Relaxed Persist Ordering Using Strand Persistency

William Wang

"As the lockdown measurements get relaxed gradually, it's clear that stricter lockdown rules can help governments manage the pandemic but can also hurt the economy. Likewise, stricter ordering rules for memory operations help simplify programming but can also hurt performance. In this joint work with the University of Michigan, we explore relaxing the ordering rules of persist operations to non-volatile memories for improved performance."

BACKGROUND

Memory consistency models define the ordering of *loads and stores* to non-overlapping addresses in memory. Likewise, *memory persistency models* define the ordering of *persists* to non-overlapping addresses in persistent memory [5]. The current Arm Instruction Set Architecture (ISA) has rather relaxed memory consistency models [10] and the ordering of persists, such as the DC CVAP instruction, follows epoch persistency [5]. In this joint work with the University of Michigan [1], we explore the more relaxed strand memory persistency model [5] to relax persist ordering for improved performance. The exploration for performance improvement via relaxed persist ordering is important, as non-volatile memory becomes real with Arm-based systems, and the language extensions and libraries to support persistent programming introduce significant performance overheads [8].

Like memory consistency models, memory persistency models can be defined at both the language level and the ISA level [2, 4]. The language level models can be stricter than, or the same as, the ISA level models. Memory persistency models explore the design space with the granularity of failure atomicity and the strictness of persist ordering as shown in Figure 1.

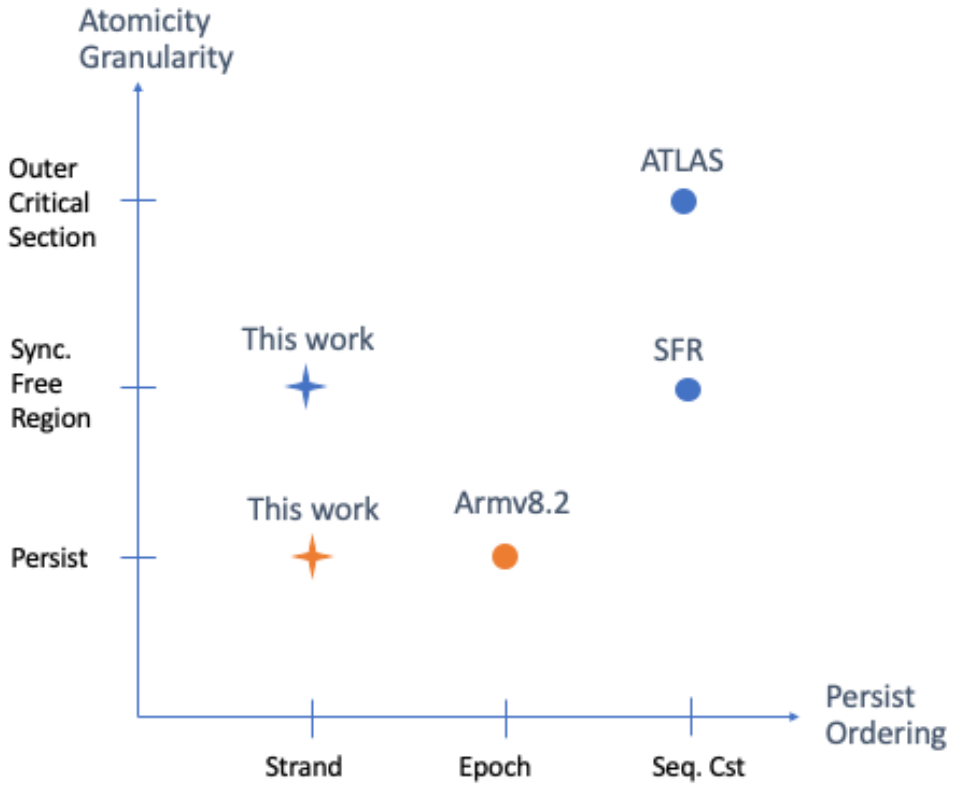

Figure 1. Memory persistency models at language and ISA levels. The ISA level persistency models are defined at the persist operations level, such as the DC CVAP instruction [7] as introduced in Armv8.2-A. The language level persistency models can be defined at coarser granularities, such as the synchronization free regions (i.e., SFR [3]) or outer critical sections (i.e., ATLAS [6]).

With language level persistency models, failure atomicity is often provided by undo-logging stores inside failure atomic sections (FASE), as exemplified in Figure 2.

```
FASE
{
  Write log-A;
  Persist log-A;
  Fence;
  Write A;
  Persist A;
  Fence;

  Write log-B;
  Persist log-B;
  Fence;
  Write B;
  Persist B;
  Fence;
}
```

Figure 2. Undo logging for failure atomic sections (FASE)

The programmer needs only to specify the start and end of the failure atomic sections in the program, as shown below. The undo logging as well as the persist and fence operations can be instrumented by compilers, as highlighted in blue in Figure 2.

```
FASE
{
  Write A;
  Write B;
}
```

Figure 3. Programmer needs only to specify the start and end of the failure atomic sections.

In Figure 2, if we allow memory persistency to be decoupled from memory consistency, the only persist ordering required within the FASE is between the persist of log write and the persist of data write in-place, as shown in Figure 4. In addition, all persists may need to be completed before the FASE can be committed.

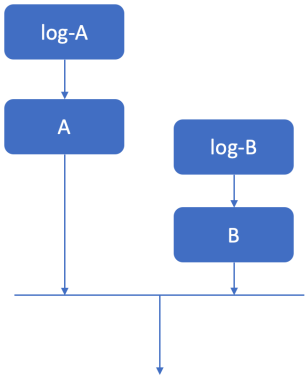

Figure 4. Ideal ordering of persists. All persists may need to be completed before the FASE can be committed.

When compiled for Armv8.2-A, the instrumented failure atomic section will feature three data synchronization barriers (DSBs) as shown in Figure 5. DSB is a fence that orders loads/stores as well as DC CVAPs before and after the barrier. The first DSB introduces unnecessary persist ordering between log-A and log-B, the second DSB introduces unnecessary persist ordering between A and B, as shown in Figure 6. The third DSB is to ensure all persists are completed when the failure atomic section can be committed.

```
FASE
{
  STORE log-A;
  DCCVAP log-A;
  DSB;
  STORE A;
  DCCVAP A;

  STORE log-B;
  DCCVAP log-B;
  DSB;
  STORE B;
  DCCVAP B;

  DSB;
}
```

Figure 5. Compiler instrumented failure atomic section for Armv8.2-A.

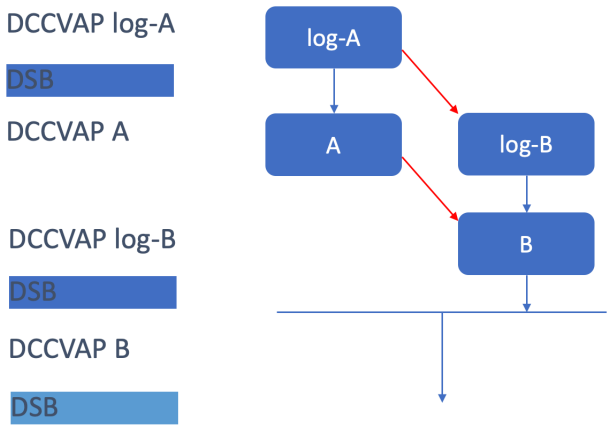

Figure 6. The first DSB introduces unnecessary persist ordering between log-A and log-B. The second DSB introduces unnecessary persist ordering between A and B. Both are highlighted in red.

PROBLEM and SOLUTION

The DSB introduces unnecessary ordering between section A and section B, as it orders all memory operations before and after the barrier. Strand persistency [1, 5] removes this unnecessary ordering, so that section A and section B can be decoupled into two different strands that have no implied ordering between them. The ordering between the log write/persist and in-place data write/persist is only required within each strand. Also, both strands A and B may need to be completed before the FASE can be committed. To specify this exact ordering, in this paper we propose three additional ISA primitives:

- **PersistBarrier**
- **NewStrand**
- **JoinStrand**

The failure atomic section as shown in Figure 3 can then be instrumented with the new ISA primitives as shown in Figure 7. The ideal ordering as specified in Figure 4 can then be implemented with the new ISA primitives as shown in Figure 8.

```
FASE
{
  STORE log-A;
  DCCVAP log-A;
  PersistBarrier;
  STORE A;
  DCCVAP A;

  NewStrand
  STORE log-B;
  DCCVAP log-B;
  PersistBarrier;
  STORE B;
  DCCVAP B;
```

```
    JoinStrand
}
```

Figure 7. Compiler instrumented failure atomic section with strand persistency ISA primitives.

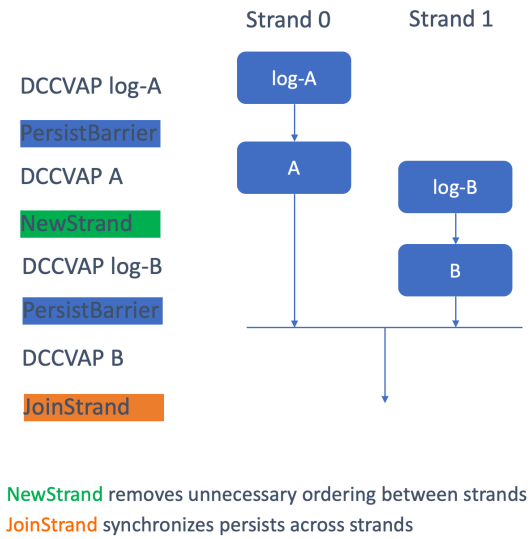

Figure 8. Strand persistency removes unnecessary ordering between strands with **NewStrand**, while providing **PersistBarrier** to order persists within strands and **JoinStrand** to synchronize persists across strands. Strand persistency can achieve the ideal persist ordering as shown in Figure 4.

RESULTS, IMPACT and FUTURE WORK

We implemented the microarchitecture to support the ISA primitives in gem5 and evaluated the performance impact on language-level persistency models such as the coupled and decoupled SFRs [3]. Strand persistency improves the performance of language persistency models by 45% on average. Details can be found in our ISCA'2020 paper [1].

The performance gain can potentially be seen across a wide range of persistent applications that leverage undo logging for failure atomicity, including most current software transactional memory libraries and language extensions for persistent memory, such as PMDK, NVM-Direct and go-pmem.

In addition to reducing the performance overhead due to persist ordering as addressed in this work, we are also interested in further performance optimizations in the architecture and microarchitecture for persistent programming, such as reducing the performance overheads of persisting, logging, and translations [8]. In addition to performance optimizations, we are also interested in helping address the persistent programming challenges with the necessary architectural and microarchitectural support [9].

TALK VIDEO AND PAPER

If you're interested in finding out more about the paper and our presentation of this work at ISCA 2020, please follow the following links.

Link to the talk video: https://isca2020.wistia.com/medias/lmx8ueqknj
Link to the paper: https://www.iscaconf.org/isca2020/papers/466100a652.pdf
Contact William Wang mailto:william.wang@arm.com

REFERENCES

1. Gogte, Vaibhav, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. “Relaxed Persist Ordering Using Strand Persistency." **ISCA 2020**
2. Kolli, Aasheesh, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, William Wang, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. "Language support for memory persistency." **IEEE Micro Top Picks 2019**
3. Gogte, Vaibhav, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. "Persistency for synchronization-free regions." **PLDI 2018**
4. Kolli, Aasheesh, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. “Language-level persistency.” **ISCA 2017**
5. Pelley, Steven, Peter M. Chen, and Thomas F. Wenisch. "Memory persistency." **ISCA 2014**
6. Chakrabarti, Dhruva R., Hans-J. Boehm, and Kumud Bhandari. “Atlas: leveraging locks for non-volatile memory consistency.” **OOPSLA 2014**
7. Arm Ltd. DC CVAP, Data or unified Cache line Clean by VA to PoP. https://developer.arm.com/docs/ddi0595/b/aarch64-system-instructions/dc-cvap
8. Wang, William, Stephan Diestelhorst. “Quantifying the performance overheads of PMDK.” **MEMSYS 2018**
9. Wang, William, Stephan Diestelhorst. “Persistent Atomics for Implementing Durable Lock-Free Data Structures for Non-Volatile Memory.” **SPAA 2019**
10. Alglave, Jade. “How to use the Memory Model Tool.” https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/how-to-use-the-memory-model-tool