Persistency Analysis

March 17, 2021

1 Introduction

An instruction `write A` persists at program point v' as long as every path from the write to v' has a `clwb A` followed by `sfence`.

$$\texttt{write A} \cdots \texttt{clwb A} \cdots \texttt{sfence} \cdots v'$$

with no other writes to the same location between the write and v'.

Likewise, `write A` persists before `write B` if $v' = $ `write B` and `write A` persists at v'.

$$\texttt{write A} \cdots \texttt{clwb A} \cdots \texttt{sfence} \cdots \texttt{write B}$$

Note that this only checks for ordering between particular writes, not general writes at these memory locations. That is, there could be another `write B` instruction that occurs before v' and `sfence`, so that A doesn't necessarily persist before the other write to B.

2 Problem Formulation

2.1 Direction of approximation

If we formulate the problem such that the existence of a L-path from a write to v' gives that the write persists at v', then that only shows that some possible program execution exists where the persist is guaranteed. That is, we let L represent paths that correctly give the right sequence of persist instructions. If there are other paths from the write to v' that isn't in L, then other program executions may not have this persistence. Thus, we must have that *every* path is in L to get the guarantee of persistence across all paths taken.

The other direction is to let L represent sequences of instructions that doesn't guarantee the persistence of the write. Then if a L-path exists from a write to v' shows that there is some program execution where the persist is *not* guaranteed at v'. The false positives here are cases where a L-path exists but is never taken in execution, and the only paths that actually execute are paths that *do* guarantee persistence. I think this formulation makes more sense.

2.2 Formulation

Given program points v, v' in the CFG, where the instruction at v is a write instruction to location A, the write potentially doesn't persist at v' (right before the statement runs) if either there doesn't exist a path from v to v' or if there exists a L-path from v to v'.

These are the potential ways that a write to A could not persist at v':

1. `clwb A` doesn't exist in the path.
2. `sfence` doesn't exist after `clwb A`.
3. (*)There is another write to the same memory location.

(*)We don't need the last way if know that v is the last write to A before v'.

2.2.1 Instructions

In the intraprocedural setting, with the usual control flow instructions, we also have

$$\text{Stmt} \to \cdots \mid \texttt{write x} \mid \texttt{clwb x} \mid \texttt{sfence}$$

where x is a location in memory and `write x` is just some instruction of the form $x := e$ for some expression e.

2.2.2 Language

In the CFG, we add to the outgoing edges for every statement of the form `clwb x` the label wb_x. To `sfence`, we add the label sf. Every other edge has empty label ε. We let the alphabet of the language $\Sigma = \{\text{wb}_x\}_{x\in X} \cup \{\text{sf}, \varepsilon\}$, where X is the set of all memory locations. If we are not assuming that the given program point for the write is the last write to the location before v', then we need to add w_A to the alphabet to signify writes. Further, if we wanted one single L, rather than a specific L for every two program points, we need to add write edge labels.

The following are the regular expressions that correspond to the potential ways that a write to A does not persist at v':

1. $(\Sigma - \{\text{wb}_A\})^*$
2. $\Sigma^*\, \text{wb}_A\, (\Sigma - \{\text{sf}\})^*$
3. $\Sigma^*\, w_A\, \Sigma^*$

The reachability language L is then either just the first two regular expressions or all three of them:

$$L = (\Sigma - \{\text{wb}_A\})^* \mid \Sigma^*\, \text{wb}_A\, (\Sigma - \{\text{sf}\})^* \quad \text{or} \quad (\Sigma - \{\text{wb}_A\})^* \mid \Sigma^*\, \text{wb}_A\, (\Sigma - \{\text{sf}\})^* \mid \Sigma^*\, w_A\, \Sigma^*$$

3 Additions from Px86

The operational semantics given in Raad et al. (POPL '20) paper gives additional persistency-related instructions present in the x86 ISA. There are the two RMW instructions $\texttt{FAA}(x, v)$ and $\texttt{CAS}(x, v, v'')$, another fence `mfence`, and the two other persist operations $\texttt{flush}_{\text{opt}}$ and `flush`. Since $\texttt{flush}_{\text{opt}}$ has equivalent specifications to `clwb`, we will only consider the addition of `flush`. Note that although `flush` simplifies persistency guarantees, it is a slower operation than $\texttt{flush}_{\text{opt}}$, which is itself slower than `clwb`.

The two additional ways that we get a persistence guarantee that one write occurs before another write is if we have `flush` between the two writes (without the need for fences), or the second write is a RMW instruction (then no fences are required). For durability or ordering guarantees, the addition of the `mfence` instruction works the same way as `sfence` (the actual difference between them is that `sfence` allows for reordering with later reads while `mfence` does not). Thus, we can easily allow for the substitution of `sfence` for `mfence`.

3.1 flush

Let's first deal with the addition of the `flush` instruction. Now there are two ways for a write instruction to have a durability or ordering guarantee:

1. `write A`$\cdots$`clwb A`$\cdots$`sfence/mfence`$\cdots v'$
2. `write A`$\cdots$`flush A`$\cdots v'$

A path has to be not of both forms for the persistency guarantees to not hold. So either both `flush A` and `clwb A` does not exist in the path or `flush A` does not exist, `clwb A` exists, but no fences. That is,

$$L = (\Sigma - \{\text{fl}_A, \text{wb}_A\})^* \mid (\Sigma - \{\text{fl}_A\})^* \, \text{wb}_A \, (\Sigma - \{\text{fl}_A, \text{sf}, \text{mf}\})^*$$

3.2 RMW instructions

If we want an ordering guarantee where the second write is a read-modify-write (RMW) instruction, then we do not need a fence after `clwb` instruction. The reasoning is that x86 guarantees stricter ordering between RMW instructions and other writes. More concretely, the ordering guarantee of Px86$_{\text{sim}}$ specified by the M-RMW operational semantics rule ensures that the buffer be empty before the RMW instruction is added directly onto the persistency buffer. That is, the first write instruction and its persist operation must be debuffered into the persistency buffer before the RMW instruction gets added to the persistency buffer. Then this guarantees that these instructions are executed in this order. Thus, the path from `write A` to `write B` where the second write is a RMW instruction does not guarantee ordering only if no persist operation on A exists in the path.

$$L = (\Sigma - \{\text{fl}_A, \text{wb}_A\})^*$$

3.3 Persistency Buffer and Operational Semantics

If we examine the operational semantics for the persistent memory model as described in the Raad et al. paper, we can get a better handle on what is required for persistency ordering between two write instructions.

Let $x, y \in \textsc{Loc}$ where $\textsc{Loc}$ is the set of all memory locations. Let v, v' be values. Write instructions (both direct stores and RMW's) are of the form $\langle x, v\rangle$ and persist instructions are of the form $\langle \text{per}, x\rangle$. Examining the two operational semantics rules for debuffering from the persistency buffer (PB), we can deduce that the only way a later write $\langle y, v'\rangle$ can be reordered before an earlier write $\langle x, v\rangle$ (assuming x, y are not on the same cache line) is if there exists a persist operation $\langle \text{per}, x\rangle$ between the two writes. Thus, the order in which the instructions are added to PB must be of the form $\langle x, v\rangle, \ldots, \langle \text{per}, x\rangle, \ldots, \langle y, v'\rangle$.

4 PMDK

For checking persistency using the PMDK operations `TX_BEGIN`, `TX_END`, and `TX_ADD()`, we can either run the graph reachability from every write instruction between `TX_BEGIN` to `TX_END` to the program point at `TX_END`. If using `TX_ADD()` function instead of the low-level persist operations, we only need to check that for every write operation, a corresponding `TX_ADD()` operation exists before the write.

5 Today

5.1 Language

The language that describes paths *not* in the language for paths. That is, it represents paths that do guarantee persistence.

$$\begin{aligned}
L^c &\to w_{x_1}\, T_1\, b_{x_1}\, T_1\, f_s\, T_1 \mid \cdots \mid w_{x_k}\, T_k\, b_{x_k}\, T_k\, f_s\, T_k \\
T_1 &\to w_{x_2} \mid \cdots \mid w_{x_k} \mid b_{x_1} \mid \cdots \mid b_{x_k} \mid T_1\, T_1 \mid f_s \mid \varepsilon \\
&\vdots \\
T_k &\to w_{x_1} \mid \cdots \mid w_{x_{k-1}} \mid b_{x_1} \mid \cdots \mid b_{x_k} \mid T_k\, T_k \mid f_s \mid \varepsilon
\end{aligned}$$

where there are k unique variables, w_x represents a write to x, b_x represents a `clwb` x statement, and f_s represents a `sfence` statement. The T_i rule represents a string of any length and any terminals except the w_{x_i} terminal.

Or written as a regular expression:

$$L^c = w_{x_1}\, T_1\, b_{x_1}\, T_1\, f_s\, T_1 \mid \cdots \mid w_{x_k}\, T_k\, b_{x_k}\, T_k\, f_s\, T_k \qquad \text{where } T_i = (\Sigma - \{w_{x_i}\})^*$$

If we consider only paths that start with a write (because it doesn't make sense to ask about persistence of a non-write), then these are the ways that a path can represent a non-persistence:

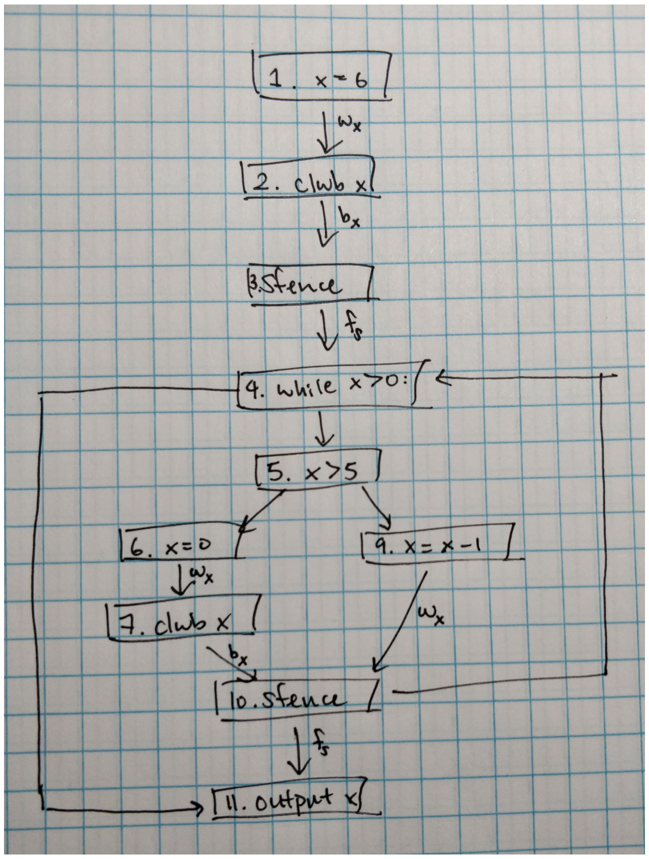
1. Doesn't start with a write:
$$(\Sigma - \{w_{x_1}, \ldots, w_{x_k}\})\, \Sigma^*$$
2. Write to the same variable appears again:
$$w_{x_1}\Sigma^* w_{x_1}\Sigma^* \mid \cdots \mid w_{x_k}\Sigma^* w_{x_k}\Sigma^*$$
3. There's no `clwb` after the write:
$$w_{x_1}\, (\Sigma - \{b_{x_1}\})^* \mid \cdots \mid w_{x_k}\, (\Sigma - \{b_{x_k}\})^*$$
4. There's no `sfence` after the `clwb`:
$$w_{x_1}\, (\Sigma - \{w_{x_1}\})^*\, b_{x_1}\, (\Sigma - \{f_s\})^* \mid \cdots \mid w_{x_1}\, (\Sigma - \{w_{x_k}\})^*\, b_{x_k}\, (\Sigma - \{f_s\})^*$$

The a path is in the language L if it's in any of the languages specified by the regular expressions above.

5.2 Example

Program 1

```
1   $x = 6$
2   clwb $x$
3   sfence
4   while $x > 0$:
5       if $x > 5$:
6           $x = 0$
7           clwb $x$
8       else
9           $x = x - 1$
10      sfence
11  output $x$
```



There is a path that is potentially non-persistent from every write to the last line. The actual path that this program will take upon execution will have the path string $w_x b_x f_s w_x f_s \in L$, so it isn't persistent, and neither is it from the second write to the last line.

5.3 PMDK: libpmemobj

(From *Programming Persistent Memory by Scargall* book)
3 API's to persist data to memory:

1. Atomic operations – simplest and fastest but limited to allocating and initializing wholly new blocks.
2. Reserve/publish – can be almost as fast as atomic but can't read data you just wrote.
3. Transactional – most flexible but slowest.

Transactional API:

```
int pmemobj_tx_add_range(PMEMoid oid, uint64_t off,
    size_t size);
int pmemobj_tx_add_range_direct(const void *ptr, size_t size);
```

```
TX_ADD(TOID o)
TX_ADD_FIELD(TOID o, FIELD)
TX_ADD_DIRECT(TYPE *p)
TX_ADD_FIELD_DIRECT(TYPE *p, FIELD)
```

```
TX_SET(TOID o, FIELD, VALUE)
TX_SET_DIRECT(TYPE *p, FIELD, VALUE)
TX_MEMCPY(void *dest, const void *src, size_t num)
TX_MEMSET(void *dest, int c, size_t num)
```

The transaction may also allocate entirely new objects, reserve their memory, and then persistently allocate them only one transaction commit. These functions include

```
PMEMoid pmemobj_tx_alloc(size_t size, uint64_t type_num);
PMEMoid pmemobj_tx_zalloc(size_t size, uint64_t type_num);
PMEMoid pmemobj_tx_realloc(PMEMoid oid, size_t size,
    uint64_t type_num);
PMEMoid pmemobj_tx_zrealloc(PMEMoid oid, size_t size,
    uint64_t type_num);
PMEMoid pmemobj_tx_strdup(const char *s, uint64_t type_num);
PMEMoid pmemobj_tx_wcsdup(const wchar_t *s,
    uint64_t type_num);
```

6 Enforcing Persist Ordering

6.1 PMTest Liu et al.

Here are the relevant PMTest operations copied from the paper:

- `global_timestamp` **(global status):** A global epoch counter that is incremented on every `sfence` encountered in the trace.
- `persist_interval` **(local status):** The interval in which certain memory location(s) may persist.
- `flush_interval` **(local status):** The interval in which certain memory location(s) may be explicitly written back to PM.
- `write(addr,size)` modifies an address range of [addr, addr+size) in the shadow memory. It first clears existing `persist_intervals` and `flush_intervals` within the address range and sets the `persist_intervals` as (`global_timestamp`, ∞). That is, this write may persist at any time moving forward.
- `clwb(addr,size)` writes back an address range of [adddr,addr+size) and the `flush_interval` is set as (`global_timestamp`,∞). That is, a writeback for these addresses has been issued and it may happen at any time moving forward. If there is an existing `flush_interval`, PMTest raises a `WARNING`.
- `sfence` enforces the ordering of prior `write` and `clwb` operations. First, it increments the `global_timestamp`. Second, it updates the `flush_interval` of prior `clwb`s so that the intervals end at the current `global_timestamp`, i.e, the writeback is complete. Third, it updates the `persist_interval` of prior `clwb`s so that the intervals end at the current `global_timestamp`, i.e, the write persisted.
- `isOrderedBefore(addrA,sizeA,addrB,sizeB)` checks whether *all* writes to [addrA,addrA+sizeA) can persist before any write to [addrB,addrB+sizeB) by checking if any of the `persist_intervals` in [addrB,addrB+sizeB) overlap with any of those in [addrA,addrA+sizeA).

The checking rule `isOrderedBefore`(A, B) checks if the persist intervals of A and B overlap. The start of PI's are only set by a `write` to the address and it is set to the current `global_timestamp`. The end of a PI is only changed by an `sfence` operation where it is set to `global_timestamp` if there was a previous `clwb` operation to the same address. Because `global_timestamp` is incremented only by an `sfence` operation and is never decremented, this implies that there ***must*** be a `sfence` before the last write to B (the write that is being checked for persist ordering).

Suppose that there isn't a `sfence` between `write`(A) and `write`(B) but that an `sfence` occurs after both writes. Let (a_1, a_2) be the PI for A and let (b_1, b_2) be the PI for B. The intervals overlap if $b_1 < a_2$. At this `sfence` call, the first thing that occurs is that `global_timestamp` is incremented. Then a_2 is set to this new `global_timestamp`, assuming that a `clwb`(A) exists before the `sfence` (if this wasn't the case, a_2 would never be changed from ∞ so $b_1 < a_2$ always). But note that this a_2 must be strictly greater than b_1 because b_1 was set by some previous `global_timestamp` and the `sfence` operation just incremented `global_timestamp`. So unless b_1 is over-written by another `write`(B), $b_1 < a_2$ and the intervals overlap.

Note. So it might just be that everything is simple as long as you don't consider rewrites. Well actually, it might be that there are multiple PI inside a single range of addresses. I'm thinking of arrays. There's also performance "bugs" that PMTest checks for.

Maybe also think about interprocedural and weird functions and loops and threads and other things like that that could mess things up. Also, note that PMTest only works for a particular execution whereas static program analysis will probably be flow-insensitive. How does the interaction of multiple `clwb` within the same address range interact with a single overall write? What's the point of maintaining `flush_interval`?

6.2 Operational Semantics