TECHNISCHE UNIVERSITÄT DRESDEN

CENTER FOR ADVANCING ELECTRONICS DRESDEN

Using **transactional memory** to build **resilient systems**

Christof Fetzer
TU Dresden

Resilience?

- **resilient computing**:
  - „*refers to the ability to provide and maintain an acceptable level of service in the face of **faults** and **challenges** to normal operation*“ [Laprie]

Motivation

- To decrease the cost of resilience, we are interested in approaches that can deal with
  - software bugs
  - hardware faults
  - operational issues

Failure Atomicity

- Can we use transactional memory to ensure **failure atomicity**?

Failure Atomicity!

- A method **m** of class **C** is **failure atomic** iff m guarantees that even in the **face of failures**:
    - **no resource leaks**
    - **all invariants hold**
    - m either **succeeds** or **state is unchanged**

Example

- HashMultiMap (downloaded from web):

```
class HashMultiMap {
  int nb_elements;
  Bucket[] b;
  // ...
  void add(Object k, Object v) {
    nb_elements++;
    b[k.hashCode()%b.length].
      append(new Pair(k, v));
  }
  int size() {
    return nb_elements;
  }
}
```

k is null

NullPointer Exception

Inconsistency:
nb_elements incremented but element not inserted

C. Fetzer, K.Högstedt, P. Felber, Automatic Detection and Masking of Non-Atomic Exception Handling, IEEE Transactions on Software Engineering, 2004.

Fixing Example

- We can try to fix the code:

```
void add(Object k, Object v) {
  if(k == null || v == null)
    throw new Exception(...);
  // Add element...
  nb_elements++;
}
```

C. Fetzer, K.Högstedt, P. Felber, Automatic Detection and Masking of Non-Atomic Exception Handling, IEEE Transactions on Software Engineering, 2004.

Fixing Example

- We can try to fix the code:

```
void add(Object k, Object v) {
  if(k == null || v == null)
    throw new Exception(...);
  // Add element...
  nb_elements++;
}
```

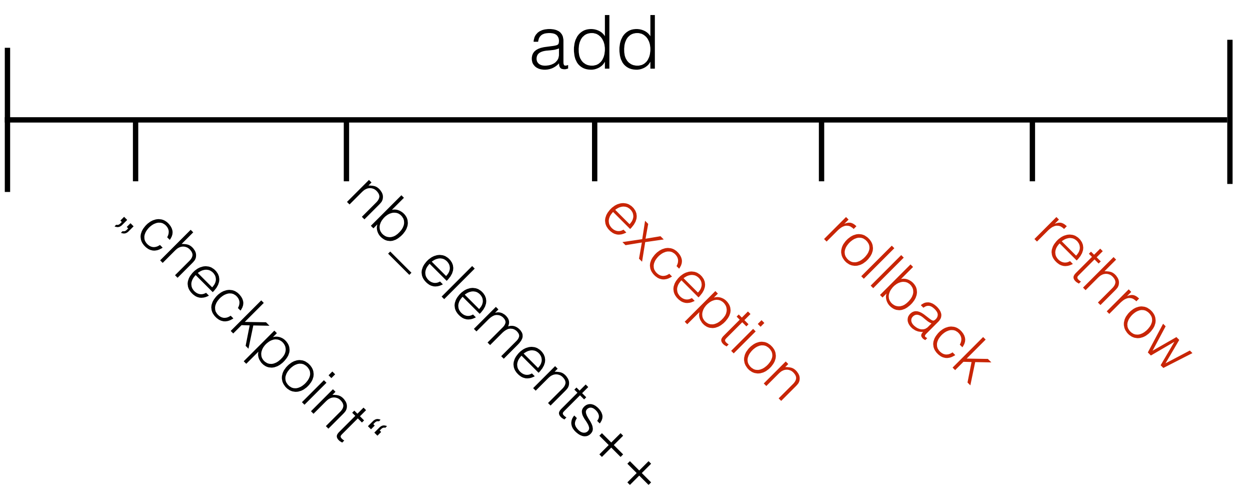
- **General approach:**
  - sort statements such that
    1. execute statements that could throw exceptions
       - they must not change state of data structure
    2. perform state changes with no-throw functions

Alternative

- Ensuring failure atomicity manually could result in fragile code

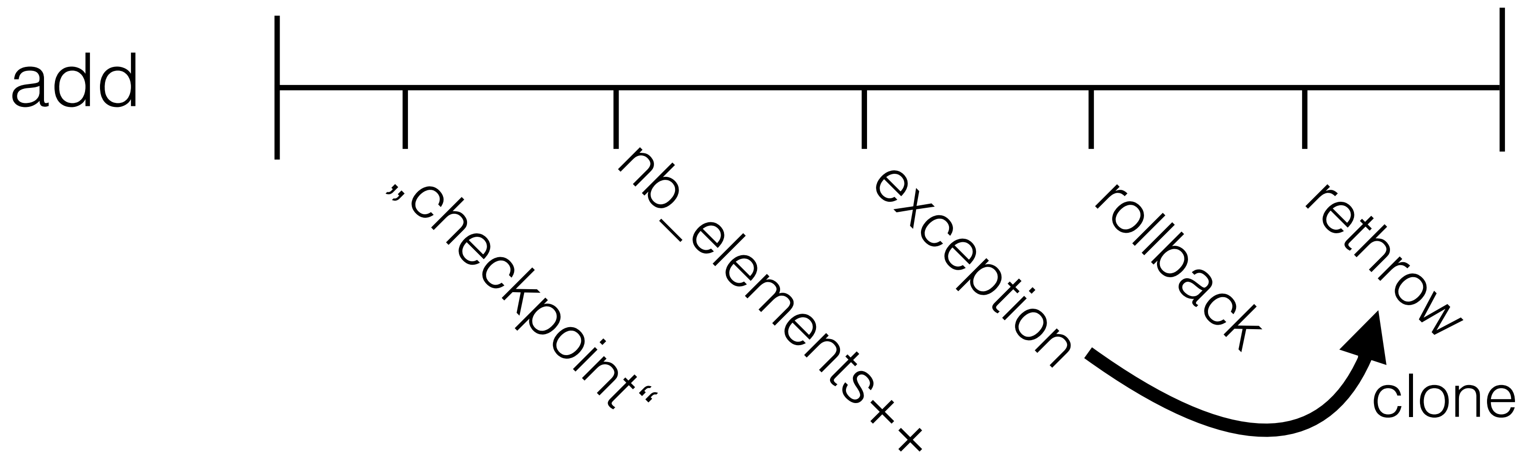
- **Alternative**: ensure failure atomicity using transactional memory

```
class HashMultiMap {
  int nb_elements;
  Bucket[] b;

  @failureAtomic
  void add(Object k, Object v) {
    nb_elements++;
    b[k.hashCode()%b.length].
      append(new Pair(k, v));
  }
  int size() {
    return nb_elements;
  }
}
```



Problem 1

- **Problem:**
  - **exceptions might result in resource leaks & leaked information**
- **My take:**
  - compiler should only permit to throw exception objects within failure-atomic blocks that can be cloned.



Composable Error Recovery with Transactional Memory (Torvald Riegel, Pascal Felber, Christof Fetzer), In Bulletin of the European Association for Theoretical Computer Science (BEATCS), volume 99, 2009.

Problem 2

- **Problem:**
  - **atomic block can only roll-back internal state**
- **My take:**
  - compiler should flag/ prevent external state changes in failure-atomic blocks.

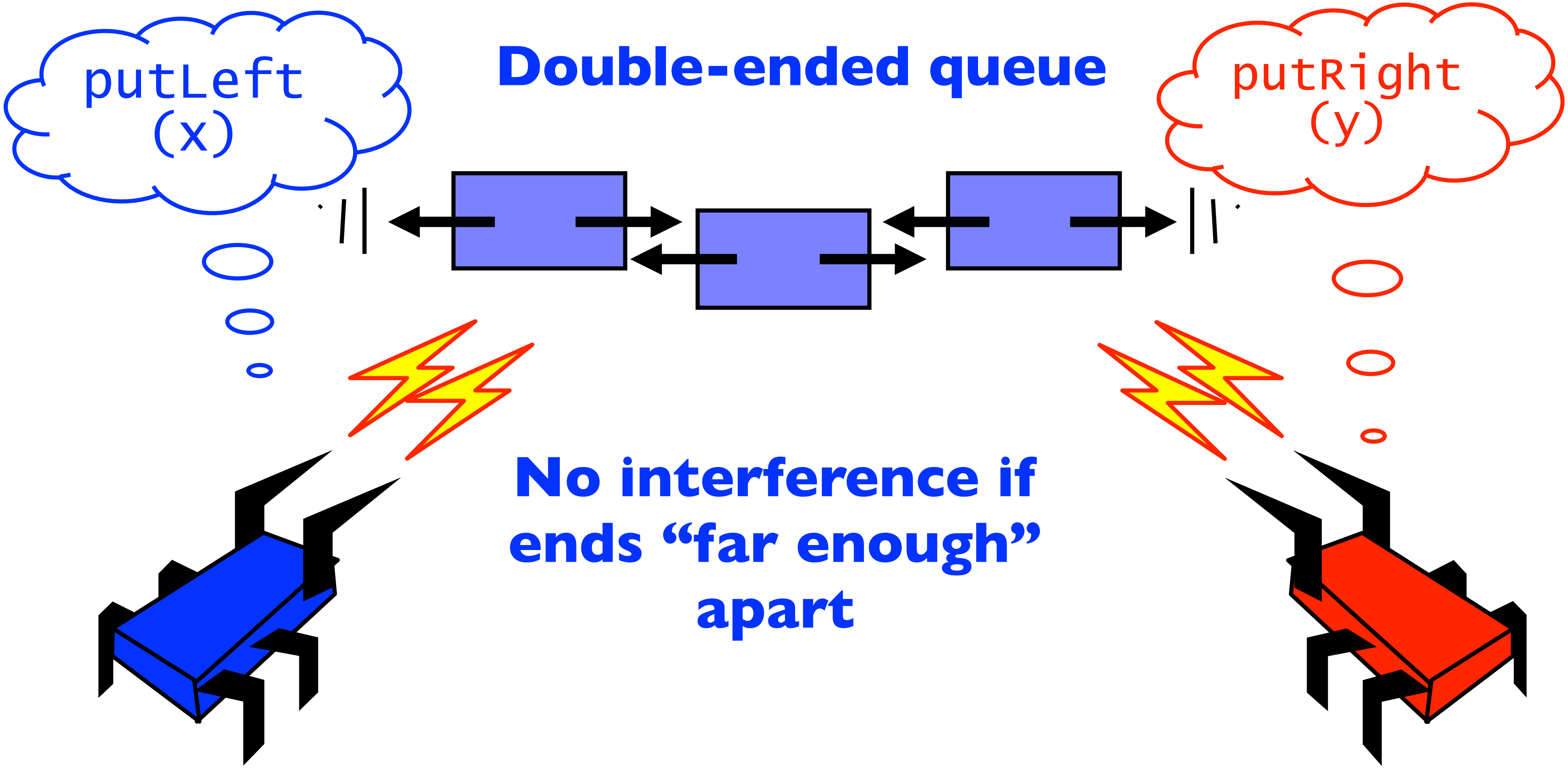
```
class HashMultiMap {
  int nb_elements;
  Bucket[] b;

  @failureAtomic
  void add(Object k, Object v) {
    nb_elements++;
     tcp.send();
    b[k.hashCode()%b.length].
      append(new Pair(k, v));
  }
  int size() {
    return nb_elements;
  }
}
```

Composable Error Recovery with Transactional Memory (Torvald Riegel, Pascal Felber, Christof Fetzer), In Bulletin of the European Association for Theoretical Computer Science (BEATCS), volume 99, 2009.

Problem 3

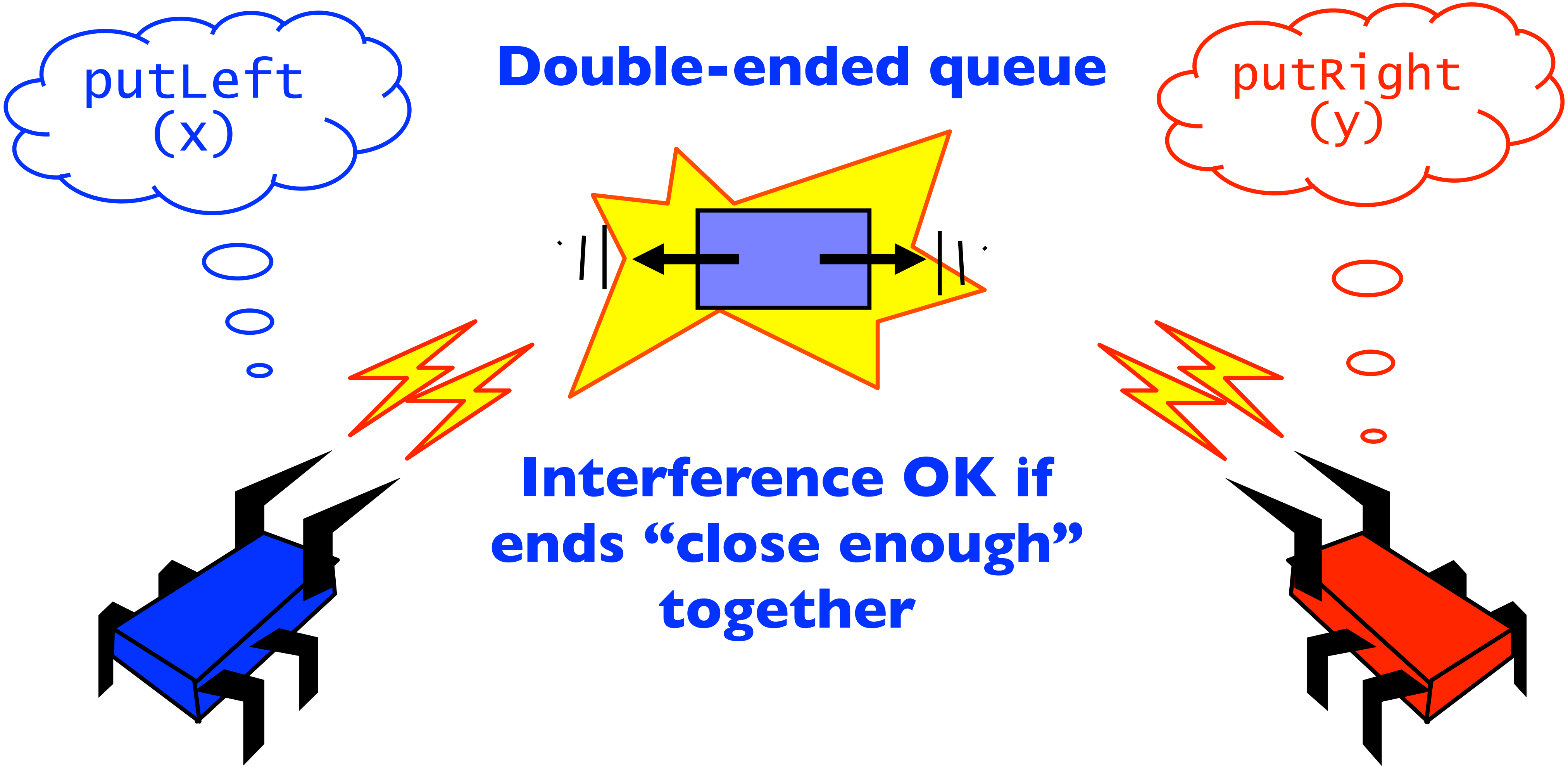
- **Problem**:
  - difficult to get such extensions into C / C++.
- **Question**:
  - does it makes sense to add **atomic** and **failure-atomic** blocks in a still evolving language (like Rust from Mozilla)?

Sadistic Homework



based on a slide by Nir Shavit and Maurice Herlihy

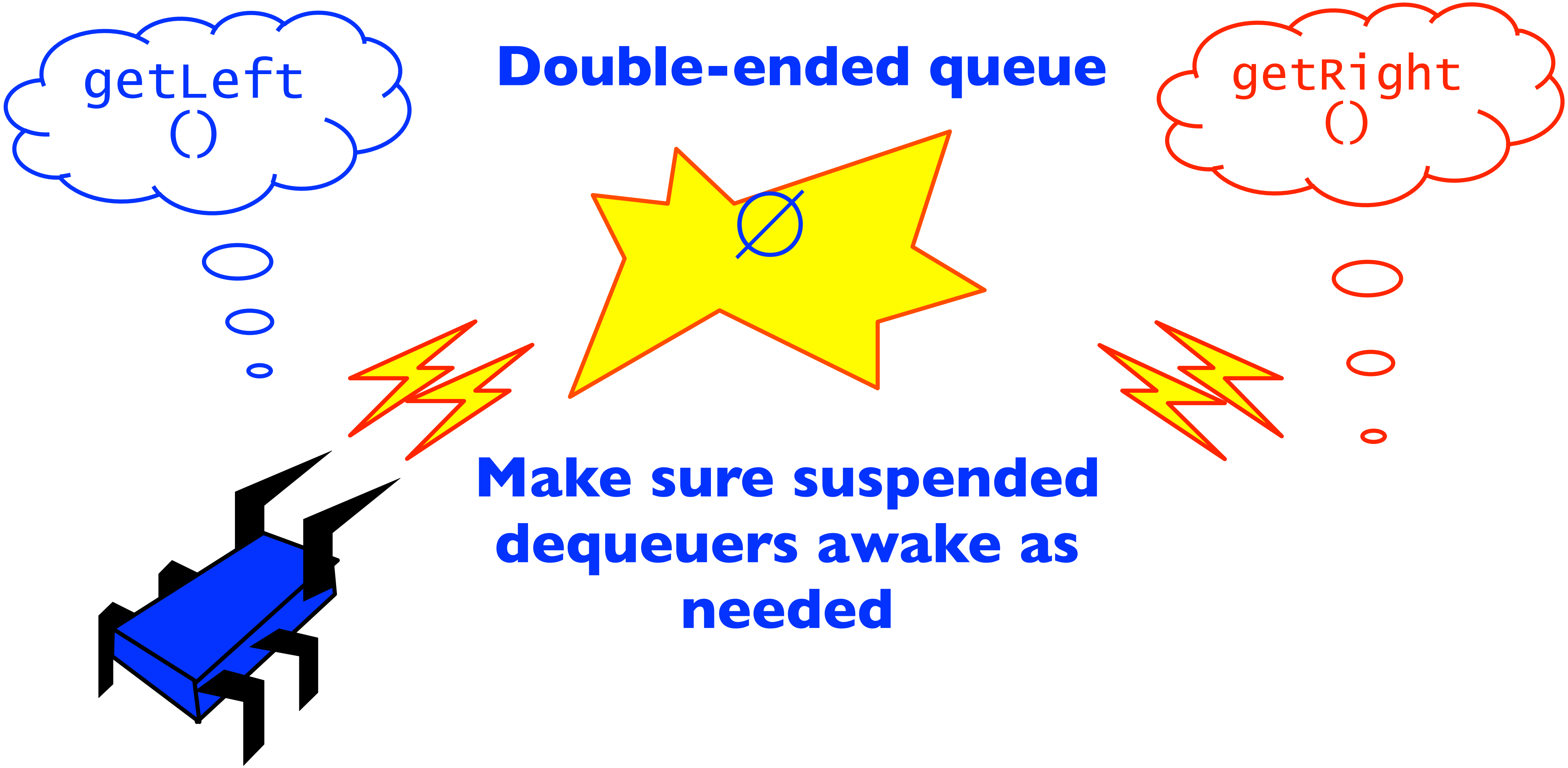
Sadistic Homework



based on a slide by Nir Shavit and Maurice Herlihy

Sadistic Homework

Double-ended queue



Make sure suspended dequeuers awake as needed

based on a slide by Nir Shavit and Maurice Herlihy

Solution

- PODC 1996, Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.

Maged Michael

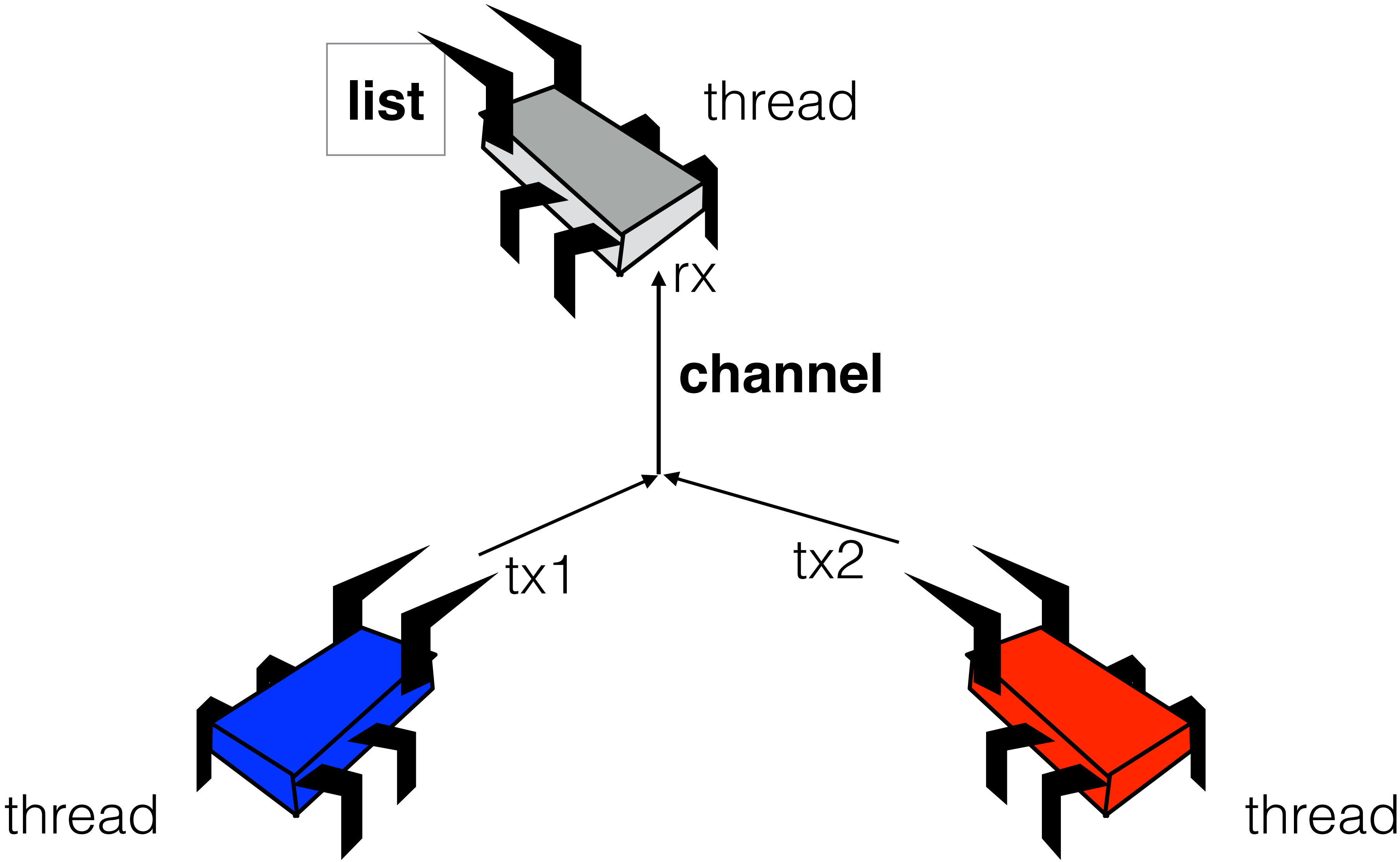
Michael Scott

Say, you are not a concurrency expert…

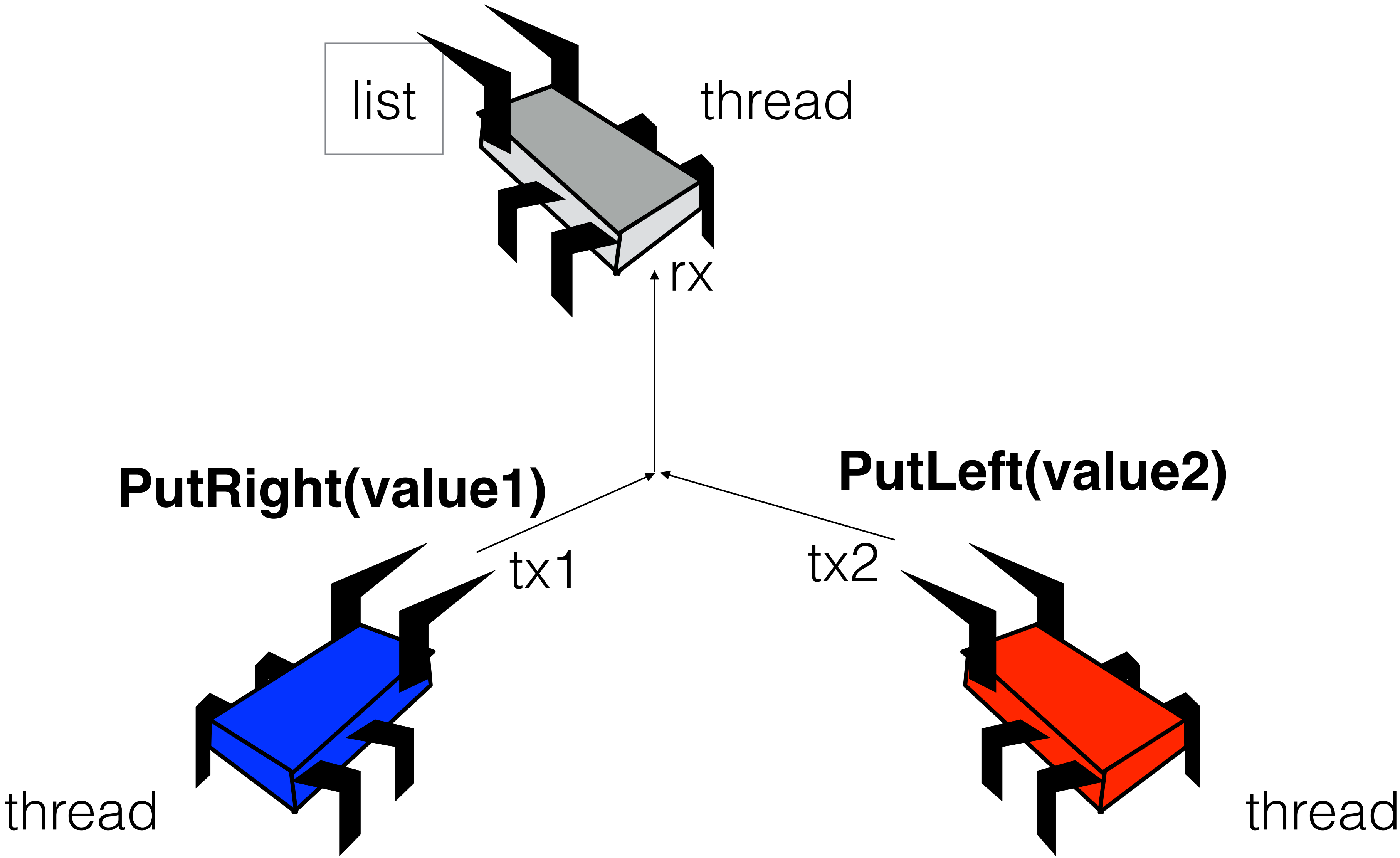
- If problem is too difficult to be solved, change the problem:
  - i.e., solve a variant of the problem!
- **Variant:**
  - Concurrent puts
  - Sequential gets

Captain Kirk

Actor Approach



Actor Model



list
thread
rx
PutRight(value1)
PutLeft(value2)
tx1
tx2
thread
thread

Actor Approach

```
let t = Thread::spawn(move || {
        let mut list : Vec<uint> = Vec::new();
        loop {
              let m = rx.recv();
              match m {
                     Msg::GetLeft(tx) => tx.send(list.remove(0)),
                     Msg::GetRight(tx)=> tx.send(list.pop()),
                     Msg::PutLeft(v)  => list.insert(0u, v),
                     Msg::PutRight(v) => list.push(v),
                     Msg::Terminate   => return,
               };
        }
       });
```

(rust code)

- **actor**: is a thread
  - waiting for messages
  - depending on what message type arrives, we add/remove elements from the list (for which we use a rust vector)

Atomic Block?

```
let t = Thread::spawn(move || {
        let mut list : Vec<uint> = Vec::new();
        loop {
                let m = rx.recv();
                match m {
                        Msg::GetLeft(tx) => tx.send(list.remove(0)),
                        Msg::GetRight(tx)=> tx.send(list.pop()),
                        Msg::PutLeft(v)  => list.insert(0u, v),
                        Msg::PutRight(v) => list.push(v),
                        Msg::Terminate   => return,
                 };
        }
       });
```

- Implicit atomic block between recv & send?
- On **abort:**
  - no state change & send error message

Implicit Atomic Block?

```
let t = Thread::spawn(move || {
        let mut list : Vec<uint> = Vec::new();
        loop {
              let m = rx.recv();
              match m {
                     Msg::GetLeft(tx) => tx.send(list.remove(0)),
                     Msg::GetRight(tx) => tx.send(list.pop()),
                     Msg::PutLeft(v)   => list.insert(0u, v),
                     Msg::PutRight(v) => list.push(v),
                     Msg::Terminate    => return,
               };
        }
       });
```

- **PutLeft/PutRight**: does not return a value
    - no possibility to indicate error!

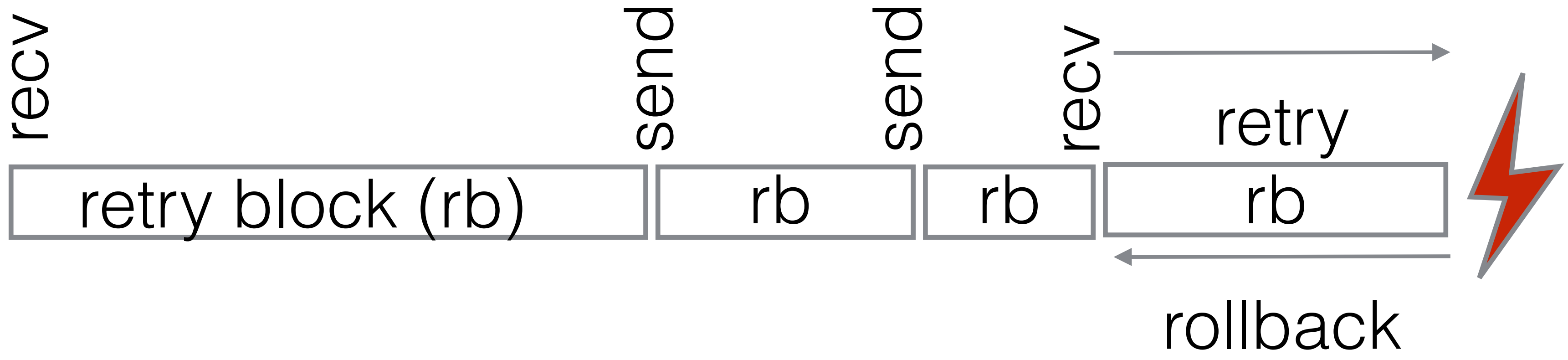
Observation 1

- **No conflict detection needed**
  - Rust is a data race-free language
- Simple **all-or-nothing** semantics is insufficient:
  - we need to have „**all**“ semantics!
- On **nothing**:
  - there is no good alternative except to **panic!**
    - **panic!** propagates via channels.

Observation 2

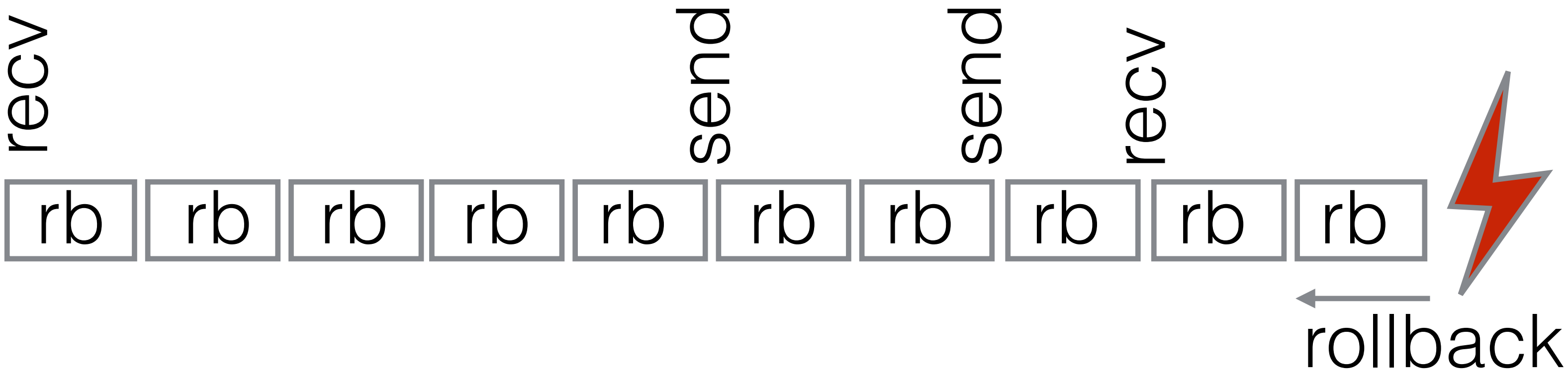
- high likelihood that **transparent retry** will mask
  - **transient hardware faults** and
  - (certain) **transient software bugs**

Masking Transient Hardware Faults



- Transient hardware faults are expected to go up
  - with decreasing feature sizes
- Safety critical systems (e.g., automotive):
  - need to **detect** transient HW faults (e.g., 99%), and
  - and to mask transient HW faults

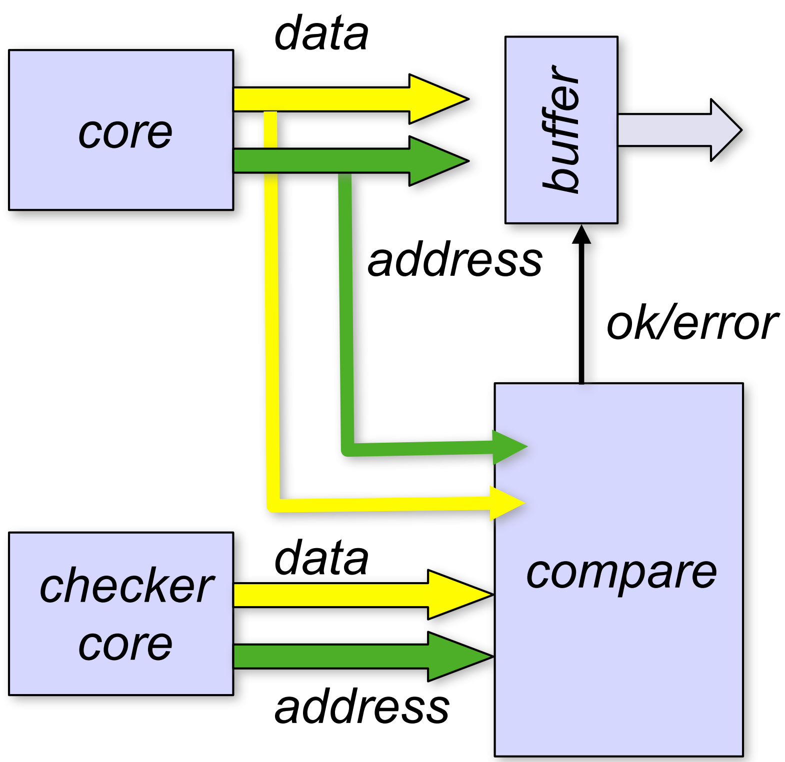
Observation 3



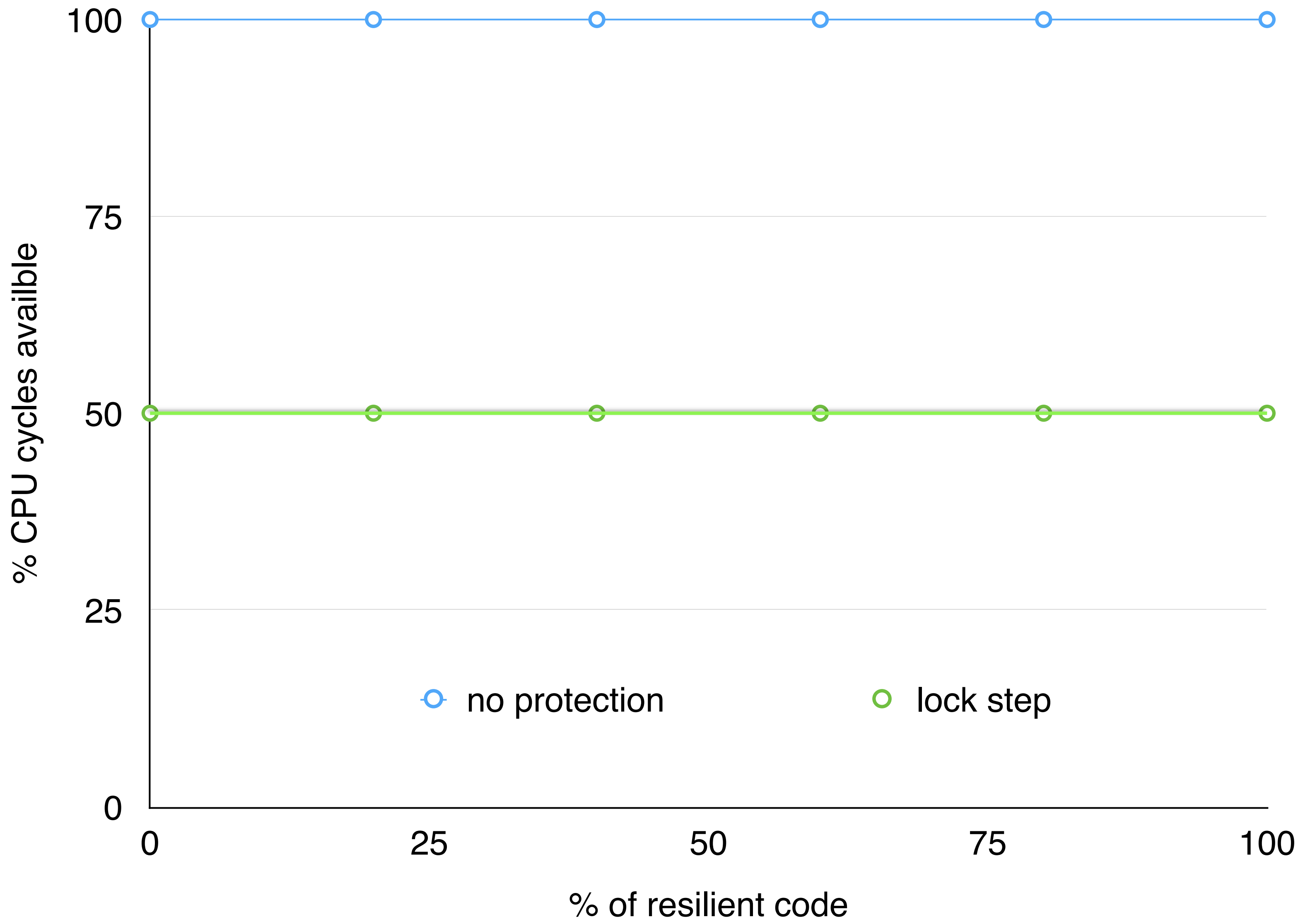
- **Retry blocks:**
  - flexibility to choose the size
  - one needs to commit before externalizing state
- **Transient software bugs**:
  - success of retry most likely proportional to transaction size

Detection: Lock Step Cores

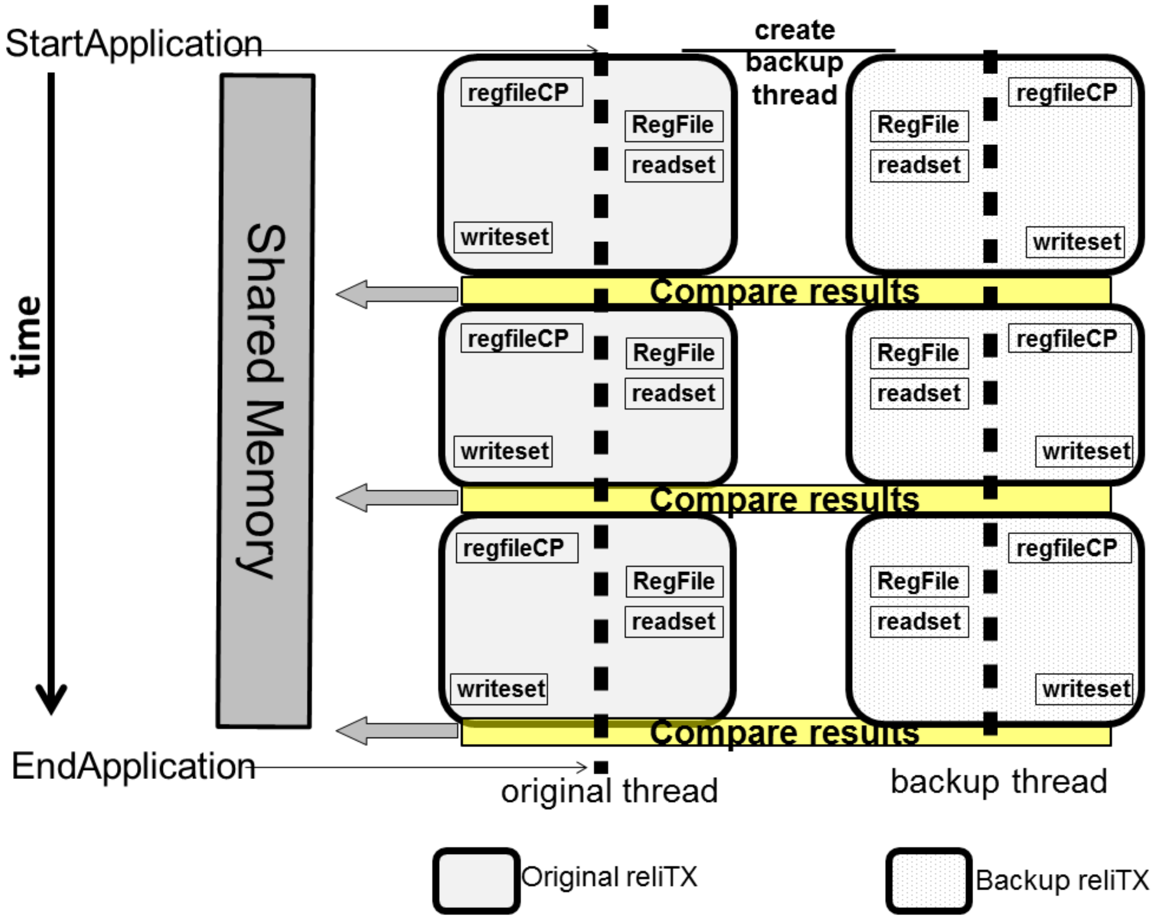
- **Lock-step cores:**
  - check that external behavior is the same
- **Problems**:
  - cores are non-deterministic behavior
  - 100% cycle overhead



Lock Step Cores



FaultTM



- Compare writesets instead of result of individual instructions

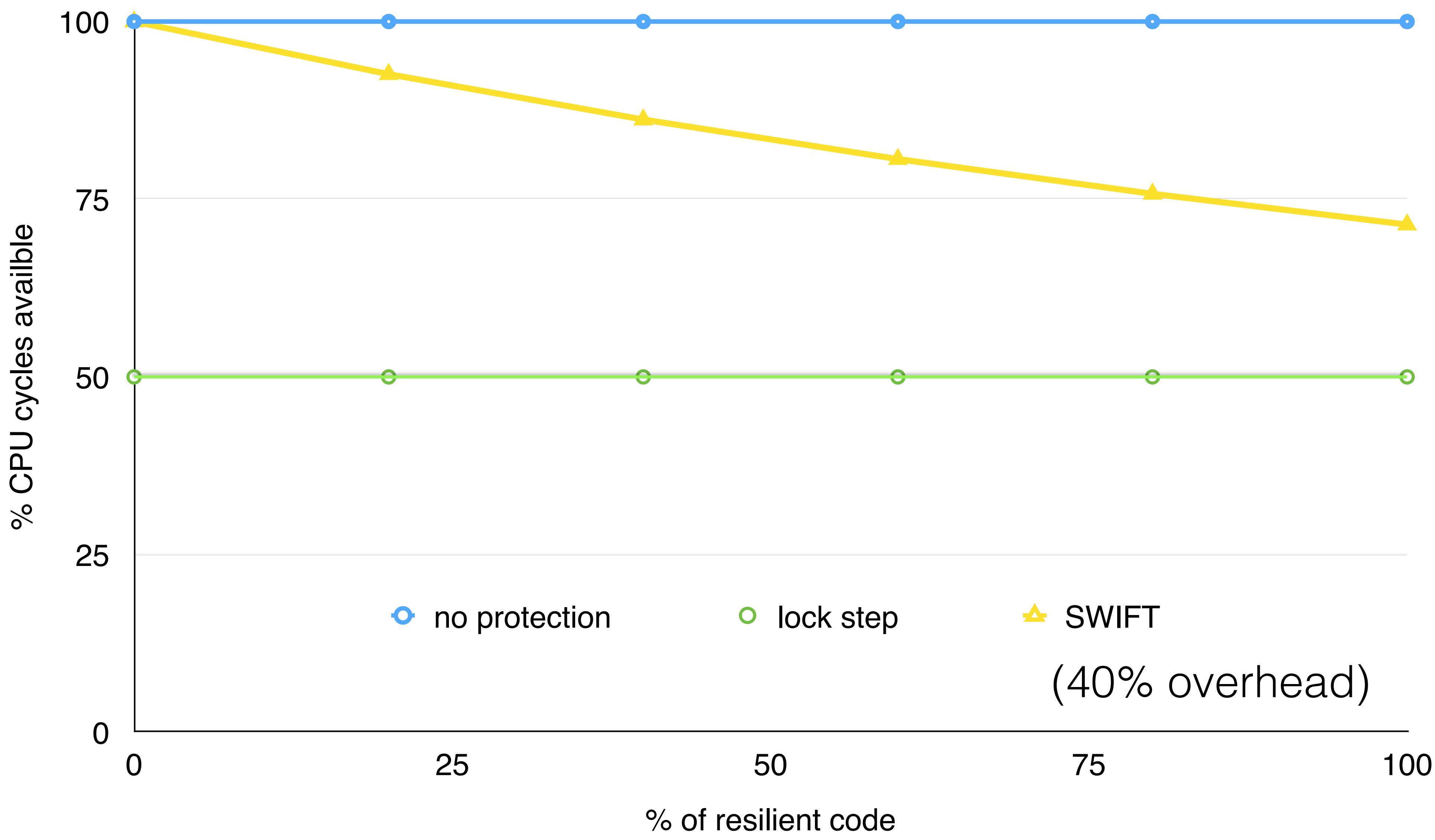
Yalcin, G., Unsal, O., Cristal, A.: FaulTM: Fault-Tolerance Using Hardware Transactional Memory. In: Design, Automation and Test in Europe DATE (2012)

Software Implemented Fault Tolerance (SWIFT)

- **Approach**:
  - use **time redundancy** instead of space redundancy
  - execute each instruction twice in sequence and check that results are the same
- **Notes**:
  - tries to exploit instruction level parallelism of modern CPUs
  - **Example**: Haswell has 8x parallel execution ports

[Reis, G.A. et al, „SWIFT: software implemented fault tolerance“, CGO2005]

SWIFT Overhead



Swift Issues

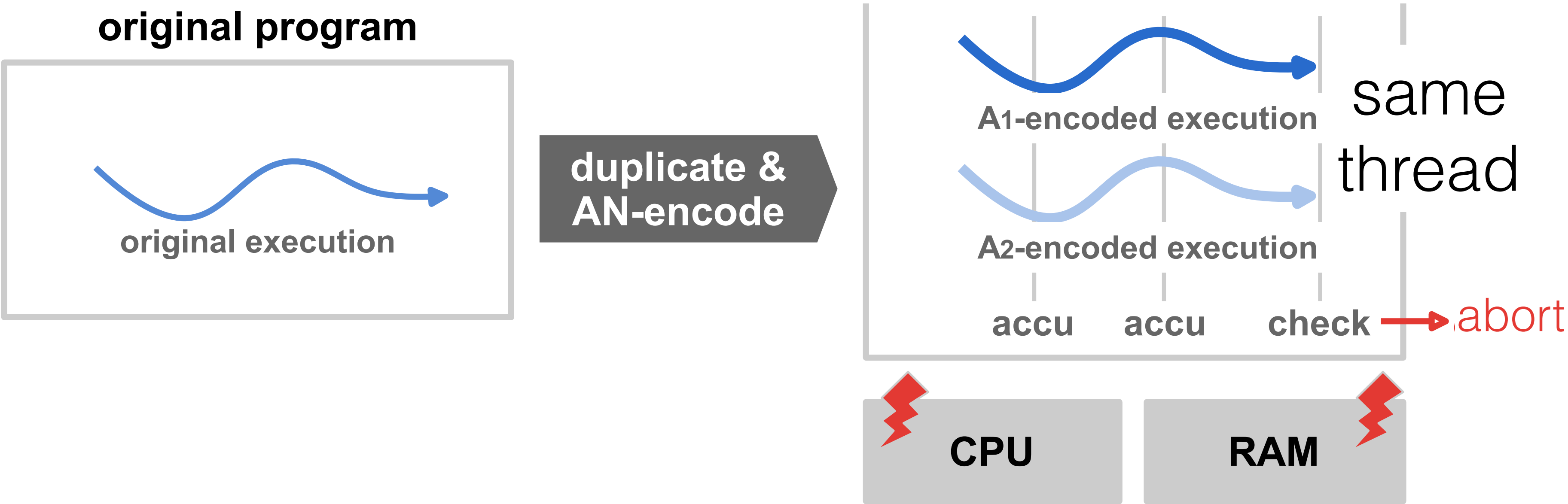
- **Large window of vulnerability!**
  - time of check to time of use issues
  - Haswell: 192-entry reorder window
- **Persistent faults**!
  - time redundancy: false negatives possible
- **Memory really a solved problem?**
  - what about memory overwrites?

AN Encoding

- **AN code**: all variables represented as multiples of some A
- **Advantages**:
  - deals with non-determinism (like SWIFT)
  - but no problems with out of order processing
- **Disadvantages**:
  - overhead
  - recovery from intermittent/persistent failures

Wamhoff, J.T.,et al: Transactional encoding for tolerating transient hardware errors. SSS ’13.

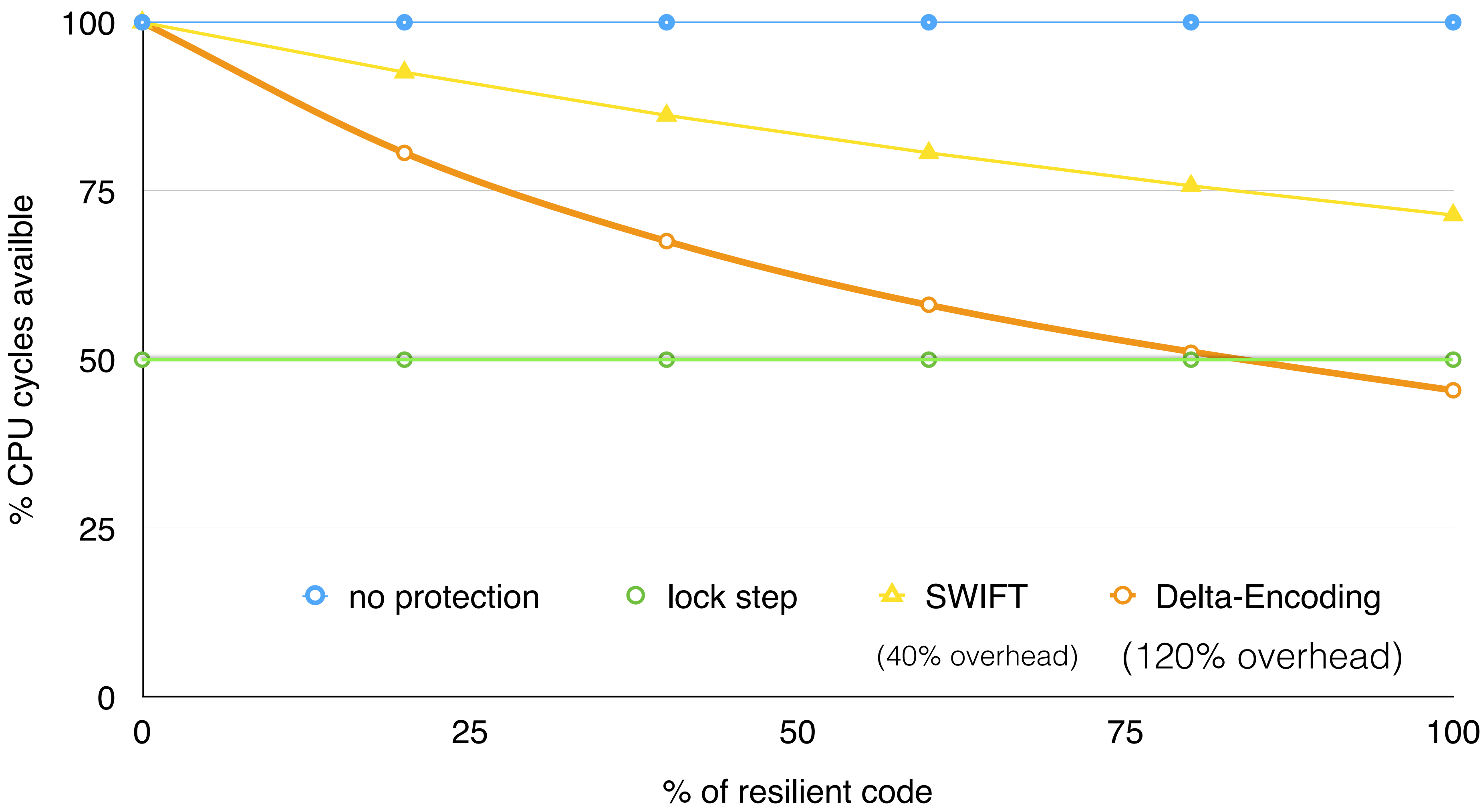
Delta Encoding



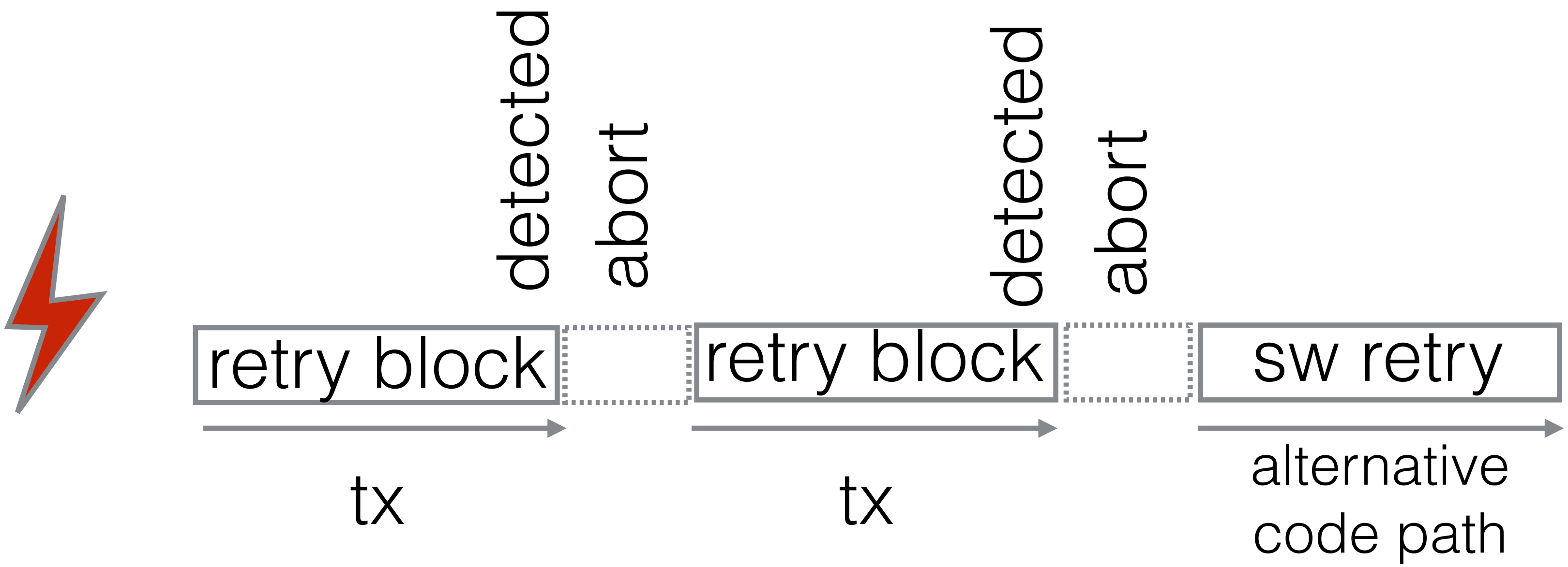
- Combination of AN-encoding and duplicate execution (similar to SWIFT)
- High detection coverage for transient, intermittent and permanent faults (99.997%).
- Acceptable slow down: 1xx %overhead

(currently under submission)

Delta-Encoding



Masking Hardware Faults



- **Idea**: (not yet implemented)
  - handle transient faults via retry
  - try to mask „persistent disagreements“ via SW retry

Questions

- **Deterministic SW bugs:**
  - can they be handled in a similar way as persistent hardware bugs?
- **Deterministic SW bugs**:
  - how to detect these?

Observation 1'

- **Objective: Horizontal Scalability**
  - any remote call can fail!
- **Objective: Robustness**
  - e.g., wrong arguments should never result in a panic!
  - use **graceful degradation** (i.e., drop request) and let client know
- **Problem**:
  - most cases we will not be able to wrap request in a transaction.

Summary

- **Approach**:
  - masking transient failures using transactional memory
    - we do not need conflict detection from TM
  - depending on expected detection and masking coverage
    - will use different detection mechanisms
  - deal with „persistent disagreements“ using a SW retry
- **A few open problems left..**

References

- Fetzer, C., Felber, P.: Transactional memory for dependable embedded systems. In: 7th Workshop on Hot Topics in System Dependability (HotDep). pp. 223–227. IEEE (2011)

- Riegel, T., Felber, P., Fetzer, C.: Composable error recovery with transactional memory. Bulletin of the European Association for Theoretical Computer Science (BEATCS) 99, (2009)

- Wamhoff, J.T., Schwalbe, M., Faqeh, R., Fetzer, C., Felber, P.: Transactional encoding for tolerating transient hardware errors. In: Higashino, T., Katayama, Y., Masuzawa, T., Potop-Butucaru, M., Yamashita, M. (eds.) Stabilization, Safety, and Security of Distributed Systems: 15th International Symposium, SSS '13, vol. 8255, pp. 1–16. Springer International Publishing (November 2013)

- Yalcin, G., Unsal, O., Cristal, A.: FaulTM: Fault-Tolerance Using Hardware Transactional Memory. In: Design, Automation and Test in Europe DATE (2012)

- Yalcin, G., Unsal, O., Cristal, A.: Fault Tolerance for Multi-Threaded Applications by Leveraging Hardware Transactional Memory. In: International Conference on Comput- ing Frontiers (2013)

- Yalcin, G., Unsal, O., Cristal, A., Hur, I., Valero, M.: FaulTM: Fault-Tolerance Using Hardware Transactional Memory. In: Workshop on Parallel Execution of Sequential Pro- grams on Multi-Core Architecture PESPMA (2010)

- Gulay Yalcin and Osman S. Unsal, Transactional Memory for Reliability, .

Pictures

- https://philosophadam.wordpress.com/2014/10/01/finding-the-calm-within-the-storm-shame-resilience-in-practice/
- http://trekcore.com/specials/rare/Massive_Kirk.jpg