Challenging Sequential Bitstream Processing via Principled Bitwise Speculation

Junqiao Qiu
jqiu004@ucr.edu
University of California, Riverside

Lin Jiang
ljian006@ucr.edu
University of California, Riverside

Zhijia Zhao
zhijia@cs.ucr.edu
University of California, Riverside

Abstract

Many performance-critical applications traverse bitstreams with bitwise computations for better performance or higher space efficiency, such as multimedia processing and bitmap indexing. However, when these bitwise computations carry dependences, the entire bitstream traversal becomes serial, fundamentally limiting the scalability.

In this work, we show that bitstream-carried dependences are actually "breakable" in many cases, with the adoption of a systematic treatment – ***principled bitwise speculation*** (PBS). The core idea of PBS stems from an analogy drawn between bitstream programs and sequential circuits, both of which transform binary sequences. In this new perspective, it becomes natural to model the dependences in bitstream programs with finite-state machines (FSM), a basic model for sequential circuits. To achieve this, PBS features an assembly of static analyses that reason about bitstream programs down to the bit level to identify the bits causing dependences, then it treats the value combinations of dependent bits as states to construct FSMs. The modeling, for the first time, enables the use of FSM speculation techniques to parallelize bitstream programs. Basically, by leveraging the state convergence of FSMs, the values of dependent bits can be predicted with much higher accuracies. In cases the prediction fails, PBS tries to directly "rectify" the wrong outputs based on bitwise logic, minimizing the mis-speculation costs. In addition, FSM shows even higher execution efficiency than the original program in some cases, making itself an optimized version to accelerate serial bitstream processing. We prototyped PBS using LLVM. Evaluation with real-world bitstream programs confirms the effectiveness of PBS, showing up to near-linear speedup on multicore/manycore machines.

CCS Concepts. • **Computing methodologies → Parallel computing methodologies**; • **Theory of computation → Formal languages and automata theory**; • **Software and its engineering → Compilers**; • **Computer systems organization → Parallel architectures**.

Keywords. Bitwise computations, Bitstream, Speculation, Parallelization, Finite-state machine, Data-flow analysis

ACM Reference Format:
Junqiao Qiu, Lin Jiang, and Zhijia Zhao. 2020. 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 (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378461

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378461

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 [24], pattern matching [5], parsing [14, 27, 28], image compression [6, 50], and voice decoding [32]. For example, in bitstream-based text pattern matching [5], a text stream is first transposed into a set of bitstreams, then searched with bitwise manipulations. Thanks to the high efficiency of bitwise operations and bitwise parallelism, the bitwise text 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 the querying in document data stores [27, 28].

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 the left shift `<<` that defines the current bit with a bit to the right and the arithmetic addition `+` which may create a carry propagating over the calculations of the following bits.

Figure 1 shows an example bitstream program called *long bitstream addition* (`LBSAdd` [4]). 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

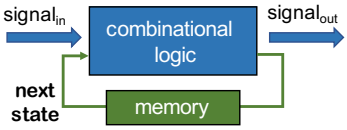
```
1  c = 0;
2  /* bitstream traversal */
3  for i = 0 to N
4    a = A[i];   b = B[i];
5    psum = a + b;
6
7    if psum == 0xff then
8      bubble = c & 1;
9    else
10     bubble = c & 0;
11
12   ta = a >> 7;  tb = b >> 7;  tp = psum >> 7;
13   tc = (ta & tb) | ((ta | tb) & (tp ^ 1));
14   C[i] = psum + c;
15   c = tc | (bubble & 1);
```

```
...10111101010011  A
         +
...01101110011010  B
         =
...00101011101101  C
```

Figure 1. Bitstream Processing Example (LBSAdd [4]).

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

bitstream program



sequential circuit

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

Figure 2. Bitstream Programs v.s. Sequential Circuits.

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 bitstream processing mainly focus on fine-grained vectorization with SIMD intrinsics (e.g., SSE2 and AVX512) [5, 14, 27, 28]. In spite of performance gains, there are limitations that hinder the productivity and efficiency of SIMD-based optimizations. First, coding with low-level SIMD intrinsics is notoriously difficult. It becomes even worse when the processing carries dependences. Take LBSAdd as an example, because the SIMD intrinsics adds numbers SIMD lane-wise, programmers have to manually resolve potential carries across SIMD lanes [4]. Second, these fine-grained dependence handling techniques cannot be extended to the coarse-grained level naturally, that is, partitioning the bitstream(s) across CPU cores, where the size of a bitstream partition goes far beyond the SIMD width, making the dependence handling a daunting task. For example, existing long bitstream addition can add up to 4096 bits [4]. In sum, existing bitstream processing heavily relies on programmers for fine-grained parallelism and fails to exploit the coarser-grained parallelism in the presence of dependences, restricting their scalability.

Overview of This Work. Complementary to the prior efforts, this work challenges the sequential bitstream processing with an automatic approach that enables speculative and coarse-grained parallelism for both non-SIMD and SIMD bitstream programs – *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 2), both of which transform binary sequences (*bits* [2] versus *pulses*). The memory in sequential circuits resembles the loop-carried dependences in bitstream programs, which are implicitly dictated 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 brought by the FSM-based modeling is the possibility of adopting speculative FSM parallelization [21, 29, 36, 52, 53] to bitstream processing. By leveraging the state convergence properties of FSMs, PBS can effectively predict future values of dependent bits, thus enabling speculative parallelism for bitstream processing. In cases the prediction fails, PBS offers a fast recovery mechanism that directly "rectifies" the wrong outputs with bitwise logic, instead of reprocessing the inputs, which reduces the mis-speculation costs. Besides prediction, we also observe that, in some cases, the constructed FSM runs more efficiently than the original bitstream program. In this way, even the serial bitstream processing can get performance improvement.

We prototyped PBS on LLVM infrastructure and evaluated it with a set of bitstream 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` [5]. Results show that, with

[1] Larger granularities (like `int` or `long`) can also be used. Without loss of generality, we use byte for easier illustration in this running example.
[2] For the lack of supports for bit arrays, programmers often use unsigned integer or char arrays to store bitstreams.

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.

Contributions. In sum, this work makes the following major contributions to bitstream processing.

- First, it offers a new perspective to sequential bitstream processing, bringing FSM-based dependence modeling to bitstream programs (Section 5).
- Second, it introduces a static analysis to rigorously find out the dependent bits in bitstream programs (Section 4).
- Third, it adopts FSM speculation to bitstream processing with customized mis-speculation handling (Section 6).
- Finally, it prototypes a speculation framework on LLVM, and confirms its effectiveness in accelerating real-world bitstream applications (Sections 7 and 8).

Next, we first provide the background of this work.

2 Background

This section introduces 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 also SIMD versions of these operators provided as the low-level intrinsics in 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 (i.e., *bitstreams*), such as, an audio record in multimedia processing [6, 32, 50], or a piece of encrypted file in cryptography [8]. In fact, even textual data can be converted into bitstreams to take advantages of SIMD intrinsics and bitwise parallelism. The idea is illustrated in Figure 3. Each byte of the textual data stream is transposed into eight individual bits stored in eight separated bitstreams. This text-to-bitstream transposition brings the paradigm of bitstream processing to many applications manipulating textual data streams, such as network intrusion detection [5] and semi-structured data analytics [27, 28].

Oftentimes, the bitstreams are processed in multiple rounds before they are eventually consumed. Figure 4 shows seven phases of bitstream transformations in XML parsing [28], where each phase 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 *base bitstreams* – B_0 to B_7 (see Figure 3). We refer to these bitstream transformations as *bitstream kernels* and their executions as *bitstream processing*.

Bitstream-Carried Dependences. As mentioned earlier, it is easy to introduce dependences into bitwise computations,

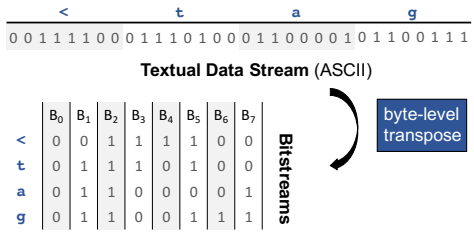


Figure 3. Text-to-Bitstream Conversion [5, 14, 27, 28].

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	. . . . .		
L_1 = ScanThru(L_0,C_0)	. . . . 1. . . . . 1. 1		
E_1 = L_1 & ¬C_1	. . . . .		

Figure 4. Bitstream Transformations in XML Parsing [28] (to make the bitstreams easier to read, zeros are marked as dots).

with the use of operators carrying multi-bit effects, such as various shift and arithmetic operators. Considering the XML parsing example in Figure 4, there are two transformations involving 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 each 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) & ¬C_0. In the code, these dependences appear as *loop-carried dependences*, a class of dependences that is often beyond the reach of existing parallelizing compilers [2, 48].

Efforts so far in optimizing bitstream processing focus on the use of SIMD intrinsics [5, 28]. However, this requires to manually redesign the bitstream processing algorithms to handle the dependences across SIMD lanes (e.g., 256 bits). Moreover, this solution cannot be naturally extended to the entire bitstream because of the limited width of SIMD lanes. To improve the scalability and the productivity, this work presents an automatic approach that enables coarse-grained parallelism for bitstream programs, called *principled bitwise speculation*. Next, we give an overview of this new approach.

3 Overview

The basic workflow of principled bitwise speculation (PBS) is summarized by Figure 5. At a high level, PBS consists of three basic modules: (i) *dependent bit analysis*, (ii) *bitstream program modeling*, and (iii) *runtime speculation*. Given a bitstream program, the dependent bit analysis conducts a series of data-flow analyses to identify the exact bits in the program variables that cause bitstream-carried dependences, referred to as *dependent bits*. The dependent bits are then fed into

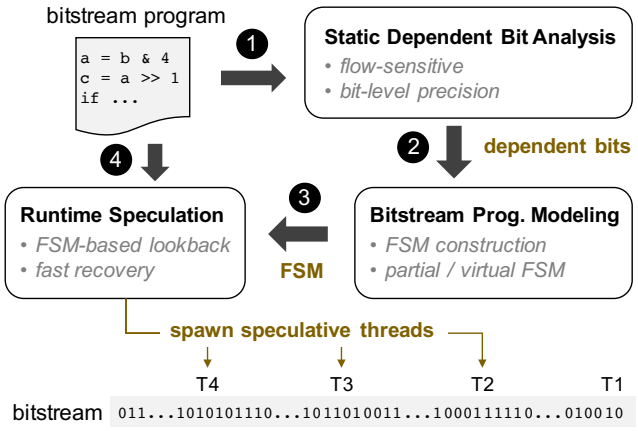


Figure 5. Workflow of Principled Bitwise Speculation (PBS)

the bitstream program modeling module, which generates a finite state machine (FSM) (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 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 over a partition of the bitstream(s). In cases the prediction fails, the runtime module also features an accelerated recovery based on the properties of bitstream programs. In the following, we present these three modules one by one.

4 Static Dependent Bit Analysis

Conventionally, 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 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.

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 $n_{[3:7]}$, actually do not contribute to the calculation of y – they are *ignored*. In another word, only $n_{[0:2]}$ are involved in this dependence, which we referred to as *dependent bits*. Similarly, L2 also indicates that the five most significant bits of y (i.e., $y_{[3:7]}$) are always zeros – they are *constants*. Therefore, a later use of y (not shown) does not necessarily depend on L2 regarding $y_{[3:7]}$. In both of the above scenarios, some instructions, by their semantics, may not have to access all the bits of variables [3]. Based on this intuition, we define the dependent bits more formally.

Definition 4.1. If the semantics of two instructions s_i and s_j requires to access the same bit of variable v, denoted as $v_{[k]}$, and at least one of them writes to $v_{[k]}$, then there exists a (data) dependence between s_i and s_j on $v_{[k]}$, where $v_{[k]}$ is the ***dependent bit***.

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

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[]`. Besides the dependences inside the loop body, there also exist dependences across loop iterations, known as *loop-carried dependences*. For instance, the L2 in the second iteration (reads n) depends on the L2 in the first iteration (writes to n). These (read-after-write) dependences are chained together across iterations, preventing any loop-level parallelizations.

However, if we look closer at the use of variable n in L3, only $n_{0:2}$ 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 $n_{3:7}$ can be ignored – their values will not affect the next instruction L3. Therefore, the loop-carried dependences 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.

Based on the above observation, the key is to find out those dependent bits that cause loop-carried dependences, which we address next with *dependent bit analysis*.

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

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 4.1, denoted as L_{main}, will be used as the running example.

Entry-Point Liveness Analysis. First, all the dependent bits should be *live* at the entry of the loop body of L_{main}, that is, they will be *semantically* used before they get redefined. Otherwise, if the bits are redefined first, they will not depend on values from prior iterations. However, it is challenging to perform bit-level liveness analysis due to the semantical variation of instructions. Existing liveness analysis [44] can reach bit sections, but not individual bits. For this reason, we first analyze the liveness of variables, then rely on a separate bitwise analysis (shown next) to prune semantically "killed" bits from live variables. As the liveness analysis only needs to compute live variables at the entry of the loop body, rather than at every program point, we refer to it as *entry-point liveness analysis*. As shown in Section 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 [3]. In the running example, this analysis finds that $\mathtt{out[i]}_{[3:7]}$ are zeros after `L3` and $\mathtt{n}_{[3:7]}$ 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 $\mathcal{B}_{unchanged}$, then the dependent bits $\mathcal{B}_{depen}$ can be calculated as follows.

$$\mathcal{B}_{depen} = (\mathcal{B}_{live} \cap \mathcal{B}_{unknown}) - \mathcal{B}_{unchanged} \quad (1)$$

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. Considering the running example,

- $\mathcal{B}_{live}$ = {$\mathtt{n}_{[0:7]}$`:L2`, $\mathtt{in[i]}_{[0:7]}$`:L2`}, where `:` is followed by the instruction(s) using the bits before redefinitions;
- $\mathcal{B}_{unknown}$ = {$\mathtt{n}_{[0:2]}$`:L2`, $\mathtt{in[i]}_{[0:2]}$`:L2`, $\mathtt{out[i]}_{[0:2]}$`:L2`}, where `:` indicates the instruction(s) using the bits;
- and $\mathcal{B}_{unchanged}$ = {$\mathtt{in[i]}_{[0:7]}$}.

Based on Equation 1, we have $\mathcal{B}_{depen}$ = {$\mathtt{n}_{[0:2]}$`:L2`}. Next, we explain how each of the three analyses work in detail.

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:

$$\text{LIVEENTRY}_{in} = \text{LIVEENTRY}_{out} - \text{DEF}(i) \cup \text{USE}(i) \quad (2)$$

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 [4]. After finishing the first instruction of the CFG, the analysis ends and the latest LIVEENTRY is outputed. Figure 6 shows an example analysis on a simplified CFG based on Figure 1. The IDs of instruction(s) using the corresponding live variables are also attached.

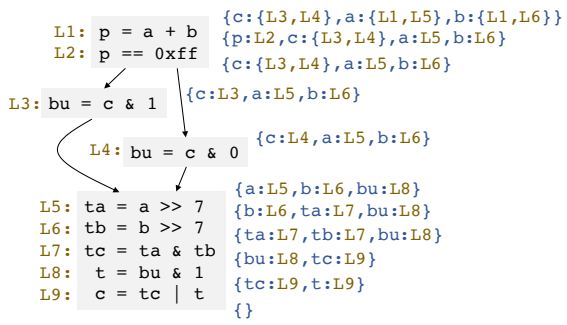


Figure 6. Entry-Point Liveness Analysis (backward).

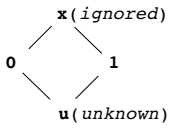
Algorithm for Bit-Status Analysis. To analyze bit statuses of different variables, we adopt an existing analysis that was developed for reducing the size of the synthesized circuits in reconfigurable hardware [3]. In this analysis, the bit status is defined as one of four cases: `unknown`, `0`, `1`, and `ignored`:

- `unknown` means the bit value cannot be inferred;

[4] Here, it assumes the variables used in the main loop body will not be used after the loop; Otherwise, a global iterative liveness analysis is needed.

- `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, thus it 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 [3].

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 $\mathcal{B}_{all}$ 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 7 shows a bit-status analysis on our running example. For limited space, only the variables with updated bit status(es) are shown.

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 7. The results of unchanged-bit analysis consist of all bits in `a` and `b`.

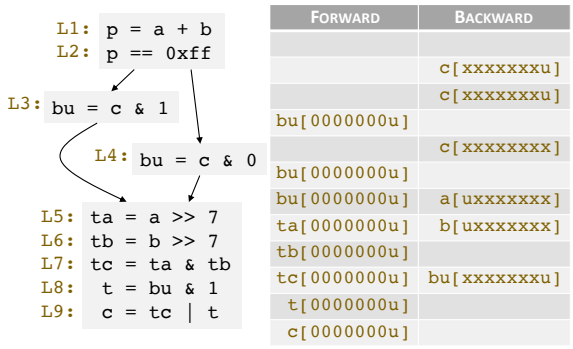


Figure 7. Bit Status Analysis (forward&backward).

Merging Results. Based on Equation 1, we can compute the dependent bit set $\mathcal{B}_{depen}$ for the running example. The calculation process is shown in Figure 8. In the end, it only includes $\texttt{c}_{[0]}$, which is used by instruction `L3`.

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.

5.1 FSM Construction

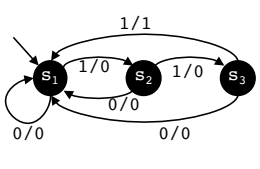
In a sequential circuit design scenario, some forms of FSMs (such as Mealy machines or Moore machines) are often first constructed to model the behaviors of the designed circuits. Then, by encoding the FSM states with binaries, the FSM is converted into a truth table. From there, the flip-flops will be determined and the circuit diagram will be generated. Figure 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 9-(b), where `CS/NS` represents the current/next state and `I/O` represents the input/output.

In the context of bitstream program modeling, we reverse the FSM-to-truth table process. That is, we first 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

$$\mathcal{B}_{depen} = (\mathcal{B}_{live} \cap \mathcal{B}_{unknown}) - \mathcal{B}_{unchanged}$$
$$= \{c_{[0:7]}:\{L3,L4\}, a_{[0:7]}:\{L1,L5\}, b_{[0:7]}:\{L1,L6\}\} \cap \{c_{[0]}:\{L3\}, a_{[0:7]}:L1, a_{[7]}:L5, b_{[0:7]}:L1, b_{[7]}:L6, \cdots\} - \{a_{[0:7]}, b_{[0:7]}\}$$
$$= \{c_{[0]}:\{L3\}, a_{[0:7]}:L1, a_{[7]}:L5, b_{[0:7]}:L1, b_{[7]}:L6\} - \{a_{[0:7]}, b_{[0:7]}\} = \{c_{[0]}:\{L3\}\}$$

Figure 8. Calculation of Dependent Bits for Example in Figures 7 and 6 (for $\mathcal{B}_{unknown}$, only the relavent elements are shown).



(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 9. Example FSM (Mealy machine) and Truth Table.

input, output, and memory element (next-state logic). In our context, we use the 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 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 output 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 in the truth table N_{row}.
(c) For each row in 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 under 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 10-(a) shows the truth table generated for our running example (see Figure 1) based on the dependent bit found in Section 4.3. In this case, the numbers of input and output bits are both eight and there is only one dependent bit. The size of the table is $2^{17} \times 26$, which is quite large even for offline generation. We will discuss how to address it shortly in Section 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 10-(b) shows the FSM constructed based on the truth table in Figure 10-(a). For space limits, only some representative transitions are shown.

It is not surprising that the FSM and the constructed truth table for our running example are essentially those used for designing a byte-level hardware adder. Essentially, the hardware adder and the bitstream program are equivalent in terms of functionality. However, the FSM-based modeling of bitstream programs does not require programmers to be familiar with hardware design and redevelop the solution from an FSM point of view. Moreover, the logic of bitstream programs could be quite complex, 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 8), because of the large numbers of dependent bits and/or input bits. We address this issue next.

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

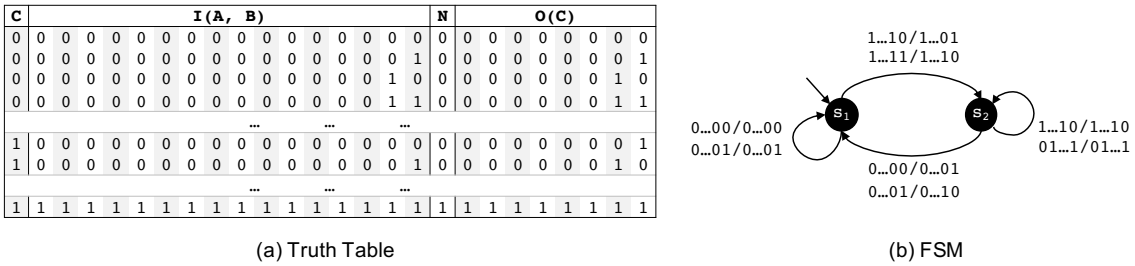


Figure 10. Truth Table and FSM (partially shown) for the Running Example (see Figure 1 and Section 4.3).

some rows potentially missing. Then, following the truth table-to-FSM construction, a partial FSM is created.

Virtual FSMs. Another way to address oversized FSMs is to completely bypass the physical FSM construction. Instead, it simulates FSM transitions with the executions of bitstream program. Basically, 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 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.

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.

6.1 FSM-based Speculation

The basic idea of speculative FSM execution stems from an interesting observation [23], that is, a future FSM state can be effectively predicted by running the FSM on a small piece of input prior to the prediction position, a technique later formalized and referred to as *lookback* [53]. Figure 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

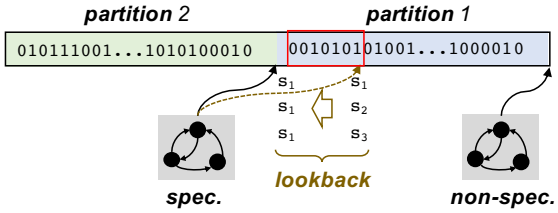


Figure 11. Lookback-based Speculative FSM Parallelization.

in Figure 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 prediction fails, a reprocessing with the correct starting state is needed 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 [53]).

To adopt FSM speculation for 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 the speculative execution. The high-level workflow remains similar to that in Figure 11, except that the speculative FSM execution becomes the speculative execution of bitstream program. 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 8) shows that a short FSM-based lookback often yields high speculation accuracies for long bitstream addition and many other bitstream programs.

Speculation with Partial/Virtual FSMs. For partial FSMs, the lookback 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. For each value combination of 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").

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 of the prior partition. If the verification fails, the corresponding input partition has to be reprocessed with the correct starting state. Despite some optimization [53] for stopping the reprocessing earlier, the reprocessing cost, in general, can significantly compromise the speculation benefits [37].

The above issue may be alleviated in speculative bitstream processing. Unlike FSMs, 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 12). If we can prove this as a *property* of the bitstream program, we can directly rectify the incorrect output bitstream, rather than reprocessing the input bitstream.

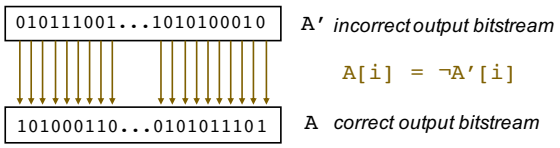


Figure 12. Example Fast Recovery from Misspeculation.

In fact, we can prove 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 call this technique *property-based fast recovery*.

7 Implementation

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

Static Analyzer. We prototyped dependent bit analysis on the 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 [5] to find out the loop induction variable, followed by an SCC (strongly connected components) analysis pass [6] to locate the body of main loop. After these preparation, the pass starts the three sub-analyses mentioned in Section 4.3 and merges the analysis results to produce the dependent bits for the given bitstream program. More details regarding the analyzer, including some of its potential limitations, will be discussed in Section 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 speculative parallelization, we use the `Pthread` library for its customizable thread settings. By default, the runtime creates the same number of threads as the number of CPU cores. The default length for the lookback (in number of bits) is set to $2 \times |\mathcal{B}_{in}|$. Using a multiple of input vector size avoids the aligning complexity. As to the property-based fast recovery, the current prototype focuses on testing the hypothesis of NOT $\neg$ relation. More hypothetic relations are planned to be added in the future versions.

8 Evaluation

This section evaluates the principled bitwise speculation, with a focus on the performance benefits.

8.1 Methodology

Benchmarks. To facilitate the evaluation, we collected eight bitstream kernels from multiple applications, ranging from semi-structured data processing [27, 28] and text pattern matching [5] to multimedia [26, 39] and bioinformatics [49]. They are listed in Table 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 end-to-end benefits, we also evaluate PBS with an open-source high-performance regular expression engine, called `icgrep` [5] (see more details in Section 8.4).

Evaluation Platform. We mainly ran our experiments on a 64-core machine equipped with an Intel Xeon Phi 7210

[5] https://llvm.org/doxygen/classllvm_1_1Loop.html
[6] https://llvm.org/doxygen/classllvm_1_1scc__iterator.html

Table 1. Bitstream Kernels in Evaluation.

Abbreviation	Brief Description	SIMD
shd_srs	Shift-hamming-distance filter kernel [49]	No
802_11a	IEEE 802.11a convolutional encoder [39]	No
8b10b_cal	IBM 8bit/10bit block encoder [39]	No
g721_upd	G.721 voice compression kernel [26]	No
quoteStr	JSON bitmap indexing from Mison [27]	No
scanThru	Ending index construction from icgrep [5]	Yes
matchStar	Matching "*" in regex from icgrep [5]	Yes
xmlParser	XML parsing kernel from Parabix [28]	Yes

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" flag enabled.

For bitstream kernels, we measured the time spent on the main loop, while for the regular expression engine, the end-to-end running time was collected. All timing results reported are the average from 10 repetitive runs.

8.2 Static Analysis and Modeling

To prepare for the static analysis, we inlined functions that are called inside the main loops to avoid precision loss from the inter-procedural analysis. The second column of Table 2 reports the number of LLVM IR instructions in the main loop of each kernel, where the static analyses are performed.

Table 2. Static Analysis Results (#Instr: number of instructions; #DB/#IB/#OB: numbers of dependent/input/output bits).

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. As shown in Table 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

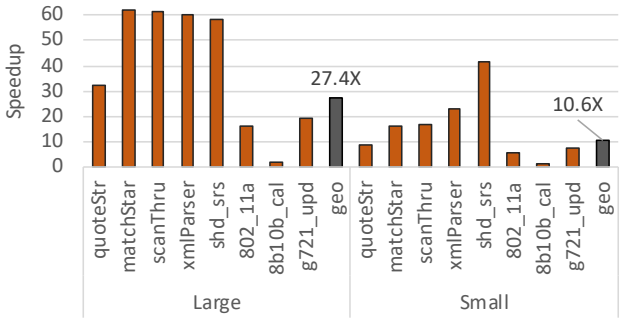


Figure 13. Speedup of Parallelized Kernels (64-core).

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 assumption is 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. Based on the number of dependent bits reported in Table 2, 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 is still impractical to generate the FSMs physically. For this reason, we adopt partial and virtual FSMs (see Section 5.2). In particular, we generate a partial FSM for kernel 8b10b_cal, which is one of the kernels with the smallest truth table. Moreover, 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 the bitstream program on a training input, we generated a 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.

8.3 Speculative Parallelization

This section evaluates the parallel performance of FSM-based speculation (Section 6) on the bitstream kernels, including their speedups, speculation accuracies, and scalabilities.

Speedup. Figure 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, the four kernels obtain 100% speculation accuracies (more discussions 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 some remedies for this kernel later in this section. For `quoteStr`, 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 `quoteStr`), 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, `quoteStr` passed the testing of NOT relation hypothesis (see Section 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, `quoteStr` achieves much higher speedup than the other benchmark `8b10b_cal` which also suffers from low speculation accuracy (see Figure 13).

***Optimizations for* `8b10b_cal`**. There are two optimizations applicable to `8b10b_cal`. First, its FSM model is found to run even faster than the original program, thus we can use the FSM to replace the bitstream program. Second, the FSM has only two states. In this case, we may aggressively execute both states, an existing FSM parallelization technique known as *enumerative parallelization* [29]. With both optimizations, we observed a 43.7X speedup on the 64-core machine, instead of 2.3X as reported earlier in Figure 13.

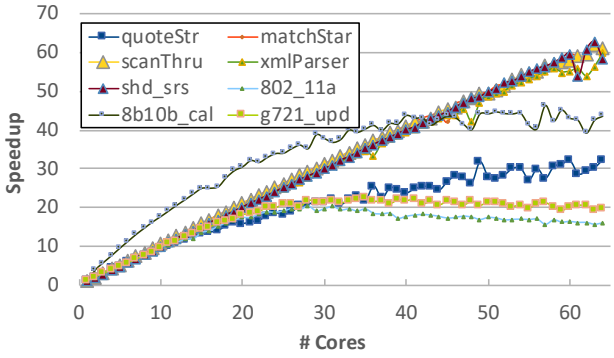


Figure 14. Scalability (Large Inputs on 64-core Machine).

Scalability. As demostrated in Figure 14, four kernels present near-linear speedups up to 64 cores (`matchStar`, `scanThru`, `xmlParser`, and `shd_srs`). `quoteStr` 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 raito.

8.4 Case Study: Enabling Data-Parallel `icgrep`

To confirm the benefits of PBS on full-fledged applications, we experimented it with `icgrep` [5], 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 from the `icgrep` paper [5] for our evaulation (see Table 3). The input textual streams are collected from a Linux server using the `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 3). In general, 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 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 fast convergence properties of their virtual FSMs. The sub-linear speedups are due to the generation of output bitstreams, which are saved in cases the users want to print out the matched contexts.

9 Related Work

As we have introduced bitstream processing in Section 2, this section focuses on other relevant topics, including bit-level analysis and speculative parallelization.

Table 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	([^@]+)@([^@]+)	22	18.1X
4	(([a-zA-Z][a-zA-Z0-9]*)://|mailto:)([^/]+)(/[^]*)?|([^@]+)@([^@]+)	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

Bit-Level Analysis. Existing research on bit-level analysis is mainly for saving hardware resources, with applications to multimedia processing and telecommunications [1, 3, 16, 25, 43, 44]. For example, Budiu and others [3] 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 [43] 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 [16] introduced a program representation to facilitate expanding traditional program analysis to the subword level. Following this work, Tallam and Gupta [44] designed a bitwidth-aware algorithm for global register allocation, showing 10%-50% reduction in register uses when compared to the traditional approaches.

Our 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.

Bit-Level Parallelism. Besides code vectorization, there are also many efforts in exploiting bit-level parallelism in specific applications [18, 42], especially for string matching [30, 31, 33] and semi-structured data indexing [27, 28]. In particular, Carribault and Cohen [7] 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.

Speculative Parallelization. There exists a rich body of work on speculative parallelization [9–13, 15, 17, 22, 35, 40, 41, 47, 54]. For space limits, we briefly introduce some of them. As an early framework, LRPD [41] was proposed for speculative loop execution with a parallel data dependence test that checks cross-iteration dependences. When the test fails, the loop would be reexecuted in serial. To better take advantage of multicore processors, Ding and others [12] introduced behavior oriented parallelization (BOP) which uses some partial information of program behaviors to assist the parallelization. For a similar purpose, copy-or-discard execution model [47] was introduced, with improvements using multiple value prediction [45] and supports of dynamic data structures [46]. In addition, there are also proposals of programming language constructs [35], speculation design for irregular applications [34], as well as compilers with speculative execution supports [38]. Some recent works also target speculative parallelization of semi-structured data stream processing, but at the byte level rather than bit level, like speculative HTML parsing [51] and speculative path query processing of XML/JSON streams [19, 20].

In sum, the prior work laid the foundations for speculative parallelization. However, for bitstream processing, programs expose special challenges to the speculative parallelization with bit-level data manipulations. It is non-trivial to adopt most of the above techniques with high effectiveness. Take the long bitstream addition as an example, the value of the dependent variable changes across loop iterations following non-trivial patterns. Traditional value predictors, such as the last-value and stride-based predictors, in fact, perform no better than a random predictor in such cases. By modeling bitstream programs with FSMs and exploiting their inherent *state convergence properties* [21, 36, 52, 53], this work provides a systematic treatment to the speculative parallelization of bitstream programs, achieving high speculation accuracies and low-misspeculation costs.

10 Conclusion

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, the *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.

Acknowledgments

The authors would like to thank the anonymous reviewers for their time and comments and Dr. Rajiv Gupta for bringing the closely related work. This material is based upon the work supported in part by National Science Foundation (NSF) Grants No. 1565928 and 1751392.

References

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

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

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

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

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

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

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

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

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

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

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

[12] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. 2007. Software behavior oriented parallelization. In *ACM SIGPlan Notices*, Vol. 42. ACM, 223–234.

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

[14] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative Distributed CSV Data Parsing for Big Data Analytics. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 883–899.

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

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

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

[18] Heikki Hyyrö. 2004. Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*. Citeseer, 16–27.

[19] Lin Jiang, Xiaofan Sun, Umar Farooq, and Zhijia Zhao. 2019. 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*. 79–92.

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

[21] Peng Jiang and Gagan Agrawal. 2017. Combining SIMD and Many/Multi-core parallelism for finite state machines with enumerative speculation. *ACM SIGPLAN Notices* 52, 8 (2017), 179–191.

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

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

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

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

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

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

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

[29] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 529–542.

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

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

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

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

[34] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *ACM Sigplan Notices*, Vol. 46. ACM, 12–25.

[35] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. 2010. Safe programmable speculative parallelism. In *ACM Sigplan Notices*, Vol. 45. ACM, 50–61.

[36] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. Microspec: Speculation-centric fine-grained parallelization for FSM computations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 221–233.

[37] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. 2017. Enabling scalability-sensitive speculative parallelization for FSM computations. In *Proceedings of the International Conference on Supercomputing*. ACM, 2.

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

slices. In *ACM Sigplan Notices*, Vol. 40. ACM, 269–279.
[39] Rodric M Rabbah, Ian Bratt, Krste Asanovic, and Anant Agarwal. 2004. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. (2004).
[40] Arun Raman, Hanjun Kim, Thomas R Mason, Thomas B Jablin, and David I August. 2010. Speculative parallelization using software multi-threaded transactions. In *ACM SIGARCH computer architecture news*, Vol. 38. ACM, 65–76.
[41] Lawrence Rauchwerger and David A Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (1999), 160–180.
[42] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. 2011. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research* 38, 2 (2011), 571–581.
[43] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. 2000. Bidwidth analysis with application to silicon compilation. In *ACM SIGPLAN Notices*, Vol. 35. ACM, 108–120.
[44] Sriraman Tallam and Rajiv Gupta. 2003. Bitwidth aware global register allocation. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 85–96.
[45] Chen Tian, Min Feng, and Rajiv Gupta. 2010. Speculative parallelization using state separation and multiple value prediction. In *ACM Sigplan Notices*, Vol. 45. ACM, 63–72.
[46] Chen Tian, Min Feng, and Rajiv Gupta. 2010. Supporting speculative parallelization in the presence of dynamic data structures. In *ACM Sigplan Notices*, Vol. 45. ACM, 62–73.
[47] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. 2008. Copy or discard execution model for speculative parallelization on multicores. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 330–341.
[48] 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. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices* 29, 12 (1994), 31–37.
[49] Hongyi Xin, John Greth, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. 2015. Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping. *Bioinformatics* 31, 10 (2015), 1553–1560.
[50] Ahmad Zandi, David G Stork, and James Allen. 1995. Compression of palettized images and binarization for bitwise coding of M-ary alphabets therefor. US Patent 5,471,207.
[51] Zhijia Zhao, Michael Bebenita, Dave Herman, Jianhua Sun, and Xipeng Shen. 2013. HPar: A practical parallel parser for HTML–taming HTML complexities for parallel parsing. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4 (2013), 1–25.
[52] Zhijia Zhao and Xipeng Shen. 2015. On-the-fly principled speculation for FSM parallelization. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 619–630.
[53] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 543–558.
[54] Craig Zilles and Gurindar Sohi. 2002. Master/slave speculative parallelization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.* IEEE, 85–96.

A Artifact Appendix

A.1 Abstract

This artifact contains the source code of a basic version of PBS and some benchmarks evaluated in this paper, including an LLVM dependent bit analysis pass, the parallel versions of bitstream kernels, and a regular expression engine `icgrep`. For each kernel, there are 5 small inputs (10MB each) and 5 large inputs (300MB each). In addition, there are 10 large inputs, up to 300MB each, collected from a Linux server using `tcpdump`, for the evaluation of `icgrep`. At last, this artifact includes bash scripts to compile the source code and generate some of the results reported in the paper.

Considering the performance measurements, the artifact needs to run on Intel Xeon Phi processor (Knights Landing) with GCC and Pthread supports. Since the static analyzer was implemented on LLVM 9.0.0, the artifact also needs the environment for installing the corresponding version of LLVM and Clang. Moreover, all source code was tested in the environment of CentOS 7.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Principled Bitwise Speculation.
- **Program:** Static analyzer and parallelized bitstream kernels.
- **Compilation:** GCC 4.8.5 and LLVM 9.0.0.
- **Binary:** The source code of PBS in a basic version, parallelized bitstream kernels, icgrep engine, and the scripts are included to generate binaries.
- **Data set:** All data were collected from the corresponding real-world applications. For each kernel and regular expression, there are totally 10 datasets with size of 10MB or 300MB.
- **Run-time environment:** The artifact has been developed and tested on Linux (CentOS 7) environment. The source code of static analyzer was compiled by Clang 9.0.0 and all the other code was compiled by GCC with Pthread.
- **Hardware:** The artifact is supposed to run on Intel Xeon Phi 7210 Processor (Knights Landing/KNL, 1.3GHz).
- **Execution:** Bash scripts are included for execution.
- **Output:** Results mainly include static analysis results and the execution time of each kernel and each regular expression.
- **How much disk space required (approximately):** 10GB
- **How much time is needed to complete experiments (approximately):** It takes about 1.5 to 2 hour, assuming LLVM and other required tools have been installed.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to access.

A zip file named `Artifact.zip`, containing the source code, scripts, as well as the datasets, is available as a public repository on Zenodo (https://doi.org/10.5281/zenodo.3610556)

A.3.2 Hardware dependencies.

We recommend testing our code on an Intel Xeon Phi architecture (Intel Xeon Phi 7210 with 1.3GHz in particular). Please make sure there is at least 15GB space available (for our artifacts and LLVM).

A.3.3 Software dependencies.

We assume the artifact runs on CentOS 7, but other similar Linux distributions should also work. We also need GCC 4.8.5 and LLVM 9.0.0 (but note that, to install LLVM 9.0.0, GCC version should be at least 5.1). To compile and run the source code with the scripts, we need Cmake 3.14.6 and Python 2.7.5. The bitstream kernels included in the artifact implement a shift-hamming-distance filter,

an IEEE 802.11a convolutional encoder, the IBM 8bit/10bit block encoder, and a JSON bitmap indexer. For more details about these kernels and their original applications, please check our paper and references. The case study application *icgrep* is an open-source regular expression engine (http://www.icgrep.com). The version we used in the artifact is the release version 1.0.

A.3.4 Data sets.

Datasets are included in this artifact for testing. They are collected from their corresponding real-world applications. They are located in the directory `Artifact/AE/data`.

A.4 Installation

After downloading and unzipping the artifact file, make sure LLVM 9.0.0 has been installed (for the static analyzer). We provide a bash script `Artifact/LLVM/install.sh` for LLVM installation.

```
$ cd Artifact/LLVM && bash install.sh
```

Remember to follow the notes to add path to your `~/.bashrc` after finishing the execution of the above script.

A.5 Experiment workflow

We provide a bash script `Artifact/AE/scripts/run.sh`, which is used for finishing the testing in one-step execution. Type the following command for testing

```
$ cd Artifact/AE/scripts && bash run.sh
```

Generating all results takes about 2 hours. There are three major parts, including conducting static analysis, performance testing for bitstream kernels, and performance evaluation of the case study application. It is suggested to execute these three parts separately, instead of using the one-step script. More details about executing different parts can be found in the scripts.

A.6 Evaluation and expected result

After running the bash script, the results will be outputted to the command line window, in three parts. First, the static analysis results will appear, showing the bit status of variables causing dependences in each kernel. Then, parts of the results (performance of 3 bitstream kernels) in Figure 13 and 14 will appear, showing the speedup (using 64 cores) over the sequential version, and the speedup curves for their scalabilities. Finally, it will report the maximum speedups reached for parallelizing `icgrep` over regular expression 2, 4, and 6, as shown in Table 3.

A.7 Experiment customization

Optionally, there are also three groups of scripts located under folder `Artifact/AE/scripts`, which can help generate the results mentioned above separately. Please follow the commands shown in these scripts to make your own compilation and testing. For example, when testing `icgrep`, you can follow the patterns shown in script `Artifact/AE/scripts/caseStudy_execute.sh` to test any other regular expression.