/n + \gamma \cdot T'_{repr}(n)} \tag{3.23}$$

Model M2+. Similarly, based on Equations 3.16 and 3.17, we extend Model M2 to Model M2+ as follows.

$$S_{M2+} = \frac{T_{seq}}{T_{spec}} = \frac{\|I\|}{C_{pred} + \alpha(n) \cdot \|I\|/n + \gamma \cdot T''_{repr}(n)} \tag{3.24}$$

where $T''_{repr}(n)$ is defined the same as that in Equation 3.16.

Augmented with architecture factors, models M1+ and M2+ are expected to provide more accurate scalability analysis results that are customized to a specific architecture.

3.6 Implementation

We implemented S3 based on the OptSpec library [138, 137], which is implemented in C language and leverages Pthread for multi-threading. At high level, there are three major components: (i) An FSM property collector for profiling state convergence properties. The collector can be tuned either online using testing inputs or offline using training inputs. The cost of online profiling has been optimized with techniques from prior work [137] (typically less than 5%); (ii) An offline architecture property collector which runs a small set of micro FSM benchmarks on the target machine to measure the resource contention $\alpha(n)$ and relative speed factor γ; And (iii) a runtime controller that implements the scalability models. Based on the collected the FSM and architectural properties, the controller calculates the optimal configuration n^*, and feeds it into speculative parallelization setting at runtime.

3.7 Evaluation

In this section, we evaluate S3 on large-scale shared memory architectures, including a standalone Xeon Phi processor with 256 logical cores. The evaluation mainly focuses on two aspects: the accuracy of scalability analysis and the performance and energy benefits from using S3. We also discuss of scalabilities of some specific FSM computations based on experimental results.

3.7.1 Methodology

We compare S3 with two methods. One is the *default setting* of OptSpec [138, 137] which uses all available cores on the machine; The other is the *exhausted searching* that provides the ground truth of optimal configuration. In specific, given a FSM, the input size and an architecture with N_{core} cores, the exhausted searching executes the FSM with its inputs on the architecture using 1 to N_{core} cores to find the optimal number of cores. Obviously, it is unreasonable to use exhausted searching in real situations as trying one configuration is already at least as costly as the executing the best configuration, not mentioning enumerating all configurations.

We run our experiments on three different architectures, which are summarized in Table 3.1. Due to space limit, we mainly focus on the results on Xeon Phi architecture. Xeon Phi runs Linux 3.10.0 with GCC 4.8.5, while the other two run Linux 3.10.0 with GCC 4.47. All programs are compiled with "-O3" optimization flag. The timing results reported are the average of 10 runs on 10 inputs, with all runtime cost included.

Table 3.1: Architectures in Evaluation

Arch.	model	freq.	#cores*	#SMT	#Sockets
Xeon Phi	Xeon Phi 7210	1.30 Hz	256	4	1
Haswell	Xeon E5-2698	2.30 Hz	64	2	2
Ivy Bridgy	Xeon E7-8860	2.27 Hz	80	1	8

*The numbers of cores shown are the number of logical cores recognized by operating systems.

The benchmarks are collected from real-world FSM applications, primarily from Snort [108], one of the most widely used open source Network Intrusion Detection Systems (NIDSs). It has a rich body of signatures/rules, most of which are specified by a Perl-

Table 3.2: FSM Benchmarks and Their Properties

FSM	#states	avg(L)	FSM	#states	avg(L)
openview	501	1.27E+07	spirit	2041	8.09E+01
tomcat	10	6.56E+06	jnlp	1211	8.09E+01
iis	32	1.27E+07	postgre	28	9.98E+04
cnc	115	9.93E+04	apache	6	2.55E+05
rtf	768	8.66E+04	mutiny	21	2.53E+05
warehouse	82	1.81E+06	buffer	10	1.12E+06
dfs	26	6.83E+04	adware	5265	1.20E+02

compatible regular expression (PCRE). We converted the PCREs to FSMs using standard regular expression to FSM conversion algorithms [2]. The inputs to the FSMs are network traffic traces collected from a Linux server and a laptop via `tcpdump`, with a total size of 18GB. Table 3.2 summarizes the 14 benchmarks used in our evaluation, including the number of states and the average convergence length collected from high-frequency state pairs, each with 120 samples.

3.7.2 Model Accuracy

Table 3.3 reports the optimal configuration n^* found by exhausted searching and the four models of `S3`, on architectures Xeon Phi and Haswell [9]. “Exs” shows the actual optimal number of cores by enumerating all configurations (i.e., the “ground truth”). Note that, the predicted optimal number of cores is bounded by the total number of cores in the tested platforms. Overall, architecture-aware models (M1+ and M2+) are more accurate than architecture-independent models (M1 and M2), especially for benchmarks with better scalabilities, thanks to their consideration of architecture factors $\alpha(n)$ and γ. The differences between M1+ and M2+ are not significant for most benchmarks, similar to M1 and M2. The

[9] The results on Ivy Bridge follow similar patterns.

largest difference happens on benchmark `rtf`, where M2 turns to be much over optimistic (80 v.s. 42). In comparison, the result of M1 is very close to the actual optimal (43 v.s. 42). Also note that the results of M2 is closer to the ground truth than M1 in general. On one hand, due to the simplification of SCR model, M2 predicts less accurately than M1 in reprocessing length (more pessimistic in most cases). On the other hand, both M1 and M2 miss the architecture factors as mentioned above, and tend to be more optimistic. Because of the "balance" that happens to M2, results from M2 turn to be closer to the real cases than M1.

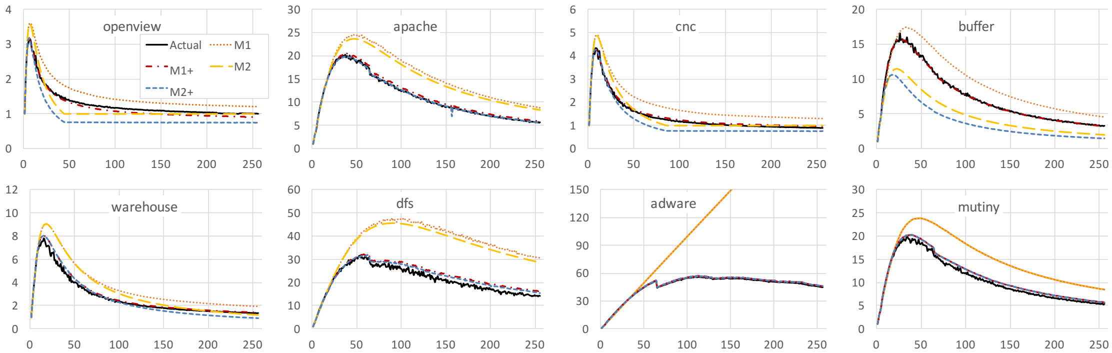


Figure 3.5: Comparison of Speedup Curves from Different Models. (x-axis is the number of cores; y-axis is the speedup)

To further examine the overall accuracy in scalability analysis, we collected the speedup curves for each benchmark on each architecture. Due to space limit, we only report some representative results in Figure 3.5. In general, the speedup curves clearly show the effectiveness of the two architectural-aware models (M1+ and M2+), whose speedup curves precisely align with the actual speedup curve for most benchmarks. Between M1+ and M2+, M2+ performs less reliable than M1+, especially on benchmarks `openview` and

Table 3.3: Optimal Configurations from Different Methods

	Xeon Phi					Haswell				
	Exs	M1	M1+	M2	M2+	Exs	M1	M1+	M2	M2+
openview	6	7	6	7	6	7	7	7	7	7
tomcat	8	9	8	8	8	12	9	11	9	11
iis	6	7	6	7	6	7	7	7	7	6
warehouse	15	18	15	18	15	21	18	22	18	22
apache	37	48	40	47	37	53	48	49	47	49
mutiny	34	49	39	47	37	44	49	47	47	47
buffer	26	33	27	22	17	34	33	40	22	22
cnc	51	83	60	75	54	62	62	59	64	59
rtf	42	43	41	80	63	54	43	49	64	62
dfs	54	100	62	90	62	59	64	57	64	57
postgre	50	85	57	74	57	56	63	55	64	55
adware	110	256	112	256	112	58	64	57	64	57
spirit	116	256	116	256	116	62	64	58	64	58
jnlp	116	256	114	256	114	60	64	59	64	59

buffer, due to its simplification of the SCR model (Section 3.4). Model M1+ shows some slight discrepancy on benchmarks buffer, openview, and mutiny. This is mainly caused by the characteristic differences between the samples and the testing inputs.

3.7.3 Performance Improvement

We next present the performance benefits of S3, comparing with the default setting of speculative FSM parallelization [138, 137]. Figure 3.6 shows the speedup (baseline is sequential FSM execution) of all five methods on 14 benchmarks and three architectures. "Exhaust" represents the ideal speedup that can be achieved by tuning the number of cores. The most performance gains come from the results on Xeon Phi, for its larger number of available cores. On average, S3 boosts the speedup from 6.1X to 16.7X with Model M1+. For architecture-independent models (M1 and M2), the improvements are slightly less, but still reaching 15X. For benchmark buffer, the speedup is improved by a factor of five (3.2X

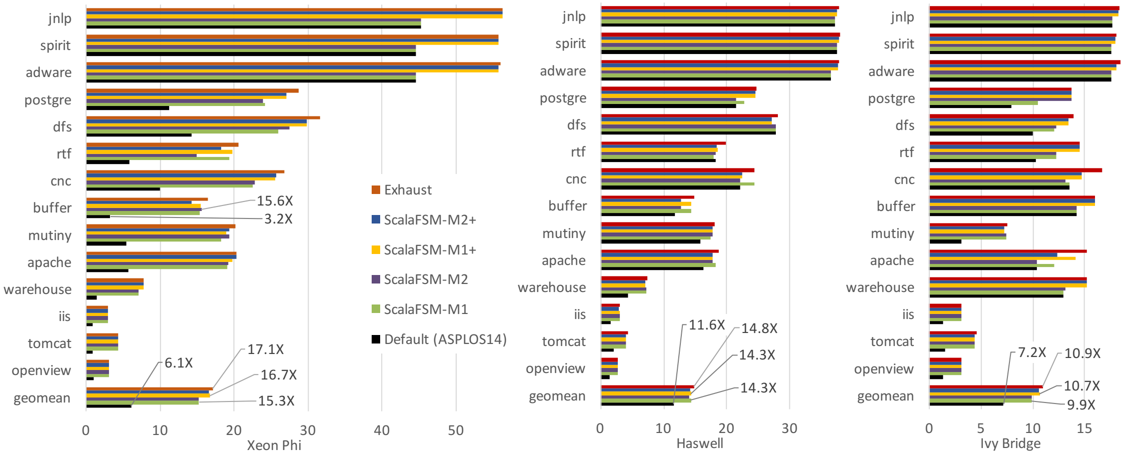


Figure 3.6: Speedup Comparison between S3 and The State of The Art [138] on Three Architectures: Xeon Phi, Haswell, and Ivy Bridge

v.s. 15.6X). Results on Haswell and Ivy Bridge follow similar trends in general, but are less significant due their limited number of available cores. Overall, the results imply the necessity of scalability-aware speculative FSM parallelization, especially considering future parallel platforms with even more number of processing units.

3.7.4 Energy Saving

Finally, we briefly discuss one side benefit of scalability-sensitive speculative FSM parallelization – energy saving. The energy saving primarily comes from the use of less number of processors. Table 3.4 reports the energy saving in percentage on Xeon Phi and Haswell architectures. On Xeon Phi, the energy saving is more significant, up to 77%, because it has more room to reduce the amount of core uses. However, on Haswell, we also observed cases with even more energy consumption. This is because when using all available cores (64 cores), though the power consumption is higher, the execution time becomes

Table 3.4: Energy Saving by S3 (baseline: Default Setting)

	Xeon Phi				Haswell			
	M1	M1+	M2	M2+	M1	M1+	M2	M2+
openview	67.7%	67.1%	67.7%	67.1%	37.2%	37.2%	37.2%	37.2%
tomcat	78.6%	75.7%	75.7%	75.7%	46.0%	44.0%	46.0%	44.0%
iis	69.5%	67.7%	69.5%	67.7%	41.2%	41.2%	41.2%	40.4%
warehouse	79.9%	80.3%	79.9%	80.3%	18.7%	19.6%	18.7%	19.6%
apache	67.2%	68.0%	67.1%	68.3%	-1.4%	-1.8%	-2.4%	-1.8%
mutiny	68.0%	69.2%	69.6%	69.5%	-0.8%	-0.6%	-0.6%	-0.6%
buffer	76.9%	77.3%	77.2%	75.3%	-11.4%	-1.4%	-18.9%	-18.9%
cnc	49.7%	54.8%	51.8%	55.8%	0.3%	1.2%	0.0%	1.2%
rtf	66.3%	66.3%	57.2%	64.1%	-8.9%	-4.7%	0.0%	0.7%
dfs	35.4%	48.1%	37.6%	48.1%	0.0%	5.0%	0.0%	5.0%
postgre	47.3%	54.5%	50.9%	54.5%	-1.6%	1.0%	0.0%	1.0%
adware	0.0%	13.7%	0.0%	13.7%	0.0%	3.6%	0.0%	3.6%
spirit	0.0%	15.2%	0.0%	15.2%	0.0%	3.9%	0.0%	3.9%
jnlp	0.0%	12.9%	0.0%	12.9%	0.0%	4.8%	0.0%	4.8%

shorter (due to smaller chunk size $\|I\| / n$), comparing with the optimal core predicted by the models (e.g., 22 cores for M2 and M2+). In another word, for speculative FSM parallelization, it is not necessary that more number of cores always leads to higher energy consumption. We leave further investigation as future work.

3.8 Summary

With a systematic scalability study, this chapter points out a principal fallacy in the existing design of speculative FSM parallelization when being ported to a larger parallel platform. To address the issue, this work introduces a series of scalability analysis models, which are tailored to both the properties of FSM computations and the characteristics of the underlying architecture. To leverage the proposed models, this chapter develops an automatic speculative FSM parallelization framework S3, which, for the first time, enables a scalability-sensitive speculative parallelization for FSM computations. For a given FSM, its input size and the architecture, `S3` can automatically compute the optimal number of cores to use and guide the speculative parallelization towards the best performance. Evaluation on FSM benchmarks with a spectrum of scalabilities demonstrates the effectiveness of the new speculative parallelization scheme, showing up to 5X speedup comparing to the state-of-the-art methods as well as up to 77% energy saving.

Chapter 4

Scalable FSM Parallelization via Path Fusion and Higher-Order Speculation

4.1 Introduction

The effectiveness of the existing parallelization schemes highly depends on the state convergence properties of the FSM. For FSMs exhibiting limited state convergence, they suffer from high overhead and poor scalability.

In this work, we introduce two techniques: *path fusion* and *higher-order speculation*, to address the scalability limitation in each basic FSM parallelization scheme, respectively. For state enumeration, to reduce the cost of maintaining multiple execution paths, we propose to fuse different paths into a single execution path. Unlike path merging, path fusion is not based on the state convergence of different execution paths. Instead, its idea stems from the classic NFA to DFA [1] conversion [2] – a way to remove the non-deterministic behaviors of NFA. Basically, path fusion encodes a *vector of states* (rather than a subset of states as in the NFA-DFA conversion) in the original FSM into a new state, based on which it generates a *fused* FSM. Thus, a single execution path of the fused FSM mimics multiple state execution paths of the original FSM. However, in principle, the fused FSM could be much larger than the original one. To address this, we explore dynamic fused FSM generation techniques, which leverage biased transitions to generate a partially fused FSM on the fly.

For state speculation, the scalability is bottlenecked by the serial validation – the validation of a speculated state has to wait for the ground truth to be propagated from the first chunk to its own chunk. To address it, we propose to speculatively validate the correctness of the speculated state, referred to as *higher-order speculation*. In another word, in the scheme of higher-order speculation, the "ground truth" for the validation itself could be speculated.

Based on this concept, we design an iterative speculation scheme for FSM parallelization, where the whole processing is organized into a series of iterations. In each iteration, a worker thread always validates its starting state using the ending state of the prior chunk (which may be incorrect). The iteration finishes when the actual ground truth, called *frontier*, reaches the last chunk. In essence, iterative speculation allows different input chunks to be validated in parallel (though speculatively), thus potentially accelerating the propagation of the ground truth, shortening the total time.

[1] Non-deterministic finite automata and deterministic finite automata.

Furthermore, to cope with the rich yet diverse properties of FSMs, we design a heuristic to automatically select the FSM parallelization scheme that can maximize the performance benefits. Finally, we implemented the proposed techniques into an FSM parallelization framework, named BoostFSM. Our evaluation using a set of real-world FSMs with a spectrum of state convergence properties confirms the effectiveness of the proposed techniques. For FSMs exhibiting limited state convergence properties, BoostFSM improves the speedups from 1.1X-4.6X to 12.3X-35.5X on a 64-core machine.

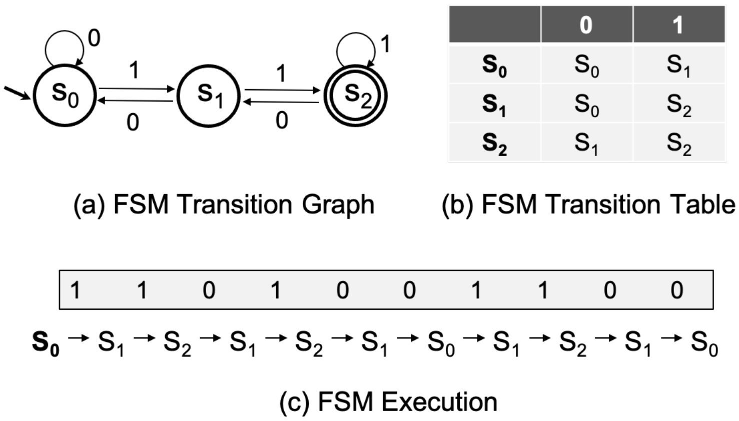
Next, we start with the background of this work.

4.2 Background

In this section, we first introduce the basics of FSM and the inherent dependences in its execution, then present the details of the two basic parallelization schemes.

FSM and Its Dependences. As shown in Figure 4.1-(a), an FSM can be represented as a directed graph, where nodes represent the *states*, edges represent the *transitions*, and labels on the edges indicate the *conditions* for the transitions to happen. The transitions can be stored in the memory as a *transition table*, as shown in Figure 4.1-(b). The size of the table is $N \times M$, where N is the number of states and M is the number of symbols.

The execution of an FSM starts from the *initial state* (S_0 in the example) and makes transitions by consuming input symbols one by one, as shown in Figure 4.1-(c). An FSM may consist of *accept states*, denoted as nodes with double circles. The meaning of accept states varies across FSM applications. For example, they may correspond to the codes in Huffman decoding [67] or matches in pattern searching [134].



(a) FSM Transition Graph

	0	1
S_0	S_0	S_1
S_1	S_0	S_2
S_2	S_1	S_2

(b) FSM Transition Table

1	1	0	1	0	0	1	1	0	0

$\mathbf{S_0} \to S_1 \to S_2 \to S_1 \to S_2 \to S_1 \to S_0 \to S_1 \to S_2 \to S_1 \to S_0$

(c) FSM Execution

Figure 4.1: FSM Example

From the FSM execution shown in Figure 4.1-(c), it is easy to find that every state (except the starting state) in the transition sequence depends on not only the input symbol but also the prior state. Together, they form a chain of *dependences*, which inherently prevent the FSM from running in parallel. A series of studies [50, 62, 138, 83, 137, 99, 100] have been conducted to "break" the dependence chain. Despite the differences in detail, they fall into two basic categories: *state enumeration* and *state speculation*. Next, we describe each of them with examples.

State Enumeration. Considering the parallelization of the FSM execution in Figure 4.1-(c), we partition the input into two chunks (see Figure 4.2). For the first chunk, we start the FSM execution from the initial state S_0. However, for the second chunk, we do not know its starting state, as it depends on the processing of the first chunk. The basic idea of state enumeration [50, 83] is to fork an execution path for each state (S_0, S_1, and S_2) in the FSM. Certainly, one of the execution paths must be correct. As demonstrated in Figure 4.2, S_1 is later found to be the actual starting state, based on the ending state of *chunk_0*. Hence, its execution path will be connected to that of the prior chunk, while the others will be discarded.

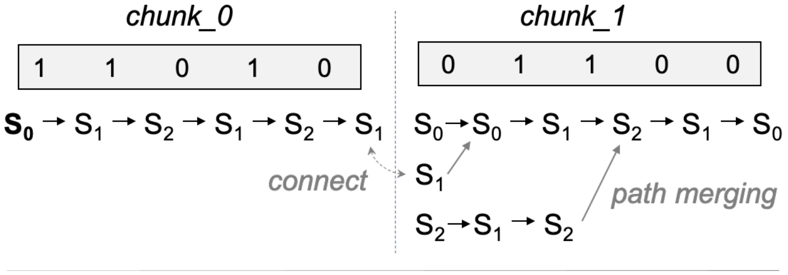

chunk_0
chunk_1
1 1 0 1 0
0 1 1 0 0
$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_1 \rightarrow S_2 \rightarrow S_1$
$S_0 \rightarrow S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_1 \rightarrow S_0$
connect
S_1
path merging
$S_2 \rightarrow S_1 \rightarrow S_2$

Figure 4.2: State Enumeration

Obviously, maintaining multiple execution paths may create significant overhead, which can compromise or even outweigh the benefits of parallelization. To reduce the overhead, prior work [83] has made an interesting observation: after certain number of transitions, different execution paths may merge. For example, the first and second paths (started with S_0 and S_1) both transition into S_0 after reading the first symbol 0. Later, the third path (started with S_2) also merges with the rest. After paths merge, only one of them needs to be maintained, thus reducing the cost of paths maintenance.

However, the effectiveness of path merging highly depends on the convergence properties of the FSM. A recent study [83] has shown that, for many real-world FSMs (from Snort [108]), most states tend to converge quickly, but a few states fail to converge after a large number of transitions. As an example, Figure 4.3 shows an FSM slightly different from that in Figure 4.1. But it exhibits a very different convergence property: no matter what input is given, none of the states converge.

To address this limited state convergence, prior work [83] explores SIMD parallelism – using different SIMD lanes for different execution paths. However, such hardware parallelism can be otherwise used to enable more fine-grained data-level parallelization – partitioning the input into more chunks [99]. Moreover, its efficiency is restricted by the SIMD width. When the number of remaining paths is more than the SIMD width, a transition needs multiple rounds of SIMD operations.

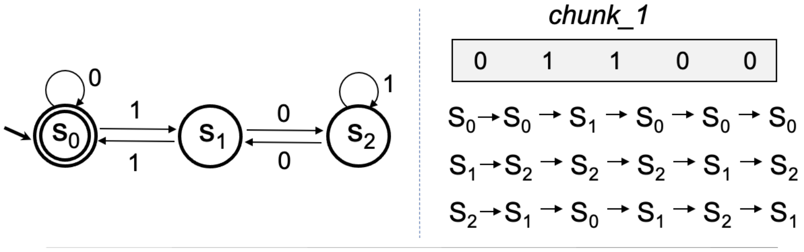


Figure 4.3: FSM Example with Poor Convergence

State Speculation. Instead of enumerating all the states at the beginning of a chunk (except for the first), another strategy is to speculate the starting state. As illustrated in Figure 4.4, state S_2 is speculated to be the starting state, with which *chunk_1* is processed. However, the speculation could be incorrect – *misspeculation*, in which case the corresponding chunk needs to be reprocessed. In the example from Figure 4.4, S_1 is later found to be the correct starting state. As a result, *chunk_1* gets reprocessed. Luckily, path merging can also be applied here between the reprocessing path and the speculated processing path. Once they merge, the reprocessing can safely stop.

As to how the speculation (state prediction) is performed, prior work [138] has made a comprehensive study. The basic idea is to use the suffix of the prior chunk as a

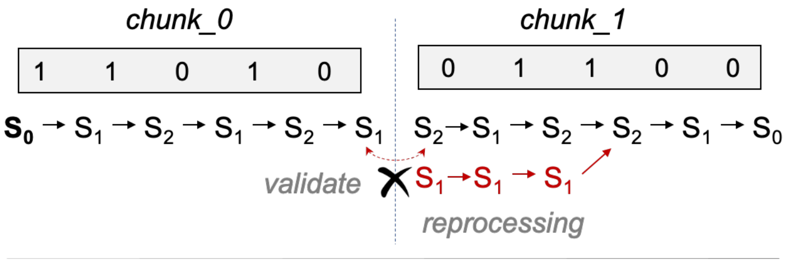

chunk_0
1 1 0 1 0
chunk_1
0 1 1 0 0
$S_0 \to S_1 \to S_2 \to S_1 \to S_2 \to S_1$
$S_2 \to S_1 \to S_2 \to S_2 \to S_1 \to S_0$
validate
$S_1 \to S_1 \to S_1$
reprocessing

Figure 4.4: State Speculation

"partial context" to infer the chance that each state is the starting state. We will provide more details about state prediction in Section 4.4.

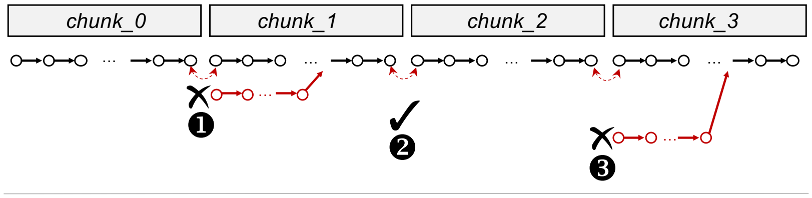


Figure 4.5: Sequential Validations

The scalability bottleneck in this speculative parallelization lies in the *sequential validations*. When the input is partitioned into multiple chunks, the validations have to be conducted in order from the second chunk to the last, as shown in Figure 4.5. Because, before the prior chunk is validated (and reprocessed under misspeculation), we are not sure if its ending state is correct. In another word, the *ground truth* has to be propagated from the first chunk, chunk by chunk. This is less a concern when the speculation accuracy is high or the reprocessing lengths are short. However, when the speculation accuracy drops and the reprocessing paths fail to converge with their speculative processing paths quickly, the bottleneck could seriously limit the scalability of speculative parallelization.

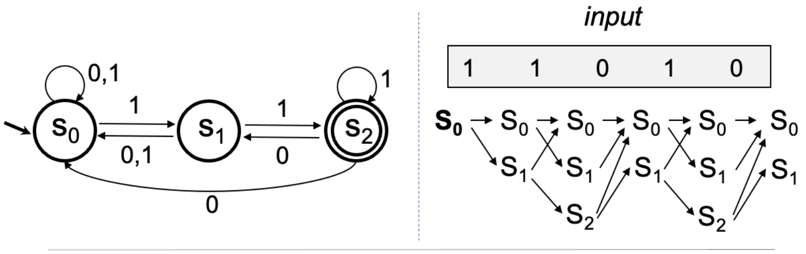
In summary, the efficiency of both parallelization schemes relies on the underlying state convergence properties of the FSM. For FSMs exhibiting limited state convergence, they all suffer from high overhead, leading to poor scalability. In the following, we will introduce two techniques, *path fusion* and *higher-order speculation*, to address the scalability limitations in the two schemes, respectively. After that, we will present a heuristic-based approach to facilitate the selection between the two augmented parallelization schemes.

4.3 Path Fusion

In this section, we present *path fusion*, a technique that fuses different FSM execution paths into a single path, to boost the efficiency of enumerative parallelization. Note that, unlike the path merging mentioned in Section 4.2, path fusion is not based on the state convergence property of FSMs. Instead, its idea is inspired by the classic NFA to DFA conversion [2]. For this reason, we first briefly review the NFA to DFA conversion and compare it with path fusion intuitively, then we present the basic algorithm of path fusion and discuss how to adopt it dynamically during the enumerative parallelization.

4.3.1 Motivation

According to automata theory, NFAs exhibit non-deterministic behaviors – a state in an NFA may transition to multiple states after reading an input symbol. Take the NFA in Figure 4.6 as an example, state S_0 transitions to both itself and S_1, after reading symbol 1. Similar non-deterministic behaviors are also shown in states S_1 and S_2. As a result, an NFA execution needs to maintain multiple current states (bounded by the total number of states), which leads to poor execution efficiency. A well-known solution to this problem is converting an NFA to a DFA using the subset construction algorithm [2].



0,1
1
0,1
1
0
1
0
S_0
S_1
S_2
input
1 1 0 1 0
S_0 S_0 S_0 S_0 S_0 S_0
S_1 S_1 S_1 S_1 S_1
S_2 S_2

Figure 4.6: NFA and Its Execution

The basic idea behind the subset construction is to map a subset of NFA states to one state of the constructed DFA. In this way, a DFA execution with one current state can simulate an NFA execution with a set of current states. For the example in Figure 4.6, by mapping $\{S_0\} \rightarrow S'_0$, $\{S_0, S_1\} \rightarrow S'_1$, and $\{S_0, S_1, S_2\} \rightarrow S'_2$, we can convert the NFA to the DFA in Figure 4.1. To find out the mapping, the construction starts from the initial state of the NFA, then it uses a worklist strategy to iteratively discover new sets of reachable states (i.e., DFA states) by making transitions on every input symbol. More details of this construction algorithm can be found in [2].

NFA vs. State Enumeration. One interesting observation we had is that state enumeration and NFA suffer from similar kind of inefficiency issues in their executions – both of them need to maintain multiple current states in general (compare Figure 4.3 and Figure 4.6). However, there are some critical differences:

- First, state enumeration maintains a *vector* of states for an ordered sequence of FSM execution paths. The ordering is essential to selecting the right execution path later during the enumerative parallelization. By contrast, an NFA only maintains a *subset* of states, without any ordering.

- Second, while the number of current states in an NFA execution may increase or decrease (see Figure 4.6), the number of current states in state enumeration can only decrease, which happens during path merging.

Despite the differences, a natural question we asked is: *can we design a technique similar to the NFA to DFA conversion for the state enumeration to address its execution inefficiency?* Fortunately, we found the positive answer to the question. In fact, by adopting a worklist-based strategy like the one used in the subset construction algorithm [2], we can generate a new (fused) FSM whose single execution path simulates multiple execution paths of the original FSM. We refer to this technique as *path fusion*. Next, we introduce its basic algorithm.

4.3.2 Static Path Fusion

The key to path fusion is to construct a new FSM, referred to as *fused FSM*, where each state corresponds to a vector of states in the original FSM. Similar to the subset construction in NFA to DFA conversion [2], we can statically construct a fused FSM without any actual inputs.

Algorithm 6 Fused FSM Construction

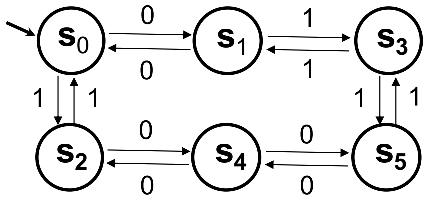
```
**Input**: FSM with $trans[S_j][c_i]$, $j \in [0, N)$, $i \in [0, C)$
**Output1**: Fused FSM with $Trans[\mathbf{S}_j][c_i]$, $j \in [0, M)$, $i \in [0, C)$
**Output2**: Mapping from fused states $\mathbf{S}$ to state vectors $V$: $map$

$V_0 = [S_0, S_1, \cdots, S_N]$
$map$.insert($V_0$, $\mathbf{S}_0$)
$cnt = 1$ /* counter of fused states */
$worklist = \{V_0\}$
**while** $worklist$ is not empty **do**
    remove an item $V$ from the $worklist$
    $\mathbf{S} = map$.find($V$)
    **for** each input symbol $c_i$ **do**
        **for** each state $S_j$ in $V$ **do**
            $V_{next}[j] = trans[S_j][c_i]$
        **if** $map$.find($V_{next}$) == null **then** /* first time meet it? */
            $map$.insert($V_{next}$, $\mathbf{S}_{cnt}$)
            $cnt = cnt + 1$
            add $V_{next}$ to $worklist$
        $\mathbf{S}_{next} = map$.find($V_{next}$)
        $Trans[\mathbf{S}][c_i] = \mathbf{S}_{next}$ /* record fused state transition */
reverse the key and value in $map$
```

Algorithm. Algorithm 6 presents a worklist-based strategy to construct a fused FSM with states $\{\mathbf{S}_0, \mathbf{S}_1, \cdots, \mathbf{S}_M\}$ from a given FSM with states $\{S_0, S_1, \cdots, S_N\}$. Initially, it sets

$V_0 = [S_0, S_1, \cdots, S_N]$, which represents the initial state $\mathbf{S}_0$ of the fused FSM, because we fork an execution path for each state S_i $(0 \leq i < N)$ in the enumerative parallelization. After that, the algorithm initiates a worklist with V_0. From there, it iteratively removes a state vector from the worklist, finds out its next state vector V_{next} for each input symbol c_i $(0 \leq i < C)$. If V_{next} represents a new fused state S_{cnt}, record their correspondence and add V_{next} back to the worklist (Line 15-18). Then, record the fused state transition (Line 20). Finally, it creates a *map* from fused states to state vectors (Line 21) which is to separate a fused path back to a state vector in the end of the processing.

Note that, by only adding new fused states to the worklist, the algorithm will always terminate, as the number of states in the fused FSM is bounded by the size of the N-dimensional vector space N^N. However, in practice, the algorithm only traverses a very small fraction of the entire vector space, as we will demonstrate shortly after a quick example.

Example. Figure 4.7 shows the fused FSM generated for the FSM example in Figure 4.3. The fused FSM consists of 6 states, whose IDs follow the order that the states are created. With the fused FSM, the three execution paths on *chunk_1* shown in Figure 4.3 can be reduced to a single execution path of the fused FSM: $\mathbf{S}_0 \rightarrow \mathbf{S}_1 \rightarrow \mathbf{S}_3 \rightarrow \mathbf{S}_1 \rightarrow \mathbf{S}_0 \rightarrow \mathbf{S}_1$. Later, if S_2 turns out to be the actual starting state of *chunk_1*, we can then immediately find that the actual ending state of *chunk_1* is S_2, the third element in the state vector corresponding to $\mathbf{S}_2$. This shows the importance of using vectors instead of subsets as the states of fused FSM – keeping the order of current states allow us to map the starting state to its ending state.



(a) Fused State Transitions

K	V	K	V
S_0	$[S_0, S_1, S_2]$	S_3	$[S_1, S_2, S_0]$
S_1	$[S_0, S_2, S_1]$	S_4	$[S_2, S_0, S_1]$
S_2	$[S_1, S_0, S_2]$	S_5	$[S_2, S_1, S_0]$

(b) Fused States → State Vectors

Figure 4.7: Static Fused FSM for the FSM in Figure 4.3

Note that, though the vector space for the FSM example in Figure 4.3 is $3^3 = 27$, the statically generated fused FSM only consists of 6 states. Next, we show this is not a special case, but a prevalent property of the fused FSMs.

Cost in Practice. Similar to the NFA to DFA conversion [2], the actual costs of fused FSM construction for real-world FSMs are significantly less than the theoretical complexity. To demonstrate this, we randomly selected 392 FSMs from the Snort [108] library. Figure 4.8 reports the actual number of states in the fused FSMs for 377 out of the 392 FSMs. We can find that, for most FSMs, the numbers of states in their fused FSMs are below N^2 (red diamonds), where N is the number of states in the original FSM. These results confirm the feasibility of static fused FSM generation for many real-world FSMs.

Despite the promises of static fused FSM generation for many real-world FSMs, we still found that, for 15 out of 392 FSMs, the algorithm fails to generate fused FSMs in 3 minutes or generates fused FSMs with over 1 million states. When the size of the generated FSM is very large, it not only requires significant amount of memory, but also may slow down the FSM execution, as the transition table would largely reside in the memory, rather

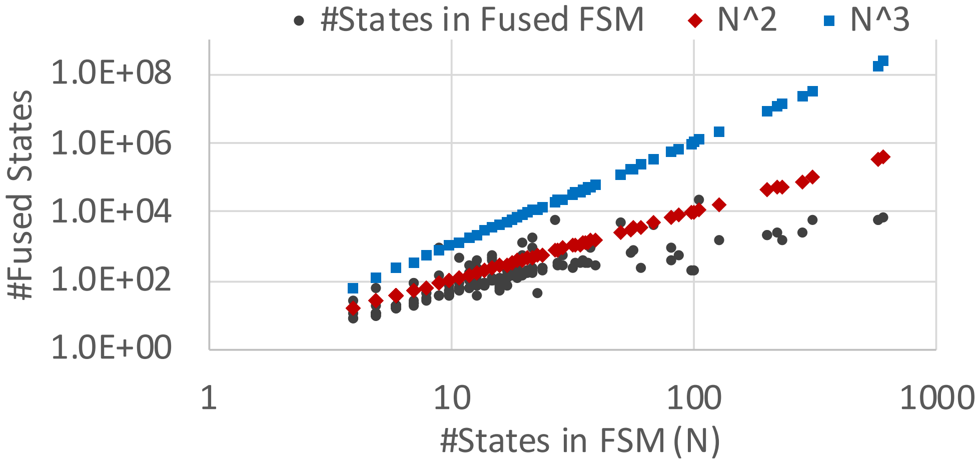


Figure 4.8: Number of States in Fused FSMs

than CPU caches. For this reason, we next explore the possibility of dynamically generating a partially fused FSM during the enumerative FSM execution.

4.3.3 Dynamic Path Fusion

Unlike static path fusion which builds the entire fused FSM with states and transitions for all possible inputs, dynamic path fusion constructs a partial fused FSM that only consists of states and transitions for a single input, with the goal to reduce the memory needs and improve the data locality. This is especially critical for FSMs whose fused FSMs otherwise could not be constructed statically in practice.

Algorithm. The application of dynamic path fusion resembles the just-in-time (JIT) compilation strategy used in modern compilers. It consists of two execution modes:

- *Basic* Mode. Given a vector of current states V and an input symbol c, this mode makes individual transitions for each state in the vector to obtain the next state vector V_{next}:

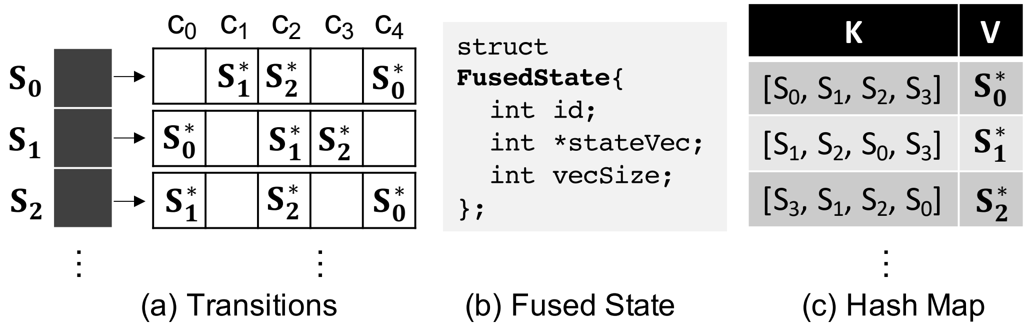
$$V_{next}[i] = trans[V[i]][c], 0 \leq i < N \tag{4.1}$$

In addition, it generates a single transition at the fused FSM level: $\mathbf{S}_{next} = Trans[\mathbf{S}][c]$, where $\mathbf{S}$ and $\mathbf{S}_{next}$ are the fused states corresponding to V and V_{next}, respectively.

- *Fused* Mode. Under this mode, the current state of the fused FSM $\mathbf{S}$ is given. After reading input symbol c, this mode attempts to make a transition at the fused FSM level $\mathbf{S}_{next} = Trans[\mathbf{S}][c]$. If the needed transition information is not available, it switches back to the *basic* mode.

With dynamic path fusion, the state enumeration scheme starts from the *basic* mode, then it switches to the *fused* mode once the transition information $Trans[\mathbf{S}][c]$ becomes available, and switches back to the *basic* mode otherwise.

Data Structures. A key design question in implementing the dynamic path fusion is how to store the transition information $Trans[\mathbf{S}][c]$. A straightforward solution is using a hash map, where the key is a combination of $\mathbf{S}$ and c, and the value is the next fused state $\mathbf{S}_{next}$. While being intuitive, it requires an invocation of a hash function for each fused state transition. Comparing to the default two-dimensional table (Figure 4.1-b), we found the cost of hash-map-based state transitions is about 7X higher. Instead, we employ a *vector of arrays* for storing the transitions of fused states (see Figure 4.9-a). Each time a new fused state is created, a "row" is added to the *vector of arrays.* In theory, if the transitions of an execution are scattered sparsely across many fused states, this data structure may waste space, similar to the two-dimensional table. However, in practice, we found that for a single input, the transitions are often concentrated to a few number of "hot" states, leading to small memory footprints even for FSMs with very large static fused FSMs (more details in Section 4.6).



	C_0	C_1	C_2	C_3	C_4
$\mathbf{S_0}$		$\mathbf{S_1^*}$	$\mathbf{S_2^*}$		$\mathbf{S_0^*}$
$\mathbf{S_1}$	$\mathbf{S_0^*}$		$\mathbf{S_1^*}$	$\mathbf{S_2^*}$	
$\mathbf{S_2}$	$\mathbf{S_1^*}$		$\mathbf{S_2^*}$		$\mathbf{S_0^*}$

(a) Transitions

```
struct
FusedState{
  int id;
  int *stateVec;
  int vecSize;
};
```

(b) Fused State

K	V
$[S_0, S_1, S_2, S_3]$	$\mathbf{S_0^*}$
$[S_1, S_2, S_0, S_3]$	$\mathbf{S_1^*}$
$[S_3, S_1, S_2, S_0]$	$\mathbf{S_2^*}$

(c) Hash Map

Figure 4.9: Data Structures for Dynamic Path Fusion

($\mathbf{S}_i^*$ is a pointer to a fused state; $[S_i, \cdots, S_j]$ is a state vector)

The `FusedState`, as shown in Figure 4.9-b, is defined with both a state `id` and its corresponding state vector to quickly switch back to the *basic* mode once the fused state transition is unavailable. Finally, to find chances of switching from the *basic* mode to the *fused* mode, a *hash map* from state vectors to fused states (see Figure 4.9-c) is maintained. Note that, to reduce memory cost, only pointers to `FusedState` are kept in the vector of arrays and the hash map. Both the sizes of the vector and the hash map equal to M_{dyn}, the number of unique fused states encountered in a single execution.

Example. Figure 4.10 illustrates an example execution with dynamic path fusion using the FSM from Figure 4.3. The thick arrows indicate the switchings between the *basic* mode

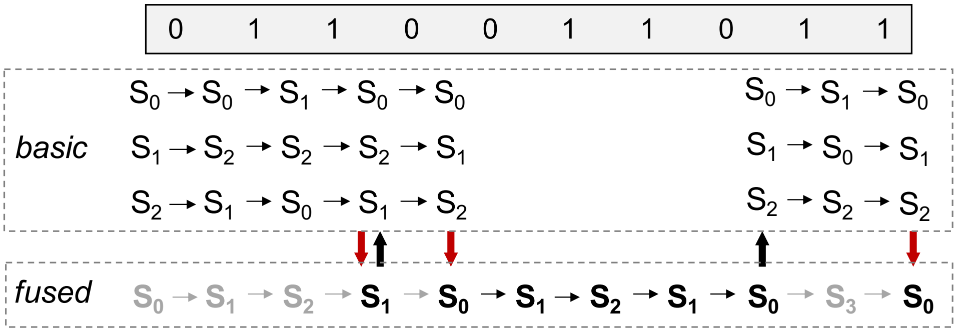


Figure 4.10: Example Execution with Dynamic Path Fusion

and the *fused* mode. The fused states and transitions in gray are generated during the *basic* mode execution. As shown later in Section 4.6, for real-world FSMs and inputs, the executions tend to be dominated by the fused state transitions, resulting in much better efficiency than the basic state enumeration.

Optimization. As mentioned earlier, path fusion is radically different from the path merging optimization (Section 4.2). In fact, they can be combined to further boost the execution efficiency. A straightforward way is separating the execution of state enumeration into two phases: (i) *path merging phase*, and (ii) *path fusing phase*, and apply the two optimizations in the two phases respectively. This design is based on the fact that states of different execution paths tend to merge quickly at the beginning of the execution [83]. Thus, once the number of current states is below a pre-defined threshold τ_n or remains unchanged for τ_l transitions, we move to the second phase and start dynamic path fusion. In cases where the path merging reduces the size of the state vector, the following dynamic path fusion will consume even less memory and make faster switches between the *basic* and *fused* execution modes.

So far, we have presented the path fusion technique which addresses the low execution efficiency of state enumeration for FSMs with limited state convergence. Next, we move to the other FSM parallelization scheme – state speculation, which also suffers from a critical scalability issue when the FSM exhibits limited state convergence.

4.4 Higher-Order Speculation

Unlike state enumeration, speculative FSM parallelization [138, 137, 99, 100] predicts it when the starting state of an input chunk is unknown, then it relies on validations and reprocessing to ensure the correctness. However, as explained in Section 4.2, when the reprocessing path fails to converge quickly with the speculative execution path, the cost would be high. This becomes especially serious when the reprocessing of different chunks is "chained" together, forming a serial bottleneck.

In this section, we address the serial reprocessing issue by introducing the concept of *speculation order*. We show that the existing FSM speculation scheme belongs to *first-order* speculation, and by raising the speculation to higher orders, it is possible to validate and reprocess different input chunks in parallel, without compromising the correctness.

4.4.1 Speculation Order

For easy reference, we formally denote the speculation at the beginning of input chunk *chunk_i* as:

$$\textsc{Spec}(i, S, C) \tag{4.2}$$

where S is the *predicted starting state* and C is the correct starting state, also referred to as the *correctness criterion.* A speculation $\textsc{Spec}(i, S, C)$ can be *validated* by replacing the predicted state S with the correct one C, denoted as:

$$\textsc{Spec}(i, S, C) \xrightarrow{validate} \textsc{Non-Spec}(i, C) \tag{4.3}$$

The validation results in a non-speculative status regarding the starting state at the beginning of input chunk *chunk_i*.

If we refer to the above speculation $\textsc{Spec}(i, S, C)$ as the *first-order* speculation, then we can generalize the concept of speculation to higher-order speculation, recursively:

Definition 7 *The* ***order*** *of speculation* $\textsc{Spec}(i, S, C)$ *is*

- $(k+1)$*-th, if and only if its validation leads to a* k*-th order speculation, denoted as*

  $$\textsc{Spec}^{k+1}(i, S, C) \xrightarrow{validate} \textsc{Spec}^{k}(i, C, C') \tag{4.4}$$

  where C' *is the correctness criterion for* $\textsc{Spec}^{k}(i, C, C')$*;*

- *first order, if and only if its validation makes the starting state non-speculative, denoted as*

  $$\textsc{Spec}^{1}(i, S, C) \xrightarrow{validate} \textsc{Non-Spec}(i, C) \tag{4.5}$$

It is important to note that in Equation 4.4, the predicted state in $\textsc{Spec}^{k}(i, C, C')$ is in fact the correctness criterion from $\textsc{Spec}^{k+1}(i, S, C)$. In another word, the correctness criterion C itself is speculative, adding another "level" of speculation.

Revisiting Existing Speculative FSM Parallelization. Based on the above formalization, it is not hard to find that all the existing speculative FSM parallelization [62, 138, 137, 99, 100], in fact, belongs to *first-order* speculation, as the correctness criteria used in their validations are always non-speculative. As shown earlier in Figure 4.5, the non-speculative correctness criterion is the correct ending state of the prior chunk S_{end_i} ($0 \leq i <$*#chunks* -1), which would not be available until the prior chunk has been validated and properly reprocessed if needed. Therefore, *chunk_i* ($0 < i \leq$*#chunks* -1) has to wait for the correctness criterion to be propagated from the first chunk, whose ending state is non-speculative by default. This essentially creates "a waiting queue" that fundamentally limits the scalability of speculative FSM parallelization. Next, we show that, by raising the speculation to higher orders, we can effectively alleviate this "waiting queue" problem.

4.4.2 Benefits of Higher-Order Speculation

In general, raising speculation order(s) for speculative FSM parallelization could bring benefits in two major aspects.

- ***Earlier & meaningful validation***. To illustrate this benefit, let us reexamine the conventional (first-order) speculation in Figure 4.5, where the validation of *chunk_3* has to wait for the completion of *chunk_2*'s validation, in order to obtain the non-speculative correctness criterion, the correct ending state of *chunk_2*, S_{end_2}. However, as shown in Figure 4.11, if we raise the speculation at *chunk_3* to the 2nd order and use the speculative ending state of *chunk_2*, S'_{end_2}, as the correctness criterion, then we can immediately start its validation and reprocessing, in parallel with the validation and

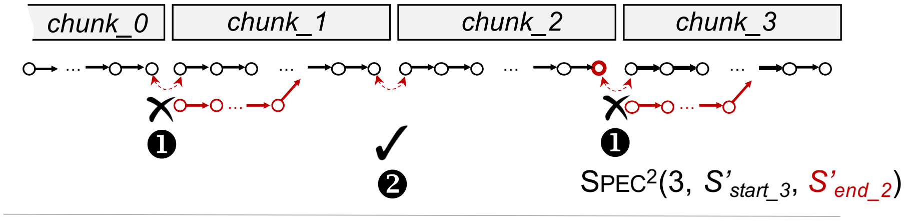


Figure 4.11: Earlier and Meaningful Validation

reprocessing of *chunk_1*. If S'_{end_2} turns out to be the correct ending state of *chunk_2*, like the case shown in the figure, then the reprocessing of *chunk_3* would be valid. In summary, higher-order speculation enables deeper-level speculative parallelism that opens chances for earlier and meaningful validations of later input chunks.

- ***Improved speculation accuracy.*** Besides extra parallelism, the other benefit of higher-order speculation comes from the improved speculation accuracy. Without loss of generality, consider *chunk_2* and *chunk_3* in Figure 4.12, whose starting states are predicted using an existing technique [62, 138, 99], denoted as S'_{start_2} and S'_{start_3}. Statistically speaking (across many inputs), their probabilities of being the correct starting states (i.e., speculation accuracies) are the same. Assume the ending state after speculatively processing *chunk_2* is S'_{end_2}, then the probability that S'_{end_2} is the correct starting state of *chunk_3* should be no less than that of S'_{start_3}; in fact, S'_{end_2} is more likely to be correct starting state of *chunk_3*, thanks to the potential state convergence during the speculative processing of *chunk_2*. That is, even if S'_{start_2} is incorrect, its execution path may converge with the correct path during the processing of *chunk_2*, resulting in a correct ending state (i.e., $S'_{end\_2} = S_{end\_2}$).

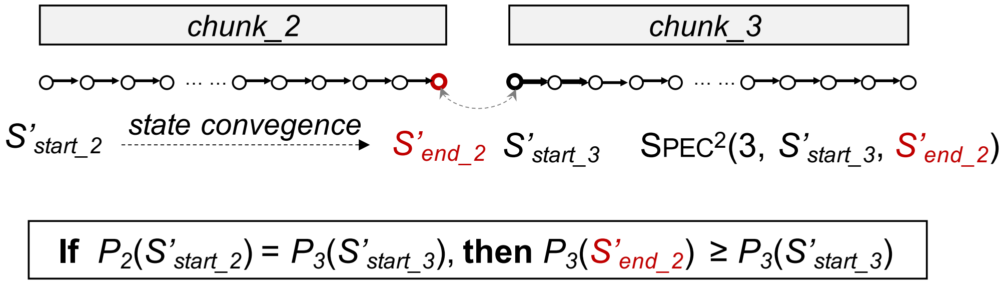

chunk_2
chunk_3
S'start_2
state convegence
S'end_2
S'start_3
SPEC²(3, S'start_3, S'end_2)
If P2(S'start_2) = P3(S'start_3), then P3(S'end_2) ≥ P3(S'start_3)

Figure 4.12: Improved Speculation Accuracy

($P_i(S)$: the probability that S is the correct starting state of *chunk_i)*

When the speculation of *chunk_3* is second order, as shown in Figure 4.12, and using S'_{end_2} (the correctness criterion) to validate (replace) S'_{start_3} as the starting state of *chunk_3*, the speculation accuracy can potentially increase.

To take the above benefits from the higher-order speculation, we next present a new speculative FSM parallelization strategy, referred to as *iterative speculation.*

4.4.3 Iterative Speculation

Unlike existing speculative FSM parallelization [62, 138, 137, 99, 100], iterative speculation organizes the whole processing into a series of iterations. Algorithm 7 summarizes its basic ideas. First, it initializes the starting state for each chunk, just like the conventional speculation, except that it also initializes an `active` flag for each thread (Line 6). After initialization, the algorithm enters into a series of iterations. In each iteration, *chunk_i* is processed only if `active[i]` is `true` (Line 10-11). Note the processing checks its path convergence with that of the last processing of *chunk_i*, that is, path merging is applied. The value of flag `active[i]` is set at the end of the prior iteration (see Line 14-19): if the starting state s_i in the last iteration is different from the ending state of the prior chunk e_{i-1}, `active[i]` is set to `true`. The algorithm terminates once all `active` flags become false.

Algorithm 7 Iterative Speculation

```
 1: for each chunk_i  do /* initialization */
 2:     if $i == 0$ then
 3:         $s_i = s_0$
 4:     else
 5:         $s_i$ = predict($i$)
 6:     active[$i$] = true /* all threads are active initially */
 7:
 8: while some active[$i$] is true do /* iterations */
 9:     for each chunk_i do in parallel
10:         if active[$i$] == true then
11:             $e_i$ = process(chunk_i, $s_i$) /* w/ path merging */
12:     barrier() /* synchronize */
13:
14:     for each chunk_i do in parallel
15:         if $s_i \neq e_{i-1}$ then /* validation */
16:             $s_i = e_{i-1}$
17:             active[$i$] = true
18:         else
19:             active[$i$] = false
```

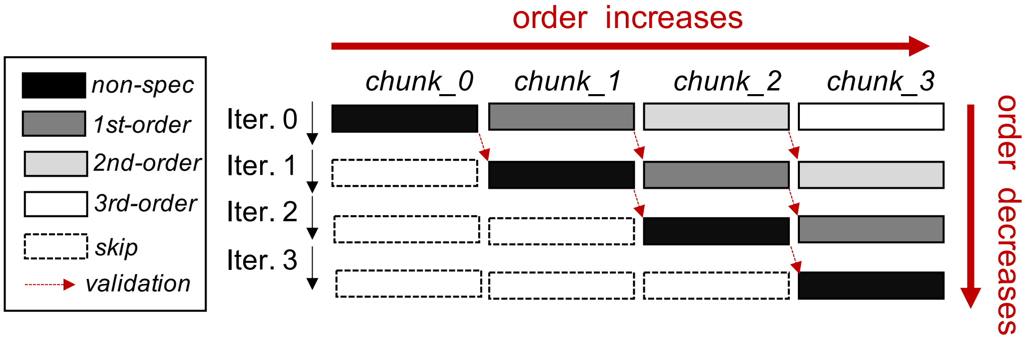


Figure 4.13: Illustration of Iterative Speculation

(for clarity, all validations fail and no path merging occurred)

Next, we explain how higher-order speculation is reflected in the above algorithm and why the algorithm in fact always terminates in *#chunks* iterations. As illustrated in Figure 4.13, we use different grayscale levels to represent different orders of speculation, with the darkest one used for the non-speculative processing. Initially, we assume that the chunks are assigned with increasing speculation orders: *chunk_i* is in *i*-th order speculation. Then, during each iteration, the latest speculation of each chunk is validated using the latest ending state from the prior chuck. As a result, its speculation order is reduced by one. Once the speculation order of a chunk becomes 0th (i.e., non-speculative), it will stay inactive, because the ending states of all its prior chunks are already fixed. Obviously, the chunk with the highest speculation order (the last chunk) determines the maximum iteration number, thus the algorithm takes exactly *#chunks* iterations to terminate.

Note that Figure 4.13, in fact, demonstrates the worst-case scenario of iterative speculation: all validations fail and no path merging occurred in any case. Consequently, iterative speculation would be equivalent to sequential processing; the same applies to the conventional speculation. In practice, as shown later in Section 4.6, iterative speculation substantially improves the performance, thanks to the improved speculation accuracy and successful earlier validations (Section 4.4.2).

So far, we have introduced path fusion and higher-order speculation to address the scalability limitations in the two basic FSM parallelization schemes, respectively. Still, an open question is which scheme works the best for a given FSM and its inputs. We address the scheme selection next.

4.5 Parallelization Scheme Selection

Including the two basic schemes, we have discussed five FSM parallelization schemes in total:

- `B-Enum`: basic state enumeration
- `B-Spec`: basic state speculation
- `S-Fusion`: state enumeration with static path fusion
- `D-Fusion`: state enumeration with dynamic path fusion
- `H-Spec`: higher-order (iterative) speculation

We also refer to the last three as *augmented schemes.* Which scheme works best depends on the characteristics of the FSM and its inputs. Based on the design of the schemes, we focus our discussion on five key properties of the FSM and its inputs: (i) number of states, (ii) state convergence rate, (iii) speculation accuracy, (iv) the feasibility to generate a static fused FSM under a memory budget, and (v) the skewness of state vector distribution. The second and last properties are defined below:

Definition 8 *State convergence rate conv*(l) *is the number of unique current states after consuming* l *input symbols in an enumerative execution.*

Definition 9 *Skewness of state vector distribution uniq*(l) *is the number of unique state vectors encountered during the processing of* l *input symbols in an enumerative execution.*

Note that our goal is NOT to precisely model the execution time of each scheme, which could be extremely challenging given the diverse and complex FSM transition behaviors, not to mention external factors like the design of state predictor and the underlying architecture. Instead, we intend to *qualitatively* reason about the conditions under which each scheme would work well in general. This qualitative performance reasoning will provide insights for us to guide the scheme selection.

Table 4.1 summarizes the suitability of each scheme under different value combinations of three out of the five properties, where 'H' means the value is high and 'L' means the opposite. For the other two properties, we assume the number of states is 'L' and it is feasible to generate a static fused FSM, in order to avoid exploding the size of the table.

There are several key points worth to mention. First, under 'HHL' and 'HHH', all schemes would work very well, except for `D-Fusion` which does not run efficiently when the number of unique state vectors is high. Also note that the speculation schemes `B-Spec` and `H-Spec` work slightly better than the enumerative schemes, because the latter require two passes of processing (see Section 4.3). Second, `S-Fusion` works well across all situations in the table, thanks to its offline fused FSM generation. However, this is under the assumption

Table 4.1: Qualitative Comparison of Parallelization Schemes

(C: state conv. rate, S: spec. accuracy, K: skewness of state vectors)

<table>
<tr><th></th><th colspan="2">Basic Schemes</th><th colspan="3">Augmented Schemes</th></tr>
<tr><th>C | S | K</th><th>B-Enum</th><th>B-Spec</th><th>S-Fusion</th><th>D-Fusion</th><th>H-Spec</th></tr>
<tr><td>H | H | H</td><td>☆☆☆☆</td><td>☆☆☆☆☆</td><td>☆☆☆☆</td><td>☆☆☆☆</td><td>☆☆☆☆☆</td></tr>
<tr><td>H | H | L</td><td>☆☆☆☆</td><td>☆☆☆☆☆</td><td>☆☆☆☆</td><td></td><td>☆☆☆☆☆</td></tr>
<tr><td>H | L | H</td><td>☆☆☆☆</td><td>☆☆☆</td><td>☆☆☆☆</td><td>☆☆☆☆</td><td>☆☆☆☆</td></tr>
<tr><td>H | L | L</td><td>☆☆☆☆</td><td>☆☆☆</td><td>☆☆☆☆</td><td></td><td>☆☆☆☆</td></tr>
<tr><td>L | H | H</td><td></td><td>☆☆</td><td>☆☆☆☆</td><td>☆☆</td><td>☆☆☆</td></tr>
<tr><td>L | H | L</td><td></td><td>☆☆</td><td>☆☆☆☆</td><td></td><td>☆☆☆</td></tr>
<tr><td>L | L | H</td><td></td><td></td><td>☆☆☆☆</td><td>☆☆</td><td>☆☆</td></tr>
<tr><td>L | L | L</td><td></td><td></td><td>☆☆☆☆</td><td></td><td>☆☆</td></tr>
</table>

that the static fused FSM is feasible to generate under the memory budget. Third, unlike `B-Spec`, `H-Spec` still works well under low speculation accuracy as long as the state convergence rate is high (i.e., 'HLL' and 'HLH'), because state convergence can improve the rate of successful validations (see Figure 4.12). In addition, even with high state convergence rate, if the number of states is high (not shown in the table), both `B-Enum` and `D-Fusion` will suffer from higher overhead at the early stage of the processing (i.e., before many states converge).

Overall, no scheme works well in all situations, however, Table 4.1 clearly shows that the augmented schemes extend the scope of effective FSM parallelization towards the lower rows of the table – the more challenging situations.

Based on the above qualitative analyses, we propose a series of heuristics to guide the FSM parallelization scheme selection, summarized by the decision tree in Figure 4.14. The heuristics start from the best parallelization scenario checking – whether the speculation accuracy is high (greater than a threshold). If so, either `B-Spec` or `H-Spec` will be selected.

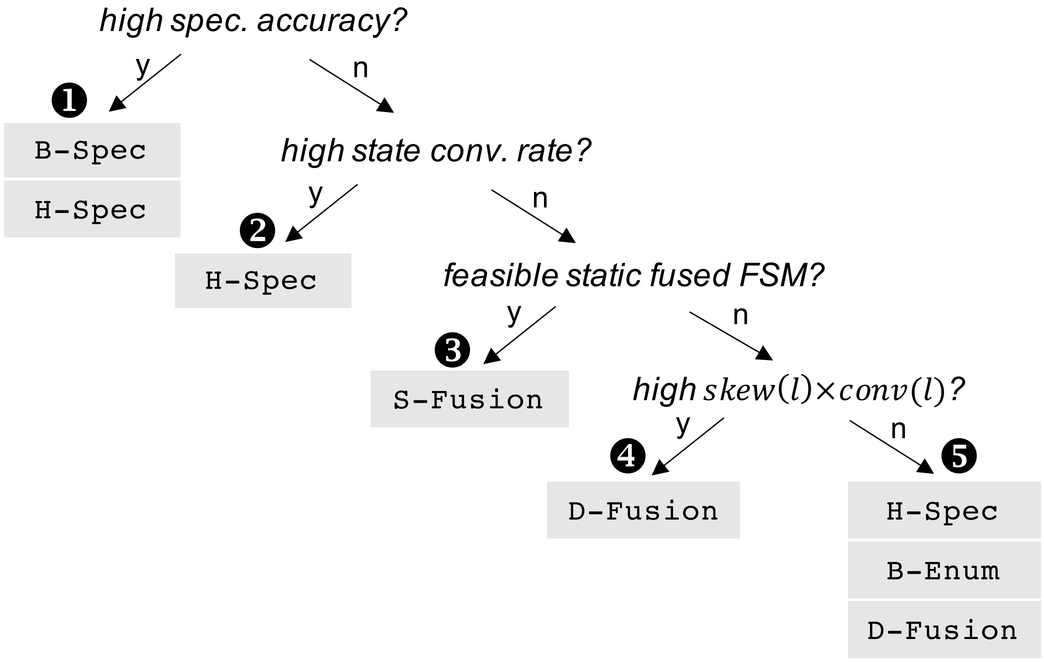


Figure 4.14: Decision Tree for Scheme Selection

They then examine the state convergence rate. If it is high, they further check the number of states. For FSMs with a large number of states, they can only choose `H-Spec`; otherwise, they can pick either `H-Spec` or `B-Enum`. On the other hand, without high convergence rate, they will try to generate a static fused FSM under the given memory budget. If that succeeds, they will use `S-Fusion`; otherwise, they will further consider `D-Fusion` by checking the skewness of state vector distribution. If it is highly skewed, they will surely pick `D-Fusion`; otherwise, it becomes the worst case scenario – the speedup would be low anyway. Three options are provided in this case, which are `H-Spec`, `B-Enum`, and `D-Fusion`.

Note that it is practically very different to select the scheme online for a given FSM and a particular input, because the cost of collecting the properties easily outweighs (or compromises) the benefits of parallelization. Instead, we target the scheme selection for the given FSM and a group of its inputs. That is, a few inputs are (randomly) selected for collecting the properties offline, based on which the scheme is selected and used online.

4.6 Evaluation

In this section, we evaluate our proposed techniques in detail, with a focus on the extra speedups they bring over the two basic parallelization schemes.

4.6.1 Methodology

We implemented the five FSM parallelization schemes that are summarized in Section 4.5, in C++ language and used `Pthread` for their parallel executions. Then, we integrated these five schemes along with the scheme selector into one multi-scheme parallelization framework, called `BoostFSM`.

Benchmarks. Table 4.2 lists the FSM benchmarks used in our evaluation with their relevant properties. The 16 benchmarks are collected from the Snort library [108], a pool of signatures in PCRE format used by the state-of-the-art Network Intrusion Detection Systems (NIDS). They are chosen to provide a good coverage of the diverse properties of FSMs and their inputs. The second column in the table shows the number of states, which ranges from 17 to 4736 (one of the largest). The next two columns report the state convergence rates for 10^3 and 10^6 input symbols, respectively (smaller is better). The fifth column shows the speculation accuracy, ranging from 0% to 100%. The next column tells the feasibilities to generate static fused FSMs (under the memory budget of 1GB/FSM, or equivalently 10^6 fused states): for five of the benchmarks, the generation is feasible. The last column reports the skewness of state vector distribution – the number of unique state vectors observed after consuming 10^5 input symbols.

The inputs to the FSMs are 20 traces of real-world network traffics collected from a Linux server using `tcpdump`. Each trace consists of 4×10^8 symbols (i.e., 400MB). The total size of inputs is 8GB. For each FSM, five traces are randomly selected to collect the properties reported in Table 4.2. More specifically, we ran `B-Enum` on the training inputs up to 10^3 and 10^6 symbols, respectively, to collect the state convergence rates. The speculation accuracies are the average of 100 times of speculation over the first 10^5 symbols of each trace. For the skewness of state vectors, we ran `B-Enum` (with path merging) over the first 1K symbols, then recorded the unique number of state vectors for the next 9K symbols of each input.

Platform. All experiments ran on a 64-core machine equipped with an Xeon Phi 7210 processor (1.3GHz/core) and 96GB RAM. The machine runs Linux 3.10.0. All programs were compiled by GCC 4.8.5 with the "`O3`" optimization flag. The timing results reported are the average of three repetitive runs over 20 inputs and we do not report 95% confidence intervals of the average when the variation is not significant.

In the following, we first evaluate the augmented schemes and the scheme selection, then, reports the scalability of each scheme over different number of cores and input sizes.

Table 4.2: FSM Benchmarks

($conv(l)$: state convergence rate; Static: feasibility to generate

a static fused FSM; $uniq(l)$: skewness of state vectors)

FSM	#States	$conv(10^3)$	$conv(10^6)$	SpecAcc	Static	$uniq(10^5)$
F1	207	2.0	2.0	0%	Yes	6.1
F2	17	2.0	2.0	5%	Yes	5.9
F3	193	46.7	20.1	1%	No	130.3
F17	507	2.0	2.0	0%	No	1040.1
F5	31	5.0	5.0	0%	Yes	4.2
F6	31	5.7	1.0	5%	No	4602.5
F7	53	3.8	1.0	9%	No	529.7
F8	22	7.3	1.0	5%	No	1948.8
F18	4736	1.0	1.0	100%	No	78.7
F10	30	1.9	1.6	76%	Yes	7.2
F11	34	7.9	1.0	4%	No	3079.3
F12	65	2.0	2.0	100%	Yes	4.2
F13	145	5.0	5.0	0%	No	8.9
F14	1045	2.0	2.0	0%	No	3.1
F15	2012	2.0	2.0	0%	No	5.3
F16	1179	2.0	2.0	34%	No	7.7

4.6.2 Performance

Table 4.3 reports the speedups of different FSM parallelization schemes using 64 cores over the sequential FSM execution. The second column shows the execution time of sequential FSM execution. Since the inputs to different FSMs are of the same size, their execution time only varies slightly. This also indicates that the number of states, in practice, have limited impacts on the execution time, as the frequently accessed state transitions often well fit into CPU caches. In the following, we first evaluate each of the augmented schemes separately, then examine the effectiveness of the scheme selection.

Static Path Fusion. First, for those benchmarks whose static fused FSMs can be generated (F1-2, F5, F10, F12), `S-Fusion` significantly raises the speedups comparing to `B-Enum`,

Table 4.3: Speedup Comparison

(Baseline: sequential execution; #threads: 64; input size: 4×10^8)

		Basic Schemes		Augmented Schemes			
FSM	Seq(s)	B-Enum	B-Spec	S-Fusion	D-Fusion	H-Spec	BoostFSM
F1	7.47	12.9	0.6	**31.2**	23.6	17.6	**31.2**
F2	7.45	13.7	1.9	**30.9**	25.1	17.8	**30.9**
F3	7.37	7.3	1.9	-	8.5	**13.0**	7.3
F17	7.53	**12.9**	0.5	-	3.6	8.7	**12.9**
F5	7.43	11.1	0.6	**31.1**	25.5	13.9	**31.1**
F6	7.43	28.5	22.9	-	13.1	**30.1**	**30.1**
F7	7.49	29.8	29.7	-	25.5	**32.7**	**32.7**
F8	7.48	29.1	20	-	19.6	**32.6**	**32.6**
F18	7.51	19.3	**37.2**	-	17.9	36.5	**37.2**
F10	7.39	14.2	1.4	**30.8**	25.1	18.3	**30.8**
F11	7.57	26.9	21.6	-	16.1	**32.6**	**32.6**
F12	7.46	13.0	**39.8**	30.9	24.9	39.2	**39.8**
F13	7.44	11.6	0.6	-	**23.9**	10.4	**23.9**
F14	7.40	12.2	0.6	-	**22.5**	16.7	**22.5**
F15	7.35	13.0	0.6	-	**23.4**	17.1	**23.4**
F16	7.46	12.7	0.9	-	**23.5**	11.2	**23.5**
Geo	7.45	15.4	3.1	31.0	18.3	19.5	25.8

from 12.9X to 31.0X on average, thanks to its path fusion and offline fused FSM generation. According to Table 4.2, after consuming 10^6 symbols, there are still 2.3 paths left on average for these FSMs. `S-Fusion` completely eliminates such overhead. On the other hand, for those FSMs whose static fused FSMs are too large to generate (i.e., over the memory budget), static path fusion cannot help. Table 4.4 reports the sizes of the statically generated fused FSMs (column "Static Fusion / #FS") and the time taken for fused FSM generations (column "Static Fusion / Time"). The memory budget is set to 1GB, equivalent to 10^6 fused states. With a larger sample set, as reported earlier in Figure 4.8, the sizes of fused FSMs are often below or close to N^2, where N is the number of states in the original FSM.

Dynamic Path Fusion. `D-Fusion` works around the memory limitation of `S-Fusion`

Table 4.4: Statistics of Path Fusion

(#FS: num. of fused states; Time: generation time; #SW: num. of switches)

	Static Fusion		Dyn Fusion			Static Fusion		Dyn Fusion	
FSM	#FS	Time	#FS	#SW	FSM	#FS	Time	#FS	#SW
F1	19899	37.1	14	1223	F18	-	-	4736	0
F2	173	0.06	7	1133	F10	2876	1.25	11	1311
F3	-	-	116	10465	F11	-	-	246	21121
F17	-	-	1209	161796	F12	6655	4.80	5	897
F5	486	0.22	5	961	F13	-	-	10	1339
F6	-	-	149	10981	F14	-	-	4	641
F7	-	-	57	830	F15	-	-	6	1025
F8	-	-	131	17129	F16	-	-	8	1245

by generating a partial fused FSM that only captures one execution. In Table 4.4, column "Dyn Fusion / #FS" reports the numbers of fused states actually generated during dynamic path fusion. Interestingly, the numbers are not only less than those in the static fused FSMs, but also often less than those in the original FSMs (F1-3, F5, F10, F12-16). In terms of performance, as reported in Table 4.3, the speedups of `D-Fusion` vary a lot across benchmarks, ranging from 3.6X to 25.5X. In fact, for some FSMs, `D-Fusion` performs worse than `B-Fusion` (F17, F6-8, F18, F11). As explained in Section 4.3.3, the effectiveness of `D-Fusion` highly depends on the number of unique fused states encountered. In Table 4.4, column "Dyn Fusion / #FS" reports these numbers, which roughly align with the speedups of `D-Fusion`. Note that F18 is a special case in that the state vector size quickly drops to one during the path merging phase, so when it enters into the path fusion phase, its fused FSM is just the original FSM. The next column in Table 4.4 ("Dyn Fusion / #SW") reports the number of switches between the *basic* and *fused* modes. The switches are the main source of overhead in dynamic path fusion due to their accesses to the hash tables (see Section 4.3.3).

Higher-Order Speculation. At last, we compare H-Spec with B-Spec. As shown in Table 4.3, H-Spec significantly boosts the speedups from 3.1X to 19.5X on average. More importantly, its improvements are consistent across all benchmarks, except for F12 and F18, in which cases, both H-Spec and B-Spec work very well (over 36X speedups), with B-Spec showing marginally better speedups. These results are consistent with our earlier discussion, that is, H-Spec performs no worse than B-Spec in principle. The improvements come from the two benefits of higher-order speculation: earlier and meaningful validations and improved speculation accuracy. Table 4.5 reports both the speculation accuracies of B-Spec and H-Spec. The initial speculation accuracy of H-Spec (Iteration-1) is the same with or similar to that of B-Spec (23% vs. 24% on average). But, over iterations, the speculation accuracies of H-Spec get improved quickly. By the third iteration, all benchmarks reach 100% speculation accuracy. On average, it takes 2.1 iterations for H-Spec to complete the processing.

In summary, the augmented schemes substantially boost the performance over the basic schemes. However, their speedups vary a lot across benchmarks, and one of them, D-Fusion, may perform even worse than its basic scheme. As shown in Table 4.3, the best schemes (in bolded font) for the benchmarks are scattered across the five parallelization schemes, which confirms the needs of scheme selection.

Scheme Selection. The FSM properties used by the scheme selector are already shown in Table 4.2. Following the heuristics in Section 4.5, the selector first checks the speculation accuracy against the threshold (98%) and finds that only benchmarks F18 and F12 meet the requirements. Thus, it selects B-Spec for these two FSMs. Then, it checks if the

Table 4.5: Speculation Accuracies

		Higher-Order Speculation			
FSM	B-Spec	Iteration-1	Iteration-2	Iteration-3	#Iterations
F1	0%	0%	100%	-	2.0
F2	61%	61%	100%	-	1.9
F3	62%	62%	97%	100%	2.6
F5	0%	0%	98%	100%	2.4
F6	5%	5%	100%	-	2.0
F7	9%	9%	100%	-	2.0
F8	5%	5%	100%	-	2.0
F10	0%	0%	100%	-	2.0
F11	5%	5%	100%	-	2.0
F12	100%	100%	-	-	1.0
F13	0%	0%	2%	100%	3.0
F14	0%	0%	100%	-	2.0
F15	0%	0%	98%	100%	2.1
F16	33%	33%	57%	100%	3.0
F17	2%	2%	57%	100%	3.0
F18	100%	100%	-	-	1.0
Avg	24%	23%	86%	100%	2.1

state convergence rate $conv(10^6)$ reaches one. If so, it chooses `H-Spec`, which happens to benchmarks F6, F7, and F8. For the remaining benchmarks, the selector further examines the feasibility to generate a static fused FSM. It obtains positive answers for benchmarks F1, F2, F5, and F10, thus assigns `S-Fusion` to them. Finally, the selector checks the skewness of the state vector distribution against the threshold (15) which shows benchmarks F13, F14, F15, F16 all satisfy the requirement. So, these FSMs are assigned with `D-Fusion`. At this point, there are still two benchmarks left: F3 an F17. Since our selector does not further examine the properties based on more specific values, by default, it selects `B-Enum`. The last column of Table 4.3 shows the choices of the scheme selection. Out of 16 cases, it only fails to pick the best scheme for F3. The failure is simply due to the fact that the heuristics stops reasoning about the performance at more fined-grained levels, which can

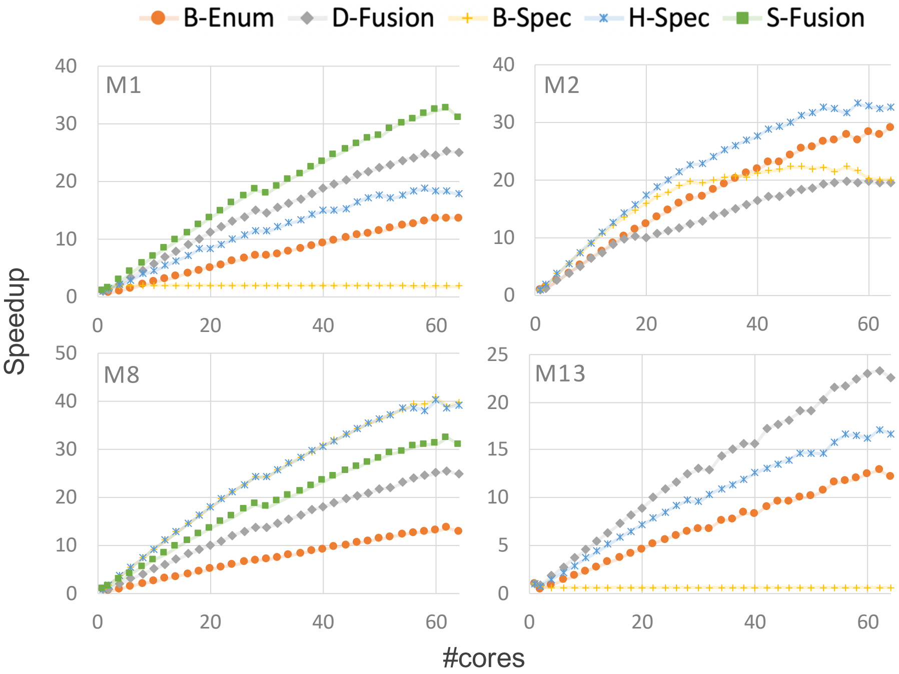


Figure 4.15: Scalability of Representative Cases

(F2, F8, F12, and F14 represent the best cases of `S-Fusion`, `H-Spec`, `B-Spec`, and `D-Fusion`, respectively)

be improved with more detailed performance modeling.

4.6.3 Scalability

In this section, we examine the scalability of different schemes in terms of both the number of cores and the input size.

Varying Number of Cores. Due to the space limit, we report the scalabilities over different number of cores for a subset of representative benchmarks, as shown in Figure 4.15. They are chosen to represent the cases where `S-Fusion` performs the best (F2), `H-Spec`

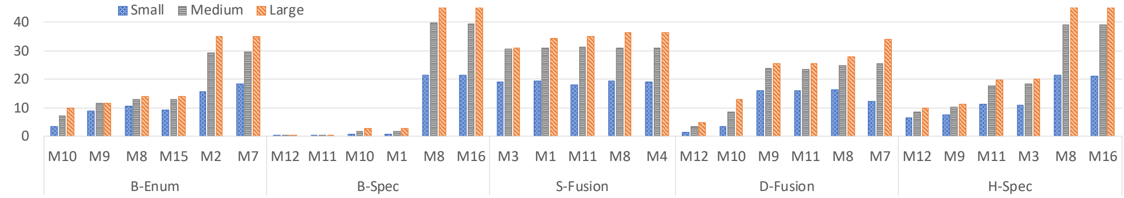


Figure 4.16: Speedups of Each Scheme under Different Input Sizes

(Due to space limits, for each scheme, we ranked the benchmarks by the speedup of Large input, then selected the top two,

the middle two, and the bottom two benchmarks to show, except for `S-Fusion`, for which all feasible ones are shown.)

performs the best (F8), `B-Spec` and `H-Spec` performs the best (F12), and `D-Fusion` performs the best (F14), respectively. In general, when the desired properties are present (see Table 4.1 and Figure 4.14), all the five schemes can scale well. On the other hand, when the properties are not ideal, some schemes suffer from worse scalabilities than others. For example, when the speculation accuracy drops and the state convergence rate is low, `B-Spec` scales poorly, and may even run slower than the serial execution (see the curves of `B-Spec` in subfigures F2, F8, and F14), due to its serial validations [100]. The other scheme that may not scale well is `D-Fusion`, as shown in subfigure F8. This is because when the input is partitioned into smaller chunks, the number of unique state vectors may not decrease proportionally, thus the overhead becomes relatively higher.

Varying Input Size. Figure 4.16 reports the speedups of the five schemes under different input sizes: small (1×10^8), medium (4×10^8), and large (8×10^8). Overall, there are clear trends that the speedups get improved as the input sizes increase for all the parallelization schemes. The trends reflect the effects of Amdhal's law. In our context, the sequential components include thread creation (64 threads), thread synchronization (validations in speculative schemes, and correct path selections in enumerative schemes), and some I/O operations (printing out the matches). In addition, for `H-Spec`, an extra benefit could be the potentially better convergence with longer input chunk (see Figure 4.12). For `D-Fusion`, as the input chunk becomes longer, the number of switches between *basic* and *fused* modes may become relatively less (happened to F7).

4.7 Summary

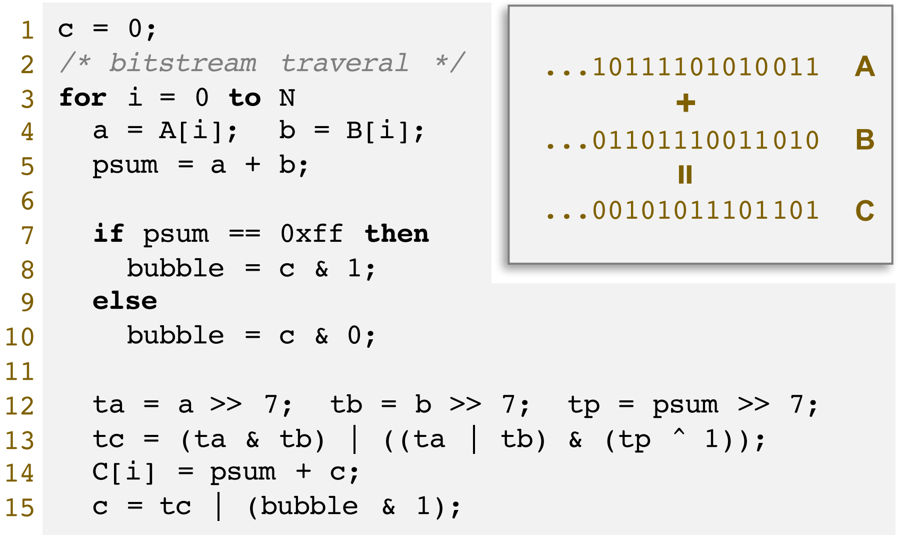
In this work, we addressed the fundamental scalability issues inherited in the two basic FSM parallelization schemes: the cost of maintaining multiple execution paths in enumerative parallelization and the sequential chunk-by-chunk validations in speculative parallelization. For the former, we proposed the technique of *path fusion*, which can fuse different execution paths into a single one, either statically or dynamically. For the latter, we introduced the concept of *higher-order speculation* which allows a speculated state to be validated speculatively. For practical uses, we also discussed the scenarios where each scheme works the best and proposed a set of heuristics to help users select the parallelization scheme. Finally, we evaluated the proposed techniques with FSMs drawn from real-world applications and of diverse characteristics. Our results showed that the proposed techniques can substantially raise the scalabilities of both parallelization schemes.

Chapter 5

Challenging Sequential Bitstream Processing via Principled Bitwise Speculation

5.1 Introduction

Bitstream processing manipulates binary values with bitwise operators (e.g., logical XOR and shift <<) over long sequences of bits. It plays critical roles in many important applications for better performance or higher space efficiency, such as bitmap indexing [68], pattern matching [15], parsing [75, 41, 76], image compression [16, 136], and voice decoding [87]. For example, in bitstream-based text pattern matching [15], a text stream is first transposed into a set of bitstreams, then the text patterns are searched with bitwise manipulations. Thanks to the high efficiency of bitwise operations and bitwise parallelism, the bitwise text



```
c = 0;
/* bitstream traveral */
for i = 0 to N
  a = A[i];   b = B[i];
  psum = a + b;

  if psum == 0xff then
    bubble = c & 1;
  else
    bubble = c & 0;

  ta = a >> 7;  tb = b >> 7;  tp = psum >> 7;
  tc = (ta & tb) | ((ta | tb) & (tp ^ 1));
  C[i] = psum + c;
  c = tc | (bubble & 1);
```

Figure 5.1: Bitstream Processing Example (`LBSAdd` [14]).

pattern matching shows significant performance improvements over the conventional "one character at a time" pattern matching schemes. Similar treatments have also been applied to semi-structured data (e.g., XML and JSON) to accelerate querying in document-based data stores [75, 76].

Despite the benefits, a fundamental challenge arises when the processing of the current bits depends on the processing results of prior bits over the course of bitstream processing, referred to as *bitstream-carried dependences*, a special case of loop-carried dependence. As a result to the dependences, the entire bitstream(s) have to be traversed in serial, seriously limiting the scalability. Unfortunately, such bitstream-carried dependences can be easily introduced with commonly used bitwise and non-bitwise operators, such as a left shift operator `<<` that defines the current bit with a bit to the right and a arithmetic addition `+` that may produce a carry propagating over the calculations of the following bits.

Figure 5.1 shows an example bitstream program called *long bitstream addition* (`LBSAdd` [14]). It adds two arbitrarily long bitstreams and put the result into a new bitstream. Note that, rather than adding the two streams bit-by-bit, this program leverages bitwise parallelism to perform byte-level addition [1], which can significantly improve the efficiency. However, the inherent dependences regarding the carry remains through out the entire bitstream processing (more discussions later).

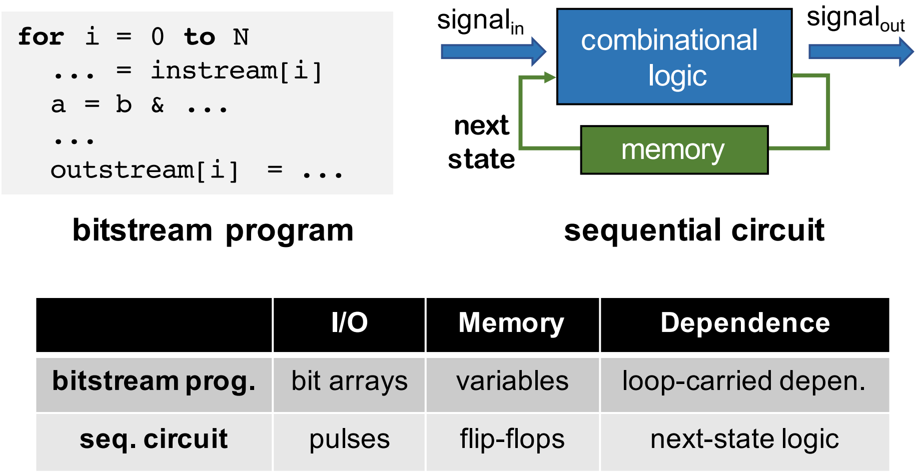
State of The Art. Existing efforts in optimizing the bitstream processing mainly focus on fine-grained vectorization with SIMD intrinsics (e.g., SSE2 and AVX512) [75, 41, 76, 15]. In spite of the performance benefits, there are limitations in the existing solutions that hinder the productivity and efficiency of bitstream processing. First, coding with low-level SIMD intrinsics is notoriously difficult. The situation is worsen in the presence of bitstream-carried dependences. Take the long bitstream addition as an example, because the SIMD intrinsics adds numbers SIMD lane-wise, programmers have to manually resolve the potential carries across SIMD lanes [14], further complicating the SIMD programming. Second, fine-grained dependence handling techniques cannot be naturally extended to enable coarse-grained parallelism, that is, partitioning the bitstream(s) across CPU cores, where the size of a bitstream partition goes far beyond of the SIMD width, making the dependence handling a daunting task. For example, existing (linear) long bitstream addition can add up to 4096 bits [14]. In sum, the existing bitstream processing with fine-grained parallelism heavily relies on programmers and fails to exploit the coarser-grained parallelism in the presence of dependences, fundamentally restricting their scalability.

[1] Larger granularities (like `int` or `long`) can also be used. Without loss of generality, we use byte for easier illustration as a running example.

Overview of This Work. Complementary to the prior efforts, this work challenges the sequential bitstream processing with an automatic approach that enables coarse-grained speculative parallelism for both non-SIMD and SIMD bitstream programs, referred to as *principled bitwise speculation* (PBS). The basic idea of PBS is inspired by an analogy that compares bitstream programs to sequential circuits in hardware (see Figure 5.2), both of which transform binary sequences (*bits* [2] versus *pulses*). The memory in sequential circuits resembles the loop-carried dependences in bitstream programs, despite that the latter are implicitly imposed by the program structures. These close correspondences motivate us to model the dependences in bitstream programs with finite-state machines (FSMs), a basic way to model sequential circuits. Note that, this modeling is often impossible for general programs whose computations can exceed the capability of FSMs. To facilitate the modeling and minimize the sizes of FSMs, PBS leverages a series of static analyses to reason about the minimum set of dependent bits in the bitstream programs. With the dependent bits, PBS constructs the FSM by treating the value combinations of dependent bits as *states* and examining different input-output pairs to reveal the *state transitions.* This reverses the FSM-to-truth table process in the sequential circuit design. For cases where the FSM is too large to construct, PBS offers *partial* or *virtual* FSM constructions. The former consists of only "hot transitions" that are frequently visited; While the latter bypasses the FSM generation, relying on the bitstream program to mimic the FSM state transitions on the fly.

A key benefit from FSM-based bitstream program modeling is that the possibility

[2] For the lack of supports for bit arrays, programmers often use unsigned integer or char arrays to store bitstreams.



```
for i = 0 to N
  ... = instream[i]
  a = b & ...
  ...
  outstream[i]  = ...
```

	I/O	Memory	Dependence
bitstream prog.	bit arrays	variables	loop-carried depen.
seq. circuit	pulses	flip-flops	next-state logic

Figure 5.2: Bitstream Programs v.s. Sequential Circuits.

of adopting existing speculative FSM parallelization [83, 138, 137, 99, 59] to bitstream processing. By leveraging the state convergence properties of FSMs, PBS can effectively predict future values of dependent bits, thus bringing speculative parallelism to bitstream processing. In cases the predication fails, PBS offers a fast recovery method that directly “rectifies” the wrong outputs based on bitwise logic, rather than reprocessing the inputs, which minimizes the mis-speculation costs. Besides prediction, we also observe that, in some cases, the constructed FSM runs more efficiently than the bitstream program, making itself an optimization to the original program. In this way, even the serial bitstream processing can get performance improvement.

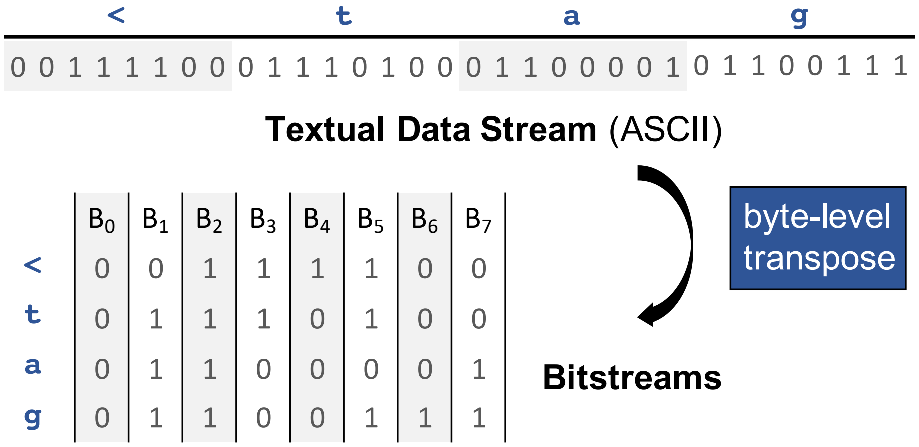
We prototyped PBS on the latest LLVM infrastructure and evaluated it with a set of bitstream processing kernels extracted from real-world applications, covering semi-structured data processing, text pattern matching, and multimedia processing. Our results show that PBS can precisely identify the dependent bits. With speculative bitstream processing, PBS brings up to 60X speedup on a 64-core machine. To demonstrate the end-to-end benefits, we also apply PBS to a state-of-the-art regular expression engine, called `icgrep` [15]. Results show that, with PBS, `icgrep` can generate data-parallel bitstream kernels to effectively leverage all the CPU cores, yielding over 20X end-to-end speedups on a 64-core machine.

5.2 Background

This section introduces the basic ideas in bitstream processing, including the dependences that the computations may carry.

Bitstream Processing. Informally, bitwise computations are computations involving bitwise operators. Commonly used bitwise operators include logical operators (e.g., `&`, `|`, `^`, `~`), shift operators (e.g., `<<`, `>>`, and `>>>`), and some specialized operators (e.g., population count `popcnt` and count leading or trailing zero `CLZ/CTZ`). Correspondingly, there are SIMD versions of these operators provided as low-level intrinsics, from instruction set extensions, such as SSE2, AVX2, and AVX512. For example, `_mm256_and_si256(s1, s2)` from AVX2 performs AND operation between 256-bit vectors.

In many applications, the inputs to bitwise computations are long binary sequences or *bitstreams*, such as, an audio record in multimedia processing [16, 136, 87], or a piece of encrypted file in cryptography [20]. In fact, even textual data streams can be converted into bitstreams to take advantages of SIMD intrinsics and bitwise parallelism. The idea is illustrated in Figure 5.3. Each byte of the textual data stream is transposed into



	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
<	0	0	1	1	1	1	0	0
t	0	1	1	1	0	1	0	0
a	0	1	1	0	0	0	0	1
g	0	1	1	0	0	1	1	1

Figure 5.3: Text Stream to Bitstreams Conversion [75, 76, 15, 41].

eight individual bits stored in eight separated bitstreams. This textual-stream-to-bitstream transposition brings bitstream processing to many applications conventionally manipulating textual data streams, such as intrusion detection over network traffic [15] and data analytics over semi-structured data streams, like XML and JSON [75, 76]. During the processing, the input bitstreams are often transformed multiple times before they are eventually consumed. Figure 5.4 shows the seven phases of bitstream transformations in XML parsing [76], where each transformation generates a new bitstream (e.g., L_0, and E_0). Note that the first three bitstreams (i.e., C_0, C_1, and C_2) are generated from the *base bitstreams* – B_0 to B_7 (see Figure 5.3). In general, we refer to these different bitstream transformations as *bitstream kernels* and their corresponding executions as *bitstream processing.*

Bitstream-Carried Dependences. As mentioned earlier, it is easy to introduce dependences into bitwise computations, with the use of operators carrying multi-bit effects, such as various shift and arithmetic operators. Considering the XML parsing example in

Transformations	<	t	a	g	>		<	v	a	l	>		<	>	e	r	r	>	<	a	]
C_0 = [a-zA-Z]	.	1	1	1	.	.	.	1	1	1	.	.	.	.	1	1	1	.	.	1	.
C_1 = [>]	.	.	.	.	1	.	.	.	.	.	1	.	.	1	.	.	.	1	.	.	.
C_2 = [<]	1	.	.	.	.	.	1	.	.	.	.	.	1	.	.	.	.	.	1	.	.
L_0 = Advance(C_2)	.	1	.	.	.	.	.	1	.	.	.	.	.	1	.	.	.	.	.	1	.
E_0 = L_0 & ¬C_0	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.
L_1 = ScanThru(L_0,C_0)	.	.	.	.	1	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	1
E_1 = L_1 & ¬C_1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1

Figure 5.4: Bitstream Transformations in XML Parsing [76] (to make the bitstreams easier to read, zeros are marked as dots).

Figure 5.4, two out of the seven transformations involve dependences: Advance(C_2) and ScanThru(L_0, C_0). The former shifts every bit in C_2 one step to the right (note that it is non-trivial to shift one bit for an entire bitstream). The latter starts from a 1 in L_0 and marks 1 right after a sequence of 1s in C_0. Essentially, ScanThru(L_0, C_0) $= (L_0 + C_0)$ & $\neg C_0$. Due to the use of shift and addition, both transformations introduce bitstream-carried dependences. In the source code, these dependences appear as the *loop-carried dependences*, a challenging class of dependences that is often beyond the reach of existing parallelizing compilers [131, 11]. Efforts so far in optimizing bitstream processing focus on the use of SIMD intrinsics [15, 76]. However, this requires to manually redesign the bitstream processing algorithms to handle the dependences across SIMD lanes (e.g., 256 bits). Moreover, the benefits would not be sustained when expanding the solution to an entire bitstream, because of the limited width of SIMD lanes. To improve the scalability as well as the productivity, this work explores a more automatic approach that enables coarse-grained speculative parallelism for bitstream programs, namely *principled bitwise speculation* (PBS). Next, we give an overview of this new approach.

5.3 Overview

The section presents the basic workflow of principled bitwise speculation (PBS), summarized by Figure 5.5. At high level, PBS consists of three core modules: (i) a *bitwise static analysis module*, (ii) a *bitstream program modeling module*, and (iii) a *runtime speculation module.* Given a bitstream program, the static analysis module first performs a series of static analyses to identify the bits in the program variables that essentially cause the bitstream-carried dependences, referred to as *dependent bits.* The dependent bits are then feed into the bitstream program modeling module, where a finite-state machine (FSM) is generated, sometimes partially or virtually, to capture the basic behaviors of bitstream programs. After these preparations, the runtime speculation module spawns a set of threads to process the input bitstream(s) speculatively. In specific, each speculative thread first leverages an FSM-based speculation technique, called *lookback*, to predict the runtime values of the dependent bits. With those values, the thread is able to speculatively execute the bitstream program. In cases the prediction fails, the runtime module also features a fast bitstream-customized approach to accelerate the recovery. In the following, we present these three modules one by one.

5.4 Static Dependent Bit Analysis

Conventionally, program dependences are defined based on the read and write of *variables.* However, for bitstream programs, such variable-level dependence analysis may not capture the dependences precisely, due to bit-level value manipulations. In this section, we present

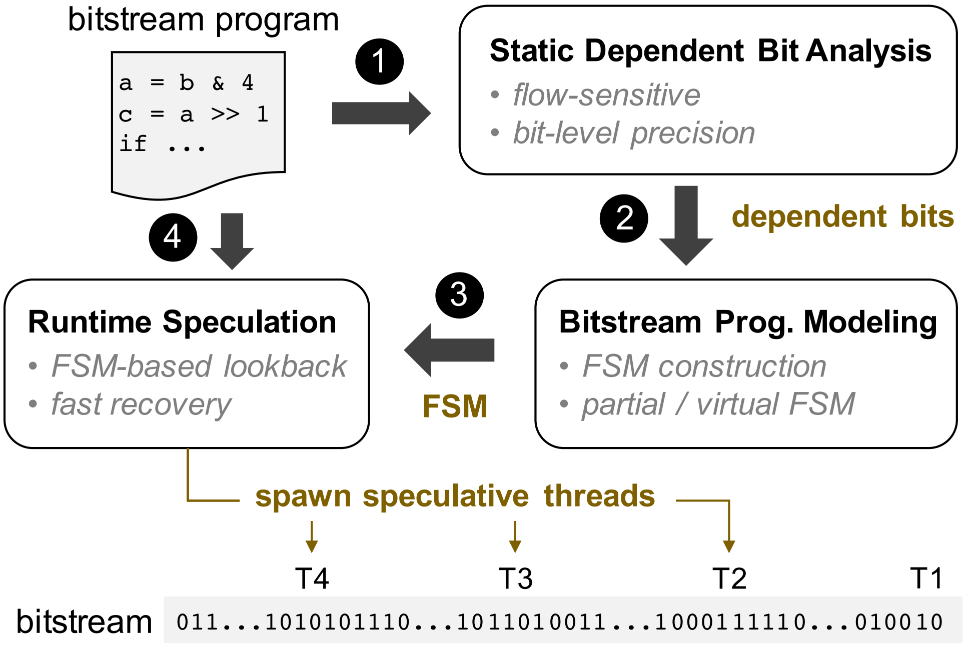


Figure 5.5: Workflow of Principled Bitwise Speculation (PBS)

an assembly of static analyses that analyze bitstream programs down to the bit level to pinpoint the exact bits causing the dependences, together referred to as *dependent bit analysis*. Before introducing its details, we first define *dependent bit* both intuitively and formally.

5.4.1 Dependent Bit: Motivation

The idea of dependent bits can be naturally extended from the dependences on variables. In general, if two instructions s_i and s_j access the same memory location M and one of them writes to M, then there exists a (data) *dependence* between s_i and s_j. For programs without bitwise operations, M usually refers to a variable (e.g., an integer or a char). In this case, we call M the *dependent variable*. Consider the following code.

```
L1: n = n + x
L2: y = n & 7
```

There exists a write-after-read dependence from `L1` to `L1` itself on variable `n` and a read-after-write dependence from `L1` to `L2` also on `n`. Conventionally, in both cases, variable `n` is referred to as the memory M in the dependence definition.

However, for the second dependence, if we break down `n` into individual bits (e.g., 8 bits), we may narrow down M to smaller granularity based on the AND operation in `L2`. In fact, a closer look at `L2` reveals that the five most significant bits of `n`, denoted as $\texttt{n}_{[3:7]}$, actually do not contribute to the calculation of `t` – they are *ignored*. In another word, only $\texttt{n}_{[0:2]}$ are involved in this dependence, which we referred to as *dependent bits*. Similarly, `L2` also indicates that the five most significant bits of `t` (i.e., $\texttt{t}_{[3:7]}$) are always zeros – they are *constants*. Therefore, a later use of `t` (not shown) does not necessarily depend on `L2` regarding $\texttt{t}_{[3:7]}$. In both of the above scenarios, some instructions, in principle, may not have to access all the bits of variables [3]. Based on this intuition, we define the dependent bits more formally.

Definition 10 *If two instructions s_i and s_j have to access the same bit of variable v (say $v_{[k]}$), and at least one of them writes to $v_{[k]}$, then there exists a (data) dependence between s_i and s_j, where $v_{[k]}$ is the* ***dependent bit****.*

The concept of dependent bits makes it possible to capture the dependences in bitstream programs in a more precise way, which is critical to the construction of FSMs, as we show later.

[3] This should not be confused with the instruction implementations that read all the bits from a register; Here, the concept is for static analysis.

Next, we put the dependence discussion in the context of bitstream processing. Consider the following example.

```
L1: for i = 0 to N
L2:   n = n + in[i]
L3:   out[i] = n & 7
```

where `in[]` is the input bitstream traversed by the `for` loop, byte by byte, and transformed to the output bitstream `out[]`. In addition to the dependence from `L2` to `L3` inside the loop body, there also exist dependences across iterations, known as *loop-carried dependences*. For instance, the `L2` in iteration 2 (reads `n`) depends on the `L2` in iteration 1 (writes to `n`). These dependences are chained together across iterations, preventing any iteration-level parallelizations.

However, if we look closer at the use of variable `n` in `L3`, only $\mathtt{n}_{0:2}$ have to be accessed and if we propagate this back to `L2`, that means, for the `n` on the right hand side of `L2`, the five most significant bits $\mathtt{n}_{3:7}$ are ignored – though used by the addition, the outputted $\mathtt{n}_{3:7}$ anyway will not be used by the next instruction `L3`. Thus, the loop-carried dependences actually only involve 3 bits of `n`, rather than 8 bits (or 32 bits for an integer), making them much more amenable to break with speculation techniques. This observation is one enabler for principled bitwise speculation.

Based on the above discussions, the key is to find out those dependent bits that cause loop-carried dependences. Next, we introduce an assembly of static analyses that can effectively identify the dependent bits, namely *dependent bit analysis*.

5.4.2 Dependent Bit Analysis

For clarity, we decompose the dependent bit analysis into three more basic analyses. We first briefly introduce each of them and how they are integrated, then present their algorithms in detail. The `for` loop example in Section 5.4.1, denoted as L_{main}, will be used as the running example.

Entry-Point Liveness Analysis. First, dependent bits should be *live* at the entry of the loop body of L_{main}, that is, they will be used before they get redefined. Otherwise, if the bits are redefined first, then they will not depend values from prior iterations. However, it is very challenging to directly perform bit-level liveness analysis due to the complex bit status caused by various bitwise operations. Existing liveness analysis [120] can reach bit sections, but not individual bits. For this reason, we first perform variable-level liveness analysis, then rely on a separate bit status analysis (discussed next) to prune irrelevant bits. We refer to the former as *entry-point liveness analysis*, which is different from the classic liveness analysis in that it only computes the live variables at the entry of the loop body, rather than every program point or basic block. As shown in Section 5.4.3, this difference reduces the iterative data-flow analysis to a single pass. In the running example, this analysis finds that both `n` and `in[i]` are live at the loop body entry.

Bit-Status Analysis. The main complexity in dependent bit analysis comes from the variation of bit status in variables – some bits may be involved in the calculation semantically while others may not (*ignored*). Furthermore, whether they are involved or not depends on if some bits of the operands are known (*constant*). Consider `y = n & m`. If $\mathtt{n}_{[0:2]}$ are known to be zeros, then $\mathtt{m}_{[0:2]}$ can be ignored. We address these complexities with an effective bit-status analysis that was previously developed for hardware synthesis [13]. In the running example, this analysis finds that `out[i]`$_{[3:7]}$ are zeros and `n`$_{[3:7]}$ are ignored in both `L2` and `L3`.

Unchanged-Bit Analysis. The last piece of analysis is to find bits that never change values through all iterations, called *unchanged bits.* Note that they are different from the constant bits which are redefined with known values (0 or 1). Unchanged bits are defined before the loop and remain unchanged through the iterations. In the running example, this analysis finds that all bits in `in[i]` are unchanged.

Putting It All Together. Now, we integrate the results of the above three analyses. Assume the bits in all live variables at the loop body entry are in bit set $\mathcal{B}_{live}$, the semantically useful bits in the loop body are in bit set $\mathcal{B}_{unknown}$, and the set of unchanged bits is $\mathcal{B}_{nochange}$, then the dependent bits $\mathcal{B}_{depen}$ can be calculated as follows.

$$\mathcal{B}_{depen} = (\mathcal{B}_{live} \cap \mathcal{B}_{unknown}) - \mathcal{B}_{unchanged} \tag{5.1}$$

In brief, the dependent bits should be (i) *live* at the entry of the loop body, (ii) *semantically useful in the loop body*, but (iii) *possibly changed* during the loop iterations. Together, the three conditions can narrow down the set of dependent bits to a minimum. Consider the running example:

- $\mathcal{B}_{live}$ = {`n`$_{[0:7]}$`:L2`, `in[i]`$_{[0:7]}$`:L2`}, where : is followed by the instruction(s) using the bits before redefinitions;

- $\mathcal{B}_{unknown}$ = {`n`$_{[0:2]}$`:L2`, `in[i]`$_{[0:2]}$`:L2`, `out[i]`$_{[0:2]}$`:L2`}, where : indicates the instruction(s) using the bits;

- and $\mathcal{B}_{unchanged} = \{\texttt{in[i]}_{[0:7]}\}$.

Based on Equation 5.1, we have $\mathcal{B}_{depen} = \{\texttt{n}_{[0:2]}\texttt{:L2}\}$. Next, we explain how each of the three analyses work in detail.

5.4.3 Algorithms

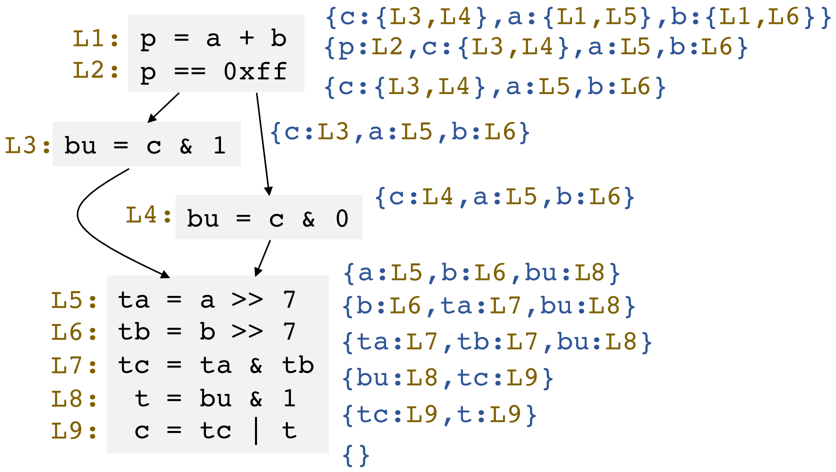
In general, dependent bit analysis follows iterative data-flow analysis over the control-flow graph (CFG) of the main loop L_{main} body. Thanks to their specific goals, two of its three sub-analyses only require one iteration to complete.

Algorithm for Entry-Point Liveness Analysis. The goal of entry-point liveness analysis is to find out which variables are live at the entry point of main loop body. The domain of the analysis is the power set of all variables appearing in the loop body and the direction of the analysis is backward. For an instruction i, the transfer function f_i is:

$$\text{LIVEENTRY}_{in} = \text{LIVEENTRY}_{out} - \text{DEF}(i) \cup \text{USE}(i) \tag{5.2}$$

At a joining point of the CFG, the meet operator $\wedge$ is union $\cup$. When the analysis starts, LIVEENTRY is initialized to $\varnothing$ at the exit of the CFG. After finishing the first instruction of the CFG, the analysis stops and outputs the latest LIVEENTRY. Figure 5.6 shows an example analysis on a simplified CFG based on Figure 5.1. The IDs of instruction(s) using the corresponding live variables are also attached.

Algorithm for Bit-Status Analysis. To analyze bit status of different variables, we adopt an existing analysis developed for hardware design [13]. In this analysis, the bit



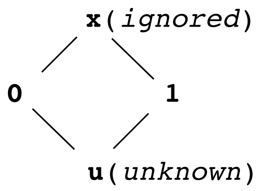
```
                          {c:{L3,L4},a:{L1,L5},b:{L1,L6}}
L1: p = a + b             {p:L2,c:{L3,L4},a:L5,b:L6}
L2: p == 0xff             {c:{L3,L4},a:L5,b:L6}

L3: bu = c & 1            {c:L3,a:L5,b:L6}

L4: bu = c & 0            {c:L4,a:L5,b:L6}

                          {a:L5,b:L6,bu:L8}
L5: ta = a >> 7           {b:L6,ta:L7,bu:L8}
L6: tb = b >> 7           {ta:L7,tb:L7,bu:L8}
L7: tc = ta & tb          {bu:L8,tc:L9}
L8:  t = bu & 1           {tc:L9,t:L9}
L9:  c = tc | t           {}
```

Figure 5.6: Example Entry-Point Liveness Analysis (backward).

status is defined as one of four cases: `unknown`, `0`, `1`, and `ignored`:

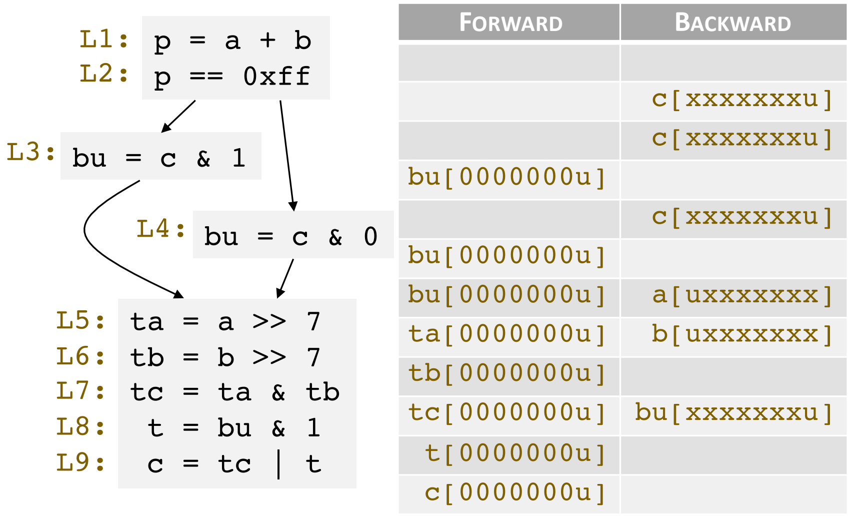
- `unknown` means the bit value cannot be inferred;
- `0` means the bit value can be inferred and it is zero;
- `1` means the bit value can be inferred and it is one;
- `ignored` means the bit does not contribute to the output.

Internally, bit-status analysis consists of two sub-analyses: *constant-bit analysis* which checks if a bit has a known value (0 or 1) and *ignored-bit analysis* which infers if a bit will be ignored with no impacts on the outputs. The former extends the constant propagation to the bit level and therefore is a forward analysis, while the latter resembles the dead code analysis and thus is backward. Both analyses operate on all the bits of all variables, denoted as $\mathcal{B}_{all}$. The following lattice diagram shows the partial order among the four bit statuses.



During the analysis, the bit status is moved up from the bottom of the lattice. That is, the analysis first initializes all bits in $\mathcal{B}_{all}$ to `u`, then it applies constant-bit analysis to mark bits with their known values (0 or 1). After that, it applies ignored-bit analysis to identify "ignored" bits. In both steps, the analysis is iterative to cope with potential inner loops of the main loop. Next, we briefly explain each of the two sub-analyses. More details can be found in [13].

First, constant-bit analysis traverses the CFG forwards to propagate bits with known values. Unlike the conventional constant propagation, the transfer function of constant bit analysis highly depends on the specific operation involved. Take instruction `bu = c & 1` as an example. By taking a logical AND with `1`, the analysis infers that $\texttt{bu}_{[1:7]}$ must be zeros. Similarly, it infers $\texttt{ta}_{[1:7]}$ are zeros too, based on `ta = a >> 7`. After constant-bit analysis, all bits in Ball are either 0, 1, or "unknown", which are the "not ignored" cases. The ignored-bit analysis starts from these bit statuses, traverses the CFG backwards, and turns some of them to "ignored" based on the specific operations. For example, the analysis infers that $\texttt{c}_{[1:7]}$ in `bu = c & 1` are `ignored`, due to the AND operation with `1`. Figure 5.7 shows a bit-status analysis on our running example. For limited space, only the variables with updated bit status(es) are shown.



```
L1: p = a + b
L2: p == 0xff
L3: bu = c & 1
L4: bu = c & 0
L5: ta = a >> 7
L6: tb = b >> 7
L7: tc = ta & tb
L8:  t = bu & 1
L9:  c = tc | t
```

FORWARD	BACKWARD
	c[xxxxxxxu]
	c[xxxxxxxu]
bu[0000000u]	
	c[xxxxxxxu]
bu[0000000u]	
bu[0000000u]	a[uxxxxxxx]
ta[0000000u]	b[uxxxxxxx]
tb[0000000u]	
tc[0000000u]	bu[xxxxxxxu]
t[0000000u]	
c[0000000u]	

Figure 5.7: Bit Status Analysis (forward&backward).

Algorithm for Unchanged-Bit Analysis. The algorithm used for unchanged-bit analysis is straightforward. To find out bits never defined in the main loop, the analysis initializes all bits in $\mathcal{B}_{all}$ as “unchanged”, then it scans every instruction in the CFG and marks bits that are defined as “changed”. In the end of the scanning, the remaining “unchanged” bits are outputted. As unchanged-bits are flow-insensitive, the analysis can traverse the CFG either forwards or backwards, in just one pass. Consider the example in Figure 5.7. The results of unchanged-bit analysis consist of all bits in `a` and `b`.

Merging Results. Based on Equation 5.1, we can compute the dependent bit set $\mathcal{B}_{depen}$ for the running example. The calculation process is shown in Figure 5.8. In the end, it only includes $\texttt{c}_{[0]}$, which is used by instruction `L3`.

$$\mathcal{B}_{depen} = (\mathcal{B}_{live} \cap \mathcal{B}_{unknown}) - \mathcal{B}_{unchanged}$$

$$= \{\texttt{c}_{[0:7]}\texttt{:}\{\texttt{L3,L4}\}, \texttt{a}_{[0:7]}\texttt{:}\{\texttt{L1,L5}\}, \texttt{b}_{[0:7]}\texttt{:}\{\texttt{L1,L6}\}\}$$

$$\cap \{\texttt{c}_{[0]}\texttt{:}\{\texttt{L3}\}, \texttt{a}_{[0:7]}\texttt{:L1}, \texttt{a}_{[7]}\texttt{:L5}, \texttt{b}_{[0:7]}\texttt{:L1}, \texttt{b}_{[7]}\texttt{:L6}, \cdots\}$$

$$- \{\texttt{a}_{[0:7]}, \texttt{b}_{[0:7]}\}$$

$$= \{\texttt{c}_{[0]}\texttt{:}\{\texttt{L3}\}, \texttt{a}_{[0:7]}\texttt{:L1}, \texttt{a}_{[7]}\texttt{:L5}, \texttt{b}_{[0:7]}\texttt{:L1}, \texttt{b}_{[7]}\texttt{:L6}\} - \{\texttt{a}_{[0:7]}, \texttt{b}_{[0:7]}\} = \{\texttt{c}_{[0]}\texttt{:}\{\texttt{L3}\}\}$$

Figure 5.8: Calculation of Dependent Bits for Example in Figures 5.7 and 5.6 (for $\mathcal{B}_{unknown}$, only the relavent elements are shown).

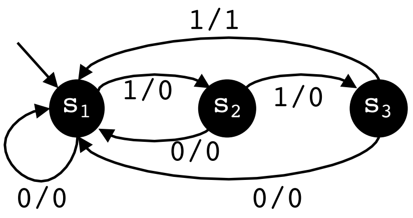
5.5 Modeling Bitstream Programs

This section discusses the roles that dependent bits play in bitstream processing, with a goal to create an *abstraction* of bitstream programs in general.

Bitstream Program Abstraction. Despite that bitstream programs may carry complex logic with various instructions, essentially, they all boil down to a transformation of some bitstream(s). This makes them resemble sequential circuits, though one is software and the other is hardware. The close correspondence motivates us to model bitstream programs with finite-state machines (FSMs), a model for sequential circuits. In the following, we first present a basic method to construct FSMs from bitstream programs, then discuss the strategies to address extremely large FSMs.

5.5.1 FSM Construction

In a typical sequential circuit design scenario, some forms of FSMs (such as Mealy machines or Moore machines) are often first constructed to model the behaviors of the designed



(a) FSM

CS		I	NS		O
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	0	0	1

(b) Truth Table

Figure 5.9: Example FSM (Mealy machine) and Truth Table.

circuits. Then, by encoding the FSM states with binaries, the FSM is converted into a truth table. From there, the flip-flops will be determined and the circuit diagram will be generated. Figure 5.9-(a) shows the FSM (Mealy machine) for designing a hardware counter that counts three consecutive ones. By encoding the three states with `00`, `01`, and `10`, a truth table can be generated, as Figure 5.9-(b), where `CS`/`NS` represent the current/next state and `I`/`O` represent the input/output.

In the context of bitstream program modeling, we reverse the FSM-to-truth table process. First, we generate a truth table, then encode the value combinations in the truth table with states to construct the FSM. Next, we elaborate the two phases in detail.

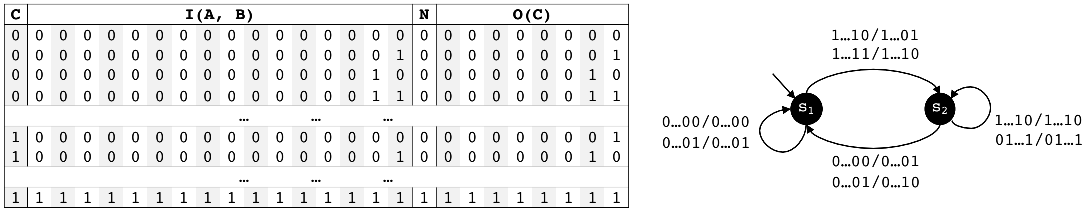
Truth Table Generation. In sequential circuit design, the truth table reflects the boolean logical relations among the input, the output, and the memory element of a sequential circuit. In the context of bitstream processing, we use truth table in a similar way, except that the memory element of a sequential circuit is replaced with the dependent bits in bitstream programs. Given a bitstream program with identified dependent bits $\mathcal{B}_{depen}$, we first identify the input bits $\mathcal{B}_{in}$ and output bits $\mathcal{B}_{out}$, that is, the bits consumed from input bitstream(s) and the bits written into the output bitstream(s) in each iteration of the main loop. Here, we assume the input bitstreams are *read-only* and the output bitstreams are *written-only*. Then, by tracking the uses of loop index in array references, we can easily identify the input and output bits, such as the input bits `A[i]` and `B[i]` and the output bits `C[i]` in our running example (see Figure 5.1). For more complicated situations, we can provide pragmas to programers for helping identify the input/output bitstreams. With those bits, we can generate the truth table as follows.

(a) List the dependent bits $\mathcal{B}_{depen}$ as both the `CS` columns and `NS` columns of the truth table. Set the input bits $\mathcal{B}_{in}$ as the `I` columns and the out bits $\mathcal{B}_{out}$ as the `O` columns of the truth table, respectively.

(b) Enumerate all the binary combinations of the bits in the `CS` and `I` columns, which, in fact, determine the total number of rows of the truth table N_{row}.

(c) For each row of the truth table, execute the bitstream program (only main loop body) by assigning the input bits and dependent bits with the corresponding values in this row. Record the resulted values of dependent bits and output bits, and fill their values to this row in the `NS` columns and `O` columns, respectively.

It is easy to find that the total number of columns in the truth table $N_{col} = |\mathcal{B}_{in}|$ $+ 2 \times |\mathcal{B}_{depen}| + |\mathcal{B}_{out}|$. Figure 5.10-(a) shows the truth table generated for our running example (see Figure 5.1) based on the dependent bits found in Section 5.4.3. In this case, the numbers of input and output bits are both eight and the number of dependent bits is one. The size of the table is $2^{17} \times 26$, which is quite large even for offline generation. We



C	I(A, B)																N	O(C)							
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	1
									…			…			…										
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
									…			…			…										
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(a) Truth Table

(b) FSM

Figure 5.10: Truth Table and FSM (partially shown) for the Running Example (see Figure 5.1 and Section 5.4.3).

will discuss how to address it shortly in Section 5.5.2. Next, we describe how to construct an FSM based on the truth table.

FSM Construction. The key idea in constructing the FSM is treating the value combinations of dependent bits as the *states*. This means, for a bitstream program with $|\mathcal{B}_{depen}|$ bits, the number of states would be $2^{|\mathcal{B}_{depen}|}$. In our running example, as there is only one dependent bit, the number of states is two: one for bit value 0 and the other for bit value 1. As to the FSM transitions, they are actually already laid out in the truth table: for the current state in `C` column(s), given the input bits in `I` column(s), the next state is shown in the `N` column(s) and the output bits are shown in the `O` column(s). The number of transitions equals the number of rows of the truth table. Figure 5.10-(b) shows the FSM constructed based on the truth table in Figure 5.10-(a). For space limits, only some representative transitions are shown.

It is not surprising that the FSM and truth table constructed for our running example are essentially those for designing a byte-level hardware adder. Essentially, the hardware adder and the bitstream program are equivalent in terms of functionality. However, note that the FSM-based modeling of bitstream programs does not require the programmers to be familiar with the hardware design and redevelop the solution from an FSM point of view. Furthermore, the logic of bitstream programs could be quite complex with the programming flexibility of high-level languages, which can make manual FSM design an extremely challenging task.

Another challenge in the FSM-based modeling is that the sizes of FSMs could be very large for real-world bitstream programs (see Section 5.8), because of the large numbers of dependent bits and/or input bits. We address this issue next.

5.5.2 Partial and Virtual FSMs

For FSMs (and truth tables) that are too large to generate in practice, we introduce *partial* and *virtual FSMs*.

Partial FSMs. The intuition is that, in many applications, the visiting frequencies of FSM transitions are biased. Hence, it is possible to use a small portion of captured transitions to cover a large number of actual transitions. As shown later, this is often sufficient to enable effective speculation for some bitstream programs. To achieve this, we use a pool of training input bitstreams to collect "hot values" of dependent bits and input bits based on their appearing frequencies. Base on the "hot values", a partial truth table is first constructed, with some rows potentially missing. Then, following the prior mentioned truth table-to-FSM construction, a partial FSM is created.

Virtual FSMs. Another way to address the oversized FSMs is to completely bypass the physical FSM construction. Instead, it simulates the FSM transitions with the executions of bitstream program. In particular, an FSM transition is *virtually* performed with an execution of the main loop body of the bitstream program, where the current state is the current values of dependent bits and the next state is the resulted values of the dependent bits after the execution. In this way, there is no need to generate the truth table. However, the dependent bits, along with the input bits, remain to be identified with the static analyses, to capture the FSM states and inputs on demand. More details regarding its uses are in Section 5.6.

Note that even though the use of virtual FSMs avoids physical FSM generations, generating physical FSMs may still be beneficial, because it enables the use of various FSM optimizations. Next, we will show how to use the FSM models to enable speculative bitstream processing.

5.6 Runtime Speculation

The section presents the idea of FSM-based speculation for bitstream processing, then introduces a novel technique that can leverage the special properties of bitstream programs to accelerate the recovery from misspeculation.

5.6.1 FSM-based Speculation

The basic idea of speculative FSM execution stems from an interesting observation [62] that a future FSM state can be effectively predicted by starting the FSM execution on the suffix of the input prior to the speculation position, a technique later formalized and referred to as *lookback* [138]. Figure 5.11 depicts the lookback-based speculative FSM execution. The input sequence is first partitioned evenly based on the number of available CPU cores.

Then, each input partition is assigned to a thread to process. Except the first thread, all the other threads run speculatively. To find out the starting state, a speculative thread runs the FSM from all states on the suffix of the prior input partition (i.e., lookback). After the lookback, the state with highest number of occurrences among the ending states is selected as the starting state. Using the FSM in Figure 5.9 as the example, after a seven-symbol lookback, all three states transition to state s_1, implying that it must be the correct starting state. In general, lookback can significantly improve speculation accuracies for many FSMs.

When a predication fails, a reprocessing with the correct starting state is applied to the corresponding partition to ensure the correctness. Thanks to the state convergence properties, the reprocessing may stop earlier when the corrected state trace "merges" with the wrong state trace (more details in [138]).

To adopt the FSM speculation techniques into bitstream processing, we first construct the FSM for the given bitstream program, then leverage the FSM-based lookback to find out the most possible state. After that, the selected state is decoded into binary values, which are then assigned to the dependent bits in the bitstream program to start speculative execution. The high-level speculation workflow remains similar to that in Figure 5.11, except that the speculative FSM execution becomes the speculative execution of bitstream programs. Considering the example of long bitstream addition, FSM-based lookback essentially provides a systematic exploration of the prior bits "close by" to find out the possibility of a produced carry. Our evaluation (Section 5.8) shows that a short FSM-based lookback often yields high speculation accuracies for long bitstream addition and many other bitstream programs.

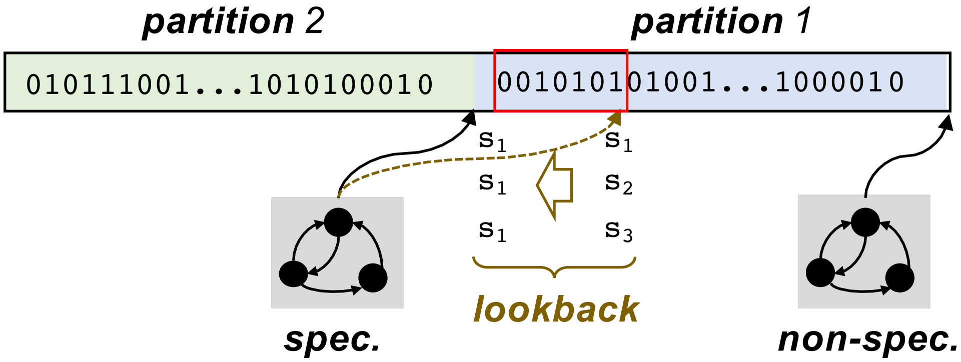


Figure 5.11: Lookback-based Speculative FSM Parallelization.

Speculation with Partial/Virtual FSMs. For partial FSMs, the lookback process is similar to the FSMs with full transition tables, except that the lookback may start with a subset of states and some FSM execution paths in the lookback may stop earlier due to the lack of the needed transitions. As a result, the accuracy of the predicted state could be reduced. In general, partial FSMs work well in cases where the FSM transitions follow a biased distribution. As to virtual FSMs, the lookback directly executes the bitstream program to mimic the FSM transitions. In specific, we start the lookback with "virtual states" – the value combinations of the dependent bits. If there are too many combinations, a subset is selected either randomly or based on some training inputs. Then, by assigning each value combination to the corresponding dependent bits, an instance of the bitstream program is run to perform "virtual transitions". At the end of the lookback, the value combination that appears mostly would be selected as the predicted values. Note that even though virtual FSMs bypass the physical FSM generation, the lookback essentially still explores the state convergence of FSMs in an implicit way ("virtually").

5.6.2 Fast Recovery from Misspeculation

In the existing FSM speculation, after parallel speculative executions, each predicted starting state is verified against the correct state – the ending state from the processing of prior input partition. When a verification fails, it needs to reprocess the corresponding input partition with the correct starting state. Despite some optimization [138] for stopping the reprocessing earlier, the reprocessing cost, in general, can still significantly compromise the speculation benefits [100].

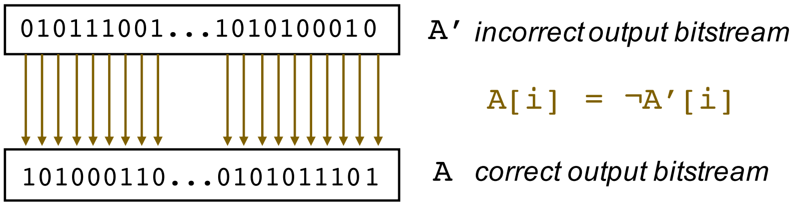


Figure 5.12: Example Fast Recovery from Misspeculation.

The above issue may be alleviated in speculative bitstream processing. Unlike the general FSM computations, the outputs of bitstream programs are binary sequence(s). Under certain conditions, the correct and incorrect output bitstreams may be correlated by some bitwise relations.

For example, the incorrect output bitstream could be the flipped version of the correct one (see Figure 5.12). If we can prove this as a *property* of the bitstream program, then we can directly recover the correct output bitstream from the incorrect one, rather than reprocessing the input bitstream.

In fact, we can prove such properties of bitstream programs offline with the help of their corresponding FSMs. Assume that the correct and incorrect output bitstreams are `O` and `O'`, respectively, and the hypothesis is that `O` $= \mathcal{P}($`O'`$)$, where $\mathcal{P}$ is a bitwise logical function. From the FSM point of view, the hypothesis can be proved as follows. For every pair of state transitions $T(s_1, I_1) = (s'_1, O_1)$ and $T(s_2, I_2) = (s'_2, O_2)$ where $s_1 \neq s_2$ and $I_1 = I_2$, we should have $O_1 = \mathcal{P}(O_2)$ as well as $O_2 = \mathcal{P}(O_1)$. Note that the proof requires $\mathcal{P}$ to be commutative, as a transition, in general, may happen in both the correct and misspeculated executions. After proving the $\mathcal{P}$ for the bitstream program, we can apply $\mathcal{P}$ to the output bitstream from any misspeculated processing to recover the correct one. We refer to this technique as *property-based fast recovery.*

5.7 Implementation

This section briefly describes some implementation details of the proposed principled bitwise speculation.

Static Analyzer. We prototyped dependent bit analysis on the latest LLVM (version 9.0.0). The analysis is implemented as an LLVM pass, called `depenBit`. The pass first identifies the main loop with the help of `BitstreamLoop` pragmas. Then, it runs an LLVM loop analysis pass [4] to find out the loop induction variable, followed by an SCC (strongly connected components) analysis pass [5] to locate the body of main loop. After these preparation, the pass starts the three sub-analyses mentioned in Section 5.4.3 and merges the analysis results to produce the dependent bits for the given bitstream program. More de-

[4] `https://llvm.org/doxygen/classllvm_1_1Loop.html`

[5] `https://llvm.org/doxygen/classllvm_1_1scc__iterator.html`

tails regarding the analyzer, including some of its potential limitations, will be discussed in Section 5.8.

FSM Generator and Speculation Runtime. In our current prototype, both the FSM generator and speculation runtime are implemented as standalone modules using `C++`. The FSM generator takes dependent bits as inputs and outputs an FSM transition table. In addition, the generator can optionally take a training input to create a partial FSM. By default, the size of partial FSMs is set to $|S| \times 1024$, where $|S|$ is the number of states. To support the speculative parallelization, we use the `Pthread` library for its more customizable thread settings. By default, the runtime creates the same number of threads as the number of available CPU cores. The default length for the lookback (in number of bits) is set to $2 \times |\mathcal{B}_{in}|$. As to the property-based fast recovery, the current prototype focuses on testing the hypothesis of NOT $\neg$ relation. More hypothetic relations will be gradually added in the future versions.

5.8 Evaluation

In this section, we evaluate the principled bitwise speculation, with a focus on the performance of speculative parallelization.

5.8.1 Methodology

Benchmarks. To facilitate the evaluation, we collected a set of eight bitstream kernels from multiple real-world applications, ranging from semi-structured data processing [76, 75] and text pattern matching [15] to multimedia applications [102, 74] and bioinformatics [133].

They are listed in Table 5.1. Three of them are implemented with SIMD intrinsics. For each kernel, we collected a set of inputs from their applications, including 10 small inputs (10MB each) and 10 large inputs (300MB each).

To demonstrate the end-to-end benefits, we also evaluate PBS with an open-source high-performance regular expression engine, called `icgrep` [15] (see more details in Section 5.8.4).

Table 5.1: Bitstream Kernels in Evaluation.

Abbreviation	**Brief Description**	**SIMD**
`shd_SRS`	Shift-hamming-distance filter kernel [133]	No
`802_11a`	IEEE 802.11a convolutional encoder [102]	No
`8b10b`	IBM 8bit/10bit block encoder [102]	No
`g721_uPK`	G.721 voice compression kernel [74]	No
`quoteStr`	JSON bitmap indexing from `Mison` [75]	No
`scanThru`	Ending index construction from `icgrep` [15]	Yes
`matchStar`	Matching "`*`" in regex from `icgrep` [15]	Yes
`xmlParser`	XML parsing kernel from `Parabix` [76]	Yes

Evaluation Platform. Our experiments are mainly conducted on a 64-core machine equipped with an Intel Xeon Phi 7210 processor (1.3GHz). The machine runs Linux 3.10.0 with supports of SSE4.2 and AVX2. As to the compilers, we use LLVM 9.0.0 for analyzing the source code and GCC 4.8.5 for generating the executables, with "`-O3`" optimization flag.

For the bitstream kernel evaluation, we measure the time spent on the main loop (i.e., bitstream traversal loop), while for the regular expression engine evaluation, we measure the end-to-end running time with everything included. All timing results reported are the average from 10 repetitive runs.

5.8.2 Static Analysis and Modeling

To prepare for the static analysis, we inlined the functions that are called inside the main loops to avoid precision loss from the inter-procedural analysis. The second column of Table 5.2 reports the number of LLVM IR instructions in the main loop of each kernel, where the static analyses are performed (#Instr denotes number of instructions in main loop; while #DB/#IB/#OB represent numbers of dependent/input/output bits).

Table 5.2: Static Analysis Results

Kernel	**#Instr.**	**#DB**	**#IB**	**#OB**
`shd_srs`	61	1	32	32
`802_11a`	72	5	32	2×32
`8b10b_cal`	140	1	32	32
`g721_upd`	143	2	3×32	3×8
`quoteStr`	61	1	2×32	32
`scanThru`	1218	1	2×256	256
`matchStar`	1226	1	2×256	256
`xmlParser`	1881	2	3×256	256

Analysis Results. In Table 5.2, columns 2-4 report the numbers of dependent bits, input bits and output bits, discovered in the bitstream kernels. For each kernel, we manually checked the source code to examine the correctness of the analysis results. In the end, our examination shows that the reported bits are both correct and precise. Among the eight kernels, five kernels are found with only one dependent bit, despite that the variables holding them are 64-bit unsigned integers or 256-bit SIMD vectors. For kernels `g721_upd` and `xmlParser`, there are two dependent bits. In both cases, the two bits come from two different variables. Kernel `802_11a` is found with the most number of dependent bits – five bits, which are all from the same variable $\texttt{shiftRegister}_{[5:3,1:0]}$. As to the input and output bits, the number ranges from 32 to 3 × 256 (3 means three bitstreams), except `g721_upd` which outputs 3 × 8 bits to three output bitstreams. It is not surprising that the last three kernels use so many input/output bits, as they are implemented with SIMD intrinsics. The static analysis time of the `DepenBit` pass, reported by LLVM, ranges from tens of milliseconds to several seconds.

Despite the success in analyzing the kernels, there exists some limitations with our current analysis implementation. One of them is the assumption that the main loop of bitstream kernels is in canonical form, which facilitates the identification of the loop induction variable and input/output bits. This could be addressed with more advanced induction variable analysis or the use of pragmas. In addition, as most bitwise operations work with non-floating point variables, our current analysis does not cover floating point variables.

FSM-based Modeling. The number of dependent bits reported in Table 5.2 indicates that the number of FSM states ranges from two to 32 (i.e., 2^5), which is quite manageable. However, due to the large number of input bits, it remaining impractical to generate the FSMs physically. For this reason, we adopt partial and virtual FSMs (see Section 5.5.2). In particular, we generate a partial FSM for kernel `8b10b_cal`, which is one of the kernels with the smallest truth table. More importantly, the value combinations of dependent bits in `8b10b_cal` follows a highly biased distribution, making it a good candidate for using a partial FSM. By running on a training input, we generated the partial FSM for `8b10b_cal` with 2 × 1024 transitions. For the other kernels, we adopt the virtual FSMs, which use the executions of bitstream programs to simulate the FSM transitions.

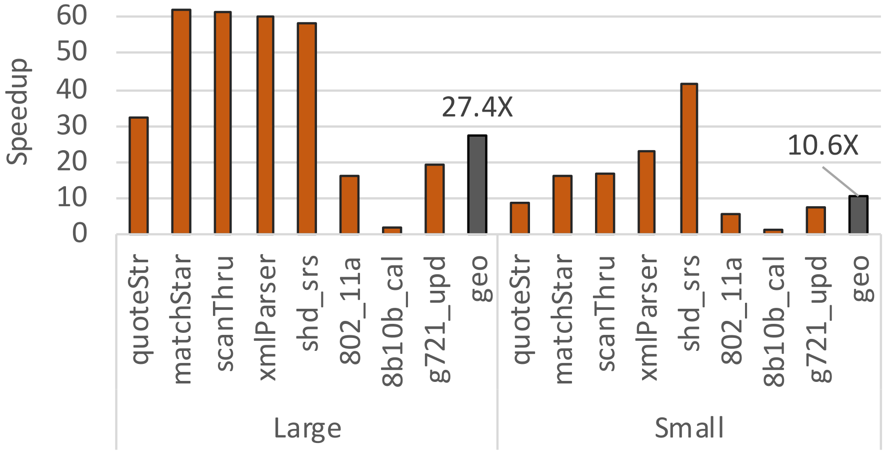


Figure 5.13: Speedup of Parallelized Kernels (64-core Machine).

5.8.3 Speculative Parallelization

This section evaluates the parallel performance of FSM-based speculation (Section 5.6) on the bitstream kernels, including their speedups, speculation accuracies, and scalabilities.

Speedup. Figure 5.13 reports the speedups of the parallelized bitstream kernels on the 64-core machine. In general, for the large inputs, the speedups tend to be higher. Because, with larger inputs, the parallelization costs (e.g., threads creation and etc.) can be better amortized by the longer executions.

In specific, four bitstream kernels (`matchSar` , `scanThru`, `xmlParser`, and `shd_srs`) achieve nearly (or slightly higher than) 60X speedups. Note that, for three of them, the speedups are on top of the vectorizations with SIMD intrinsics. There are two main reasons for their higher speedups than the other kernels. First, all the four kernels achieve 100% speculation accuracies (will be discussed more shortly). Second, they do not generate long output streams; instead, their output bits are directly consumed by the following steps. This makes the bitstream kernels more computation-bound, thus reaching higher speedups with more CPU cores.

Among them, `8b10b_cal` achieves the least speedup (2.3X for large inputs). This is mainly due to its limited speculation accuracy (around 50%). We will discuss more about this kernel later in this section, including a couple of optimizations. For `qouteStr`, the speculation accuracy, in fact, is similar to `8b10b_cal`. However, as we will show later, this kernel is qualified for the *property-based fast recovery*. With this technique, it is able to reach 32.3X speedup for large inputs, despite the limited speculation accuracy. Finally, for kernels `802_11a` and `g721_upd`, the speculation accuracies are also 100%. However, due to the need for generating long output bitstreams, they become more I/O-bound as more and more CPU cores are added (as shown later in scalability), reaching 16.2X and 19.6X speedups for large inputs, respectively.

Speculation Accuracy. As mentioned earlier, six out of eight kernels achieve 100% speculation accuracies in our tested cases, with the default lookback length (i.e., $2 \times |\mathcal{B}_{out}|$). This confirms the effectiveness of FSM-based speculation, which systematically explores the possible changes of bit values ("transitions") under the "partial context" of input bits nearby. For the other two kernels (`8b10b_cal` and `qouteStr`), their speculation accuracies are only about 50% as the two states of their FSMs rarely converge. Fortunately, these two kernels are eligible for some optimizations, as explained next.

Optimization with Fast Recovery. First, `qouteStr` passed the testing of NOT relation hypothesis (see Section 5.6.2) for fast recovery. This means, if a misspeculation occurs, it is possible to directly flip the incorrect output bitstream to get the correct one, rather than reprocessing the input bitstream. With fast recovery, `qouteStr` achieves much higher speedup than the other benchmark `8b10b_cal` which also suffers from low speculation accuracy (see Figure 5.13).

***Optimizations for* 8b10b_cal**. We found two optimizations for `8b10b_cal`. First, its FSM model, in fact, executes faster than the original program, thus can replace the latter. Second, the FSM has only two states. In this case, we may aggressively execute both states, an FSM parallelization technique known as *enumerative parallelization* [83]. With both optimizations, we observed a 11.3X speedup on the 64-core machine, instead of 2.3X as reported earlier in Figure 5.13.

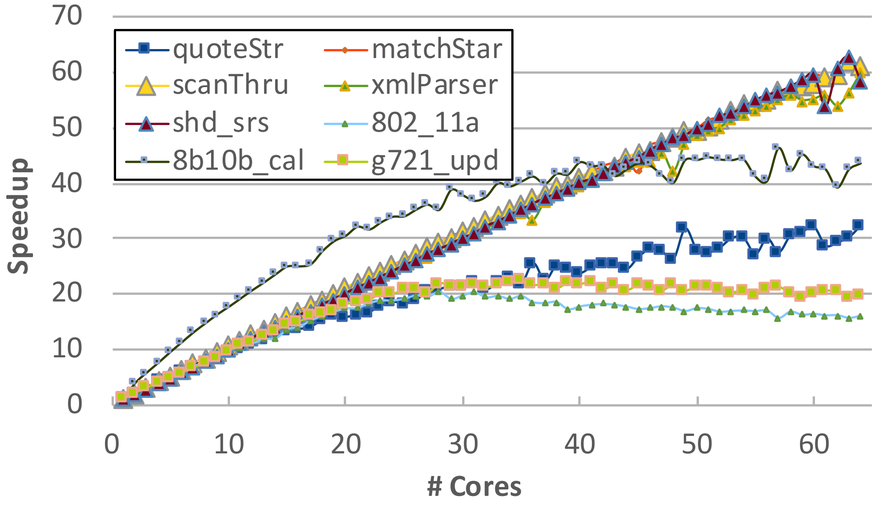


Figure 5.14: Scalability (Large Inputs on 64-core Machine).

Scalability. As demostrated in Figure 5.14, four kernels present near-linear speedups (`matchStar`, `scanThru`, `xmlParser`, and `shd_srs`), up to 64 cores. `qouteStr` and `8b10b_cal` also scale up to 64 cores, but their speedups are more close to their maximums. The speedups of 802_11a and g721_upd saturate with around 20-30 cores. In general, the scalability mainly depends on the speculation accuracy, reprocessing costs, and the I/O-to-computation ratio.

5.8.4 Case Study: Enabling Data-Parallel icgrep

To confirm the benefits of PBS on full-fledged applications, we experimented it with icgrep [15], a regular expression engine with SIMD parallelism. icgrep compiles a regular expression into a bitstream program (in LLVM IR) to find matches in a text stream. Here, we use the same regular expressions as the ones for evaluating icgrep [15] (see Table 5.3). The input textual streams are collected from a Linux server using tcpdump tool.

First, our dependent bit analysis shows that the number of dependent bits in the generated bitstream programs ranges from 19 to 36 (the 3rd column of Table 5.3). For more complex regular expressions, the number of dependent bits tends to increase. Given the relatively large numbers of dependent bits, we opt for virtual FSM-based speculation. The last column of Table 5.3 reports their maximum speedups on the 64-core machine, ranges from 10.4X to 27.6X. The speedups come from the high speculation accuracies. In fact, for all the six generated bitstream programs, we observed 100% speculation accuracies, thanks to the convergence properties of their virtual FSMs. The sub-linear speedups are due to the fact that they also generate the output bitstreams, which are saved in cases the users want to print out the matched contexts.

Table 5.3: Evaluation of PBS on `icgrep`.

ID	Regular Expression	#DB	Speedup
1	@	19	10.4X
2	([0-9][0-9]?)/([0-9][0-9]?)/([0-9][0-9]([0-9][0-9])?)	27	17.5X
3	([ˆ@]+)@([ˆ@]+)	22	18.1X
4	(([a-zA-Z][a-zA-Z0-9]*)://|mailto:)([ˆ/]+)(/[ˆ]*)?—([ˆ@]+)@([ˆ@]+)	36	21.9X
5	[ ](0x)?([a-fA-F0-9][a-fA-F0-9])+[.:,?!]	26	21.7X
6	[A-Z]((([a-zA-Z]*a[a-zA-Z]*[ ])*[a-zA-Z]*e[a-zA-Z]*[ ])*[a-zA-Z]*s[a-zA-Z]*[ ])*[.?!]	32	27.6X

5.9 Summary

This work treats sequential bitstream programs from a new perspective, by analogizing them to the sequential circuits. Inspired by their similarities, this work proposes to model bitstream programs with FSMs. To facilitate the modeling, this work integrates multiple static analyses to systematically reason about the bits in program variables that cause the loop-carried dependences, namely, *dependent bit analysis*. With the identified dependent bits, an FSM is constructed for the bitstream program, following a modified truth table approach used in the conventional circuit design. For FSMs that are too large to generate, this work also introduces partial and virtual FSMs as alternatives. This *FSM modeling* enables the use FSM speculation techniques for parallelizing bitstream programs. To reduce the cost of misspeculation, this work further proposes *fast recovery* that leverages the logical property of bitstream programs to avoid reprocessing. Finally, evaluation with real-world bitstream programs and a regular expression engine confirms the effectiveness of the proposed techniques, achieving significant performance improvements on multicore/manycore machines.

Chapter 6

Related Work

6.1 Speculative Parallelization

Due to the dependences in state transistions, existing ways to parallelizing FSM rely on either enumeration-based parallel prefix-sum and its variations [73, 83], or speculative parallelization [138, 137, 99]. The former can be treated as a special case of speculative parallelization, where the "speculation" enumerates all the states, hence always covers the correct one. From this perspective, though the models proposed in this work can be reused for the former with simple extensions.

Some other FSM parallelization work focus on a few specific FSM applications, such as browser front-end [62] and JPEG decoder [67]. The basic ideas in these work were later formalized by Zhao and others [138] by introducing a concept called *principled speculation*. Other examples include hot state prediction for FSMs in intrusion detection [79] and speculative parsing [63]. Some studies also design and implement parallel Non-deterministic Finite Automata (NFA) [140], which naturally exposes parallelism, hence are relatively easier to parallelize, comparing with their DFA counterparts. Some prior work have also explored bit-parallel fine-grained parallelism for FSMs by converting FSM computations into a sequence of bit operations [84, 76], and the combination of both fine-grained and coarse-grained speculative parallelism [99].

The idea of speculative parallelization has been studied for years. These work include designing new programming language constructs [97] and developing parallelization frameworks [105, 31, 103, 123, 36]. Some of these studies have explored parallelism in irregular programs [70, 47, 95], which share some similarities with the parallelization of FSM computations, given that FSMs essentially run on an irregular data structure (a graph). Quinones and others [101] use pre-computation for speculative threading, which shares the idea with speculative FSM parallelization in that both exploit some contrstraints of the computation to facilitate speculative execution.

6.2 Enumerative Parallelization

There are few prior work on enumerative parallelization, due to the infeasibility in enumerating all the possible cases in general. Some early works [3, 127] examine the potential of enumerating different execution paths under control branches. If FSM transitions are hard-coded, rather than being stored in a transition table, the enumerative FSM parallelization would be similar to branch enumeration. N-way programming model [28] enumerates different algorithms or implementations of the same tasks and selects the one that finishes earliest. In more specific application areas, Malki and others [80] leverage the property of rank convergence to enable coarse-grained parallelization of dynamic programming computations, which is a form of state enumeration. In a similar way, Raychev and others [106] use symbolic execution to parallelize user-defined aggregations in big data frameworks, where a symbolic value is an abstraction of all the enumerative cases. More close to FSMs, there are a series of works [91, 58, 57] on enumerative parallelization of pushdown automata, which consist of an FSM and a stack, for processing semi-structured data like XML and JSON.

6.3 Bit-Level Analysis and Parallelism

Existing research on bit-level analysis is mainly for saving hardware resources, with applications to multimedia processing and telecommunications [10, 13, 45, 69, 117, 120]. For example, Budiu and others [13] proposed bitvalue analysis that finds unused and constant bits in C programs to improve their performance on specialized architectures with non-standard bitwidths. This analysis has been adopted by this work as part of the dependent bit analysis. Under a similar context, Stephenson and other [117] introduced a compiler, called Bitwise, to minimize the number of bits used by each operand in both integer and floating point programs. The compiler has shown promising results in architectural synthesis. Alternatively, Gupta and others [45] introduced a program representation to facilitate expanding traditional program analysis to the subword level. Following this work, Tallam and Gupta [120] designed a bitwidth-aware algorithm for global register allocation, showing 10Our work is deeply inspired by the above bitwise analysis. However, to the best of our knowledge, this is the first work that leverages bit-level analysis for program parallelization.

Besides code vectorization, there are also many efforts in exploiting bit-level parallelism in specific applications [55, 111], especially for string matching [85, 86, 93] and semi-structured data indexing [75, 76]. In particular, Carribault and Cohen [18] examined bit-parallel matching algorithms with register promotion optimizations. In general, these efforts bring potential applications that can benefit from our coarse-grained parallelization techniques.

Chapter 7

Conclusions

In this dissertation, we tackled the granularity, scalability modeling and optimization, and applicability issues in FSM-driven computations by integrating practical parallel programming techniques with rigorous program analysis and optimizations. More specifically, we first looked into the possibility of introducing speculation techniques at the instruction-level and SIMD-level. By restructuring the FSM transition loop and introducing multi-level speculation, we broke the barriers of adopting fine-grained speculative parallelism. The resulted FSM speculation framework achieves up to 4X performance boost comparing to the state of the art.

Following the granularity investigation work, we further examined the *scalability* of speculative FSM parallelization by developing a systematic performance model, which can effectively predict the best configuration in terms of the number of CPU cores to use for optimal performance and the effective use of energy (up to 5X speedup as well as up to 77% energy saving).

Then we tried to address the fundamental scalability issues inherited in the two basic FSM parallelization schemes: the cost of maintaining multiple execution paths in enumerative parallelization and the sequential chunk-by-chunk validations in speculative parallelization. For the former, we proposed the technique of *path fusion*, which can fuse different execution paths into a single one, either statically or dynamically. For the latter, we introduced the concept of *higher-order speculation* which allows a speculated state to be validated speculatively. For practical uses, we also discussed the scenarios where each scheme works the best and proposed a set of heuristics to help users select the parallelization scheme.Our evaluations over FSMs drawn from real-world applications and of diverse characteristics showed that the proposed techniques can substantially raise the scalabilities of both parallelization schemes.

Despite the promising results of speculative FSM parallelism, the benefits were limited to computations that explicitly use the FSM models. In the last project of dissertation, we looked beyond the FSM computations and interestingly found an important class of non-FSM computations – bitstream processing – that may also benefit from speculative FSM parallelization. In fact, we discovered that the inherent data dependences in bitstream programs can often be accurately modeled as FSMs, inspired by the fact that both sequential circuits (using FSM models) and bitstream programs consume and output binary sequences. The discovery led to the development of an assembly of bitwise static analyses for reasoning about the dependent bits and a truth table to FSM conversion technique. Together, the proposed technique, namely *principled bitwise speculation*, provides a rigorous treatment for parallelizing arbitrary bitstream program. The results of this work directly benefit many bitstream-based applications, such as bitmap construction, multimedia processing, and data indexing.

Bibliography

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2008.

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques.* Addison wesley Boston, 1986.

[3] Pritpal S Ahuja, Kevin Skadron, Margaret Martonosi, and Douglas W Clark. Multipath execution: Opportunities and limits. In *Proceedings of the 12th international conference on Supercomputing*, pages 101–108, 1998.

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

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

[6] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[7] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 351–367, 2020.

[8] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb 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.

[9] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. Scalable algorithms for nfa multistriding and nfa-based deep packet inspection on gpus. *IEEE/ACM Transactions on Networking*, 24(3):1704–1717, 2015.

[10] Rajkishore Barik and Vivek Sarkar. Enhanced bitwidth-aware register allocation. In *International Conference on Compiler Construction*, pages 263–276. Springer, 2006.

[11] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, and Thomas Lawrence. Parallel programming with Polaris. *Computer*, 29(12):78–82, 1996.

[12] R Bodik. Browsing web 3.0 on 3.0 watts: Why browsers will be parallel and implications for education, 2008.

[13] Mihai Budiu, Majd Sakr, Kip Walker, and Seth C Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *European Conference on Parallel Processing*, pages 969–979. Springer, 2000.

[14] Robert D Cameron, Ehsan Amiri, Kenneth S Herdy, Dan Lin, Thomas C Shermer, and Fred P Popowich. Parallel scanning with bitstream addition: An xml case study. In *European Conference on Parallel Processing*, pages 2–13. Springer, 2011.

[15] Robert D Cameron, Thomas C Shermer, Arrvindh Shriraman, Kenneth S Herdy, Dan Lin, Benjamin R Hull, and Meng Lin. Bitwise data parallelism in regular expression matching. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 139–150. IEEE, 2014.

[16] Scott O Campbell, Greg Adams, and Jeffrey M Braaten. Data compression of bit map images, March 11 1997. US Patent 5,611,024.

[17] Brian D Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming langauges. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[18] Patrick Carribault and Albert Cohen. Applications of storage mapping optimization to register promotion. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 247–256, 2004.

[19] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. infant: Nfa pattern matching on gpgpu devices. *ACM SIGCOMM Computer Communication Review*, 40(5):20–26, 2010.

[20] Yao-Jen Chang, Wende Zhang, and Tsuhan Chen. Biometrics-based cryptographic key generation. In *2004 IEEE International Conference on Multimedia and Expo (ICME)(IEEE Cat. No. 04TH8763)*, volume 3, pages 2203–2206. IEEE, 2004.

[21] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[22] Cristiana Chitic and Daniela 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.

[23] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[24] Marcelo Cintra and Diego R Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. *ACM SIGPLAN Notices*, 38(10):13–24, 2003.

[25] Marcelo Cintra and Diego R Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576, 2005.

[26] Marcelo Cintra, José F Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 13–24, 2000.

[27] Marcelo Cintra and Josep Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 43–54. IEEE, 2002.

[28] Romain E Cledat, Tushar Kumar, and Santosh Pande. Efficiently speeding up sequential computation through the n-way programming model. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 537–554, 2011.

[29] Sutapa Datta and Subhasis Mukhopadhyay. A grammar inference approach for predicting kinase specific phosphorylation sites. *PloS one*, 10(4):e0122294, 2015.

[30] Yanlei Diao, Peter Fischer, Michael J Franklin, and Raymond To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings 18th International Conference on Data Engineering*, pages 341–342. IEEE, 2002.

[31] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior-oriented parallelization. In *PLDI*, 2007.

[32] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[33] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *ACM SIGPLAN Notices*, volume 39, pages 71–81. ACM, 2004.

[34] Laurent Falquet, Marco Pagni, Philipp Bucher, Nicolas Hulo, Christian JA Sigrist, Kay Hofmann, and Amos Bairoch. The prosite database, its status in 2002. *Nucleic acids research*, 30(1):235–238, 2002.

[35] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 533–545, 2015.

[36] Min Feng, Rajiv Gupta, and Yi Hu. Spicec: Scalable parallelism via implicit copying and explicit commit. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, 2011.

[37] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved DFA for fast regular expression matching. *Computer Communication Review*, 38(5):29–40, 2008.

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

[39] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[40] Carlos García, Roberto Lario, Manuel Prieto, Luis Piñuel, and Francisco Tirado. Vectorization of multigrid codes using SIMD ISA extensions. In *IPDPS*, pages 8–pp, 2003.

[41] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. Speculative distributed CSV data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data*, pages 883–899. ACM, 2019.

[42] Todd J Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata. In *International Conference on Database Theory*, pages 173–189. Springer, 2003.

[43] Anshul Gupta and Vipin Kumar. The scalability of fft on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, 1993.

[44] Manish Gupta and Rahul Nim. Techniques for speculative run-time parallelization of loops. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Computer Society, 1998.

[45] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. A representation for bit section based analysis and optimization. In *International Conference on Compiler Construction*, pages 62–77. Springer, 2002.

[46] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[47] Maurice Herlihy and Eric 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.

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

[49] Ben Hertzberg and Kunle Olukotun. Runtime automatic speculative parallelization. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 64–73. IEEE, 2011.

[50] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[51] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The International Journal of High Performance Computing Applications*, 14(4):330–346, 2000.

[52] Kaixi Hou, Hao Wang, and Wu-chun 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.

[53] Paul G Howard and Jeffrey Scott Vitte. Parallel lossless image compression using huffman and arithmetic coding. In *Data Compression Conference, 1992. DCC'92.*, pages 299–308. IEEE, 1992.

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

[55] Heikki Hyyrö. Bit-parallel lcs-length computation revisited. In *In Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA*. Citeseer, 2004.

[56] Nobuhiro Ide, Masashi Hirano, Yukio Endo, Shin'ichi Yoshioka, Hiroaki Murakami, Atsushi Kunimatsu, Toshinori Sato, Takayuki Kamei, Toyoshi Okada, and Masakazu Suzuoki. 2.44-gflops 300-mhz floating-point vector-processing unit for high-performance 3d graphics computing. *IEEE Journal of Solid-State Circuits*, 35(7):1025–1033, 2000.

[57] Lin Jiang, Xiaofan Sun, Umar Farooq, and Zhijia Zhao. Scalable processing of contemporary semi-structured data on commodity parallel processors-a compilation-based approach. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2019.

[58] Lin Jiang and Zhijia Zhao. Grammar-aware parallelization for scalable xpath querying. *ACM SIGPLAN Notices*, 52(8):371–383, 2017.

[59] Peng Jiang and Gagan Agrawal. Combining simd and many/multi-core parallelism for finite state machines with enumerative speculation. In *Proceedings of the 22nd ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 179–191, 2017.

[60] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[61] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. Automatic vectorization of tree traversals. In *PACT*, pages 363–374, 2013.

[62] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[63] Blake Kaplan. Speculative parsing path. `http://bugzilla.mozilla.org`.

[64] Chuanle Ke, Lei Liu, Chao Zhang, Tongxin Bai, Bryan Jacobs, and Chen Ding. Safe parallel programming using dynamic dependence hints. In *ACM SIGPLAN Notices*, volume 46, pages 243–258. ACM, 2011.

[65] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast track: A software system for speculative program optimization. In *2009 International Symposium on Code Generation and Optimization*, pages 157–168. IEEE, 2009.

[66] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep 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.

[67] Shmuel Tomi Klein and Yair Wiseman. Parallel huffman decoding with applications to jpeg files. *The Computer Journal*, 46(5):487–497, 2003.

[68] Nick Koudas. Space efficient bitmap indexing. In *CIKM*, pages 194–201, 2000.

[69] Arvind Krishnaswamy and Rajiv Gupta. Dynamic coalescing for 16-bit instructions. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(1):3–37, 2005.

[70] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 2007.

[71] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 339–350. ACM, 2006.

[72] Vipin Kumar and Anshul Gupta. Analysis of scalability of parallel algorithms and architectures: A survey. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, pages 396–405, New York, NY, USA, 1991. ACM.

[73] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.

[74] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 330–335. IEEE, 1997.

[75] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: a fast JSON parser for data analytics. *Proceedings of the VLDB Endowment*, 10(10):1118–1129, 2017.

[76] Dan Lin, Nigel Medforth, Kenneth S Herdy, Arrvindh Shriraman, and Rob Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[77] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. Why gpus are slow at executing nfas and how to make them faster. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 251–265, 2020.

[78] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *ACM conference on International conference on supercomputing*, pages 273–282, 2013.

[79] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte regular expression matching with speculation. In *RAID*, 2009.

[80] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing dynamic programming through rank convergence. *ACM SIGPLAN Notices*, 49(8):219–232, 2014.

[81] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *ACM Sigplan Notices*, 44(6):166–176, 2009.

[82] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin 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.

[83] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 529–542, 2014.

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

[85] Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Annual Symposium on Combinatorial Pattern Matching*, pages 14–33. Springer, 1998.

[86] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences.* Cambridge university press, 2002.

[87] Mark Nelson and Jean-Loup Gailly. *The data compression book.* M & t Books New York, 1996.

[88] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. Demystifying automata processing: Gpus, fpgas or micron's ap? In *Proceedings of the International Conference on Supercomputing*, pages 1–11, 2017.

[89] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, volume 41, pages 132–143. ACM, 2006.

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

[91] Peter Ogden, David Thomas, and Peter Pietzuch. Scalable xml query processing using parallel pushdown transducers. *Proceedings of the VLDB Endowment*, 6(14):1738–1749, 2013.

[92] Yinfei Pan, Ying Zhang, Kenneth Chiu, and Wei Lu. Parallel xml parsing using meta-dfas. In *e-Science and Grid Computing, IEEE International Conference on*, pages 237–244. IEEE, 2007.

[93] Hannu Peltola and Jorma Tarhio. Alternative algorithms for bit-parallel string matching. In *International Symposium on String Processing and Information Retrieval*, pages 80–93. Springer, 2003.

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

[95] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[96] Manohar K Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, 2003.

[97] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe programmable speculative parallelism. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2010.

[98] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. Challenging sequential bitstream processing via principled bitwise speculation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 607–621, 2020.

[99] Junqiao Qiu, Zhijia Zhao, and Bin Ren. MicroSpec: Speculation-centric fine-grained parallelization for FSM computations. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 221–233. IEEE, 2016.

[100] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. Enabling scalability-sensitive speculative parallelization for fsm computations. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[101] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI*, 2005.

[102] Rodric M Rabbah, Ian Bratt, Krste Asanovic, and Anant Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. 2004.

[103] Arun Raman, Hanjun Kim, Thomas R Mason, Thomas B Jablin, and David I August. Speculative parallelization using software multi-threaded transactions. In *ACM SIGARCH computer architecture news*, volume 38, pages 65–76. ACM, 2010.

[104] Rajeev K Ranjan, Adnan Aziz, Robert K Brayton, Bernard Plessier, and Carl Pixley. Efficient bdd algorithms for fsm synthesis and verification. *IWLS95, Lake Tahoe, CA*, 253:254, 1995.

[105] Lawrence Rauchwerger and David A Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[106] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 153–167, 2015.

[107] Bin Ren, Gagan Agrawal, James R Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. SIMD parallelization of applications that traverse irregular data structures. In *CGO*, pages 1–10, 2013.

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

[109] Indranil Roy and Srinivas Aluru. Finding motifs in biological sequences using the micron automata processor. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 415–424. IEEE, 2014.

[110] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Flexamata: A universal and efficient adaption of applications to spatial automata processing accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–234, 2020.

[111] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

[112] Sekhar R Sarukkai, Pankaj Mehra, and Robert J Block. Automated scalability analysis of message-passing parallel programs. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):21–32, 1995.

[113] Priti Shankar, Amitava Dasgupta, Kaustubh Deshmukh, and B Sundar Rajan. On viewing block codes as finite automata. *Theoretical Computer Science*, 290(3):1775–1797, 2003.

[114] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 207–218. ACM, 2008.

[115] J Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[116] J Gregory Steffan and Todd C Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, pages 2–13. IEEE, 1998.

[117] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *ACM SIGPLAN Notices*, volume 35, pages 108–120. ACM, 2000.

[118] Arun Subramaniyan and Reetuparna Das. Parallel automata processor. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 600–612. IEEE, 2017.

[119] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *Proceedings of the 50th*

Annual IEEE/ACM International Symposium on Microarchitecture, pages 259–272, 2017.

[120] Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *ACM SIGPLAN Notices*, volume 38, pages 85–96. ACM, 2003.

[121] Chen Tian, Min Feng, and Rajiv Gupta. Speculative parallelization using state separation and multiple value prediction. In *ACM Sigplan Notices*, volume 45, pages 63–72. ACM, 2010.

[122] Chen Tian, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *ACM Sigplan Notices*, volume 45, pages 62–73. ACM, 2010.

[123] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicores. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 330–341. IEEE, 2008.

[124] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT*, pages 327–337, 2009.

[125] Robert 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.

[126] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *International workshop on recent advances in intrusion detection*, pages 116–134. Springer, 2008.

[127] Steven Wallace, Brad Calder, and Dean M Tullsen. Threaded multiple path execution. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, pages 238–249. IEEE, 1998.

[128] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. An overview of micron's automata processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–3, 2016.

[129] Ke Wang, Yanjun Qi, Jeffrey J Fox, Mircea R Stan, and Kevin Skadron. Association rule mining with the micron automata processor. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 689–699. IEEE, 2015.

[130] Zhen-Gang Wang, Johann Elbaz, Françoise Remacle, Raphaël David Levine, and Itamar Willner. All-DNA finite-state automata with finite memory. *Proc. Natl. Acad. Sci. U.S.A.*, 107(51):21996–22001, Dec 2010.

[131] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.

[132] Yang Xia, Peng Jiang, and Gagan Agrawal. Scaling out speculative execution of finite-state machines with parallel merge. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 160–172, 2020.

[133] Hongyi Xin, John Greth, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping. *Bioinformatics*, 31(10):1553–1560, 2015.

[134] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy 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. ACM, 2006.

[135] Xiaodong Yu and Michela Becchi. Gpu acceleration of regular expression matching for large datasets: exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 1–10, 2013.

[136] Ahmad Zandi, David G Stork, and James Allen. Compression of palettized images and binarization for bitwise coding of m-ary alphabets therefor, November 28 1995. US Patent 5,471,207.

[137] Zhijia Zhao and Xipeng Shen. On-the-fly principled speculation for FSM parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 619–630, 2015.

[138] Zhijia Zhao, Bo Wu, and Xipeng 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.

[139] Craig Zilles and Gurindar Sohi. Master/slave speculative parallelization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pages 85–96. IEEE, 2002.

[140] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng 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.