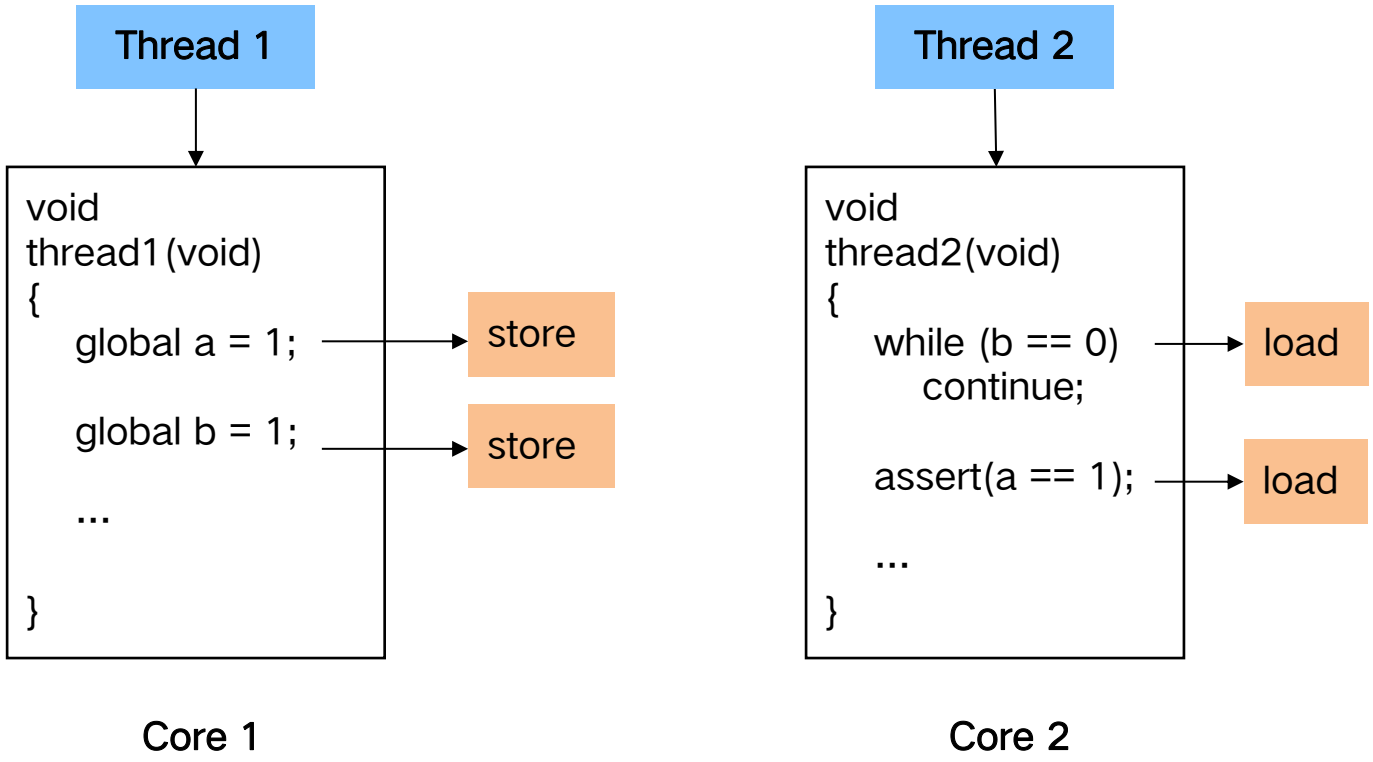
Hawkeyes：Addressing Weak Memory Order in Program Migration Based on Instruction Windows

—— Presenter: Zhangqi Zhu

1. WMM bugs —— A basic example



Running on x86:

- Thread 1 performs two store operations in order.
- The update 'a = 1' is visible before 'b = 1' to Thread 2
- Thread2 will not encounter an assertion error

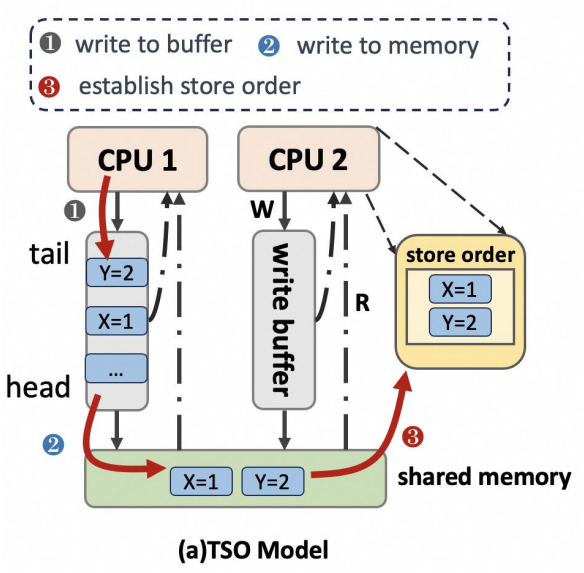
Running on Arm:

- The two store operations from Thread 1 may be reordered
- Allowing 'b=1' to execute and update before 'a=1'
- Thread 2 could encounter an assertion error

2. Difference between x86 and Arm memory models

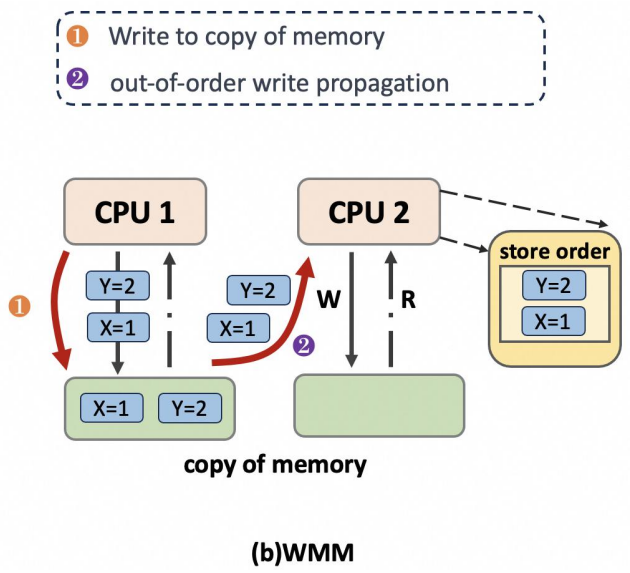
平头哥

- x86 — TSO(Total Store Order)



- Incorporates a FIFO–based write buffer for processors
- Preserving the order of write instructions while enabling direct reading instructions from the write buffer

- Arm — WMM(Weak Memory Model)

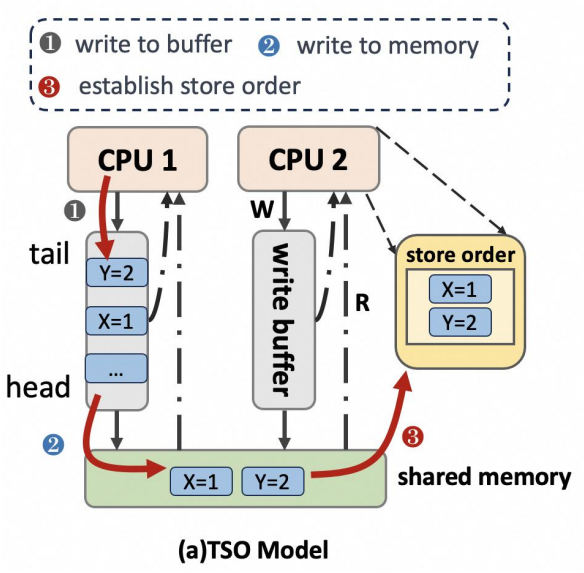


- Each processor possesses its complete memory copy, allowing independent reading and writing
- Permitting reordering as the writes propagate

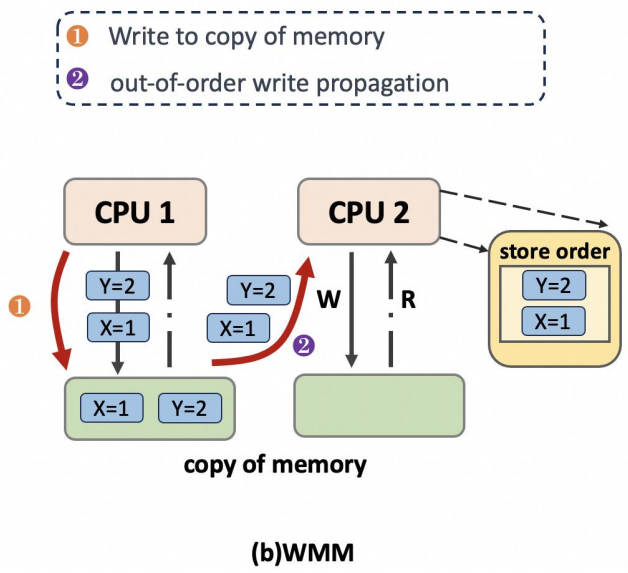
	X86 (TSO)	ARM (WMM)
Loads reordered after loads	N	Y
Loads reordered after stores	N	Y
Stores reordered after stores	N	Y
Stores reordered after loads	Y	Y

3. Advantages and disadvantages

- x86 — TSO(Total Store Order)



- Arm — WMM(Weak Memory Model)



Advantages:

- Offers a simpler programming model for multi–threading, enhancing portability across different systems and architectures.

Disadvantages:

- TSO relies on relatively complex hardware architectures to enforce memory synchronization, which can add overhead.
- In systems with high parallelism, TSO may result in an excessive number of memory synchronization operations, potentially degrading performance.

Advantages:

- Can deliver higher performance, fully utilizing the throughput of the ALU.

Disadvantages:

- Requires developers to explicitly manage memory synchronization through memory barrier instructions, making it challenging to write correct and efficient multithreaded code.

4. How to solve the WMM bugs —— Memory barrier insertion

- **Memory Barrier** is a type of instruction that imposes a strict ordering constraint on how the CPU and compiler perform memory operations, preventing instruction reordering before or after the barrier instruction.
- Can be realized through hardware mechanisms, software solutions, or a combination.

```
void thread1(void)
{
    global a = 1;
    global b = 1;
    ...
}
```

Memory barrier

Barrier Type	Example	Functions
Load-Load barrier	Ldr ; dmb ld ; Ldr	Ensure read memory instruction issued in strict order
Store-Store Barrier	Str ; dmb st ; Str	Ensure write memory instruction issued in strict order
Load-Store Barrier	Ldr ; dmb ish ; Str	Ensure load instruction issued prior to store instruction
Store-Load Barrier	Str ; dmb ; Ldr	Ensure store instruction issued prior to load instruction

- Memory barrier need to be added by the application developers. Those who have long been developing under the TSO model may find it challenging to identify potential WMM issues in the code when migrating programs to ARM architecture.
- WMM bugs are sporadic and dependent on specific business use cases, making them difficult to locate and identify through testing. —— We need an automated tool!

5. Existing solutions

- **Expert**
  - Rely on human experts to manually introduce any additional barriers in the code base
  - Neither practical or scalable
- **Atomic Transformation**
  - Convert global variables to atomics or confining them with robust locks and memory barriers
  - Really poor performance.
- **Dynamic Data Race Detection (Tsan/C11 Tester)**
  - Check variable locations in race conditions for atomic transformations
  - Does not fully match WMM bugs
- **Atomig**
  - Converts the program into LLVM IR and completes automated memory barrier insertion and modification to LLVM IR through pattern matching, analyzing explicit and implicit synchronization patterns
  - Still a relatively high false positive rate and miss rate

6. Our solution —— Hawkeyes

平头哥

- Hawkeyes dynamically detects memory access conflict during program runtime and gathers memory access information and thread details.
- It then analyzes micro-instruction differences between two conflicting memory access instructions to determine if they are in the same instruction window.
- Hawkeyes will ultimately generate a report at the source code level, containing information about the locations at risk for WMM bugs and recommendations for the types of memory barriers to insert.

7. Design 1： Memory Access Conflict Detection

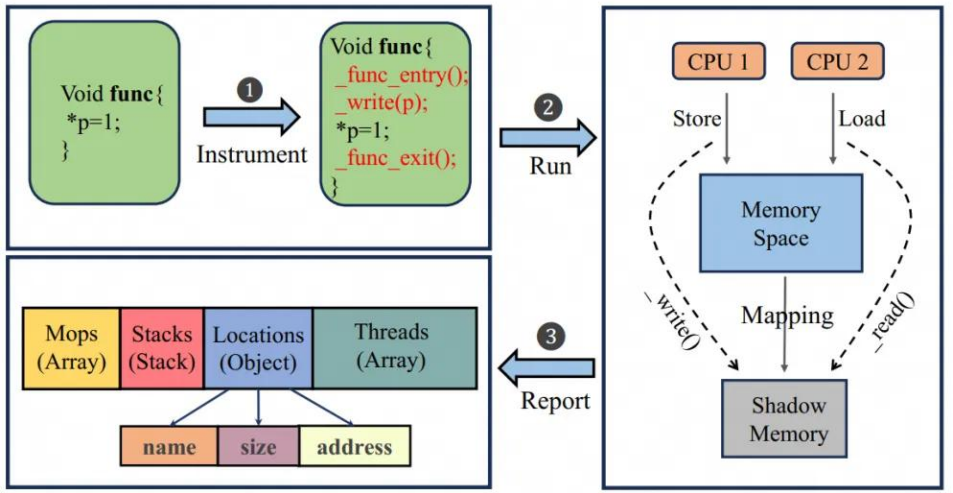
平头哥

Memory access conflict:

- Happens when multiple threads access the same memory space during program execution (where each thread may have multiple memory operations accessing the same shared memory).
- WMM bugs may occur only when two threads each have memory operations accessing two same shared memory.

Detection Workflow:

- Hawkeyes employs instrumentation techniques during the program compilation to capture memory access information.
- Hawkeyes utilizes shadow memory to record runtime state information for the instrumented program.
- The stored state information includes observed synchronization events, locks held by each thread, current memory locations, and other system details



8. Design 2：Instruction Window Analysis

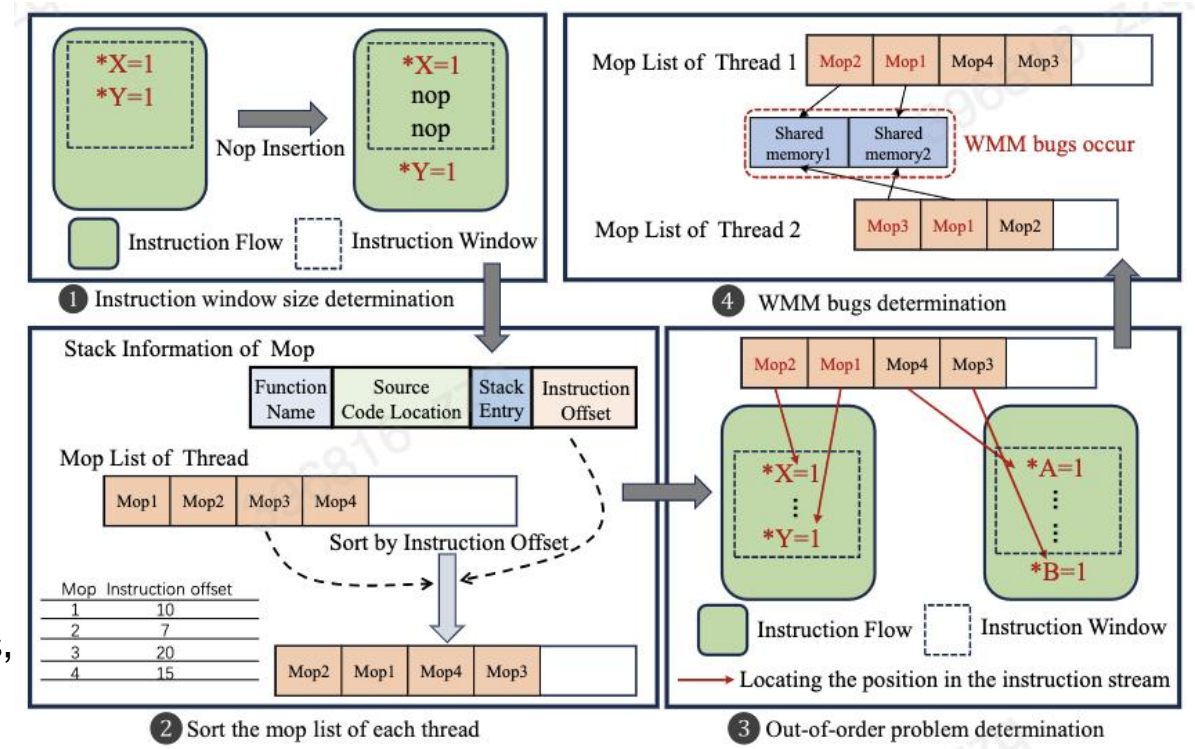
平头哥

Instruction windows:

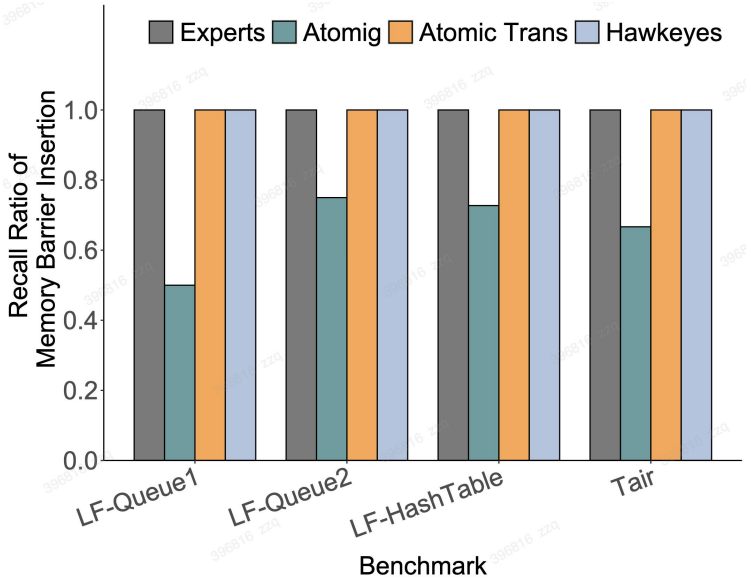
- The instruction window, constitutes a vital hardware structure widely employed in contemporary processors.
- Pairs of instructions, with a number of micro-instructions falling outside the size of the instruction window, will not encounter any disorder in their execution order.

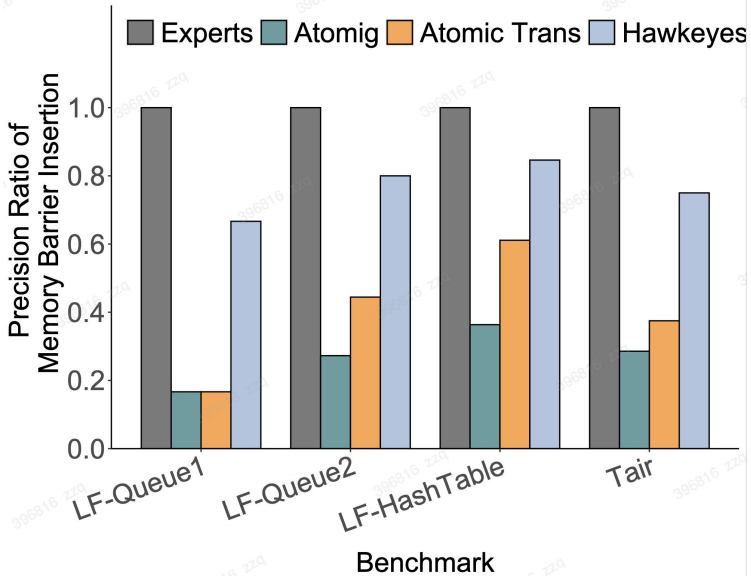
Analysis Workflow:

- Determine the Instruction window size
- Sort the mop list of each thread
- Determine the out-of-order problem
  - If two adjacent memory operations are within the same instruction window, then it might encounter instruction execution reordering issues.
- Determine the WMM bugs
  - For two sets of memory operations in different threads accessing the same two memory access conflict locations, if they both suffer from out-of-order problem, WMM bugs may occur.



9. Evaluation：Recall ratio and Precision ratio





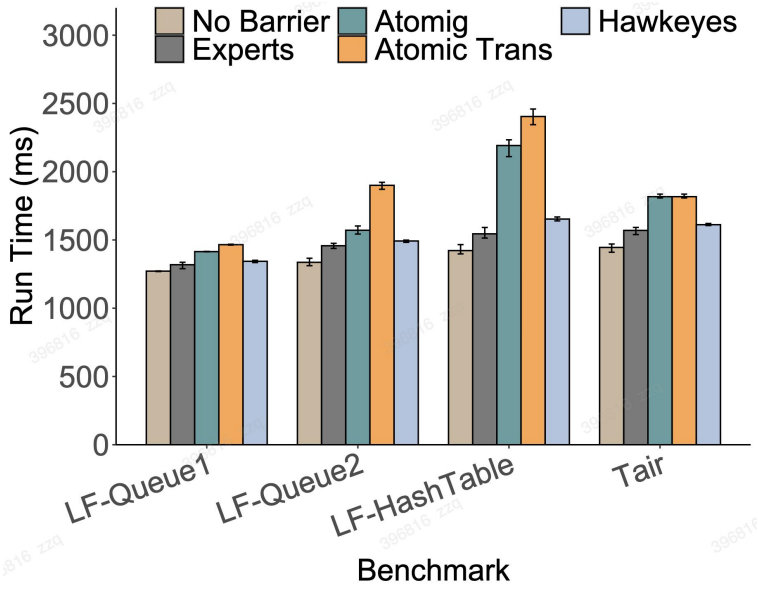
Recall Ratio Analysis:

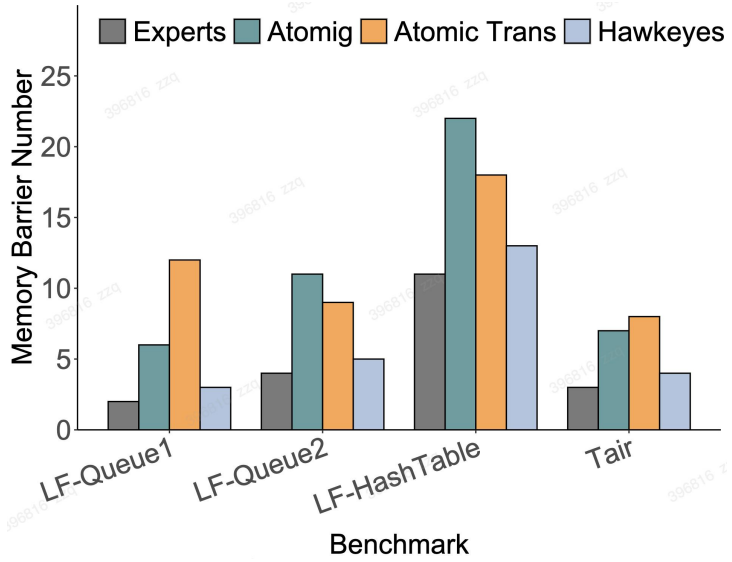
- We compare the positions where the mentioned approaches insert memory barriers with the original locations of memory barriers in the source code.
- Experts, Atomic Trans, and Hawkeyes exhibit covers all potential locations.
- Atomig achieves a coverage range of only 50.0%-75.0% for WMM bugs

Precision Ratio Analysis:

- We compare the positions where memory barriers were inserted in each approach with the potential locations where WMM errors may occur.
- Hawkeyes improves the average PP of insertion by 127.1% and 75.3% compared to Atomig and Atomic Trans through inserting memory barriers only when:
  - there is a potential for memory access conflicts
  - two instructions within the same instruction window
  - there are not any existing memory barrier

10. Evaluation：Program Performance





Program Performance Analysis:

- Hawkeyes reduces the average program runtime compared to Atomig and Atomic Trans by 6.4% and 13.5%, respectively.
- In contrast to manual memory barrier addition, Hawkeyes incurs only a marginal increase(e.g. 5.6%)
- The performance improvement can be attributed to the reduction in the number of memory barrier insertions.
- On average, Hawkeyes has notably decreased the number of memory barrier insertions by 37.1% compared to Atomig and Atomic Trans.

Thanks

Contact Information：zhuzhangqi.zzq@alibaba-inc.com