GPGPU Pipeline Visualization for RISC-V SIMT Architecture

CARRV 2024

Yu-Yu Hsiao†, Liang-Chou Chen, Chung-Ho Chen

National Cheng Kung University, Taiwan

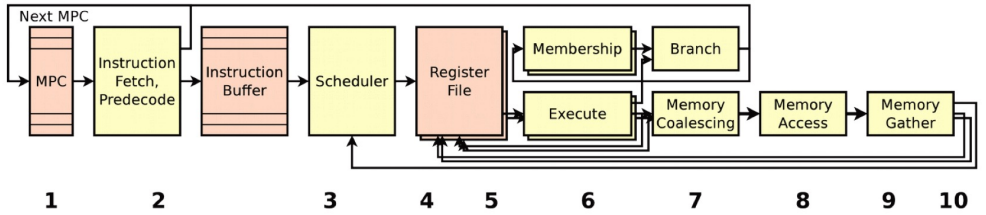
†n28111534@gs.ncku.edu.tw

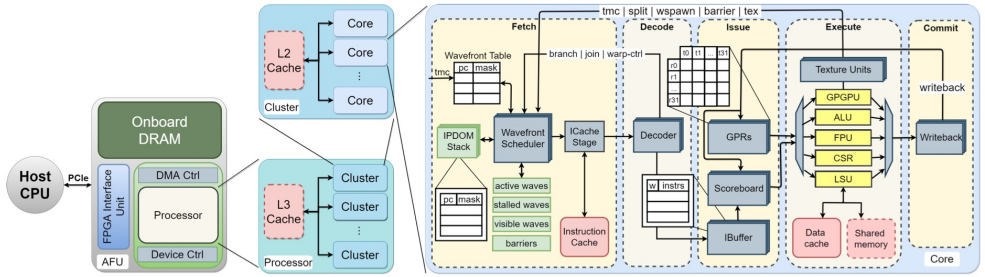
Outline

- Motivation
  - RISC-V-based GPGPUs
  - Pipeline Visualization for CPUs
- Pipeline Visualization for RISC-V SIMT Architecture
  - Enabling SIMT Execution Model on RISC-V
  - Case Studies

RISC-V-based GPGPUs

- Simty (CARRV 2017)
  - Dynamic inter-thread vectorization
  - FPGA prototype
  - Written in synthesizable VHDL
- Vortex (CARRV 2019, MICRO 2021)
  - LLVM-based software stack for OpenCL
  - Texture and graphic support
  - FPGA prototype
  - Written in synthesizable SystemVerilog
  - Incorporates a cycle-level simulator (C++)



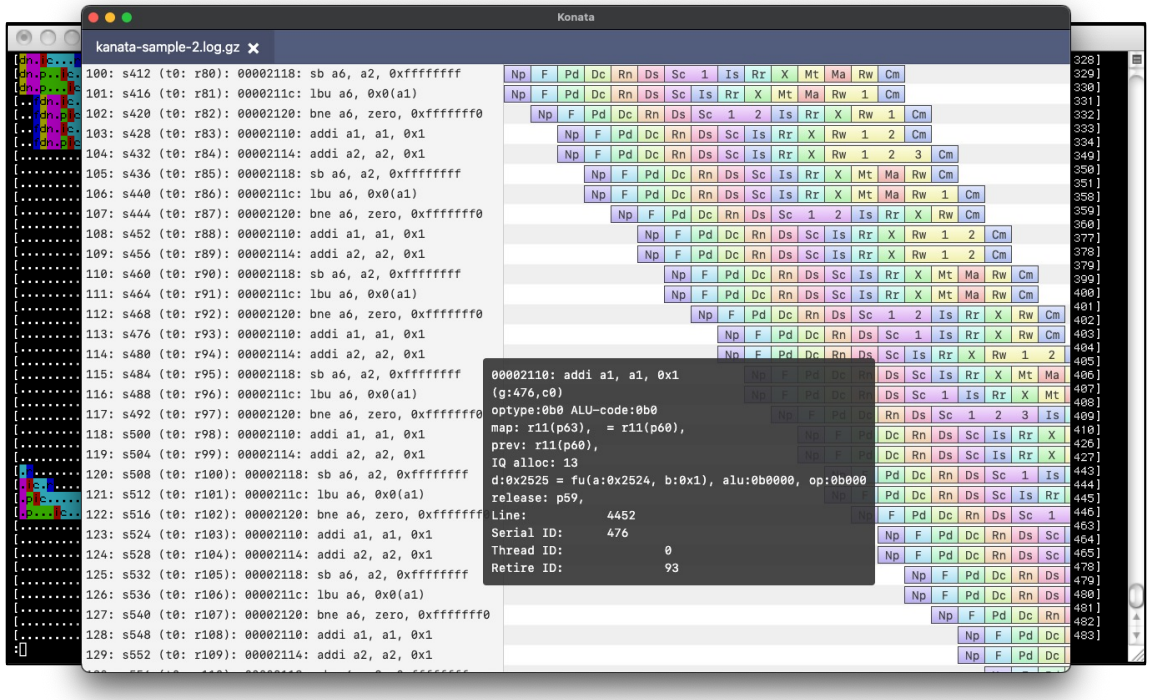


GPGPU architecture exploration for CUDA:

- GPGPU-Sim (ISPASS 2009)
- Accel-Sim (ISCA 2020)

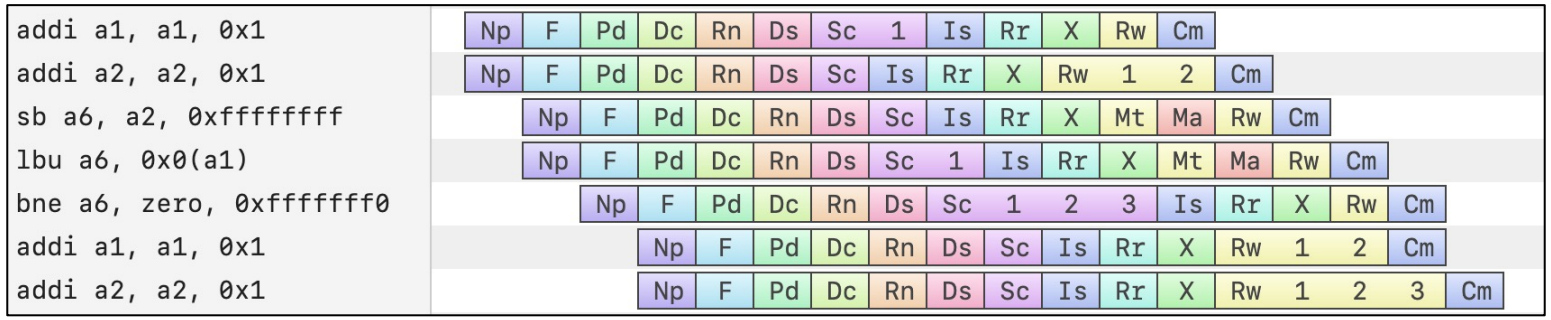
Pipeline Visualization for CPUs (1/2)

- Pipeline visualization has become a well-established technique in the CPU architecture domain
  - gem5 text-based pipeline view, AndesClarity, Konata



Pipeline Visualization for CPUs (2/2)

- **Example**: Dhrystone running on a RISC-V out-of-order superscalar processor, RSD[1]

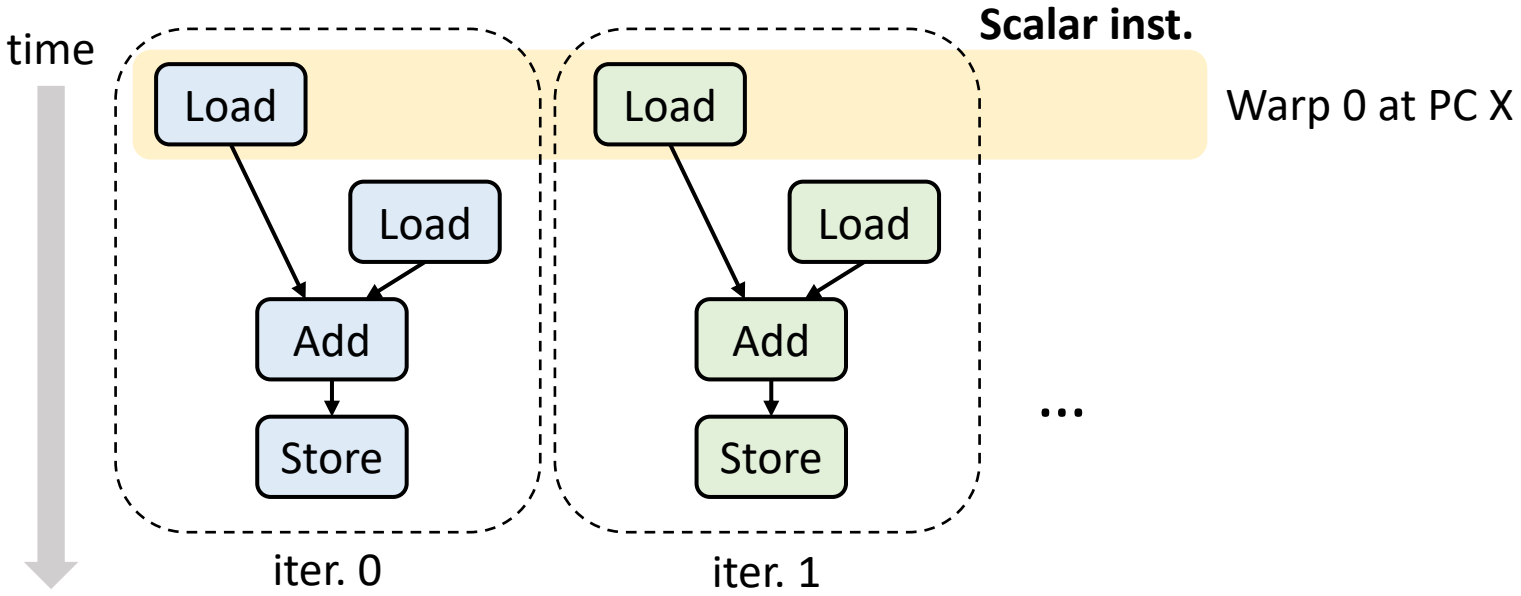


  - **Superscalar pipeline**: Two instructions are executed simultaneously at the same stage
  - **Out-of-order execution**: Instructions are executed out of program order
  - **In-order fetch and retirement**: Despite being executed out-of-order, the instructions are fetched and committed in the program order

[1] https://github.com/rsd-devel/rsd

SPMD on SIMT Machine

```
// tid ∈ [0,N)
C[tid] = A[tid] + B[tid];
```



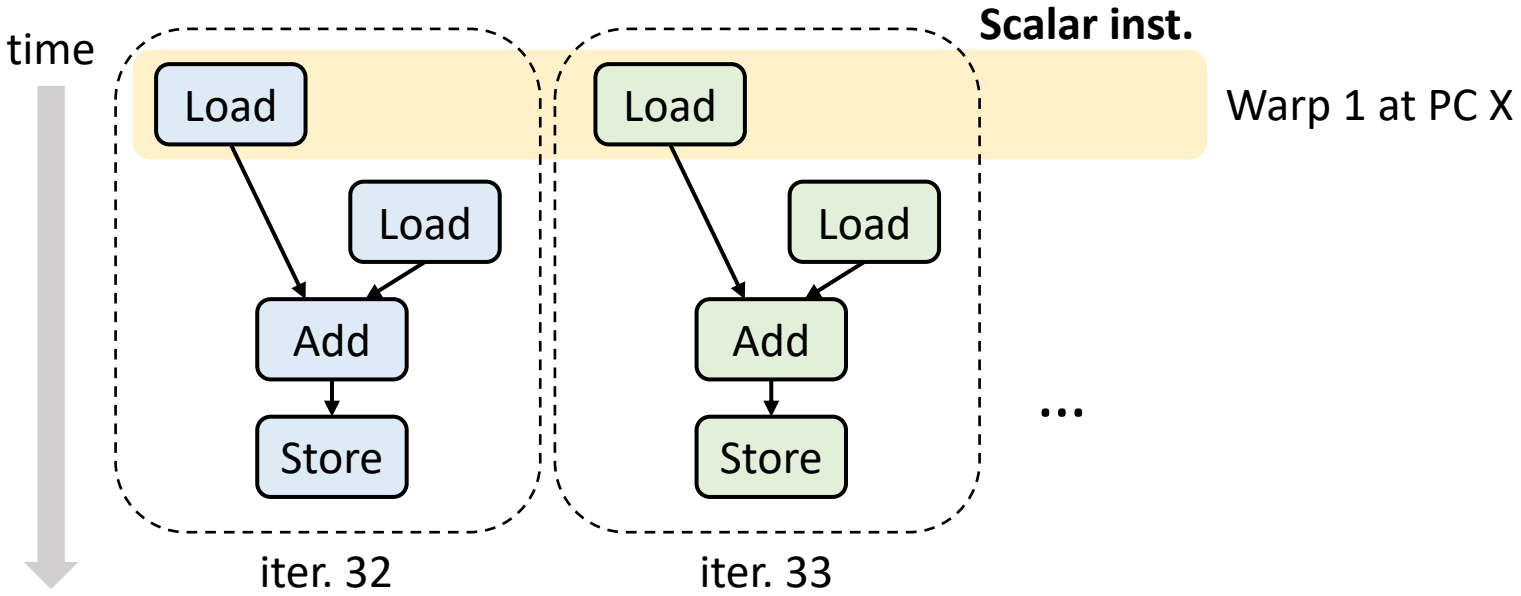
Assume:

- thread/warp = 32
- iteration/thread = 1

- **Warp**: A set of threads that executes the same instruction (i.e., at the same PC)

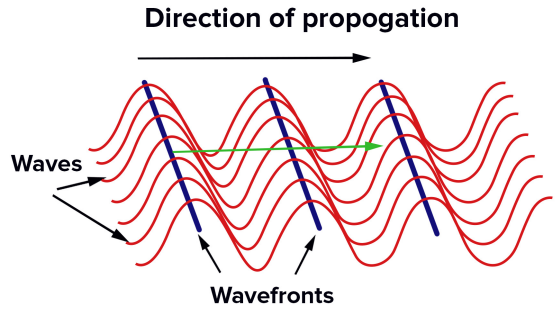
SPMD on SIMT Machine

```
// tid ∈ [0,N)
C[tid] = A[tid] + B[tid];
```



Assume:

- thread/warp = 32
- iteration/thread = 1

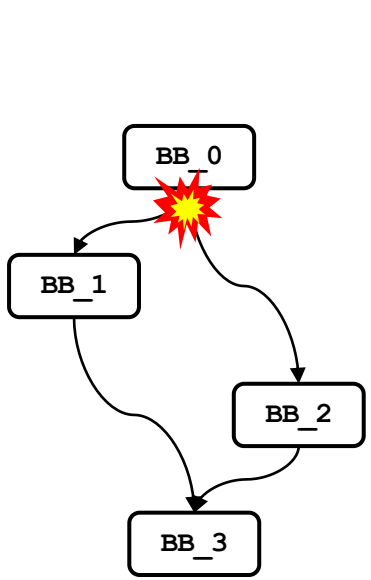


- **Warp**: A set of threads that executes the same instruction (i.e., at the same PC)
- All warps in a SIMT machine are scheduled and executed seamlessly in a fine-grained multithreaded pipeline

The Branch Divergence Issue

- Each thread can have conditional control flow instructions
- Threads in the same warp can execute different control flow paths

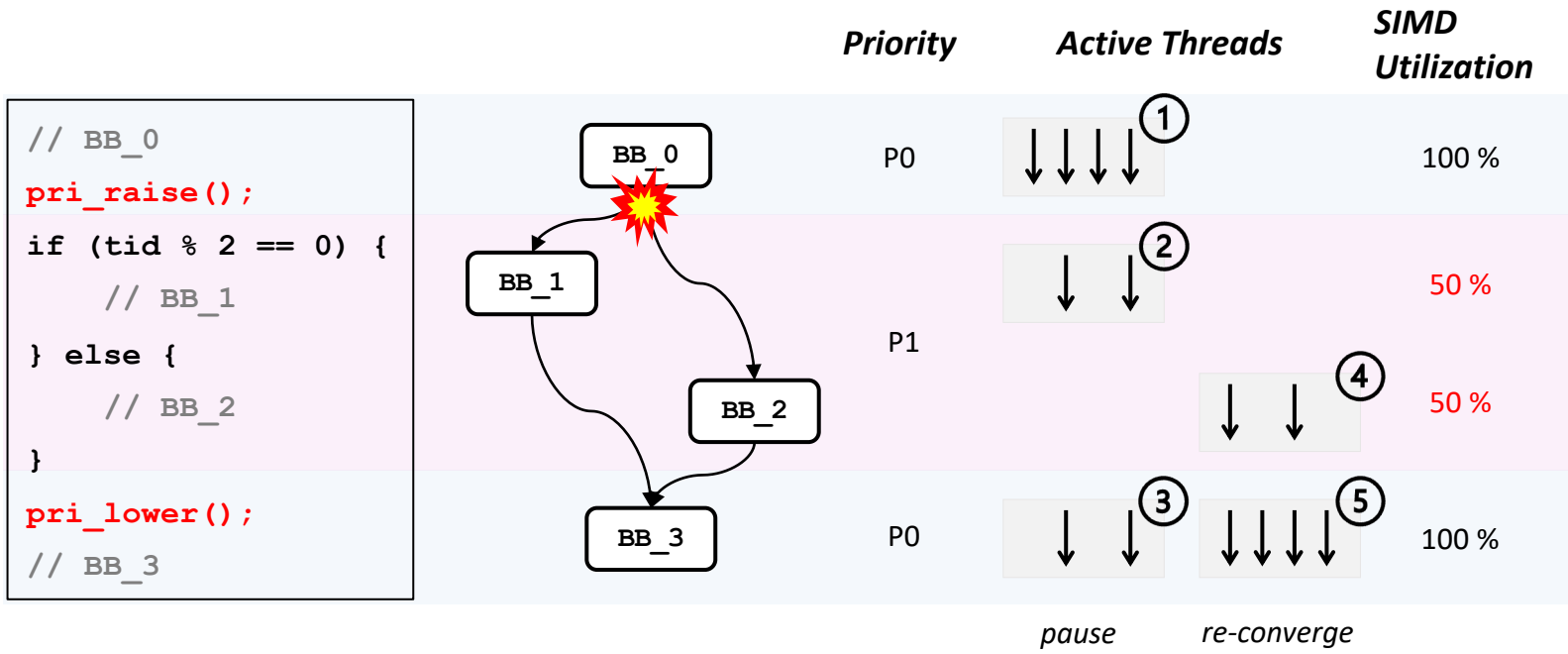
```
// BB_0

if (tid % 2 == 0) {
    // BB_1
} else {
    // BB_2
}

// BB_3
```



Priority *Active Threads* *SIMD Utilization*

ICS-First Re-convergence Scheme[1]

- **Idea**: Give inner conditional statements a higher priority
  - Faster threads will pause and wait for reconvergence at the outer statement due to priority degradation

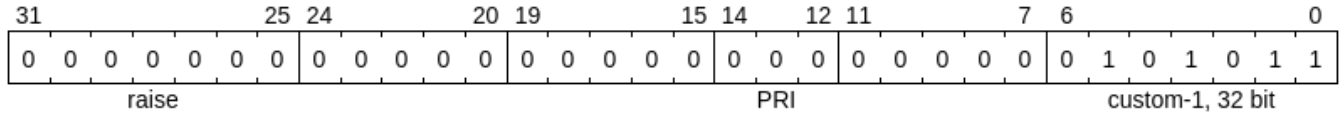


[1]Kuan-Chung Chen and Chung-Ho Chen. 2018. Enabling SIMT Execution Model on Homogeneous Multi-Core System. ACM Transactions on Architecture and Code Optimization, 15, 1, (Mar. 22, 2018), 6:1–6:26. doi: 10.1145/3177960.

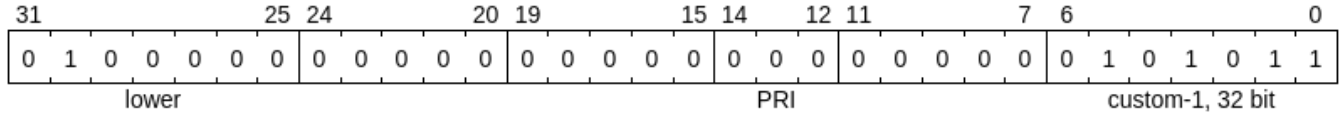
Proposed ISA Extension for our GPGPU

- We employ ICS-First algorithm to handle branch divergence
- 3 additional instructions using custom-1 opcode
- Patched LLVM compiler for inserting priority adjustment instructions automatically

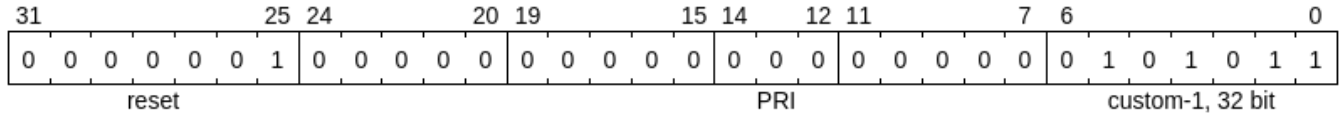
fsa.pri.raise



fsa.pri.lower

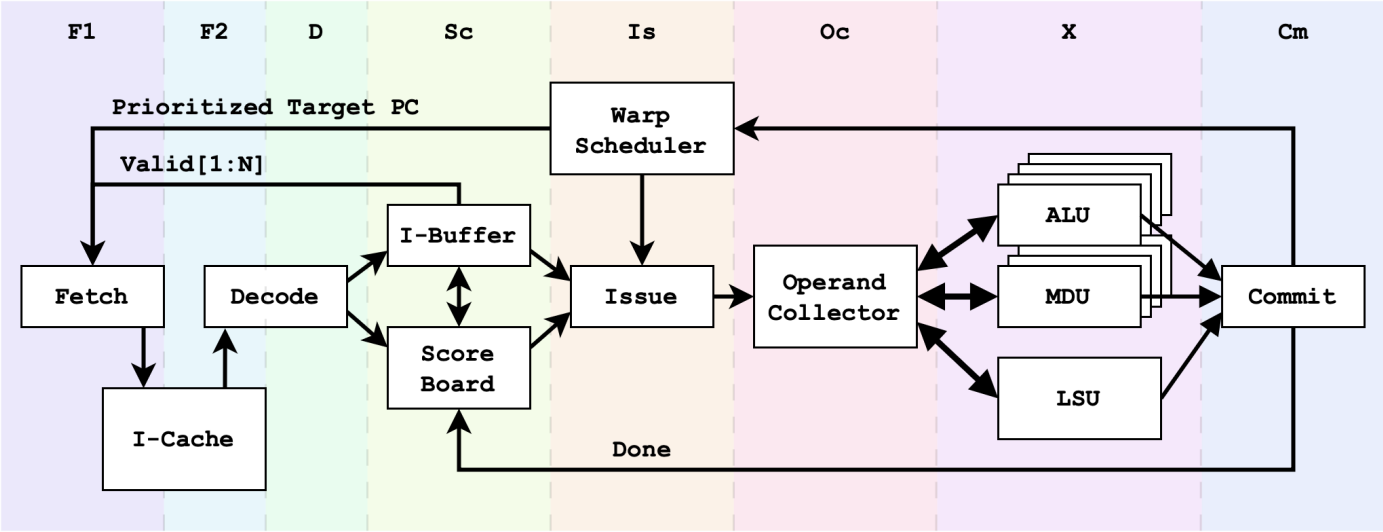


fsa.pri.reset



RISC-V SIMT Pipeline Architecture

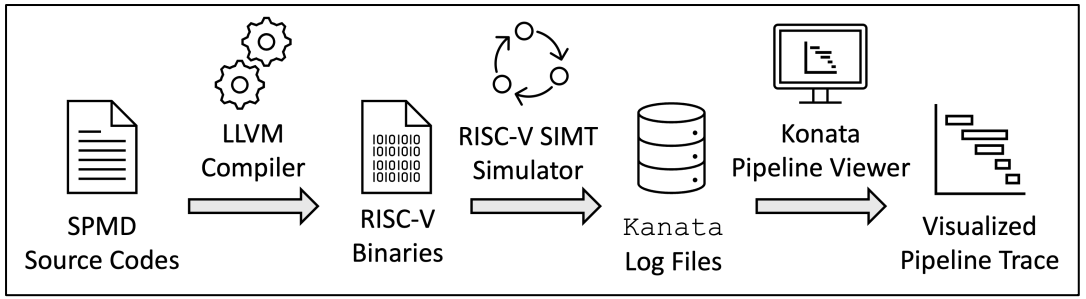
- Microarchitecture of modeled SM
  - Based on the architecture of **GPGPU-Sim** and **CASLab-GPU**[1]
  - 8-stage in-order issue OoO commit pipeline, priority-based PC arbitration
  - Dedicated fetch scheduler



[1]Y. -X. Su, J. -H. Jheng, D. -J. Chen and C. -H. Chen, "Development of an Open ISA GPGPU for Edge Device Machine Learning Applications," 2019 Eleventh International Conference on Ubiquitous and Future Networks (ICUFN), Zagreb, Croatia, 2019, pp. 214-217, doi: 10.1109/ICUFN.2019.8806196.

Pipeline Visualization Workflow

- We implement a cycle-based RISC-V SIMT simulator
- Run execution-driven simulation on a compiled SPMD program
- Logs are dumped during the simulation
- Open and render the pipeline trace in Konata pipeline viewer



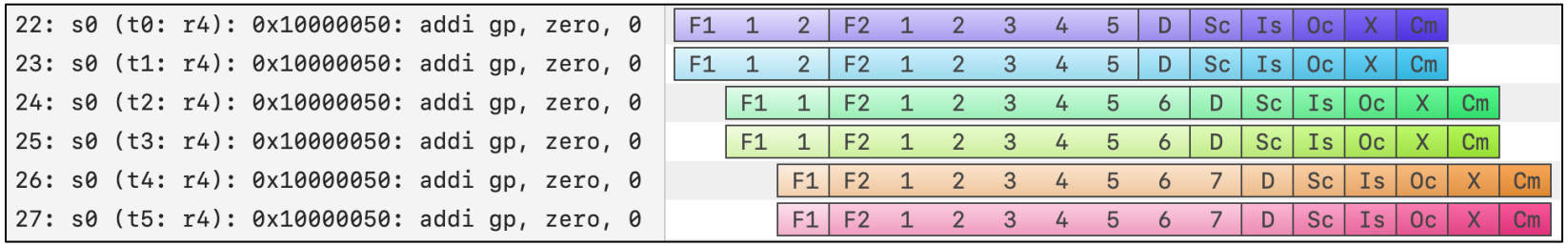
Log size:
w/o compression: 200 bytes per instruction
w/ compression (gz): 40 bytes per instruction

```
Kanata  0004
C=      -1
C       1
I       0       0       0
S       0       0       F1
I       1       0       0
S       1       0       F1
C       1
I       2       0       0
S       2       0       F1
I       3       0       0
S       3       0       F1
C       1
C       1
...
```

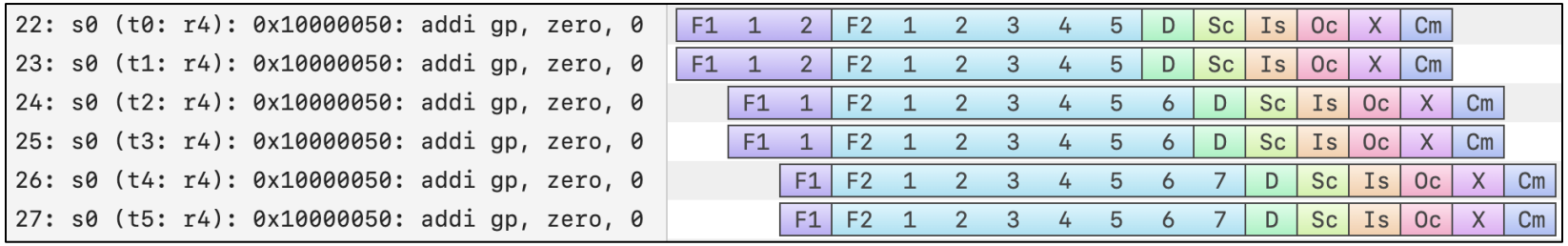
Kanata **Log Format**

Pipeline Visualization for SIMT Architecture (1/2)

- In SIMT, instructions in the pipeline may belong to different warps
  - Use different colors to identify the warp ID of the instruction

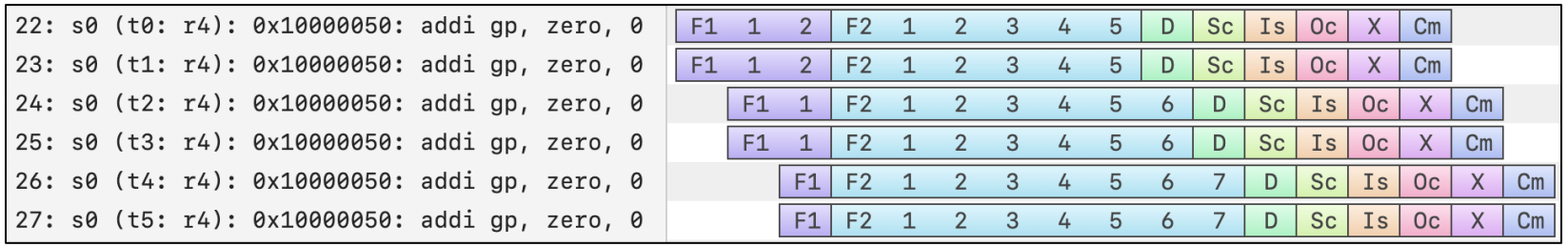


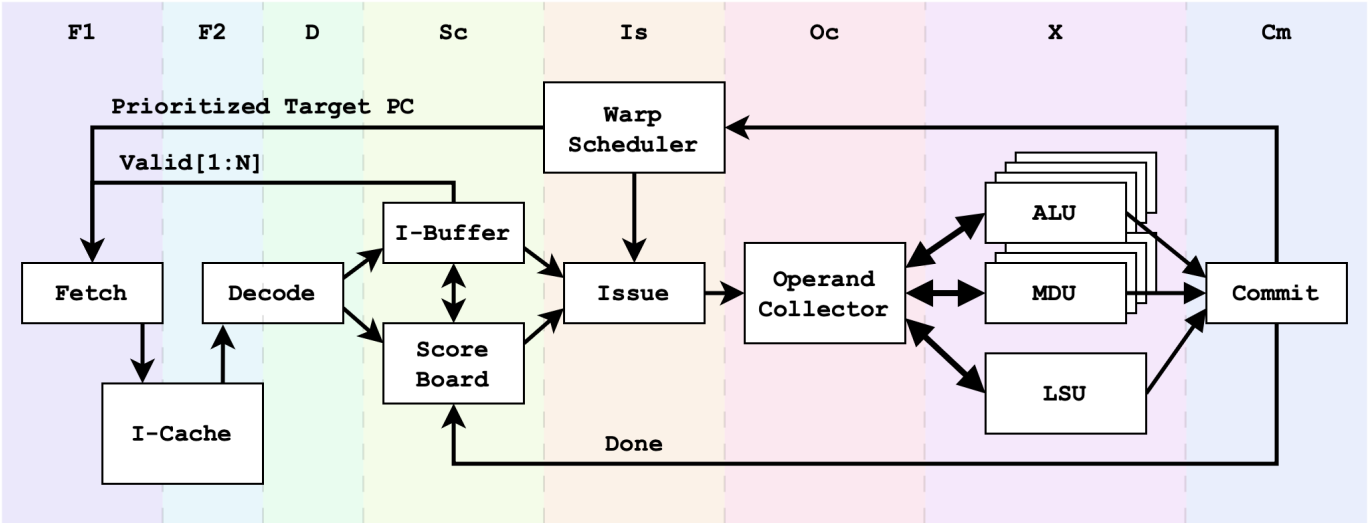
- To trace the execution of individual instructions throughout the pipeline
  - Use different colors to identify the pipeline stage (just as CPUs)



Pipeline Visualization for SIMT Architecture (2/2)

- Meaning of the generated pipeline trace **color scheme** and **symbol**



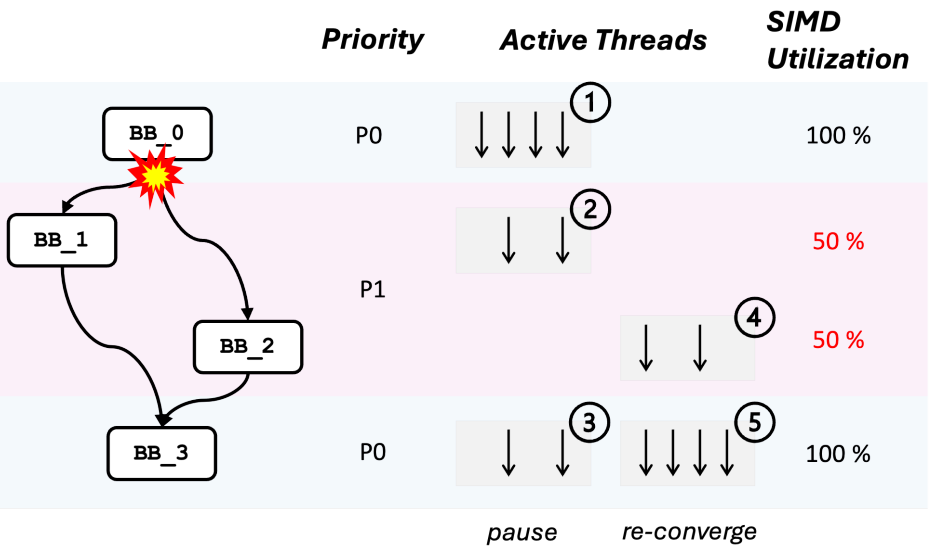


Symbol	Description
F1	The fetch request of the instruction is issued
F2	I-cache responds to the instruction fetch request
D	The instruction is decoded
Sc	The instruction is waiting for scheduling
Is	The instruction is scheduled and issued
Oc	The instruction is collecting operands
X	The instruction is being executed by computing units
Cm	The instruction commits architectural changes
123...	The extra elapsed cycles in the stage

Case Study (1/3): Handling Control Divergence

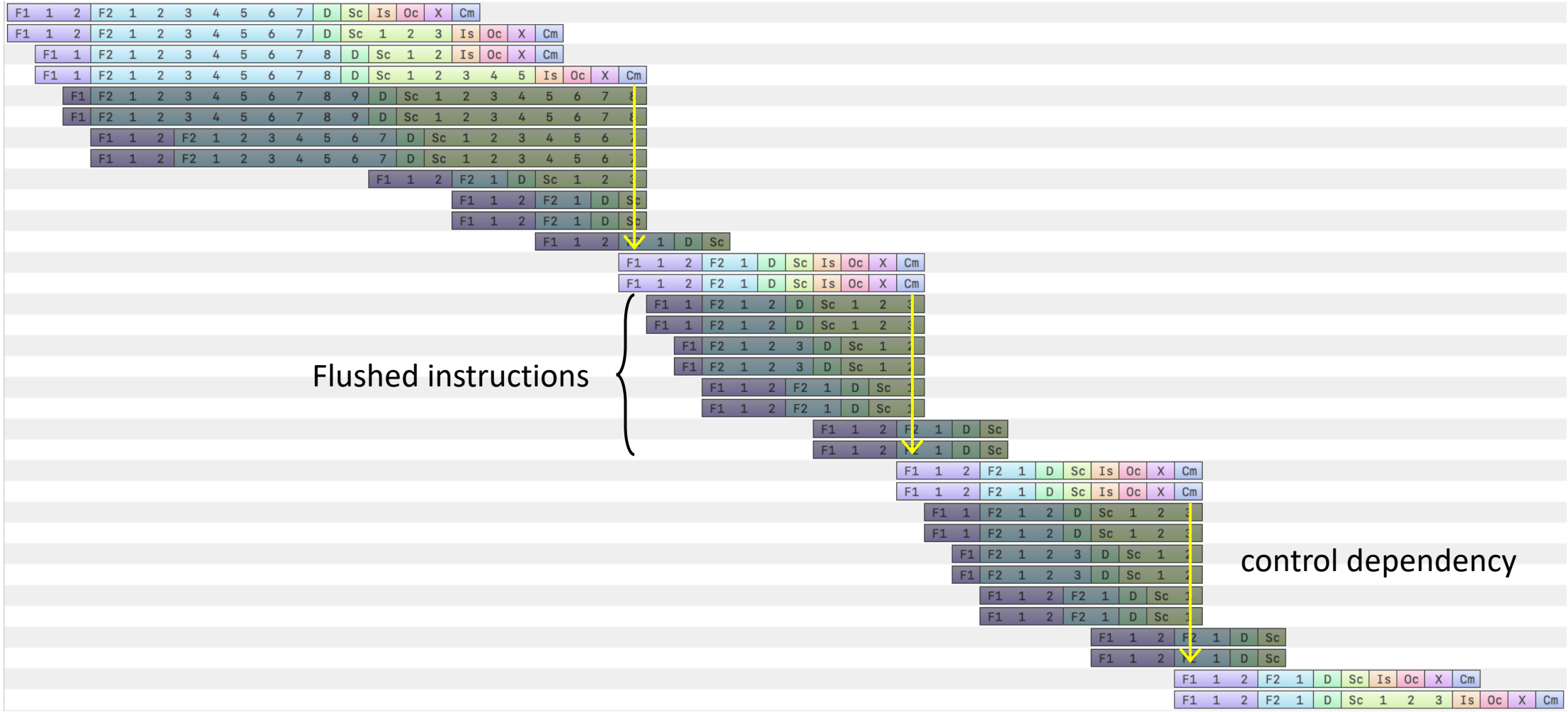
- We reproduce the situation mentioned earlier with the assembly code
  - **Branch taken**: BB_1
  - **Not taken**: BB_2

```
// BB_0
pri_raise();
if (tid % 2 == 0) {
    // BB_1
} else {
    // BB_2
}
pri_lower();
// BB_3
```

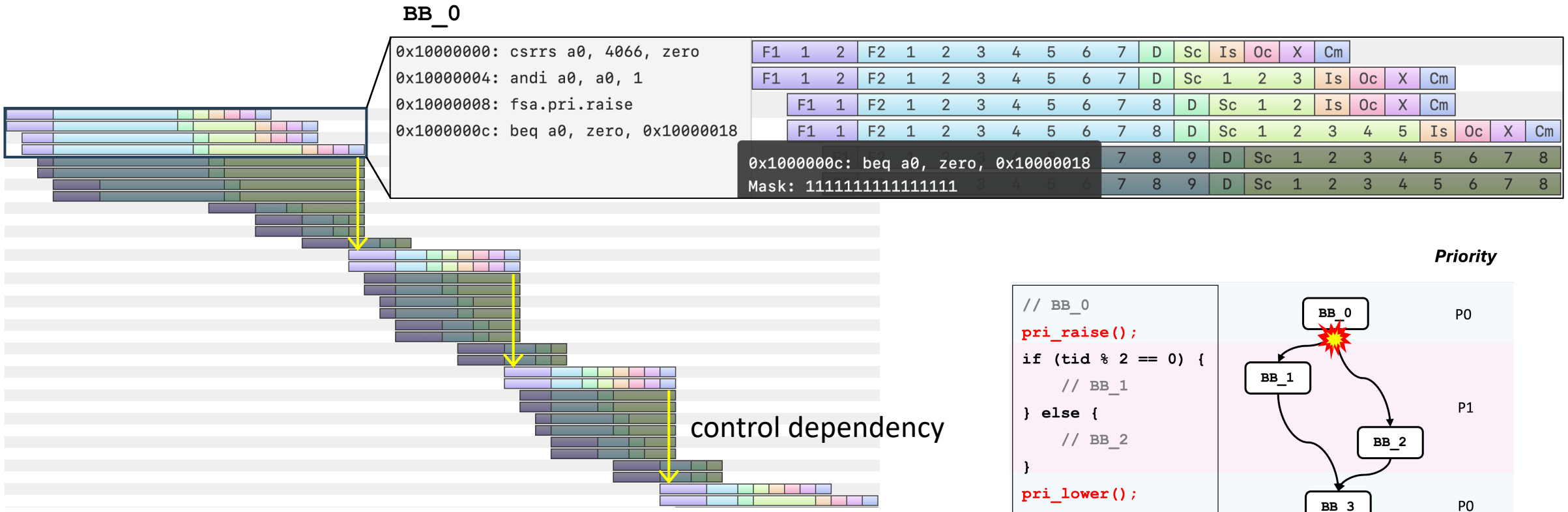



```
.section .text
.global main
main:
bb0:
  csrr a0, 0xfe2   # read tid
  andi a0, a0, 1   # tid % 2
  fsa.pri.raise
  beqz a0, bb1
bb2:
  slli a0, a0, 1
  j bb3
bb1:
  addi a0, a0, -1
bb3:
  fsa.pri.lower
  ret
```

Case Study (1/3): Handling Control Divergence



Case Study (1/3): Handling Control Divergence

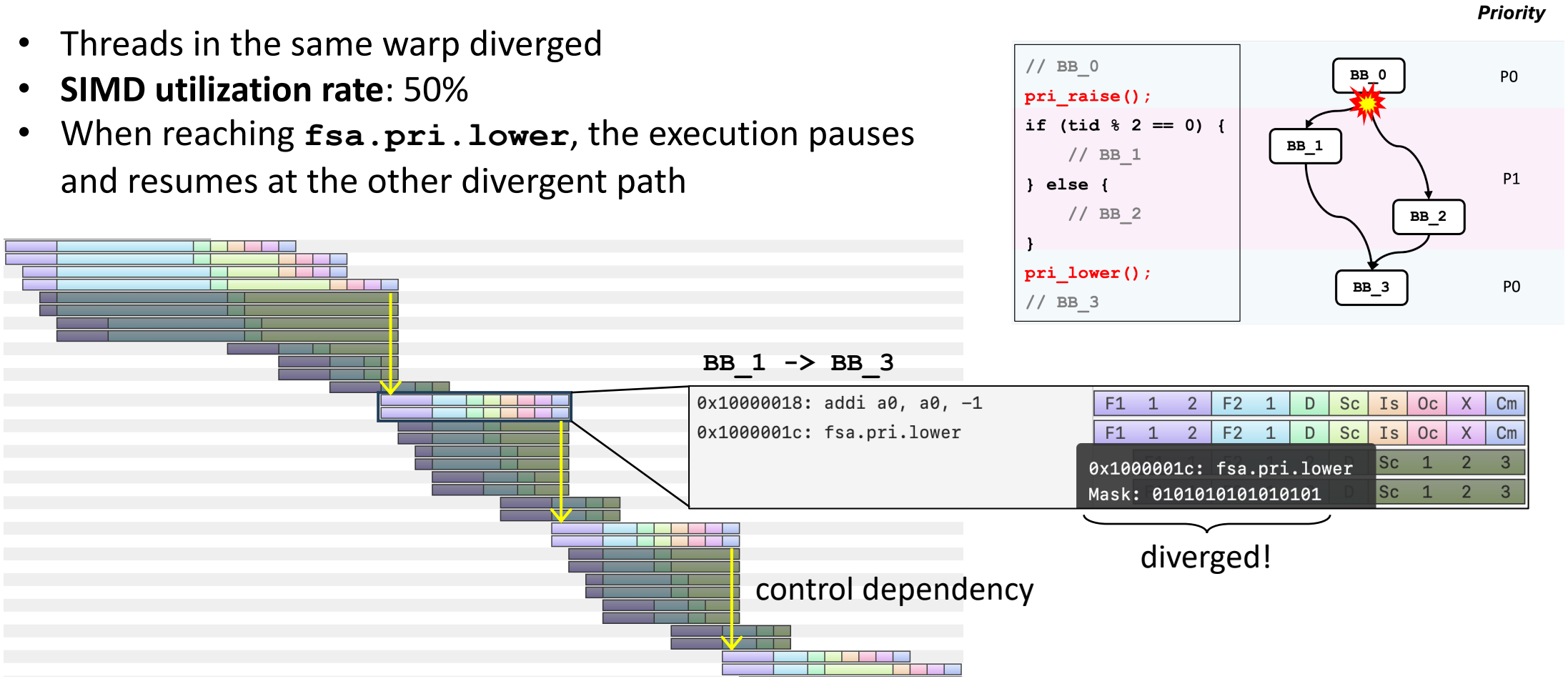


Priority

```
// BB_0
pri_raise();
if (tid % 2 == 0) {
    // BB_1
} else {
    // BB_2
}
pri_lower();
// BB_3
```

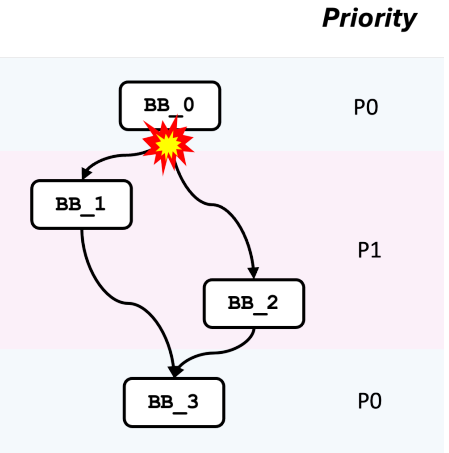
Case Study (1/3): Handling Control Divergence

- Threads in the same warp diverged
- **SIMD utilization rate**: 50%
- When reaching **`fsa.pri.lower`**, the execution pauses and resumes at the other divergent path

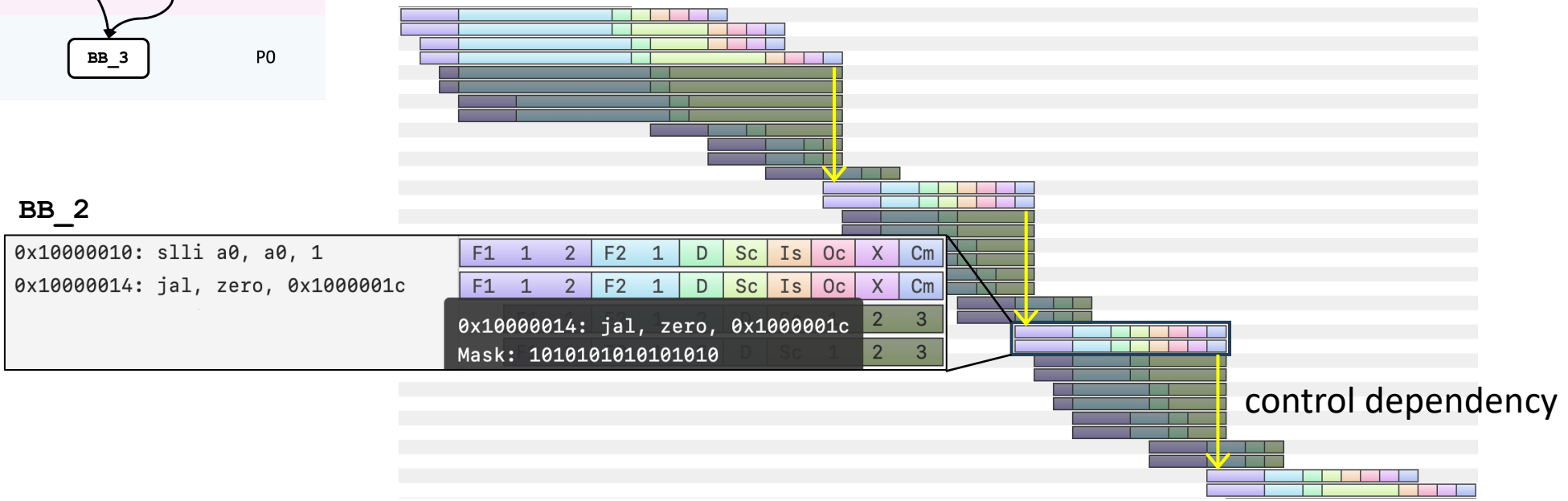


Case Study (1/3): Handling Control Divergence

```
// BB_0
pri_raise();
if (tid % 2 == 0) {
    // BB_1
} else {
    // BB_2
}
pri_lower();
// BB_3
```

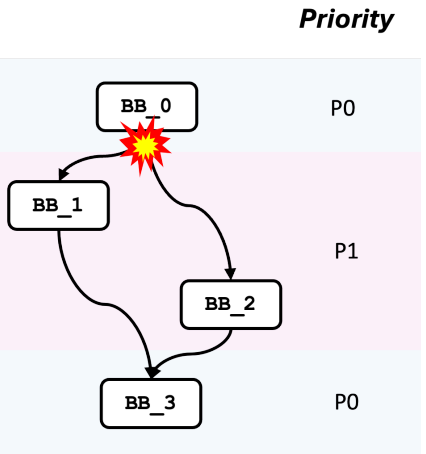


- SM continues to execute the other divergent path
- Threads in the same warp still diverged (with mask inverted)



Case Study (1/3): Handling Control Divergence

```
// BB_0
pri_raise();
if (tid % 2 == 0) {
    // BB_1
} else {
    // BB_2
}
pri_lower();
// BB_3
```

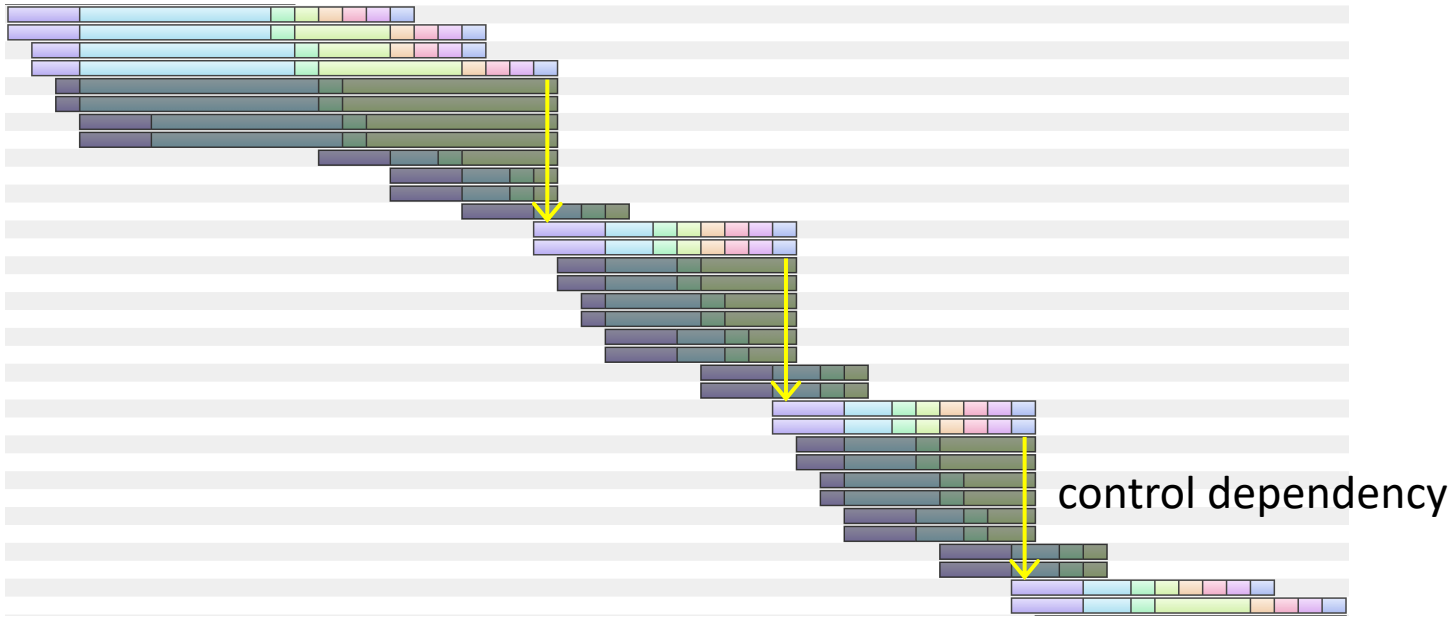


- All threads now share the same priority (P0), resulting in re-convergence

Case Study (1/3): Handling Control Divergence

- ICS-First algorithm handles branch divergence effectively
- Control dependencies have a great impact on performance

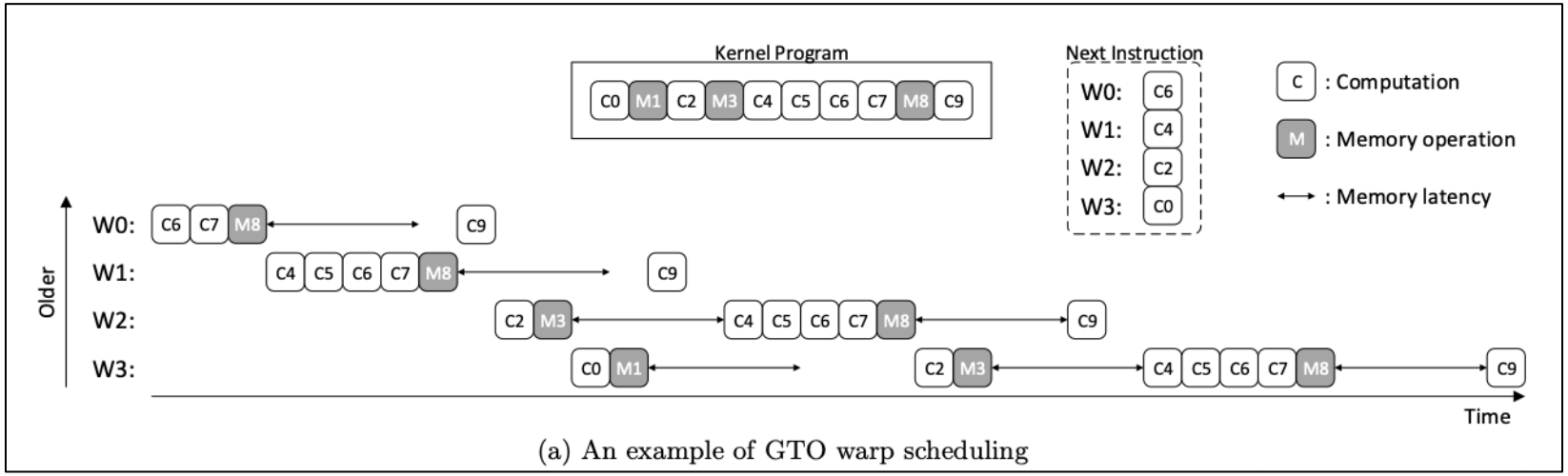
  ⇒ SIMT processor should be able to hide the latency of control flow changes by proactively scheduling distinct warps for execution



Case Study (2/3): Latency Hiding Through TLP

- Stream Multiprocessors (SM) utilize concurrency to hide latencies
  - If one warp stalls, the SM can efficiently use fine-grained multi-threading to switch to another warp
  - Higher Thread-Level Parallelism (TLP)

GTO: Greedy-Than-Oldest

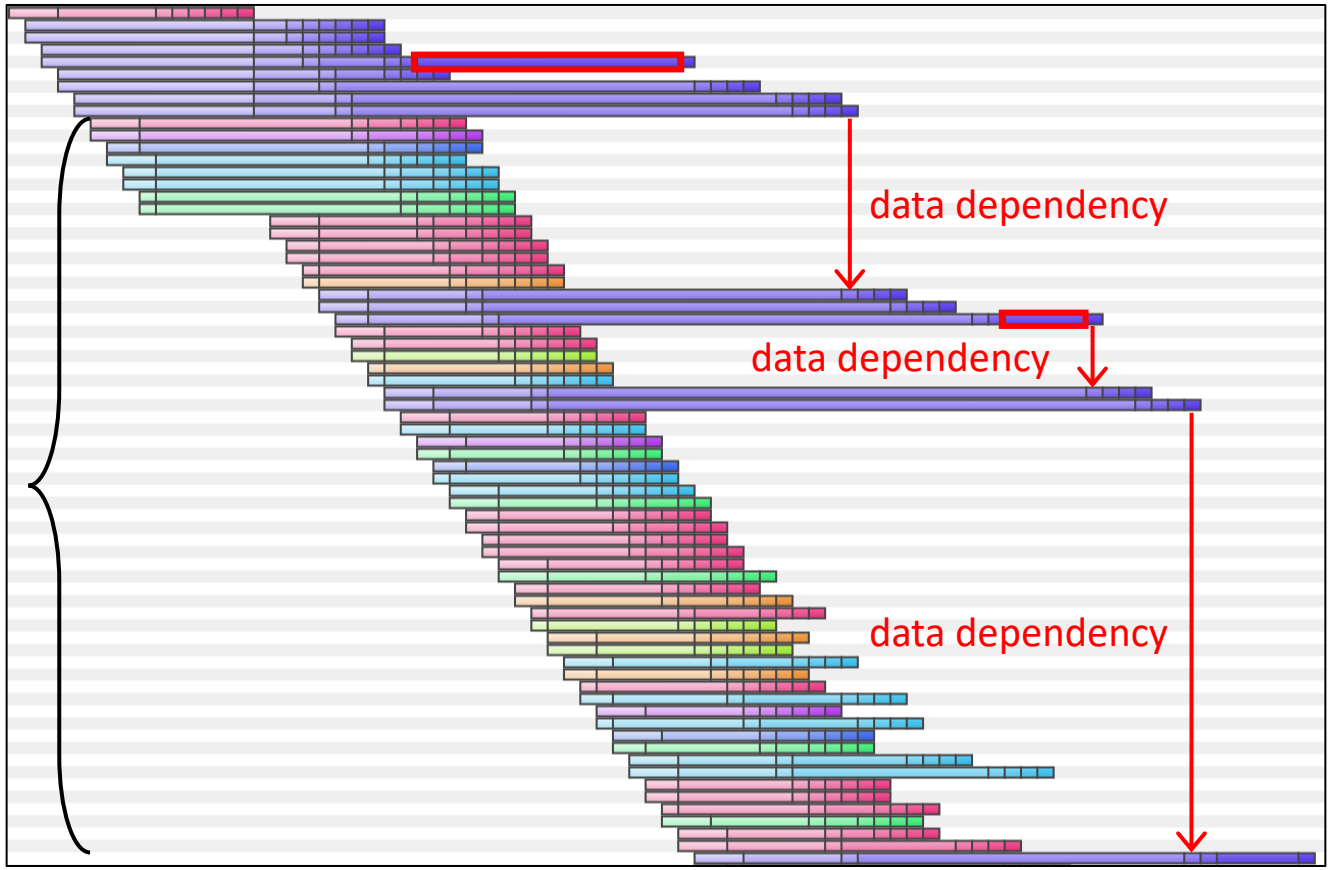


(a) An example of GTO warp scheduling

ref: https://dl.acm.org/doi/10.1145/2807591.2807598

Case Study (2/3): Latency Hiding Through TLP

- **Demo**: A load-use-store code snippets



Latency hiding by scheduling other warps for execution (high TLP)

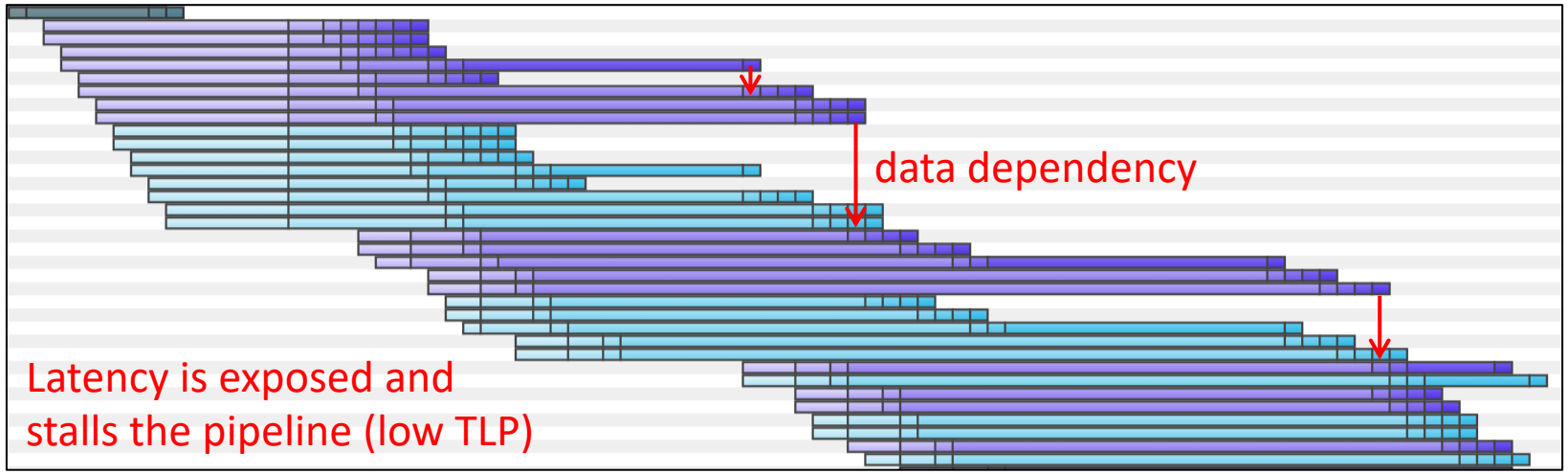
Configuration:

- 8 warps per core
- 4 threads per warp
- GTO scheduler

Case Study (2/3): Latency Hiding Through TLP

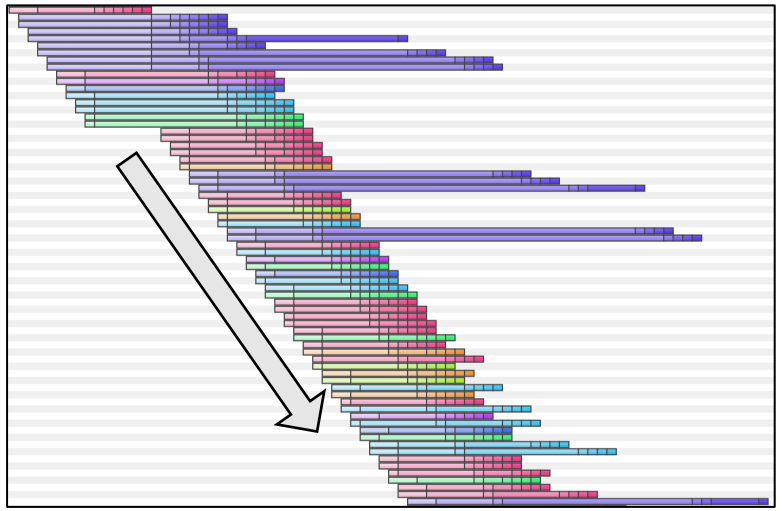
- **Demo**: A load-use-store code snippets

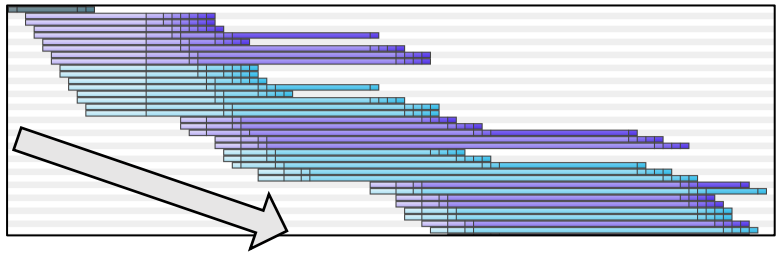
Configuration:
- 2 warps per core
- 16 threads per warp
- GTO scheduler

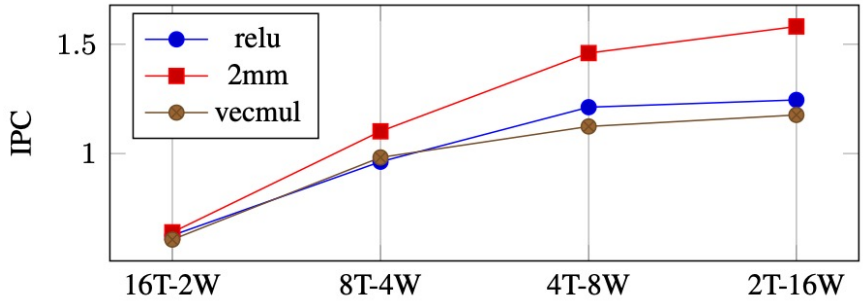


Case Study (2/3): Latency Hiding Through TLP

- **Demo**: A load-use-store code snippets
- The **slope** of the trace roughly indicates the IPC performance
  - **Steeper**: Instructions are issued and committed more frequently (higher IPC)
  - **Flatter**: Pipeline is stalled due to the presence of long-latency instructions (lower IPC)

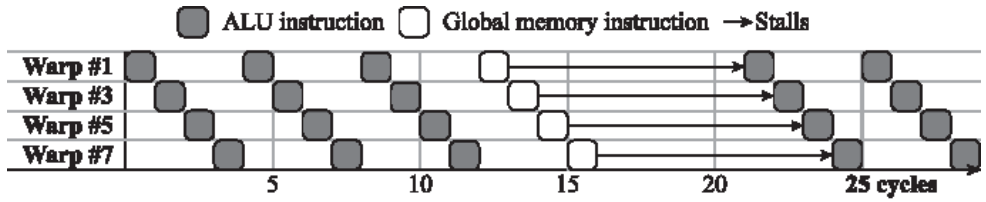




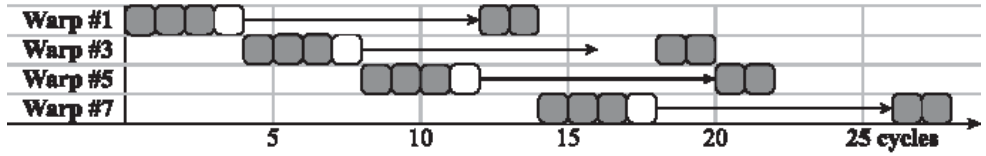


Case Study (3/3): LRR & GTO Scheduling Policies

- **Loosely Round Robin (LRR)**
  - Warps are prioritized for scheduling in round-robin order
  - **"Loosely"**: If a warp cannot issue during its turn, the next warp in round-robin order is given the chance to issue
- **Greedy-Then-Oldest (GTO)**
  - Prioritizes the execution of a single warp until it encounters a blocking latency event



(a) Loosed Round Robin (LRR) warp scheduler

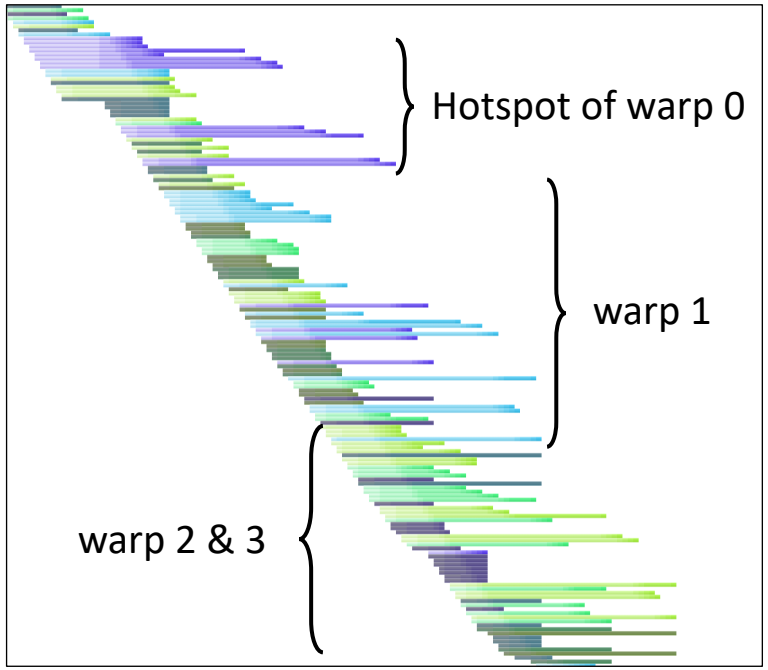


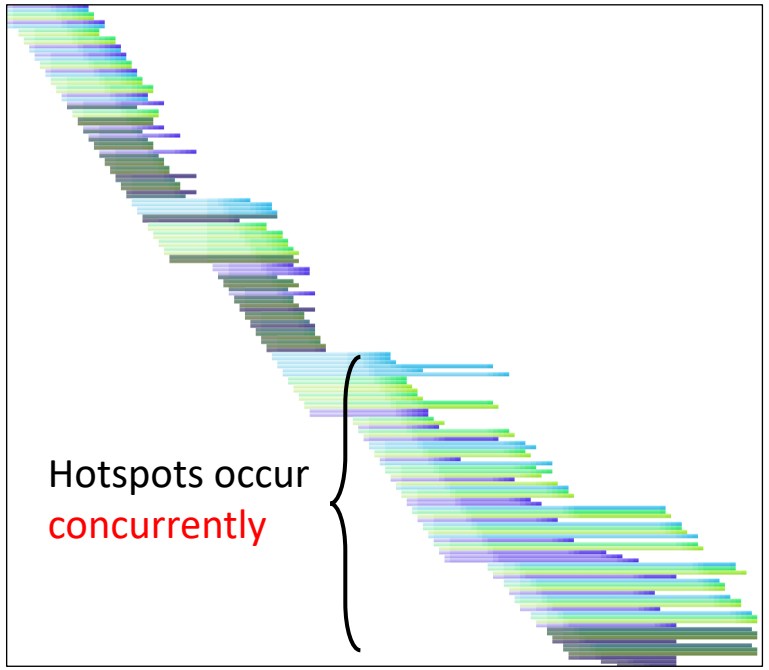
(b) Greedy-Then-Oldest (GTO) warp scheduler

Case Study (3/3): LRR & GTO Scheduling Policies

- We run the same workload with **LRR** and **GTO**
  - **GTO**: The load-use-store program hotspot is scattered
  - **LRR**: All warps reach the hotspot concurrently

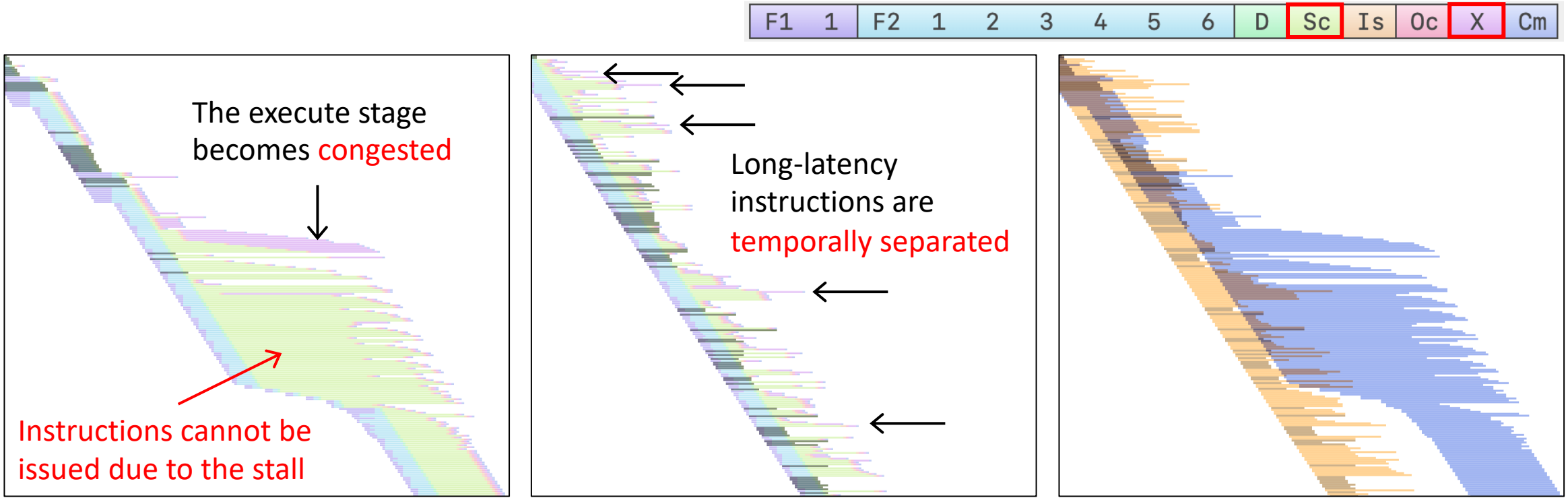
Configuration:
- 4 warps per core
- 8 threads per warp
- GTO scheduler





Case Study (3/3): LRR & GTO Scheduling Policies

- The concurrent occurrence of program hotspots can potentially stall the pipeline (workload: 2mm)



Conclusion and Future Work

- Pipeline visualization is also helpful for SIMT architectures
  - Architecture exploration
  - Performance bottleneck analysis
  - Education in computer architecture
- Rendered pipeline trace can serve as a reference model for RTL implementations
  - Validate the behavior of the implementation
  - Improve the accuracy of the cycle-based model
- Integration to SOTA GPU simulation platforms (Accel-Sim)

Thank you!