No Barrier in the Road: A Comprehensive Study and Optimization of ARM Barriers

Nian Liu†‡, Binyu Zang†‡, Haibo Chen†‡
† Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
‡ Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
{nianliu,byzang,haibochen}@sjtu.edu.cn

Abstract

In this paper, we present the first comprehensive performance characterization and optimization of ARM barriers on both mobile and server platforms. We draw a set of observations through several abstracted models and validate them in scenarios where barriers are intensively used. We find that (1) order-preserving approaches without involving the bus significantly outperform other approaches, and (2) the tremendous overhead mostly comes from barriers strictly following remote memory references. Usually, such barriers are inserted when threads are exchanging data, and they are used to ensure the relative order between storing the data to a shared buffer and setting a flag to inform the receiver. Based on the observations, we propose a new mechanism, *Pilot*, to remove such barriers by leveraging the single-copy atomicity to piggyback the flag with the data. Applying *Pilot* only requires minor changes to applications and provides 10%-360% performance improvements in multiple benchmarks, which are close to the ideal performance without barriers.

CCS Concepts • **Software and its engineering → Software performance**; **Mutual exclusion**; **Synchronization**; **Process synchronization**.

Keywords barrier, synchronization, concurrency, lock

1 Introduction

ARM processors have been widely used in embedded and mobile devices due to their lower price and higher energy-efficiency compared to x86 processors. Moreover, with the rapid increase of its computing power, ARM has become a competitive alternative to x86 in data centers [4, 17, 37, 38, 46, 47] and has already been deployed in Internet companies [22, 40]. One of the most significant differences between ARM and x86 is that ARM uses a weakly-ordered memory model (WMM) [2, 8], rather than total store order (TSO) [20, 34, 45] in many x86 processors. WMM does not guarantee the order between any pairs of non-dependent memory accesses. As a result, hardware barriers (also known as memory fences) should be carefully inserted to preserve the order between two non-dependent memory accesses. However, it is well-known that barriers may tremendously hurt the performance if it frequently appears on the critical path.

ARM provides various kinds of hardware barriers [2], but their performance characteristics have not been well-explored yet as the performance impacts of barriers are difficult to reason about due to the complexity of applications [41]. Besides, since the overhead and the performance differences among barriers are less evident in ARM mobile processors and those processors are not designed to handle heavy workloads [39, 44], the performance characteristics of barriers have been largely overlooked.

However, with the widespread use and promising future of ARM processors in server scenarios, we cannot let barriers become a barrier in the road of unleashing all potential performance. In this paper, we propose several abstracted models to eliminate irrelevant variables and study the performance characteristics of barriers. Two different kinds of ARM processors are considered in our study, including server processors and mobile processors, with threads placed either in different or the same NUMA node. Based on the study, we draw a set of observations and highlight some as follows.

With the increasing complexity of the bus architecture, the performance impact of barriers becomes more significant and dramatically varies. Since most ARM barriers are likely to be implemented with the bus involved, the increasing complexity of the bus architecture significantly enlarges the overhead of barriers and the performance variations among barriers. Therefore, in ARM servers, both the overhead of barriers and the choices of order-preserving approaches should be taken into serious consideration.

A determining factor in a barrier's overhead is the location where the barrier is inserted. Barriers strictly following remote memory references[1] (RMR) significantly

[1] An access to a shared object is a remote memory reference if the object is not cached or its cached copy is invalid [14].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6818-6/20/02...$15.00
https://doi.org/10.1145/3332466.3374535

reduce the benefits of load or store buffers and cause considerable overhead, even though some of them do not drain those buffers and do not block any following non-memory operations. However, the overhead is much smaller when barriers are away from the RMR.

Order-preserving approaches without involving the bus significantly outperform other approaches. Most ARM barriers are implemented to send ACE (AXI Coherency Extensions) barrier transactions to the bus and wait for the responses, and thus their overhead highly depends on the bus [2]. However, processors are able to maintain the order between load operations and program-order later memory accesses without involving the bus, given that they can recognize whether the load operations have finished. Hence, leveraging dependencies, as well as weak barriers, including DMB (Data Memory Barrier) ld and LDAR (Load-Acquire), provides significantly better performance.

While mostly a weaker barrier leads to a lighter overhead, sometimes store-release barrier (STLR) has a more substantial performance impact. According to our study, it mostly obeys the intuition that a weaker barrier does introduce a lighter overhead. The overhead caused by barriers can be concluded into the following list[3].

DSB > DMB full > DMB st > DMB ld ≈ LDAR ≥ Dep

The performance impact of STLR (Store-Release), however, is not stable. Sometimes, it introduces a more significant overhead compared with a stronger barrier (DMB full). According to our study, its overhead lies between DSB and DMB st.

We validate those observations in two different scenarios where barriers are intensively used [1, 29], including memory-based communications and synchronization primitives, and find that the substantial overhead caused by barriers mostly comes from those strictly following the RMR. Usually, such situations happen when threads are exchanging data, and a barrier is inserted to ensure the relative order between storing the data to a shared buffer and setting a flag to inform the receiver. To eliminate the overhead, we propose a new mechanism, *Pilot*, which targets at removing those performance-critical barriers by leveraging the single-copy atomicity to piggyback the flag with the data and broadcast both of them at once. We apply *Pilot* to both scenarios coved in the paper, and it provides 20%-360% performance improvements in multiple micro-benchmarks and 10%-60% improvements in PARSEC dedup benchmark [3] and various data structure benchmarks.

[2] In recent ACE5, processors are recommended to terminate barriers internally if the system is multi-copy atomic (MCA) for a similar reason [36].

[3] Dep in the list stands for dependencies; different options of DSB (Data Synchronization Barrier) such as DSB full, DSB st and DSB ld have a similar performance.

Thread 1	Thread 2
data = 23; flag = DONE;	while (flag != DONE); local = data;
Initial State	flag = BUSY data = 0
TSO Forbidden	local != 23
WMM Allowed	local != 23

Table 1. Different behaviors in TSO and WMM.

2 Background

In this section, we provide a brief background on ARM's WMM and order-preserving approaches under such model.

2.1 Weakly-ordered Memory Model

In TSO, only loads may be reordered with earlier stores to different locations [21]. However, in WMM, reordering of any non-dependent memory operations is permitted [2].

Table 1 is an instance where TSO and WMM behave differently. Thread 1 is supposed to transfer the value 23 to thread 2. It stores the value into a shared variable *data* and sets a *flag* to indicate that the *data* is ready. Thread 2 will spin on the *flag* until it is set to DONE and read the *data*. Since the orders between stores and between loads are preserved in TSO, thread 2 is guaranteed to observe the value 23 set by thread 1. However, in WMM, the *flag* may become observable before the *data* does, and thread 2 may load the *data* before ensuring the *flag* has been set to DONE. Therefore thread 2 is allowed to see a different value other than the value 23 wrote by thread 1, which is not the programmers' intention.

2.2 Order-preserving Options

ARM provides several ways to preserve the order between two memory operations under WMM as below [2, 28, 43].

Data Memory Barrier (DMB) prevents reordering of memory accesses across the barrier. Instead of waiting for previous accesses to become observable in the specified domain, DMB only maintains the relative order between memory accesses. Meanwhile, DMB does not block any non-memory access operations. DMB takes a parameter which specifies a domain it applies and the types of access, including any to any (DMB full[4]), store to store (DMB st), load to load/store (DMB ld), to which the barrier operates.

Data Synchronization Barrier (DSB) prevents reordering of any instructions across the barrier. DSB will make sure that all masters[5] in the specified domain can observe the previous operations before issuing any subsequent instructions.

[4] In previous works, DMB sy is used to represent the same meaning. However, DMB sy also implies the shareability domain of DMB. We use DMB full to describe only the types of access which it operates.

[5] A component that initiates transactions [1], which can be regarded as cores in this paper.

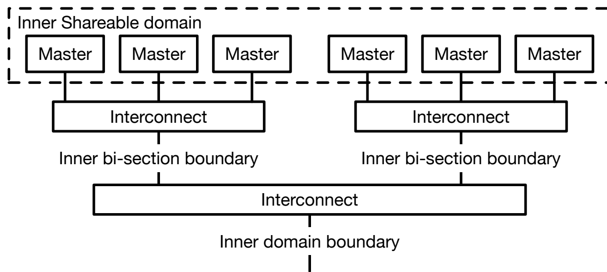


Figure 1. Boundaries in an ARM example system [1].

Since it blocks all the following instructions, DSB always introduces the most significant overhead among all approaches. DSB also takes the same parameters as DMB.

Load-Acquire (LDAR)/Store-Release (STLR) are a pair of one-way barriers introduced in ARMv8. LDAR blocks all following memory accesses until LDAR finishes; STLR ensures all loads and stores are observable before STLR.

Besides barrier instructions, dependencies also can preserve the order between memory operations.

Data Dependency (DATA Dep) exists when the value to be stored depends on the value loaded previously. By establishing a bogus data dependency (e.g., xor the loaded value with itself and add it to the value which will be stored), it can preserve the order between any load-store pairs.

Address Dependency (ADDR Dep) exists when the target addresses of the following memory operations depend on the value loaded previously. By constructing bogus address dependencies (e.g., xor the loaded value with itself and add it to the target addresses of following memory operations), it can preserve the order between any load-load/store pairs.

Control Dependency (CTRL) can preserve the order between a load to program-order-later store operations in the conditional branch when the loaded value is used to compute the condition [28]. However, when preserving the order between load operations, an Instruction Synchronization Barrier should be inserted along with the control dependency **(CTRL+ISB)** to flush the pipeline and ensure that the following loads are executed after the condition is satisfied. Constructing bogus control dependencies can preserve the order between any load-load/store pairs.

2.3 Barriers from a Hardware Perspective

There are four different domains in an ARM system, including non-shareable, inner shareable, outer shareable, and system domains. For each domain, there are two different boundaries: bi-section boundary and domain boundary. Bi-section boundary is downstream of a subset of master components, and domain boundary is downstream of all master components. Figure 1 shows both boundaries in an ARM example system. We only focus on the inner shareable domain (ish) in this paper, which contains all cores in the system.

Since ARM only defines the behavior, any implementations followed the specification are allowed. We provide one typical implementation as below. When a barrier instruction reaches the issue queue of the ARM processor, it blocks different types of subsequent instructions according to the type of the barrier. Then the barrier instruction is sent to the load-store unit. Barriers that require the assistance from the bus issue an ACE barrier transaction. There are two different barrier transactions. DMB and DSB would normally translate to memory barrier transaction and synchronization barrier transaction, respectively. Before receiving the response for the barrier transaction from the bus, the barrier instruction cannot retire, and the subsequent instructions cannot be issued. However, weaker barriers like DMB ld and LDAR are likely to be implemented without sending anything to the bus as the processors can identify whether loads have finished without involving the bus [36].

2.4 Usages of Barriers

There are three primary uses of the barriers as below [1, 29].

Memory-based communication is achieved by first writing the data and then setting a flag to indicate that the data is available. A barrier should be added to ensure that all receivers are able to observe the data before or at the same time when the flag is observable. Memory-based communication can be used to implement the producer-consumer model and the lock-free data structures. This part will be discussed in Section 4.

Synchronization primitives (e.g., mutex locks) are widely utilized in applications as they provide more transparent and straightforward semantics. Unlike the locks' implementations in x86, barriers should be carefully inserted in both lock and unlock procedures to guarantee the correctness. Those barriers are likely to introduce a substantial overhead. This part will be discussed in Section 5.

Device drivers also require barriers to preserve the order. The host communicates with peripheral devices through sideband signaling communication by generating a signal to indicate that the data is available. DSB is inserted to ensure that the data is observable before the signal arrives. Since such situations are difficult to optimize and highly depend on the devices, we will not discuss it in this paper.

3 Performance Characteristics of Barriers

It is difficult to study the performance characteristics of barriers directly from real-world applications due to its complexity. Barriers may affect applications indirectly (e.g., the contentions to shared objects). To eliminate irrelevant variables, we introduce several abstracted models in this section.

3.1 Target Platforms

Since the instruction set only defines the behavior for correctness but not imply any performance characteristics, which

Name	Kirin960	Kirin970	Raspberry Pi 4	Kunpeng916
Architecture	Cortex A-73 Cortex A-53	Cortex A-73 Cortex A-53	Cortex A-72	Cortex A-72
Cores	4 + 4	4 + 4	4	2 x 32
Frequency (GHz)	2.1	2.36	1.5	2.4
Interconnect	ARM CCI-550	ARM CCI-550	Unknown	Hydra Interface
OS	Linaro 4.14	Debian 4.9.78	Ubuntu 5.3.0	Debian 4.9.20

Table 2. Target Platforms

can only be explored through specific implementation. To make the results general enough, we have carefully choose several platforms, including kirin 960/970, which has been widely used in successful commercial products (e.g., used in hundreds of millions smartphones), raspberry pi 4, which is a popular embedded system, and kunpeng 916, which is one of the most advanced ARM server available in the market and has been widely deployed in data centers. The specifications of the four target platforms are detailed in Table 2.

3.2 Abstracted Models

Programs which use barriers on the critical path can be abstracted into a loop containing barrier instructions along with other instructions. They vary in three ways: the occurrence frequency of barriers, the memory operations around the barriers and the choice of barriers. The occurrence frequency of barriers decides how barriers influence overall performance. The type and number of memory operations, as well as the statuses of the target cache lines, determine the overhead of barriers. Finally, the different choices of barriers lead to different performance impacts. We study the performance characteristics of barriers by analyzing several groups of abstracted models which diverse in these ways.

Algorithm 1: Assembly Code of Abstracted Models

```
1  Loop:
2      add x0, x0, 64
3      add x1, x1, 64
4      ldr/str x3, [x0]
5      BARRIER_LOC_1
6      NOPs
7      BARRIER_LOC_2
8      ldr/str x4, [x1]
9      add x2, x2, 1
10     cmp x2, BUFSIZE
11     ble Loop
```

We implement the abstracted models in assembly. As shown in Algorithm 1, a thread loads or stores values to different cache lines in line 4 and line 8. Besides choosing different kinds of barriers, we also consider whether the barrier follows strictly after memory operations. We provide two alternative locations where barriers can be inserted, marked as BARRIER_LOC_1 and BARRIER_LOC_2 in line 5 and line 7. Meanwhile, NOPs in line 6 are used to simulate different occurrence frequencies of barriers. Since barriers are primarily used to achieve synchronization, target cache lines of memory accesses around barriers are likely to be frequently transferred among cores. To simulate this, we create two threads which execute the code one by one exclusively and bind them to different cores in the system. We also provide sufficient buffer to restrict the performance bottleneck to the loop in Algorithm 1.

Firstly, we remove all memory operations to explore the intrinsic overhead of barriers in Section 3.3. Then we classify those models into two categories according to whether the bus is involved in Section 3.4 and Section 3.5.

3.3 Intrinsic Overhead of Barriers

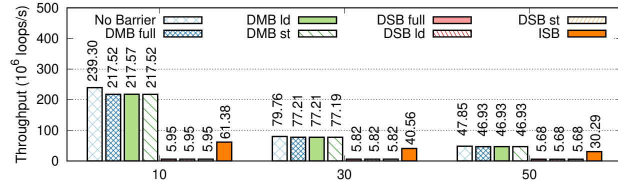
We study the performance impacts from barriers themselves by removing all memory operations on the critical path. Figure 2 shows the throughput of the abstracted model with barriers appearing at different frequencies. Similar properties are found in kirin960/970, raspberry pi 4 and kunpeng916 and can be concluded into the following observation.

Observation 1: *The intrinsic overhead of barriers is stable and intuitive.*

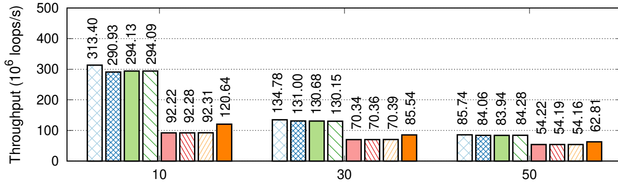
DMB provides the lightest overhead as it does not block any subsequent non-memory operations. ISB flushes the pipeline and thus has a larger overhead. DSB causes substantial overhead as it blocks all the upcoming instructions before receiving the response from the bus. Since there is no memory operation around the barrier, DMB and DSB with different options do not lead to different performance. In summary, the substantial and diverse performance impacts of barriers are the outcome along with memory operations around rather than from barriers themselves.

3.4 Order-preserving with the Bus Involved

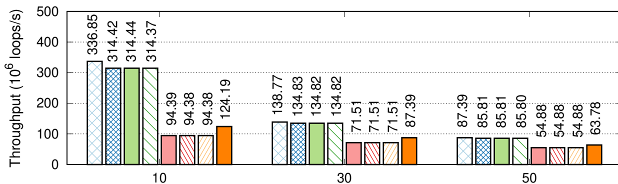
Since the store buffers of ARM processors are allowed **not** to provide order guarantees, processors cannot identify whether stores are committing in a FIFO order. Thus, when the order between stores and following memory accesses should be preserved, processors are likely to achieve this



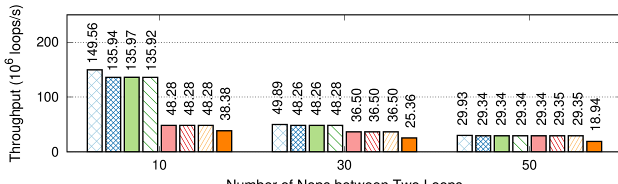
(a) Kunpeng916



(b) Kirin960

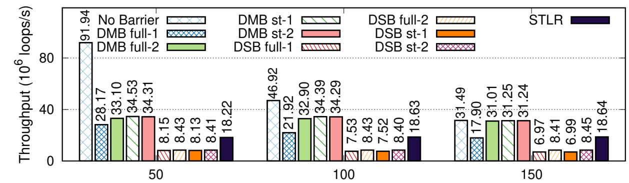


(c) Kirin970

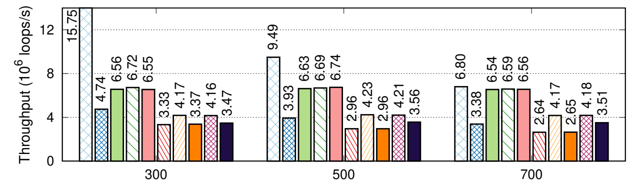


(d) Raspberry Pi 4

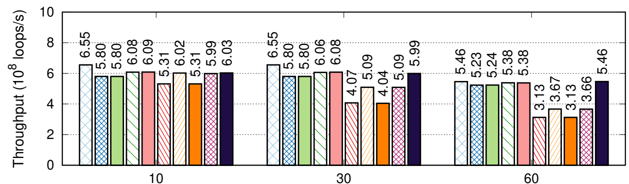
Figure 2. Throughput of the abstracted model which does **not** contain any memory operations. Barriers in the legend are placed on the critical path.



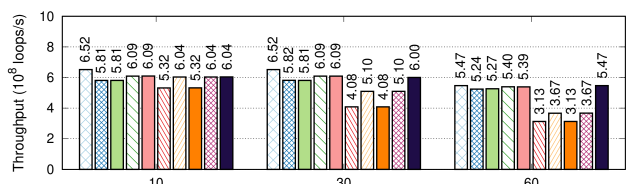
(a) Threads are bound to the **same** NUMA node in kunpeng916.



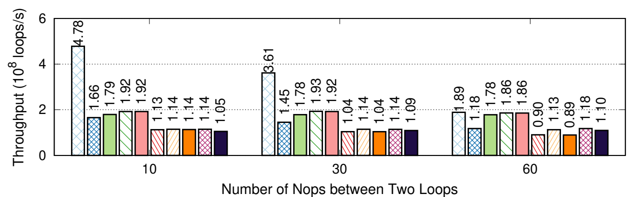
(b) Threads are bound to **different** NUMA nodes in kunpeng916.



(c) Threads are bound to the big core cluster in kirin960.



(d) Threads are bound to the big core cluster in kirin970.



(e) Threads are bound to different cores in raspberry pi 4.

Figure 3. Throughput of the abstracted model which contains two store operations under different configurations.

goal by sending ACE barrier transactions to the bus[6]. Such a situation happens when maintaining the order from store to program-order later memory operations. In this section, a specific model is discussed to represent this scenario without loss of generality. Store operations towards different cache lines are added to the critical path in line 4 and line 8 of Algorithm 1, and the barrier protects the order among them.

Figure 3 shows the throughput of the model under different configurations. In kirin960/970, both threads are bound to the big cluster, given that the processors' architectures of different clusters are not identical. X-1 and X-2 in the figures represent the cases when barrier X is inserted at BARRIER_LOC_1 and BARRIER_LOC_2 respectively. The following observations are derived from the result.

[6] Other implementations are allowed if they obey the specification, but may lead to a larger performance penalty.

Observation 2: *A determining factor in a barrier's overhead is the location where the barrier is inserted.*

Barriers which follow strictly after the RMR are likely to introduce a substantial overhead. When using DMB full, DSB full and DSB st in kunpeng916, a considerable performance variation exists between X-1, where barrier follows strictly after an RMR, and X-2, where numerous of nops are inserted

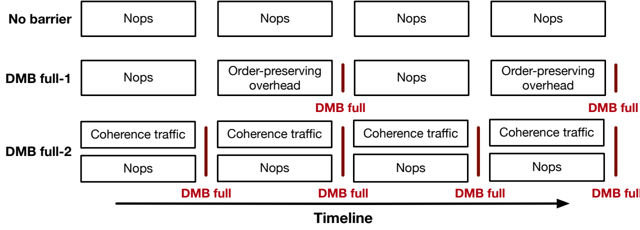


Figure 4. Timeline of the tipping point when the execution of nop instructions entire hide the barrier's overhead.

between the RMR and the barrier. The performance differences are much smaller in kirin960/970 and raspberry pi 4 due to the simpler bus architecture.

An intuitive explanation for this phenomenon is that before receiving the response from the bus, DMB and DSB slow down or even block the following instructions. The response will not be sent until previous snoop transactions have finished and the corresponding ACE barrier transaction has reached the specified boundaries [1]. It takes considerable time, especially when cross-node snooping is involved, which will expose on the critical path if the upcoming instructions cannot execute concurrently. Previously, store buffer is introduced to reduce the problem. However, the barriers which strictly follow the RMR reduce the benefit of store buffer and expose a massive overhead even though barriers like DMB do not drain the store buffer.

To validate that barriers do influence the execution of nops, we consider a tipping point where the performance of *DMB full-2* is the same as *No Barrier* when the execution of nop instructions entirely hides the barrier's overhead as shown in Figure 4. And *DMB full-1* should be exactly one half the performance of *DMB full-2* under such situation. Such a situation happens when there are 150 nops or 700 nops as shown in Figure 3(a) and 3(b), the performance ratio of *DMB full-1* to *DMB full-2* happens to be $\frac{17.90}{31.01} \approx \frac{3.38}{6.54} \approx \frac{1}{2}$.

But as specified in ARMv8's manual [19] and validated in Section 3.3, DMB does **not** block any non-memory operations. Since the actual hardware is vendor-defined, one possible explanation is that DMB may cause some performance bottlenecks in the pipeline (e.g., saturating the reorder buffer) and indirectly influences the execution of nops.

However, DMB st does not have such property. DMB st provides weaker semantic and cumulative properties, which may lead to a more radical implementation. It should be noted that DMB st still introduces tremendous overhead as it blocks the following store operations, especially when threads are bound to different NUMA nodes.

Observation 3: *Store-release barrier, however, does not always outperform stronger barriers.*

Store-release barrier (STLR) does not perform well in kunpeng916 and raspberry pi 4, even though it provides a weaker semantic than DMB full. Such a phenomenon has been observed for a bunch of times in our other experiments. According to the results of our experiments, the overhead of STLR is not stable and lies between DSB full and DMB st. As no accurate description about the implementation of STLR is available, performance comparison with DMB full is needed before using STLR.

Observation 4: *With the increasing complexity of the bus architecture, the performance impact of barriers becomes more significant and dramatically varies.*

We have to pick a much smaller number of nops in Figure 3(c), 3(d) and 3(e) so that we can distinguish the performance variations among barriers in kirin960/970 and raspberry pi 4. Moreover, the performance difference between *No_barrier* and other cases are less evident, which indicates that the overhead of barriers is much smaller in mobile processors. Since barriers in this section require bus involving, the main reason behind this is that the mobile processors have simpler bus architecture. Thus in ARM servers, both the overhead of barriers and the choices of order-preserving approaches should be taken into serious consideration.

Observation 5: *Crossing nodes is a killer.*

When two threads are bound to the same NUMA node, a much smaller number (150) of nop instructions is required to hide the overhead of DMB full. In such a situation, the memory barrier transaction only needs to reach the inner bi-section boundaries, which is the boundary of a part of the domain, and wait for the snooping transactions to finish before sending the response [1]. Thus, DMB full has a much smaller overhead when no cross-node snooping are involved. However, DSB does not benefit from the locality as the synchronization barrier transaction has to reach the inner domain boundary, which is the boundary of the whole domain, before the response can be sent. Therefore, the performance variation between DSB and DMB dramatically increases when threads are bound to one NUMA node.

Implications. In conclusion, DMB st is the best choice when preserving the order between store operations, as it introduces the minimum performance impact. However, when a stronger semantic is needed, DMB full should be kept away from the RMR by either avoiding cache misses or separating them with other operations for better performance. Even though STLR provides weaker semantic over DMB full, it may lead to a heavier overhead. Performance comparison against DMB full is required before using STLR.

3.5 Order-preserving without the Bus Involved

When the order between loads and following memory operations should be preserved, the processor is able to identify whether loads have finished without involving the bus. Other than dependencies, barriers with weaker guarantees (DMB ld and LDAR) are also likely to be implemented without sending anything to the bus [36]. In this section, we choose

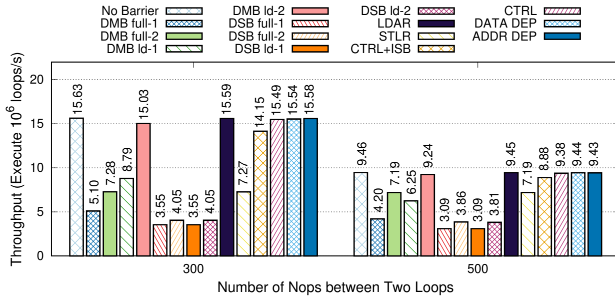


Figure 5. Throughput of the abstracted model, which contains a load and a store operation, and threads are bound to **different** NUMA nodes in kunpeng916.

<table>
<tr><th>From / To</th><th>Load</th><th>Loads</th><th>Store</th><th>Stores</th><th>Any</th></tr>
<tr><td>Load</td><td rowspan="2">A Dep or LDAR[1]</td><td rowspan="2">A Dep or DMB ld[1]</td><td colspan="2" rowspan="2">DMB full</td><td rowspan="2">DMB full</td></tr>
<tr><td>Loads</td></tr>
<tr><td>Store</td><td rowspan="2">A/D/C Dep or LDAR[1]</td><td rowspan="2">A/D/C Dep or DMB ld[1]</td><td colspan="2">DMB st</td><td>DMB full[2]</td></tr>
<tr><td>Stores</td><td colspan="2" rowspan="2">DMB full</td><td rowspan="2">DMB full</td></tr>
<tr><td>Any</td><td>A Dep or LDAR[1]</td><td>A Dep or DMB ld[1]</td></tr>
</table>

[1] A/D/C represent address/data/control dependencies, respectively. Though dependencies provide better performance even when preserving order between multiple memory accesses, LDAR/DMB ld outperforms other barriers when it is difficult to construct dependencies. ARMv8.3 provides a new Load-Acquire RCpc barrier, which is not supported by kunpeng916 but may provide better parallelism than LDAR here.

[2] STLR can be used here. Performance comparison against DMB full is needed before using STLR.

Table 3. Suggestions for selecting different order-preserving approaches under different scenarios.

a specific model to represent such scenario without loss of generality. A load in line 3 and a store towards a different cache line in line 8 of Algorithm 1 are added.

Figure 5 shows the throughput when two threads are bound to different NUMA nodes of kunpeng916. Results from other configurations are consistent with our observations and omitted due to the space limit. As shown in the figure, DMB full/ld and DSB full/ld strictly following after the RMR (X-1) introduce larger overhead than those do not (X-2), which supports **observation 2**. Moreover, STLR does not outperform stronger DMB full as we revealed in **observation 3**. Since the order can be preserved without involving the bus, we obtain a new observation below.

Observation 6: *Order-preserving approaches without involving the bus significantly outperform other approaches.*

Constructing bogus data/address/control dependencies provide fine-grained protection and bring no harm to the parallelism. Meanwhile, nothing will be sent to the bus. Thus, they introduce almost no overhead. However, since ISB flushes the pipeline, CTRL+ISB introduces some overhead when preserving the order between loads. Besides those dependencies, barriers which have similar semantic, such as DMB ld and LDAR, provide similar performances as they are also likely to be implemented without involving the bus.

Implications. When preserving the order from loads to program-order later memory operations, constructing bogus dependencies provides the lowest overhead and bring no harm to the parallelism. When it is difficult to do so, LDAR and DMB ld outperforms other barriers and can be used here.

We conclude the implications in Table 3.

4 Characterizing and Optimizing Barriers in Memory-based Communications

We validate our observations in two different scenarios where barriers are intensively used. Firstly, we focus on the barriers used in memory-based communications.

4.1 Barriers in Producer-consumer Model

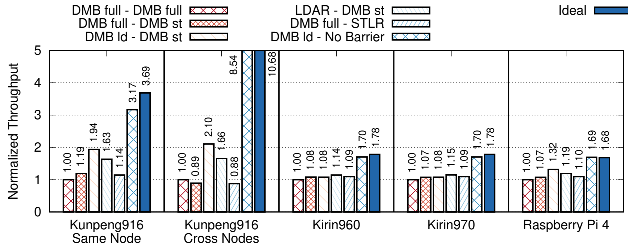
Producer-consumer model is one of the most typical use cases of memory-based communications and is essential in the performance of pipeline parallelism [3, 15]. Algorithm 2 shows the implementation of the producer in a single-producer single-consumer model. Locks are required when multiple producers or consumers using the same circular buffer, which will be further discussed in Section 5.

Algorithm 2: Producer Implementation

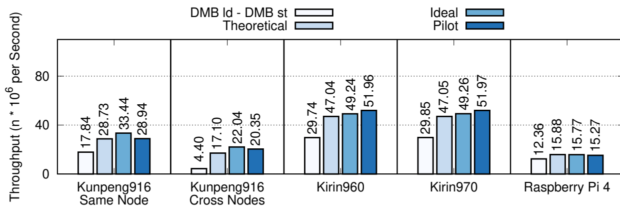
```
  Data: Shared: prodCnt = 0, consCnt = 0, buffer;
1 while prodCnt - consCnt = BUFF_SIZE do
2     nop;
3 Barrier;
4 buffer[prodCnt % BUFF_SIZE] ← produceMsg();
5 Barrier;
6 prodCnt ← prodCnt + 1;
```

The producer waits until there are some available buffers in line 1 & 2. Barrier in line 3 guarantees that all following loads and stores happen after ensuring the buffer is available. It should be a full barrier if any store is supposed to happen before (e.g., messages are passing by reference). Otherwise, a weaker load barrier is enough when directly passing a value. The producer then fills the buffer in line 4. Since the buffer is shared with the consumer, this operation is likely to be an RMR. Another barrier is added in line 5 to preserve the order between filling the buffer and informing the consumer.

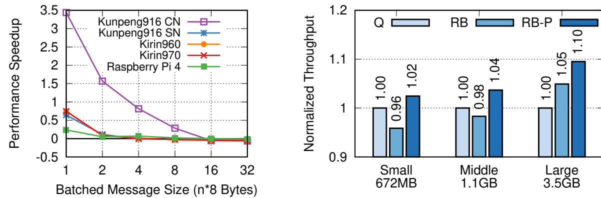
The implementation of the consumer is similar and thus omitted. The consumer loads the counter, reads the message after ensuring the message is available, and updates its counter. Since light-weighted load barriers or dependencies can preserve the order between those accesses in the consumer, we focus on the performance impact of barriers in the producer by placing enough nops in **produceMsg** and



(a) Normalized throughput of producer-consumer model under different configurations. *X* - *Y* in the legend indicates the barriers used in line 3 and line 5 of Algorithm 2 respectively. *Ideal* directly removes all barriers, which leads to a wrong result but can serve as a reference.



(b) Throughput of producer-consumer model after applying *Pilot* under different configurations.



(c) Performance speedup when batching more messages.

(d) Normalized compress speed of PARSEC dedup under different workloads.

Figure 6. Barriers in producer-consumer model.

providing sufficient buffers to ensure that consumers have enough time dealing the messages. Thus, we only change the barriers in the producer in the following experiments.

4.2 Performance Impacts of Barriers in Producer-consumer Model

The producer and the consumer are bound to different cores in the same or different NUMA nodes in kunpeng916, the big core cluster in kirin960/970 and cores in raspberry pi 4. Results shown in Figure 6(a) support our previous observations. Firstly, the combination *DMB ld - DMB st* or *LDAR - DMB st* has the best performance, which supports **observation 6**. And as revealed in **observation 3**, when binding threads to different NUMA nodes in kunpeng916, STLR does not outperform stronger DMB full. Both the overhead and the performance variations among barriers are much more noticeable in kunpeng916 than in kirin960/970 and raspberry pi 4 as we conclude in **observation 4**. Finally, as we point out in **observation 2 & 5**, the tremendous overhead is **mostly** coming from the barrier in line 5 which strictly follows the RMR in line 4, and directly removing the barrier (*DMB ld - No Barrier*) provides significant performance improvements, which are close to *Ideal*, in all platforms. Note that even when using DMB st, which does not influence the non-memory operations as DMB full does in observation 2, an enormous performance impact (5x) still exists when crossing NUMA nodes.

4.3 Using *Pilot* to Remove the Fatal Barrier

Since the substantial overhead is caused by the barrier strictly following after an RMR, we propose a new mechanism, *Pilot*, to remove the barrier as it does in the train.

Pilot leverage the 64-bit single-copy atomicity provided by ARM aarch64 to piggyback the flag with the message and broadcast them in a single atomic store. Theoretically, it causes a loss of information since at least one value should be used to represent the flag. This problem can be solved by adding a fallback mechanism. Algorithm 3 & 4 show the implementations of sender and receiver after applying *Pilot*.

Algorithm 3: *Pilot* Sender Side Implementation

Data: Shared: $flag = 0, data = 0;$
Local : $newData, oldData = 0, cnt = 0;$
Const : $hashPool;$

1 $newData \leftarrow newData$ ^ $hashPool[cnt ++ \% \text{SIZE}];$
2 **if** $newData = oldData$ **then**
3 $\quad flag \leftarrow flag$ ^ $1;$
4 **else**
5 $\quad data \leftarrow newData;$
6 $\quad oldData \leftarrow newData;$

In line 1 of Algorithm 3, the sender shuffles the newly produced data by doing an XOR with a prepared seed in *hashPool* to reduce the possibility of collision with previous data. Line 2 and line 3 are a fallback mechanism to deal with the corner case when data is still the same as the prior data even after shuffling. But typically, the newly produced data is different from old data so that we can directly inform the receiver by setting the shared *data* into the newly produced one in line 5. The sender then stores the current data into a local buffer which will be used in the next round in line 6.

In Algorithm 4, the receiver spins in line 1-4 and keeps checking the value of the shared *data* and the shared $flag$. When *data* has been set to different values in line 1, it means that a new data has been received. When $flag$ has been set to different values in line 2, it means that the newly produced data has the same value as the prior data even after shuffling. The receiver then stores the new $flag$ in line 3. In both cases, the newly produced data has been stored in *data*. The receiver returns the transferred data by doing XOR with the same prepared seeds with the sender in line 6.

Algorithm 4: *Pilot* Receiver Side Implementation

```
  Data: Shared: flag = 0, data = 0;
        Local  : oldFlag = 0, oldData = 0, cnt = 0;
        Const : hashPool;
1 while data = oldData do
2   if flag ≠ oldFlag then
3     oldFlag ← flag;
4     break;
5 oldData ← data;
6 return oldData ^ hashPool[cnt ++ % SIZE]
```

The correctness of *Pilot* is guaranteed by the single-copy-atomicity. The system exists two correct state according to whether the newly-produced data is globally visible. The sender can make the data globally visible through one single store operation. When the newly-produced data is not the same as the old one, the store operation is towards the shared data. Otherwise, it is towards the shared flag. The single-copy-atomicity guarantees that the value can be visible all-together. Thus the system will fall into one of two correct state at any time.

4.4 Applying *Pilot* to Producer-consumer

Applying *Pilot* to both producer and consumer can remove the barrier which causes the substantial overhead in line 5 of Algorithm 2, and the corresponding load barrier in the consumer. Since the amount of the buffers is limited, we still need a shared counter and the barrier in line 3 of Algorithm 2 to ensure the correctness. However, the overhead of this barrier is less crucial as shown in Figure 6(a).

4.5 Evaluation of *Pilot* in Producer-consumer

Although *Pilot* is relatively straightforward, it brings a considerable performance improvement, which comes from two aspects. **Firstly**, the barrier which follows strictly after the RMR and causes a tremendous overhead is removed. **Secondly**, *Pilot* reduces the number of touched cache lines. Both of them contribute to the overall performance improvement.

In Figure 6(b), we compare the throughput after applying *Pilot* with the original implementation which has the best performance (DMB ld - DMB st). Moreover, we also include the *Theoretical* performance after applying *Pilot* by removing the corresponding barriers which *Pilot* avoids and the *Ideal* performance by directly removing all barriers.

Comparing with the original implementation, applying *Pilot* introduces 62%, 363%, 75%, 74% and 24% performance improvements under different environments. The performance improvements from *Theoretical* to *Pilot* are owing to the reduction of the touched cache lines, which is more noticeable when transferring data across NUMA nodes. Moreover, since the overhead of other barriers is less significant, *Pilot* achieves similar performance with *Ideal.*

When transferring more than 64-bit data, *Pilot* can be applied to every 64-bit-long slice of data. Figure 6(c) shows the performance improvements when more messages are batched. The improvement declines with the increasing length of the data array since barriers appear less frequently and multiple slices of data share the overhead from one barrier. Even so, the performance improvement is still significant when crossing NUMA nodes. Since the additional procedures in *Pilot* are all local operations, it does not introduce a noticeable overhead even in the worst case ($< 5\%$).

Finally, we apply *Pilot* to PARSEC dedup [3] benchmark, which uses pipeline parallelism to compress files. The original lock-based queue, which serves as a communication buffer among different stages in dedup, is replaced with a lock-free ring buffer in our experiments. Applying *Pilot* to the ring buffer provides 1.8x and 2.2x speedups in micro-benchmark when threads are bound to the same or different NUMA nodes of kunpeng916, respectively. Since file I/O is dedup's performance bottleneck [7, 23], we remove the file operations and gather the output in memory to focus on the improvements in communications among stages. As shown in Figure 6(d), the native ring buffer (RB) sometimes has worse performance as it increases the contention over the communication buffer. But after applying *Pilot* (RB-P), a 10% speedup is achieved compared with the original lock-based queue (Q).

However, *Pilot* cannot be applied to lock-free data structures. Flags in those cases always have more complex meanings (e.g., a pointer which points to a new object) and cannot be easily combined with the transferred data.

5 Characterizing and Optimizing Barriers in Synchronization Primitives

Another scenario where barriers are intensively used is to implement synchronization primitives. In this section, we focus on the mutex lock, which is one of the most typical synchronization primitives.

5.1 Barriers in Mutex Locks

Mutex locks can be divided into in-place locks and delegation locks according to where the critical section executes [48]. In-place locks are implemented by waiting on shared variables before entering the critical section, such as ticket lock and MCS lock [30, 30]. It has been widely used in the existing systems, including the Linux kernel [9], as it is simple but efficient in most cases, especially at low contention levels. Barriers are needed in both lock and unlock procedure of in-place lock to ensure that all memory accesses in the critical section do not reorder with lock and unlock operations.

Delegation lock, however, is aimed at improving cache locality and eliminating lock acquisitions and releases from the critical path [32] by choosing a lock server that handles all the critical sections. Previous works [13, 14, 18, 25, 26, 35,

42, 48] have proposed various delegation locks which can be classified into two different types. The first kind uses a dedicated core to serve as a lock server, including RCL [25, 26] and FFWD [42]. The second kind picks one lock competitor and upgrades it to the lock server at run time. SAML [48], flat-combining lock [18] and CC/DSMSynch [14] are locks of this kind. Dedicated lock servers can provide better performance at high contention levels while migratory sever locks are more flexible and can be deployed in more scenarios.

Algorithm 5: Delegation Lock Server

Data: Shared: *reqPool*, *respPool*;
Local : *core*, *req*, *resp*, *oldFlagPool* = 0;

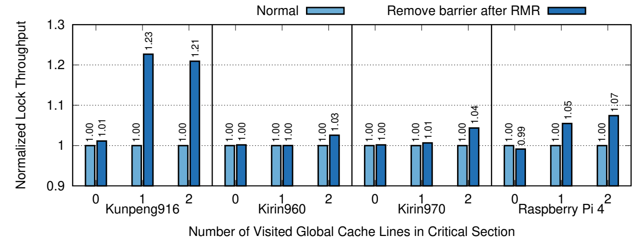
1 $req \leftarrow reqPool[core]$;
2 **if** $req \text{ -> } flag \neq oldFlagPool[core]$ **then**
3 $\quad oldFlagPool[core] \leftarrow req \text{ -> } flag$;
4 $\quad$ Barrier;
5 $\quad resp \leftarrow respPool[core]$;
6 $\quad resp \text{ -> } ret \leftarrow req \text{ -> } criticalSection(req \text{ -> } arg)$;
7 $\quad$ Barrier;
8 $\quad resp \text{ -> } flag \leftarrow resp \text{ -> } flag \wedge 1$;

Algorithm 5 is the pseudo-code of delegation lock server. The server reads the request in line 1, executes the critical section for the client in line 6, and writes the response in line 8. Barriers used to maintain the order between the modifications to the shared variables are no longer needed in delegation lock, as all critical sections execute in one lock server and thus the order is guaranteed by hardware. However, barriers are still needed in delegation locks (line 4 and line 7) to ensure the correctness.

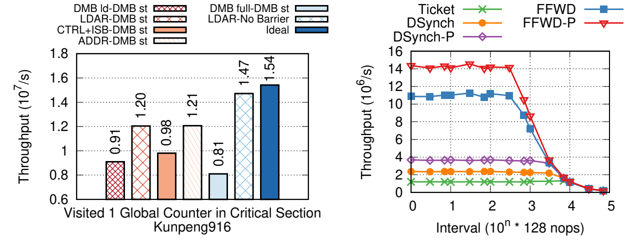
We implement the ticket lock by referring to the Linux kernel's implementation and implement FFWD [42] in liblock [25]. We design a micro-benchmark which creates various threads competing for the same lock. In the critical section, threads read and modify a specified number of shared cache lines and increment a counter. For ticket lock, those counters are all local variables and will be collected at the end to calculate average throughput. And for FFWD, there is only one global counter as all critical sections are executed in the lock server. After releasing the lock, different amount of nops are inserted to simulate different contention levels.

5.2 Performance Impacts of Barriers in Mutex Locks

We create 63 threads and bind them to individual physical cores in kunpeng916, 4 threads in the big core cluster in kirin960/970 and 4 threads in raspberry pi 4. The results of delegation locks in mobile processors are omitted as it is hard to saturate the lock server due to its limited number of cores.



(a) Normalized throughput of ticket lock. The barrier in the unlock procedure is set to the one in the legend.



(b) Throughput of delegation lock. *X-Y* indicates *X* is used in line 4 and *Y* is used in line 7 of Algorithm 5.

(c) Performance comparison after applying *Pilot*.

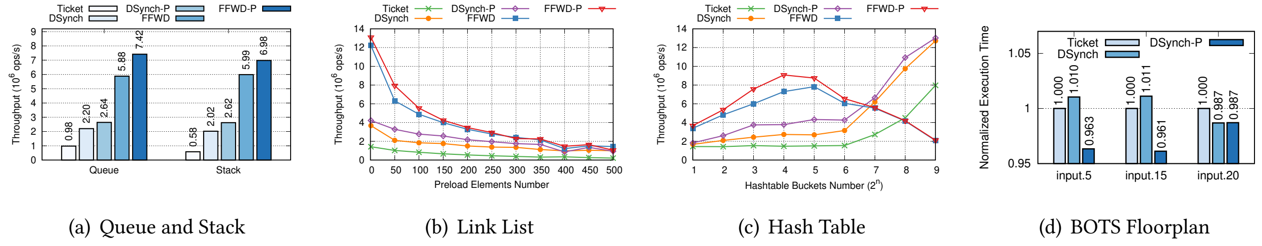
Figure 7. Barriers in mutex locks.

Again, the micro-benchmark results in Figure 7(a) & 7(b) validate our observations. Firstly, as revealed in **observation 6**, DMB ld/LDAR or dependencies have better performance in delegation lock. And as we conclude in **observation 4**, the overhead of barriers is more noticeable in kunpeng916 than in mobile processors. Finally, when visiting several global cache lines in the critical section, the barrier in the unlock procedure of in-place lock follows strictly after an RMR, and thus its overhead becomes evident (23%) as we point out in **observation 2**. Such tremendous overhead also exists in delegation lock as the barrier in line 7 follows strictly after the RMR in line 6 of Algorithm 5. Directly removing the barrier (*LDAR-No Barrier*) provides a significant improvement (22%), which is close to *Ideal*.

5.3 Applying *Pilot* to Delegation Locks

It is difficult to remove the barriers in in-place locks as the lock is unaware of the memory operations in critical sections and barriers are irreplaceable to achieve the strong semantic that mutex lock provides. However, barriers' overhead can be reduced by limiting the contention to one NUMA node for a period which diminishes the appearances of cross-NUMA node accesses [5, 6, 10, 11, 27].

Different with in-place lock, we can easily apply *Pilot* to delegation lock and eliminate the significant overhead caused by barriers (line7) strictly following after an RMR (line 6), as both of them are a part of lock algorithm. Algorithm 6 shows the implementation of the delegation lock server after



(a) Queue and Stack (b) Link List (c) Hash Table (d) BOTS Floorplan

Figure 8. Data Structure and BOTS Benchmarks

applying *Pilot*. The modification towards the delegation lock client is similar and thus omitted.

Algorithm 6: Delegation Lock Server with *Pilot*

```
Data: Shared: reqPool, respPool;
      Local  : core, req, resp, ret, oldRetPool = 0,
               oldFlagPool = 0, cntPool = 0, hash;
      Const : hashPool;
1  req ← reqPool[core];
2  if req -> flag ≠ oldFlagPool[core] then
3      oldFlagPool[core] ← req -> flag;
4      Barrier;
5      resp ← respPool[core];
6      hash ← getHash(hashPool, cntPool[core] ++);
7      ret ← req -> criticalSection(req -> arg) ^ hash;
8      Barrier;
9      if ret ≠ oldRetPool[core] then
10         resp -> ret ← ret;
11         oldRetPool[core] ← ret;
12     else
13         resp -> flag ← resp -> flag ^ 1;
```

The lock server has the same procedure as the original implementation, such as reading and handling the request in line 1-5 of Algorithm 6. But rather than directly storing the return value to the response buffer and then setting a flag, lock server uses *Pilot* to inform the client. A barrier is still needed (line 8) to ensure that the possible existing modifications to more client-local variables in the critical section are observable before the response is observable. However, this barrier does **not** introduce substantial overhead since almost **no** RMR takes place before the barrier for the following two reasons. **Firstly**, since all critical sections execute in one lock server, the modifications towards shared variables have become local operations. **Meanwhile**, irrelevant modifications to local variables are not supposed to appear in the critical section for optimal performance, and thus the 64-bit return value is enough in most scenarios. Even in the worst cases that applications modify more local variables in the critical section, it only causes the performance to downgrade to a similar level as the original implementation.

5.4 Evaluation of *Pilot* in Delegation Locks

We apply *Pilot* to two different delegation locks, including a dedicated server lock, FFWD, and a migratory server lock, DSMSynch. We compare with the original implementation which uses appropriate barriers and achieves the best performance among different combinations.

Figure 7(c) shows the performance improvement in microbenchmark after applying *Pilot*. To simulate different contention levels, we vary the number of nops between two acquisition in the x-axis. Comparing with the original implementation, 56% and 32% performance improvements are achieved at high contention levels after applying *Pilot* to DSMSynch (DSynch-P) and FFWD (FFWD-P) respectively. The performance only downgrades to a similar level as the original implementation at a lower contention level. The improvement is more significant in DSMSynch than FFWD as FFWD is designed to batch several requests, and thus the overhead caused by barriers are shared among them. This mechanism is aiming at being store buffer-friendly by reducing the number of touched cache lines. However, it also partially hides the barriers' overhead.

We also include several data structure benchmarks. Firstly, we evaluate the performance improvement in Queue and Stack. In our experiments, both data structures are protected by a global lock and threads insert and then remove a member from them. *Pilot* provides stable benefit as the critical sections of those operations are simple, short, and irrelevant to their size. From Figure 8(a), applying *Pilot* to DSMsynch (DSynch-P) and FFWD (FFWD-P) provides 20% and 26% improvements in Queue, 30% and 16% improvements in Stack.

Then, we implement a sorted linked list by referring [16]. Threads insert 1 member then remove 1 member from the list after every 10 queries. Since the length of the critical section increases along with the list, we vary the preloaded members in the list. As shown in Figure 8(b), a maximum of 55% and 25% improvements for DSMSynch and FFWD appear when there are 50 preloaded members. Again, *Pilot* introduces no extra overhead, even in the worst cases.

Finally, we implement a hash table based on the linked list. Each bucket of the table is attached with a linked list and a lock. As the number of cores is limited, lock servers of FFWD are bound to an already used core after using 16 dedicated cores. 512 members are preloaded into the hash table and placed in different buckets uniformly. In our experiments, threads insert 1 member then remove 1 member from the list after every 10 queries. We vary the number of buckets in the x-axis of Figure 8(b). A maximum 61% improvement appears when there are 32 buckets after applying *Pilot* to DSMSynch. However, the improvement reduces with the increasing number of buckets as fewer threads acquire the same lock, and thus *Pilot* is barely used under such situation. Even so, we can still observe a 10% performance improvement. As for FFWD, a maximum of 24% improvement takes place when there are 16 buckets in the hash table.

Besides data structure benchmarks, we also evaluate the performance of delegation lock with *Pilot* applied in the BOTS [12] floorplan benchmark, which computes the optimal floorplan distribution of a number of cells. As the BOTS benchmark uses OpenMP, only the migratory server lock, DSMSynch (Dsynch), can be seamlessly integrated. Applying *Pilot* (Dsynch-P) can reduce up to 4% execution time under different input sizes. The improvement is less significant due to that the lock is not its performance bottleneck.

6 Discussion and Related Work

There are many trade-offs in the design of the memory model in the processor. For example, in the x86 processor, the store operations in the store buffer have to wait for previous store operations to be globally visible before commit, and the load buffer should be able to be snooped by coherency traffic, which leads to more complex hardware design. However, the ARM processor allows store operations to be reordered in the store buffer. Therefore the store buffer is less likely to be full, and the implementation of load/store buffer in the ARM processor could be much simpler and thus much power-efficient compared to the x86 processor. But as we observed in this work, the WMM has to take more effort when the orders between memory operations should be preserved, which influence the performance of applications significantly. None is a silver bullet.

There is not much work evaluating, reasoning, and characterizing the performance impact of ARM barriers. Ritson et al. propose an approach to profile the performance impact of different barriers under different applications [41]. However, they do not explore the characteristic of different barriers and fail to provide any advice to cut the overhead down. Ou et al. evaluate the overhead of forbidding reordering of loads and stores to explore the cost of avoiding out-of-thin-air results [33]. Yet, they fail to consider enough factors as we do. To our knowledge, we are the first to analyze the performance characteristics of ARM barriers in ARM server processors.

Prior work reduces ARM Barriers' overhead by eliminating redundant barriers at compile-time [31] or proposing new hardware primitives [24]. Different from those works, *Pilot* leverages the single-copy atomicity to remove the barrier, which causes the major overhead based on our observations, and thus provides noticeable performance improvements.

ARM also has noticed the potential performance bottleneck of their previous design and moves to MCA recently [36]. We confirm their worries in off-the-shelf systems and address the importance of barriers in the performance of ARM server processors. Characterizing the performance impacts of order-preserving approaches in the next-generation ARM processors remains an interesting future work.

7 Conclusion

This paper presents a comprehensive performance characterization and optimization of ARM barriers. We draw a set of observations from our experiments and provide a list of suggestions for developers to help them get minimal influence by WMM when porting existing applications to ARM platforms. Furthermore, we propose *Pilot*, which uses the knowledge from our observations to eliminate the tremendous overhead caused by performance-critical barriers efficiently. Applying *Pilot* provides considerable performance improvements in multiple benchmarks.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is supported in part by the National Natural Science Foundation of China (No. 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and a grant from Huawei Technologies. Haibo Chen (haibochen@sjtu.edu.cn) is the corresponding author.

References

[1] ARM AMBA. 2011. AXI and ACE Protocol Specification.

[2] ARM ARM. 2018. Architecture Reference Manual. *ARMv8, for ARMv8-A architecture profile* (2018).

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[4] Enrico Calore, Filippo Mantovani, and Daniel Ruiz. 2018. Advanced Performance Analysis of HPC Workloads on Cavium ThunderX. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*. IEEE, 375–382. https://doi.org/10.1109/HPCS.2018.00068

[5] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High Performance Locks for Multi-Level NUMA Systems. (2015), 215–226. https://doi.org/10.1145/2688500.2688503

[6] Milind Chabbi and John Mellor-Crummey. 2016. Contention-Conscious, Locality-Preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article Article 22, 14 pages. https://doi.org/10.1145/2851141.2851166

[7] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite. *ACM Trans. Archit. Code Optim.* 12, 4, Article Article 41 (Dec. 2015), 22 pages. https://doi.org/10.1145/2829952

[8] Nathan Chong and Samin Ishtiaq. 2008. Reasoning about the ARM Weakly Consistent Memory Model. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08) (MSPC '08)*. Association for Computing Machinery, New York, NY, USA, 16–19. https://doi.org/10.1145/1353522.1353528

[9] Jonathan Corbet. 2014. MCS locks and qspinlocks. https://lwn.net/Articles/590243/.

[10] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-Combining NUMA Locks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 65–74. https://doi.org/10.1145/1989493.1989502

[11] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.* 1, 2, Article Article 13 (Feb. 2015), 42 pages. https://doi.org/10.1145/2686884

[12] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*. IEEE Computer Society, 124–131. https://doi.org/10.1109/ICPP.2009.64

[13] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 325–334. https://doi.org/10.1145/1989493.1989549

[14] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/2145816.2145849

[15] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/1168857.1168877

[16] Vincent Gramoli. 2015. More than You Ever Wanted to Know about Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2688500.2688501

[17] SD Hammond, C Hughes, MJ Levenhagen, CT Vaughan, AJ Younge, B Schwaller, MJ Aguilar, KT Pedretti, and JH Laros. [n. d.]. Evaluating the Marvell ThunderX2 Server Processor for HPC Workloads. ([n. d.]).

[18] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[19] Arm Holdings. [n. d.]. ARM Cortex-A Series Programmer's Guide for ARMv8.

[20] SPARC International Inc and David L Weaver. 1994. *The SPARC architecture manual.* Prentice-Hall.

[21] Intel Intel. 64. The Intel® 64 and IA-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part* 1, 64 (64), 64.

[22] Data Center Knowledge. 2015. PayPal Deploys ARM Servers in Data Centers.

[23] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-Fly Pipeline Parallelism. *ACM Trans. Parallel Comput.* 2, 3, Article Article 17 (Sept. 2015), 42 pages. https://doi.org/10.1145/2809808

[24] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. 2014. Fence Scoping. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, Trish Damkroger and Jack J. Dongarra (Eds.). IEEE Computer Society, 105–116. https://doi.org/10.1109/SC.2014.14

[25] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2012. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 65–76. https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi

[26] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2016. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.* 33, 4, Article Article 13 (Jan. 2016), 62 pages. https://doi.org/10.1145/2845079

[27] Victor Luchangco, Daniel Nussbaum, and Nir Shavit. 2006. A Hierarchical CLH Queue Lock. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings (Lecture Notes in Computer Science)*, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner (Eds.), Vol. 4128. Springer, 801–810. https://doi.org/10.1007/11823285_84

[28] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A tutorial introduction to the ARM and POWER relaxed memory models. *Draft available from http://www. cl. cam. ac. uk/~pes20/ppc-supplemental/test7. pdf* (2012).

[29] Paul E McKenney. 2010. Memory barriers: a hardware view for software hackers. *Linux Technology Center, IBM Beaverton* (2010).

[30] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729

[31] Robin Morisset and Francesco Zappa Nardelli. 2017. Partially Redundant Fence Elimination for X86, ARM, and Power Processors. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3033019.3033021

[32] Adam Morrison. 2016. Scaling Synchronization in Multicore Programs. *Commun. ACM* 59, 11 (Oct. 2016), 44–51. https://doi.org/10.1145/2980987

[33] Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results. *Proc. ACM Program. Lang.* 2, OOPSLA, Article Article 136 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276506

[34] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 391–407. https://doi.org/10.1007/978-3-642-

03359-9_27

[35] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. 1999. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Vol. 16. Citeseer, 95.

[36] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article Article 19 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158107

[37] Milos Puzovic, Srilatha Manne, Shay GalOn, and Makoto Ono. 2016. Quantifying Energy Use in Dense Shared Memory HPC Node. In *4th International Workshop on Energy Efficient Supercomputing, E2SC@SC 2016, Salt Lake City, UT, USA, November 14, 2016*. IEEE Computer Society, 16–23. https://doi.org/10.1109/E2SC.2016.008

[38] Nikola Rajovic, Paul M. Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. 2013. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? , Article 40 (2013), 12 pages. https://doi.org/10.1145/2503210.2503281

[39] Nikola Rajovic, Paul M Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. 2013. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 40.

[40] J Rath. 2013. Baidu deploys marvell armbased cloud server.

[41] Carl G. Ritson and Scott Owens. 2016. Benchmarking Weak Memory Models. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article Article 24, 11 pages. https://doi.org/10.1145/2851141.2851150

[42] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (Much) Faster than You Think. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 342–358. https://doi.org/10.1145/3132747.3132771

[43] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/1993498.1993520

[44] Andreas Selinger, Karl Rupp, and Siegfried Selberherr. 2016. Evaluation of mobile ARM-based SoCs for high performance computing. In *Proceedings of the 24th High Performance Computing Symposium, Pasadena, HPC 2016, part of the 2016 Spring Simulation Multi-conference, SpringSim '16, CA, USA, April 3-6, 2016*, Josef Weinbub, Marc Baboulin, William I. Thacker, and Lukás Polok (Eds.). ACM, 21. http://dl.acm.org/citation.cfm?id=2972990

[45] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

[46] Sean White. 2014. The AMD Opteron A1100 Processor Codenamed "Seattle". In *IEEE Hot Chips*.

[47] Daniel Yokoyama, Bruno Schulze, Fábio Borges, and Giacomo Mc Evoy. 2019. The survey on ARM processors for HPC. *The Journal of Supercomputing* (08 Jun 2019). https://doi.org/10.1007/s11227-019-02911-9

[48] Mingzhe Zhang, Francis C. M. Lau, Cho-Li Wang, Luwei Cheng, and Haibo Chen. 2016. Scalable Adaptive NUMA-Aware Lock: Combining Local Locking and Remote Locking for Efficient Concurrency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article Article 50, 2 pages. https://doi.org/10.1145/2851141.2851176