Execution Dependence Extension (EDE)

ISA Support for Eliminating Fences

Thomas Shull (Oracle Labs); Ilias Vougioukas, Nikos Nikoleris, Wendy Elsasser (Arm Research); Josep Torrellas (University of Illinois in Urbana-Champaign)

Int. Conf. on Computer Architecture (ISCA), June 2021

i-acoma group

Executive Summary

- Propose new instructions which explicitly state their fine-grain ordering requirements
  - Net impact: fewer fences in applications → improved performance
- Present two practical hardware implementations that efficiently enforce ordering requirements
  - One in issue queue, another in writeback buffer
- Demonstrate effectiveness of new instructions in Non-Volatile Memory (NVM) applications
  - Average workload speedups of 18% (IQ) and 26% (WB)

Out-of-Order (OoO) Execution

- Processors execute instructions out of order
  - Improves Performance
    - ↑ Parallelism
    - ↓ Stalling
- Must still honor intra-thread data dependences
  - Memory Dependence: access same memory address
  - Register Dependence: register read after write
  - Reorder data dependence → indecipherable result

The Fence Instruction

- Execution Behavior:
  - Guarantees instructions before fence complete before instructions after fence execute
- Example Use Cases:
  - Multi-threaded applications: prevent data races
  - Non-Volatile Memory (NVM) applications: enforce write ordering
- Drawbacks:
  - Hurt performance
  - *Coarse grain*: not suited for describing fine-grain ordering requirements

Fence Example: Peterson's Algorithm

Variables

```
bool p0_flag;
bool p1_flag;
int turn;
```

Processor 0

Required Ordering

```
p0_flag = true;
turn = 1;
FENCE
while (p1_flag && turn == 1) {
   // busy wait
}

// critical section
                ...
```

Processor 1

Required Ordering

```
p1_flag = true;
turn = 0;
FENCE
while (p0_flag && turn == 1) {
   // busy wait
}

// critical section
                ...
```

Solution: Express *Execution Dependences*

Execution Dependence Definition:

- A required ordering between two individual instructions, where the effects of the sink instruction cannot be observed until the source instruction is complete

- For ordering requirements not expressed through register or memory dependences

- New Instructions: Execution Dependence Extension (EDE)
  - Instructions which explicitly define execution dependences

- Processor must honor them
  - Net Effect: need fewer fences

EDE Use Case: Peterson's Algorithm

Variables

```
bool p0_flag;
bool p1_flag;
int turn;
```

Processor 0

Execution Dependence

```
p0_flag = true;
turn = 1;
FENCE
while (p1_flag && turn == 1) {
   // busy wait
}

// critical section
                ...
```

Processor 1

Execution Dependence

```
p1_flag = true;
turn = 0;
FENCE
while (p0_flag && turn == 1) {
   // busy wait
}

// critical section
                ...
```

EDE Definitions

- Dependence Producer – source of execution dependence
- Dependence Consumer – sink of execution dependence

- EDE dependence combinations
  1. Dependence Producer only
  2. Dependence Consumer only
  3. Both Dependence Consumer & Producer
  4. No dependence

- Execution Dependence Key (EDK): Link from execution dependence producer to consumer(s)

- Execution Dependence Map (EDM): Map of active dependence producers
  - EDK = Map [EDK → In-flight instruction]

Decoding Execution Dependences

Instruction Format:
INS (EDK_{Def}, EDK_{Use}), <original operands>

Example Instruction:

INS (#3, #2) //Assume instruction's in-flight tag is T4

Steps:

1. Read EDM[EDK_{Use}]
2. Set EDM[EDK_{Def}]

Execution Dependence:
T12 -> T4

Execution Dependency Map (EDM)

EDK #	In-Flight Tag
1	T33
2	T12
3	T23
4	--
5	T41
--	--
15	--

Decoding Execution Dependences

Instruction Format:
INS (EDK_{Def}, EDK_{Use}), <original operands>

Example Instruction:

INS (#3, #2) //Assume instruction's in-flight tag is T4

Steps:

1. Read EDM[EDK_{Use}]
2. Set EDM[EDK_{Def}]

Execution Dependence:
T12 -> T4

Execution Dependency Map (EDM)

EDK #	In-Flight Tag
1	T33
2	T12
3	T4
4	--
5	T41
--	--
15	--

Decoding Execution Dependences

Instruction Format:
INS (EDK_{Def}, EDK_{Use}), <original operands>

EDK #0 is *Zero Key* and is ignored

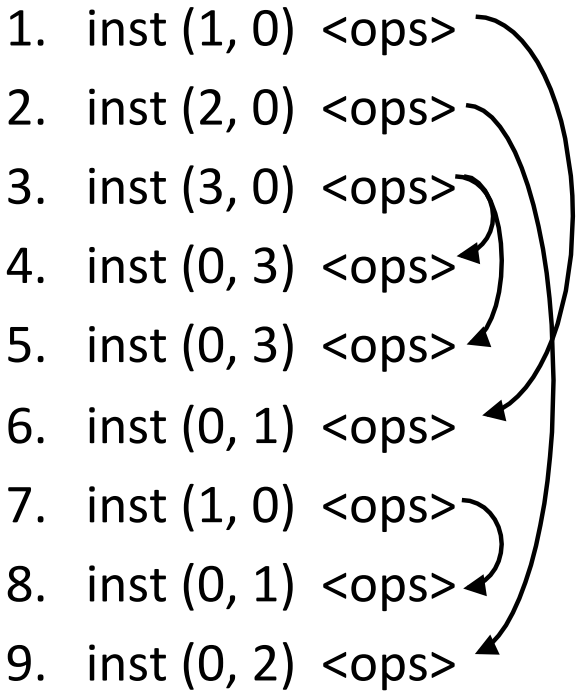
Steps:

1. Read EDM[EDK_{Use}]
2. Set EDM[EDK_{Def}]

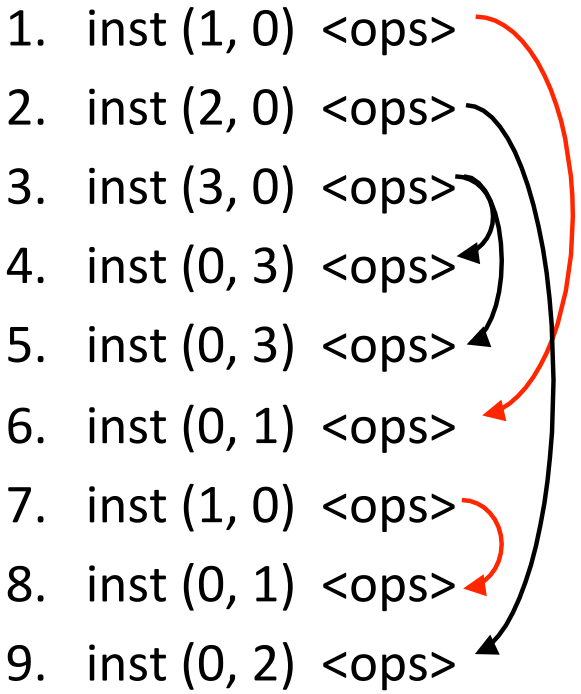
Execution Dependency Map (EDM)

EDK #	In-Flight Tag
1	T33
2	T12
3	**T4**
4	--
5	T41
--	--
15	--

Execution Dependence Linking Example



Execution Dependence Linking Example



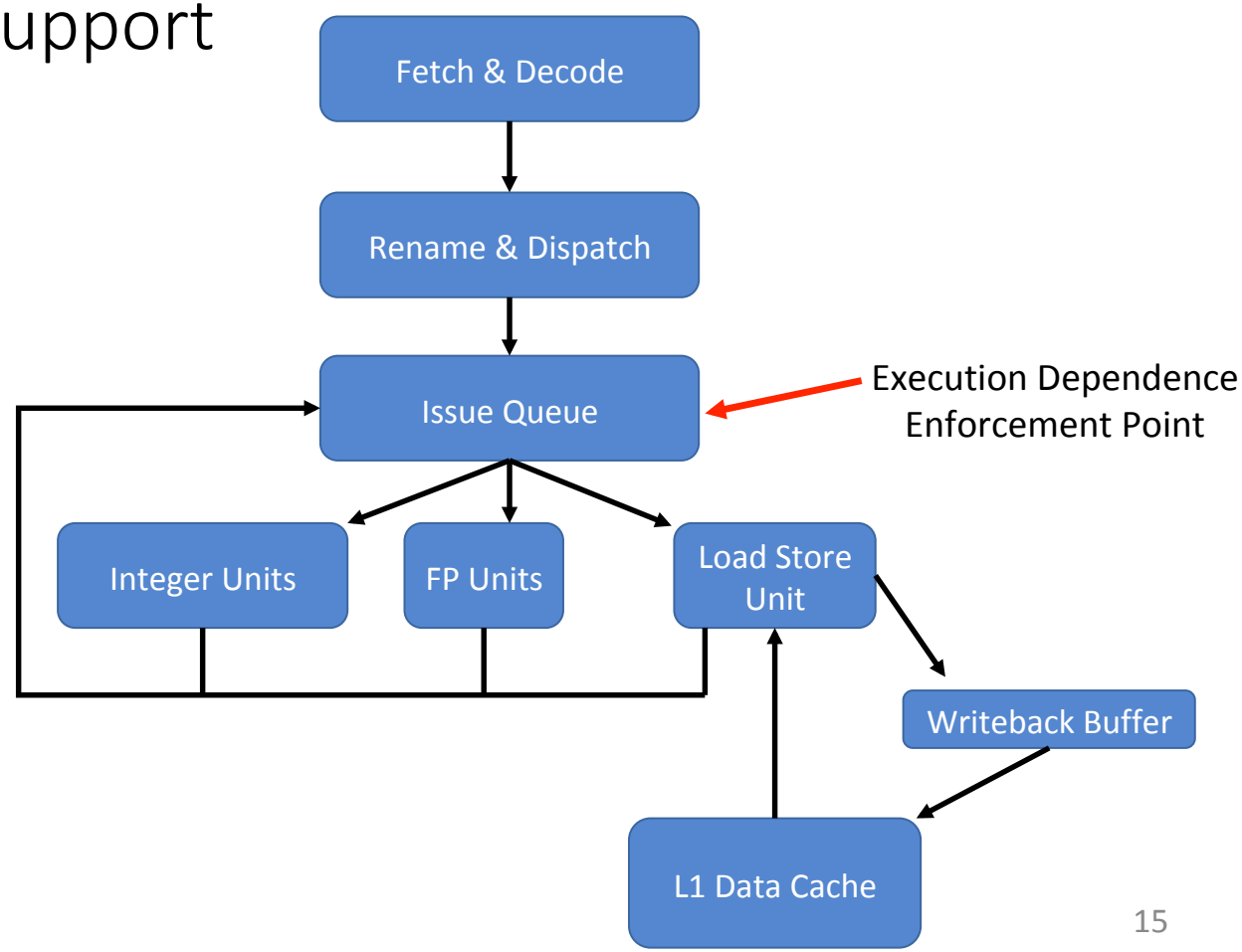
Details in Paper:
Control instructions for more elaborate linking patterns

i-acoma group

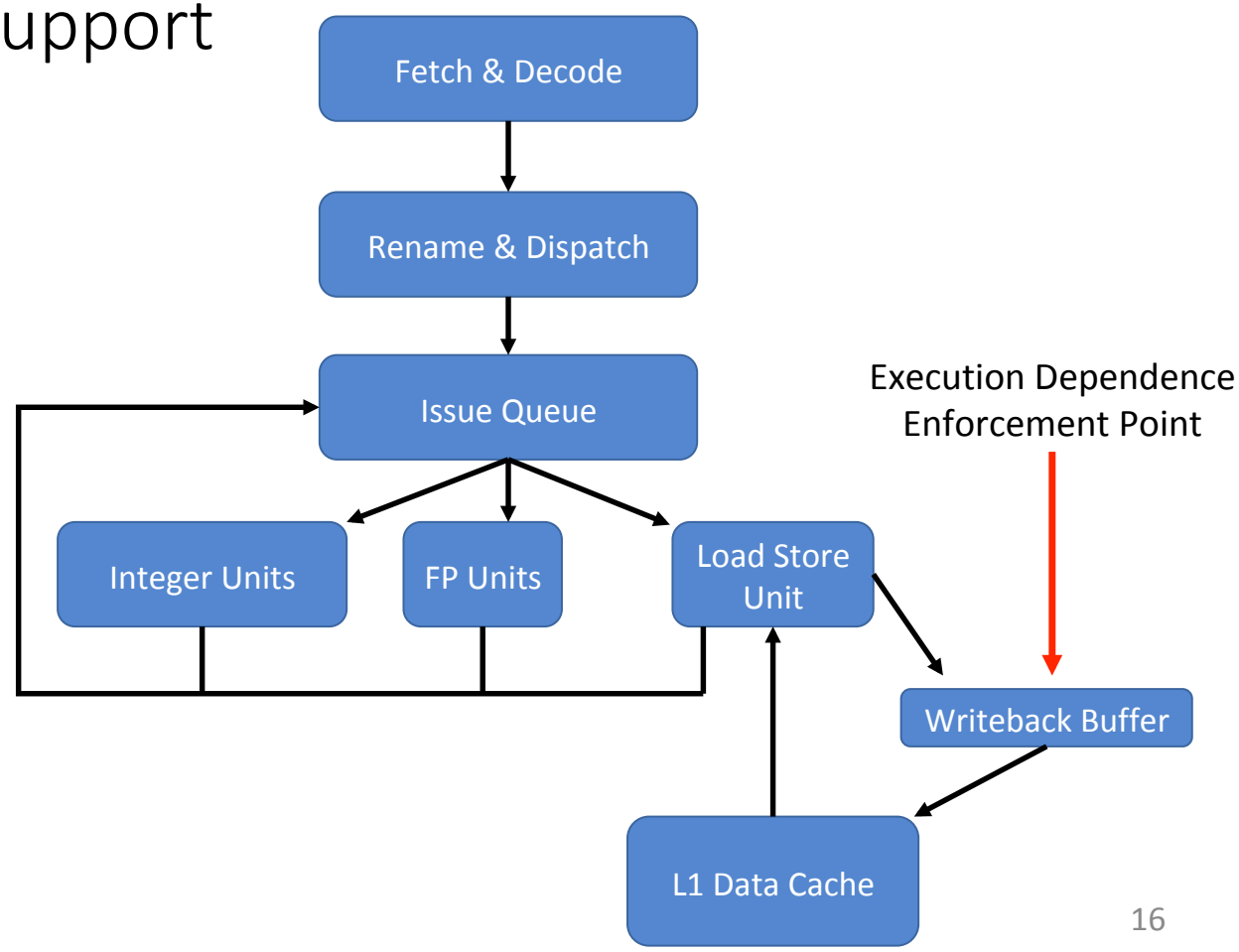
EDE Memory Instruction Variants

- Execution behavior is same as original instructions, but now also describe execution dependences
- For NVM
  - Stores
  - Cacheline Writebacks
- For multi-threaded applications
  - Stores
  - Loads
  - Synchronization primitives
    - Compare&Swap, Fetch&Increment, etc.

EDE Hardware Support Design #1: *IQ*



EDE Hardware Support Design #2 : *WB*



EDE Use Case: NVM Failure-Atomic Regions

Characteristics of a failure-atomic region (i.e., transaction):

- Either all or no memory operations persistently & atomically commit
- Important abstraction for Non-volatile memory (NVM) applications
  - Help applications maintain a recoverable state
- Usually implemented via undo logging

Fences in Undo Logging

1. Write and Persist Log

W [Log_A] = (A_{addr}, A_{old})

CLWB [Log_A]

Required Ordering

FENCE

2. Update Value

W [A_{addr}] = A_{new}

EDE Reducing Fence Overhead

Current Execution

W [Log_A] = (A_{addr}, A_{old})
CLWB [Log_A]
FENCE
W [A_{addr}] = A_{new}
W [Log_B] = (B_{addr}, B_{old})
CLWB [Log_B]
FENCE
W [B_{addr}] = B_{new}
W [Log_C] = (C_{addr}, C_{old})
CLWB [Log_C]
FENCE
W [C_{addr}] = C_{new}

EDE Execution

W [Log_A] = (A_{addr}, A_{old})
CLWB (1, 0) [Log_A]
FENCE
W (0, 1) [A_{addr}] = A_{new}
W [Log_B] = (B_{addr}, B_{old})
CLWB (2, 0) [Log_B]
FENCE
W (0, 2) [B_{addr}] = B_{new}
W [Log_C] = (C_{addr}, C_{old})
CLWB (3, 0) [Log_C]
FENCE
W (0, 3) [C_{addr}] = C_{new}

Details in Paper:
Why trying to reduce # of fences by reordering instructions is not viable

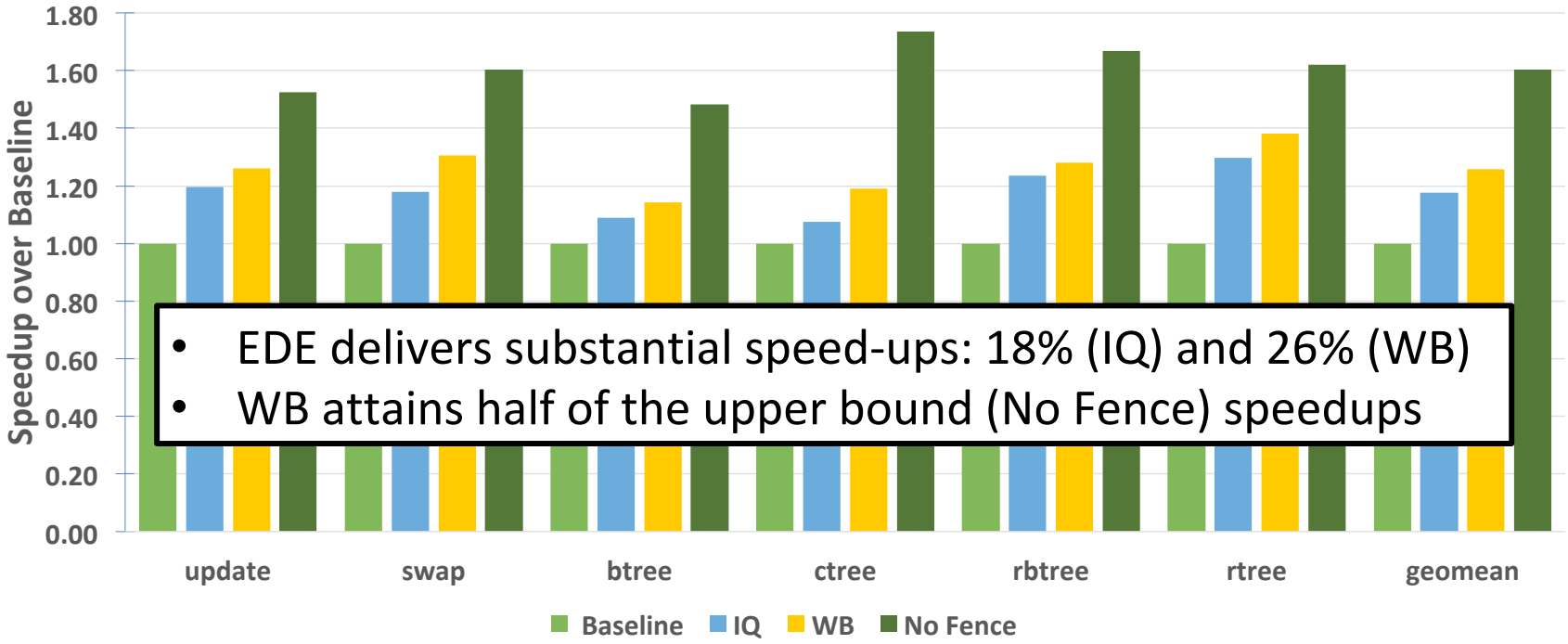
Evaluating EDE on NVM Applications

- Create Clang+LLVM intrinsics for EDE instructions
  - Target Arm AArch64 ISA
- Port NVM applications to use EDE
  - 2 kernels and 4 persistent data structures from Persistent Memory Development Kit (PMDK)
- Implement EDE hardware in gem5

Baseline	All necessary Arm AArch64 fences added
IQ	EDE orderings enforced in issue queue
WB	EDE ordering enforced in write buffer
No Fence	No fences. Allows unsafe reorderings. Upper bound.

Execution Speedups Attained by EDE

↑ Better



- EDE delivers substantial speed-ups: 18% (IQ) and 26% (WB)
- WB attains half of the upper bound (No Fence) speedups

Other EDE Use Cases

- Lock Free Algorithms
  - Hazard pointer announcement (described in paper)
  - Circular buffers

- Managed Language Operations
  - Data initialization
  - Class loading
  - Concurrent garbage collection
  - Shared variable reference counting

- More details in paper

EDE Compiler Support

- EDE strength: compatibility with compilers
- Compilers can internally track execution dependences
  - Compilers already track data, memory, and control dependences
- Eliminating fences allows the compiler to perform more optimizations
  - More opportunities to move code around
- Compiler can automatically perform EDE responsibilities
  - Assign EDK #s to execution dependences
  - Identify execution dependences

Additional Details in Paper

- EDE control instructions
- Overhead of fences in NVM applications
- EDE hardware architecture details
  - EDM checkpointing
  - How EDE instructions propagate through pipeline
- EDE multithread use cases and compiler support

Conclusions

1. Propose EDE extension: new instructions which explicitly state their fine-grain ordering requirements
   - Net impact: fewer fences in applications → improved performance

2. Describe two practical hardware implementations
   - One in issue queue, another in writeback buffer

3. Demonstrate effectiveness of EDE in NVM applications
   - Average workload speedups of 18% (IQ) and 26% (WB)

4. High potential to apply EDE to many multithreaded scenarios