Trade-off between performance and transparency for in-NVMM checkpointing

Ana Khorguani, Thomas Ropars, Noel De Palma

Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

July 9, 2021

What is Non-Volatile Memory?

Historically, memory was divided into two parts:

- Storage (SSD, HDD)
- Volatile memory (DRAM)

Non-Volatile Memory offers best from the both worlds:

- Support for data persistence
- Byte-addressability (accessed by CPU's load and store instructions)
- Performance closer to DRAM
  - Faster than any storage device with orders of magnitude
  - Around 6 times slower than DRAM

Non-Volatile Memory Modules

Non-Volatile memory modules became available from 2019

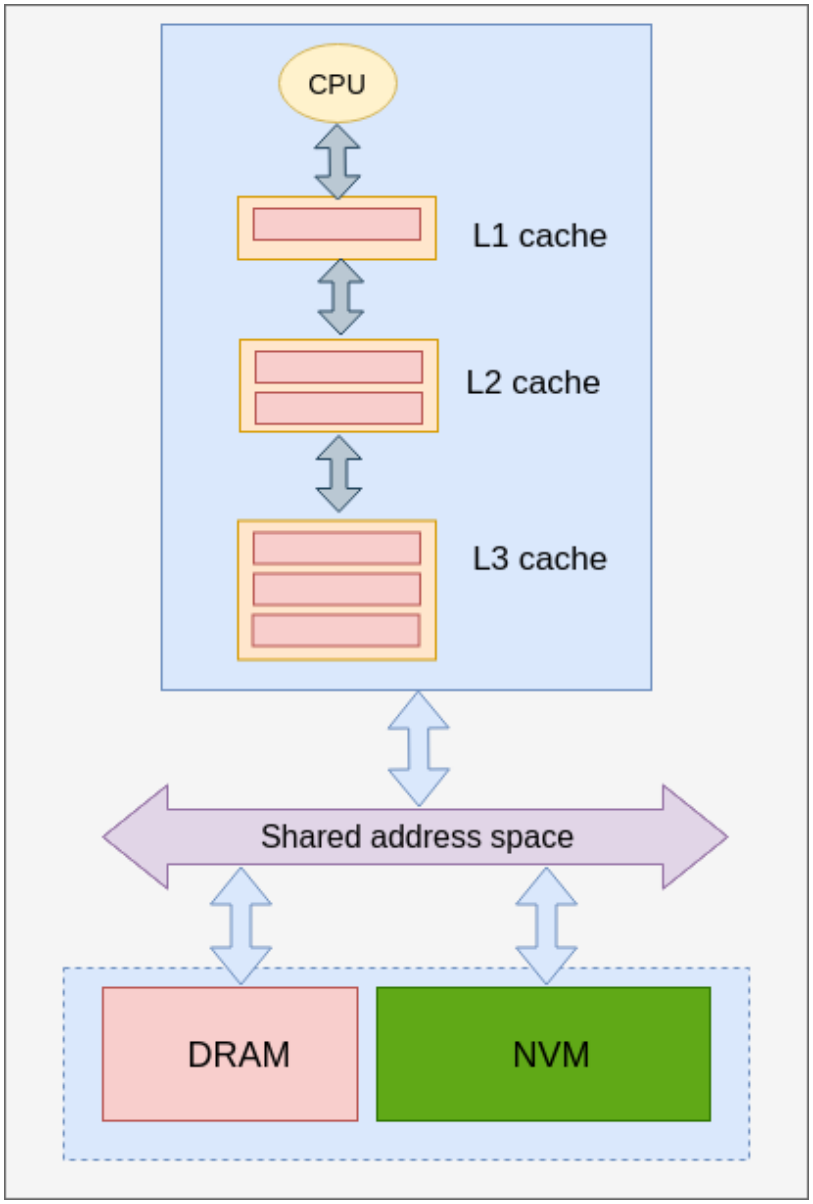
Figure: Intel Optane Persistent Memory

Motivation and Challenges

Use NVM to:

- Make applications failure resilient
  - Restart the application after a failure by loading data from NVM
  - Support multi-threaded applications
  - Handle the consistency issues of the data structures
- Maintaining high performance
  - The frequency of persisting data on NVMM will affect performance
- Considering the complexity of the modifications to the applications
  - Can fault tolerance be achieved transparently?

Memory architecture



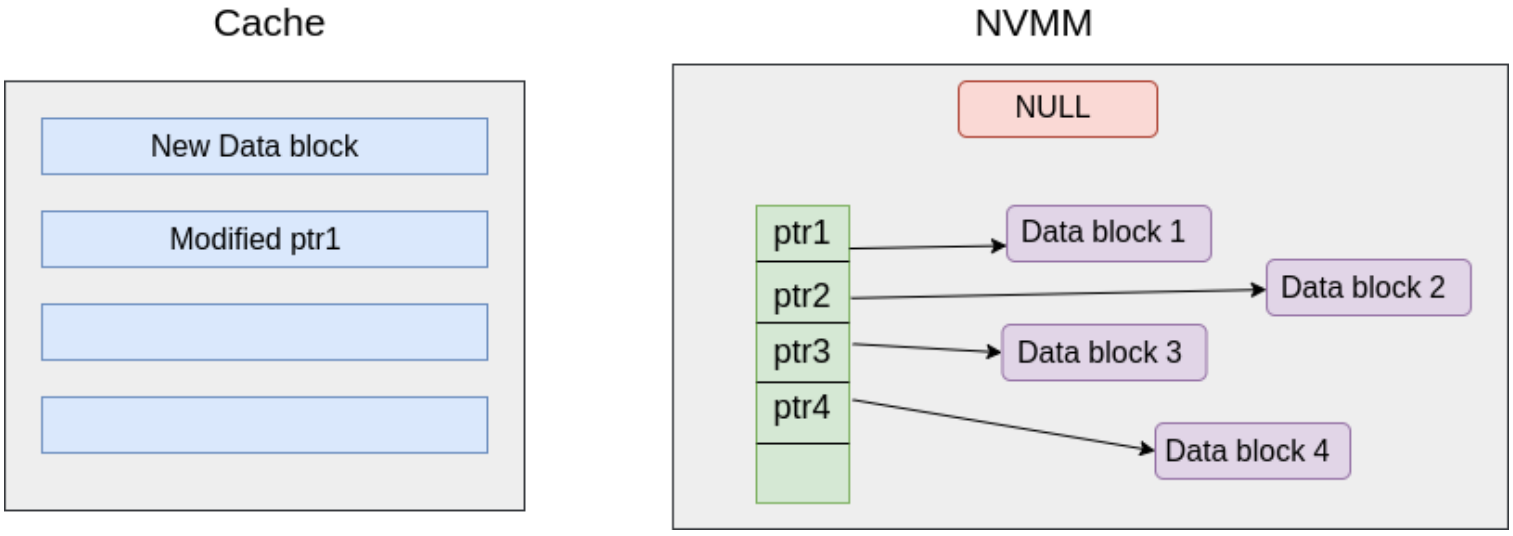
New memory hierarchy:

- Volatile
  - Caches
  - DRAM
- Persistent
  - NVMM

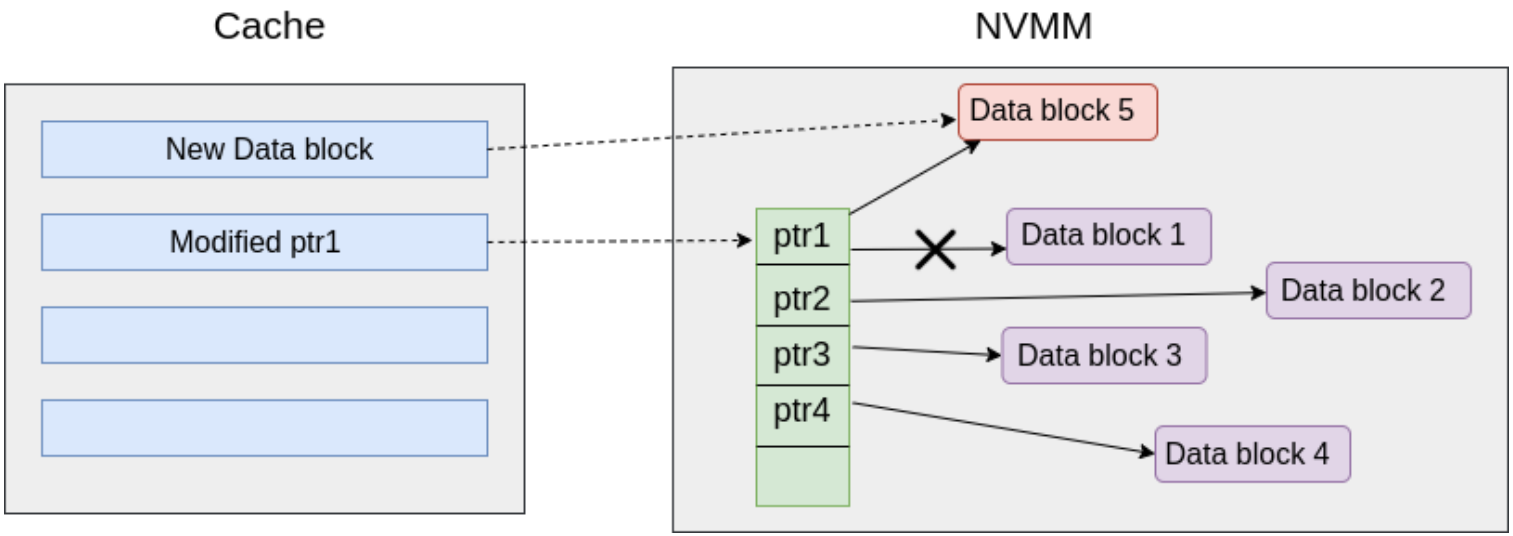
On cache line evictions, updated cache lines are possibly written back out of order to NVMM

Consistency issues

We care about the data movement from cache to NVMM:

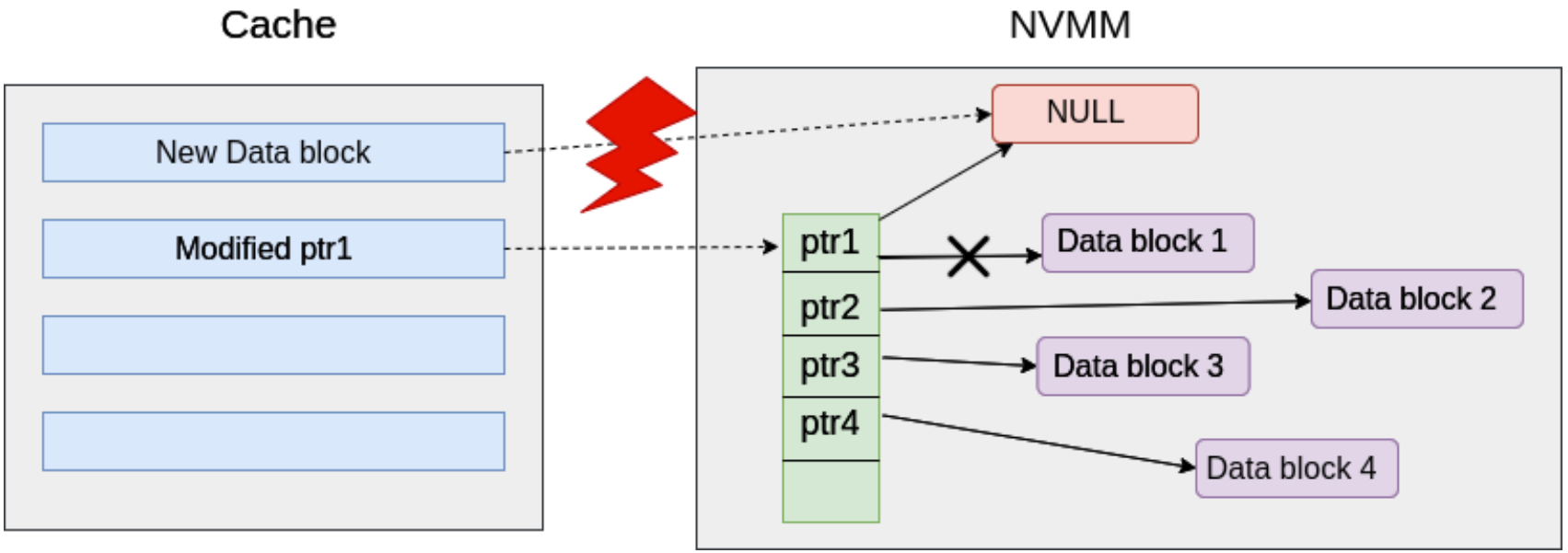


Expected final state of the data structure in NVMM:

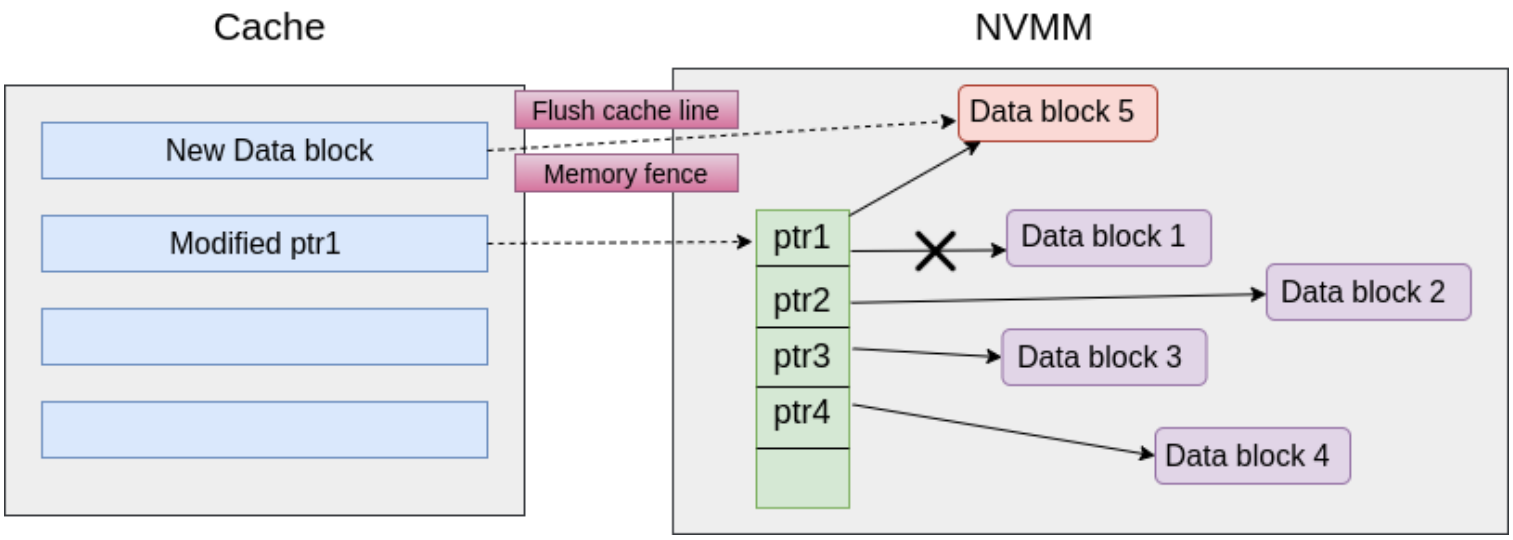


Consistency issues

What if the system crashes after modifying the pointer without updating the data block?



Intel processor flush instructions



Intel explicit flush instruction:

- `Clwb (@address)`

Introduces high overhead due to:

- Invalidating the cache lines
- Putting restrictions on out-of-order executions

Challenges of transparency

What is needed to transparently make applications fault tolerant?

Without the programmer's intervention, the system needs to:

- Detect the data structures that need to be saved in NVMM
- Detect the updates of these data structures
- Guarantee the consistency of these updates

Our Contribution

1. We propose a new highly efficient in-NVMM checkpointing technique designed for multi-thread applications
2. We analyse and highlight the shortcomings of most of the existing `transparent` solutions
3. We illustrate the evaluation of our design on a real hardware
   - The overhead of our algorithm can be as low as 20-35%
   - Our technique can perform 1.5-6$\times$ better than state-of-the-art solutions.

Coming up...

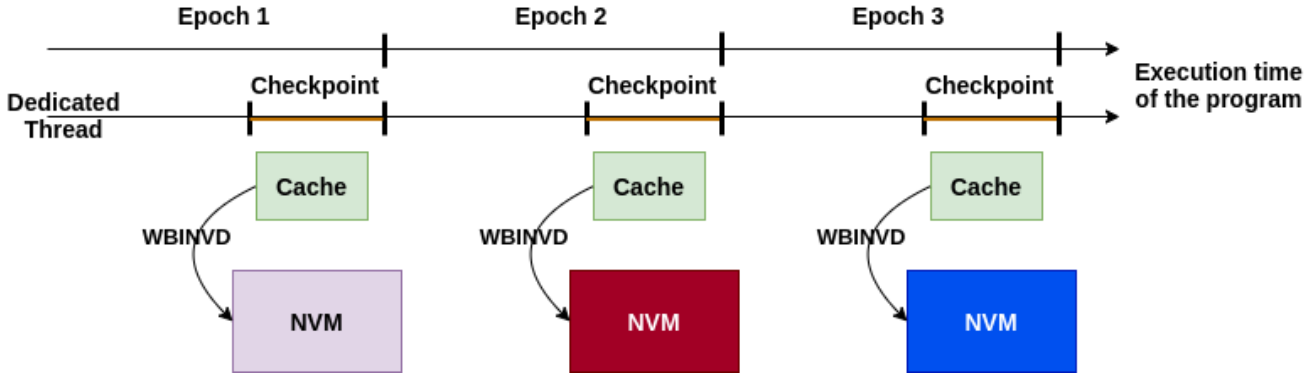
Our technique

Our approach

We propose a technique where we rely on a limited intervention of the programmer to apply our solution to existing applications

Our technique is based on:

- Checkpoints
  - The program execution is divided into epochs
  - At the end of the epoch, the data is persisted by flushing the entire content of the cache



Our approach

Our checkpoint technique is completed with:

- In-Cache-Line Log [Cohen et al., ASPLOS 2019]
  - An efficient undo log, without need of flush instructions
  - The log entries are in the same cache lines as the data fields
  - Relies on Persistent Cache Store Order (PCSO) memory ordering model
- Checkpoint atomic sections (CASEs)
  - Guarantee that the epochs end only when the data is consistent
  - Guarantee that critical sections are executed with respect of the checkpoints

Producer-consumer modified with our approach

```
1  produce(chunk_t *chunk):
2      pthread_mutex_lock(queue->mutex);
3      while queue is full:
4          pthread_cond_wait(queue->notFull, queue->mutex);

6      begin_case();
7      log_InCLL(&queue->data[queue->head]);
8      queue->data[queue->head] = chunk;
9      log_InCLL(&queue->head);
10     queue->head = (queue->head + 1)%queue->size;
11     end_case();

13     pthread_cond_signal(queue->notEmpty);
14     pthread_mutex_unlock(queue->mutex);
```

Coming up...

Existing transparent techniques

State of the Art

Different approaches have been proposed to leverage NVMM:

- Systems, based on transactions using logging mechanisms [Dudetm, ACM SIGPLAN 2017], [Clobber, ASPLOS 2021
- Lock-based algorithms, which flush the modifications as soon as they occur [iDO, MICRO 2018]
- Software checkpoints [PMThreads, PLDI 2020]

Transparent systems

The idea behind achieving transparency for lock-based algorithms is to:

- Define failure-atomic sections (FASEs)
  - Program's regions between lock acquire and release
- Guarantee consistency of modifications inside FASE

The programmer needs to take care of the other modifications that are needed to guarantee fault-tolerance.

Transparent systems

Both iDO and PMThreads offer transparency:

- iDO is a compiler-directed approach
  - It flushes the modified cache lines during the program execution
- PMThreads is an user-space runtime library
  - It uses shadow paging technique and keeps track of the write operations

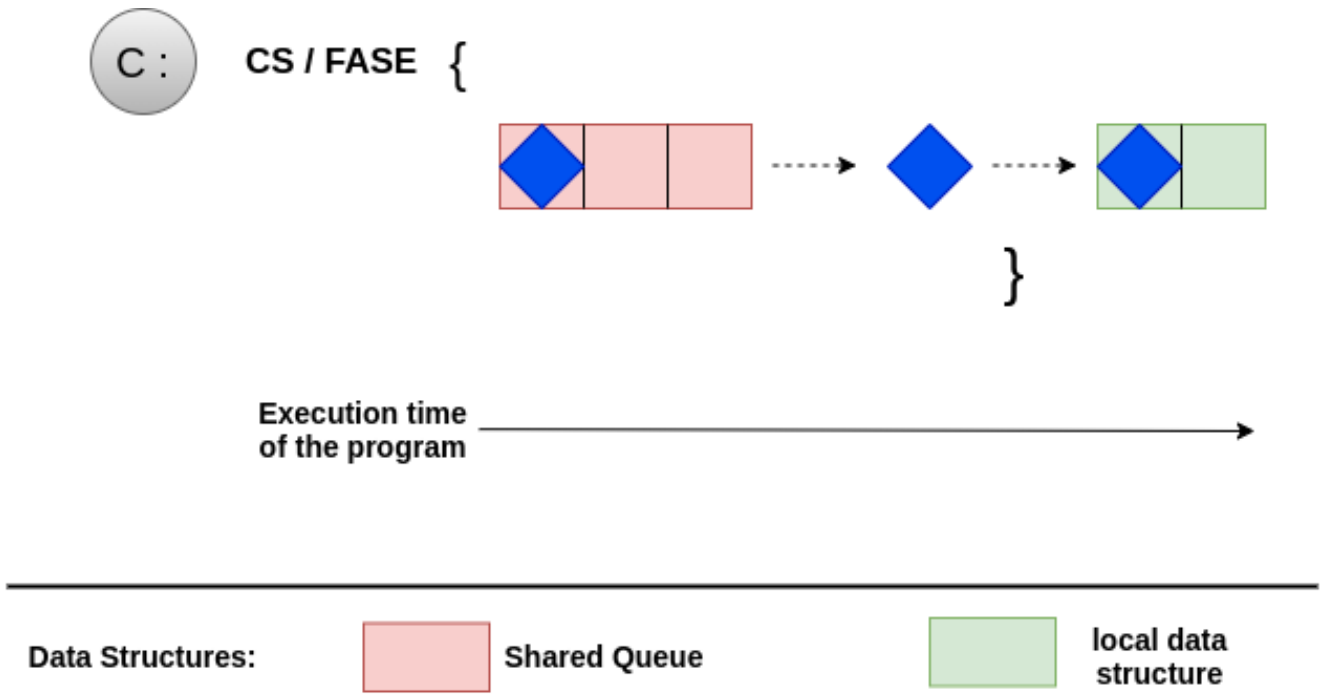
Correctness issue of transparent systems

Can designs that offer transparent modifications really make applications fully fault tolerant automatically?

The answer to this question is: NO

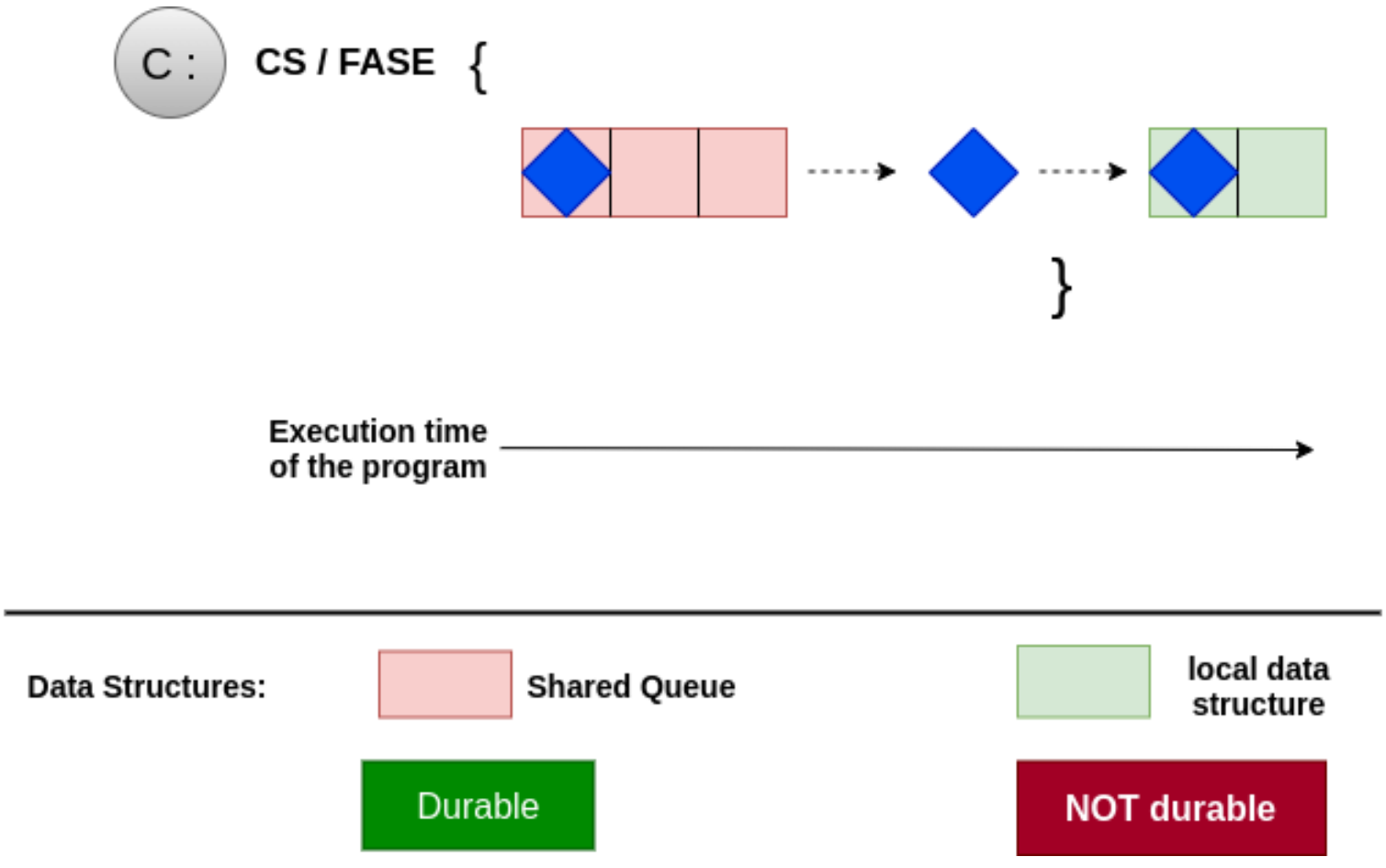
Producer-consumer example

Let us consider one Consumer thread of a producer-consumer program:



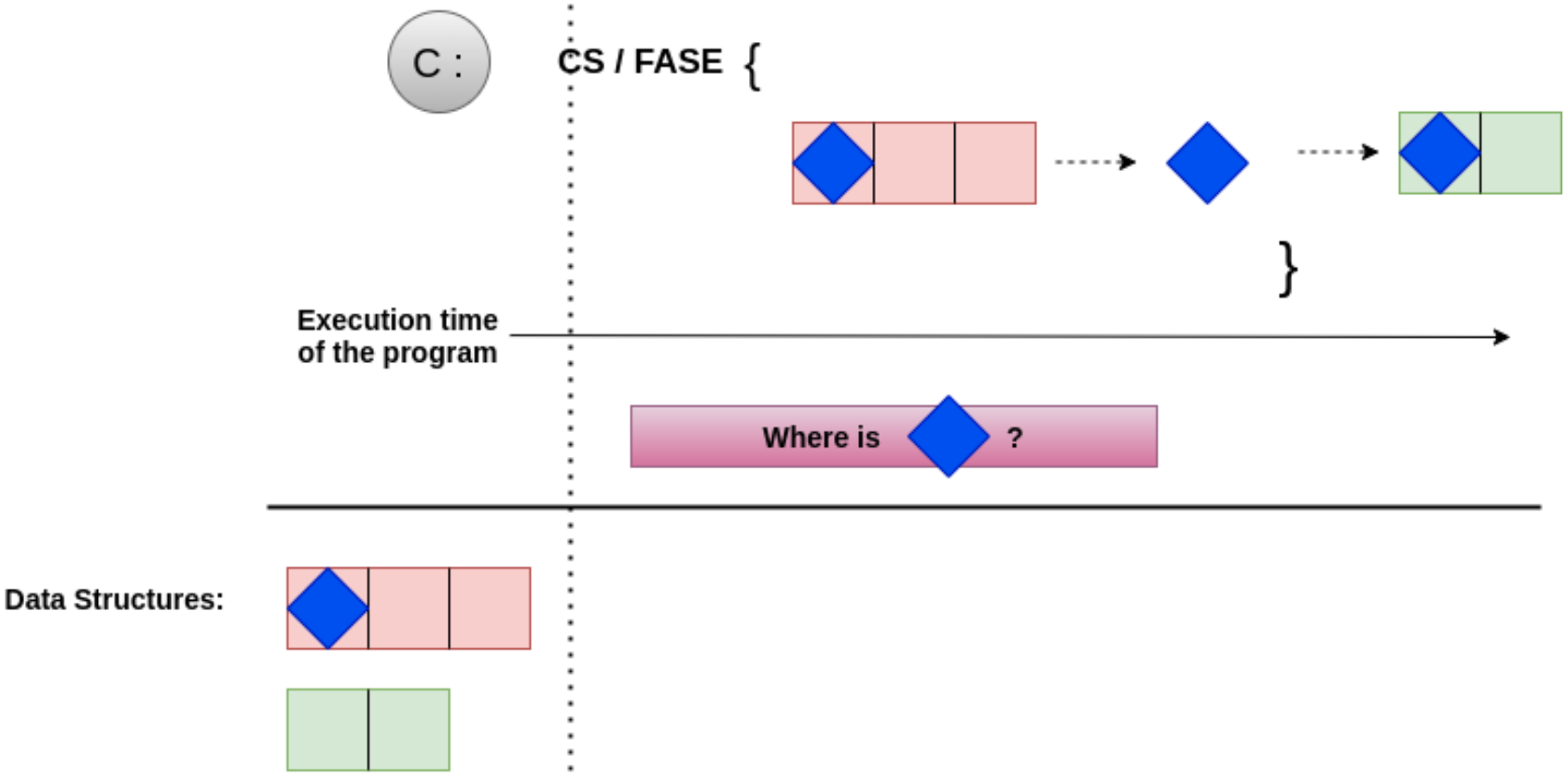
Durability guarantees

Identifying the data that should be durable, is achieved only for the shared data structures



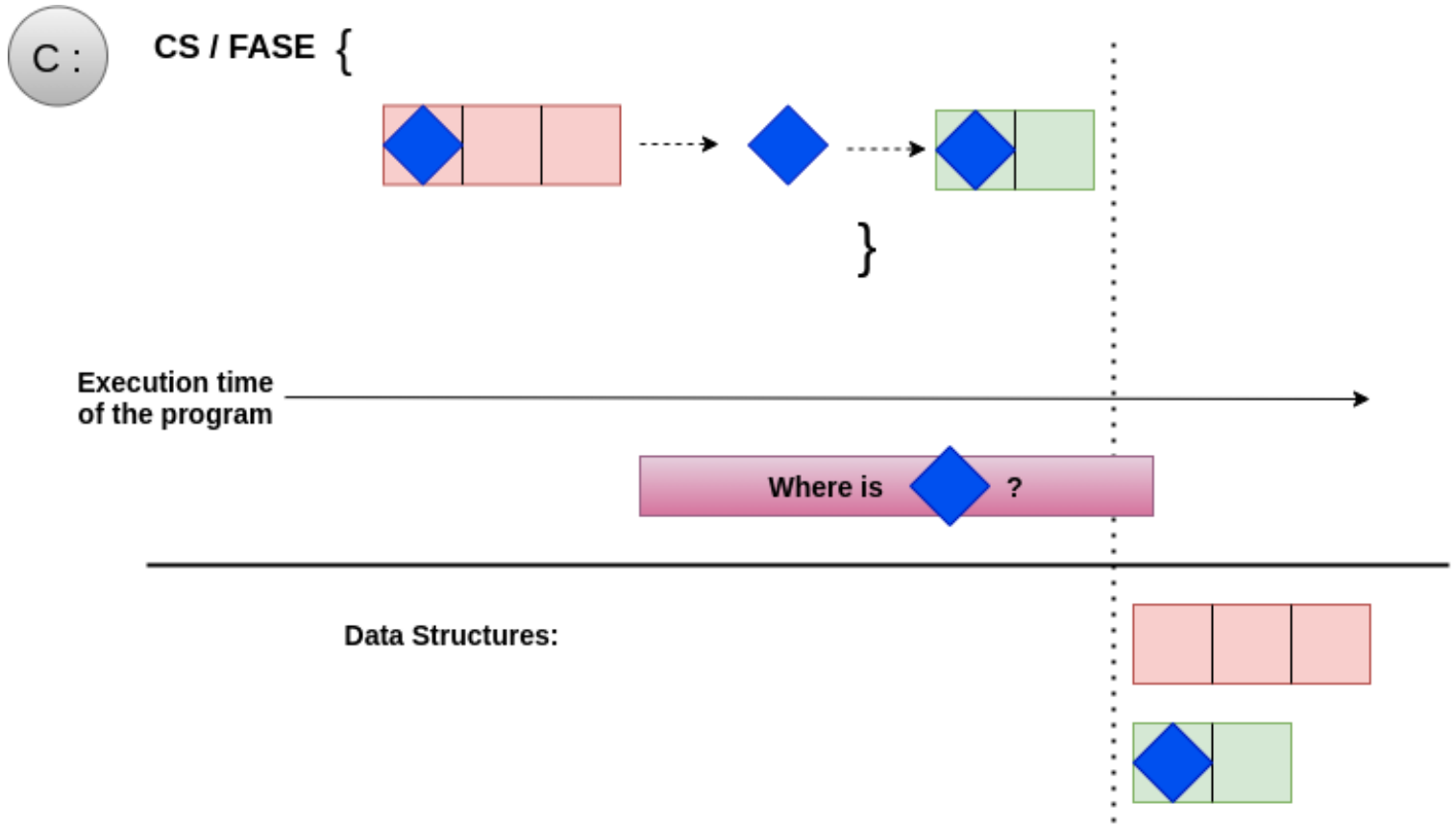
Allocation of the element

Where is the data element during the program execution?



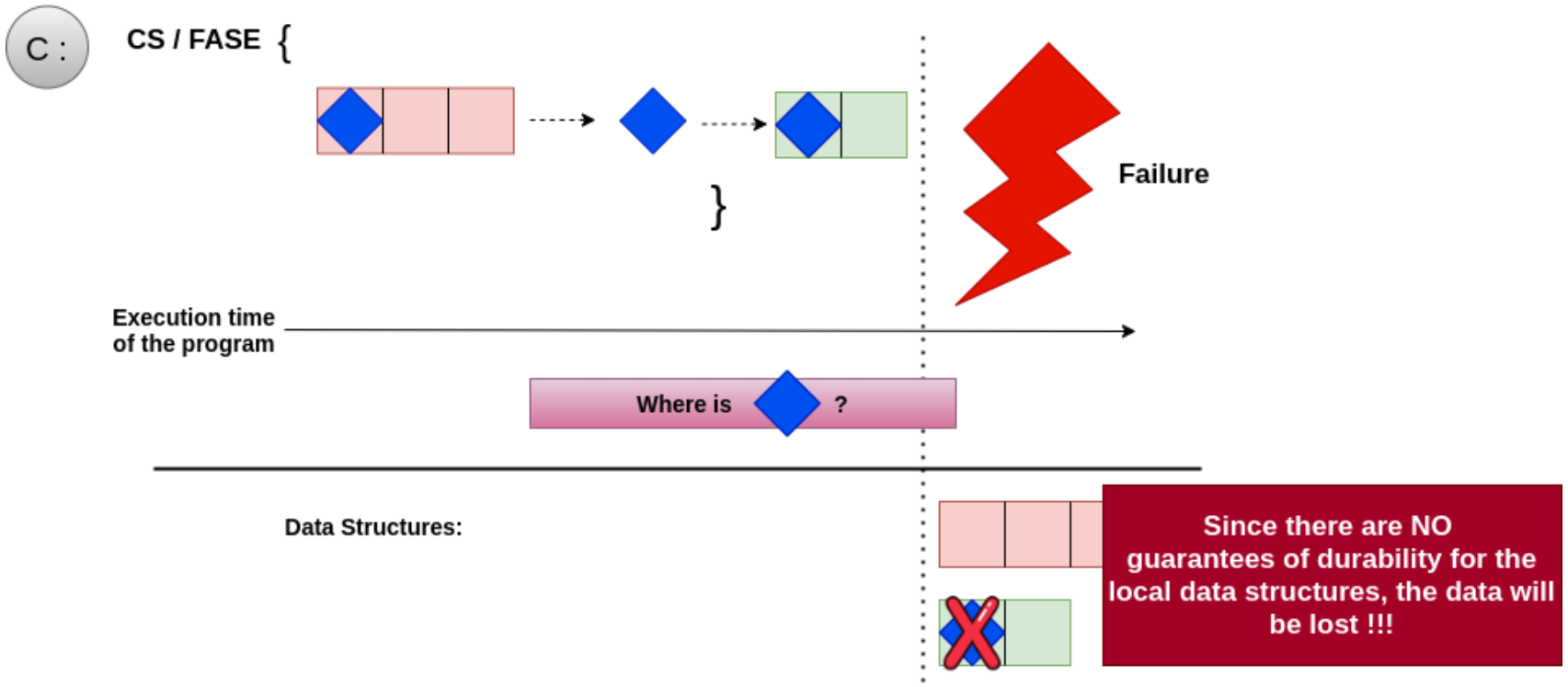
Allocation of the element

Where is the data element during the program execution?



Local data structures are also important !!!

What happens if a failure occurs at this point?



- With systems that offer transparency, in reality, programmer's intervention is still required for correctness.

Coming up...

Evaluation

Experimental setup and Workloads

Hardware and software setup:

- Two Intel Xeon Gold 5218 CPUs (64 logical cores)
- 384 GiB of DRAM and 1.5 TiB of Intel's Optane DC Persistent Memory
- Checkpoint frequency - 64 msec

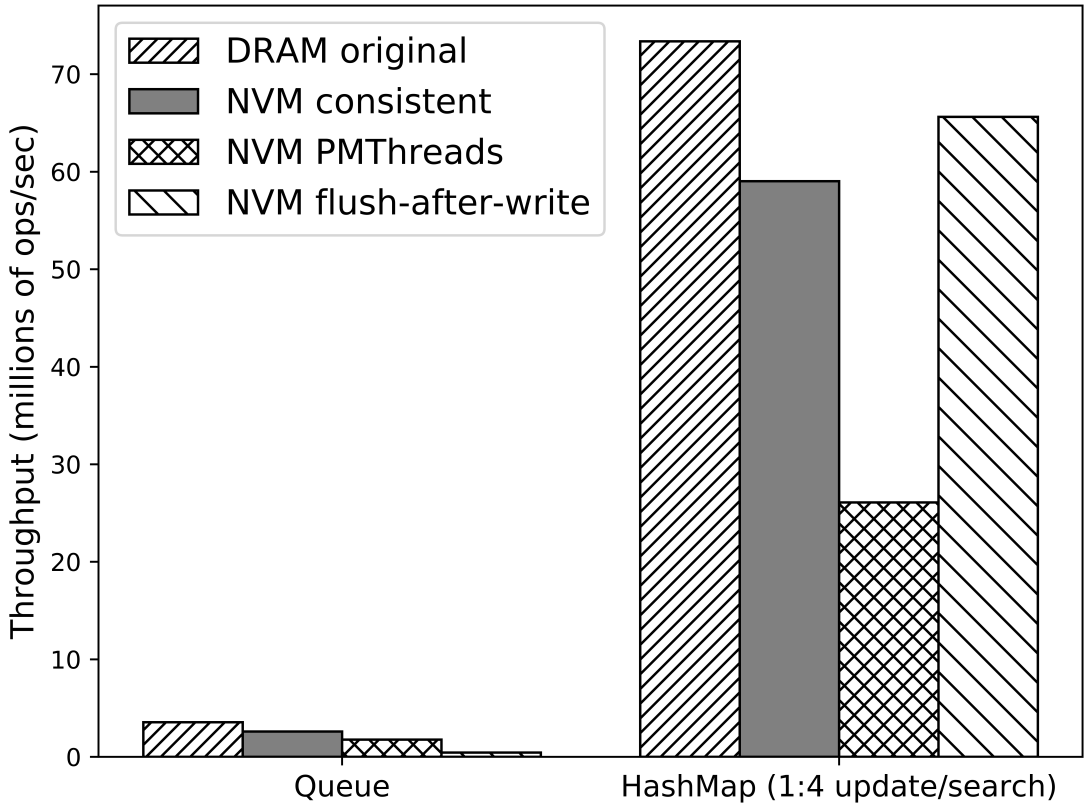
We evaluate 4 benchmarks:

- Highly efficient concurrent data structures: MS-Queue and HashMap
- A data-parallel computation: PARSEC Swaptions app
- A data compressing pipeline: PARSEC Dedup app

Performance with MS-Queue and HashMap

Using one socket

Comparison with PMThreads and solutions based on flush-after-write:

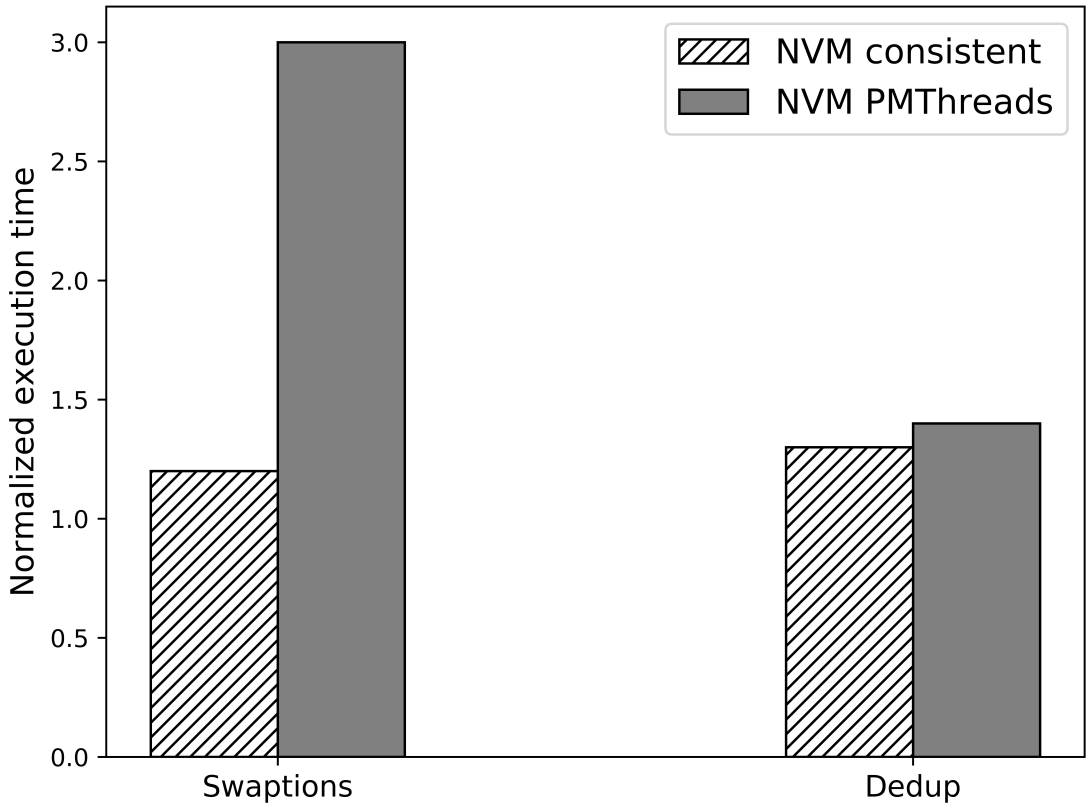


- Highest overhead for our solution: 35%
- More than 1.5× better than PMThreads
- Up to 6× better than flush-after-write

Performance with Parsec apps

Using two sockets

Comparison with PMThreads:



- Highest overhead for our solution: 30%
- Up to 2.5× better than PMThreads

Modifications to the applications

The modifications that our approach implies, in number of lines:

	Lines mod. inside CS	Lines mod. outside CS	Lines of code
Queue	14	13	590
Swaption	0	21	1135
Dedup	23	206	3351

Conclusion

In Conclusion:

- We illustrated that to achieve fault-tolerance, the programmers intervention is required even with the existing transparent solutions
- We presented a highly efficient in-NVMM checkpointing solution

By evaluating our algorithm on real hardware, we showed:

- The overhead of our solution is as low as 20-35%
- Our solution outperforms state-of-the-art techniques

Future Work:

- Automatizing the modification of the code that can be applied without programmer's intervention

Thank you for your attention

Trade-off - What do we prioritize?

- Failure-free execution performance
- Limiting the computation lost in the event of a failure

Listing 1: original code

```
1 for(int i=0; i<10; i++){
2     //update persistent x
3 }
```

Listing 2: modified code

```
1 //declare new volatile ↩
      variable: y
2 y=x;
3 for(int i=0; i<10; i++){
4     //update volatile y
5 }
6 x=y;
```

Performance with Swaptions app

The affect of Trade-off on performance