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
Executive Summary

• Propose new instructions which explicitly state their fine-grain ordering requirements
  • Net impact: fewer fences in applications $\rightarrow$ 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

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

// critical section
...

Processor 1

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

// critical section
...

Processor 1
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}]  Execution Dependence: T12 -> T4
2. Set EDM[EDK_{Def}]
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)

<table>
<thead>
<tr>
<th>EDK #</th>
<th>In-Flight Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T33</td>
</tr>
<tr>
<td>2</td>
<td>T12</td>
</tr>
<tr>
<td>3</td>
<td>T4</td>
</tr>
<tr>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>5</td>
<td>T41</td>
</tr>
<tr>
<td>15</td>
<td>--</td>
</tr>
</tbody>
</table>
Decoding Execution Dependences

**Instruction Format:**
INS (EDK\textsubscript{Def}, EDK\textsubscript{Use}), <original operands>

**Execution Dependency Map (EDM)**

<table>
<thead>
<tr>
<th>EDK #</th>
<th>In-Flight Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T33</td>
</tr>
<tr>
<td>2</td>
<td>T12</td>
</tr>
<tr>
<td>3</td>
<td>T4</td>
</tr>
<tr>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>5</td>
<td>T41</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>15</td>
<td>--</td>
</tr>
</tbody>
</table>

**Steps:**
1. Read EDM[EDK\textsubscript{Use}]
2. Set EDM[EDK\textsubscript{Def}]

EDK #0 is *Zero Key* and is ignored
Execution Dependence Linking Example

1. inst (1, 0) <ops>
2. inst (2, 0) <ops>
3. inst (3, 0) <ops>
4. inst (0, 3) <ops>
5. inst (0, 3) <ops>
6. inst (0, 1) <ops>
7. inst (1, 0) <ops>
8. inst (0, 1) <ops>
9. inst (0, 2) <ops>
Execution Dependence Linking Example

1. inst (1, 0) <ops>
2. inst (2, 0) <ops>
3. inst (3, 0) <ops>
4. inst (0, 3) <ops>
5. inst (0, 3) <ops>
6. inst (0, 1) <ops>
7. inst (1, 0) <ops>
8. inst (0, 1) <ops>
9. inst (0, 2) <ops>

Details in Paper:
Control instructions for more elaborate linking patterns
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

Fetch & Decode

Rename & Dispatch

Issue Queue

Integer Units

FP Units

Load Store Unit

Execution Dependence Enforcement Point

Writeback Buffer

L1 Data Cache
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
   \[ W[\text{Log}_A] = (A_{addr}, A_{old}) \]
   \[ \text{CLWB}[\text{Log}_A] \]
   \[ \text{FENCE} \]

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

<table>
<thead>
<tr>
<th>Baseline</th>
<th>All necessary Arm AArch64 fences added</th>
</tr>
</thead>
<tbody>
<tr>
<td>IQ</td>
<td>EDE orderings enforced in issue queue</td>
</tr>
<tr>
<td>WB</td>
<td>EDE ordering enforced in write buffer</td>
</tr>
<tr>
<td>No Fence</td>
<td>No fences. Allows unsafe reorderings. Upper bound.</td>
</tr>
</tbody>
</table>
Execution Speedups Attained by EDE

- 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