Execution Dependence Extension (EDE): ISA Support for Eliminating Fences

Thomas Shull*, Ilias Vougioukas†, Nikos Nikoleris†, Wendy Elsasser†, Josep Torrellas‡
*Oracle Labs§ †Arm Research ‡University of Illinois at Urbana-Champaign
tom.shull@oracle.com, {ilias.vougioukas, nikos.nikoleris, wendy.elsasser}@arm.com, torrella@illinois.edu

***Abstract*—Fence instructions are a coarse-grained mechanism to enforce the order of instruction execution in an out-of-order pipeline. They are an overkill for cases when only one instruction must wait for the completion of one other instruction. For example, this is the case when performing undo logging in Non-Volatile Memory (NVM) systems: while the update of a variable needs to wait until the corresponding undo log entry is persisted, all other instructions can be reordered. Unfortunately, current ISAs do not provide a way to describe such an *execution dependence* between two instructions that have no register or memory dependences. As a result, programmers must place fences, which unnecessarily serialize many unrelated instructions.**

To remedy this limitation, we propose an ISA extension capable of describing these execution dependences. We call the proposal *Execution Dependence Extension (EDE)*, and add it to Arm's AArch64 ISA. We also present two hardware realizations of EDE that enforce execution dependences at different stages of the pipeline: one in the issue queue (*IQ*) and another in the write buffer (*WB*). We implement IQ and WB in a simulator and test them with several NVM applications. Overall, by using EDE with IQ and WB rather than fences, we attain average workload speedups of 18% and 26%, respectively.

***Index Terms*—Fences, Instruction ordering, ISA extensions.**

I. Introduction

Fence instructions are used to enforce a certain order of instruction execution in out-of-order pipelines [10], [27], [46]. For example, a full fence stalls the execution of all subsequent instructions until all older instructions have completed. Memory fences have a similar effect for only memory access instructions, while other types of fences affect other sets of instructions. In all cases, however, a fence is a *coarse-grained* mechanism to order the execution of groups of instructions.

In some cases, however, the requirements of a program only necessitate that the execution of one instruction wait for the completion of one other instruction. In this case, what is needed is a *fine-grained* mechanism to order the two instructions, while allowing the execution of all other instructions to be reordered arbitrarily. Using a fence is an overkill that results in unnecessary execution stalls.

One example of such environment is when using fences to order the persist operations in Non-Volatile Memory (NVM) systems [58]. Consider the case of undo logging, where an element's undo log entry needs to be persisted before the element can be updated. To enforce such ordering, the only current option is to add a fence between the log persist and the element update. Unfortunately, this approach unnecessarily constrains the allowed instruction execution reorderings. For instance, updates to different log entries should be able to execute in parallel, but the insertion of fences forces the serialization of independent instructions.

Ideally, we would like to be able to describe *execution dependences* and have the hardware enforce them. We define an execution dependence as a required ordering between two individual instructions, where the effects of the sink instruction (i.e., the consumer) cannot be observed until the source instruction (i.e., the producer) is complete. In current processors, execution dependences are honored by the underlying hardware if the two instructions have a register dependence (i.e., they communicate through a register) or a memory dependence (i.e., they access the same memory location). However, one cannot make current processors honor execution dependences between two unrelated instructions – e.g., the execution dependence that exists in NVM undo logging between the store to the log entry and the store to the element.

To reduce the need for fences, in this paper, we propose an ISA extension capable of describing execution dependences. We call our new proposal *Execution Dependence Extension (EDE)*, and have added it to Arm's AArch64 ISA [10].

With EDE, we create a new class of instructions and a convention for defining execution dependences. Specifically, we augment some instruction opcodes with a new *key set* called the *Execution Dependence Key (EDK)* set, which is used to link together execution dependence producers and consumers. Via this new key set, an execution dependence can be defined. The linking of instructions via EDKs is very similar to how two data dependent instructions are linked together through registers. With EDE, one can precisely convey a variety of instruction-level execution orderings, including one to one, one to many, and many to one instruction orderings. This description is done without fences, thereby allowing all other instructions to proceed in any order.

Given the execution orderings conveyed by EDE, it is the responsibility of the hardware to honor them. In this paper, we propose two implementations, which we call *IQ* and *WB*. In IQ, EDE's execution dependences are enforced at the issue queue: the hardware monitors execution dependences alongside register and memory dependences to decide when instructions can be executed.

While IQ achieves significant performance gains, there are

§Work performed while Thomas Shull was affiliated with the University of Illinois at Urbana-Champaign and Arm Research.

opportunities for further improvement. This is because IQ stalls instructions early in the pipeline. Hence, we also propose an aggressive design that enforces execution dependences in the write buffer after the instructions have retired. We call this design WB.

To evaluate the performance impact of EDE, we implement IQ and WB in a simulator that models an Arm A72-like core. To demonstrate EDE's utility in NVM environments, we develop NVM kernels and port Persistent Memory Development Kit (PMDK) [3] applications to use EDE instead of fences. Overall, by using EDE with IQ and WB, we attain average workload speedups of 18% and 26%, respectively.

This paper makes the following contributions:

- Identify a type of instruction execution dependence that can benefit from a mechanism for fine-grained instruction ordering.
- Propose EDE, an ISA extension that allows these instruction execution dependences to be conveyed to the hardware.
- Propose two practical hardware realizations of EDE.
- Evaluate EDE and observe significant performance improvements.

II. Background and Motivation

A. Non-Volatile Memory: Fences and Atomic Regions

Recently, low-latency byte-addressable Non-Volatile Memory (NVM) has become commercially available [28]. However, there are multiple levels of volatile caches between the processor and NVM. Hence, one needs to ensure that a processor write to NVM propagates its value beyond the caches and reaches NVM. For this reason, x86-64 processors have introduced the `CLWB` instruction [27], which writes back a cache line to NVM while also retaining the line in the cache.

In Armv8.2-A [8], Arm introduced new instructions to propagate writes to the persistence domain. Specifically, Arm has added the *Data or unified Cache line Clean by Virtual Address to Point-of-Persistence (`DC CVAP`)* [10] instruction to its AArch64 ISA. Like `CLWB`, `DC CVAP` ensures that the value at the provided virtual address is sent to NVM.

In both x86-64 and AArch64, `CLWB` and `DC CVAP` follow a relaxed memory ordering. This means that, when using these instructions, the order in which the updates reach NVM is not deterministic. Hence, to guarantee a given ordering, fences must be inserted. Specifically, on x86-64, to guarantee that all older `CLWB` instructions have completed before younger writes and younger `CLWB`s do, one needs to insert a store fence (`SFENCE`) [27] instruction.

In Arm systems, fences are called barriers. On AArch64, a system-wide data synchronization barrier (*`DSB sy`*) [10], [50] is needed to order `DC CVAP`s relative to other instructions. `DSB`s impose an ordering on *all* instructions: before any instruction younger than a `DSB` can execute, all older instructions must finish. Note that AArch64 also provides a data memory barrier (`DMB`), which only imposes an ordering on memory instructions. However, this instruction does not currently order `DC CVAP`s.

```
1 p_array[0] = 6;
2 p_array[1] = 9;
3 p_array[2] = 42;
```

(a) Application code.

```
1 void p_uint64::operator=(const uint64_t &new_val) {
2   log_value(&_val);
3   update_value(&_val, new_val);
4 }
```

(b) Framework code to inject persistence operations.

Fig. 1. Updating a persistent array.

```
1  void log_value(uint64_t *val) {
2    uint64_entry *slot = undo_log->reserve_uint64();
3    slot->addr = val;
4    slot->val = *val;
5    asm volatile(
6      "dc cvap %x0"
7      "dsb sy"
8      : : "r"(slot)
9      : "memory");
10 }
```

(a) Framework function to log & persist original value.

```
1  void update_value(uint64_t *val, uint64_t new_val) {
2    *val = new_val;
3    asm volatile(
4      "dc cvap %x0"
5      : : "r"(val)
6      : "memory");
7  }
```

(b) Framework function to update & persist value.

Fig. 2. Framework implementation of persistence operations.

One of the key features that NVM frameworks provide to users is failure-atomic support, i.e., the ability to allow a collection of stores to persist atomically. In a *failure-atomic region* [14], [18], [24], [55], [60], either all the stores persist at the same time, or none of them persist. Frameworks traditionally support failure-atomic regions with undo logging, redo logging, or copy-on-write [37]. These operations typically require the insertion of many fences.

B. Fence Overhead in NVM Applications

To understand how fences negatively affect performance, let us consider how a persistent application implements undo logging with fences. In Figure 1(a), we show a code snippet that updates three elements of a persistent array of 64-bit values (`p_uint64`). To minimize the effort needed for a developer to create a persistent application, it is common for persistent frameworks to use operator overloading to automatically perform the necessary persistent actions. In Figure 1(b), we show how a persistent framework overloads the C++ assignment operator used in each of the three statements of Figure 1(a) to transparently perform the needed persistent actions. In the figure, the code first logs the original value (`_val`) via function `log_value` and then updates `_val` to its new value (`new_val`) via function `update_value`.

In Figures 2(a) and (b), we show how the undo logging and the value update are persistently performed. In `log_value`,

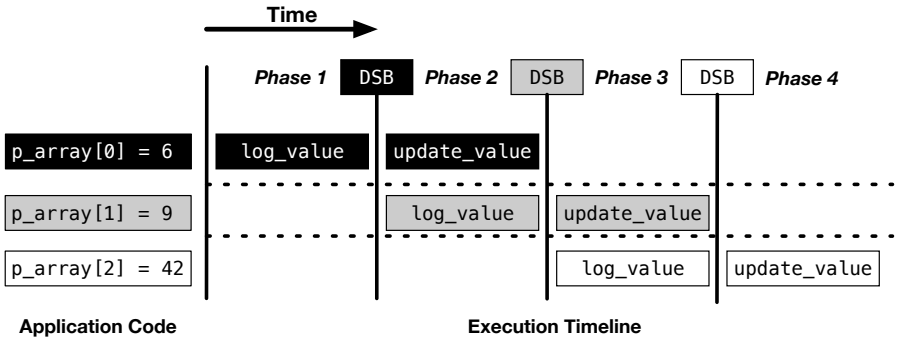

Fig. 3. Reordering limitations imposed by DSBs. Time goes from left to right.

```
1  // x0 contains p_array ptr
2
3  // p_array[0] = 6
4  // x2 contains slot ptr
5  ldr  x1, [x0] ; load original value
6  stp  x0, x1, [x2] ; store addr & val
7  dc cvap, x2 ; persist slot
8  dsb sy ; wait for slot to persist
9  mov  x3, #6 ; load new value
10 str  x3, [x0] ; store new value
11 dc cvap, x0 ; persist new value
12
13 // p_array[1] = 9
14 // asm is same as above,
15 // modulo ptr locations/stored value
16
17 // p_array[2] = 42
18 // asm is same as above,
19 // modulo ptr locations/stored value
```

Fig. 4. AArch64 assembly for Figure 1(a).

a slot within the undo log is first reserved (Line 2). Next, the address and original value are stored in the log (Lines 3 and 4). Finally, the log entry is persisted (Lines 5-9). Note that, to ensure that the correct persist instructions are emitted, it is necessary to write `asm volatile` code, as is done in Lines 5-9. As described in Section II-A, on AArch64, one needs to issue a `DC CVAP` (Line 6) and `DSB` (Line 7). The `DSB` is necessary to guarantee that the log entry is persisted before the next update can reach memory. In `update_value` (Figure 2(b)), first the new value is stored to the variable's location (Line 2) and then a `DC CVAP` is issued to guarantee the value reaches NVM (Line 4). Here a `DSB` is unnecessary, as the value only needs to become persistent by the end of the failure-atomic region.

Figure 4 shows the assembly instructions generated by the compiler for the code shown in Figure 1(a). Line 5 loads the original value of `p_array[0]`. On Line 6, both the original value and its address are stored into the log via AArch64's *pairwise-store* (`STP`) instruction. On Line 7 a `DC CVAP` is issued to persist the log entry. Since `STP` is 16-byte aligned, both stored values are on the same cache line; hence, only one `DC CVAP` is necessary to persist both values. Line 9 moves the new value to be stored into a register, while Lines 10 and 11 perform the update and persist of the new value.

While Figure 4 only shows the assembly generated for `p_array[0]=6`, the assembly for `p_array[1]=9` and `p_array[2]=42` is nearly identical to the first update; the only difference is that `x0` is updated to point to the storage locations for `p_array[1]` and `p_array[2]`, and that 9 and 42 are stored to the proper new locations.

1) The Problem with Fences: Unfortunately, fences are a coarse-grained mechanism to order instructions. They enforce the order of many unrelated instructions, which can add a significant performance overhead. To see how, consider Figure 3, which shows the impact that `DSB`s have on the three updates in Figure 1(a). The figure shows a timeline of when each of the updates' operations can execute relative to the `DSB`s. In the figure, time goes from left to right. For clarity, we have also colored the operations associated with each update differently: black for `p_array[0]=6`, gray for `p_array[1]=9`, and white for `p_array[2]=42`.

In the figure, we label the execution phases created by the `DSB`s. During phase 1, only the `log_value` related to `p_array[0]=6` can execute. This is because the `DSB` at the end of `log_value` prevents the instructions in `p_array[0]=6`'s `update_value` and those in `p_array[1]=9` and `p_array[2]=42` from executing. In Phase 2, both `update_value` from `p_array[0]=6` and `log_value` from `p_array[1]=9` can execute concurrently. However, the second `DSB` blocks instructions from `p_array[1]=9`'s `update_value` and also from `p_array[2]=42`. As can be seen in the figure, in total four phases are necessary to complete these three updates.

The three `p_array[]` updates in the original code are to different addresses and also reserve different log entry slots. Since the updates are independent, ideally the processor could overlap the execution of instructions from the three updates. Specifically, to execute these instructions should only take two phases: the `log_value` of each update could execute in parallel, while the `update_value` of each update should only wait for its corresponding `log_value` to complete. However, because a `DSB` is needed to enforce the required persistence orderings, these updates are unnecessarily serialized.

Code patterns such as Figure 3 are commonplace for failure-atomic regions when using NVM frameworks. Furthermore, it is usually not possible to reorder the instructions to limit the fence overhead for two reasons. First, the compiler has no information about the intention behind the fences inserted by the framework code and hence is unable to move operations

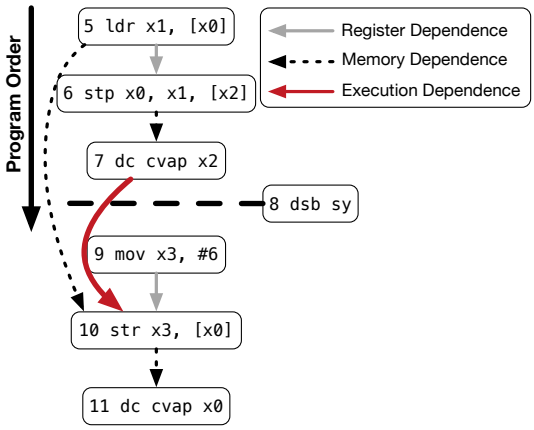

Fig. 5. Dependence graph for the instructions in Figure 4.

around them. Second, because the persistent operations are often transparently inserted by the framework code, such as in Figure 1(b), the user is unable to manually reorder operations.

III. Main Idea

A. Inability to Convey Instruction Execution Dependences

The issue pointed out in the previous section is part of a larger problem: currently, it is not possible to convey arbitrary instruction dependences within the ISA. Currently, only register and memory dependences can be conveyed by the ISA. Register dependences are communicated by two instructions referring to the same register, while memory dependences are conveyed by two instructions accessing the same address. However, logical ordering dependences also often exist which are not expressible via a memory location or a register.

In Figure 5, we show the same instructions as in Figure 4, only now with the register (gray arrows) and memory dependences (dashed arrows) in the code labeled. In the figure, register dependences go from instructions which define a register to instructions which use the same register. Memory dependences, on the other hand, chain together all accesses to a given address.

This collection of register and memory dependences imposes a set of scheduling restrictions on out-of-order processors. In order to keep the functional units full and throughput high, processors try to execute instructions as early as possible. However, processors still must respect all register and memory dependences within the execution.

In our example, in cases where a `DC CVAP` must follow an `STP` and `STR`, such as in lines 6→7 and 10→11 of Figure 5, the memory dependence between the pair of instructions ensures they are not sent to memory out of order. However, as explained before, we also need to ensure that the `DC CVAP` of Line 7 executes before the `STR` of Line 10. To convey this in Figure 5, we introduce the notion of an *execution dependence*. In the figure the execution dependence between `dc cvap x2` and `str x3, [x0]` is represented by a red arrow.

An execution dependence denotes a required ordering of the two instructions. It means that, for correctness, the execution dependence's source operation must complete before the dependence's sink operation can be observed. Note that when an instruction "completes" is defined as when the operation makes observable changes. Section IV-B1 describes completion in more detail.

Since `dc cvap x2` does not produce a register used by `str x3, [x0]`, nor do these instructions access the same memory address, the processor does not naturally respect this execution dependence. Therefore, it is necessary to insert a `DSB` in the code, as shown in Figure 5. Unlike register and memory dependences, the `DSB` enforces an ordering across *all* instructions. Unfortunately, this is presently the only option one has to ensure our execution dependence is honored by out-of-order processors.

B. Encoding Execution Dependences within Instructions

Given the above limitations, in this paper we propose to add new instructions which convey execution dependences. Execution dependences explicitly enforce the completion ordering of the specified instructions. Collectively, we call our new instructions the *Execution Dependence Extension (EDE)*.

In our new instructions, in addition to the traditional memory and register dependences, an execution dependence on an arbitrary prior instruction can also be defined. We enable this by introducing several new concepts: the Execution Dependence Key (EDK) set, EDK-producing instructions, and EDK-consuming instruction. The main idea is that EDE allows for instructions to be linked so that the sink instruction (EDK consumer) cannot be observed until the source instruction (EDK producer) is complete.

The benefit of EDE is that, by explicitly describing execution dependences between instructions at the ISA level, the number of fences needed within applications is substantially reduced. For instance, in Figure 5, by using EDE, it is possible to convey the required execution dependence between `dc cvap x2` → `str x3, [x0]` without the need for a `DSB` in between the two instructions. By reducing the number of fences, the processor is able to achieve greater performance by allowing more instructions to be executed in parallel.

C. Hardware Support for EDE

Via EDE, it is possible to convey instruction execution dependences to the hardware. However, it is the hardware's responsibility to use this provided information to maximize an application's performance. While many hardware implementations of EDE are possible, in this paper we choose to evaluate two practical options. Our first implementation, called IQ, enforces execution dependences at the issue queue. The execution of an EDK-consuming instruction is delayed until the corresponding EDK-producing instruction has completed.

While IQ is effective, it is possible to achieve further performance benefits. Indeed, since stores and cacheline writeback instructions do not make any observable memory changes until after they retire, IQ unnecessarily stalls the execution of an EDK-consuming store or cacheline writeback instruction early in the pipeline.

To prevent such stalling from happening, we also propose a more aggressive design which enforces the ordering of store and cacheline writeback instructions at the write buffer. We call this design WB. In WB, store and cacheline writeback instructions are allowed to retire before their execution dependences are satisfied. However, at the point where their changes might be pushed to memory (i.e., in the write buffer), the execution dependences are enforced. Overall, by allowing these EDK-consuming store and cacheline writeback instructions to retire, WB enables subsequent independent instructions to proceed unencumbered.

IV. EDE ISA Definition

In this section, we describe EDE's specification. First, we describe the high-level concepts needed to define execution dependences. Afterwards, we introduce the EDE instructions.

A. EDE Concepts

To allow execution dependences to be conveyed by the ISA, we provide new abstractions to define a dependence source, a dependence sink, and the linking together of the source and sink. We call the dependence source instruction the *dependence producer* and the dependence sink instruction the *dependence consumer*. We link them together with *Execution Dependence Keys (EDK)s*.

1) Execution Dependence Keys: EDKs provide a way to link two instructions together and, therefore, convey execution dependences. Like traditional registers, EDKs are directly encoded into instructions. However, unlike registers, no data is read or written. Instead, EDKs are used to index an *Execution Dependence Map (EDM)*. The EDM holds EDK-to-instruction key-value pairs. After an instruction is decoded, the EDM is accessed, and the instruction's EDKs are used to determine any execution dependences. Specifically, first, the EDM is searched to check if the instruction is the sink of any execution dependence. Second, if the instruction is the source of any execution dependence, the EDM is updated. Section IV-C shows examples of how EDKs are used.

We define sixteen EDKs (EDK #0 − EDK #15). Throughout the rest of the paper, we refer to an EDK operand as `EDK #`, where `#` refers to the key being accessed. The EDM map itself only has to hold fifteen, not sixteen entries. This is because `EDK #0` serves as a *zero key*. When the zero key is encoded into an instruction, it means that this field is not being used and can be ignored. This is needed when a given instruction is not an execution dependence source or not a sink.

2) Dependence Producers: A dependence producer instruction is the source of an execution dependence that can be consumed by one or more dependence consumer instructions. A dependence producer provides an EDK that is used to access the EDM. The EDM is updated to store the new EDK-to-instruction link in the appropriate slot. In this way, subsequent dependence consumer instructions using the same EDK will be able to query this EDM entry and be linked to the producer.

3) Dependence Consumers: A dependence consumer instruction is dependent on one or more prior dependence producer instructions. A dependence consumer provides an EDK that is used to access the EDM. If an entry in the EDM is found for the provided EDK, then it means that this dependence consumer must wait for the instruction in the EDM entry to finish. A dependence consumer merely queries the EDM, and does not modify it in any way. Note that multiple dependence consumers can depend on the same dependence producer.

B. New Instructions

Based on these concepts, we now define the instructions introduced by EDE. We describe the new memory instruction variants first, and then the new control instructions.

1) EDE's New Memory Instruction Variants: We propose new variants of memory instructions, which take the following operands:

`EDK`$_{def}$`, EDK`$_{use}$`, <original operands>`

where `EDK`$_{def}$ is the instruction's dependence producer key, `EDK`$_{use}$ is the instruction's dependence consumer key, and `<original operands>` are the instruction's original operands—e.g., `<REG`$_{val}$`, [REG`$_{mem}$`]>` for `STR`. When we write an instruction, we place a parenthesis before `EDK`$_{def}$ and after `EDK`$_{use}$.

When using this instruction variant, it is possible for an instruction to be both a source of a dependence (as given by `EDK`$_{def}$) and a sink of a dependence (as given by `EDK`$_{use}$). It is also possible for an instruction not to be a source or a sink of a dependence by using the zero key (EDK #0) in the appropriate operand field.

In this format, it is only possible for a consumer instruction to be dependent on one previous instruction. However, as will be explained in Section IV-B2, this limitation is removed by using the EDE control instructions.

While this execution dependence format can be added to many memory instructions, in this paper, we only add it to AArch64's stores and cacheline writebacks. In Section VIII, we discuss applying execution dependence variants to loads and synchronization primitives.

A key component of these instruction variants is the definition of when a dependence producer instruction has completed. Only after the dependence producer has completed is it safe for the corresponding dependence consumer to be sent to memory. In our design, store (`STR`(H,B)) and pairwise-store (`STP`) instructions are completed once the stored value(s) are visible to all processors. Writeback instructions (`DC CVAP`) are completed once the corresponding data is guaranteed to be persisted. These are the same definitions for completion used by the original AArch64 instructions.

2) EDE's New Control Instructions: EDE introduces three new instructions to handle unusual control `JOIN` (`EDK`$_{def}$`, EDK`$_{use_1}$`, EDK`$_{use_2}$), `WAIT_KEY (EDK)`, and `WAIT_ALL_KEYS`.

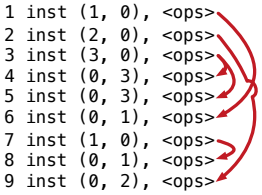

Fig. 6. Example of how EDKs can be used.

JOIN (EDK$_{def}$, EDK$_{use_1}$, EDK$_{use_2}$). This instruction can wait on up to two prior dependence producers and is completed once both of its producers have completed. Via multiple `JOIN`s, it is possible for an instruction to have execution dependences with multiple prior dependence producers. `JOIN` is also useful for EDK resolution when multiple control paths merge. For instance, if an instruction should wait on `EDK #1` from one path and `EDK #2` from another, one would insert a `JOIN (EDK`$_{def}$`, EDK #1, EDK #2)` before the instruction to support the desired outcome.

WAIT_KEY (EDK) and **WAIT_ALL_KEYS**. In the presence of function calls, without intervention, it is possible for a callee function to overwrite an EDK key in use by the caller function and cause incorrect execution dependences to be linked. To prevent this from happening, EDE introduces the `WAIT_KEY (EDK)` instruction. This instruction is both a dependence producer and a consumer of the same key. In addition, unlike other instructions, `WAIT_KEY (EDK)` is only considered complete once all prior dependence producers of the matching key have also finished. Therefore, this instruction can be used after a function call to ensure all necessary dependences are met. Details about how to define a calling convention for EDE are presented in Section IX-B.

The `WAIT_ALL_KEYS` instruction prevents *all* subsequent consumer instructions from executing until *all* prior dependence producers and consumers complete. This instruction can be useful to ensure that all persistency operations in a large code section have finished.

C. EDE Example

In Figure 6, we show how EDKs can be used to define the execution dependences among a series of instructions. In the figure, the execution dependences are shown with arrows. There are execution dependences between Instructions 1→6, 2→9, 3→(4,5), and 7→8. Notice how, by using different EDKs, it is possible to define execution dependences between multiple instructions concurrently. For example, Instruction 1 stalls the execution of Instruction 6 by using EDK #1, while not affecting the execution of the other instructions. In addition, as shown in lines 1→6 and 7→8, EDKs can be reused to establish new execution dependences.

Figure 7 shows how to apply EDE to the persistence operations described in Figures 2(a) and (b). Remember that, in the previous `log_value`, it was necessary to insert a `DSB` on Line 7 to ensure that Line 6 in `log_value` completed before Line 2 of `update_value` executed. Now, these instructions can use our execution dependence format to convey this dependence. Specifically, Line 6 of `log_value` in Figure 7(a)

```
1  void log_value(uint64_t *val) {
2    uint64_entry *slot = undo_log->reserve_uint64();
3    slot->addr = val;
4    slot->val = *val;
5    asm volatile(
6      "dc cvap (1,0), %x0"
7      "dsb sy"
8      : : "r"(slot)
9      : "memory");
10 }
```

(a) Labeling dependence producer with EDK #1.

```
1  void update_value(uint64_t *val, uint64_t new_val) {
2    asm volatile(
3      "str (0,1), %x1, [%x0]"
4      "dc cvap %x0"
5      : : "r"(val), "r"(new_val)
6      : "memory");
7  }
```

(b) Labeling dependence consumer with EDK #1.

Fig. 7. Applying EDE to the persistence operations of Figure 2.

is modified to record that `DC CVAP` is a dependence producer of `EDK #1` and consumes the zero key. Similarly, Line 3 of `update_value` in Figure 7(b) is modified to record that `STR` is a dependence consumer of `EDK #1` and produces the zero key. By doing so, the `DSB` is no longer necessary and can be removed.

V. Hardware Implementation

A. Mapping Producer-Consumer Pairs

As described in Section IV-A1, the Execution Dependence Map (EDM) is a fifteen-entry map that holds EDK-to-instruction key-value pairs. In our implementation, to represent the dependence producer in an EDM entry, we store the producer's in-flight instruction ID.

After an instruction with EDKs is decoded, the hardware accesses the EDM. If the instruction has a consumer EDK and the EDM entry for that EDK is empty, then the instruction does not have any execution dependence. If, instead, the EDM entry for that EDK is not empty, the instruction is registered to have an execution dependence on the corresponding in-flight instruction. Furthermore, if the decoded instruction has a producer EDK, then the proper EDM entry is updated to store the instruction's ID.

Once an instruction has completed from EDE's perspective, it is necessary to remove its entry from the EDM. Therefore, when a dependence-producing instruction completes, the corresponding EDM entry is queried. If the hardware finds an ID that matches that of the completed instruction, then the EDM entry is cleared.

1) Checkpointing EDM State: In out-of-order processors, squashes are sometimes needed to flush speculative state out of the pipeline. In this situation, the EDM must be reverted to a non-speculative state. To accomplish this, we keep two copies of the EDM: one at the current non-speculative state ($EDM_{no-spec}$) and another at the current speculative state (EDM_{spec}) – a technique commonly used for register mappings [26]. Throughout normal execution, the EDM$_{spec}$ is used by the front end. However, on a pipeline squash, EDM$_{no-spec}$

Fig. 8. Comparison between ideal and IQ execution timelines. Time goes from left to right.

is copied into EDM_{spec} before execution restarts. Note that, due to the similarity between the EDM and a register file, it is also straightforward to extend this support to maintain a consistent EDM state when multiple checkpoints are taken.

B. Implementing EDE's Memory Instructions

Once execution dependences are identified, the processor's back end must ensure that all dependences are upheld. While many strategies are possible, in this paper we propose two solutions: *IQ*, which enforces execution dependences in the issue queue, and *WB*, which waits until the write buffer to enforce these dependences. We describe each design below.

1) Enforcing Dependences in the Issue Queue: In IQ, all required execution dependence orderings are enforced in the issue queue. This is done by adding an additional step to instructions' wakeup logic. Normally, an instruction is deemed ready to execute once all of its register and memory dependences have been met. In IQ, we add the extra step of also monitoring the status of the instruction's execution dependences. Specifically, we add the *execution dependences ready (eDepReady)* flag to each instruction in the issue queue. When an instruction enters the issue queue, the hardware checks to see if it has any outstanding execution dependences. If it does, *eDepReady* is cleared; otherwise, it is set. If *eDepReady* is clear, it will later get set once the instruction's execution dependences are satisfied. When *eDepReady* is set and all the other dependences are satisfied, the instruction is marked as ready to execute. The existing issue queue scheduling logic is otherwise unmodified.

When an instruction completes, IQ sets the *eDepReady* of all matching dependence consumer instructions in the issue queue. This action may cause multiple instructions to become ready to execute.

2) Drawbacks of IQ: While IQ is effective, its performance is limited. To see why, note that EDK-consuming stores and cacheline writebacks do not make any observable memory changes until after they retire. Hence, it is suboptimal to stall their execution due to an execution dependence early in the pipeline as is required by IQ.

To better understand this limitation, consider Figures 8(a) and (b). Each figure shows a different execution timeline for a set of four EDE instructions with two execution dependences. Note that the instructions access different addresses. The timeline follows an abstract pipeline with *Issue*, *Execute*, and *Retire* stages. After that, there is a *Push to Mem* step where the entry in the write buffer corresponding to the store or writeback instruction is pushed to the memory system. We show the *Push to Mem* step as taking longer than a pipeline stage because it is subject to memory subsystem latencies. In the timelines, time goes from left to right. There are gaps between the stages when an instruction is stalled.

Figure 8(a) shows an ideal timeline that enforces the execution dependences. As in conventional pipelines, instructions go through *Issue* and *Retire* in order. To enforce the execution dependence between the first and second instructions, all that is required is that the second instruction delays its *Push to Mem* step (and hence remains invisible to other processors) until the first instruction completes—i.e., it finishes its own *Push to Mem* step.

The third instruction does not have an execution dependence a prior instruction. Hence, it can proceed to begin its *Push to Mem* step immediately after it retires. The fourth instruction is dependent on the third one. Therefore, it must wait to perform its *Push to Mem* step until the third instruction has completed.

In contrast, figure 8(b) shows the timeline with IQ. Since IQ enforces execution dependence ordering at the issue queue, the second instruction is unable to proceed to *Execute* until the first instruction completes its *Push to Mem* step. Then, even though the third instruction is independent of the second one, it has to order its *Retire* after the second instruction's *Retire*. Finally, the fourth instruction is dependent on the third one and, therefore, orders its *Execute* after the third instruction's *Push to Mem* step. Overall, IQ is unable to unlock the ideal amount of parallelism between these instructions, and the execution takes a significantly longer time.

While the code example provided is very simple, this pattern of pairwise instruction dependences is common in NVM applications. As described in Section II-B, one motivating factor for EDE is to allow multiple log updates to proceed in parallel. Unfortunately, IQ does not enable all the possible parallelism.

3) Enforcing Dependences in the Write Buffer: To resolve the performance limitations of IQ, we introduce a second im-

plementation called WB where execution dependences are resolved in the write buffer. Instructions that consume execution dependences are allowed to continue executing as normal until retirement—irrespective of whether their producer instructions have completed. After retirement, the hardware controls in which order instructions in the write buffer are allowed to push their data to the memory subsystem.

The execution timeline of WB is like the one in Figure 8(a). By allowing instructions to retire without stalling for execution dependences, WB has noticeable performance benefits over IQ. Section V-D describes the design of WB in more depth.

C. Implementing EDE's Control Instructions

The `JOIN (EDK`$_{def}$`,EDK`$_{use_1}$`,EDK`$_{use_2}$`)`, `WAIT_KEY (EDK)`, and `WAIT_ALL_KEYS` control instructions are implemented as follows.

JOIN (EDK$_{def}$, EDK$_{use_1}$, EDK$_{use_2}$). This instruction monitors two execution dependence sources. In both IQ and WB, once the two source instructions complete, then the `JOIN` instruction can complete.

WAIT_KEY (EDK) and **WAIT_ALL_KEYS.** These instructions monitor the completion of many older instructions. Every time that an EDE instruction completes, younger WAIT instructions waiting for a certain EDK (or for all EDKs) perform a check on whether any older instruction is a dependence producer or consumer of that EDK (or of any EDK). If none is, the WAIT instruction can complete.

Section V-D describes the implementation of these instructions in WB in more depth.

D. WB Implementation

In WB, the write buffer contains logic to enforce the EDE dependences. Recall that, after a store or writeback instruction S with EDKs is decoded, the hardware accesses the EDM. If S is the consumer of an execution dependence, S reads from the EDM the ID of the in-flight instruction that is the source of the dependence. S carries this ID in a src_{ID} tag down the pipeline and into the write buffer. When S retires and its entry is deposited into the write buffer, the hardware performs a CAM operation to check if the write buffer contains the entry corresponding to the src_{ID} instruction. If it does not, S's entry clears its src_{ID} field and starts pushing its data to memory. In addition, every time that an entry S_1 in the write buffer is pushed to memory, the hardware checks if any younger entry S_2 in the write buffer has a src_{ID} tag set to S_1. If so, S_2's entry clears its src_{ID} field and starts pushing its data to memory.

When a `JOIN` instruction is decoded, it accesses the EDM and potentially reads two source tags (src_{ID_1} and src_{ID_2}). This special instruction carries these two tags down the pipeline and into the write buffer. Since the write buffer entry for the `JOIN` instruction has no data, one of the src_{ID} tags can use the entry's data field. The logic is similar to regular writes and writebacks, except that the two src_{ID} tags of the JOIN have to be cleared for the JOIN instruction to be removed from the write buffer.

Finally, to support `WAIT_KEY` and `WAIT_ALL_KEYS`, we introduce a set of counters to track both the per-EDK and the overall number of EDE instructions in the write buffer. All stores, writebacks, and `JOIN` instructions carry their EDK tags (EDK$_{def}$, EDK$_{use_1}$ and, for `JOIN`, EDK$_{use_2}$) down the pipeline and into the write buffer. This information is used to increment the matching counters when the instruction enters the write buffer, and to decrement the same counters when the instruction completes. When a `WAIT_KEY` or `WAIT_ALL_KEYS` instruction is ready to retire, it checks the appropriate counter, and can only retire once the counter reaches zero.

VI. Experimental Setup

A. Simulator Environment

To evaluate EDE, we implement both IQ and WB in the gem5 simulator [11] and run several persistent applications. We add EDE's instructions both to gem5's AArch64 frontend, and as built-ins in Clang and LLVM [33] version 8.0.

In the simulator, we use an out-of-order (OoO) configuration that models an Arm A72-like processor [9]. The main simulator architectural parameters are shown in Table I. We have also modified the simulator to model a hybrid DRAM plus NVM memory system with Asynchronous DRAM Refresh (ADR) support. In our setup, both NVM and DRAM requests are sent to one controller. However, the physical address space is split so that part of the address space targets NVM while the other part targets DRAM. The DRAM interface models 2400MHz DDR4, while the NVM interface has asymmetric read and write latencies, and includes a persistent 128-slot on-DIMM buffer. This buffer temporarily buffers updates to the NVM.

TABLE I
Architectural parameters.

ISA	AArch64 + EDE extension
Compiler	Clang-LLVM 8.0 + EDE built-ins
Processor Parameters (Arm A72-like core)	
Processor	OoO core, 3-instr decode width, 3GHz
Ld-St queue	16 entries each
Write buffer	16 entries
L1 I-cache	32KB, 2-way, 2-cycle access latency
L1 D-cache	48KB, 3-way, 1-cycle access latency
L2 cache	256KB, 16-way, 12-cycle access latency
L3 cache	1MB/core, 16-way, 20-cycle access latency
Main-Memory Parameters	
Capacity	DRAM: 2GB; NVM: 2GB
NVM latency	150ns read; 500ns write
NVM line size	256B
NVM on-DIMM buffer	128 slots
DRAM type	2400MHz DDR4
DRAM ranks per channel	2
DRAM banks per rank	16

B. Applications

To evaluate the performance impact of EDE on persistent applications, we use a combination of kernel applications and applications available from the PMDK [3] repository. We list the applications in Table II. We created two kernel applications that perform a series of modifications to an array: in *update*,

an operation consists of updating a random element in the array, while in *swap*, an operation is to swap the values of two random elements in the array. In both applications, undo logging is used to maintain crash consistency.

We also evaluate EDE on a collection of applications from PMDK's *pmembench* benchmark suite. This is accomplished by changing the PMDK framework code to leverage EDE while performing undo logging. As shown in Table II, our evaluation tests four data structures: *btree*, *ctree*, *rbtree*, and *rtree*. In each application, a single operation consists of inserting a new element into the data structure.

In our applications, multiple operations are grouped into a transaction. Specifically, in our simulations, we set the applications to have 100 operations per transaction and to run 1,000 transactions, resulting in each application performing 100,000 operations. In the simulations, we run each application to completion while precisely simulating the time spent performing these operations.

TABLE II
APPLICATIONS EVALUATED.

Kernel Applications	
update	Perform updates on random elements in an array.
swap	Perform pairwise swaps between random array elements.
PMDK Applications	
btree	B-tree implementation with between 3 and 7 keys per node.
ctree	Crit-bit trie [40] implementation.
rbtree	Red-black tree implementation with sentinel nodes.
rtree	Radix tree implementation with radix 256.

C. Architecture Configurations

We compare the five architecture configurations shown in Table III. *Baseline* (*B*) uses `DSB`s as needed to ensure a correct crash-consistent ordering in AArch64. *Store Barrier Unsafe* (*SU*) uses AArch64's store barrier instruction (`DMB st`) to only enforce ordering between store instructions. This configuration allows the hardware to perform reorderings that violate AArch64's crash-consistency requirements. *SU* is shown to approximate the overhead of the store fences needed within NVM applications on x86-64 machines. Both *IQ* and *WB* use EDE instead of `DSB`s to properly order persists and stores. Finally, in *Unsafe* (*U*), all `DSB`s from the program are removed. With *U*, the hardware may perform reorderings that violate crash-consistency requirements.

TABLE III
ARCHITECTURE CONFIGURATIONS.

Configuration	Description
Baseline (*B*)	Use `DSB`s to enforce ordering.
Store Barrier Unsafe (*SU*)	Use `DMB st` to only enforce store ordering. Similar to x86-64 `SFENCE`. Allows unsafe reorderings.
IQ	Use EDE and target IQ hardware.
WB	Use EDE and target WB hardware.
Unsafe (*U*)	No fences. Allows unsafe reorderings.

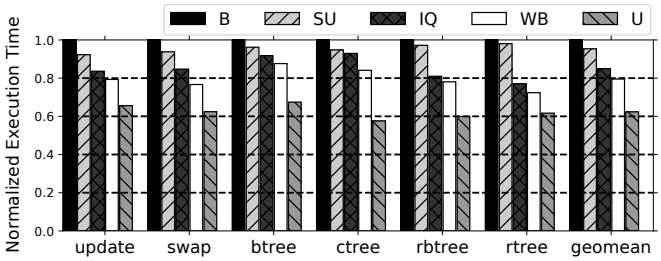

Fig. 9. Application execution time.

VII. EVALUATION

A. Execution Time

Figure 9 shows the execution time for the applications and configurations described in Section VI. For each application, all execution times are normalized to *B*. Based on their geometric mean, *SU*, *IQ*, *WB*, and *U* reduce the execution time by 5%, 15%, 20%, and 38%, respectively.

Most applications attain similar amounts of improvement. *SU* outperforms *B* since `DMB st`s only block store instructions, not all instructions like `DSB`s. Across all applications, *IQ* outperforms *B* and *SU*. This is because EDE uses a fine-grained mechanism to order dependent instructions, while *B* and *SU* use the course-grained mechanism provided by fences. Likewise, *WB* performs better than *IQ* across all applications. This is because, as discussed in Section V-B2, since IQ enforces execution dependences at the issue queue, it is unable to maximize the amount of parallelism described with EDE. Since WB enforces execution dependences in the write buffer, it is able to allow more overlapping of writes.

On average, *WB* is able to attain 54% of the execution time reduction of *U* (i.e., 20% versus 38%). Hence, it recovers a significant portion of the time spent ensuring that NVM is updated in a crash-consistent order. Remember that, since *U* removes all fences from the code, it allows reorderings that can prevent data recovery.

Overall, our results show that EDE is able to significantly speed-up workloads while maintaining a crash-consistent ordering. Indeed, by using EDE with IQ and WB rather than fences, we attain average workload speedups of 18% and 26%, respectively.

B. Issue Queue Throughput

In Figure 11, we show the distribution of the number of instructions issued each cycle. In the figure, each pattern represents the percentage of cycles the processor issued the labeled number instructions. As shown on the y-axis, cumulatively, the bars for each setup stack up to account for 100% of the simulated cycles. Note that, although our simulated architecture has a 3-instruction decode width, the issue queue has a width of 8.

As shown in the figure, all implementations issue 0 instructions in the majority of cycles. This is to be expected, as writes to NVM have a significant latency and can cause the pipeline to fill. It can be shown that, on average, the IPC of the applications is 0.40, 0.42, 0.46, 0.49, and 0.64 for the *B*, *SU*, *IQ*, *WB*, and *U* configurations, respectively.

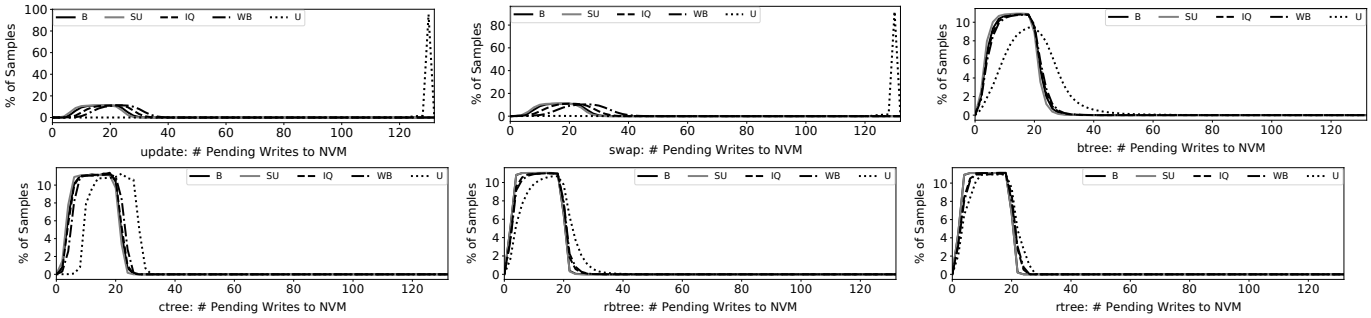

Fig. 10. Distribution of pending writes to NVM in the persistent 128-slot on-DIMM buffer in the NVM interface.

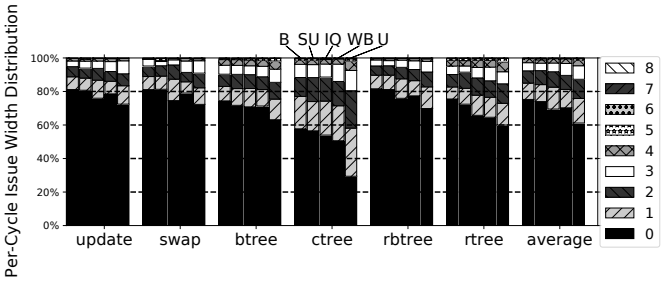

Fig. 11. Distribution of the number of instructions issued each cycle.

Across all the applications, *IQ* and *WB* spend fewer cycles being unable to issue instructions than *SU* and *B*. On average, *IQ* and *WB* both spend 30% of the cycles issuing instructions. However, *WB* in general issues more instructions during these active cycles. Specifically, when issuing instructions, *WB* is able to issue on average 8% more instructions than *IQ*. This is because *WB* does not block execution-dependent instructions at the issue queue, while *IQ* forces them to remain there until the execution dependences are satisfied.

C. Pending Writes to NVM

As described in Section VI, the NVM interface has a persistent 128-slot on-DIMM buffer that holds pending writes before they are merged into the NVM. In general, it is desirable to have a large number of pending writes in the buffer: it provides more opportunities for write coalescing, and also enables a higher writeback throughput. However, because fences in the processor pipeline stall the execution, it can be hard for crash-consistent applications to issue NVM writes quick enough to keep the buffer full.

Figure 10 shows, for each application and configuration, the distribution of the number of pending writes in the persistent 128-slot on-DIMM buffer. Each configuration is represented by a different type of line. The x axis shows the number of pending NVM writes in the buffer, while the y axis shows the percentage of samples with the observed number of pending NVM writes. A sample is taken each time a store reaches the NVM media.

Across all the applications, *U* has the highest number of pending NVM writes. This is because *U* does not issue fences, which stall writes. For the kernel applications, *U* is able to keep the buffer full, since the kernels write to NVM at a high frequency. In the PMDK applications, since other work must be done to maintain the data structures, the number of pending NVM writes is lower.

For the other configurations, although it is hard to see, *WB* has, on average, slightly more pending writes to NVM than the other configurations. As discussed in previous sections, since *WB* allows faster writebacks to NVM than the other configurations, this behavior is to be expected.

VIII. Future Work: Using EDE in Multi-threaded Applications

A. Example of Fence Overhead

While this paper focuses on the overhead of fences in NVM applications, fences also impose a significant overhead in many multi-threaded application domains. In such domains, fences are often used to enforce instruction orderings that guarantee data integrity.

For example, fences are needed to support memory reclamation in lock-free data structures. When many threads are concurrently viewing and modifying a lock-free data structure, a well-known problem is deciding when it is safe to reclaim the memory of an element that has been removed from the data structure. One popular solution is to use *hazard pointers* [38] to ensure that items are not prematurely freed. Hazard pointers are an efficient way for threads to announce which elements they are currently using. Before a thread can safely free an element, it must check all the values in active hazard pointers, to ensure that no reference to the element exists.

The key operation of hazard pointers is updating, or *announcing*, the element that a thread is about to access and that, hence, must be kept alive. Figure 12 shows how this announcement operation can be performed using AArch64. The code starts by moving into registers: (i) the pointer to the element's location (assumed in Line 1) and (ii) the hazard pointer (assumed in Line 2). Then, the element's location is retrieved (Line 3). Next, the element's location is stored in the hazard pointer to announce that this thread is using the element (Line 4). Afterwards, the pointer to the element's location is read again (Line 6), and checked to make sure the pointer still points to the original element location (Line 7). If not, this process must be repeated until the thread is able to successfully announce the element's location that it is about to access (Line 8). Note that since the only side effect of the announcement is to ensure that the announced location is not

```
1 // x1 contains ptr to element's location
2 // x2 contains hazard ptr
3 Loop: ldr x3, [x1] ; load element's location
4 str x3, [x2] ; announce element's location
5 dmb sy ; full fence: wait for announcement
6 ldr x4, [x1] ; load element's location again
7 cmp x4, x3 ; compare both locations
8 b.ne Loop ; try again if locations differ
```

Fig. 12. Hazard pointer announcement in AArch64.

freed, the algorithm is correct even if the thread temporarily announces an element that it does not end up using.

The key step of the algorithm is validating that the location announced did not change before the announcement was made. In order words, it must be guaranteed that the second load (Line 6) *happens after the announcement* (Line 4). Unfortunately, on both x86-64 and AArch64, enforcing this load-store dependence currently requires a full fence—a `DMB` instruction in AArch64 (Line 5).

Using EDE, it is possible to remove the full fence on Line 5. This can be achieved by using `str (1, 0), x3, [x2]` in Line 4 and `ldr (0,1), x4, [x1]` in Line 6.

B. Multi-Threaded Uses of EDE

The need to use fences to enforce orderings in multi-threaded applications extends well beyond the hazard pointer example described above. Indeed, all algorithms that rely on announcement-based solutions (e.g., [7], [12], [29], [39]) likely contain fences that could be eliminated via defining explicit execution dependences. The benefits of EDE will be especially pronounced in AArch64, where lock-free algorithms typically require many fences to guarantee that steps are not performed out of order.

In Java, EDE could also be used to enforce multi-threaded orderings required by the Java Memory Model [36], such as guaranteeing that `final` object fields are initialized before they are read by another thread. In addition, Java Virtual Machine (JVM) implementations traditionally store metadata alongside object fields, which also must be initialized before another thread can access the object. This coordination of small data subsets is an ideal target for EDE.

Another example where multi-threaded coordination of data is needed is in the kernel's use of circular buffers. Kernels often use circular buffers to store tracing and logging data collected throughout runtime. Ideally, the buffers should handle being read and updated by multiple threads concurrently. Via EDE, popping and pushing data from circular buffers can be performed in a lock-free manner without fences.

Finally, concurrent garbage collectors, such as ZGC [4] and Shenandoah [22] require careful orchestration of the movement of data alongside concurrent updates, class initialization, and dynamic code loading. In this environment, EDE can be used to minimize the overheads of garbage collection barriers.

Our future work involves evaluating these multi-threaded EDE use cases, plus additional important multi-threaded uses that we may discover.

C. EDE Support for Loads and Synchronization Primitives

This paper only discusses adding new EDE memory instruction variants to AArch64's store and cacheline writeback instructions. However, to support EDE use cases in multi-threaded environments, we also need to add new EDE instructions and hardware support for loads and synchronization primitives.

A detailed description of the EDE instruction definition and hardware support for loads and synchronization primitives can be found in [56].

IX. Future Work: Compiler Support for EDE

A. Integration in a Compiler's Internal Representation (IR)

As described in Section VI, in this paper we leverage EDE through the use of new built-in intrinsics added to Clang and LLVM. However, it is desirable to integrate EDE into the compiler more fully. Specifically, we believe that a compiler's internal representation (IR) can be augmented to incorporate execution dependences. For instance, in JVM compilers, a sea-of-nodes [15] representation is commonly used. Like Figure 5, this representation creates a dependence graph to record register, memory, and control dependences between instructions. It is straightforward to also introduce execution dependences into the representation. Doing so would allow compilers to perform aggressive optimizations without illegally reordering instructions with execution dependences.

For many of the use cases described in Section VIII, it is possible for a compiler to automatically create execution dependences during its initial IR generation. Similar automatic support would also be possible for persistent applications once persistency support is integrated into languages. It is also possible to expose EDE to framework developers via new ordering types for C/C++ atomics [1] and Java VarHandles [2], so that intrinsics are no longer needed.

Finally, by fully integrating EDE into compilers, it is possible for EDKs to be *virtualized* and for the compiler to automatically assign logical EDK values. Existing register allocation techniques such as graph coloring [13] and linear scan [62] are straightforward to repurpose for EDK assignments.

B. EDK Calling Convention

Along with integrating support for EDE into a compiler's IR, it is also necessary to establish a calling convention for EDE to ensure library compatibility. Normally, registers not used to pass parameters are divided into two categories: *caller-saved* and *callee-saved* registers. In a similar manner, for EDKs we introduce the concept of caller-saved and callee-saved *keys*.

For caller-saved keys, the caller must assume that the key will be overwritten within the callee function. Hence, for each caller-saved key K, after the function returns and before the next instruction that consumes K, a `WAIT_KEY (K)` instruction must be inserted. This `WAIT_KEY` ensures that the consumer of key K after the call return waits for the completion of the producer of key K before the call.

```
1  // X is caller-saved, Y is callee-saved
2  inst (X, 0), <ops>
3  inst (Y, 0), <ops>
4  call foo ; may overwrite EDKs
5  WAIT_KEY (X) ; waits on lines #2 & #9
6  inst (0, X), <ops> ; waits on line #5
7  inst (0, Y), <ops> ; waits on line #10
```

Caller function.

```
8  // function foo:
9  inst (X, 0), <ops>
10 inst (Y, Y), <ops> ; waits on line #3
```

Callee function.

Fig. 13. Correct EDK usage across function calls.

For callee-saved keys, the caller function performs no action. However, within the callee function, for each callee-saved key K, either (i) a `WAIT_KEY (K)` is inserted or (ii) all instructions that are a dependence producer of K must also be a dependence consumer of K. Once again, these instructions in the callee ensure that a consumer of key K after the call return waits for the completion of the producer of key K before the call.

Figure 13 shows an example of how to correctly maintain caller- and callee-saved keys across a function call. In the figure, `X` is a caller-saved key and `Y` is a callee-saved key.

X. Related Work

Past research has investigated the reorderings of read and write instructions that are allowed by different memory consistency models (e.g., [5], [48], [52]). Our work does not change the memory consistency model: EDE only selectively adds additional ordering constraints, in a manner that is much finer-grained than how fences do it.

Prior studies have measured the overhead of fences within relaxed architectures and found that there is notable opportunity to ease restrictions and improve performance [20], [21], [34], [35], [57], [59]. Some proposals [20], [21], [35] selectively enforce fences based on monitoring coherence messages. While this leaves the software unchanged, the hardware cost of tracking coherence messages is high. Other approaches [34] limit the scope of which subsequent instructions are affected by the fence. These techniques only cover a subset of EDE's behaviors.

Recent works focus on different persistency models (e.g., [30], [41], [42], [49]), and describe how the underlying hardware is allowed to reorder writes to NVM. One example relevant to our work is Strand persistency [25]. Strand persistency can only describe a subset of the orderings describable via EDE. Indeed, strand persistency is limited in that the instructions in a strand need to be contiguous in program order. EDE eliminates this restriction by supporting any instruction-to-instruction dependence, effectively supporting the equivalent of strands whose code is interleaved with one another.

Other works propagate writes to NVM through a designated buffer instead of the normal cache hierarchy [32], [41], [43]. EDE is orthogonal to buffered persistency and can complement systems that use it.

Several works propose to accelerate failure-atomic regions either in hardware [19], [25], [31], [44], [45], [51], [53], [54], [64] or software [6], [16], [17], [23], [47], [61], [63]. While many of the above works accelerate NVM logging, EDE provides a simple mechanism that does not alter the consistency model and can also be leveraged in uses cases outside of the persistency domain to accelerate multi-threaded workloads on AArch64. Hence, given EDE's multifaceted use cases, we believe that EDE is a compelling extension.

XI. Conclusion

Current ISAs are unable to describe an execution dependence between two instructions that have no register or memory dependence. Hence, programmers use fences and unnecessarily slow down execution. To remedy this, we proposed the Execution Dependence Extension (EDE), which allows for execution dependences to be encoded in the ISA.

We described two different hardware implementations of EDE, namely IQ and WB. Each one has different trade-offs in complexity versus performance potential. To evaluate EDE's impact, we implemented both IQ and WB in a simulator and ran several NVM applications. Overall, by using EDE with IQ and WB rather than fences, we attained average workload speedups of 18% and 26%, respectively.

EDE also has high potential to enable more aggressive reordering of memory accesses in multi-threaded codes. Hence, with EDE, one may be able to fully realize the performance potential of relaxed memory consistency models and lock-free algorithms. We are currently exploring these avenues.

Acknowledgment

This work was supported by NSF grant CNS 1763658.

References

[1] "C++ atomic types and operations." [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html

[2] "JEP 193: Variable handles." [Online]. Available: https://openjdk.java.net/jeps/193

[3] "Persistent Memory Development Kit." [Online]. Available: http://pmem.io/pmdk/

[4] "ZGC - OpenJDK Wiki." [Online]. Available: https://wiki.openjdk.java.net/display/zgc/Main

[5] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli, "The semantics of Power and ARM multiprocessor machine code," in *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 13–24. [Online]. Available: https://doi.org/10.1145/1481839.1481842

[6] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 439–451.

[7] J. H. Anderson and M. Moir, "Universal constructions for multi-object operations," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 184–193. [Online]. Available: https://doi.org/10.1145/224964.224985

[8] Arm, "Armv8-A architecture evolution," https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-evolution, 2016.

[9] Arm, "Arm Cortex-A72 MPCore processor: Technical reference manual," http://infocenter.arm.com/help/topic/com.arm.doc.100095_0003_06_en/cortex_a72_mpcore_trm_100095_0003_06_en.pdf, 2020.

[10] Arm, "Arm architecture reference manual Armv8, for Armv8-A architecture profile," https://static.docs.arm.com/ddi0487/fb/DDI0487F_b_armv8_arm.pdf, 2021.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[12] G. E. Blelloch and Y. Wei, "LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS," in *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, ser. LIPIcs, H. Attiya, Ed., vol. 179. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 5:1–5:17. [Online]. Available: https://doi.org/10.4230/LIPIcs.DISC.2020.5

[13] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47 – 57, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0096055181900485

[14] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 433–452. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660224

[15] C. Click and M. Paleczny, "A simple graph-based intermediate representation," in *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, ser. IR '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 35–49. [Online]. Available: https://doi.org/10.1145/202529.202534

[16] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 441–454. [Online]. Available: https://doi.org/10.1145/3297858.3304046

[17] N. Cohen, M. Friedman, and J. R. Larus, "Efficient logging in non-volatile memory by exploiting coherency protocols," *Proceedings of the ACM on Programming Languages (PACMPL)*, 2017. [Online]. Available: http://infoscience.epfl.ch/record/231400

[18] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589

[19] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for SCM with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 77–89.

[20] Y. Duan, N. Honarmand, and J. Torrellas, "Asymmetric memory fences: Optimizing both performance and implementability," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[21] Y. Duan, A. Muzahid, and J. Torrellas, "WeeFence: Toward making fences free in TSO," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[22] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '16. New York, NY, USA: ACM, 2016, pp. 13:1–13:9. [Online]. Available: http://doi.acm.org/10.1145/2972206.2972210

[23] E. Giles, K. Doshi, and P. J. Varman, "Hardware transactional persistent memory," *CoRR*, vol. abs/1806.01108, 2018. [Online]. Available: http://arxiv.org/abs/1806.01108

[24] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 46–61. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192367

[25] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *Proceedings of the 47th Annual International Symposium on Computer Architecture*, ser. ISCA 2020. ACM, 2020.

[26] W. W. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1496–1514, 1987.

[27] Intel, "Intel 64 and IA-32 architectures software developer's manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-\\instruction-set-reference-manual-325383.pdf, 2016.

[28] Intel, "3D XPoint: A breakthrough in non-volatile memory technology," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html, 2018.

[29] P. Jayanti and S. Petrovic, "Efficient wait-free implementation of multiword LL/SC variables," *Proceedings - International Conference on Distributed Computing Systems*, pp. 59–68, 07 2005.

[30] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 660–671.

[31] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 361–372.

[32] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[33] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[34] C. Lin, V. Nagarajan, and R. Gupta, "Fence scoping," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 105–116.

[35] C. Lin, V. Nagarajan, and R. Gupta, "Address-aware fences," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 313–324. [Online]. Available: https://doi.org/10.1145/2464996.2465015

[36] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: ACM, 2005, pp. 378–391. [Online]. Available: http://doi.acm.org/10.1145/1040305.1040336

[37] V. J. Marathe, A. Mishra, A. Trivedi, Y. Huang, F. Zaghloul, S. Kashyap, M. I. Seltzer, T. Harris, S. Byan, B. Bridge, and D. Dice, "Persistent memory transactions," *CoRR*, vol. abs/1804.00701, 2018. [Online]. Available: http://arxiv.org/abs/1804.00701

[38] M. M. Michael, "Hazard pointers: safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[39] M. Moir, "Practical implementations of non-blocking synchronization primitives," in *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 219–228. [Online]. Available: https://doi.org/10.1145/259380.259442

[40] D. R. Morrison, "Patricia—practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, p. 514–534, Oct. 1968. [Online]. Available: https://doi.org/10.1145/321479.321481

[41] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with WHISPER," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 135–148. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037730

[42] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 401–410. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151018

[43] T. M. Nguyen and D. Wentzlaff, "PiCL: A software-transparent, persistent cache log for nonvolatile main memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 507–519.

[44] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "SSP: Eliminating redundant writes in failure-atomic NVRAMs via shadow sub-paging," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 836–848. [Online]. Available: https://doi.org/10.1145/3352460.3358326

[45] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+redo logging for persistent memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 336–349.

[46] Oracle, "Oracle SPARC architecture 2011," https://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf, 2016.

[47] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 371–386. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915251

[48] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-TSO," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 391–407.

[49] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 265–276. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665712

[50] A. Raad, J. Wickerson, and V. Vafeiadis, "Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360561

[51] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 672–685. [Online]. Available: https://doi.org/10.1145/2830772.2830802

[52] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-TSO: A rigorous and usable programmer's model for X86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, p. 89–97, Jul. 2010. [Online]. Available: https://doi.org/10.1145/1785414.1785443

[53] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for NVM," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 178–190. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124539

[54] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 175–186. [Online]. Available: https://doi.org/10.1145/3079856.3080240

[55] T. Shull, J. Huang, and J. Torrellas, "AutoPersist: An easy-to-use Java NVM framework based on reachability," in *International Conference on Programming Language Design and Implementation (PLDI)*, June 2019.

[56] T. Shull and J. Torrellas, "Execution dependence extension (EDE): Enabling the potential of relaxed memory consistency models," http://iacoma.cs.uiuc.edu/iacoma-papers/isca21_2_tr.pdf, May 2021, Technical Report, University of Illinois at Urbana-Champaign.

[57] M. F. Spear, M. M. Michael, M. L. Scott, and P. Wu, "Reducing memory ordering overheads in software transactional memory," in *2009 International Symposium on Code Generation and Optimization*, 2009, pp. 13–24.

[58] Storage Networking Industry Association (SNIA), "NVM programming model v1.2," https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf, 2017.

[59] O. Trachsel, C. V. Praun, and T. R. Gross, "On the effectiveness of speculative and selective memory fences," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[60] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 91–104. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379

[61] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endow.*, vol. 7, no. 10, p. 865–876, Jun. 2014. [Online]. Available: https://doi.org/10.14778/2732951.2732960

[62] C. Wimmer, "Linear scan register allocation for the Java HotSpot Client Compiler," Master's thesis, Johannes Kepler University Linz, 2004.

[63] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 167–181. [Online]. Available: http://dl.acm.org/citation.cfm?id=2750482.2750495

[64] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 421–432. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540744