QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks

Thomas Shull, Jian Huang, Josep Torrellas
University of Illinois at Urbana-Champaign

April 14, 2019
VEE'19 Session 3

Programming NVM – The Good

- Emerging Non-Volatile Memory (NVM) devices offer an enticing combination of performance, capacity, and persistency
- Programs will no longer have to serialize data out to secondary storage for durability
- Now have access to persistent memory at a byte-level granularity

Real NVM Systems

- Real products with NVM are currently available:
  - Intel Optane SSDs
  - Viking NVRAM (Flash-backed DRAMs)
  - Intel Optane Persistent DC Modules (formally announced April 2)

Programming NVM – The Bad

Leveraging NVM to create persistent applications is tricky:

- Entire memory hierarchy is not durable
  - Processor caches are volatile
- Data must be written back from caches to achieve persistency
  - Perform combination of non-temporal stores and cacheline writebacks (CLWBs)
  - Fences (SFENCEs) must be inserted to guarantee writebacks have completed
- Software measures must be taken to ensure failure-atomicity for a collection of writes
  - Hardware only guarantees atomicity at cacheline level granularity
- To simply this process, frameworks for developing persistent applications are being introduced

Emerging Programmer-Friendly NVM Frameworks

- Initially, NVM frameworks' features closely matched the underlying hardware

- Now, new NVM frameworks are being introduced which drastically reduce the programmer burden

Presentation Outline

1. Summarize implementation details of emerging programmer-friendly NVM frameworks
2. Describe a significant overhead of current implementations (persistence checks)
3. Characterize behavior of persistence checks
4. Propose QuickCheck, a technique to *bias* persistence checks to limit their overhead
5. Evaluate QuickCheck's performance across volatile and non-volatile applications.

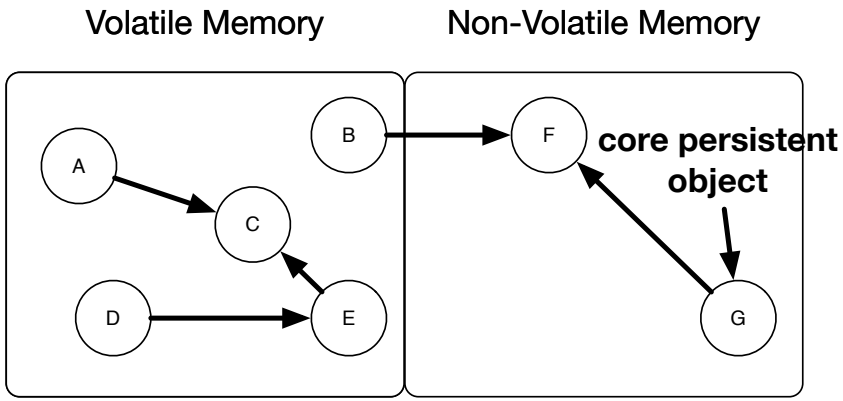
NVM Programming Model

- In this paper we describe and evaluate against the *AutoPersist* framework
- The AutoPersist framework attempts to minimizes the effort of creating persistent applications
- Its runtime dynamically detects which objects should be persistent, moves them to NVM, and inserts the necessary logging, CLWBs, and SFENCEs
- For more details, see forthcoming paper: *AutoPersist: An Easy-To-Use Java NVM Framework Based on Reachability* (PLDI'19)

NVM Programming Model

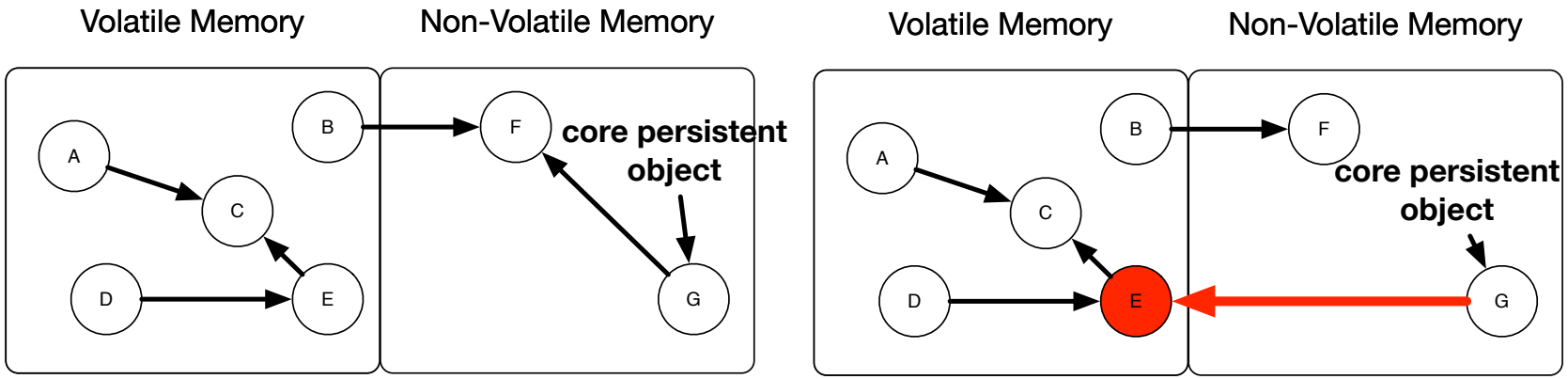
- In the AutoPersist programming model, the user must identify only a ***core persistent object set*** and failure-atomic regions
- The runtime then ensures all objects reachable from the core persistent object set are in NVM
  - *Transitive closure* of the core persistent object set is placed in NVM automatically
  - Requires dynamically moving objects to NVM throughout execution

Dynamically Moving Objects to NVM



(a) Initial Heap State

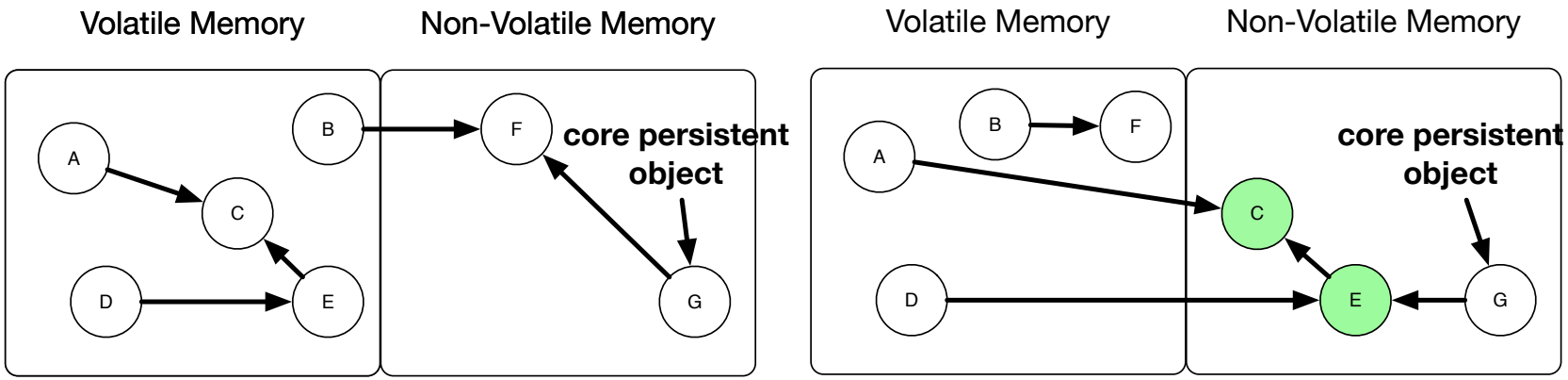
Dynamically Moving Objects to NVM



(a) Initial Heap State

(b) Model Violation

Dynamically Moving Objects to NVM



(a) Initial Heap State

(b) Runtime Maintaining Correct Heap State

Implementation Details

To provide its support, currently AutoPersist:

- Extends the semantics of several JVM bytecodes to perform the necessary runtime actions
  - Adds extra actions around stores & loads to/from object fields and arrays
    - Will not cover handling loads in this talk
    - Fully described in our paper
- These extra actions only execute when handling a persistent object
  - Actions are guarded by a *persistence check* and *persistence check branch*
  - Persistence check - check to determine if the object is persistent or not
  - Persistence check branch - the conditional branch to the runtime actions

Problem – Many Persistence Checks

- Model requires many persistence checks and conditional actions before stores (and loads) are performed
  - Storing to **core persistent object**?
  - Storing to an object reachable from a **core persistent object**?
  - In a failure-atomic region?
- Model also requires actions after store operations
  - Insert CLWB?
  - Insert SFENCE?

Store Field Example

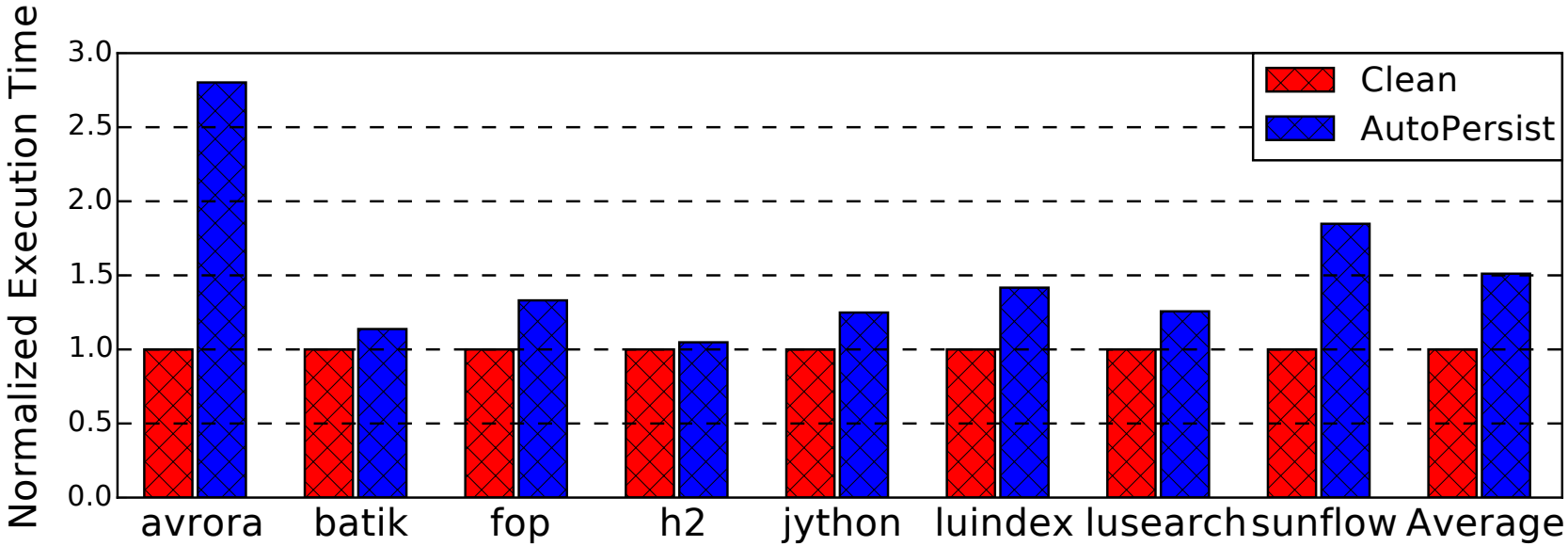
Modified store operation

```
 1: procedure storeField(Object o, Field f, Value v)
    [Start Persistence Check Code]
 2:     if isPersistent(o) then
 3:         Move value to NVM if necesssary
 4:         Log (object, field, value_old) tuple if in failure-atomic region
 5:     end if
    [End Persistence Check Code]
 6:     writeField(o, f, v)
    [Start Persistence Check Code]
 7:     if isPersistent(o) then
 8:         Add a cacheline writeback for the store
 9:         Add fence if not in failure-atomic region
10:     end if
    [End Persistence Check Code]
11: end procedure
```

Evaluating Overhead of Persistence Checks

- To determine the worst-case overhead of persistence checks, we run AutoPersist on the DaCapo Benchmark Suite
  - Since DaCapo benchmarks do not have persistence markings, the actions guarded by persistence checks will never be activated
- AutoPersist runs on a modified version of Maxine 2.0.5
  - Maxine is a research JVM originally developed by Oracle
- Compare AutoPersist against an unmodified version of Maxine (*Clean* configuration)
- Run on real Intel system with NVM (Intel Optane DC persistent memory modules)

Evaluating Overhead of Persistence Checks



- *AutoPersist* is up to 180% slower than *Clean*
- On average, *AutoPersist* is 51.1% slower than *Clean*

QuickCheck Contribution

In our paper we make the following contributions:

- Characterize how often actions guarded by persistence checks are activated in persistent applications
- Propose QuickCheck, a technique to *bias* each persistence check branch based on profiling information collected during warmup
- Implement QuickCheck on top of the AutoPersist framework
- Achieve significant speedups across both volatile and non-volatile applications against AutoPersist

Characterizing Behavior of Persistence Checks

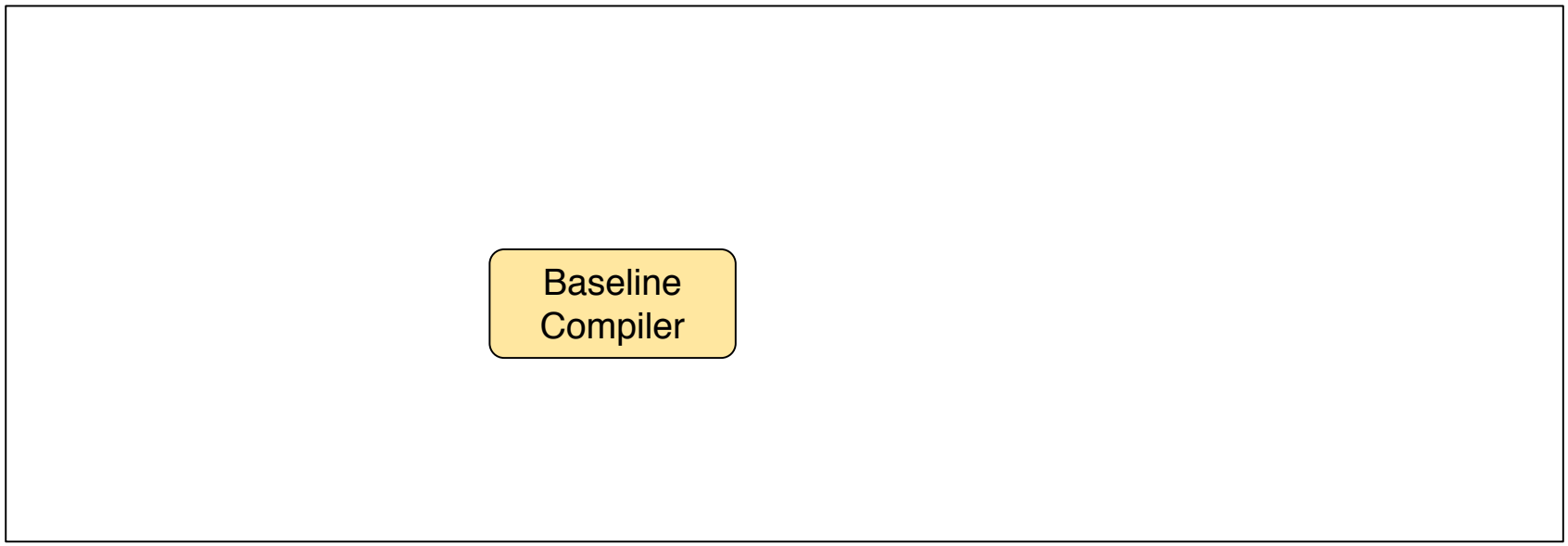
- We record the behavior of each action site guarded by a persistence check
- Categorize each action into one of three categories based on how frequently the action is activated:
  - *Always Activated*, *Sometimes Activated*, or *Never Activated*
  - Categorize on a per action site basis
- Run YCSB benchmark suite on two different AutoPersist implementations of Memcached (have different storage backends)

Data Structure	Always	Sometimes	Never
Functional Map	0.03%	0.05%	99.86%
B^+ Tree	0.09%	0.02%	99.87%

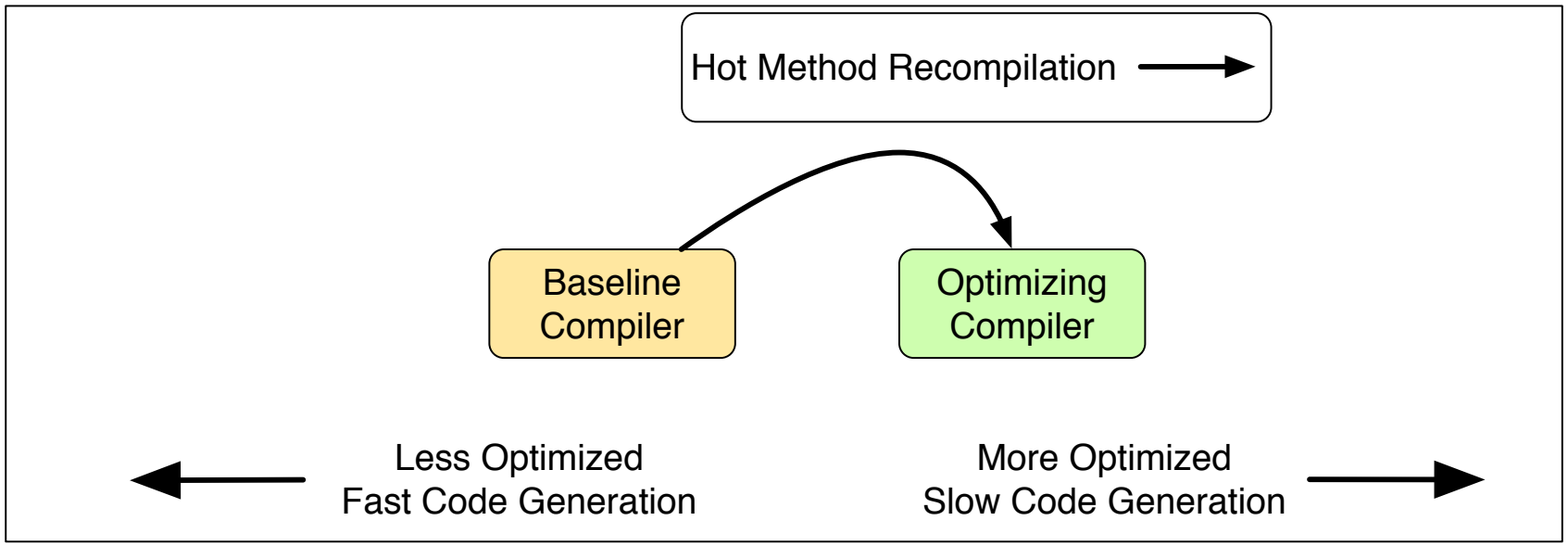
QuickCheck Main Idea

- Leverage multi-tiered compilation to improve the performance of persistence checks
  - Have distinct profiling and optimization phases
- Can predict whether a persistence check's guarded action will be activated or not
  - Use this information to influence code generation of persistence check branches
- Very few persistence check sites have activated actions
  - Can further eliminate the overhead of persistence checks by *speculatively removing these actions* from the code

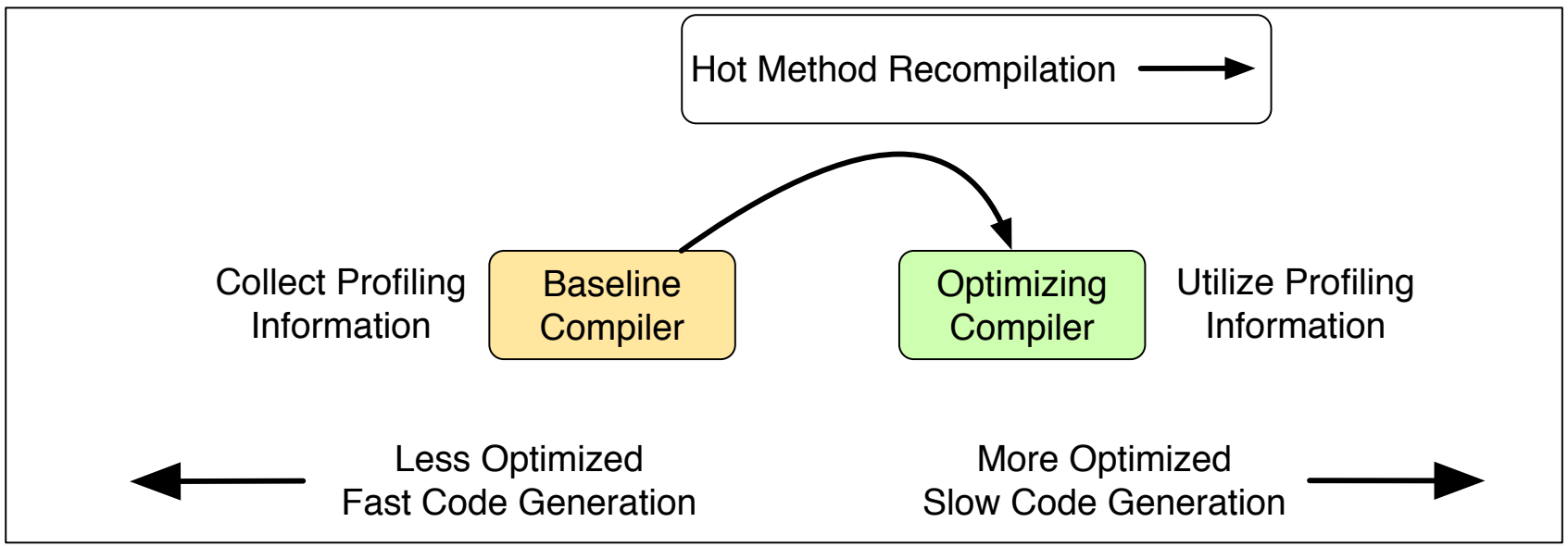
Normal Java Execution Process



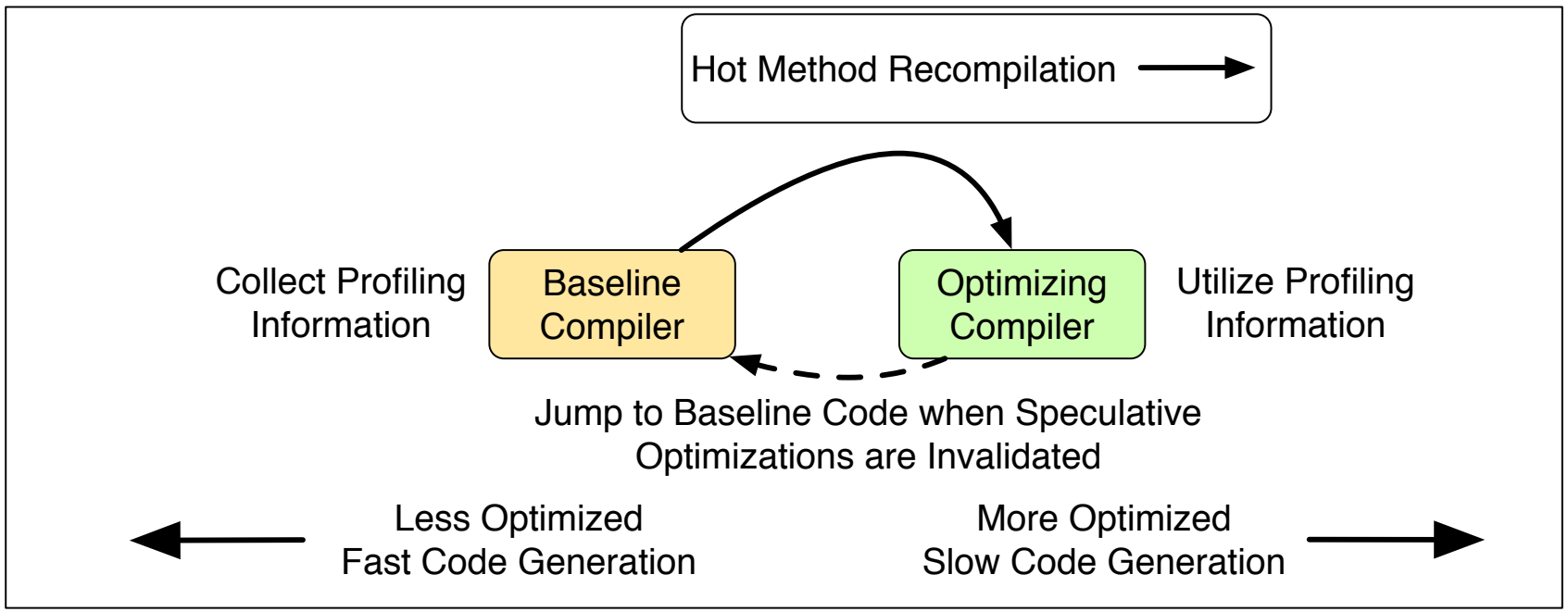
Normal Java Execution Process



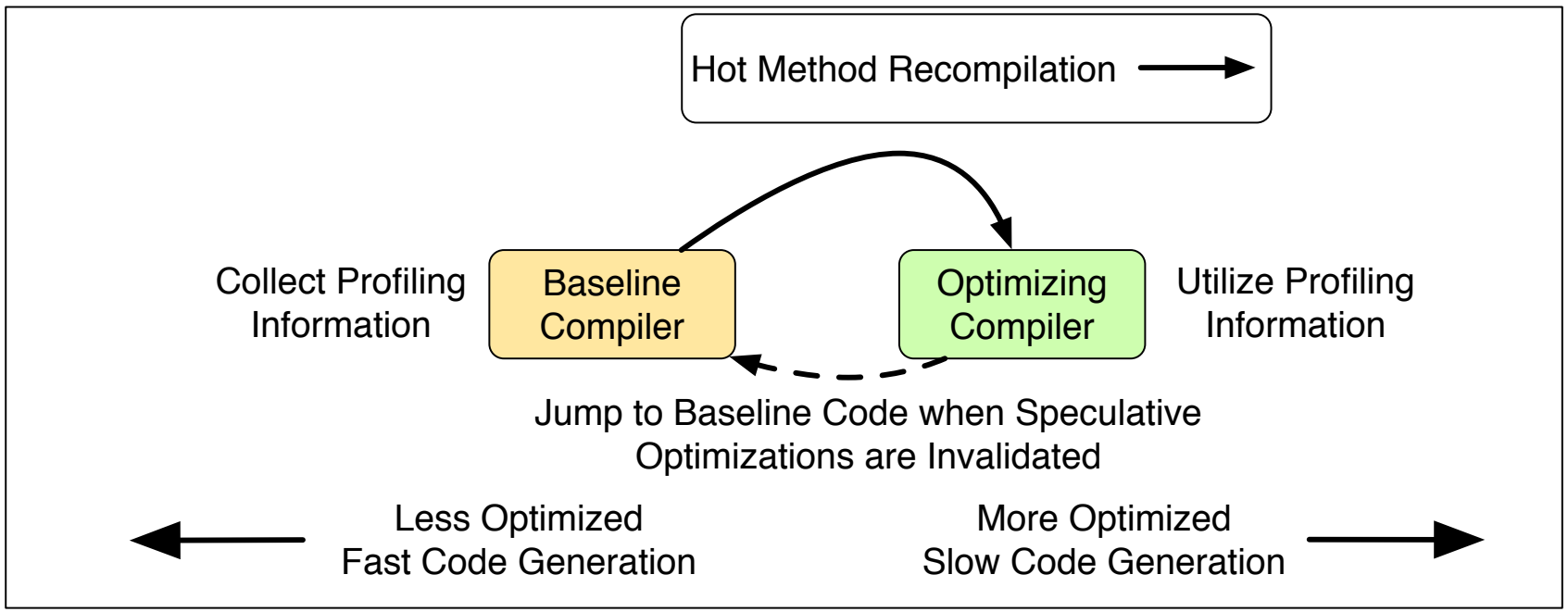
Normal Java Execution Process



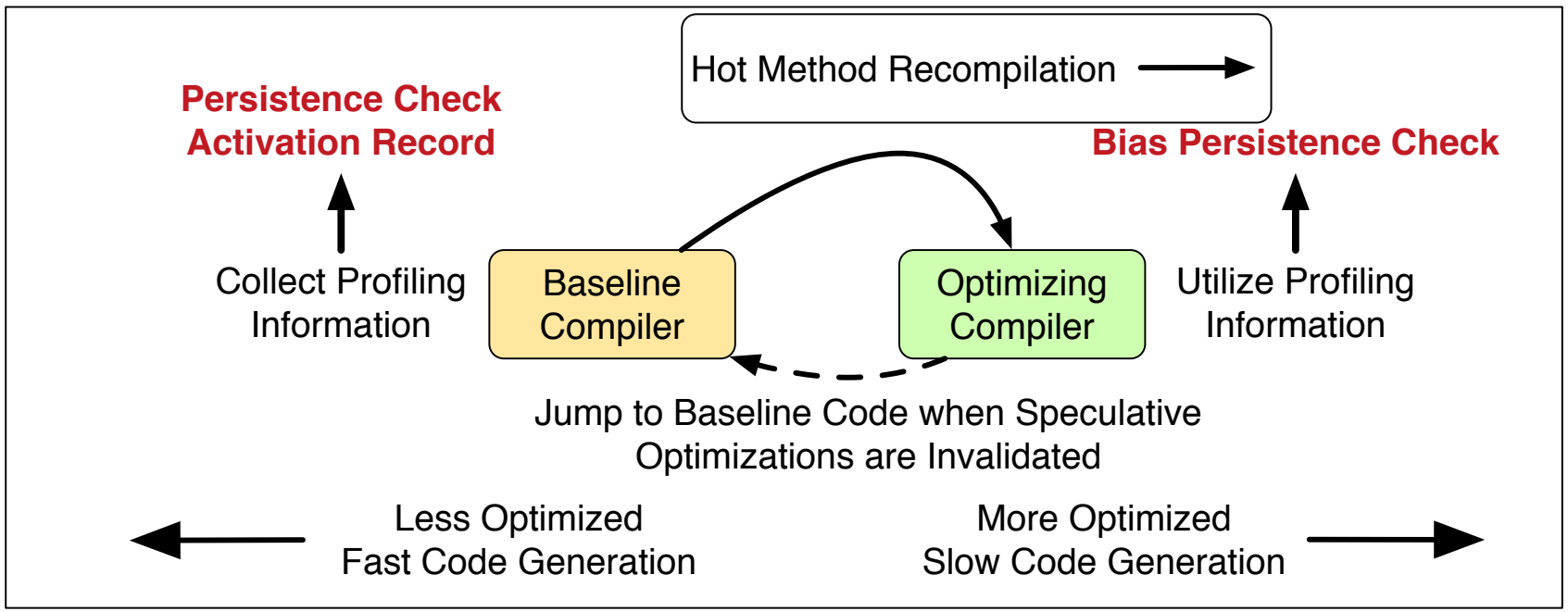
Normal Java Execution Process



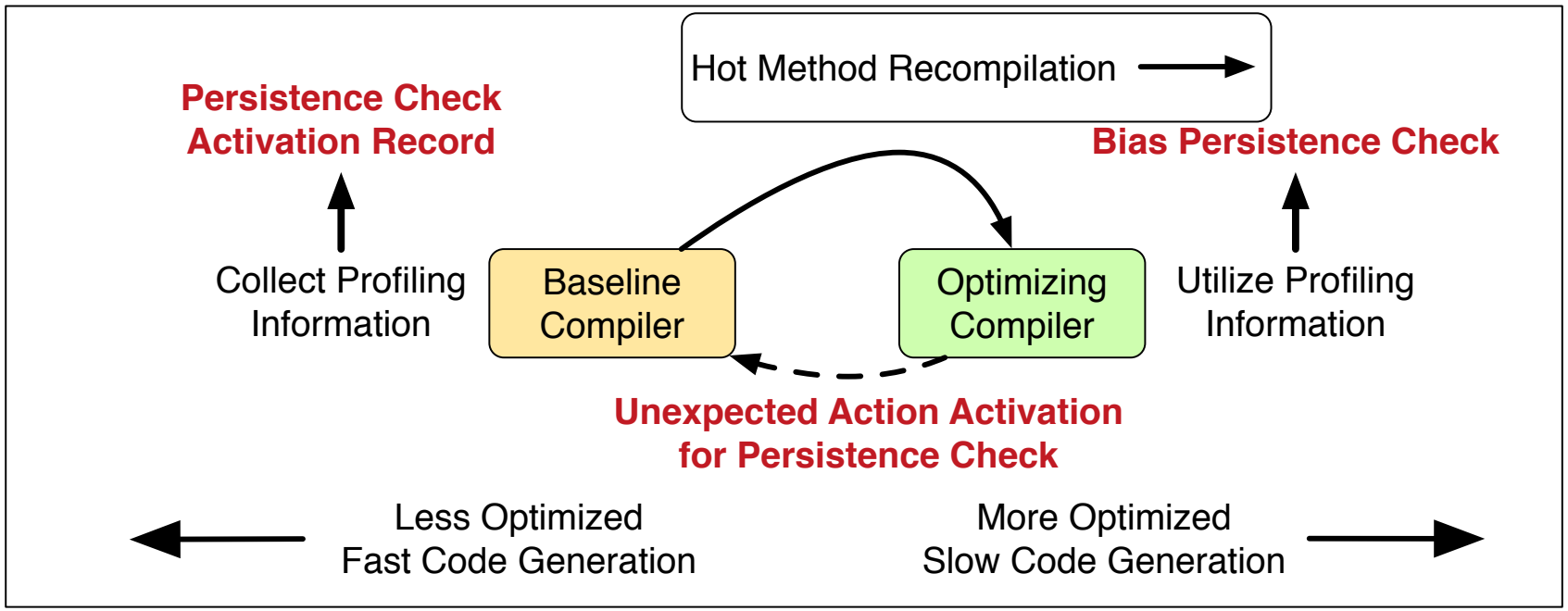
Visual Representation of QuickCheck Main Idea



Visual Representation of QuickCheck Main Idea



Visual Representation of QuickCheck Main Idea



Profiling and Determining Persistence Check Bias

- In Baseline Compiler:
  - Add two new profile counters for each check site: `activated` and `bypassed`
  - Counters are dynamically updated during execution as persistence checks are encountered
- In Optimizing Compiler:
  - During recompilation, categorize each persistence check site based on its action activation rate:
  - ***likely:*** activated over 95% of the time.
  - ***unbiased:*** activated 5%–95% of the time.
  - ***unlikely:*** activated less than 5% of the time.
  - ***very_unlikely:*** not activated during profiling.

Biasing Persistence Checks

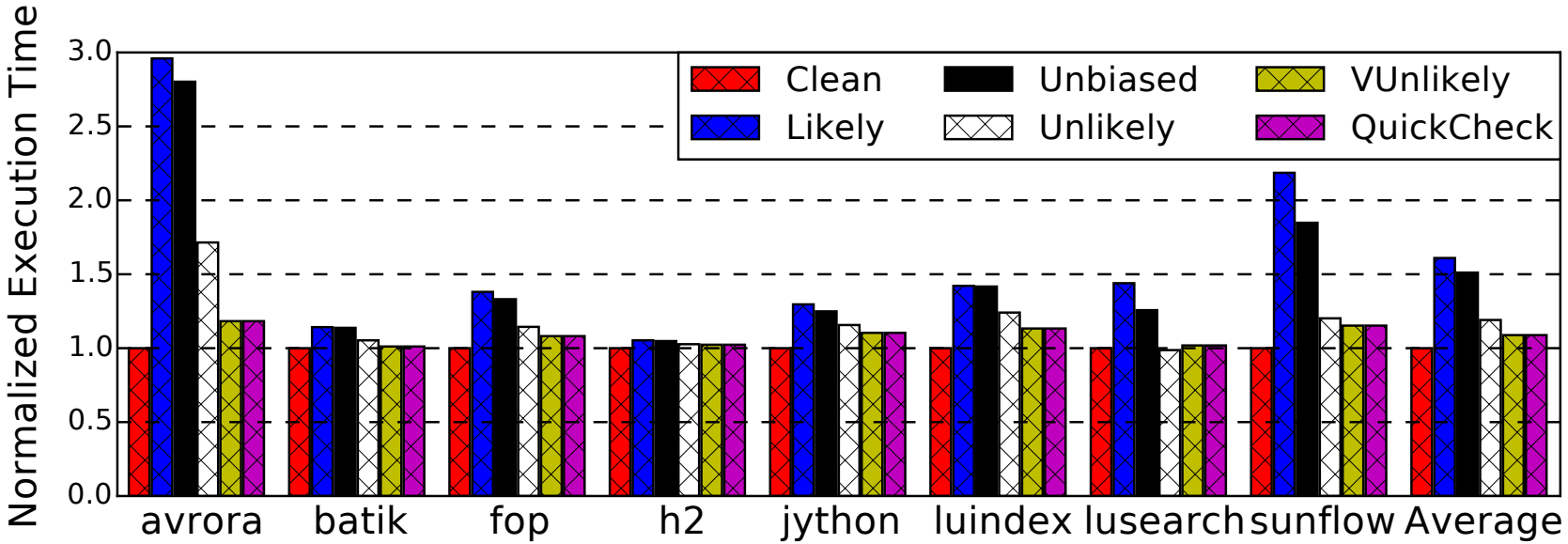
- Generate code for each persistence check based on its assigned category
- ***likely***, ***unbiased***, and ***unlikely*** persistence checks:
  - Set the taken weight of the persistence check branch to 95%, 50%, and 5%, respectively
  - Will cause the compiler to optimize code to favor the expected branch path
- ***very_unlikely*** persistence checks:
  - Set the taken weight of the persistence check branch to 1%
  - Remove guarded action from code
    - Add a jump back to baseline code in its place

Very_Unlikely Store Field Example

New store operation for very_unlikely biased persistence check

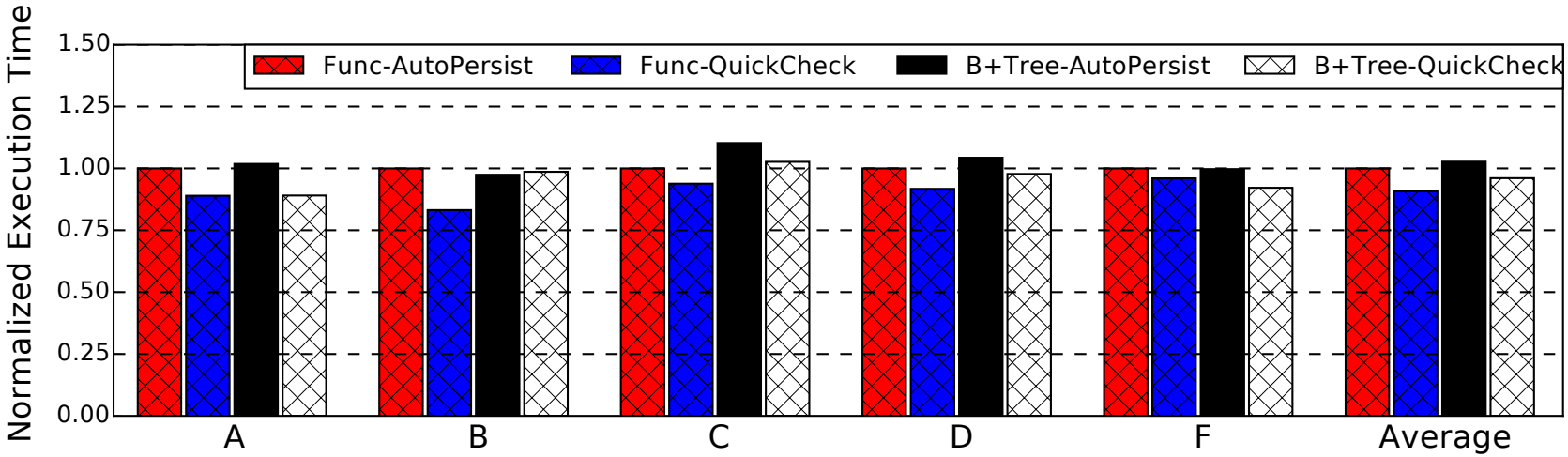
```
1: procedure STOREFIELD(Object o, Field f, Value v)
   [Start Persistence Check Code]
2:     if isPersistent(o) then
3:         [Jump back to Baseline Code]
4:     end if
   [End Persistence Check Code]
5:     writeField(o, f, v)
6: end procedure
```

QuickCheck Results – DaCapo



- On average, Likely, Unbiased, Unlikely, and VUnlikely perform 61%, 51%, 19%, and 8.8% worse than Clean
- Results show setting branch weights within the compiler can significantly impact performance
- QuickCheck performs the same as VUnlikely (8.8% worse)

QuickCheck Results – YCSB



- On average, QuickCheck is 9.3% and 6.5% faster than AutoPersist for the Functional Map and B$^+$ Tree backends
- Improvements much smaller than DaCapo due to the overhead of CLWBs and SFENCEs needed when performing persistent stores

Opportunities for Further Improvements

- Bias branches within persistence check's guarded action
- Adjust branch weights at a finer granularity
- Implement QuickCheck in other ISAs
  - Instead of converting *very_unlikely* checks to a branch and trap, can use predictated instructions to cause traps on check failures

Extra Details in Paper

Our paper contains many more details, including:

- Why load JVM bytecodes also need persistence checks
  - AutoPersist sometimes converts volatile objects into *forwarding* objects
- How QuickCheck profiles and biases persistence checks within load JVM bytecodes
- Extra evaluation
  - Check overhead on Scala DaCapo benchmarks
  - Other Memcached Backends
  - QuickCheck's performance on kernels

QuickCheck Summary

- Identified that persistence checks have significant overheads, yet very predictable behaviors
- Proposed QuickCheck, a technique to *bias* persistence checks to reduce their overhead
- Described how to apply our biasing technique to influence code generation
- Achieved significant speedups across both volatile and non-volatile applications against an existing NVM framework

QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks

Thomas Shull, Jian Huang, Josep Torrellas
University of Illinois at Urbana-Champaign

April 14, 2019
VEE'19 Session 3