# Stream-based Memory Specialization for General Purpose Processors

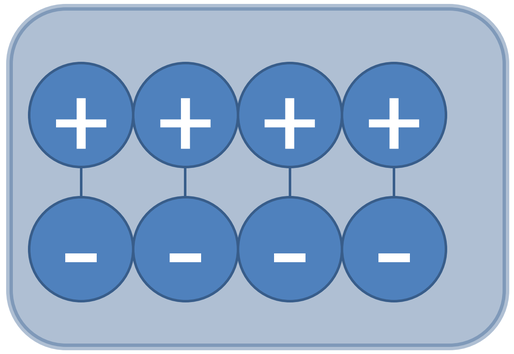
Zhengrong Wang

Tony Nowatzki

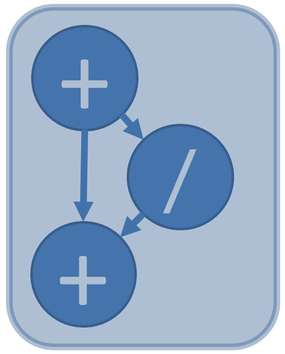
# Computation & Memory Specialization

Computation Structure

SIMD

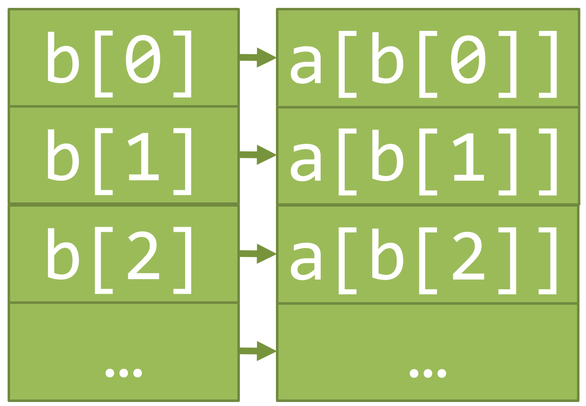


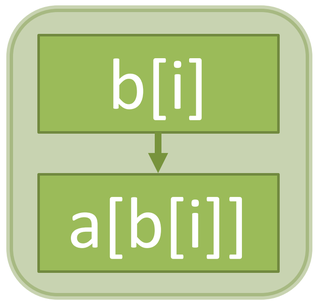

Compound Instruction




ISA abstraction for certain computation pattern.

Memory Structure



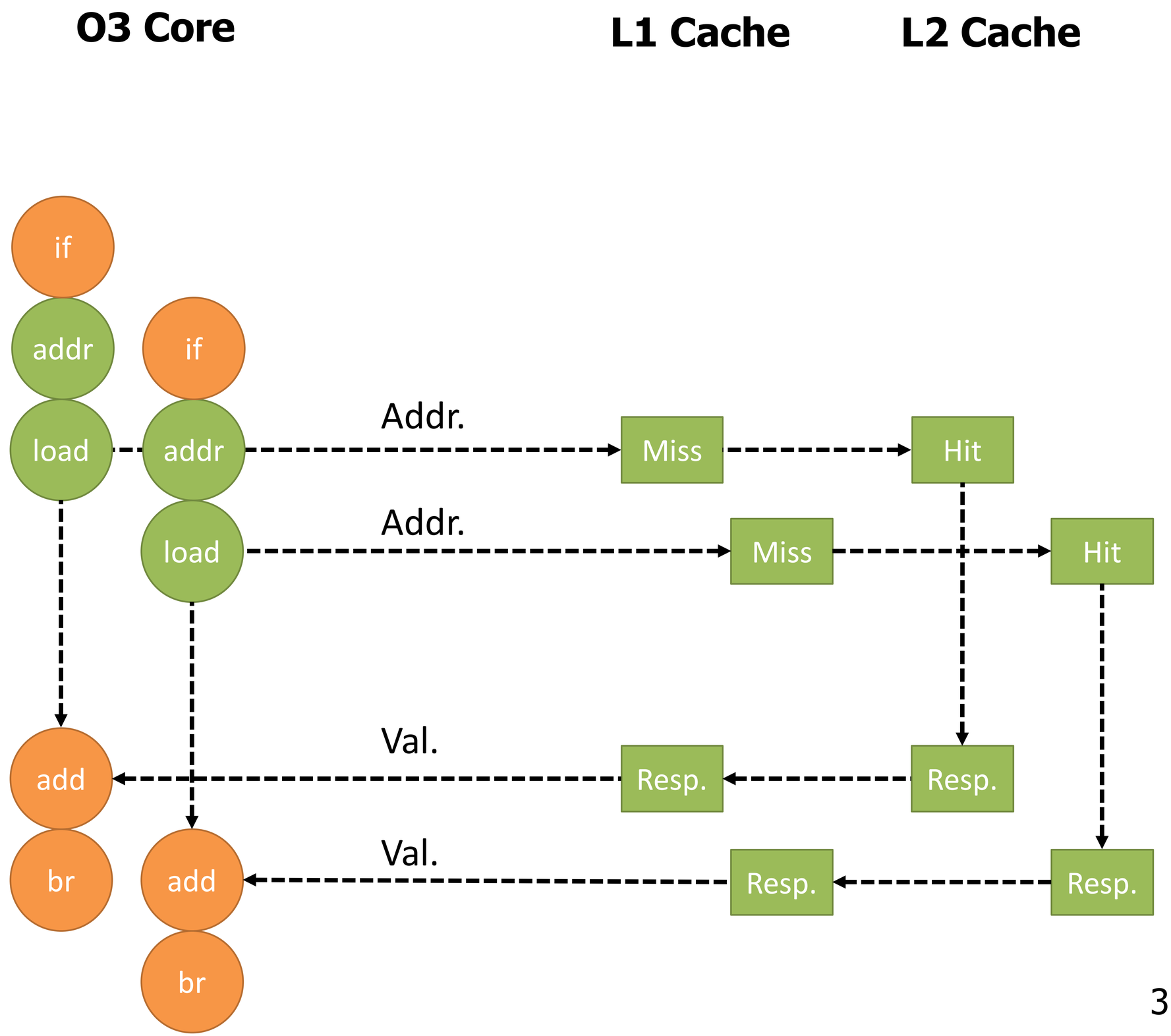




Stream

New ISA abstraction for memory access pattern?

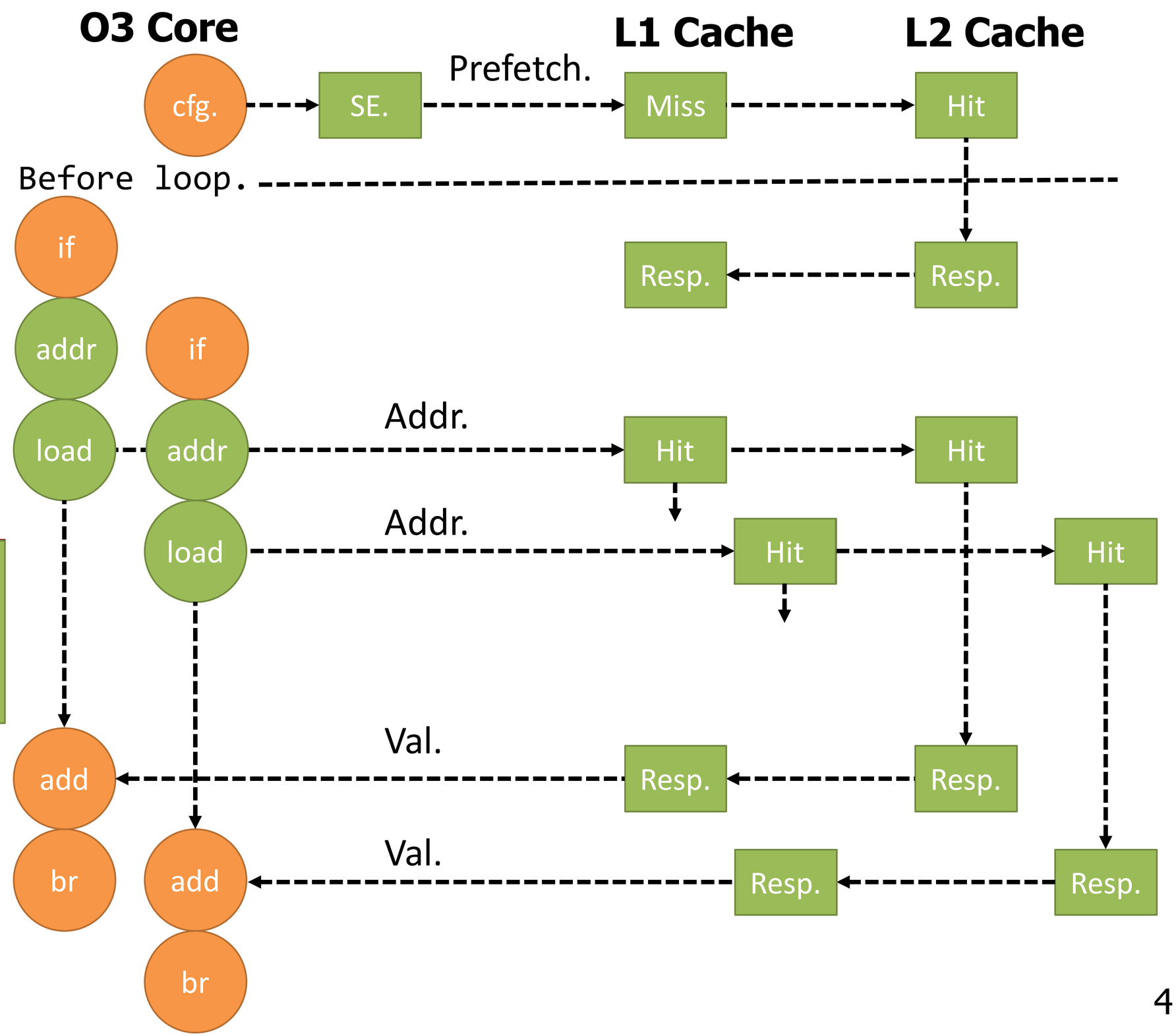
# Conventional Memory Abstraction

```
while (i < N) {
  if (cond)
    v += a[i];
  i++;
}
```



# Opportunity 1: Prefetch with Ctrl. Flow

```
cfg(a[i]);
while (i < N) {
  if (cond)
    v += a[i];
  i++;
}
```



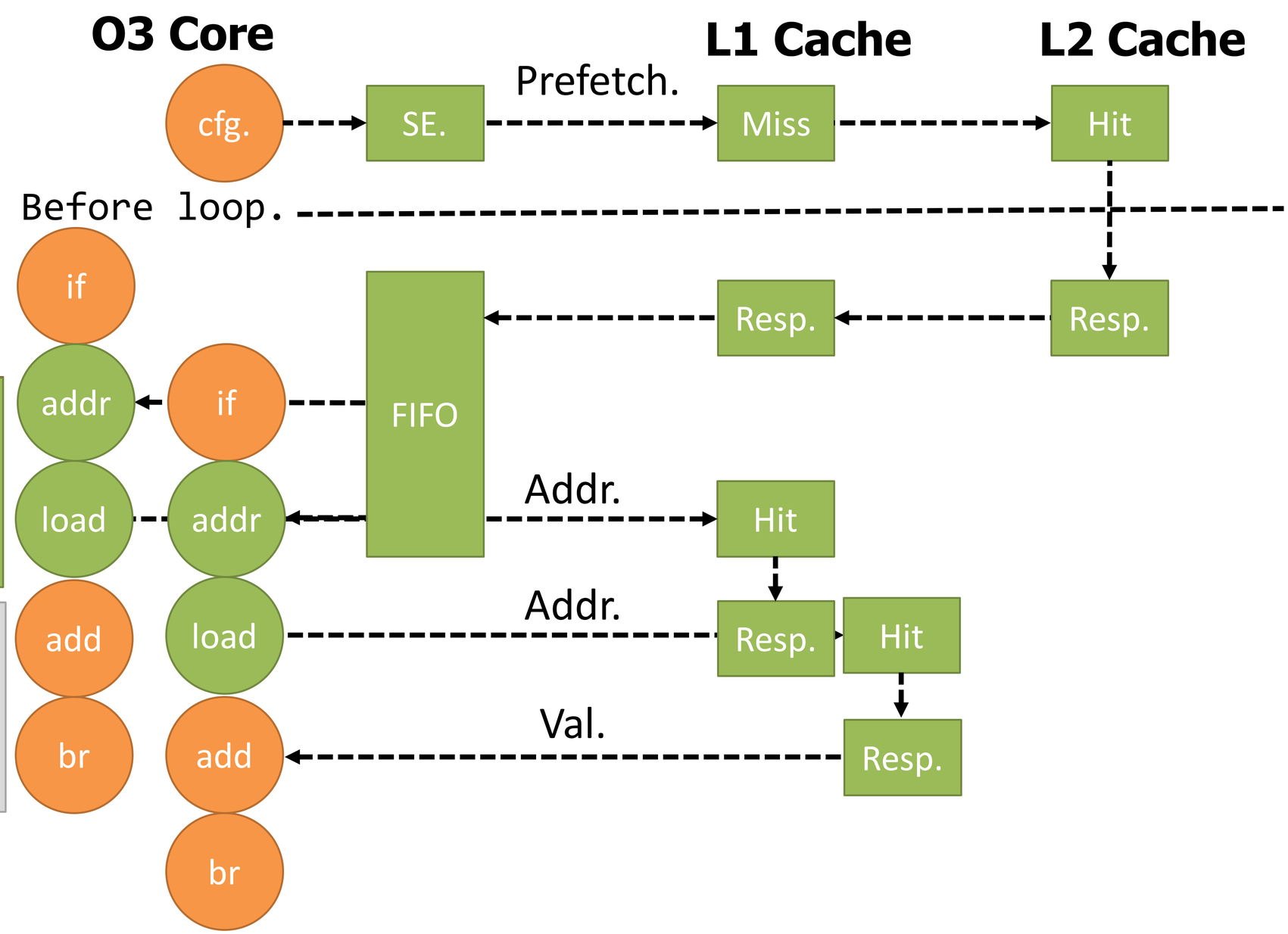

Opportunity 1: Prefetch with control flow.

# Opportunity 2: Semi-Binding Prefetch

```
s_a = cfg();
while (i < N) {
  if (cond)
    v += s_a;
  i++;
}
```

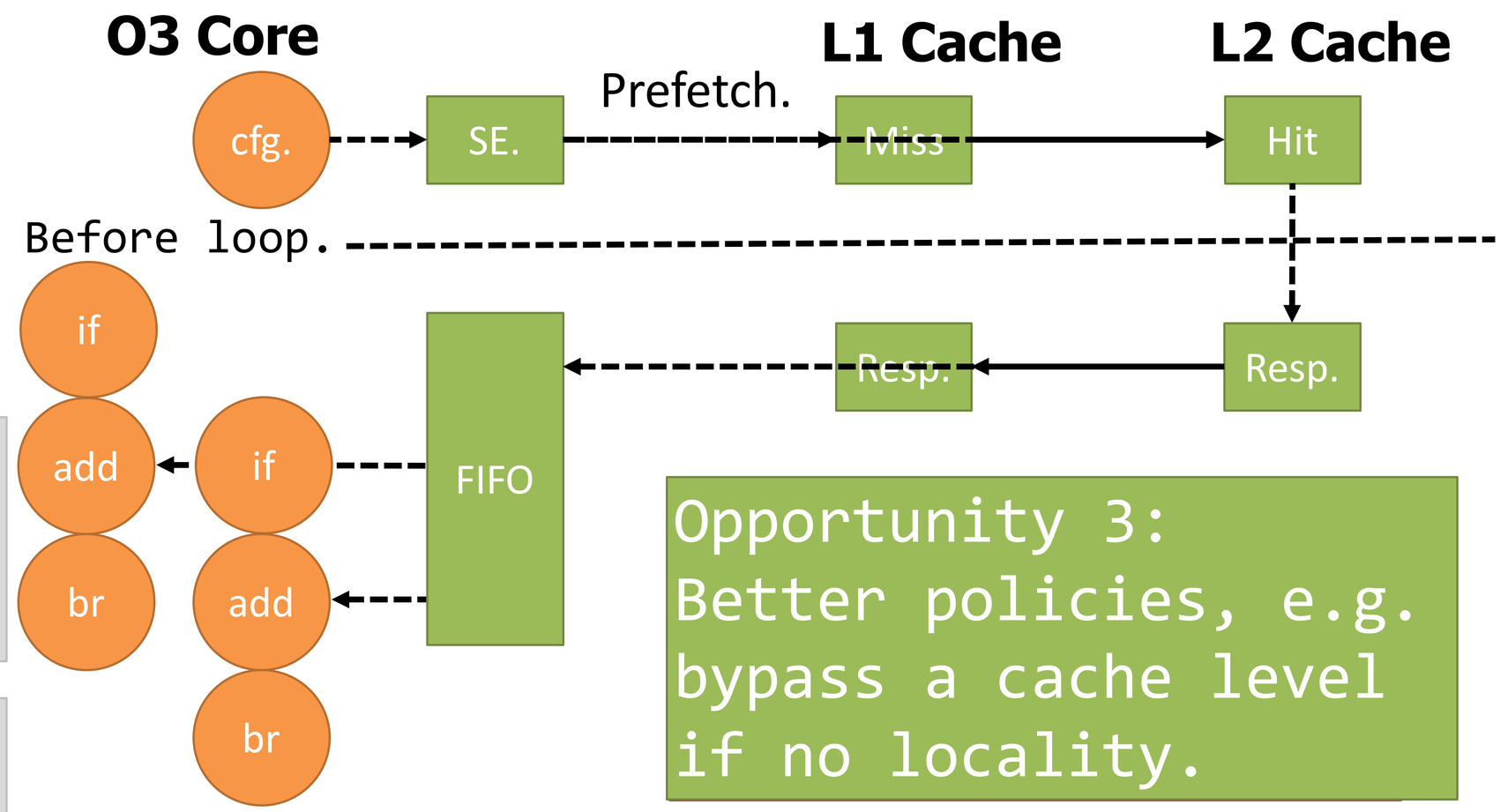
Opportunity 2: Semi-binding prefetch.

Opportunity 1: Prefetch with control flow.



# Opportunity 3: Stream-Aware Policies

```
s_a = cfg();
while (i < N) {
  if (cond)
    v += s_a;
}
```




Opportunity 2: Semi-binding prefetch.

Opportunity 1: Prefetch with control flow.

Opportunity 3: Better policies, e.g. bypass a cache level if no locality.

# Stream: A New ISA Memory Abstraction

- Potential Benefits of Stream Abstractions
  - Energy-efficient prefetching.
  - Offload Memory from expensive OOO Pipeline
  - Leverage stream information in cache policies.
- This Work
  - Analyze programs to determine the requirements of the ISA
  - Develop a hardware-neutral ISA extension*: decoupled-stream ISA*
  - Develop micro-architecture extensions for an OOO processor
- Evaluation
  - 60% memory accesses → streams.
  - 1.37× speedup over a traditional O3 processor.

# Outline

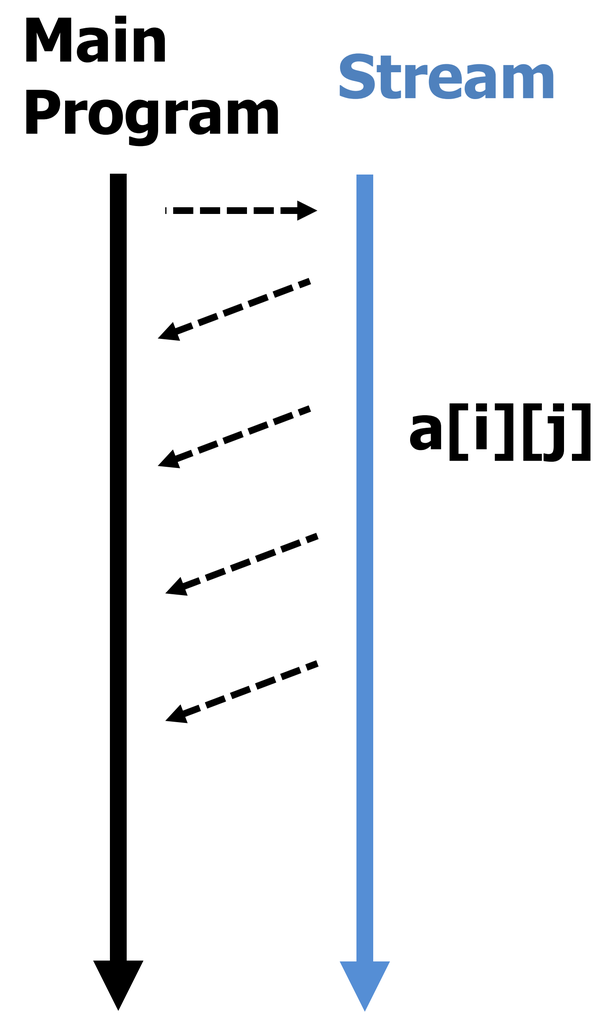
- Stream Characteristics.
- Stream ISA Extension.
- Stream-Aware Policies.
- Microarchitecture Extension.
- Evaluation.

# Outline

- Stream Characteristics.
- Stream ISA Extension.
- Stream-Aware Policies.
- Microarchitecture Extension.
- Evaluation.

# Stream Definition

- Stream -- Decoupled portion of a program:
  - Accesses memory at most one times
  - No internal control flow
  - Has no dependences from the main program to the stream besides configuration and termination

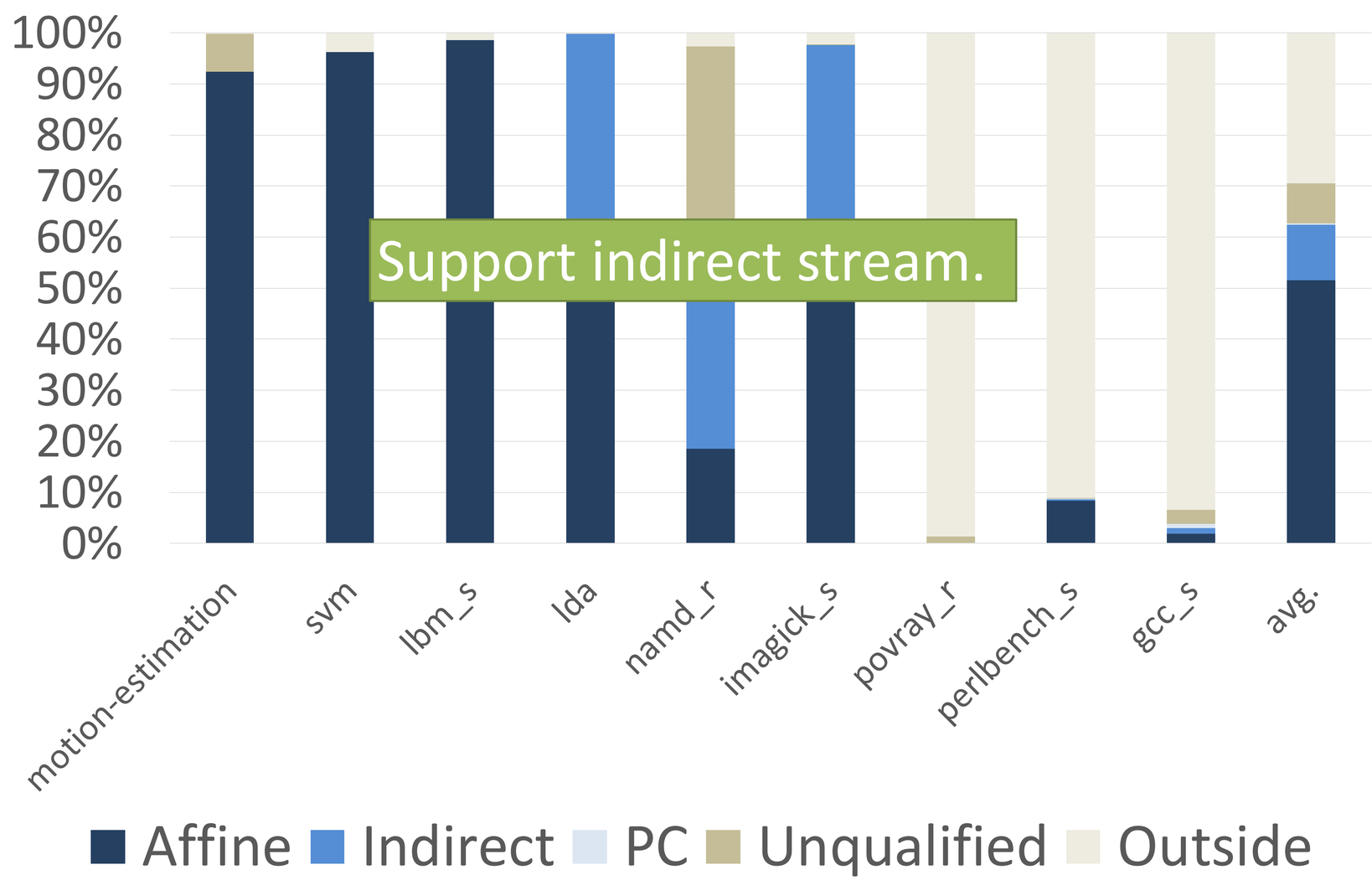



```
for (int i = 0; i < M; ++i) {
  for (int j = 0; j < N; ++j) {
    … = a[i][j];
```

# Stream Characteristics – Stream Type

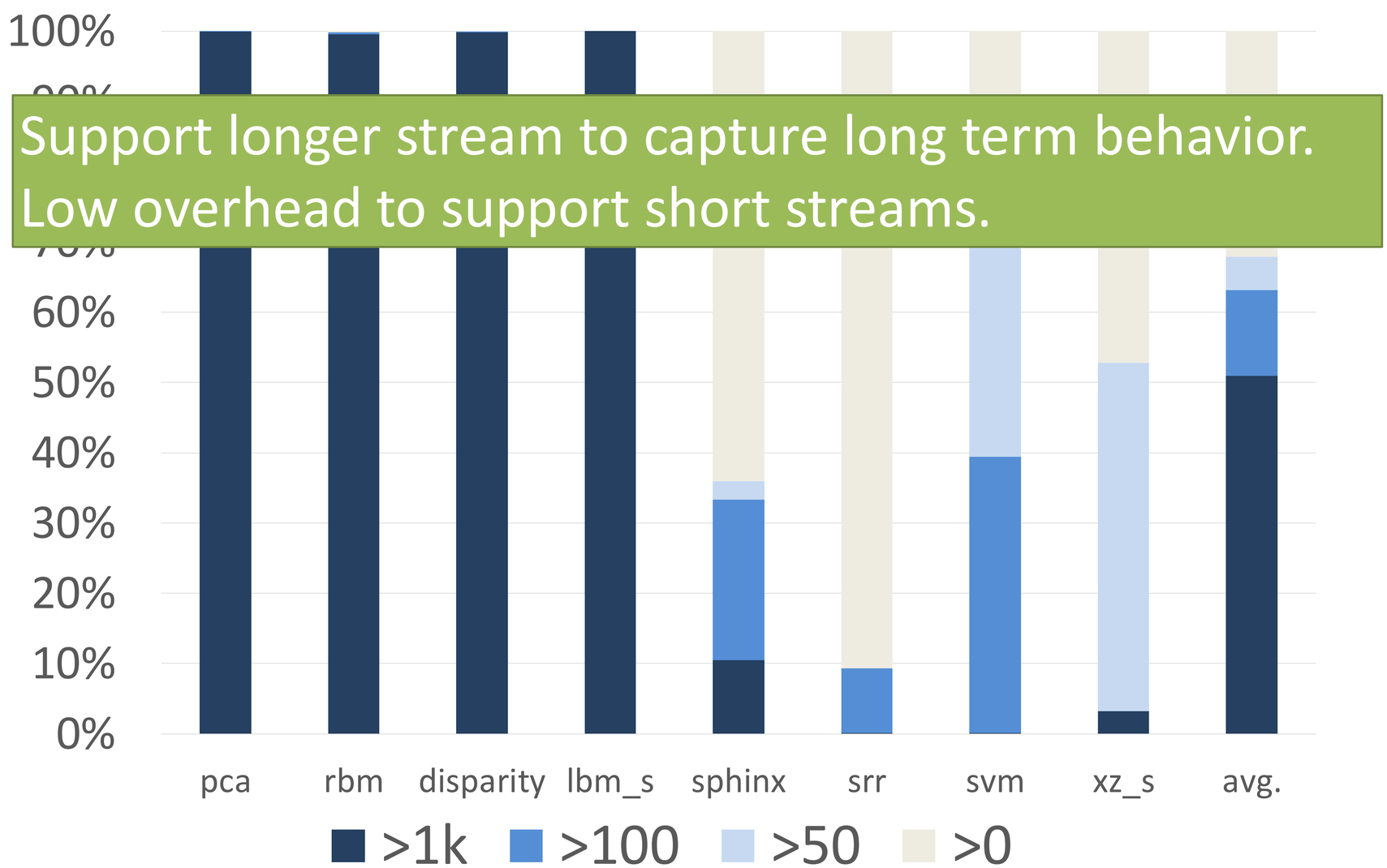
Trace analysis on CortexSuite/SPEC CPU 2017.

- 51.49% affine, 10.19% indirect.
- Indirect streams can be as high as 40%.



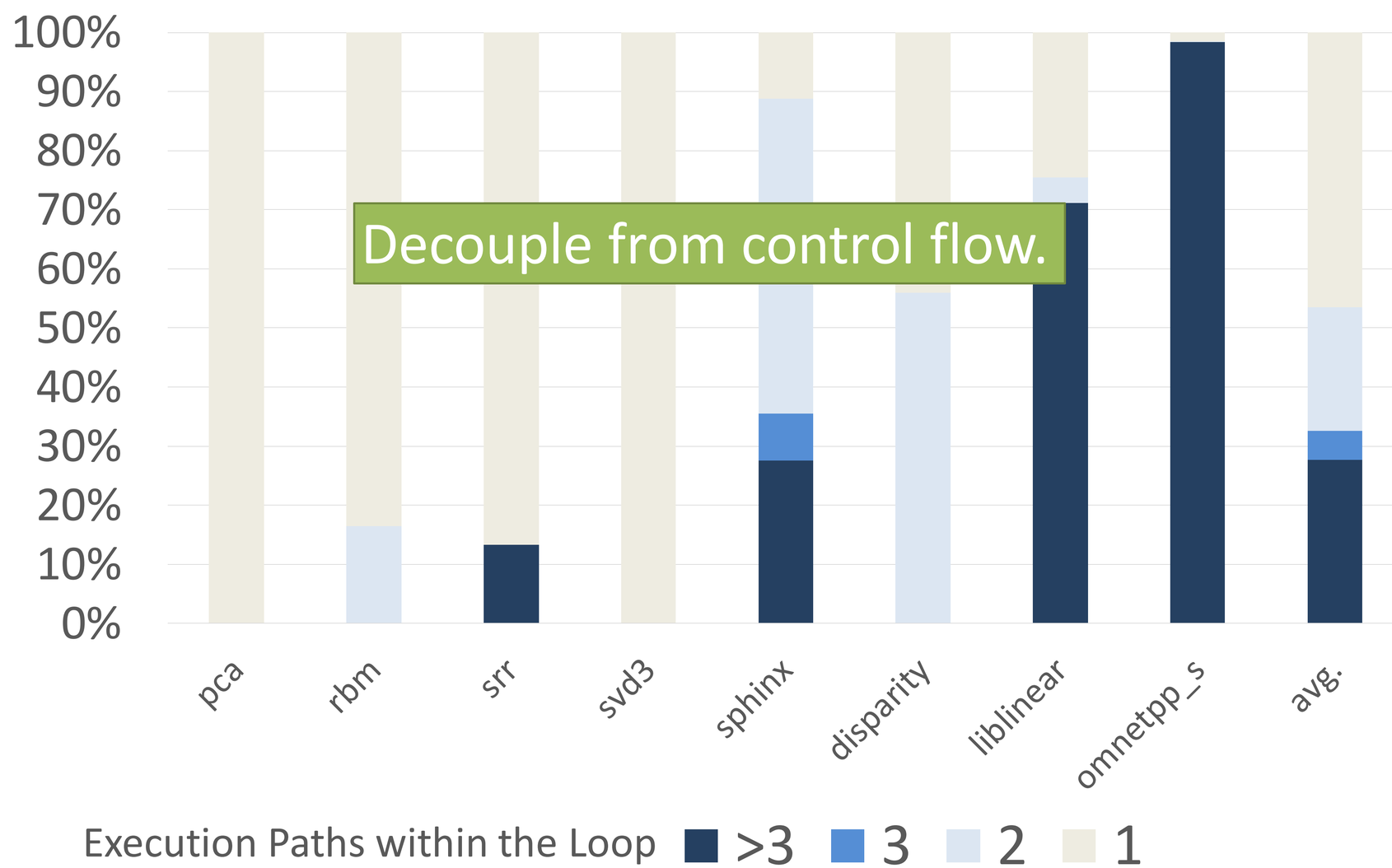
# Stream Characteristics – Stream Length

- 51% stream accesses from stream longer than 1k.
- Some benchmarks contain short streams.



# Stream Characteristics – Control Flow

- 53% stream accesses from loop with control flow.



# Outline

- Stream Characteristics.
- Stream ISA Extension.
- Stream-Aware Policies.
- Microarchitecture Extension.
- Evaluation.

# Decoupled Stream ISA Components

- Setup/Teardown
  - stream_config(`s_i`) -- setup stream parameters
  - stream_end(`s_i`) -- deallocates streams
- Pseudo Register
  - Register mapped to stream data
  - Allows offset to access different elements of stream
- Supporting Control Flow
  - Decoupled stream advance from use of pseudo-register
  - step(`s_i`) – update the position of a stream by one iter.

# Stream ISA Extension – Basic Example

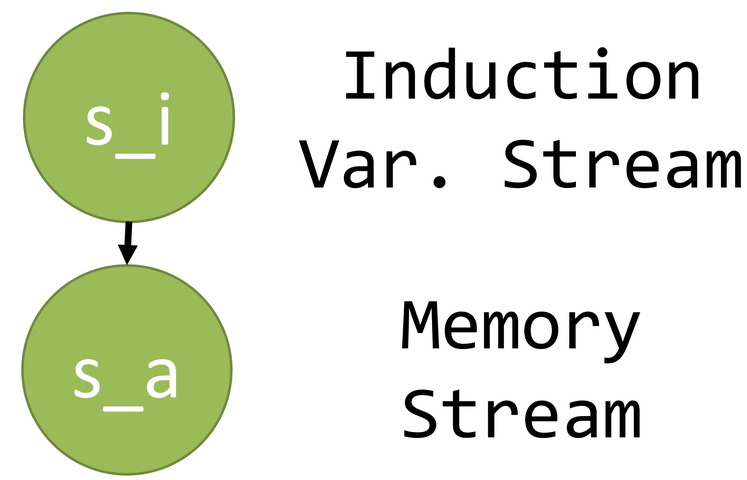
**Original C Code**

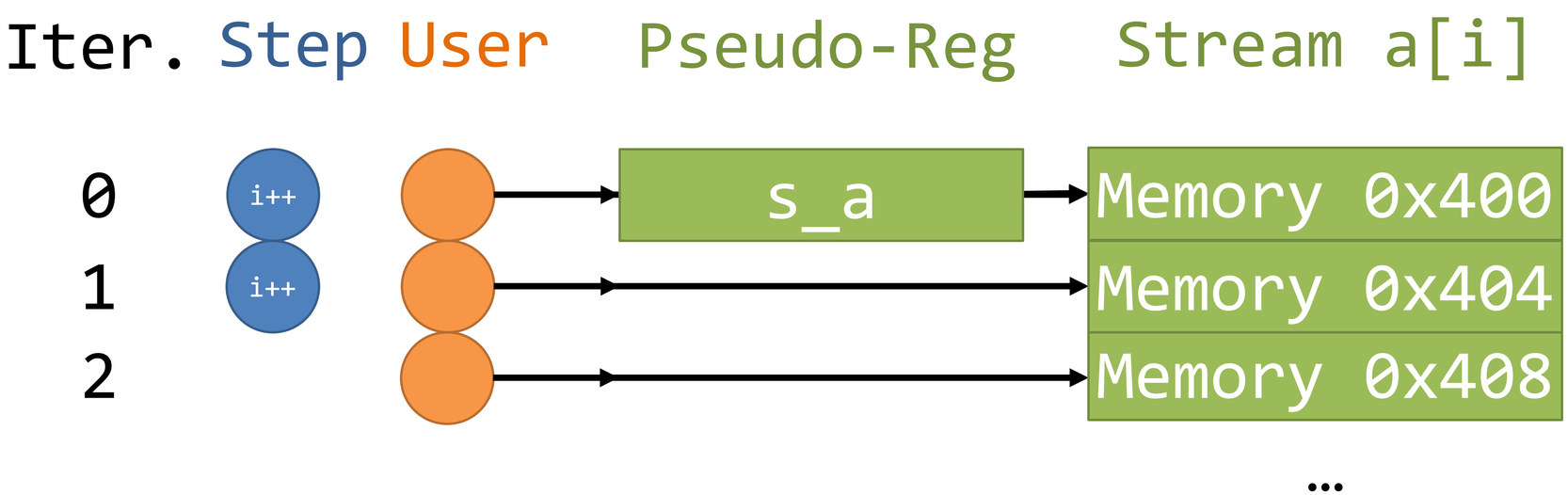
```
int i = 0;
while (i < N) {
  sum += a[i];
  i++;
}
```

**Stream Decoupled Pseudo Code**

```
stream_cfg(s_i, s_a);
while (s_i < N) {
  sum += s_a;
  stream_step(s_i);
}
stream_end(s_i, s_a);
```

**Stream Dependence Graph**



# Stream ISA Extension – Control Flow

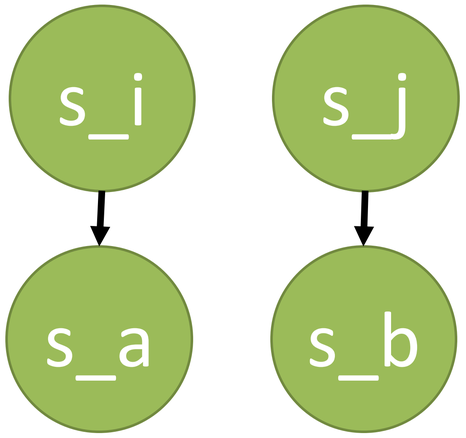
**Original C Code**

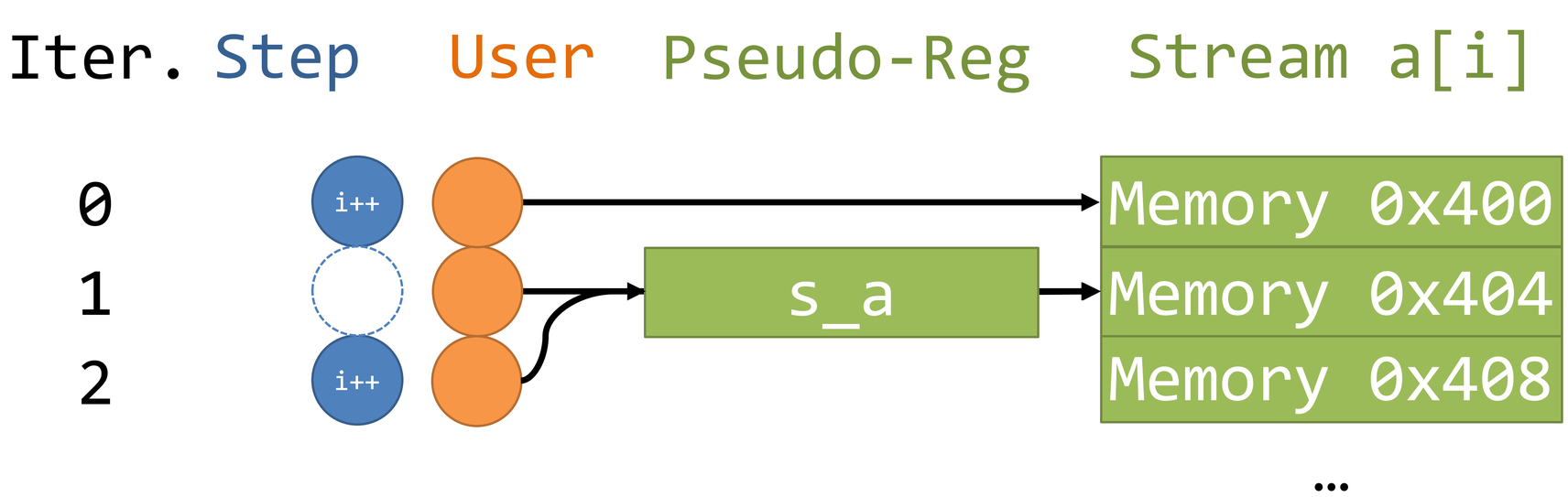
```
int i = 0, j = 0;
while (cond) {
  if (a[i] < b[j])
    i++;
  else
    j++;
}
```

**Stream Decoupled Pseudo Code**

```
stream_cfg(s_i, s_a, s_j, s_b);
while (cond) {
  if (s_a < s_b)
    stream_step(s_i);
  else
    stream_step(s_j);
}
stream_end(s_i, s_a, s_j, s_b);
```

**Stream Dependence Graph**



# Stream ISA Extension – Indirect Stream

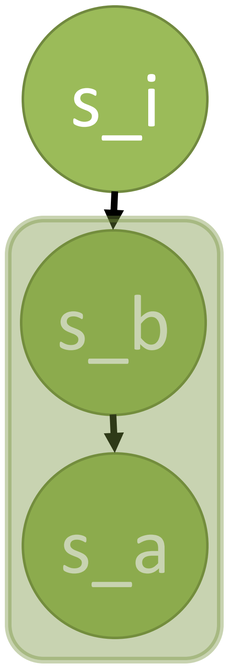
**Original C Code**

```c
int i = 0;
while (i < N) {
  sum += a[b[i]];
  i++;
}
```

**Stream Decoupled Pseudo Code**

```c
stream_cfg(s_i, s_a, s_b);
while (s_i < N) {
  sum += s_a;
  stream_step(s_i);
}
stream_end(s_i, s_a, s_b);
```

**Stream Dependence Graph**



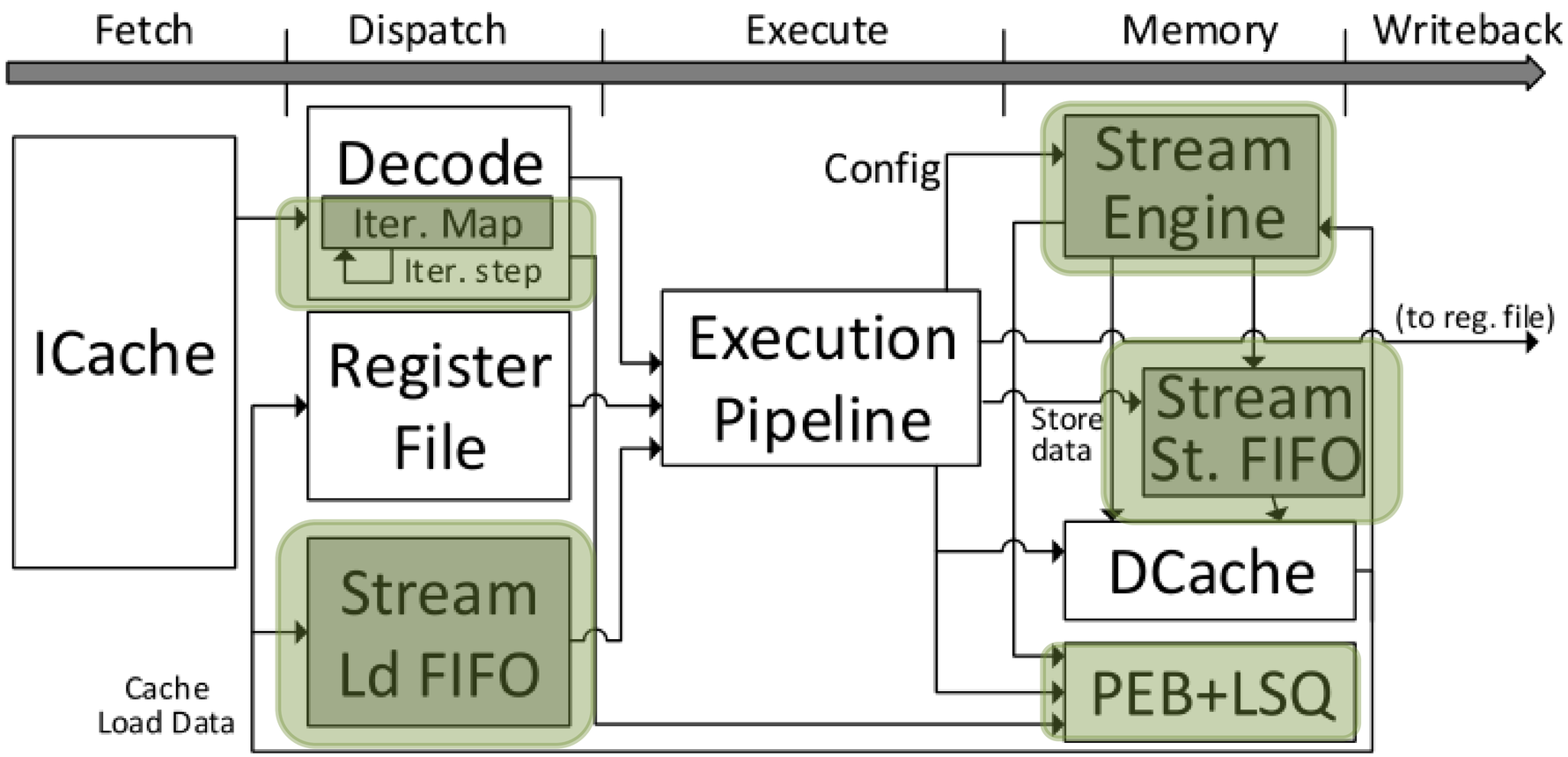

# Implications of Decoupled-Stream Extensions

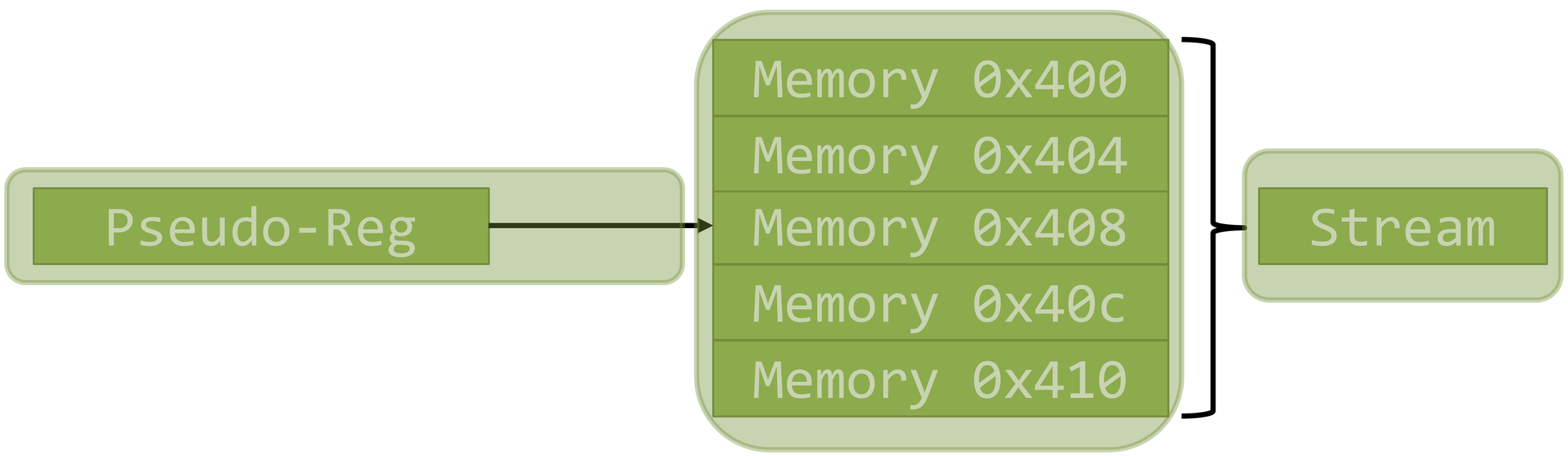
- New architectural states:
  - Stream configuration.
  - Current iteration's data
- Maintain the memory order.
  - Load → first use of the pseudo-register after configured/stepped.
- New speculation in ISA:
  - Stream elements will be used.
  - Streams are not too short (overfetch).
- "Vectorize" access that was not vectorizable before
  - From the perspective of L1-Cache->Core
  - Regular parts of irregular program regions

# Outline

- Stream Characteristics.
- Stream ISA Extension.
- Stream-Aware Policies.
- Microarchitecture Extension.
- Evaluation.

# Microarchitecture



# Microarchitecture – Misspeculation

- Control misspeculated stream_step.
  - Decrement the iteration map.
  - No need to flush the FIFO and re-fetch data (decoupled) !
- Other misspeculation.
  - Revert the stream states, including stream FIFO.
- Memory fault delayed until the use of the element.

# Outline

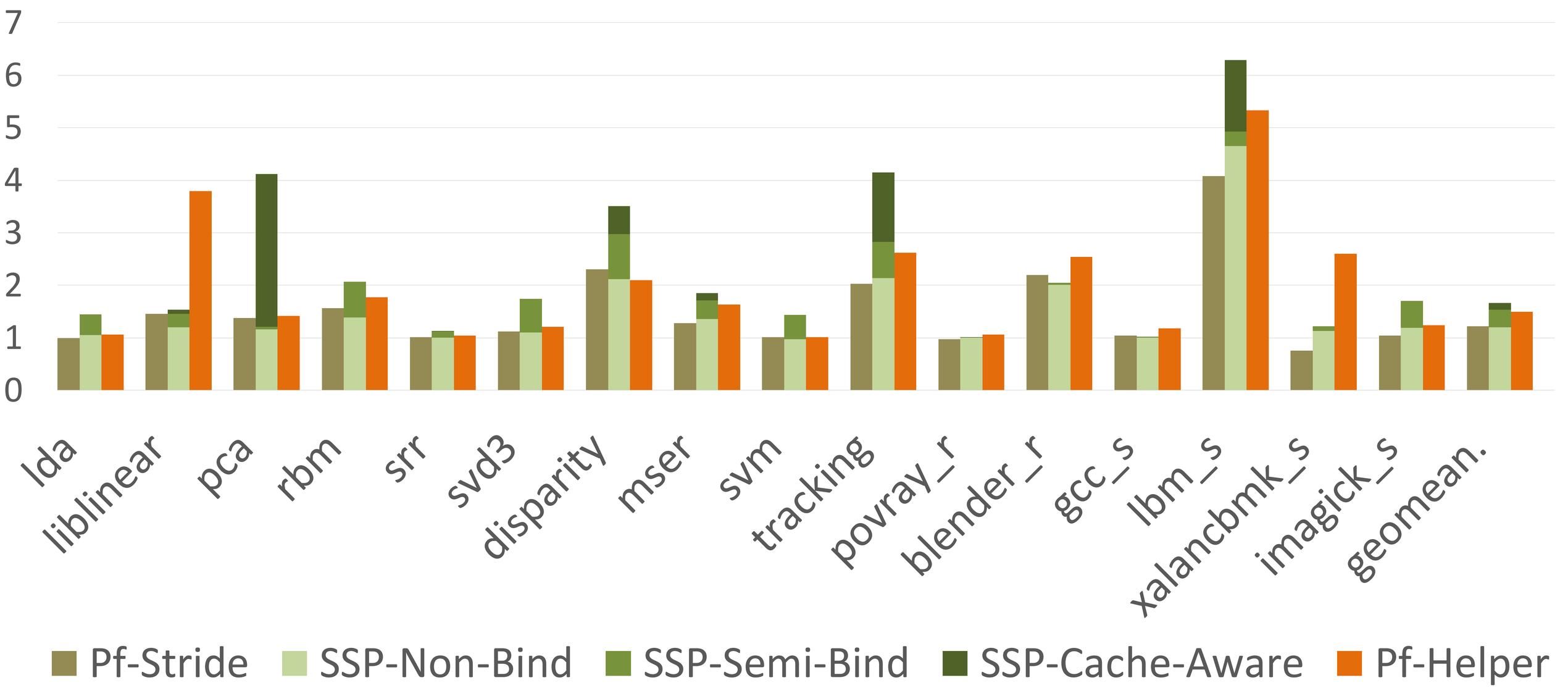
- Stream Characteristics.
- Stream ISA Extension.
- Microarchitecture Extension.
- Stream-Aware Policies.
- Evaluation.

# Methodology

- Compiler in LLVM:
  - Identify stream candidates.
  - Generate stream configuration.
  - Transform the program.
- Gem5 + McPAT simulation.
- 33 Benchmarks:
  - SPEC2017 C/CPP benchmarks.
  - CortexSuite.
- SimPoint:
  - 10 million instructions' simpoints.
  - ~10 simpoints per benchmark.

| | |
|---|---|
| CPU | 2.0GHz 8-Way OoO Cores<br>8-wide fetch/issue/commit<br>64 IQ, 32 LQ, 32 SQ, 192 ROB<br>256 Int RF, 256 FP RF<br>speculative scheduling |
| Function Units | 6 Int ALU (1 cycle)<br>2 Int Mult/Div (3/20 cycles)<br>4 FP ALU (2 cycles)<br>2 FP Mult/Div (4/12 cycles)<br>4 SIMD (1 cycle) |
| Private L1 ICache | 32KB / 8-way<br>8 MSHRs / 2-cycle latency |
| Private L1 DCache | 32KB / 8-way<br>8 MSHRs / 2-cycle latency |
| Private L2 Cache | 256KB / 16-way<br>16 MSHRs / 15-cycle latency |
| To L3 Bus | 16-byte width |
| Shared L3 Cache | 8MB / 8-way<br>20 MSHRs / 20-cycle latency |
| DRAM | 2 channel / 1600MHz DDR3 12.8 GB/s |

# Results – Overall Performance




Traditional Stride Prefetcher

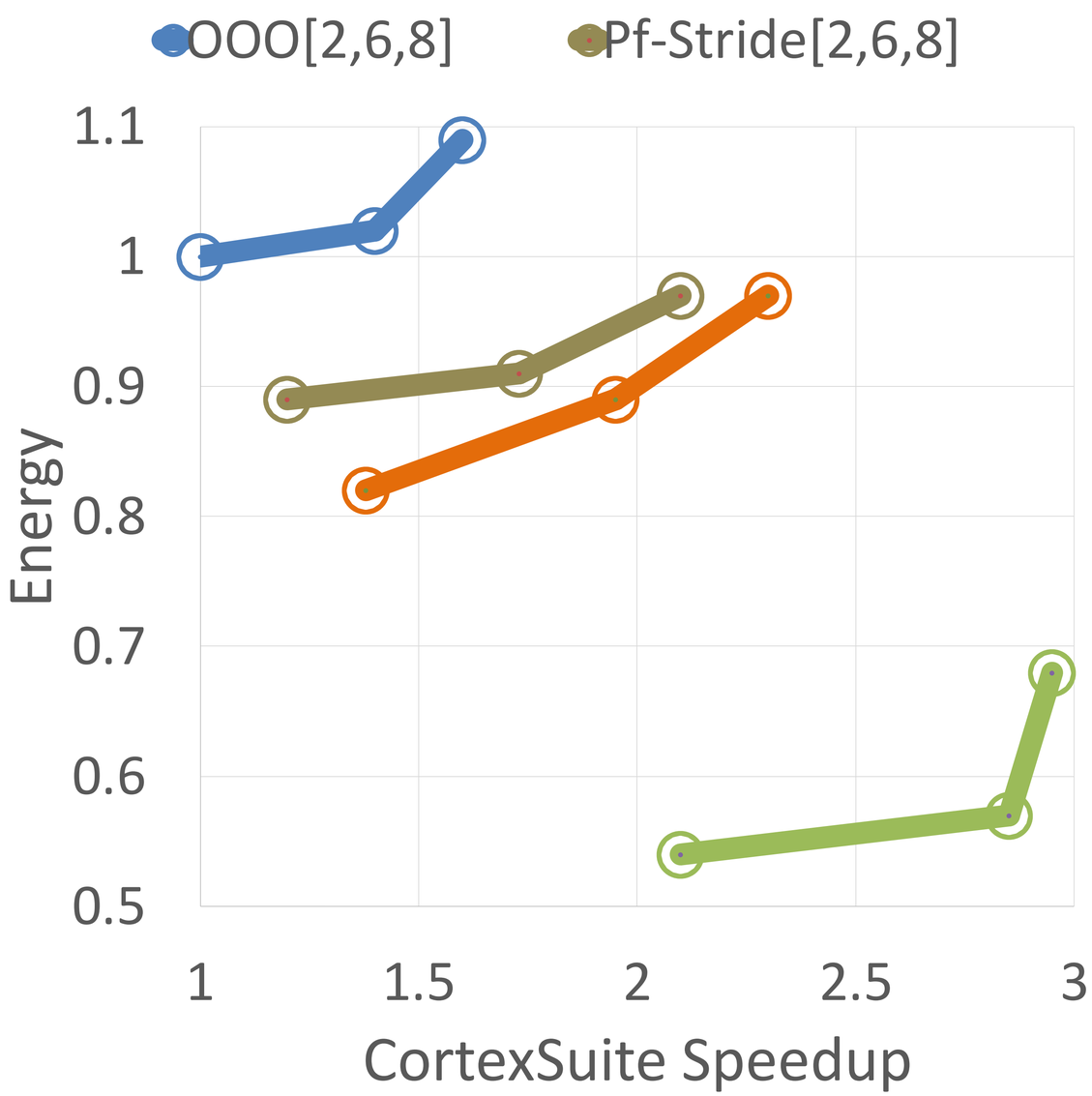
Stream is just a prefetch

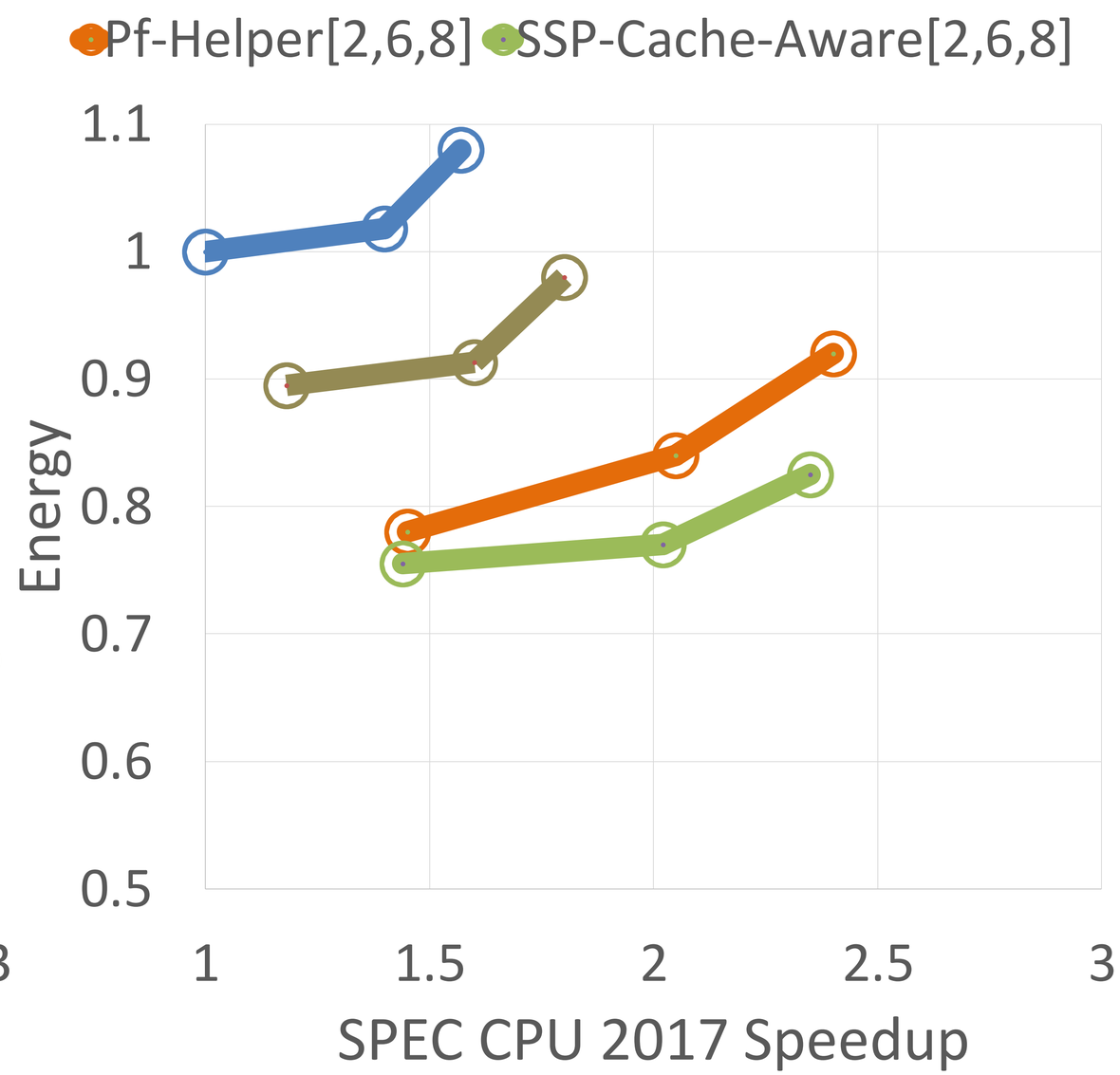
Stream instructions re decoupled

Stream-based Cache Bypassing

Ideal Prefetch 1K instructions ahead

# Results – Design Space Interaction

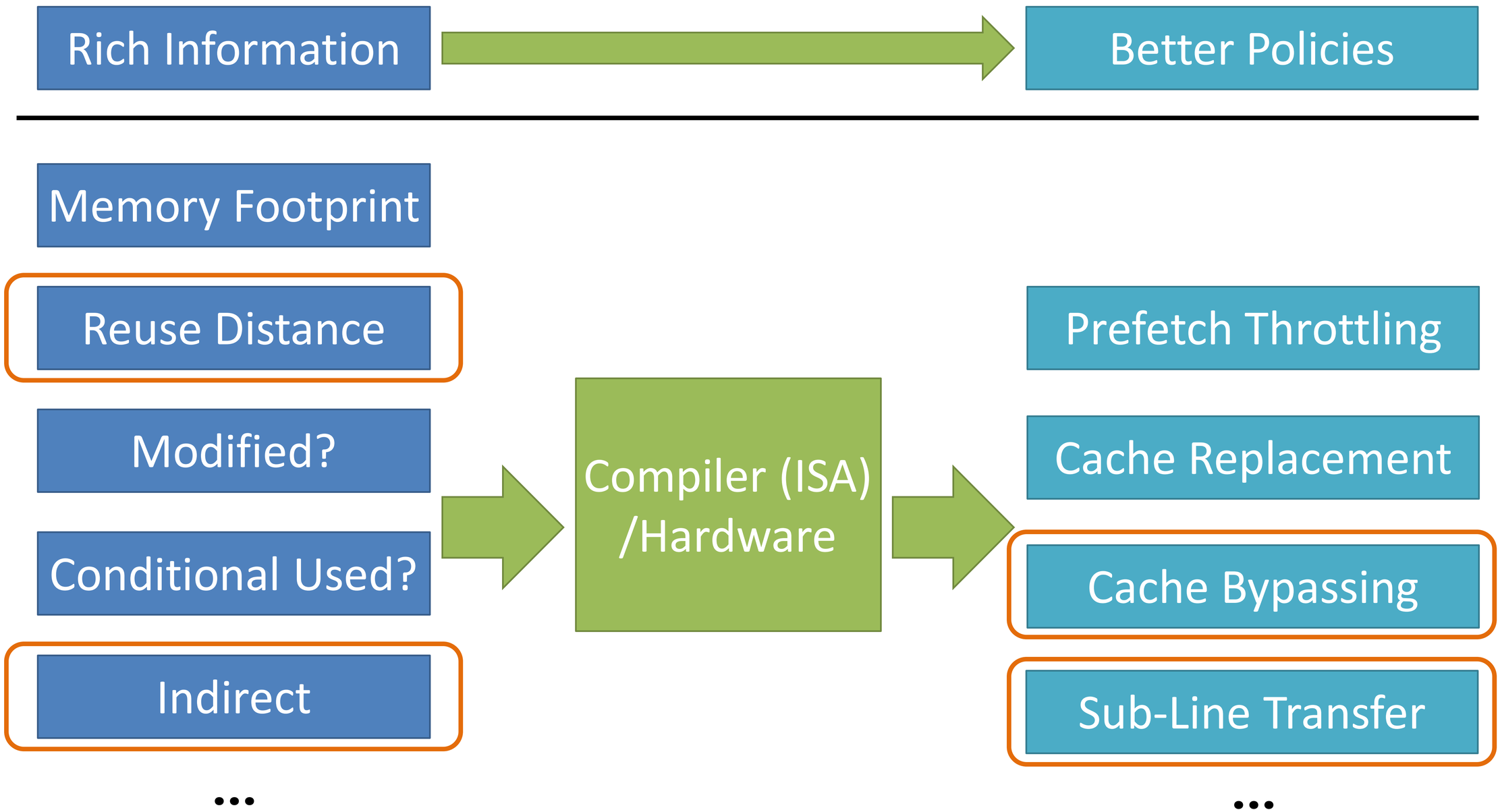


# Conclusion

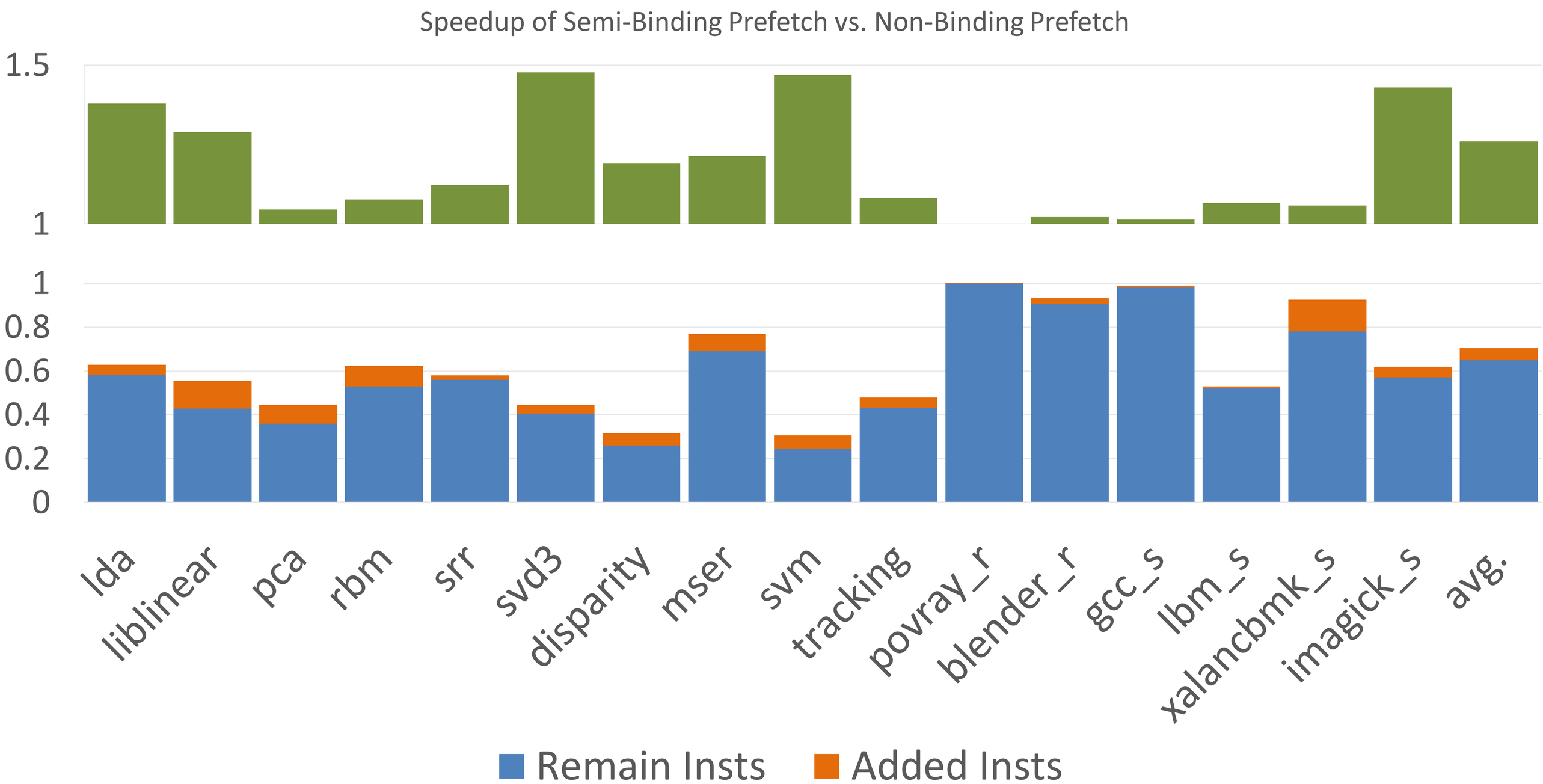
- Stream as a new memory abstraction in ISA.
  - ISA/Microarchitecture extension.
  - Stream-aware cache bypassing.
- New paradigm of memory specialization.
  - New direction for improving cache architectures.
  - Combine memory and computation specialization.

# Stream-Aware Policies



# Backup Slides

# Results – Semi-Binding Prefetching



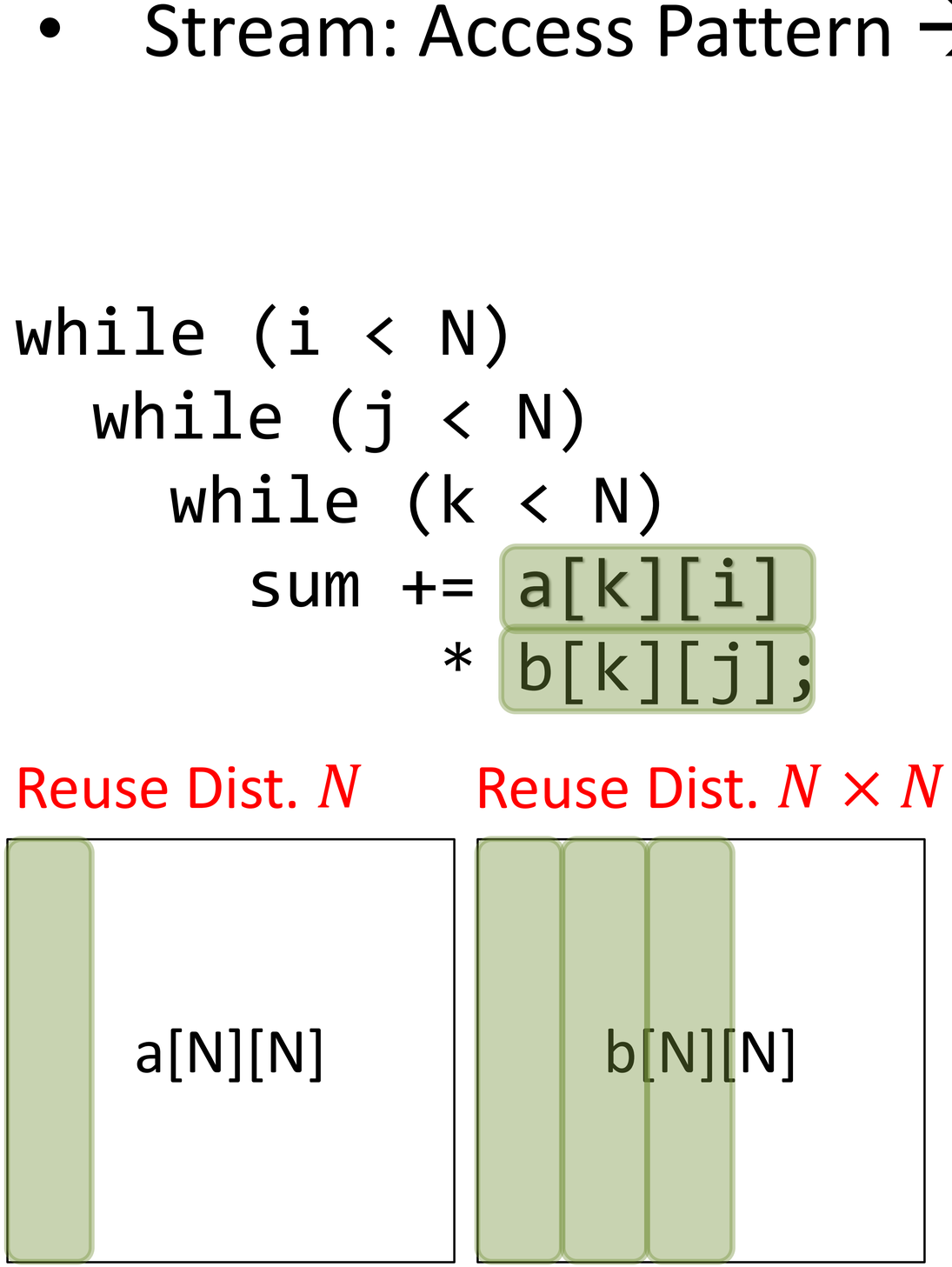
# Related Work

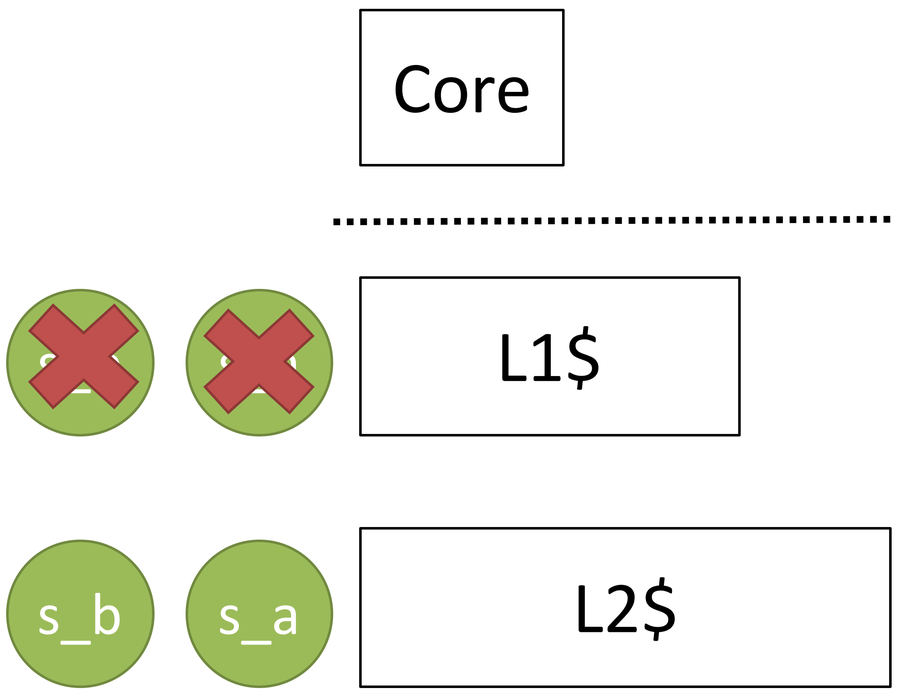
- Decouple access execute.
  - Outrider [ISCA'11], DeSC [MICRO'15], etc.
  - Ours: New ISA abstraction for the access engine.
- Prefetching.
  - Stride, IMP [MICRO'15], etc.
  - Ours: Explicit access pattern in ISA.
- Cache bypassing policy.
  - Counter-based [ICCD'05], LLC bypassing [ISCA'11], etc.
  - Ours: Incorporate static stream information.

# Stream-Aware Policies – Cache Bypass

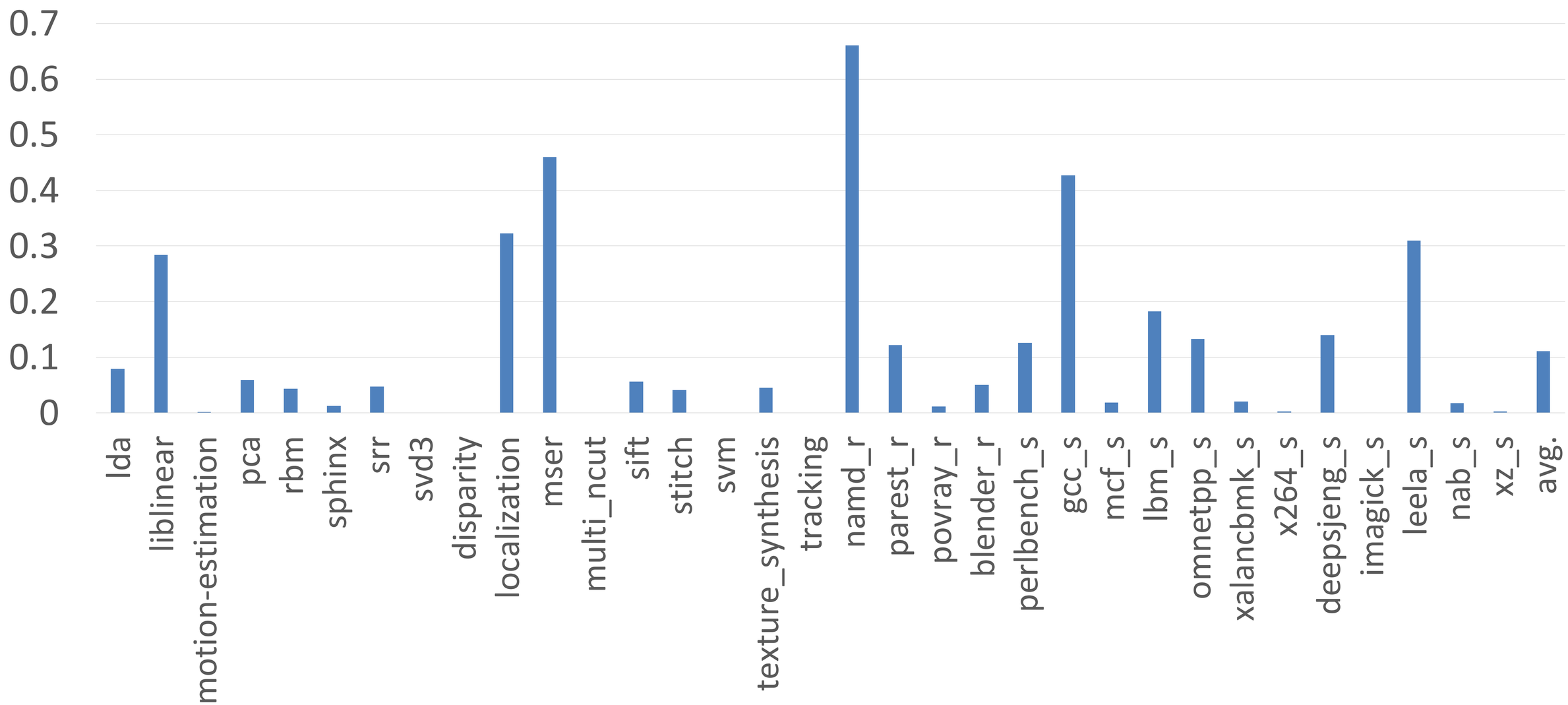
- Stream: Access Pattern → Precise Memory Footprint.

```
while (i < N)
  while (j < N)
    while (k < N)
      sum += a[k][i]
           * b[k][j];
```



# Results – Unused Stream Requests



# Stream ISA Extension – Compiler in LLVM

- Identify stream candidates.
  - Search backwards on operands of candidates.
  - Detect pattern and dependency between candidates.

Identify Stream Candidates

- Select stream candidates.
  - Check decouplable pattern.
  - Limited number of pseudo-registers.

Select Stream Candidates

- Code generation.
  - Stream configuration, e.g. type, parameters, etc.
  - Transform the program.

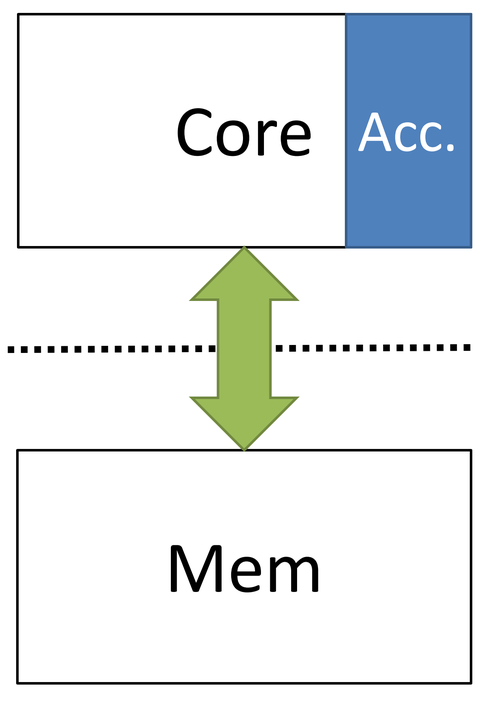
Code Generation

# Stream: A New ISA Memory Abstraction

Higher level of abstraction embeds richer semantic info.

Opens up new opportunities.

1.37x speedup over a hardware stride prefetch.




(Address, Value):
(0x400, 1)
(0x404, 2)
(0x408, 3)
…

Stream in ISA

- What stream patterns to support?
- How to deal with control flow?
- ISA microarchitecture co-design?
- …

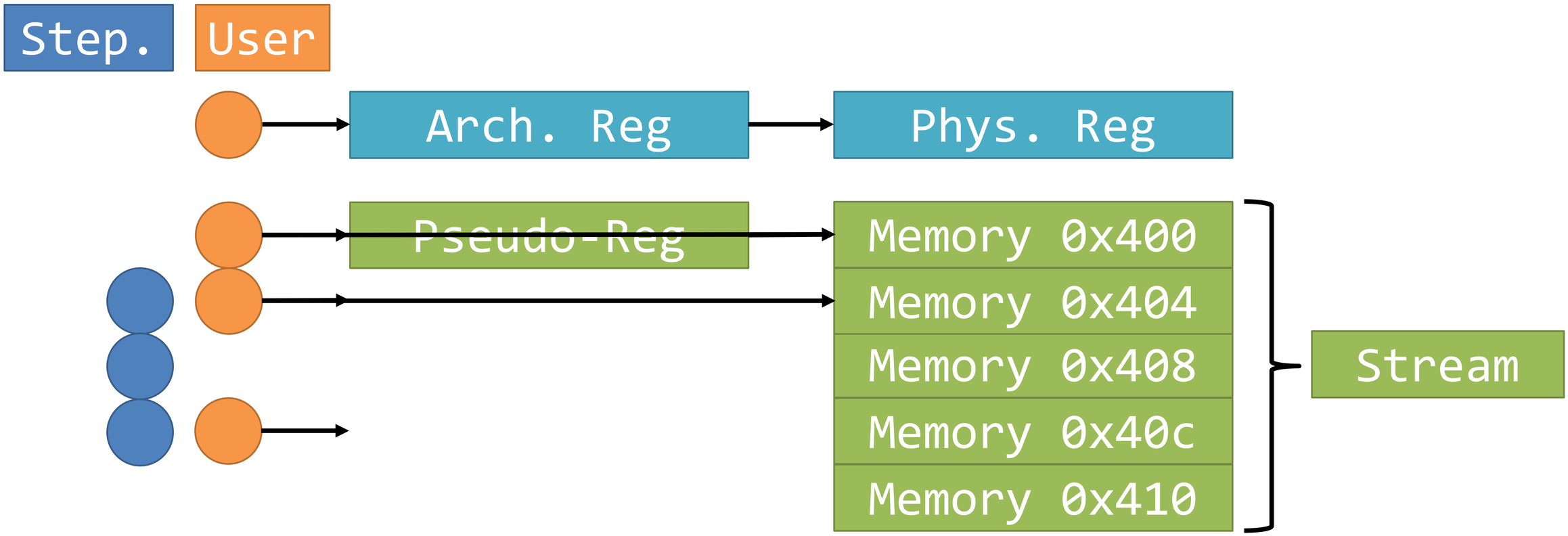
- Prefetch with control flow.
- Decouple address computation.
- Better cache policies.
- …

# Insight: Stream as a new ISA abstraction

- Higher level abstraction than single memory requests.
  - Access pattern, memory footprint, etc.
  - More accurate & efficient than hardware approaches, e.g. hardware prefetching.
- Rich information brings new opportunities:
  - More accurate & timely prefetching.
  - Removing redundant work, e.g. address computation.
  - Better cache policies.

# Stream ISA Extension – Basic Principles

- Stream: A decoupled sequence of values/addresses.
- Usage of stream data:
  - Referred by a pseudo-register.
  - Explicitly stepped by a stream_step inst.



# Stream ISA Extension – Nested Loop

- Indirect Stream.
- Decoupled from control flow.
- Longer streams: Configure stream in outer loop.
- Coalesce stream.

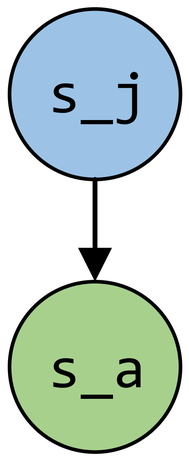
**Original C Code**

```
int i = 0;
while (i < M) {
  int j = 0;
  while (j < N) {
    sum += a[i][j];
    j++;
  }
  i++;
}
```

**Stream Decoupled Pseudo Code**

```
int i = 0;
stream_cfg(s_j, s_a);
while (i < M) {
  while (s_j < N) {
    sum += s_a;
    stream_step(s_j);
  }
  stream_step(s_j);
  i++;
}
stream_end(s_j, s_a);
```

**Stream Dependence Graph**



# Stream ISA Extension – Coalesce Stream

- Indirect Stream.
- Decoupled from control flow.
- Longer streams (nested loop).
- Coalesce stream.

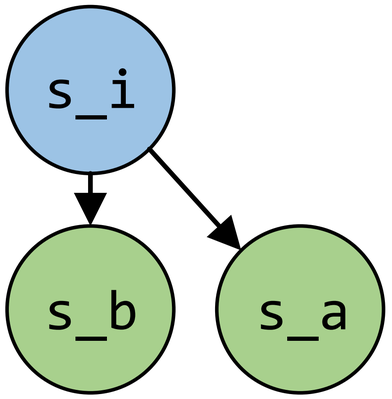
**Original C Code**

```
int i = 0;
while (i < N) {
  b[i] =   a[i].x
         + a[i].y;
  i++;
}
```

**Stream Decoupled Pseudo Code**

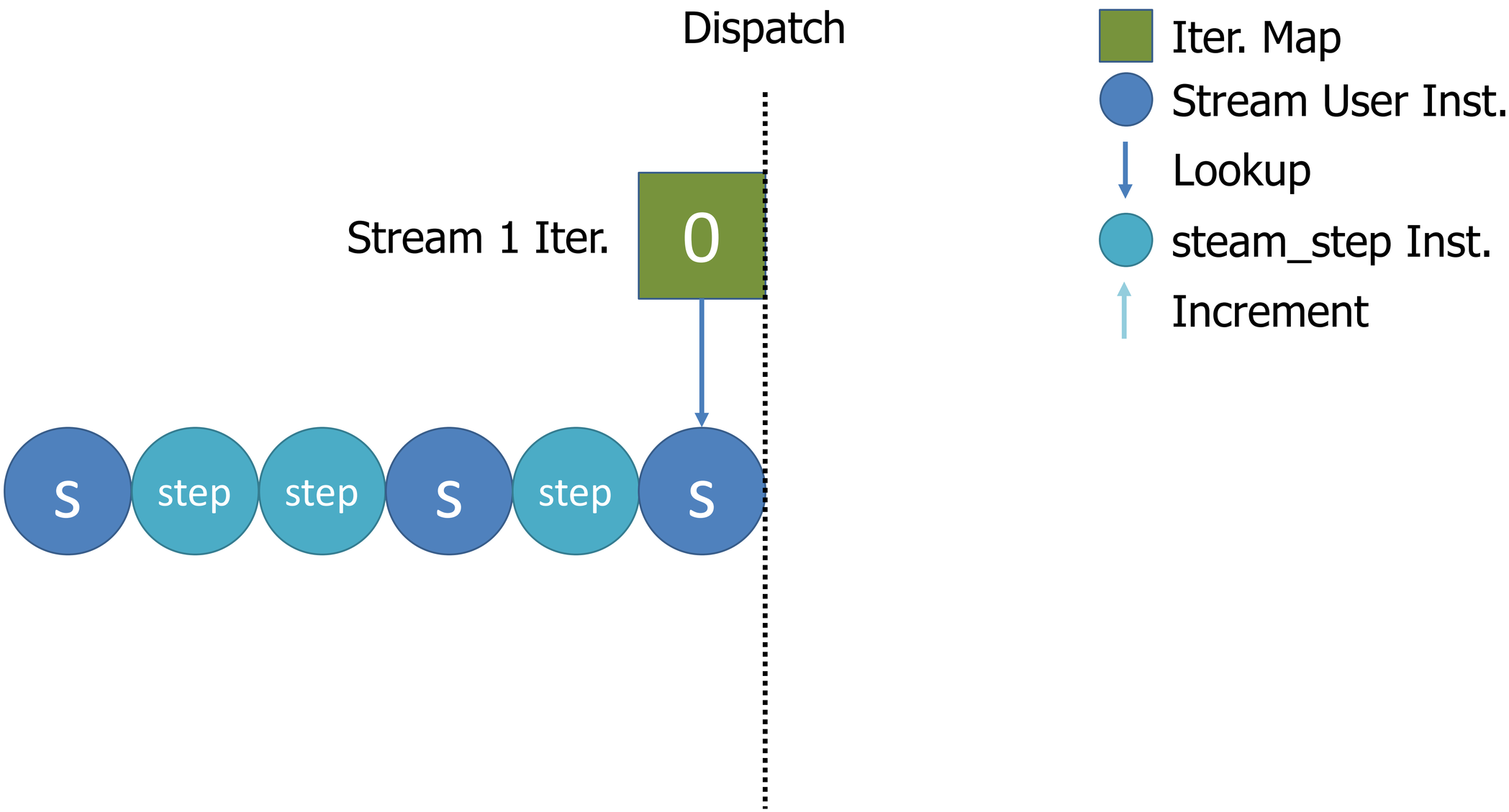
```
stream_cfg(s_i, s_a, s_b);
while (s_i < N) {
  s_b = s_a.x + s_a.y;
  stream_step(s_i);
}
stream_end(s_i, s_a, s_b);
```

**Stream Dependence Graph**



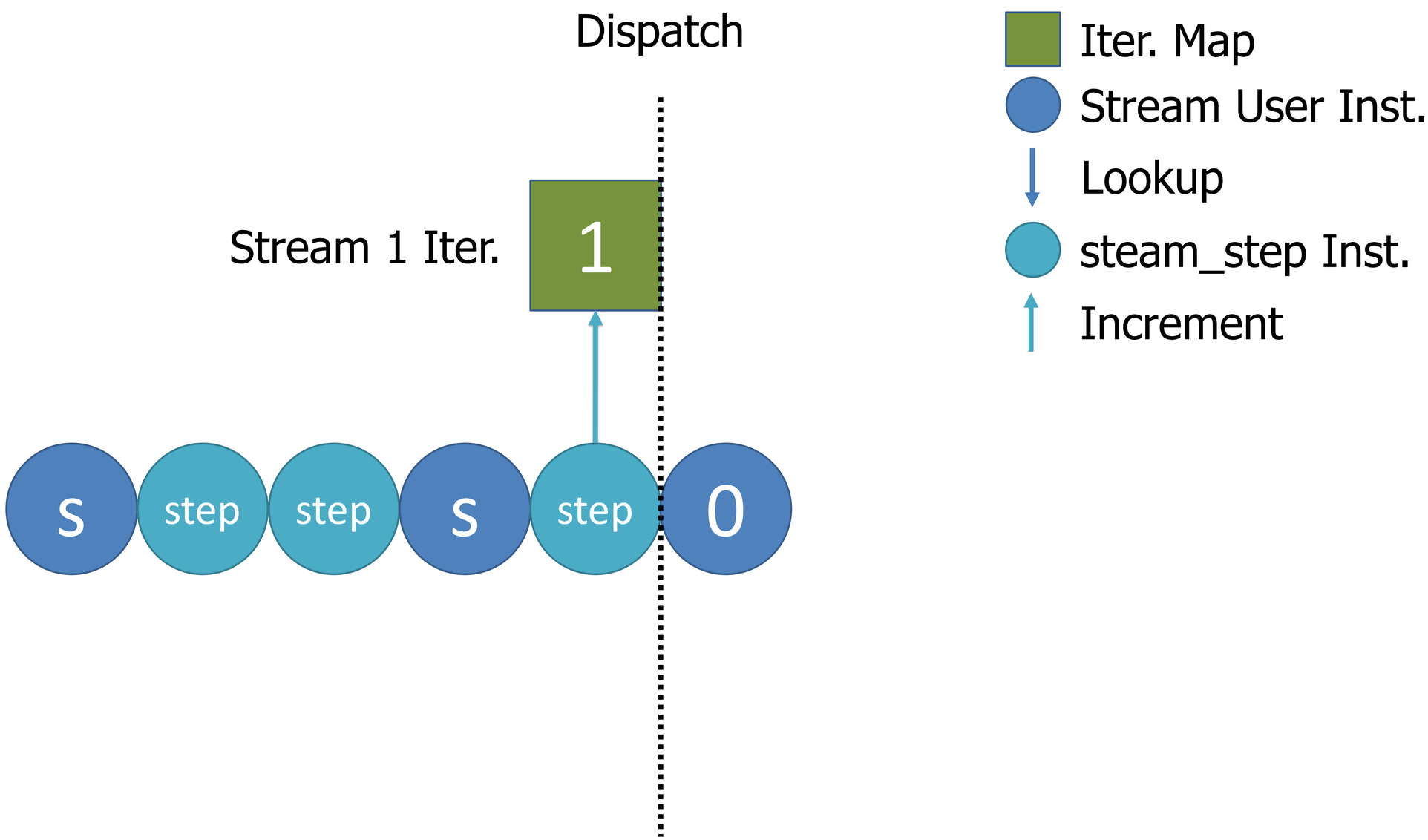
# Microarchitecture – Core View

Goal: What is the current iteration of a stream?



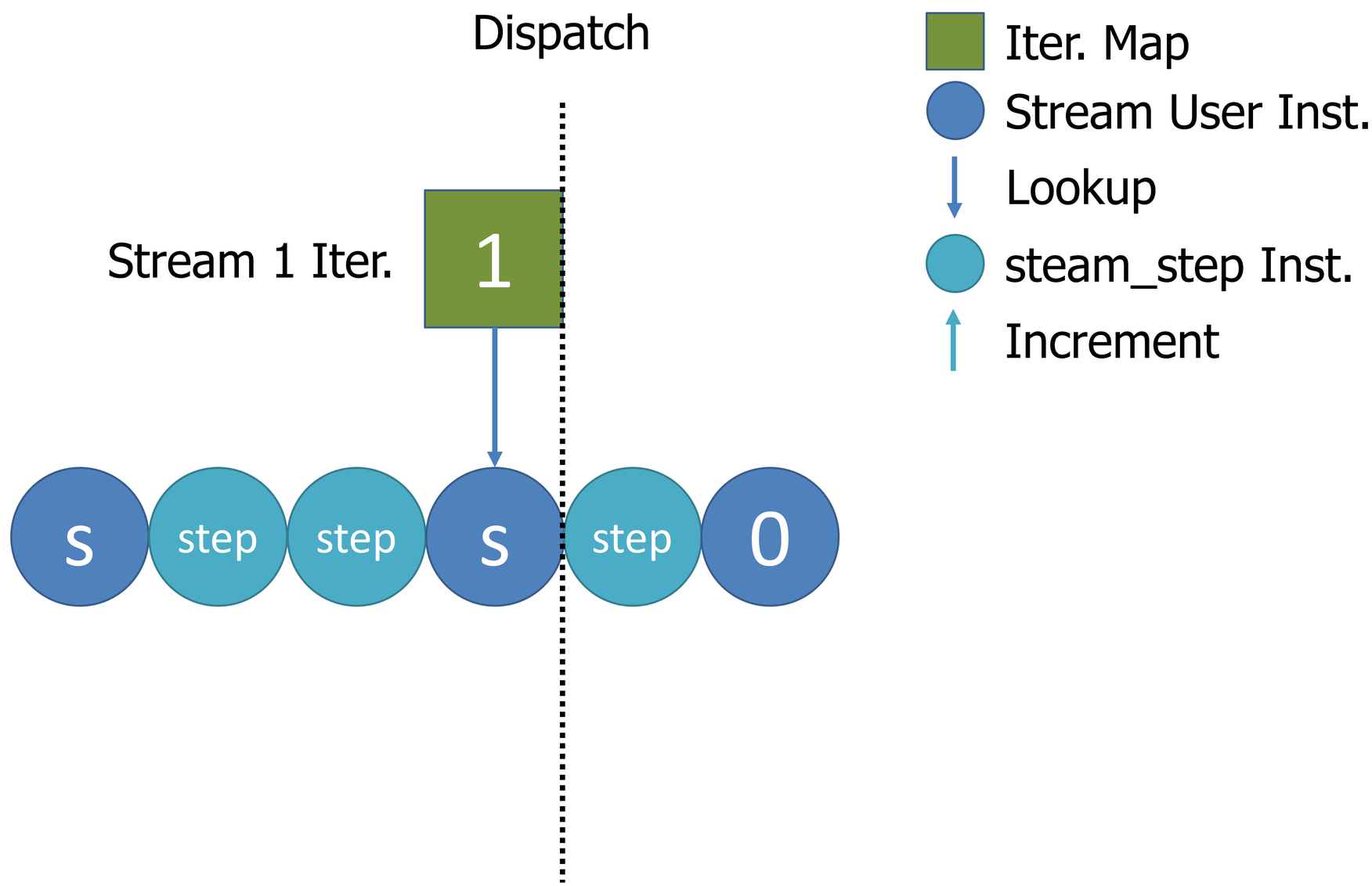
# Microarchitecture – Core View

Goal: What is the current iteration of a stream?



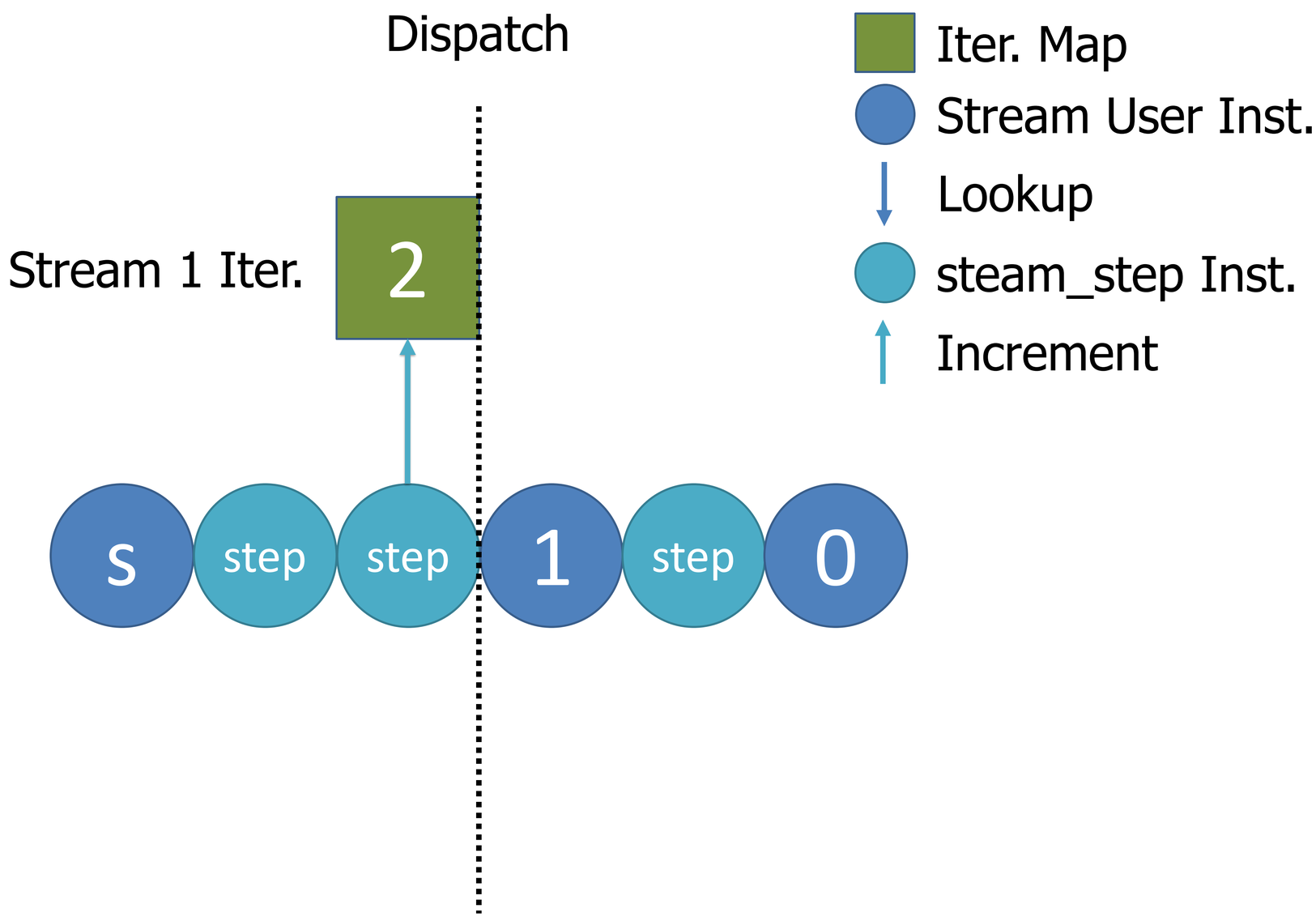
# Microarchitecture – Core View

Goal: What is the current iteration of a stream?



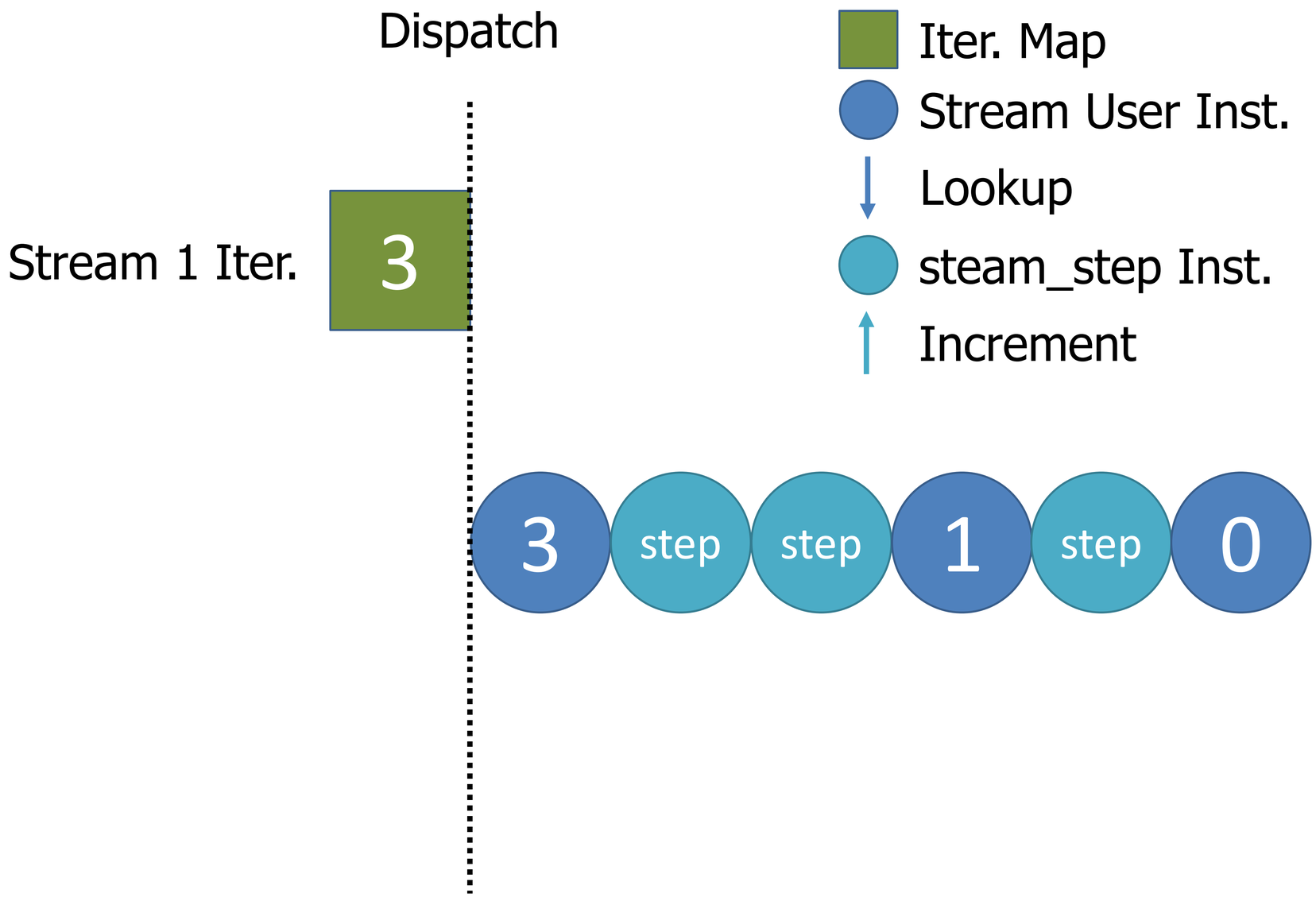
# Microarchitecture – Core View

Goal: What is the current iteration of a stream?



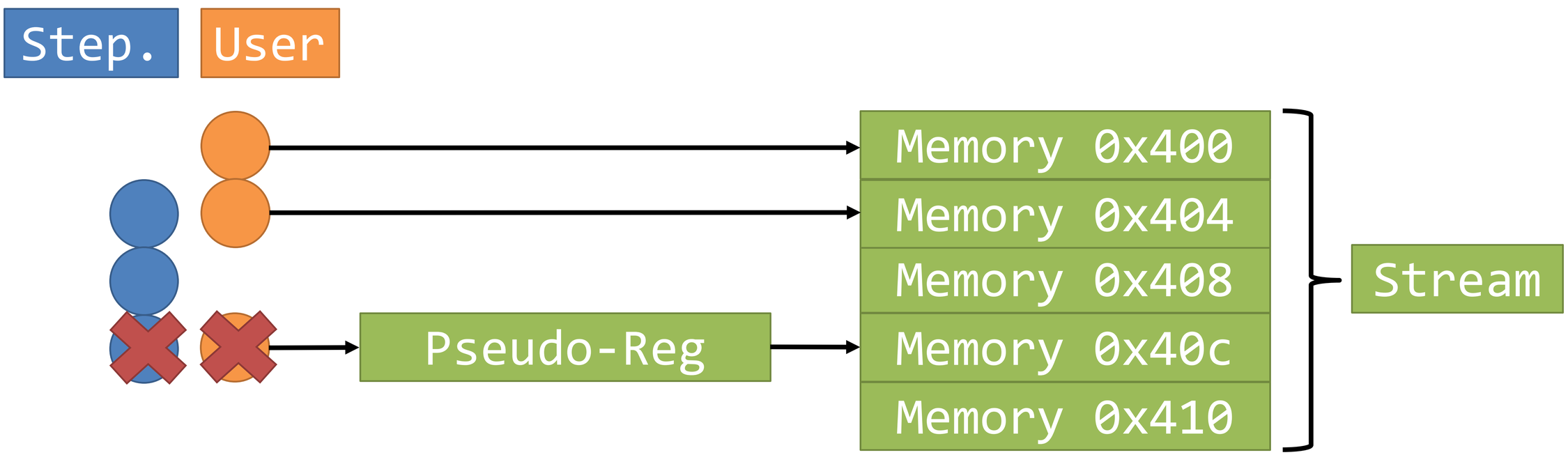
# Microarchitecture – Core View

Goal: What is the current iteration of a stream?



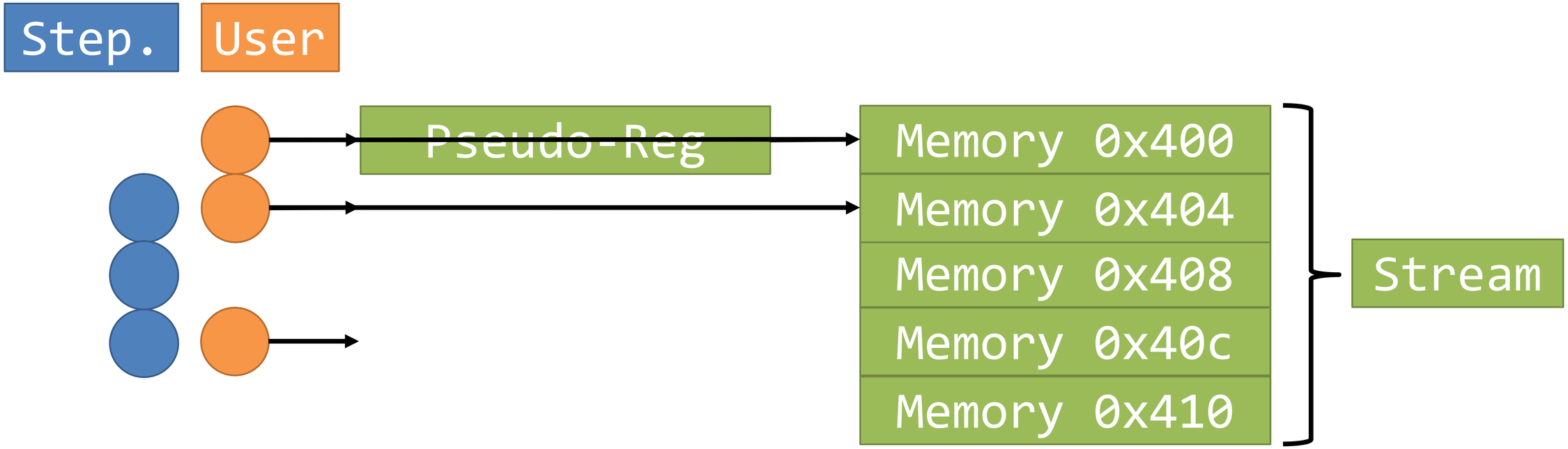
# Microarchitecture – Revert stream_step

- Control misspeculated stream_step.
  - Decrement the iteration map.
  - No need to flush the FIFO and re-fetch data (decoupled) !



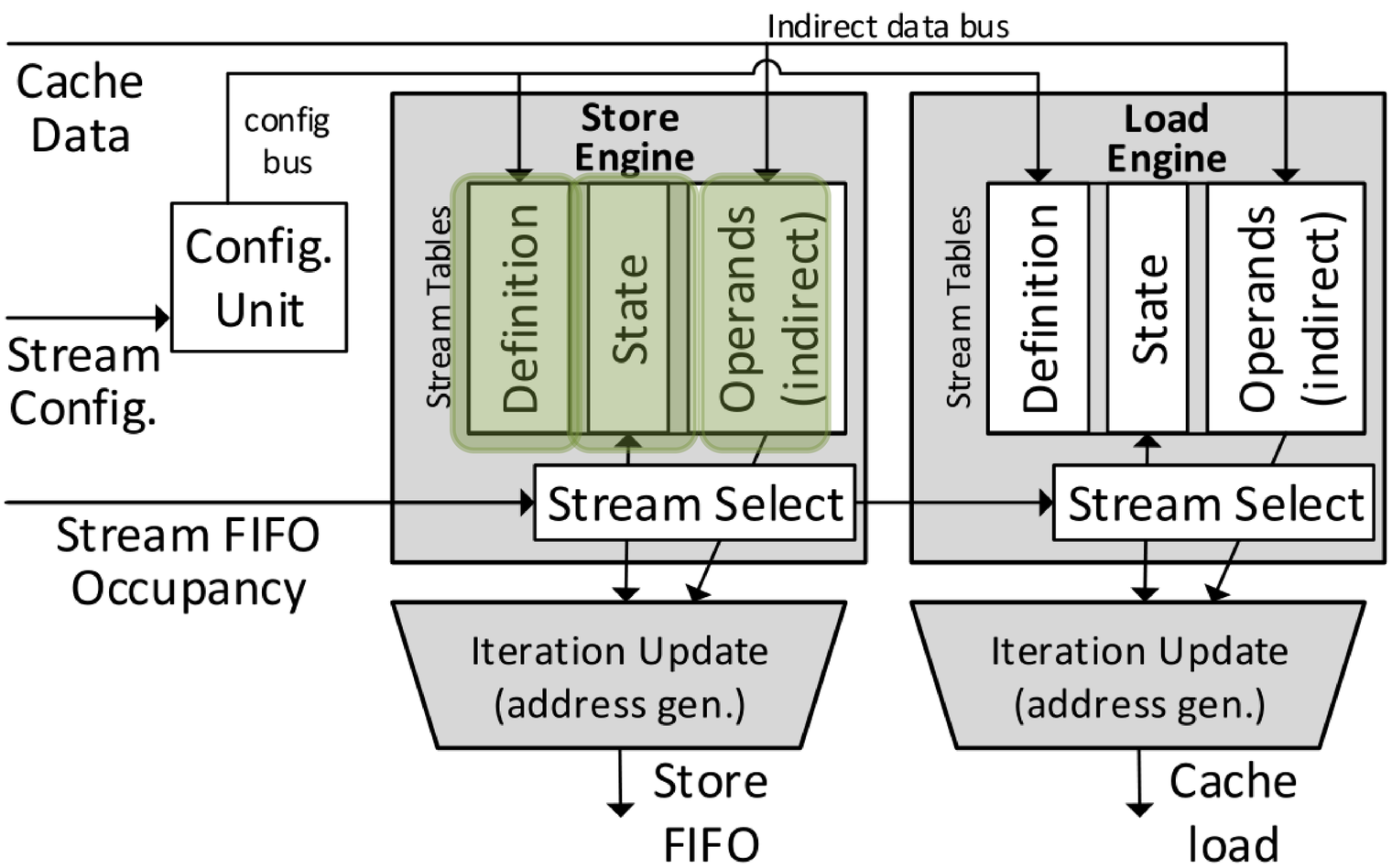
# Stream ISA Extension – Stepping

- Explicitly stepping with stream_step instruction.



# Microarchitecture – Stream View

Managed by the Stream Engine.




Definition:
Address pattern parameters,
Initial address, stride, etc.

State:
Current iteration.

Operands:
Index from cache.
For indirect streams.

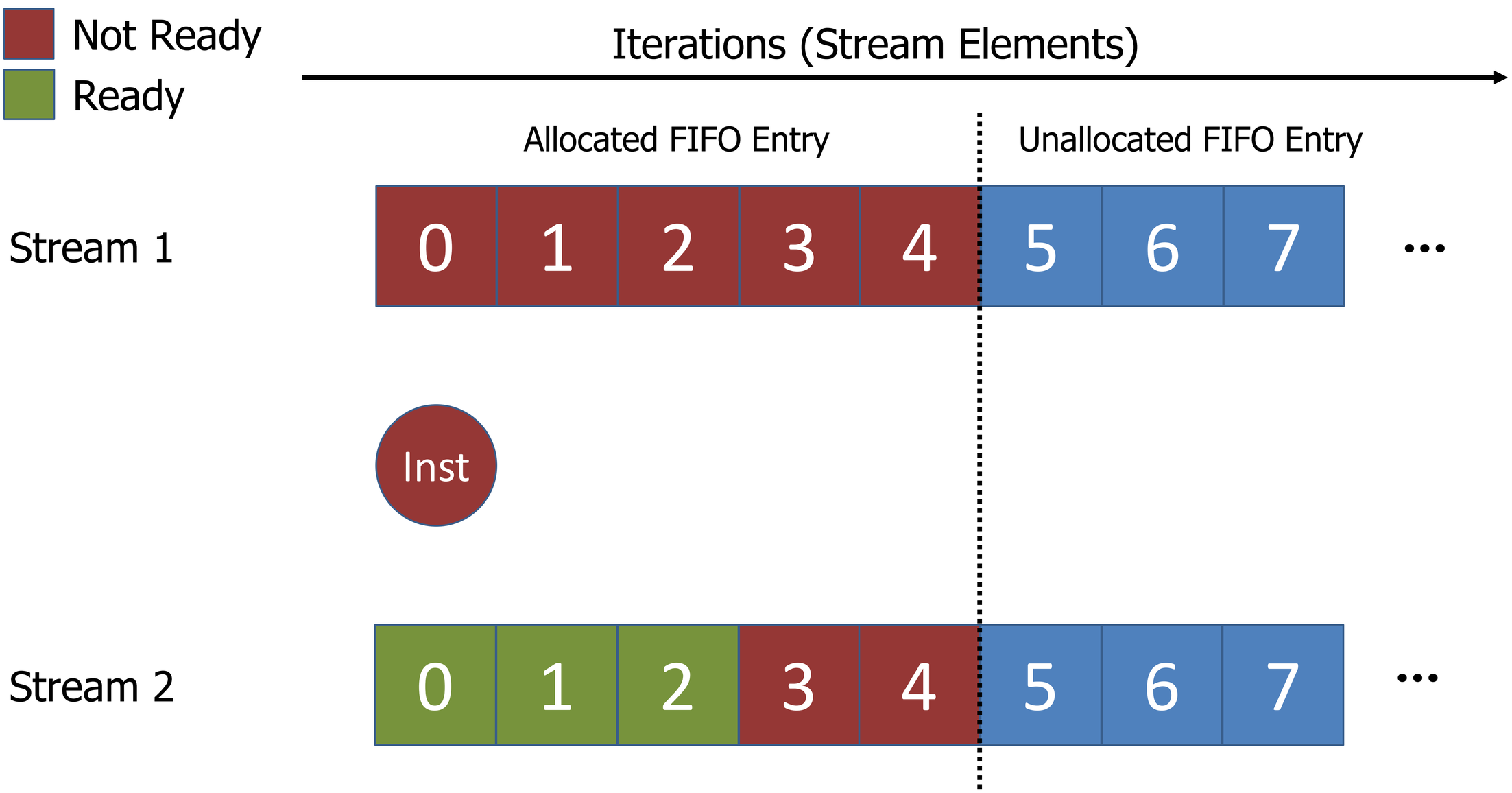
# Microarchitecture – Aliasing Streams

- Leverage the LSQ to handle aliasing.
  - Extend the LSQ with PEB to record prefetching stream elements.
  - Insert into the LSQ for the user instruction that semantically triggers the stream access.
  - Revert and restart at memory order violation.

# Stream-Aware Policies - Throttling

## Goal: Prefetch just in time.

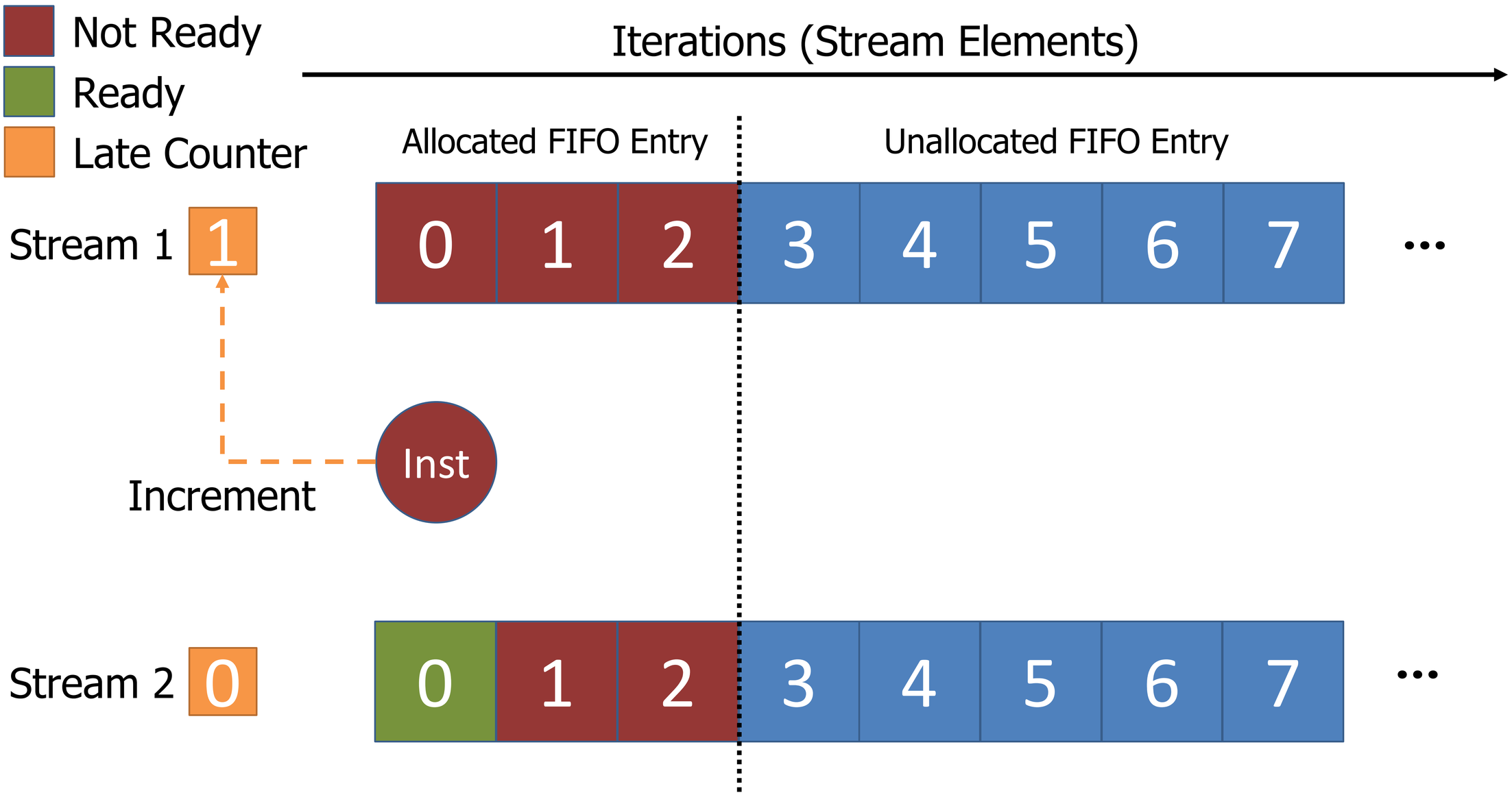
Evenly distribute FIFO among streams leads to untimely prefetching.



# Stream-Aware Policies - Throttling

## A “Late” counter per stream.

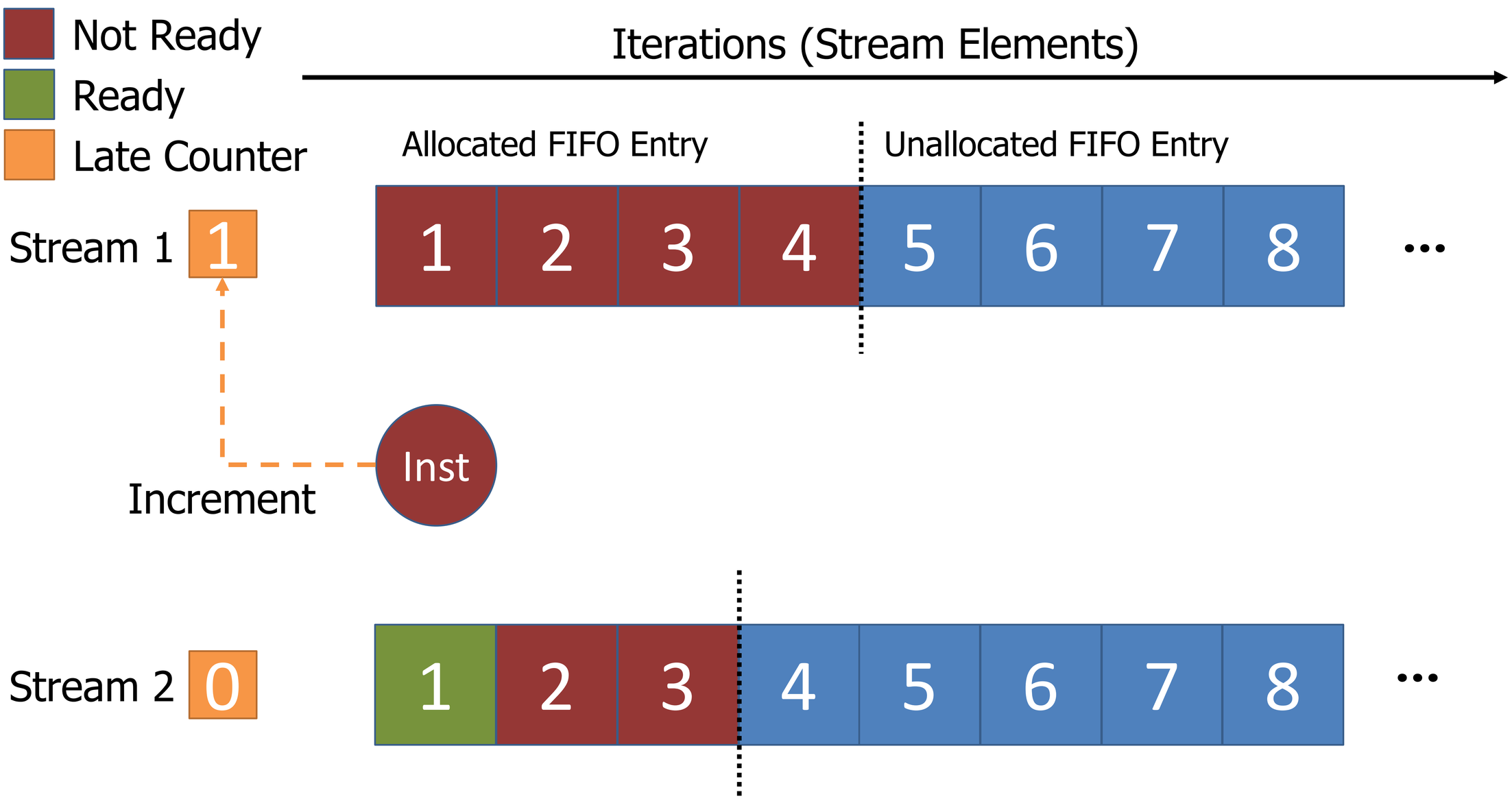
Increased every time a user instruction blocked by the stream.



# Stream-Aware Policies - Throttling

## A “Late” counter per stream.

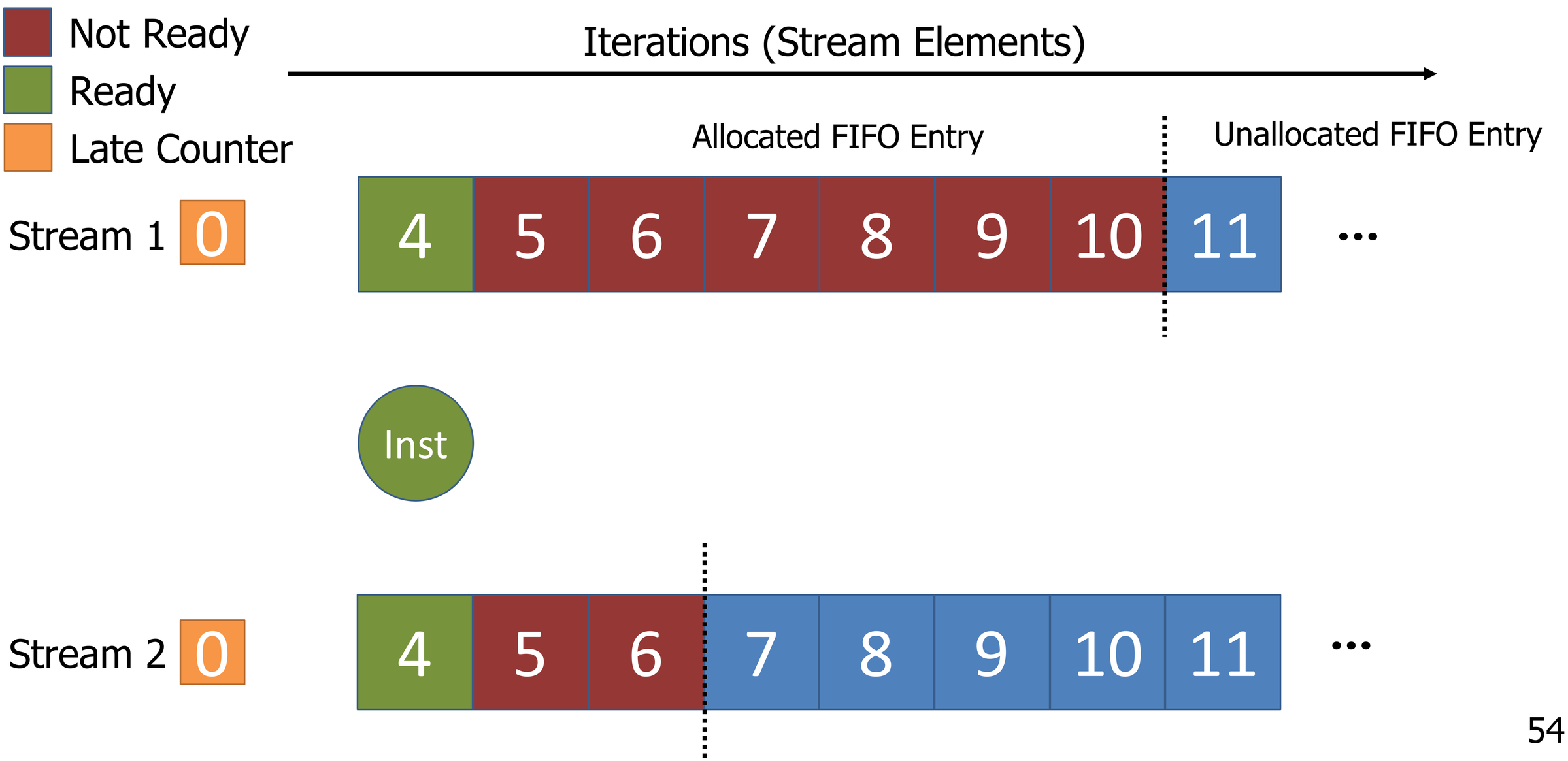
Allocate more FIFO entries to the stream with high “Late” count.



# Stream-Aware Policies – Throttling

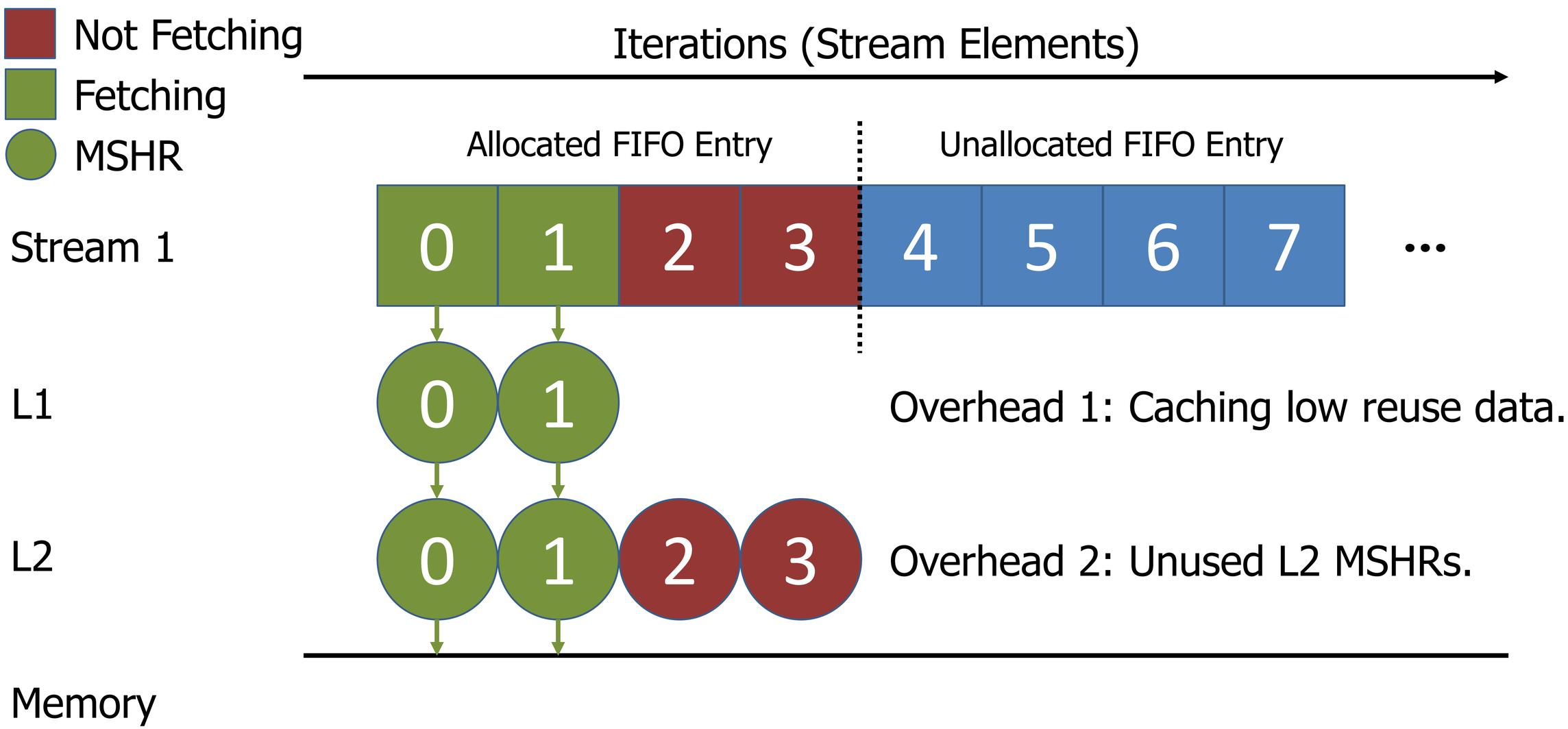
## A “Late” counter per stream.

After some iterations, stream 1 has enough entries to hide the latency.



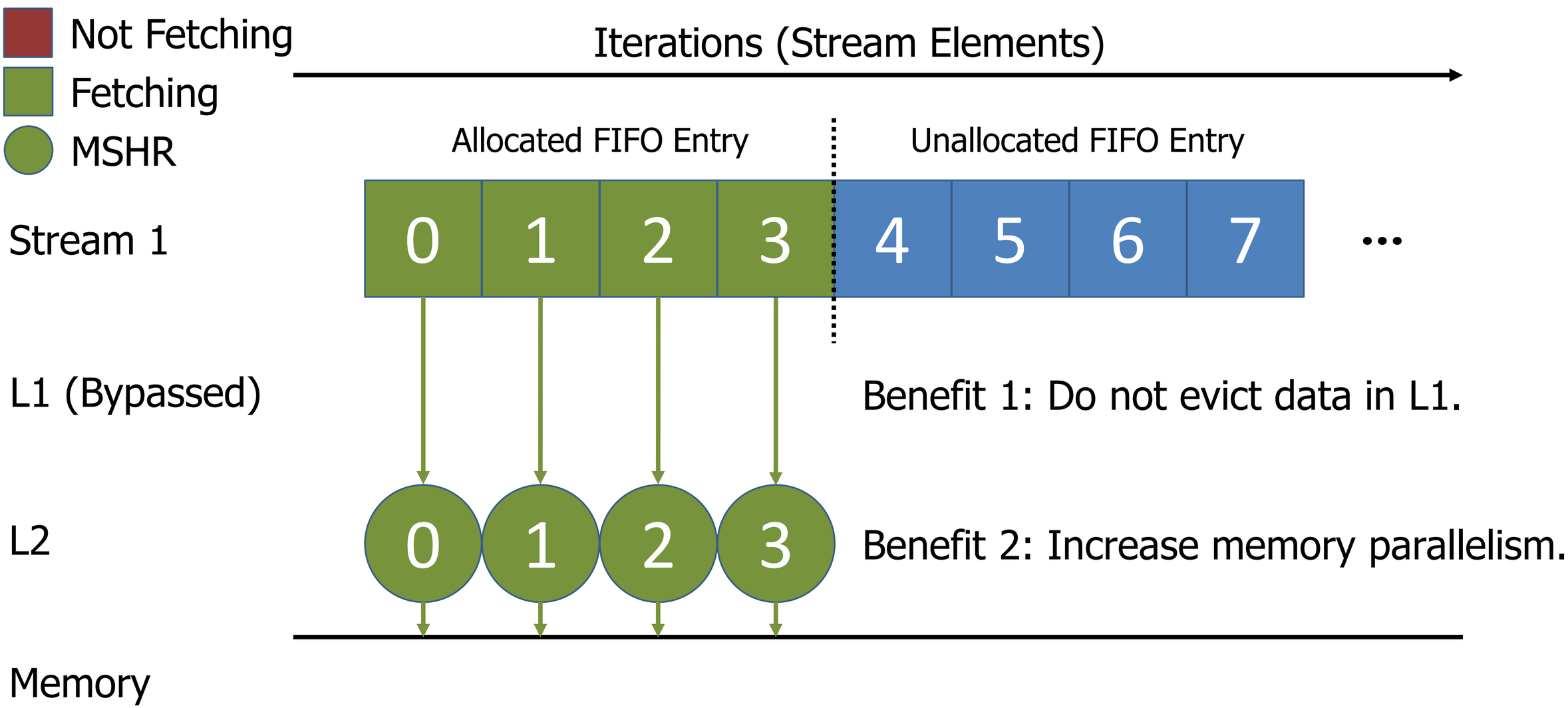
# Stream-Aware Policies – Cache Bypass

- Stream: Access Pattern -> Precise Memory Footprint.
  - Precise estimate the memory footprint when configuring streams.
  - Bypass.



# Stream-Aware Policies – Cache Bypass

- Stream: Access Pattern -> Precise Memory Footprint.
  - Precise estimate the memory footprint when configuring streams.
  - Bypass streams with low reuse.



# Stream-Aware Policies – Cache Bypass

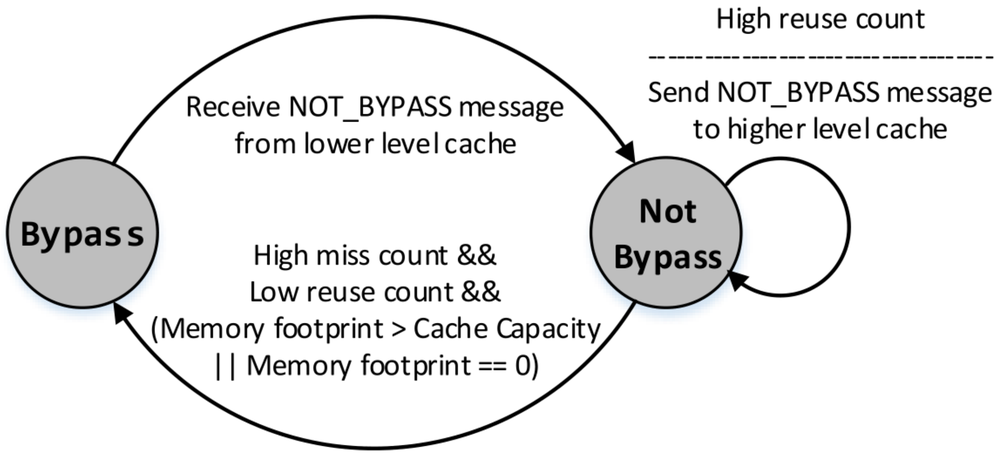
## Goal: Correctly identify streams with low reuse.

A stream table to combine static & dynamic information.

Tags are augmented with the stream id.

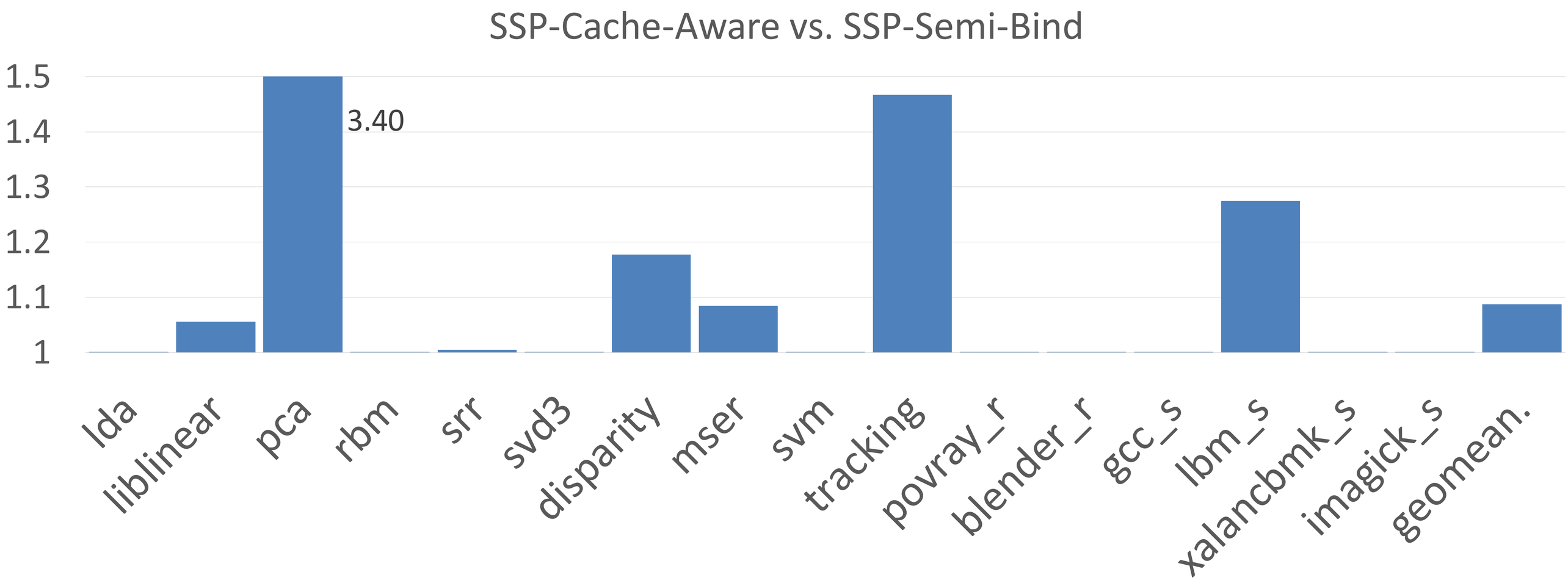
A FSM to bypass streams with:

- ✓ High miss count &&
- ✓ Low reuse count &&
- ✓ (Large footprint || No footprint info)




| Field | Description | Field | Description |
|---|---|---|---|
| `sid` | Stream id | `miss` | # cache misses |
| `footprint` | Est. mem. footprint | `reuse` | # cache reuses |
| `request` | # stream requests | `bypass` | Whether to bypass |

# Results – Stream-Aware Cache Bypassing




Non-Binding Prefetch

Semi-Binding Prefetch

Stream-Aware Cache

## Original

```
...
.LBB0_1:
    movsxd  rdx, r8d
    mov     edi, [4*rdx + a]
    movsxd  rcx, eax
    cmp     edi, [4*rcx + b]
    jge     .LBB0_3
    add     esi, [4*rdx + c]
    add     edx, 1
    mov     r8d, edx
    cmp     r8d, 1023
    jle     .LBB0_5
    jmp     .LBB0_6
.LBB0_3:
    add     eax, 1
    cmp     r8d, 1023
    jg      .LBB0_6
.LBB0_5:
    cmp     eax, 1024
    jl      .LBB0_1
    ...
```

**Register to Stream Mapping**

```
r8d -> iv stream s_i
eax -> iv stream s_j
[4*rdx + a] -> memory stream s_a
[4*rcx + b] -> memory stream s_b
[4*rdx + c] -> memory stream s_c
```

## Stream Specialized

```
...
    s_cfg
.LBB0_1:
    cmp     s2, s3
    jge     .LBB0_3
    add     esi, s4
    s_step  s0
    cmp     s0, 1023
    jle     .LBB0_5
    jmp     .LBB0_6
.LBB0_3:
    s_step  s1
    cmp     s1, 1023
    jg      .LBB0_6
.LBB0_5:
    cmp     s1, 1024
    jl      .LBB0_1
.LBB0_6:
    s_end
    ...
```

**Stream to Pseudo Register Mapping**

```
Stream s_i -> s0
Stream s_j -> s1
Stream s_a -> s2
Stream s_b -> s3
Stream s_c -> s4
```