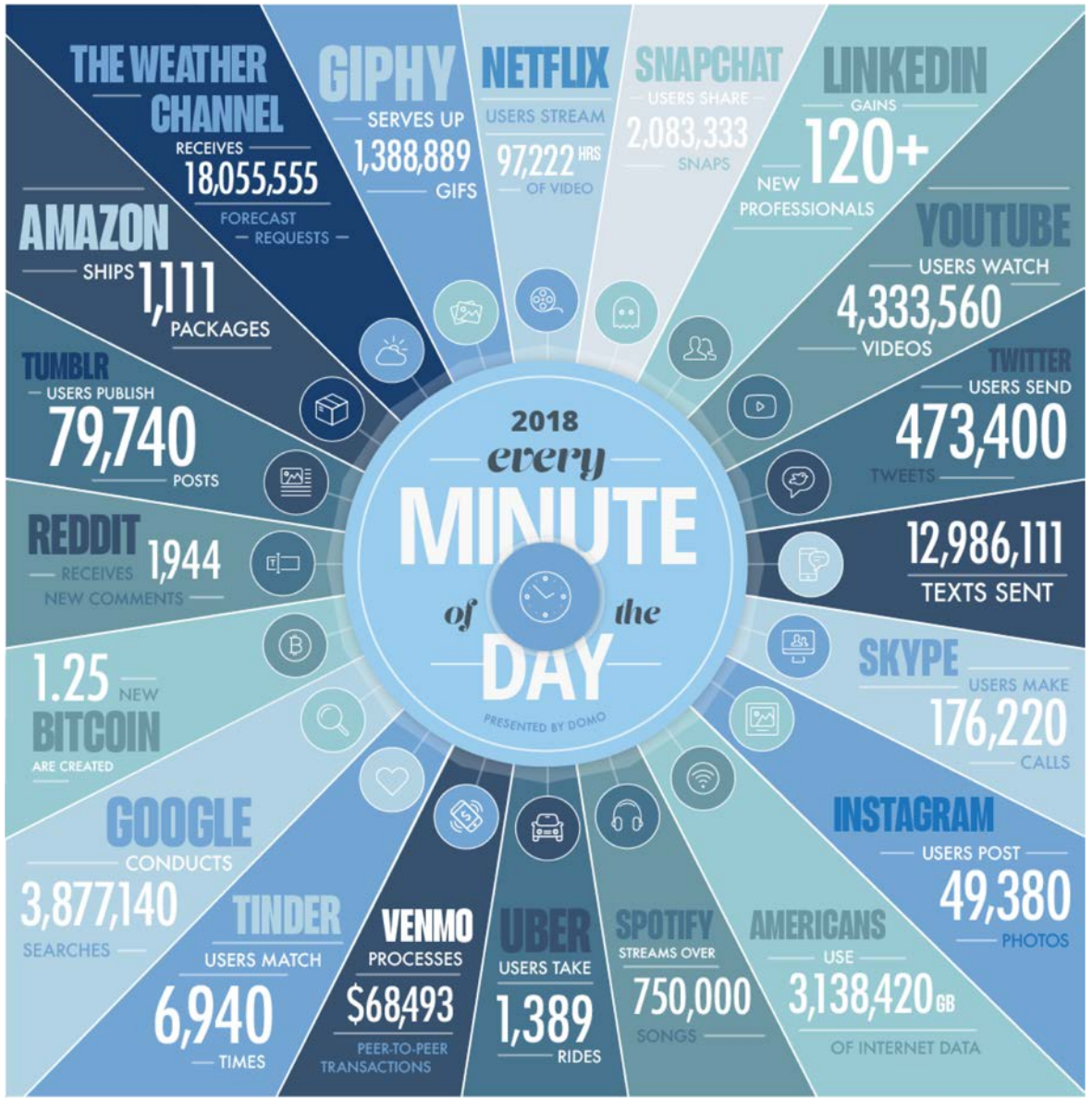
Debugging Support for Pattern-Matching Languages and Accelerators

Matthew Casias†, **Kevin Angstadt***, Tommy Tracy II†,
Kevin Skadron†, Westley Weimer*

†Department of Computer Science, University of Virginia
*Computer Science and Engineering, University of Michigan

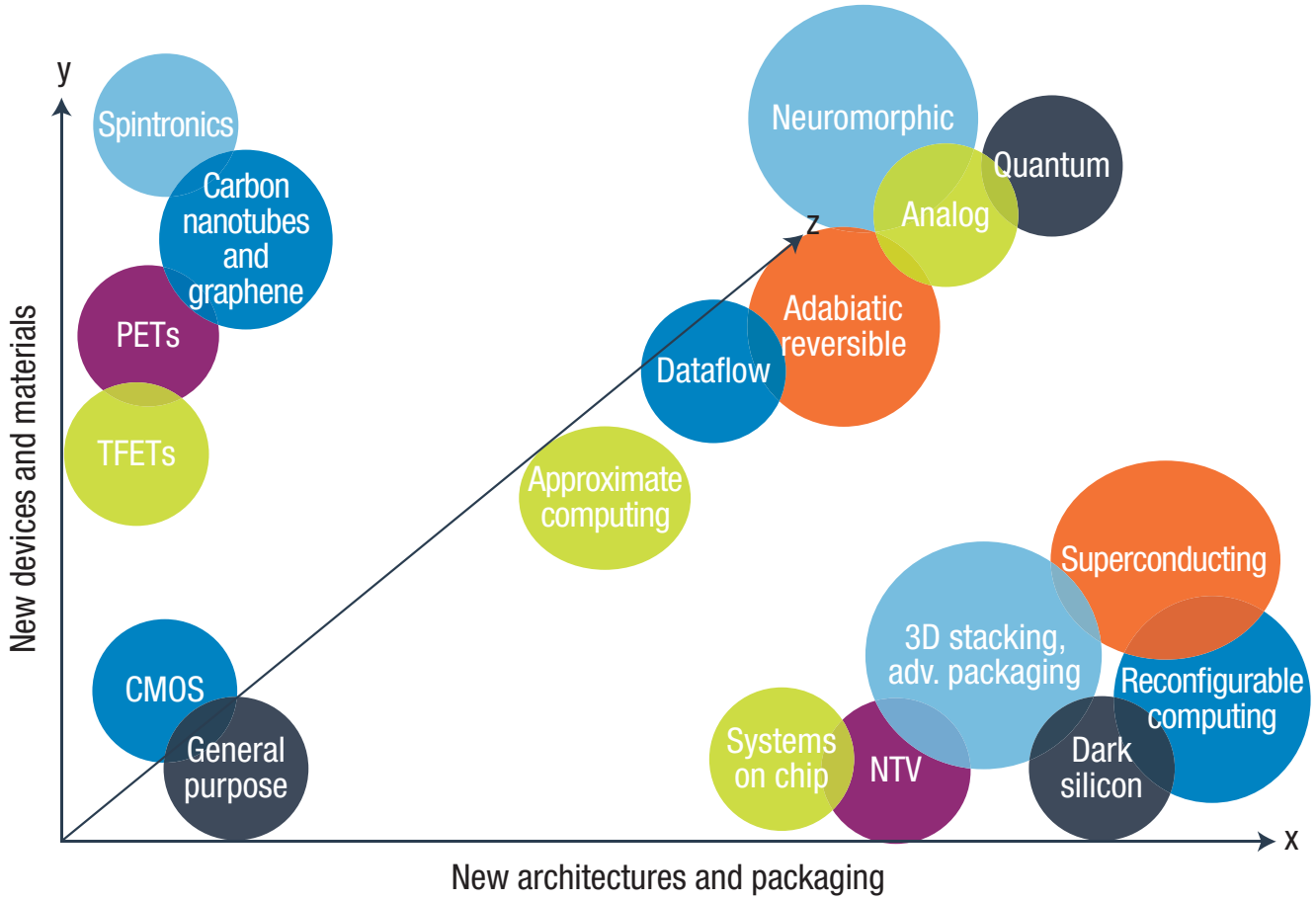
This work was funded in part by: NSF grants CCF-1629450, CCF-1619123, and CNS-1619098; the Jefferson Scholars Foundation; the Virginia-North Carolina Louis Stokes Alliance for Minority Participation; and support from CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

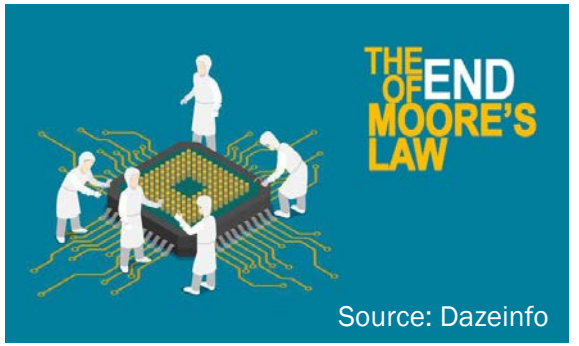


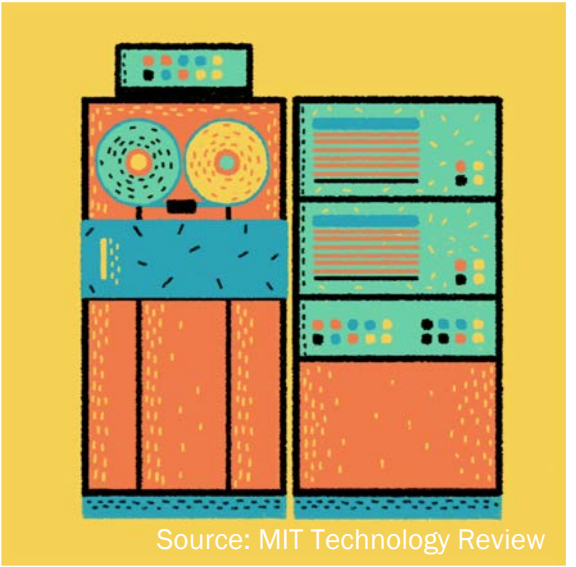
> By 2020, it's estimated that for every person on earth, 1.7MB of data will be created every second.

DOMO, "Data Never Sleeps 6.0". 2018

Physical Limits Spark Creativity

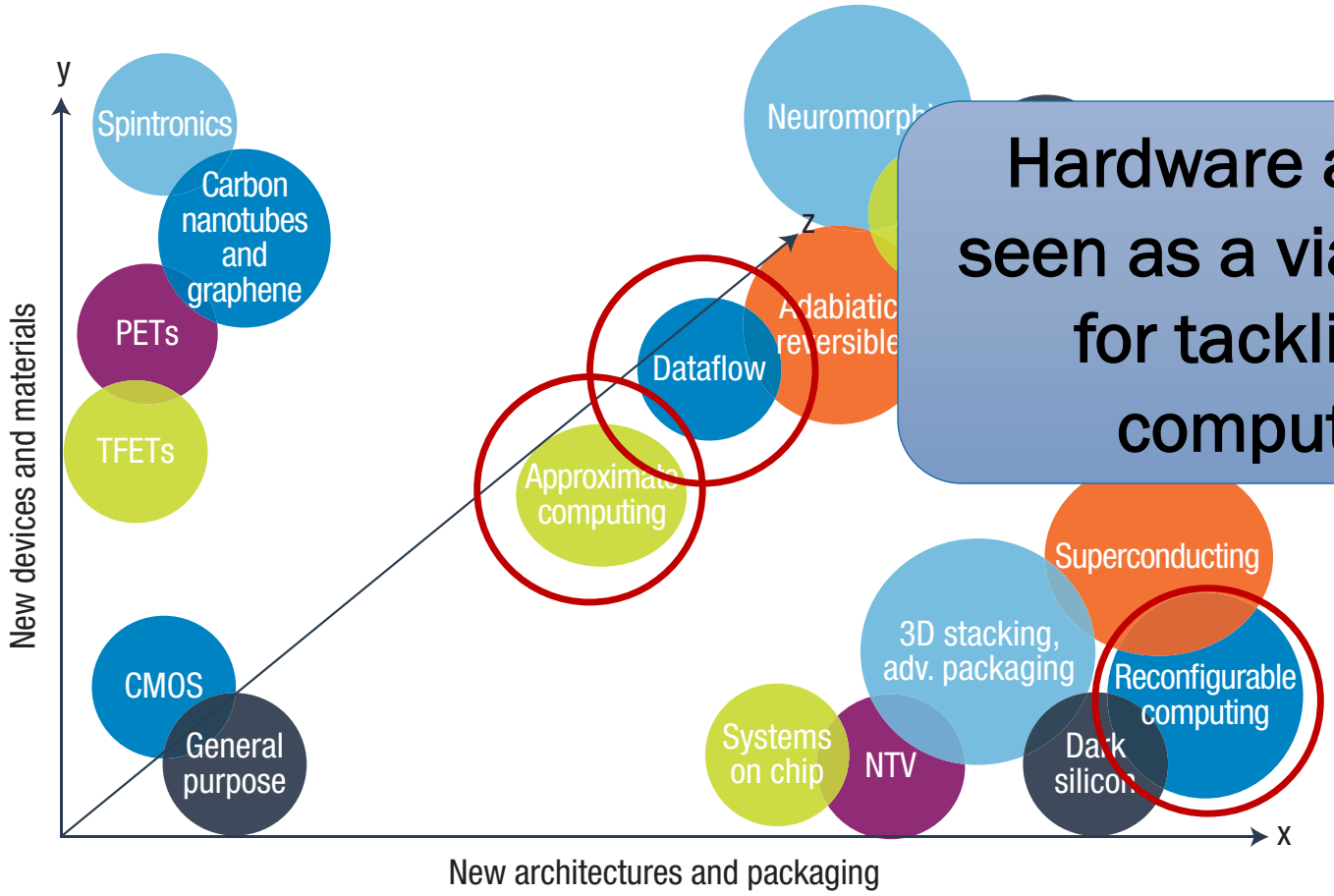






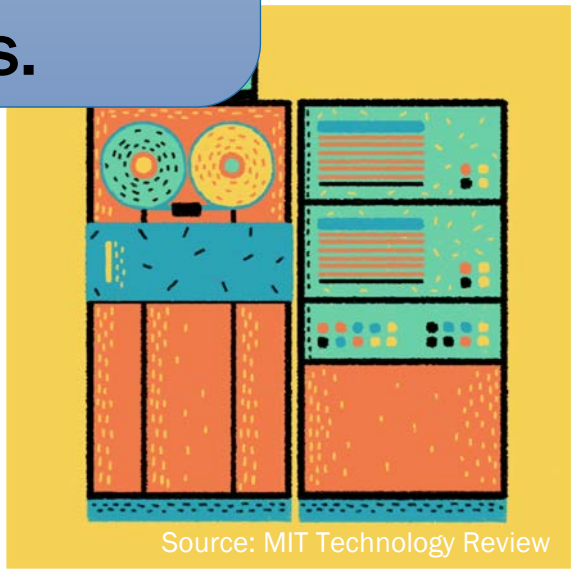
J. M. Shalf and R. Leland, "Computing Beyond Moore's Law". IEEE Computer, 2015.

Physical Limits Spark Creativity



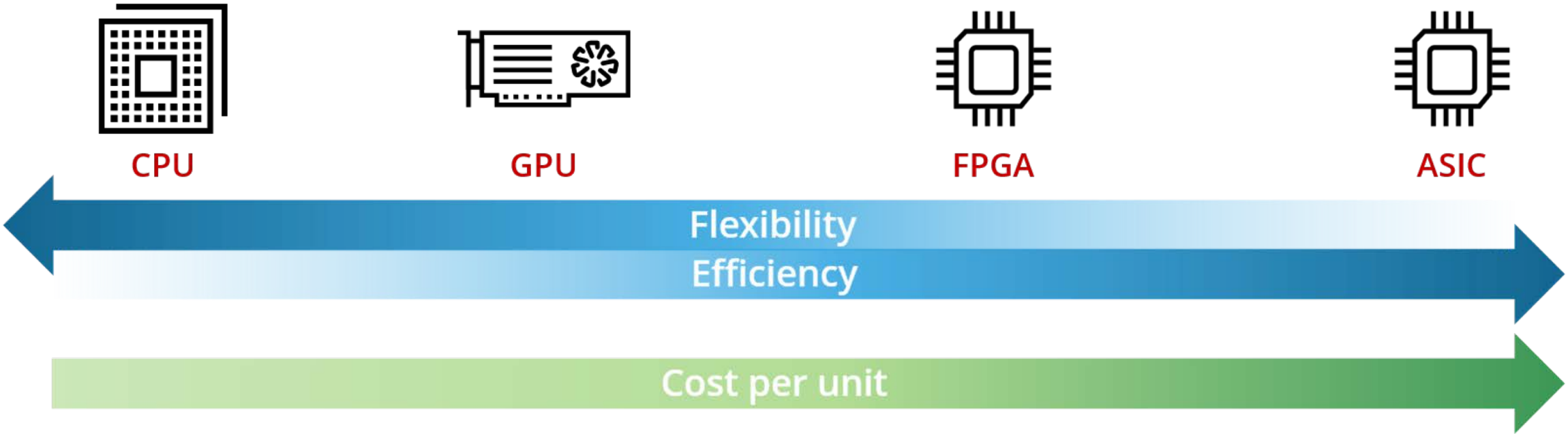
Hardware accelerators are seen as a viable path forward for tackling increasing compute demands.

Source: Dazeinfo


Source: MIT Technology Review

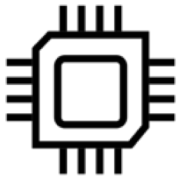
J. M. Shalf and R. Leland, "Computing Beyond Moore's Law". IEEE Computer, 2015.

How Accelerators Help

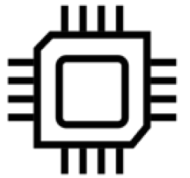


A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018

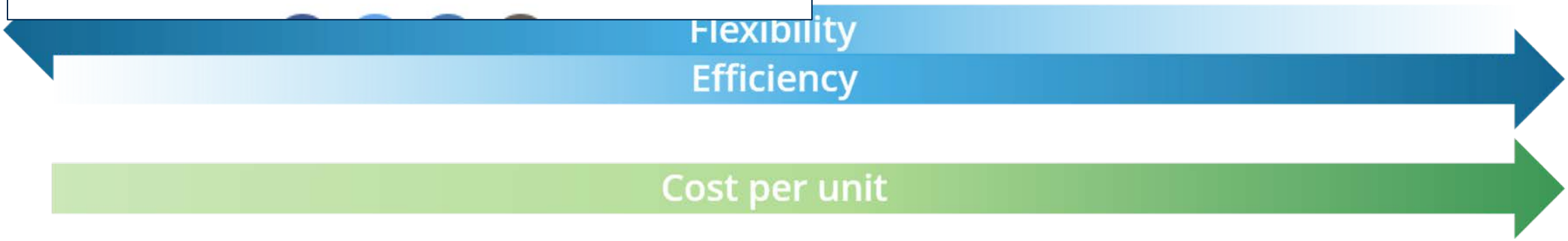
How Accelerators Help



FPGA

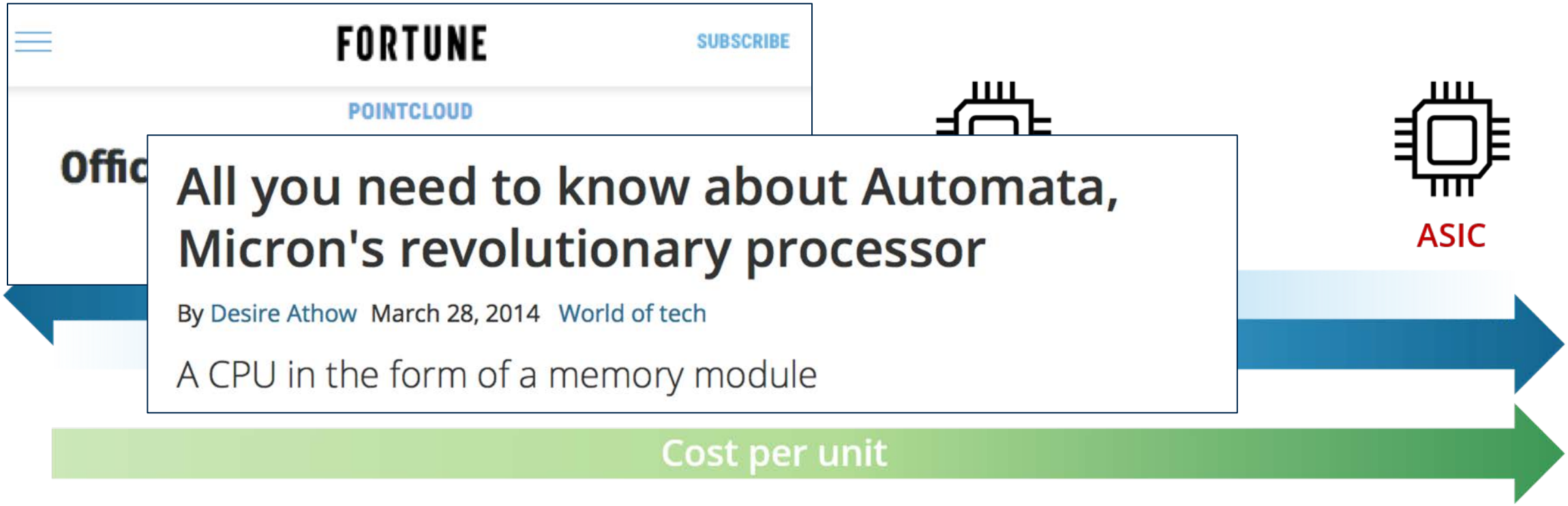


ASIC



A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018

How Accelerators Help



A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018

How Accelerators Help

A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018

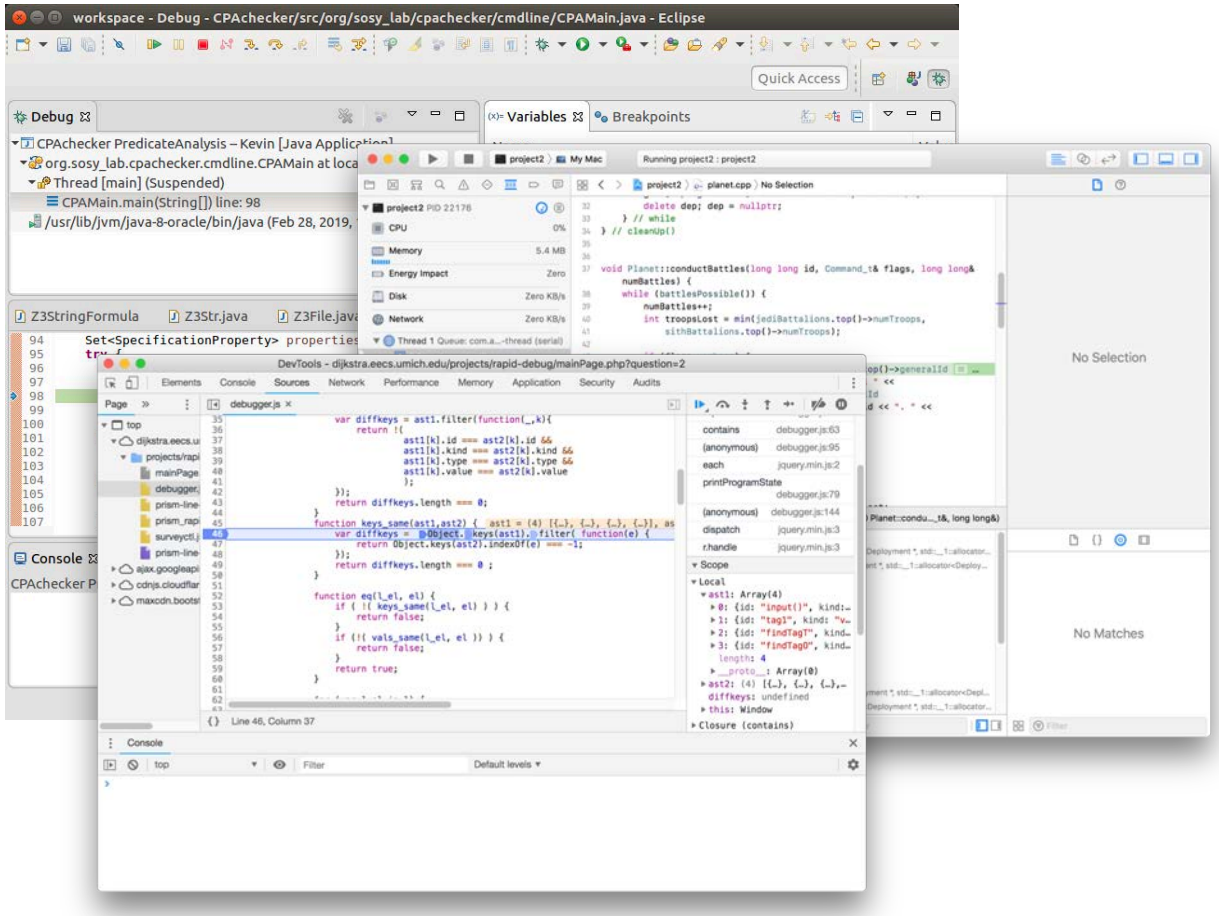
How Accelerators Help

A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018

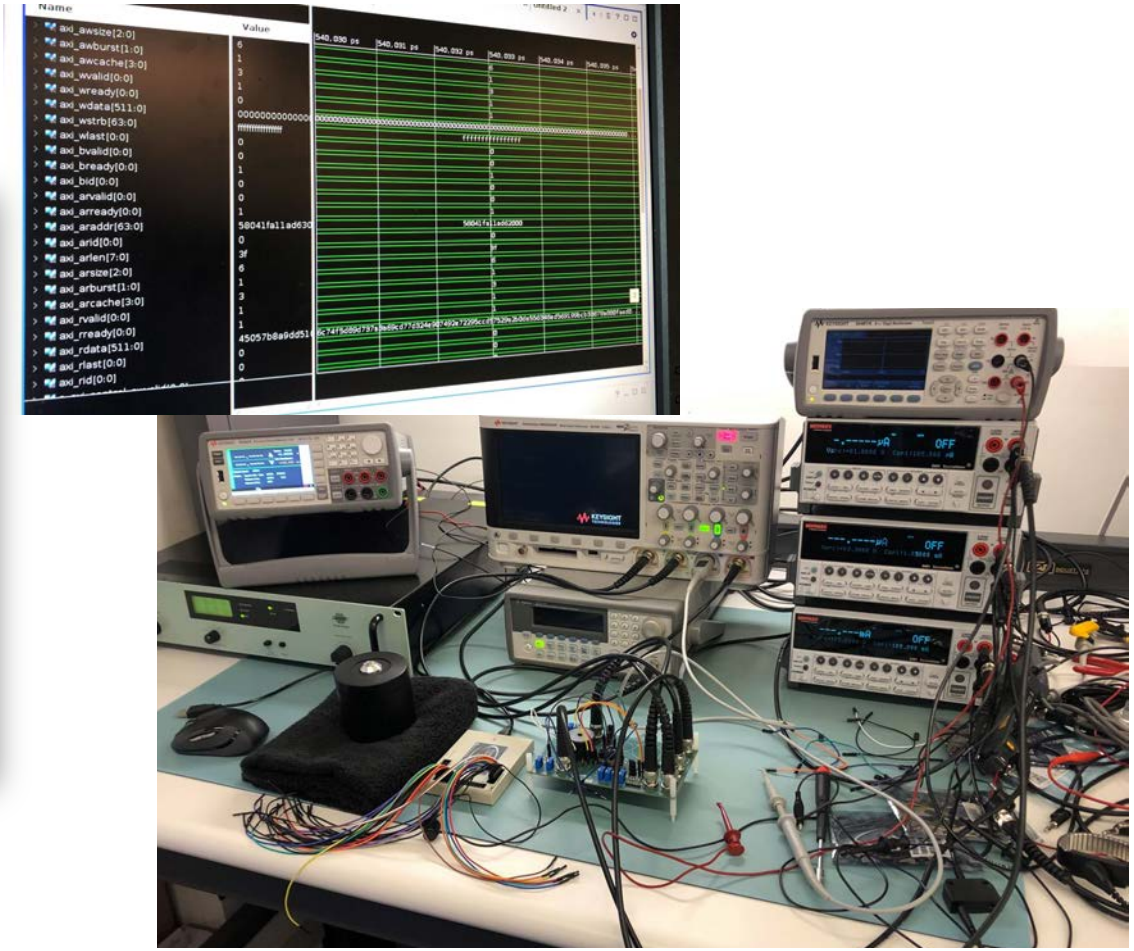
Maintenance and Debugging is Key

- Developers only spend a fraction of their time writing new code
  - Remaining time is spend understanding and modifying code
  - Using a debugger to understand code and localize faults is common
- Hardware developers spend even more time debugging and analyzing designs
- With the recent focus on HLS (e.g., OpenCL, SDAccel, etc.), we want debuggers that support these languages
  - Simulation/Emulation on CPU is likely too slow—that's why we're using an accelerator in the first place!

A Tale of Two Debugging Worlds



What Software Developers Expect



What Hardware Developers Use

GOAL

Support interactive debugging of high-level languages on hardware accelerators

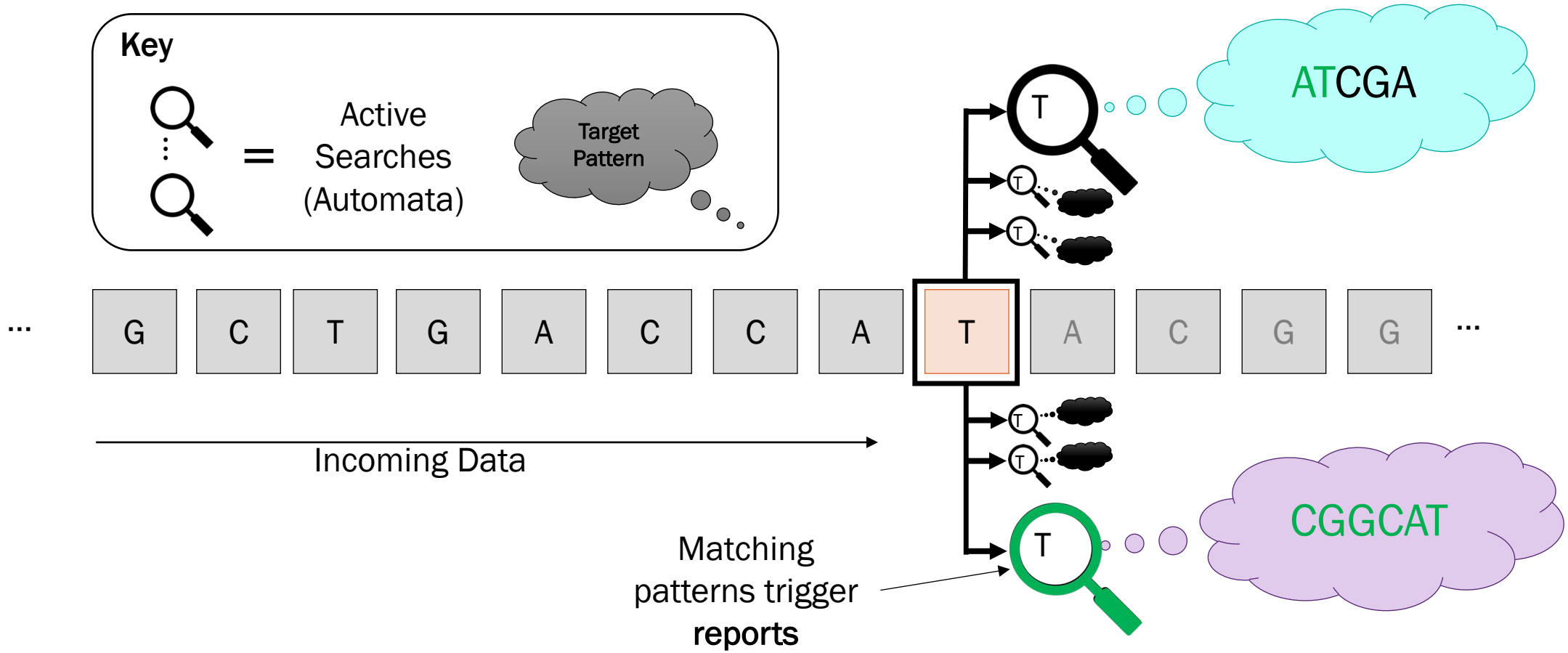
OVERVIEW

- RAPID Programming Model
- Debugging System
  - Technical Approach
- Experimental Evaluation
- Open Challenges

RAPID at a Glance

- Provides concise, maintainable, and efficient representations for pattern-identification algorithms
- Conventional, C- or Java-style language with domain-specific parallel control structures
- Compilation strategy supports execution on AP, FPGAs, CPUs, and GPUs
  - Finite automata (NFAs)/State machines used as an intermediate representation of computation

Domain-Specific Code Abstraction



RAPID (Automata) in the Big Data World

Detecting Intrusion Attempts in Network Packets

Learning Association Rules with an *a priori* approach

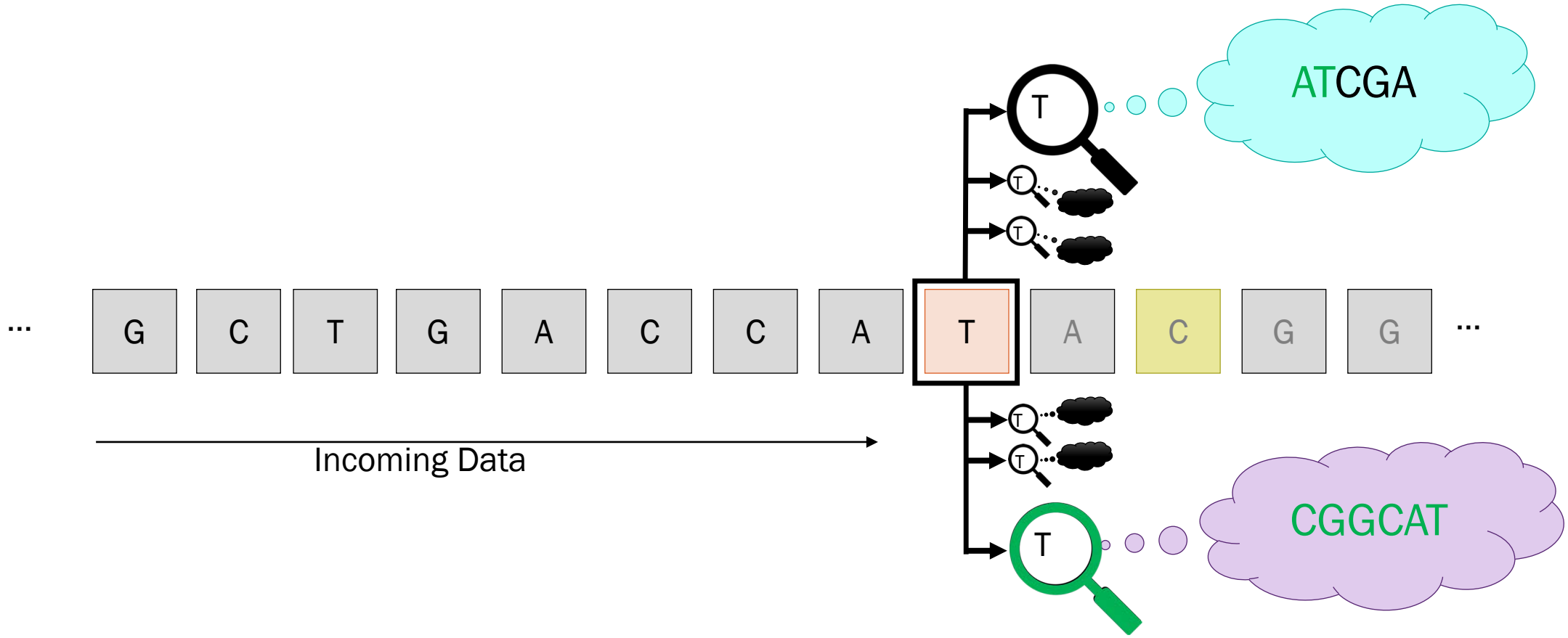
Detecting incorrect POS tags in NLP

Looking for Virus Signatures in Binary Data

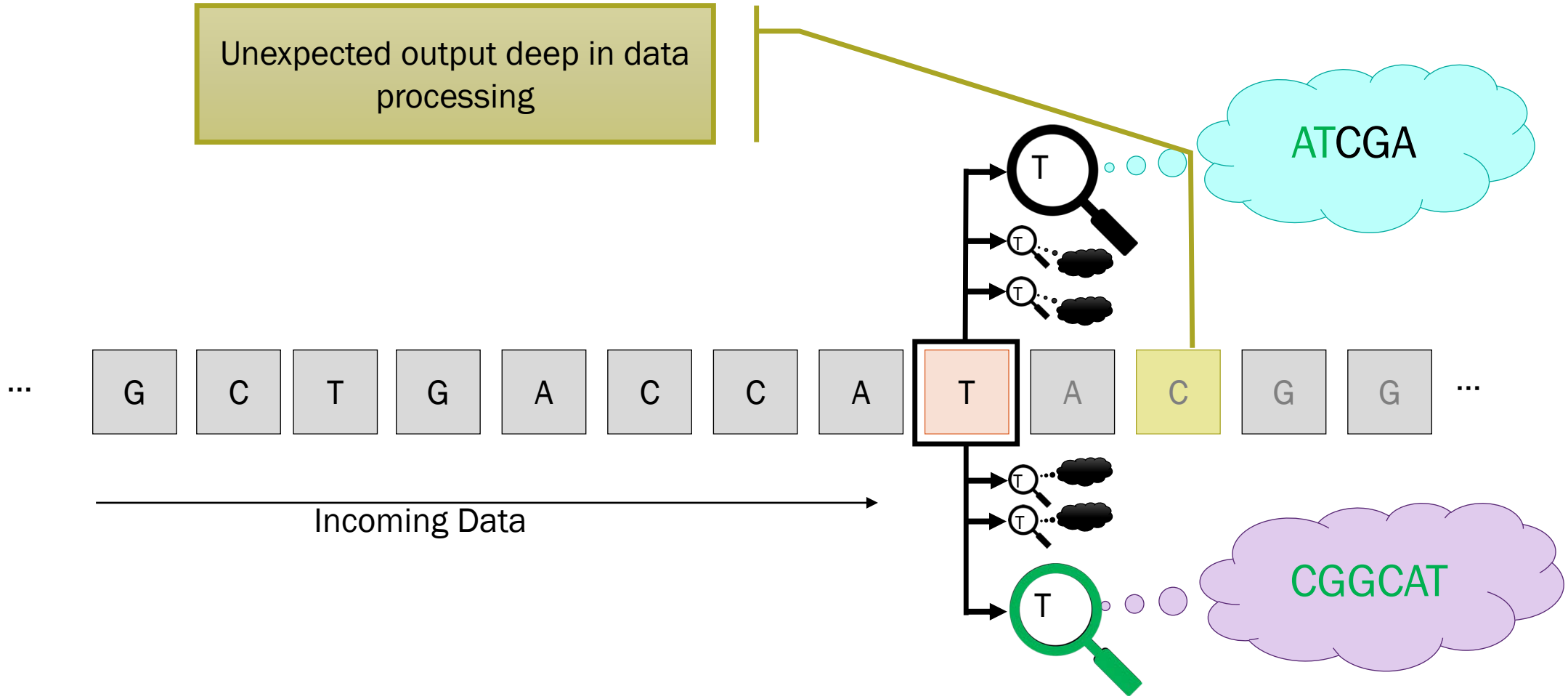
Detecting Higgs Events in Particle Collider Data

Aligning DNA Fragments to the Human Genome

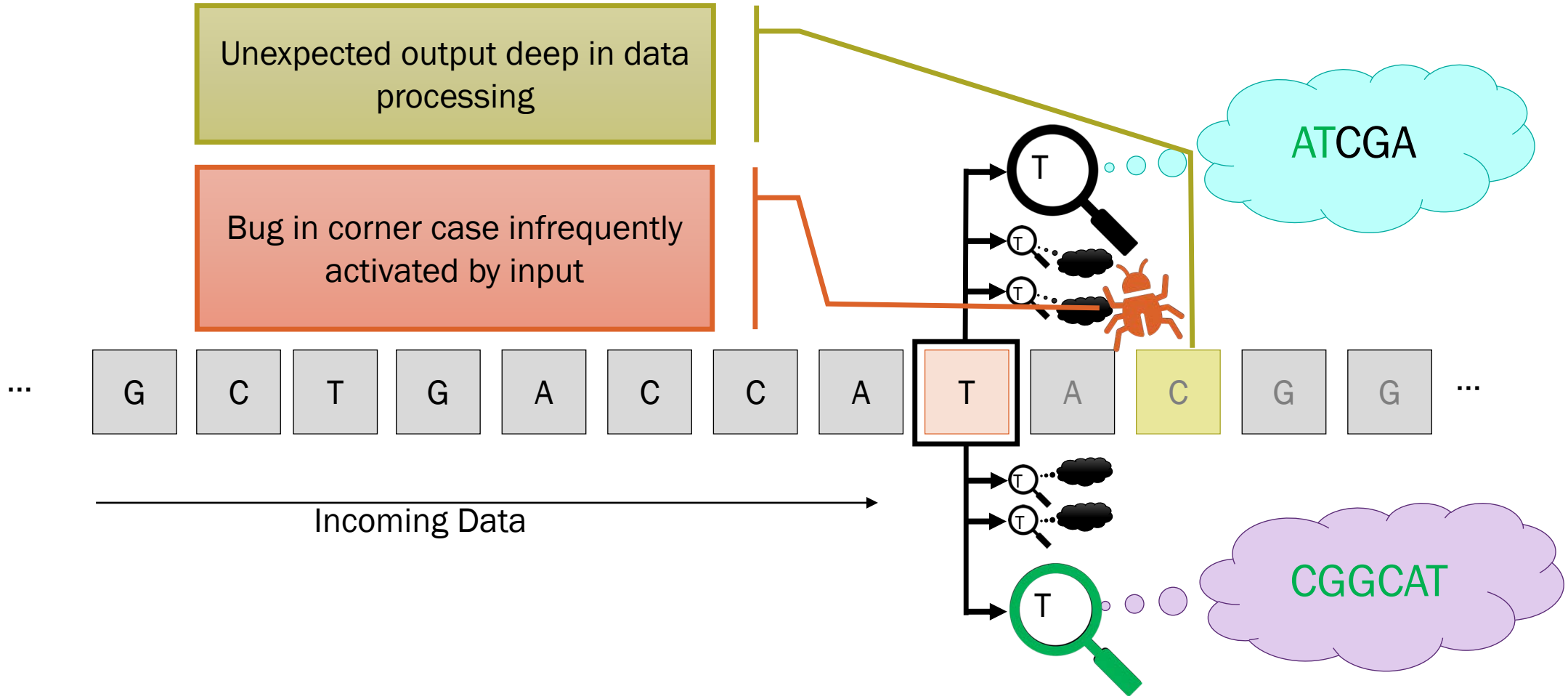
Houston, we have a problem!



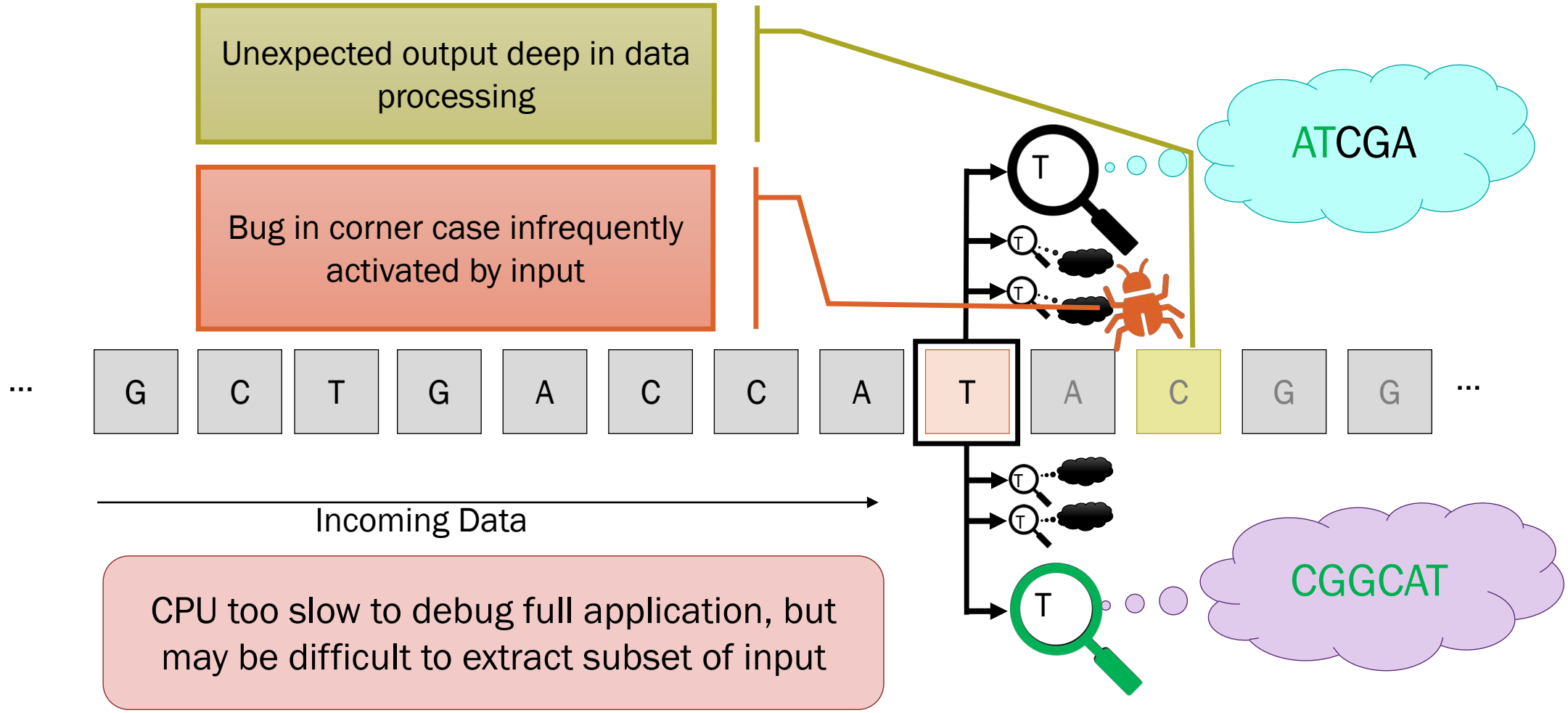
Houston, we have a problem!



Houston, we have a problem!



Houston, we have a problem!



“A [debugger is a] program designed to help detect, locate, and correct errors in another program. It allows the developer to **step through the execution** of the process and its threads, **monitoring memory, variables, and other elements** of process and thread context.”

MSDN Windows Dev Center

(https://msdn.microsoft.com/en-us/library/windows/desktop/ms679306(v=vs.85).aspx)

Multi-Step Process

1. First, we must halt execution and extract current program state from the processor

   Insight: leverage existing hardware/signal monitoring

2. Then, we lift the extracted state to the semantics of the source language

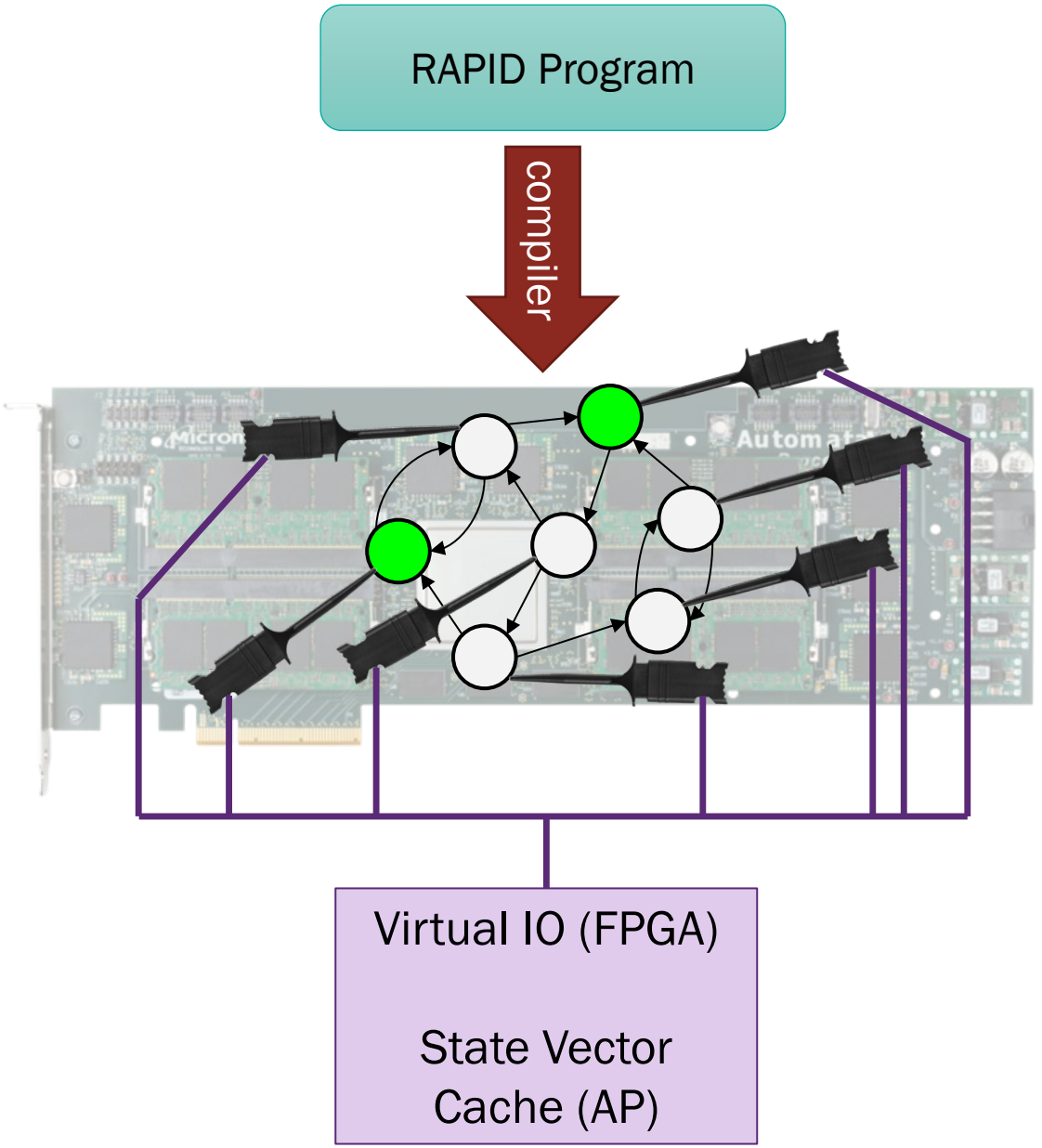
   Insight: generate mapping from expressions/statements to hardware elements during compilation

Where do we stop?

- **Breakpoints** annotate expressions/statements to specify locations to pause execution for inspection
  - Traditional notion relies on instructions stream
  - Mechanism does not apply directly to architectures with no instructions (e.g., FPGAs, AP)
- **Key Insight:** Automata computation driven by input
  - Set breakpoints on input data, not instructions
  - Supports use case of stopping computation at abnormal behavior
  - Can also provide abstraction of traditional breakpoints

Capturing State

- Process input data up to breakpoint
- State of automata is **compact**
  - O(n) in the number of states of the NFA
  - **State vector** captures relevant execution information
- Repurpose existing hardware to capture
  - AP: State vector cache already stores active states
  - FPGA: Integrated Logic Analyzers (ILAs) and Virtual I/O pins (VIOs) allow for probing of activation bits
- Cache state vectors to decrease latency

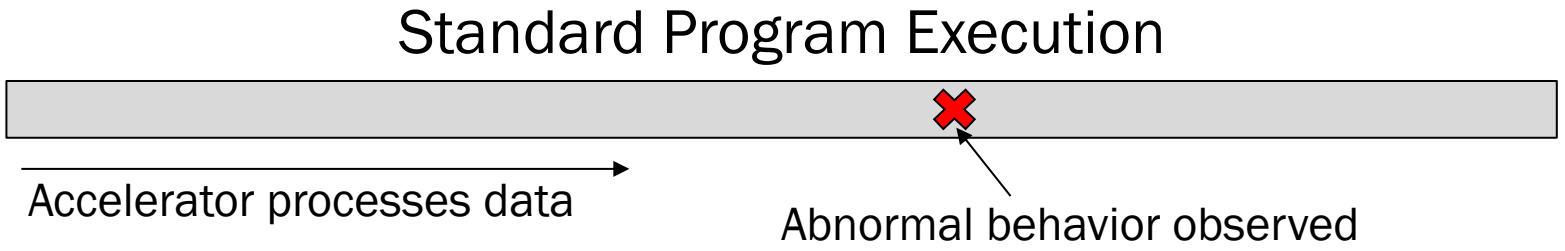


Lifting Hardware State to Source-Level

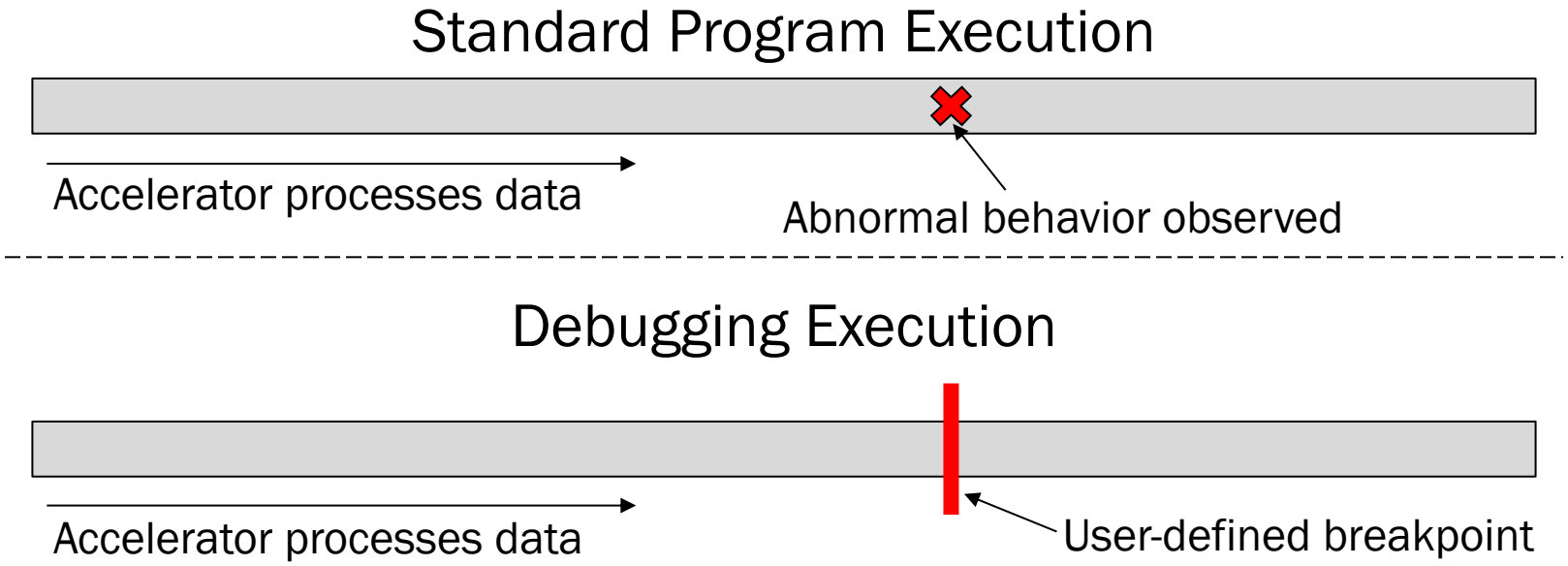
- Modify the RAPID compiler to generate debugging tables
  - Approach for the RAPID compiler is similar to traditional compilation
  - For every line, which NFA states does it map to?
  - For every line, what variables are in scope and what are their values (or which hardware resources hold their values)?
- At breakpoint, apply mappings in reverse
- Now have:
  - Expressions currently executing
  - Values of in-scope variables

Putting it all together

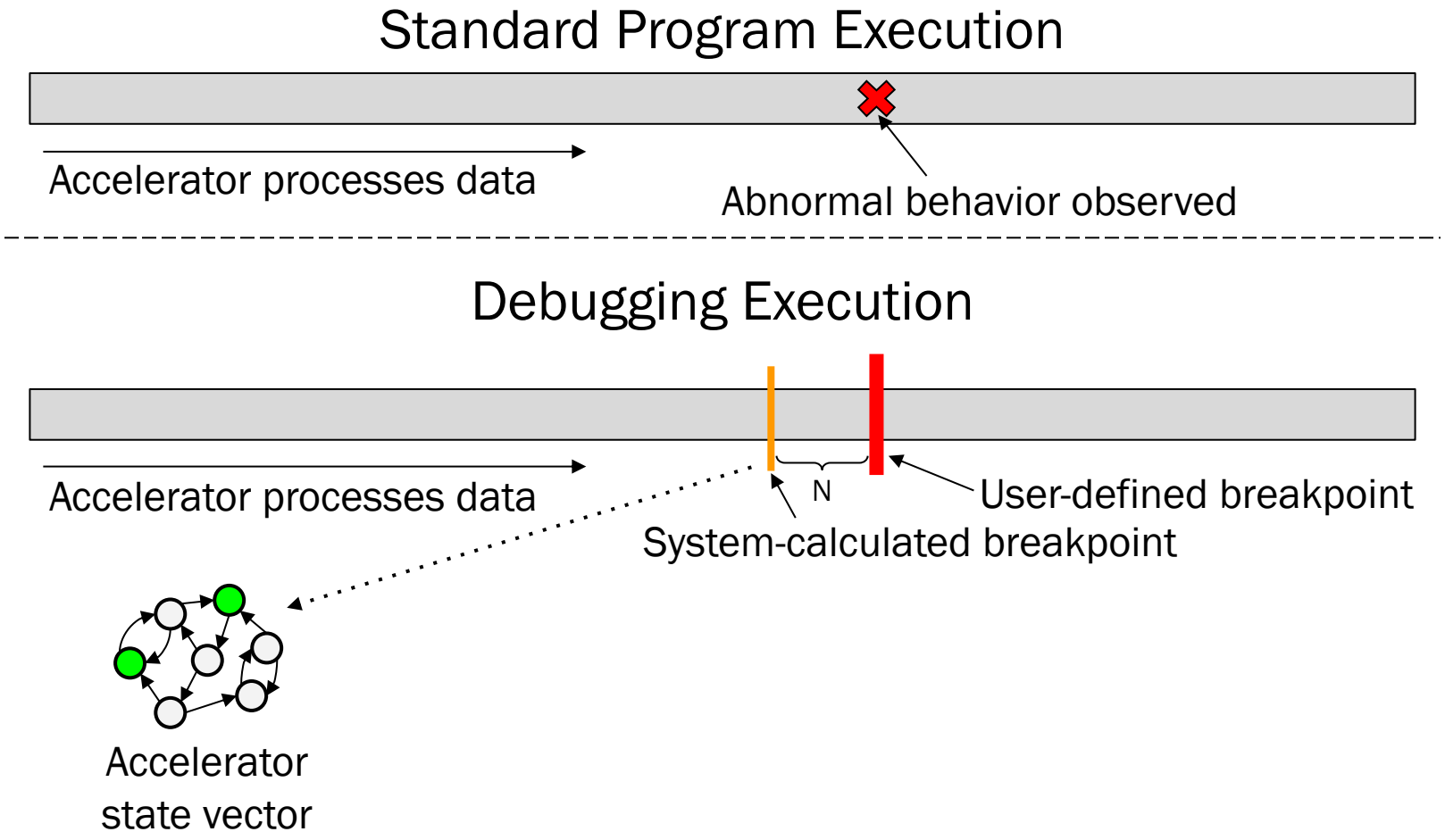
Putting it all together



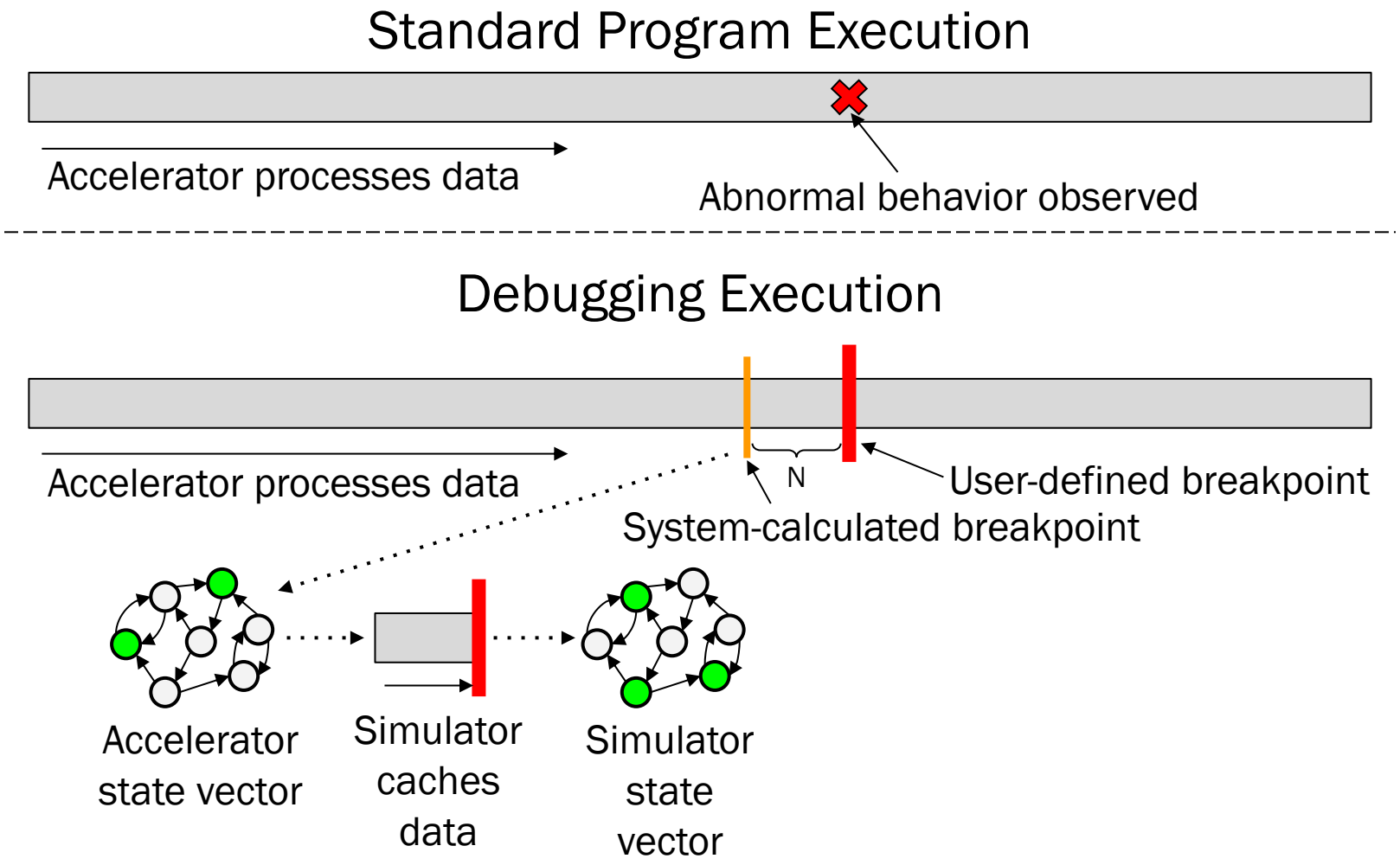
Putting it all together



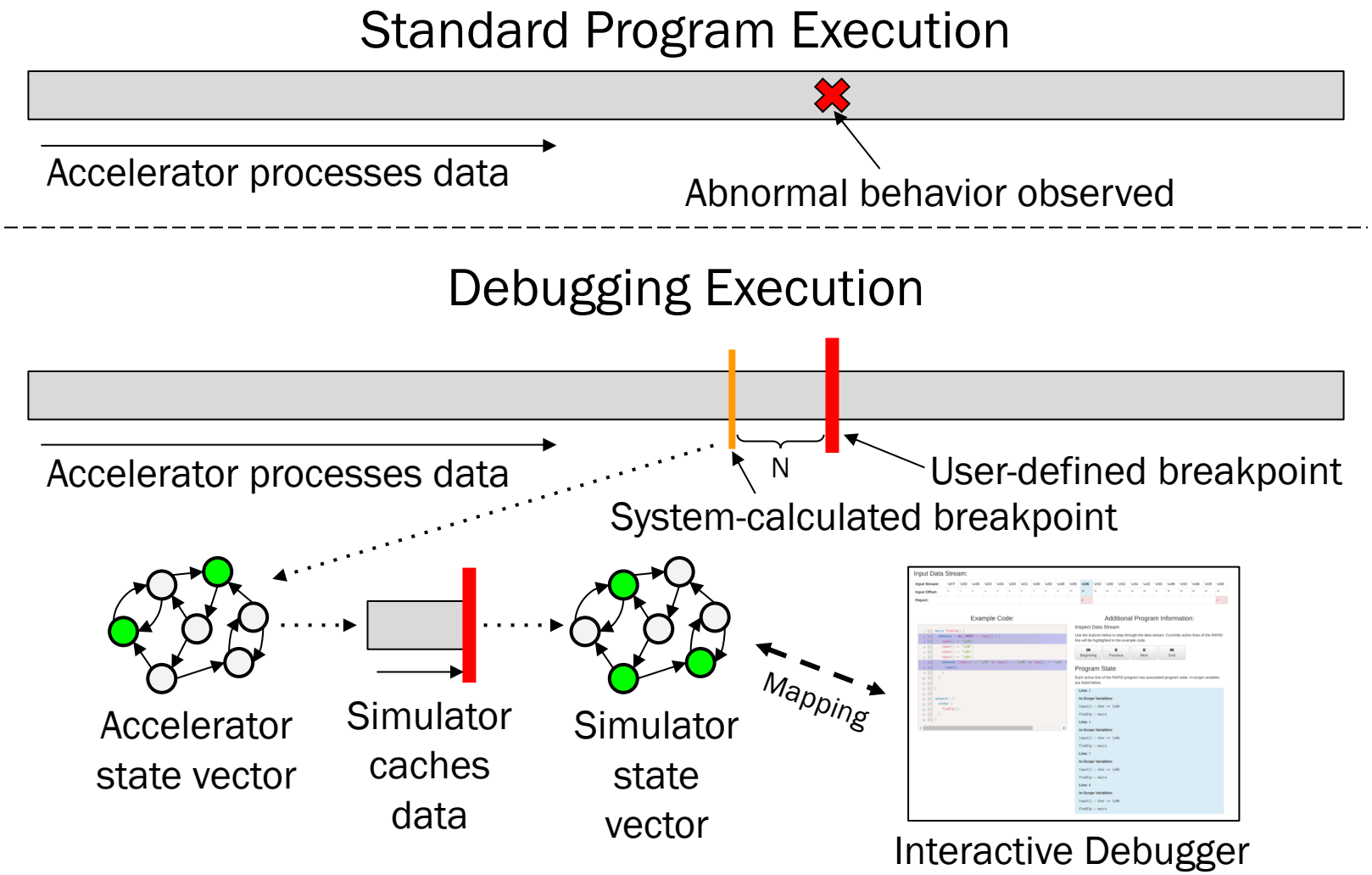
Putting it all together



Putting it all together



Putting it all together



Evaluation: Hardware and Software

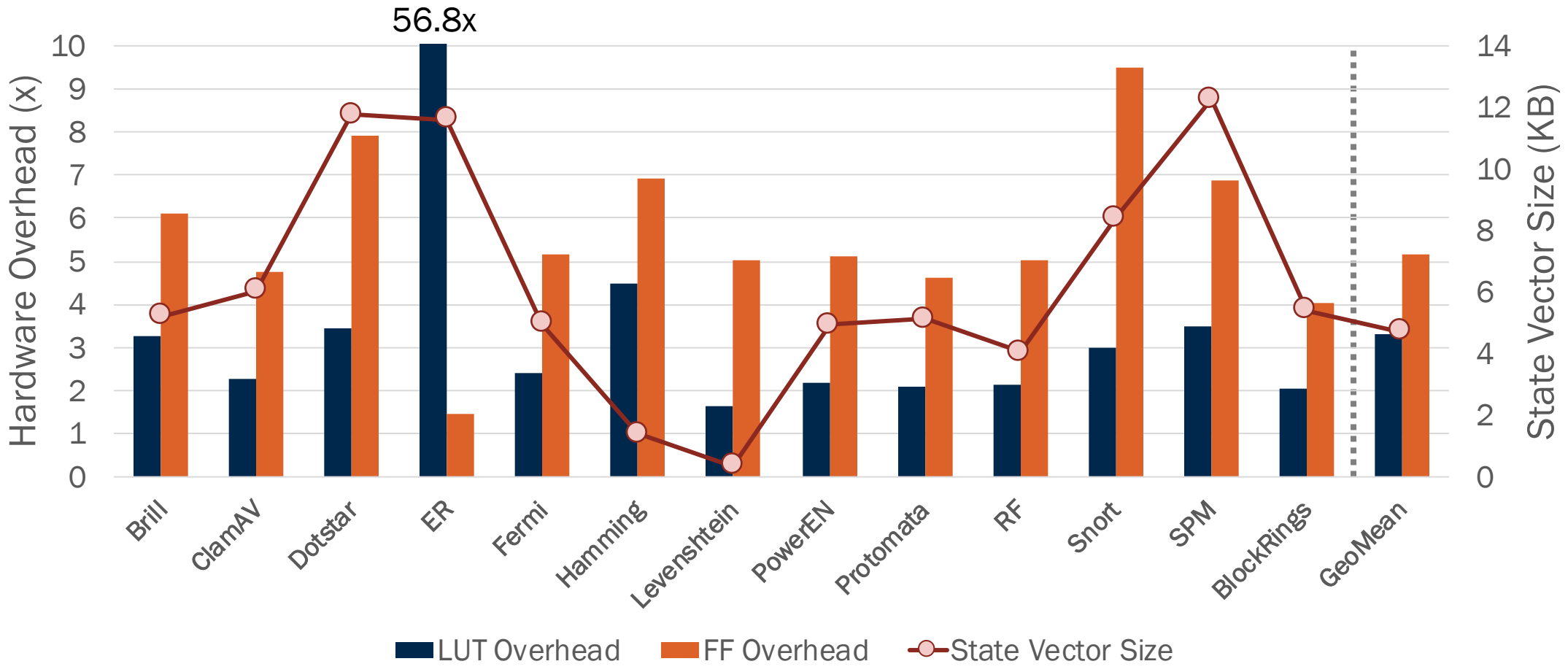
- Measure **performance** and **scalability** of FPGA-based automata debugging
  - ANMLZoo benchmark Suite (14 real-world automata applications)
  - Focus on **overheads** (e.g., hardware resources and clock frequency) introduced by adding debugging hardware
- Measure **ease of use** with a human study
  - Participants given **fault localization** tasks in mock development environment
  - Ten RAPID programs with indicative bugs
  - Focus on **accuracy**, **time**, and **developer experience**

Experimental Methodology

- **Evaluation System**
  - **CPU:** 3.70 GHz 4-core Intel Core i7-4820K (32 GB RAM)
  - **FPGA:** Alphadata board rev 1.0 (Xilinx Kintex-Ultrascale xcku060-ffva1156-2-e)
- **Synthesis:**
  - REAPR modified to produce Verilog with VIO for debugging
  - Xilinx Vivado 2017.2
- **Debugging Simulation:**
  - RAPID compiler modified to generate debugging information
  - VASim modified to produce and capture state vector information

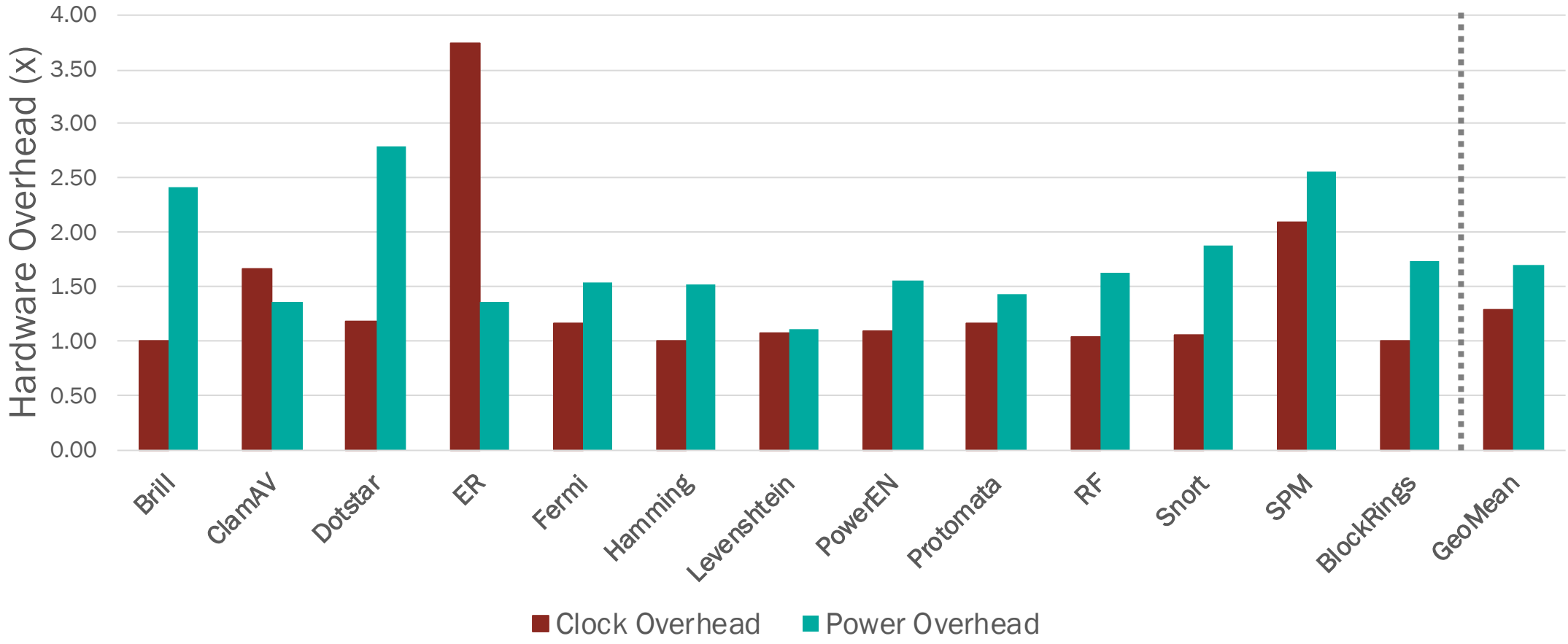
FPGA Hardware Overheads

Overheads are manageable, but vary widely



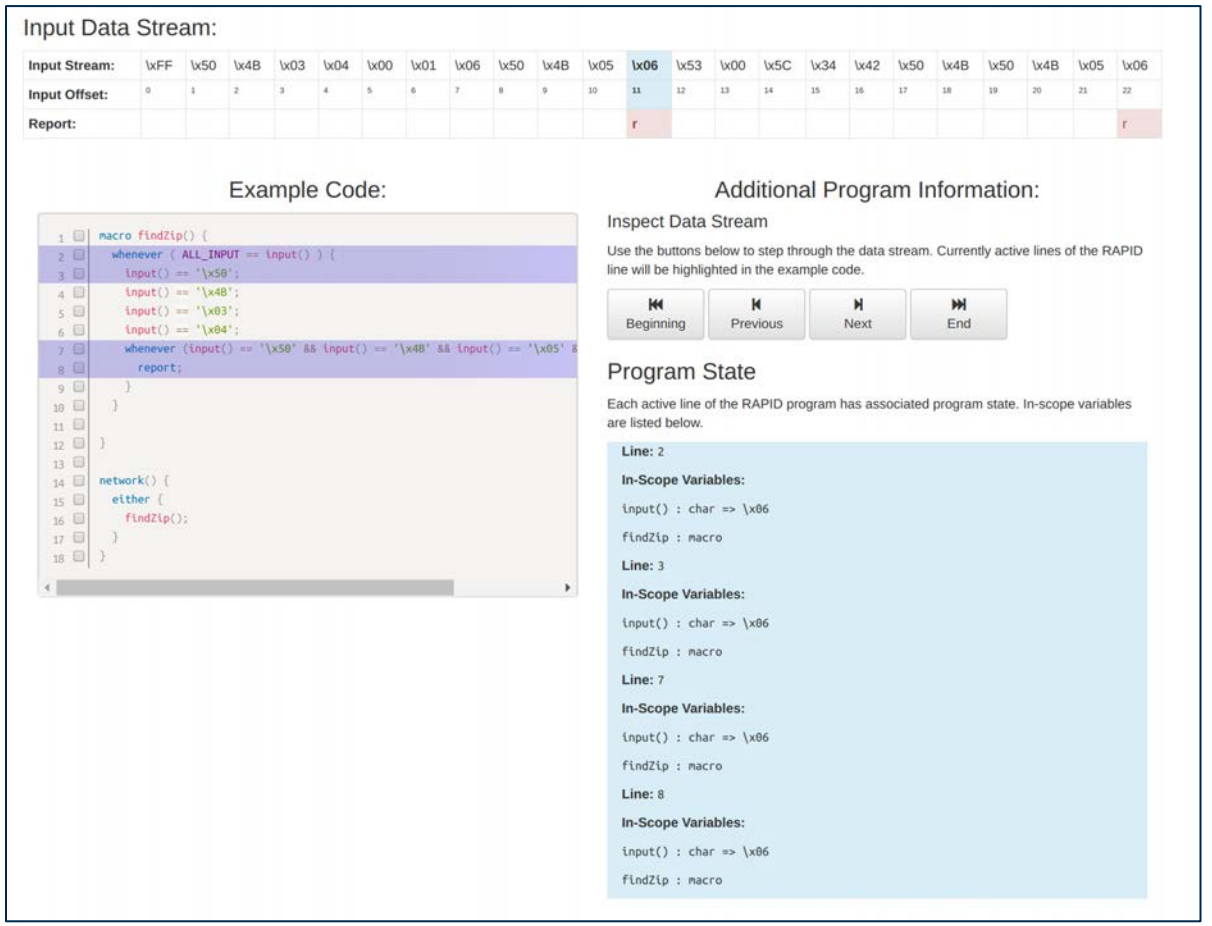
FPGA Performance Overheads

82% clock frequency on average — still fast on debug builds



Human Study Results

- N=61 participants (predominantly UVA students)
- Asked to identify location of bug in source code and describe
- Statistically significant improvement in fault localization accuracy by 22% ($p = 0.013$)
- Helps novices and experts alike



Open Challenges

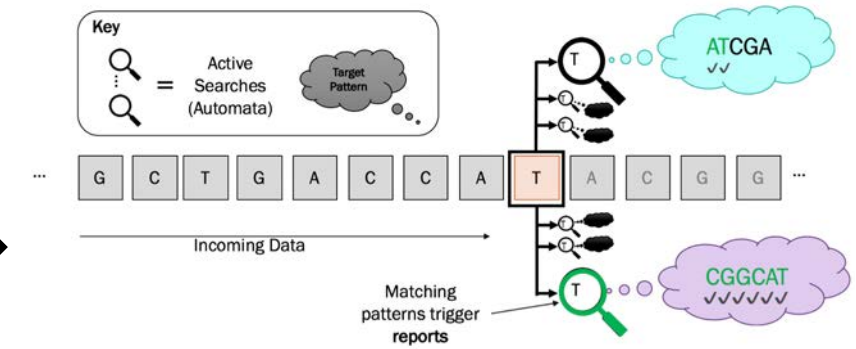
Hardware Design

- Reduce overhead of program state capture
- Decouple clocking for program execution
- Dynamic selection of program state to monitor and capture

Software Tools

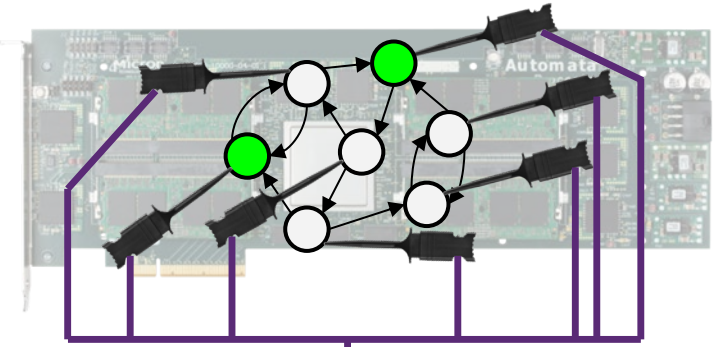
- Program analyses to limit captured state size
- Further study of acceptable overheads and latencies
- Additional support for existing (e.g., watchpoints) and new debugging abstractions

Conclusion



RAPID Programming Model as Stepping Stone

Accelerator-Assisted Debugging



Debugger

22% increase in fault localization accuracy

Low overheads for server-class FPGAs

82% original clock freq.

Bonus Slides

Example RAPID Program

If all symbols in item set match, increment counter

Spawn parallel computation for each item set

Sliding window search calls *frequent* on every input

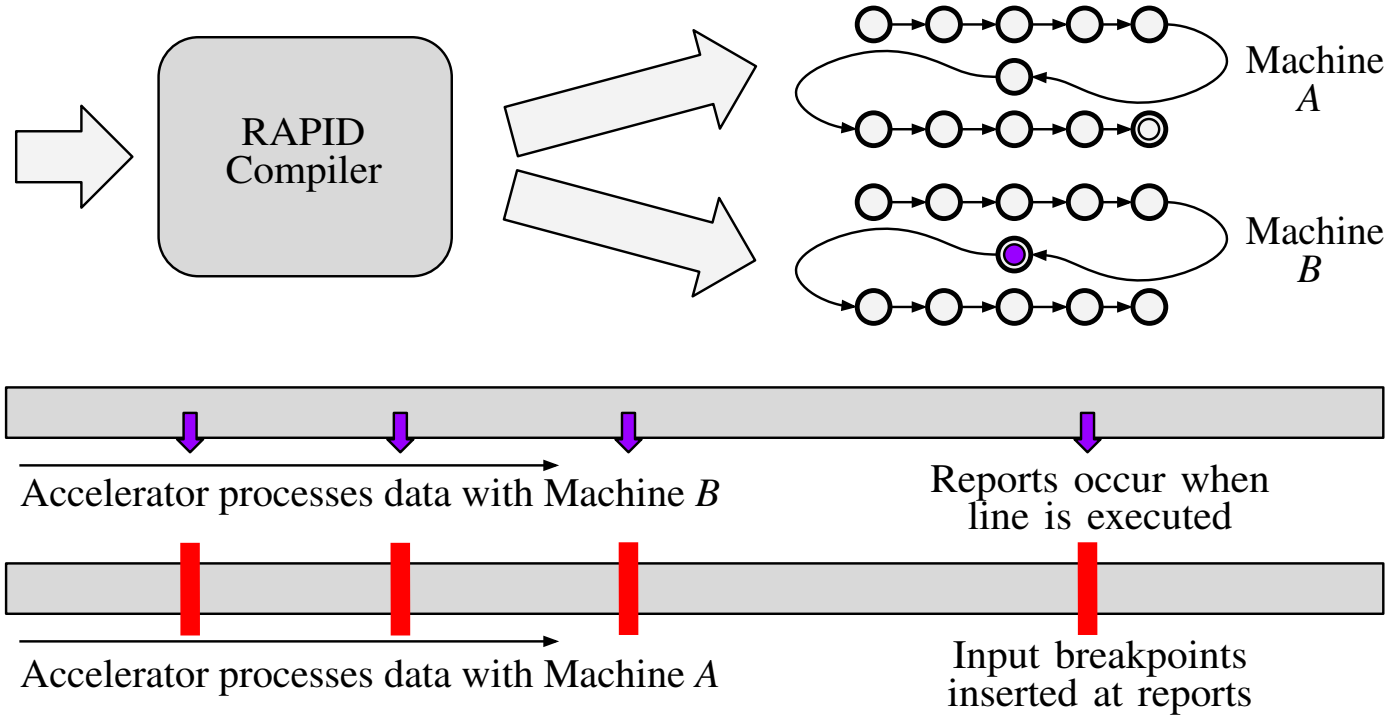
Trigger *report* if threshold reached

```
macro frequent (String set, Counter cnt) {
        foreach(char c : set) {
            while(input() != c);
        }
        cnt.count();
}

network (String[] set) {
    some(String s : set) {
        Counter cnt;
        whenever(START_OF_INPUT == input())
            frequent(s,cnt);
        if (cnt > 128)
            report;
    }
}
```

Traditional Breakpoints

RAPID Program

```
macro helloWorld() {
  whenever(  ALL_INPUT == input() ) {
    foreach(char c : "Hello") {
      c == input();
    }
    input() == ' ';
    foreach(char c : "world") {
      c == input();
    }
    report;
  }
}

network() {
  helloWorld();
}
```



Summary Statistics

Subset	Average Time (min.)	Average Accuracy	Participants
All	8.17	50.3%	61
Intermediate Undergraduate Students	7.30	49.2%	37
Advanced Undergraduate Students	10.14	50.0%	21
Graduate Students and Professional Developers	5.07	66.7%	3