HAFT

Hardware-Assisted Fault Tolerance

Dmitrii Kuvaiskii
Rasha Faqeh
Pramod Bhatotia
Christof Fetzer
Technische Universität Dresden

Pascal Felber
Université de Neuchâtel

Hardware Errors in the Wild

- Online services run in **huge data centers**

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception

Google

amazon

Microsoft

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
- Focus on faults that result in **arbitrary data corruptions**

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
- Focus on faults that result in **arbitrary data corruptions**

Amazon S3 Availability Event
http://status.aws.amazon.com/s3-20080720.html

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
- Focus on faults that result in **arbitrary data corruptions**

Amazon S3 Availability Event
http://status.aws.amazon.com/s3-20080720.html

Data corruption with Opteron CPUs
https://bugzilla.kernel.org/show_bug.cgi?id=7768

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
- Focus on faults that result in **arbitrary data corruptions**

Amazon S3 Availability Event
http://status.aws.amazon.com/s3-20080720.html

Data corruption with Opteron CPUs
https://bugzilla.kernel.org/show_bug.cgi?id=7768

Defective S3 load balancer
https://forums.aws.amazon.com/thread.jspa?threadID=22709

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
- Focus on faults that result in **arbitrary data corruptions**

Amazon S3 Availability Event
http://status.aws.amazon.com/s3-20080720.html

Data corruption with Opteron CPUs
https://bugzilla.kernel.org/show_bug.cgi?id=7768

Defective S3 load balancer
https://forums.aws.amazon.com/thread.jspa?threadID=22709

Google's Mesa Data Warehousing System

„ …corruption can occur **transiently in CPU** or RAM. Guarding against such corruptions is an **important goal** in Mesa's overall design… "

Protecting Against Data Corruptions

Principled approaches ⟷ Ad-hoc approaches

Protecting Against Data Corruptions

Principled approaches ⟷ Ad-hoc approaches

Byzantine Fault Tolerance

Protecting Against Data Corruptions

Principled approaches ⟷ Ad-hoc approaches

Byzantine Fault Tolerance

- ✔ Tolerates arbitrary faults
- ✘ Pessimistic fault model
- ✘ High resource overheads

Protecting Against Data Corruptions

Principled approaches ⟷ Ad-hoc approaches

Byzantine Fault Tolerance

- ✔ Tolerates arbitrary faults
- ✘ Pessimistic fault model
- ✘ High resource overheads

Checksums / Assertions

Protecting Against Data Corruptions

Principled approaches ⟷ Ad-hoc approaches

Byzantine Fault Tolerance	**Checksums / Assertions**
✔ Tolerates arbitrary faults	✔ Low performance overheads
✘ Pessimistic fault model	✘ Only anticipated faults
✘ High resource overheads	✘ Manual and error-prone

Protecting Against Data Corruptions

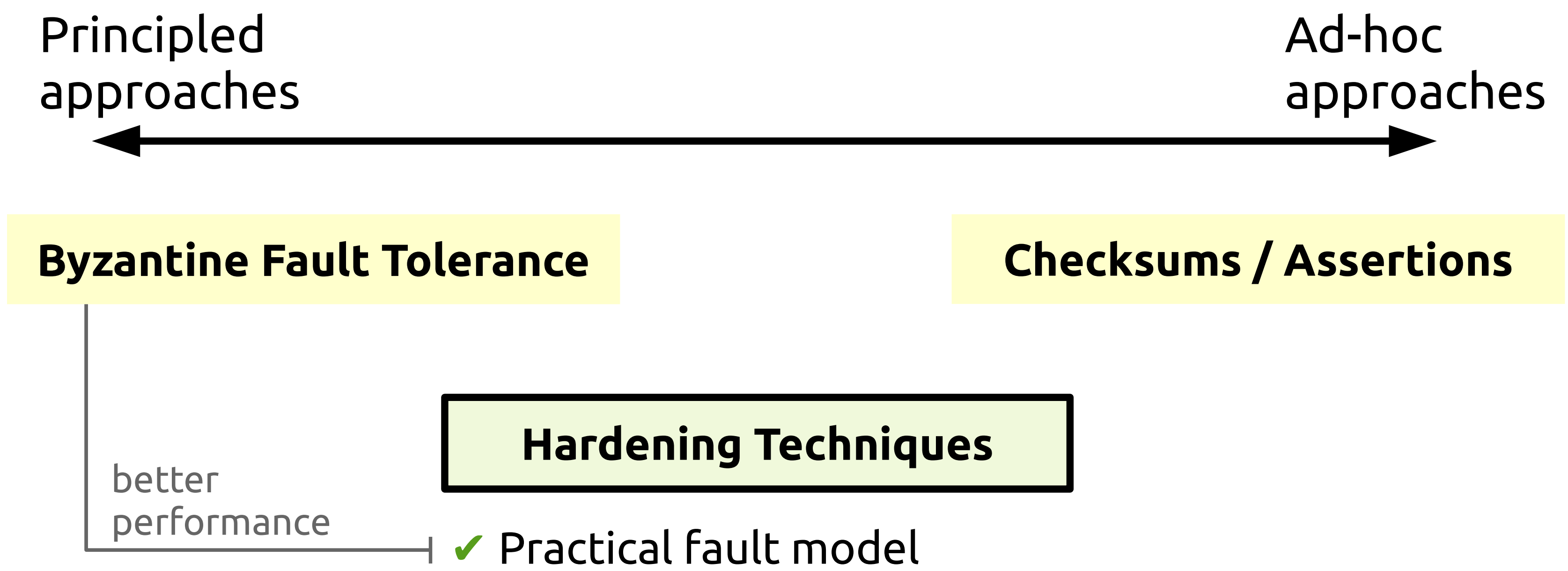
Principled approaches ⟷ Ad-hoc approaches

Byzantine Fault Tolerance

Checksums / Assertions

Hardening Techniques

Protecting Against Data Corruptions



Protecting Against Data Corruptions

Principled approaches ⟷ Ad-hoc approaches

Byzantine Fault Tolerance

Checksums / Assertions

Hardening Techniques

- ✔ Practical fault model (better performance)
- ✔ Disciplined protection (better fault coverage)

Hardening

Hardening

Limitations:

Hardening

Limitations:

✘ Non-transparent
- Manual changes in source code
- Specific languages / programming models

Hardening

Limitations:

- ✘ Non-transparent
  - Manual changes in source code
  - Specific languages / programming models
- ✘ Impractical
  - Only single-threaded programs
  - Only fail-stop execution

Hardening

Limitations:

- ✘ Non-transparent
  - Manual changes in source code
  - Specific languages / programming models
- ✘ Impractical
  - Only single-threaded programs
  - Only fail-stop execution
- ✘ Inefficient
  - Requires spare cores / deterministic execution
  - Memory overhead

Design Goals for HAFT

- ✔ Transparent
- ✔ Practical
- ✔ Efficient

Design Goals for HAFT

- ✔ Transparent
  - No changes in source code
  - Shared-memory programming model
- ✔ Practical
- ✔ Efficient

Design Goals for HAFT

✔ Transparent

- No changes in source code
- Shared-memory programming model

✔ Practical

- Multithreaded programs
- Fault detection *and* fault recovery

✔ Efficient

Design Goals for HAFT

- ✔ Transparent
  - No changes in source code
  - Shared-memory programming model
- ✔ Practical
  - Multithreaded programs
  - Fault detection *and* fault recovery
- ✔ Efficient
  - No spare cores, no deterministic execution
  - No memory overhead (rely on ECC)

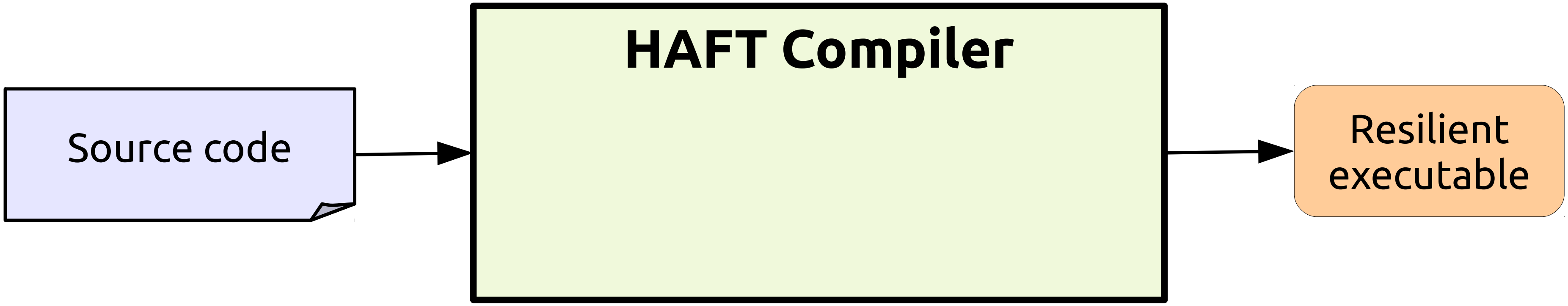
– ~~Motivation~~

– **Design**

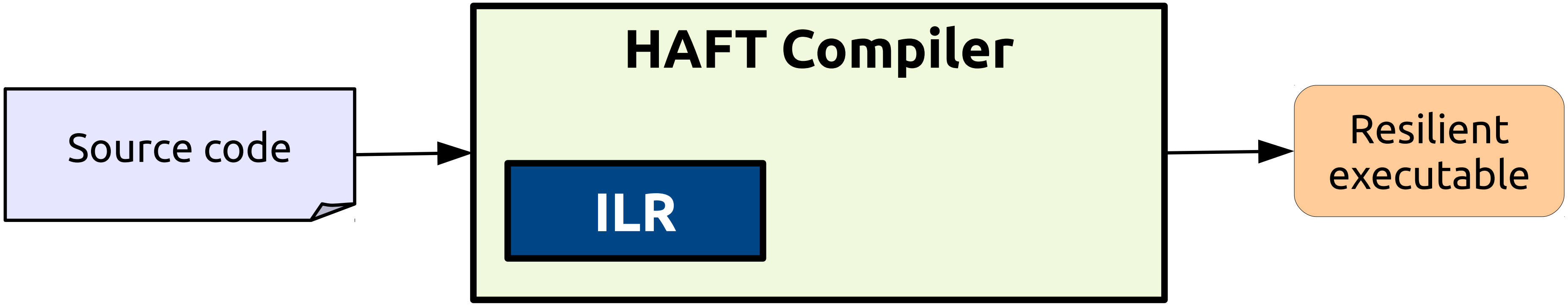
– Optimizations

– Evaluation

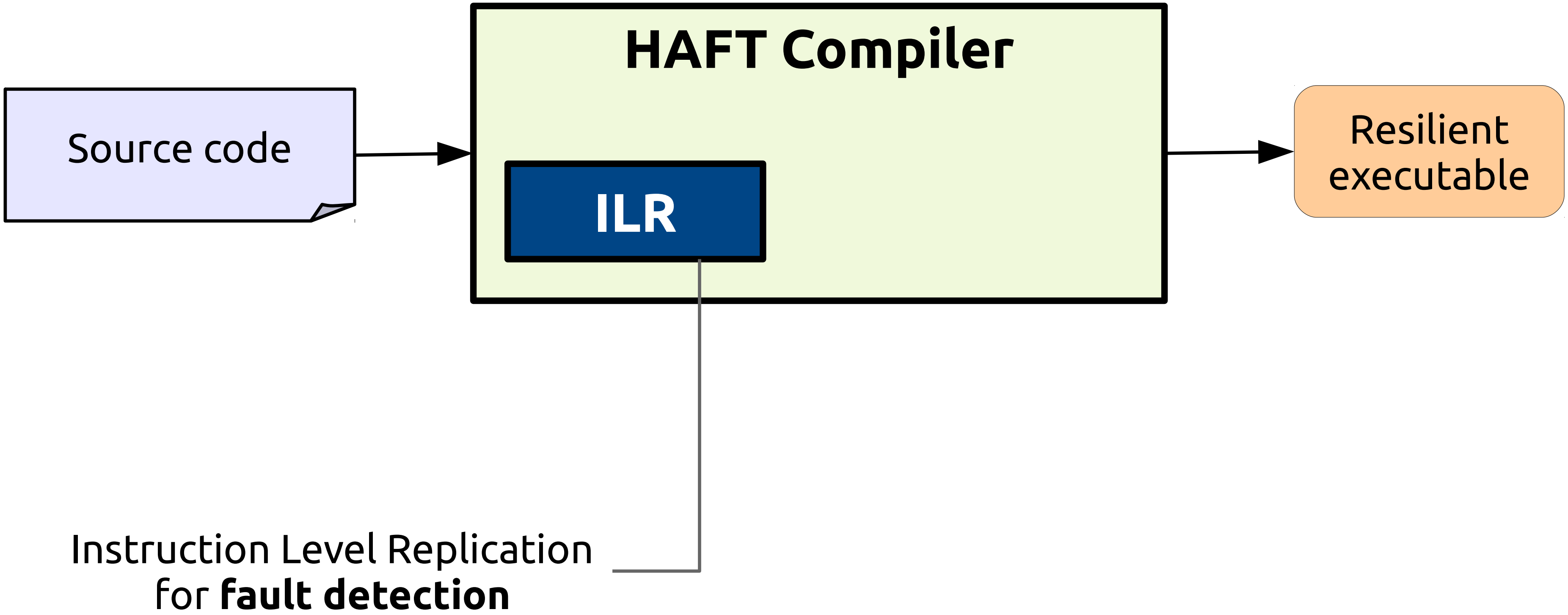
HAFT



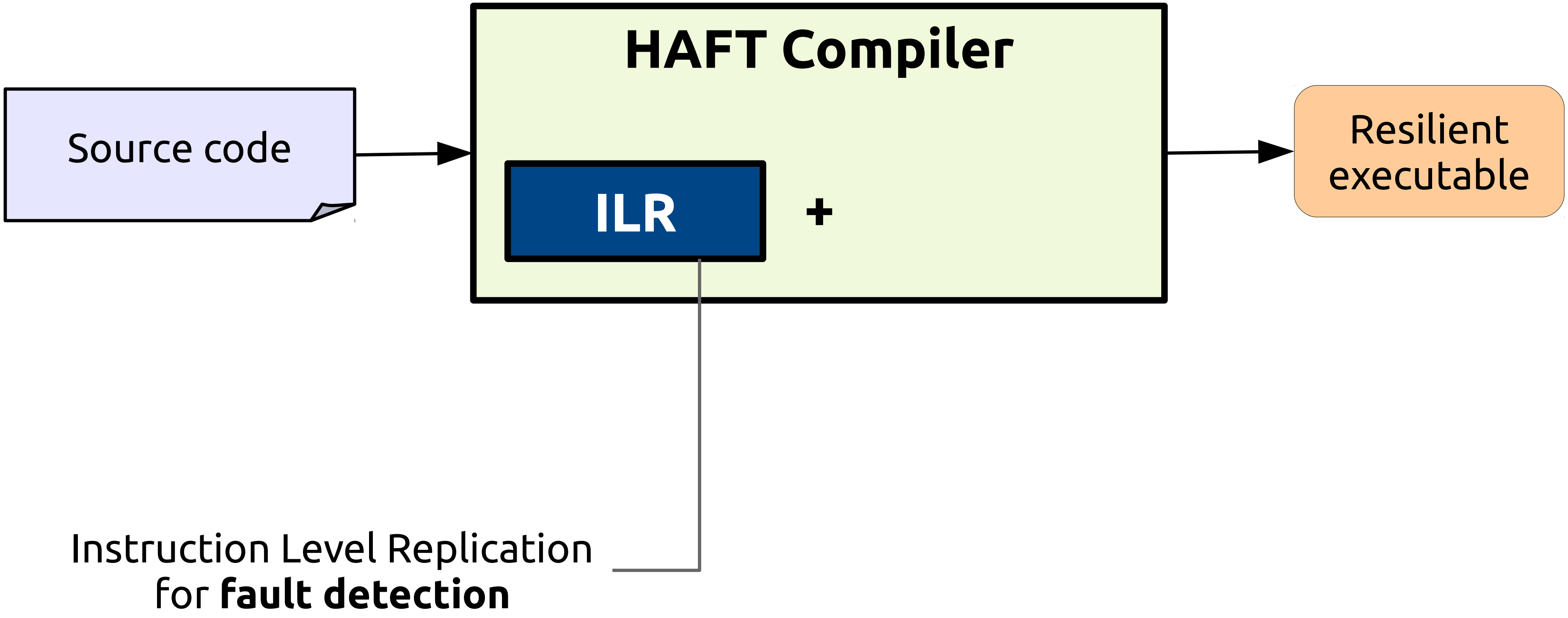
HAFT



HAFT

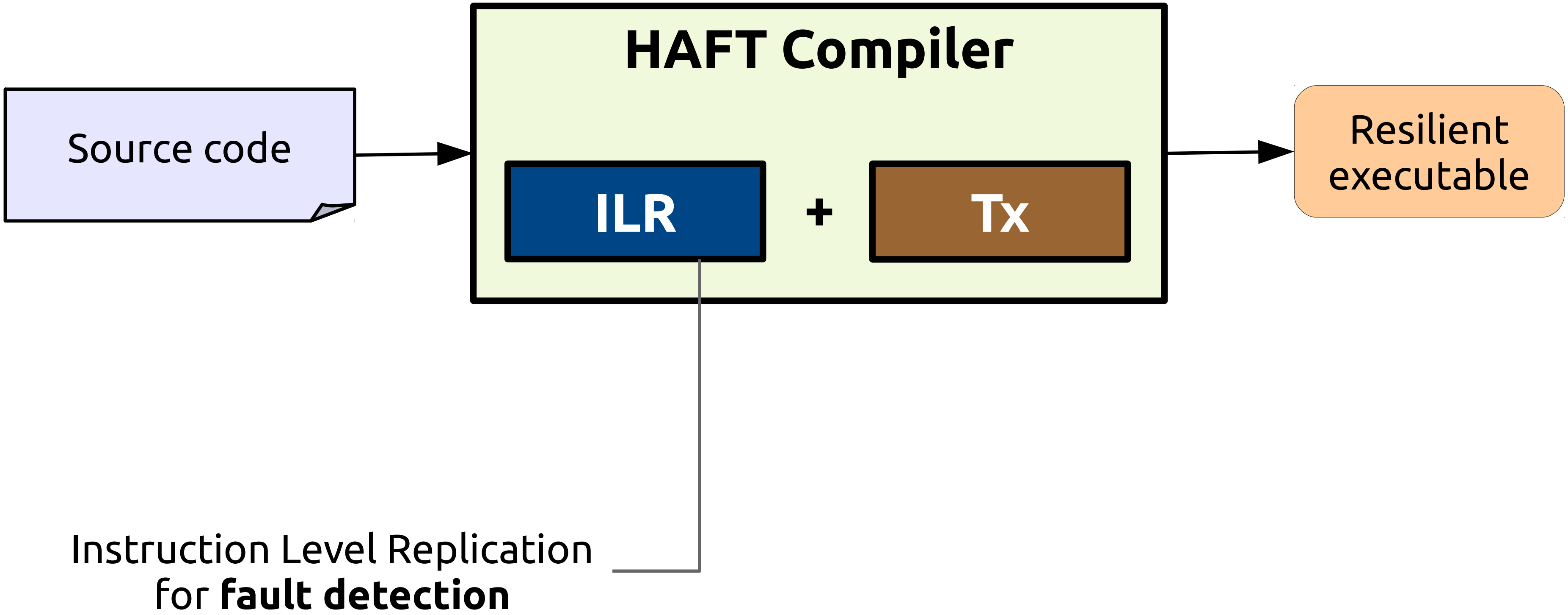

Source code
HAFT Compiler
ILR
Resilient executable
Instruction Level Replication for **fault detection**

HAFT

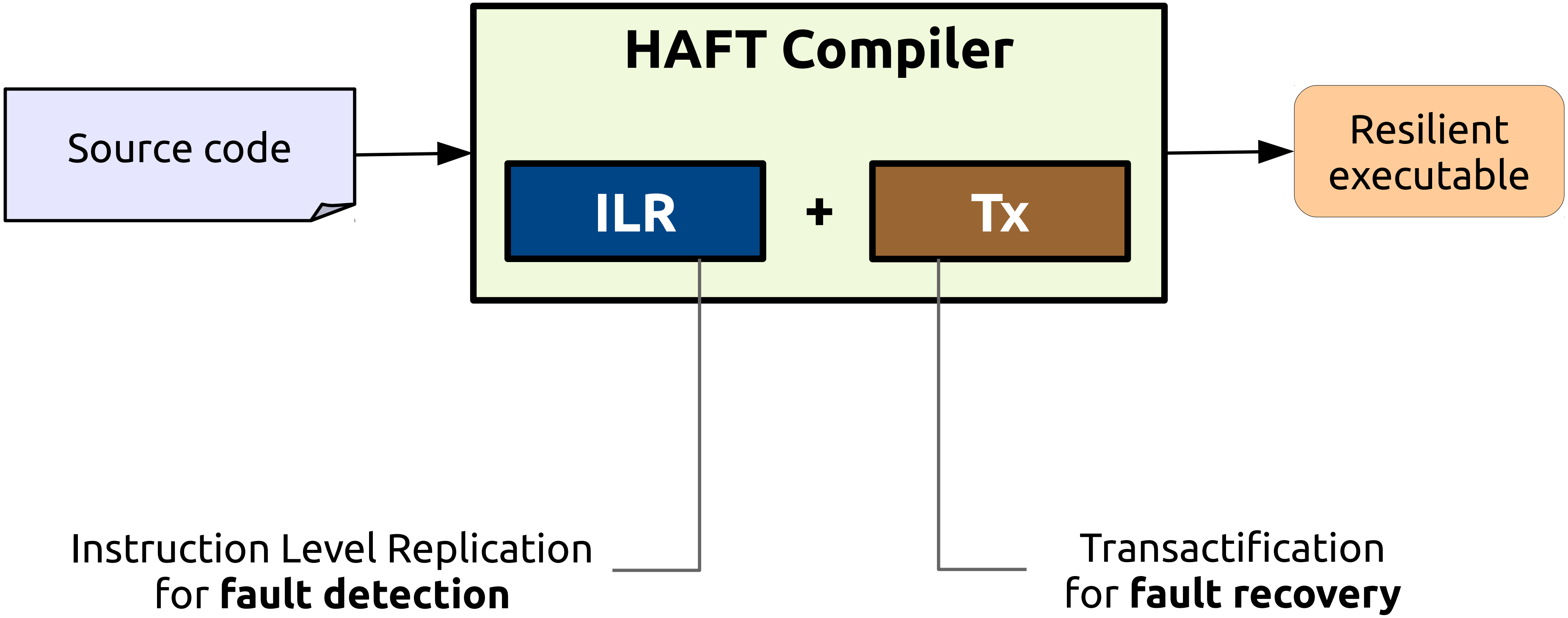


Source code
HAFT Compiler
ILR
+
Resilient executable
Instruction Level Replication for **fault detection**

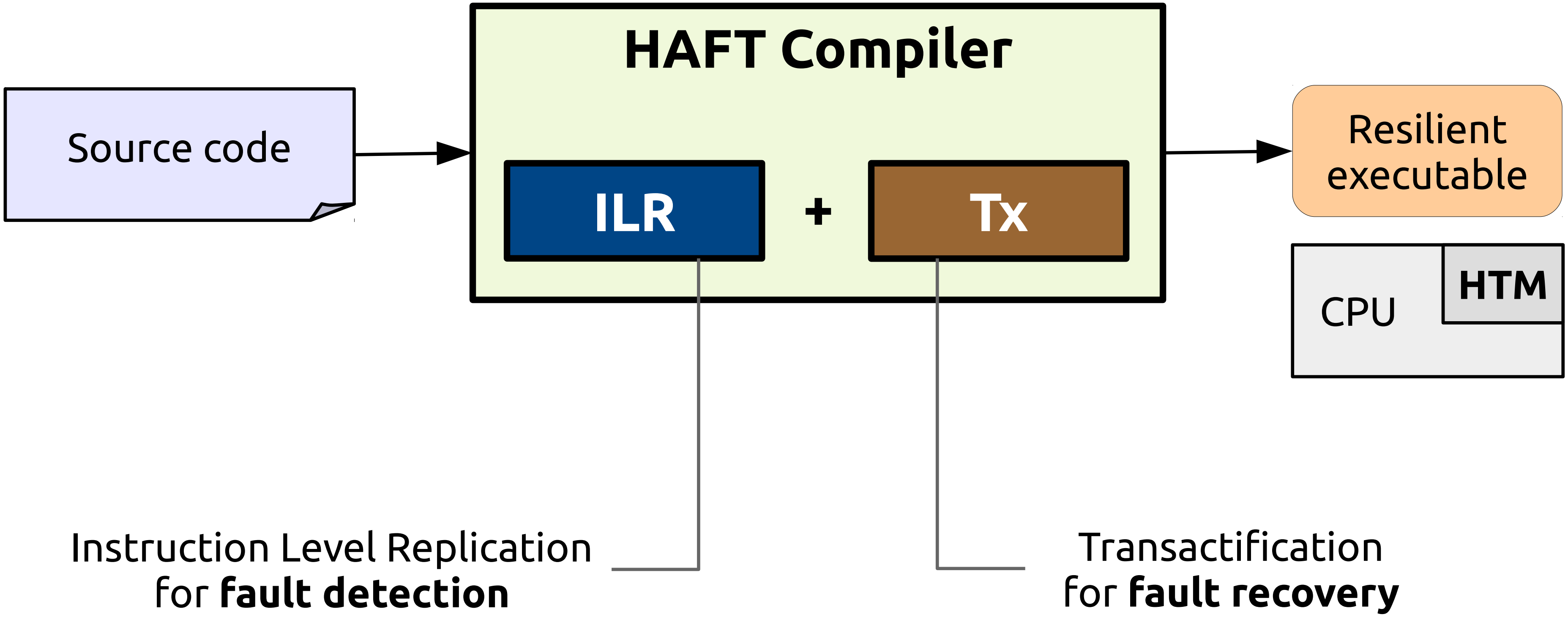
HAFT


HAFT Compiler
Source code
ILR
+
Tx
Resilient executable
Instruction Level Replication
for **fault detection**

HAFT


Source code
HAFT Compiler
ILR
+
Tx
Resilient executable
Instruction Level Replication for fault detection
Transactification for fault recovery

HAFT

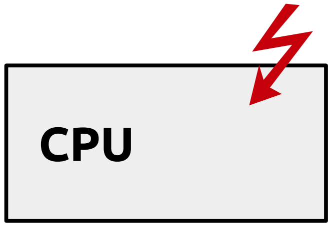


HAFT Compiler
Source code
ILR
+
Tx
Resilient executable
CPU
HTM
Instruction Level Replication for **fault detection**
Transactification for **fault recovery**

Fault Model

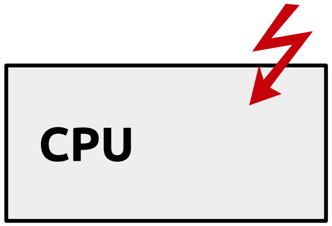
Fault Model

– Protect against **transient faults in CPU**
  - corruptions in CPU registers
  - miscomputations in CPU execution units

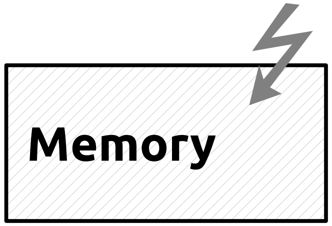


Fault Model

– Protect against **transient faults in CPU**
  - corruptions in CPU registers
  - miscomputations in CPU execution units



– Memory is **protected by other means**
  - DRAM protected by ECC
  - CPU caches protected by ECC and parity



HAFT: Code Transformation

Native

```
z   = add x, y

store z, 0x10
```

HAFT: Code Transformation

Native	Instr Level Replication **ILR**
`z = add x, y`	`z = add x, y` `z2 = add x2, y2`
`store z, 0x10`	`store z, 0x10`

HAFT: Code Transformation

Native	Instr Level Replication **ILR**

`z  = add x, y`	`z  = add x,  y`
	`z2 = add x2, y2`
	`d  = cmp neq z, z2`
	`br d, crash`

`store z, 0x10`	`store z, 0x10`

HAFT: Code Transformation

Native

```
z  = add x, y



store z, 0x10
```

Instr Level Replication
ILR

```
z  = add x,  y
z2 = add x2, y2
d  = cmp neq z, z2
br d, crash
store z, 0x10
```

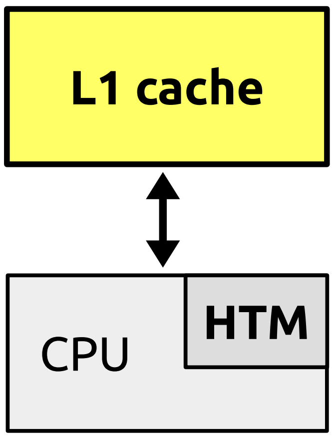
Transactification
Tx

```
tx-begin
z  = add x,  y
z2 = add x2, y2
d  = cmp neq z, z2
br d, tx-abort
store z, 0x10
tx-end
```

Challenge of Transactification

Commodity HTM (Intel TSX) implementations

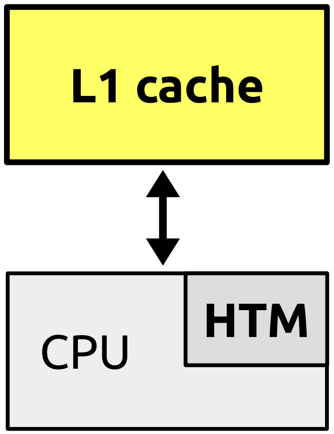
- for synchronization **not** fault recovery
- for small-sized well-behaved transactions



Challenge of Transactification

Commodity HTM (Intel TSX) implementations

- for synchronization **not** fault recovery
- for small-sized well-behaved transactions



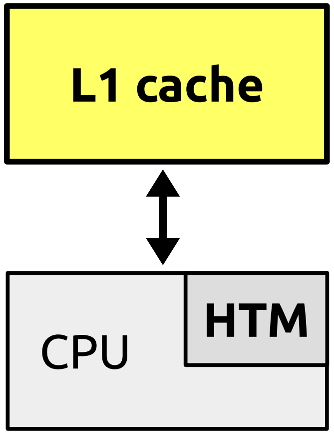
Need **right size** of HW transactions

- large and rare → high abort rate
- small and frequent → high perf overhead

Challenge of Transactification

Commodity HTM (Intel TSX) implementations

- for synchronization **not** fault recovery
- for small-sized well-behaved transactions



Need **right size** of HW transactions

- large and rare → high abort rate
- small and frequent → high perf overhead

Solution: **dynamic transaction boundaries**

- track number of instructions executed
- start new transaction whenever exceed predefined threshold

- ~~Motivation~~
- ~~Design~~
- **Optimizations**
- Evaluation

– ~~Motivation~~

– ~~Design~~

– **Optimizations**

1. Shared memory accesses
2. Lock elision

– Evaluation

- ~~Motivation~~
- ~~Design~~
- **Optimizations**
    1. Shared memory accesses
    2. Lock elision
- Evaluation

See the paper for other optimizations

Optimization 1: Shared Memory Accesses

Optimization 1: Shared Memory Accesses

- **Motivation** Checks on memory accesses are expensive
  - **40%** instructions are loads and stores

Optimization 1: Shared Memory Accesses

- **Motivation** Checks on memory accesses are expensive
  - **40%** instructions are loads and stores

```
d   = cmp neq adr, adr2
br d, xabort
val = load adr
```

Optimization 1: Shared Memory Accesses

- **Motivation** Checks on memory accesses are expensive
  - **40%** instructions are loads and stores
- **Idea** Distinguish atomic and non-atomic accesses

```
d   = cmp neq adr, adr2
br d, xabort
val = load adr
```

Optimization 1: Shared Memory Accesses

- **Motivation** Checks on memory accesses are expensive
  - **40%** instructions are loads and stores
- **Idea** Distinguish atomic and non-atomic accesses

Explicit atomic

```
d   = cmp neq adr, adr2
br d, xabort
val = load atomic adr
```

Protected non-atomic

```
val  = load adr
val2 = load adr2
```

Optimization 1: Shared Memory Accesses

- **Motivation** Checks on memory accesses are expensive
  - **40%** instructions are loads and stores
- **Idea** Distinguish atomic and non-atomic accesses
- **Impact** Up to **40%** better performance

Explicit atomic

```
d   = cmp neq adr, adr2
br d, xabort
val = load atomic adr
```

Protected non-atomic

```
val  = load adr
val2 = load adr2
```

Optimization 2: Lock Elision

Optimization 2: Lock Elision

- **Motivation** Small critical sections are expensive
  - **3** transactions for each

Tiny critical section

```
...
lock(&l)
counter++
unlock(&l)
...
```

Optimization 2: Lock Elision

- **Motivation** Small critical sections are expensive
  - **3** transactions for each

Tiny critical section

```
xend      ...
xbegin    lock(&l)
xend      counter++
xbegin    unlock(&l)
          ...
```

Optimization 2: Lock Elision

- **Motivation** Small critical sections are expensive
  - **3** transactions for each
- **Idea** Use Tx for recovery *and* lock elision

Tiny critical section

```
...
lock(&l)          xend
                  xbegin
counter++
                  xend
unlock(&l)        xbegin
...
```

Optimization 2: Lock Elision

- **Motivation** Small critical sections are expensive
  - **3** transactions for each
- **Idea** Use Tx for recovery *and* lock elision

Tiny critical section

```
...
lock_wrapper(&l)
counter++
unlock_wrapper(&l)
...
```

Optimization 2: Lock Elision

- **Motivation** Small critical sections are expensive
  - **3** transactions for each
- **Idea** Use Tx for recovery *and* lock elision
- **Impact** Up to **30%** better throughput

Tiny critical section

```
...
lock_wrapper(&l)
counter++
unlock_wrapper(&l)
...
```

– ~~Motivation~~

– ~~Design~~

– ~~Optimizations~~

– **Evaluation**

- ~~Motivation~~
- ~~Design~~
- ~~Optimizations~~
- **Evaluation**
  1. Performance overheads
  2. Reliability
  3. Real-world application: Memcached

- ~~Motivation~~
- ~~Design~~
- ~~Optimizations~~
- **Evaluation**
  1. Performance overheads
  2. Reliability
  3. Real-world application: Memcached

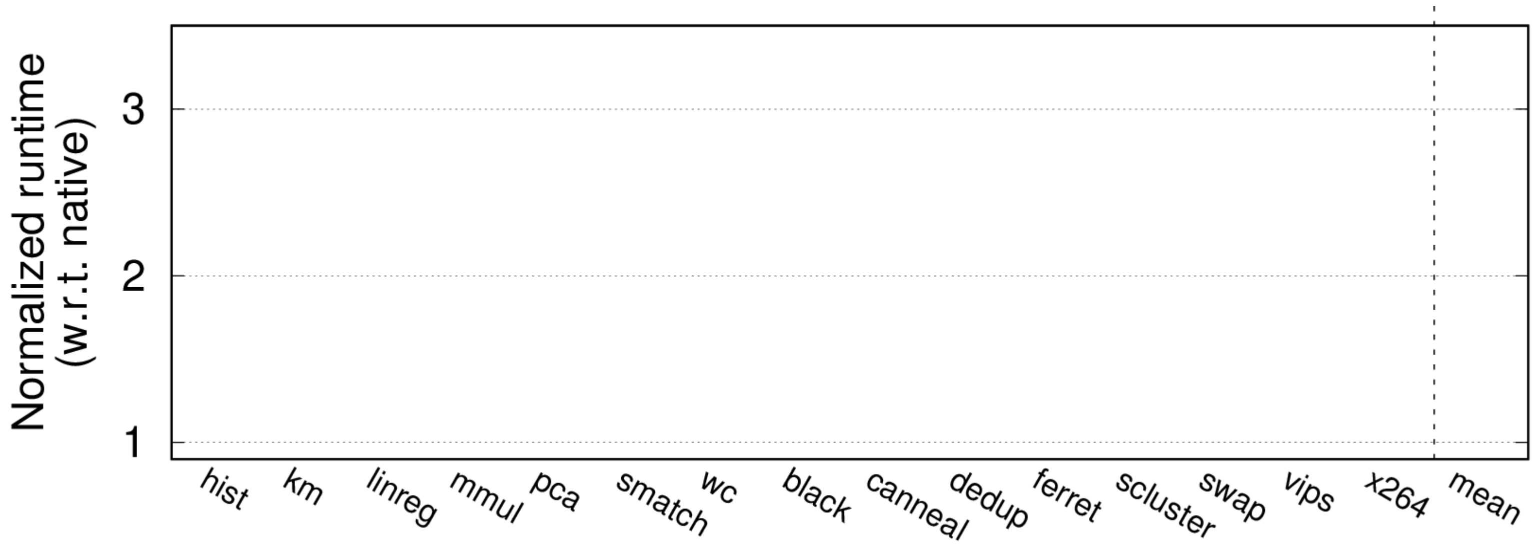
See the paper for more results

Performance Overheads

14-core Intel Haswell CPU

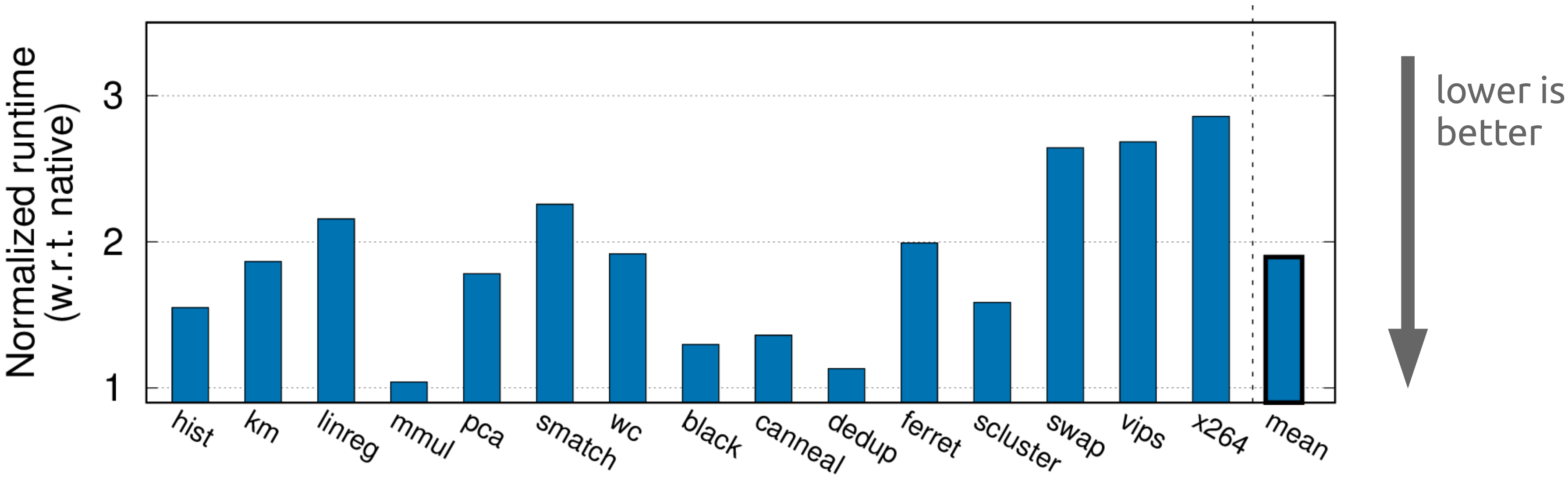
Performance Overheads

14-core Intel Haswell CPU



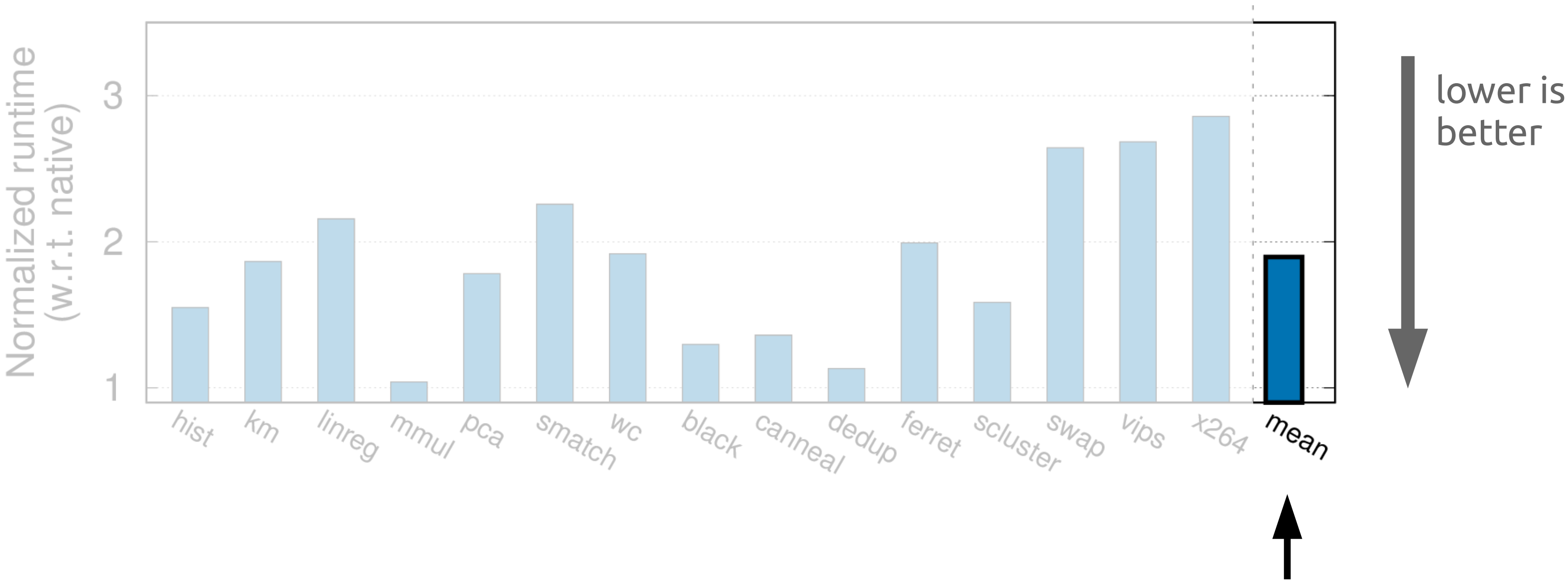
Performance Overheads

14-core Intel Haswell CPU



Performance Overheads

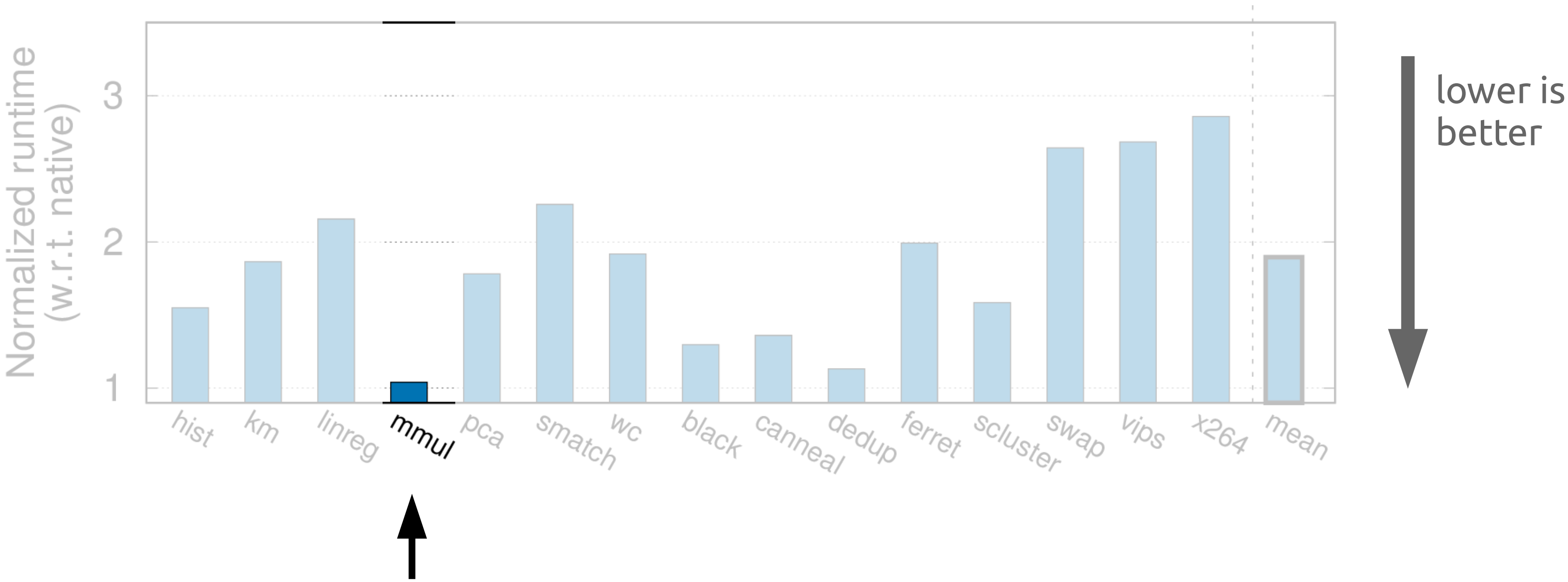
14-core Intel Haswell CPU



Average performance overhead is **2×**

Performance Overheads

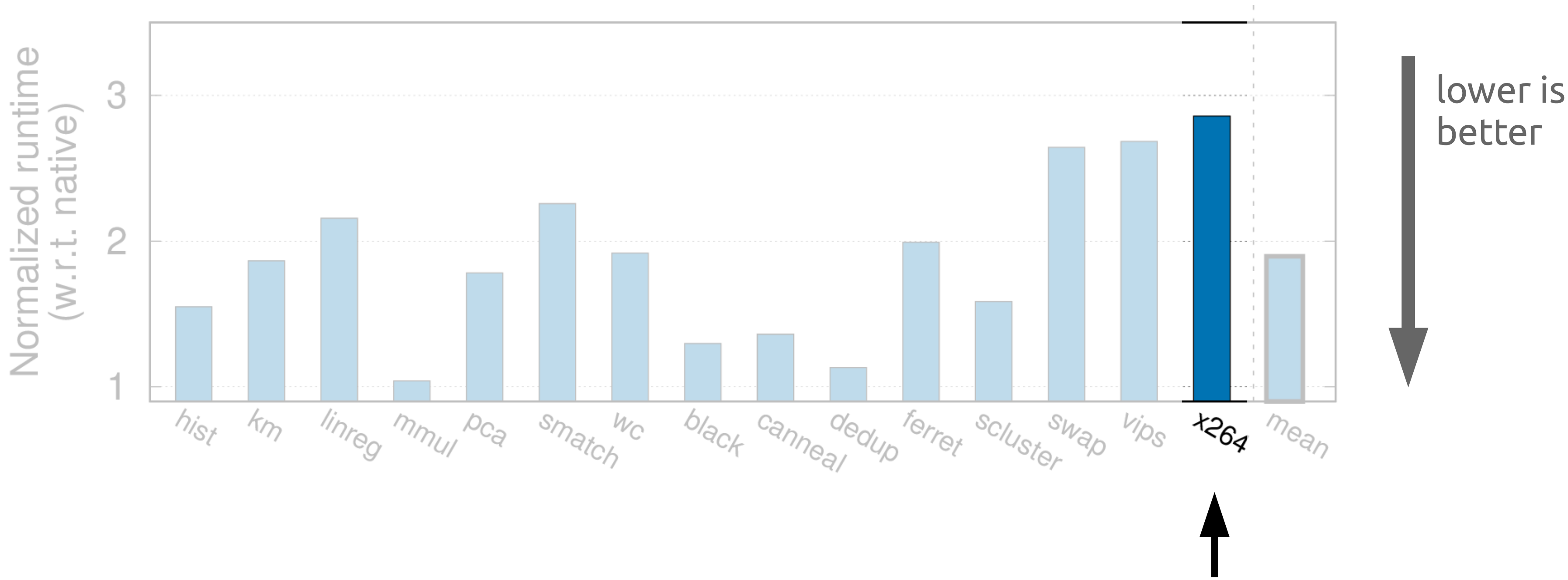
14-core Intel Haswell CPU



Amortized by very low cache locality

Performance Overheads

14-core Intel Haswell CPU



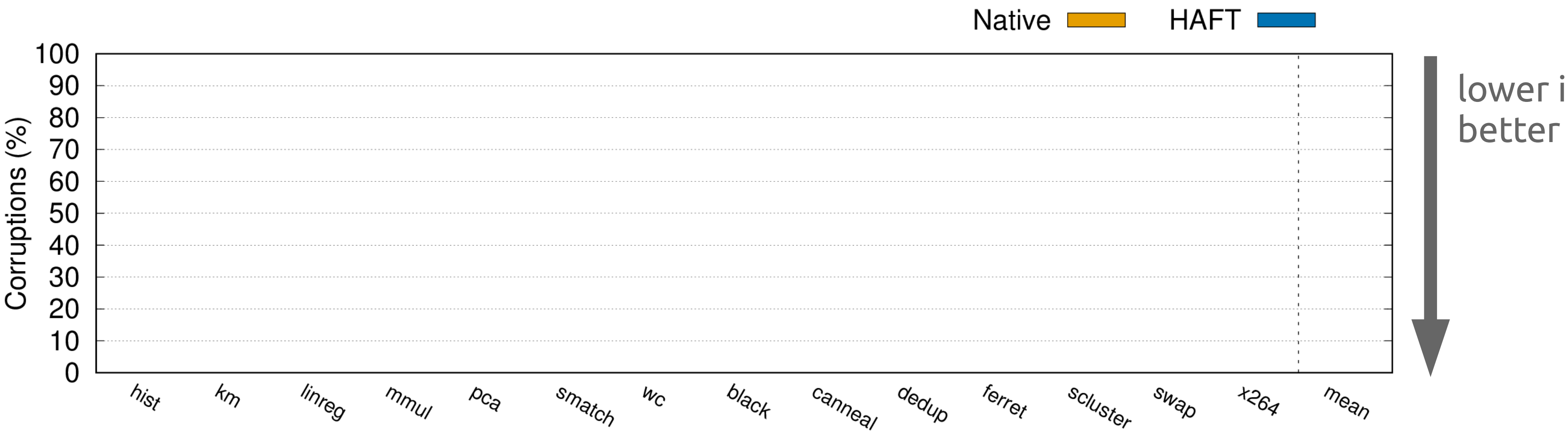
(1) small-sized transactions and (2) high original ILP

Data Corruptions

2500 fault injections per benchmark

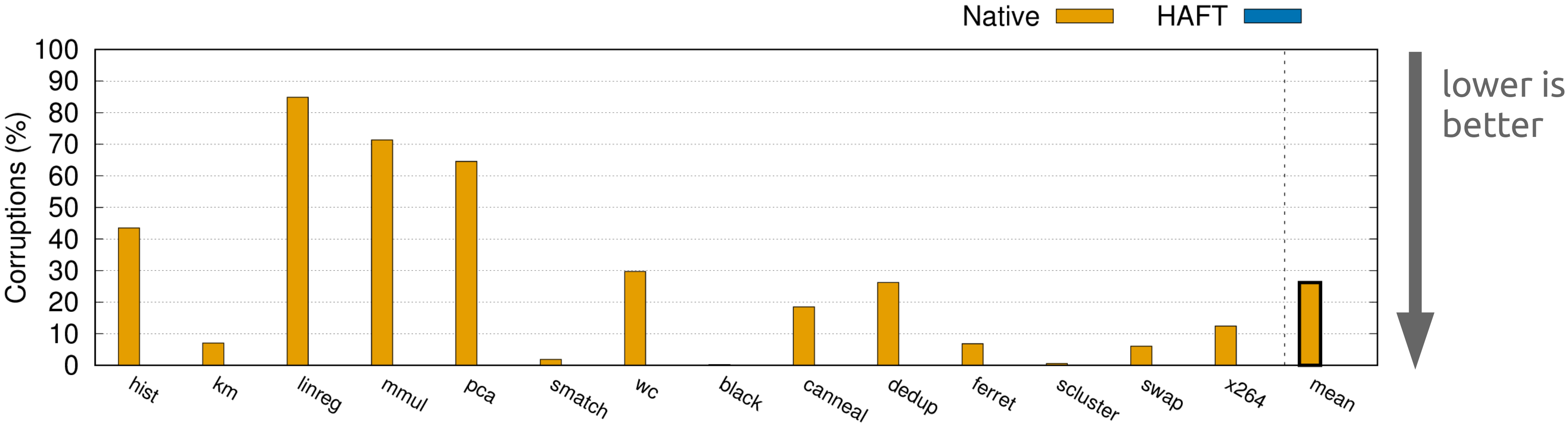
Data Corruptions

2500 fault injections per benchmark



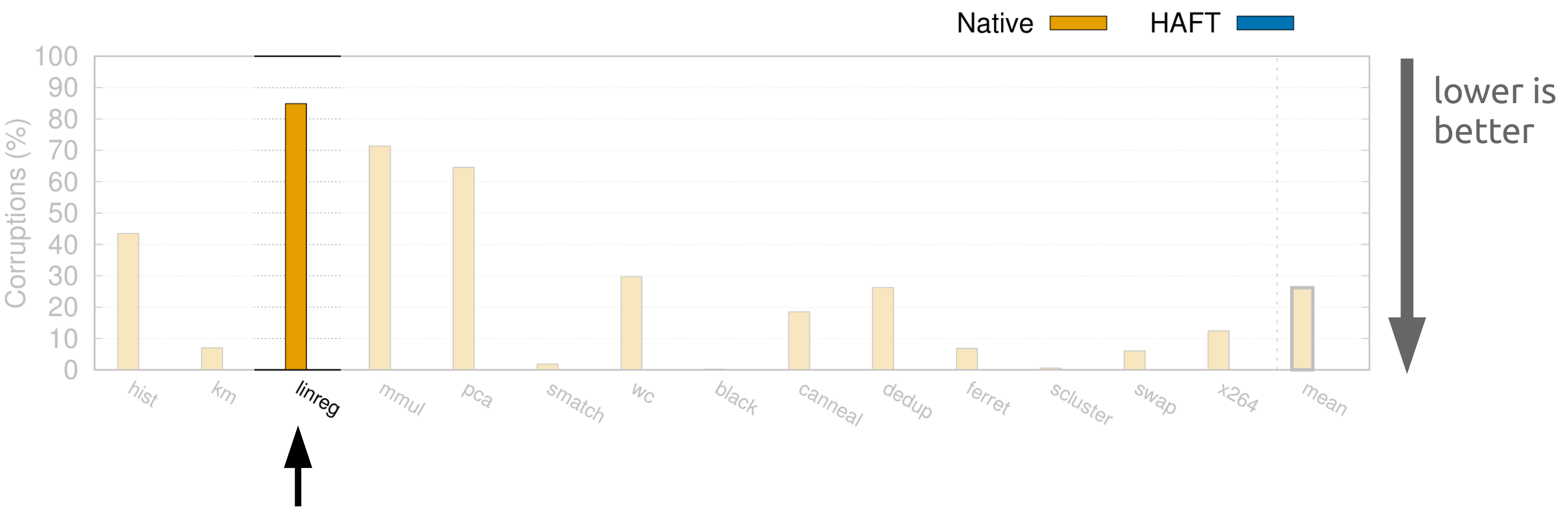
Data Corruptions

2500 fault injections per benchmark



Data Corruptions

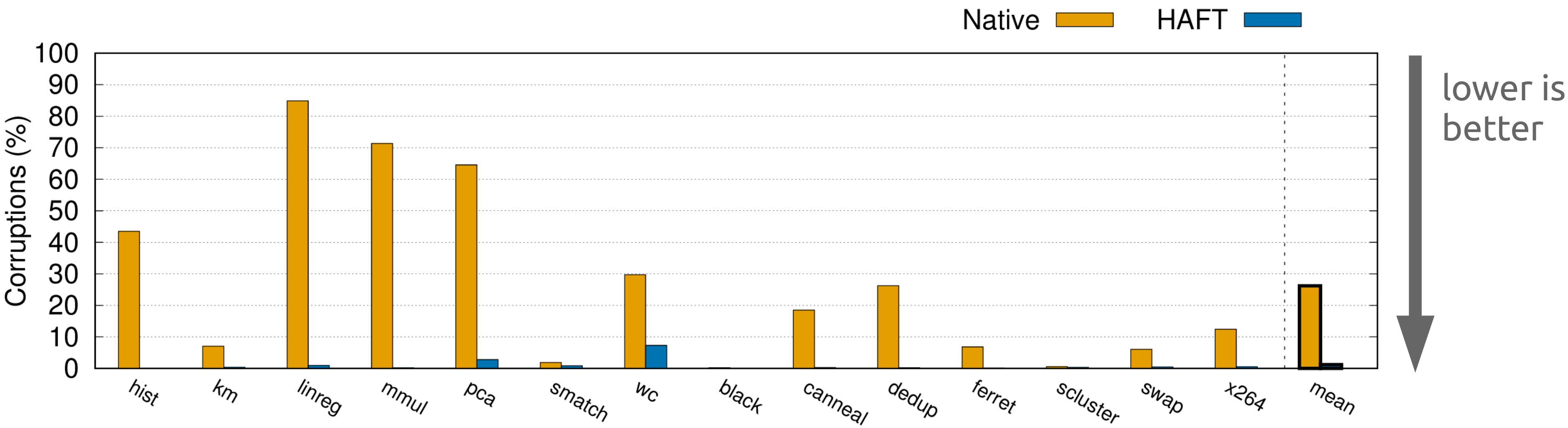
2500 fault injections per benchmark



Out of 2500 fault injections, **83%** led to data corruption in output

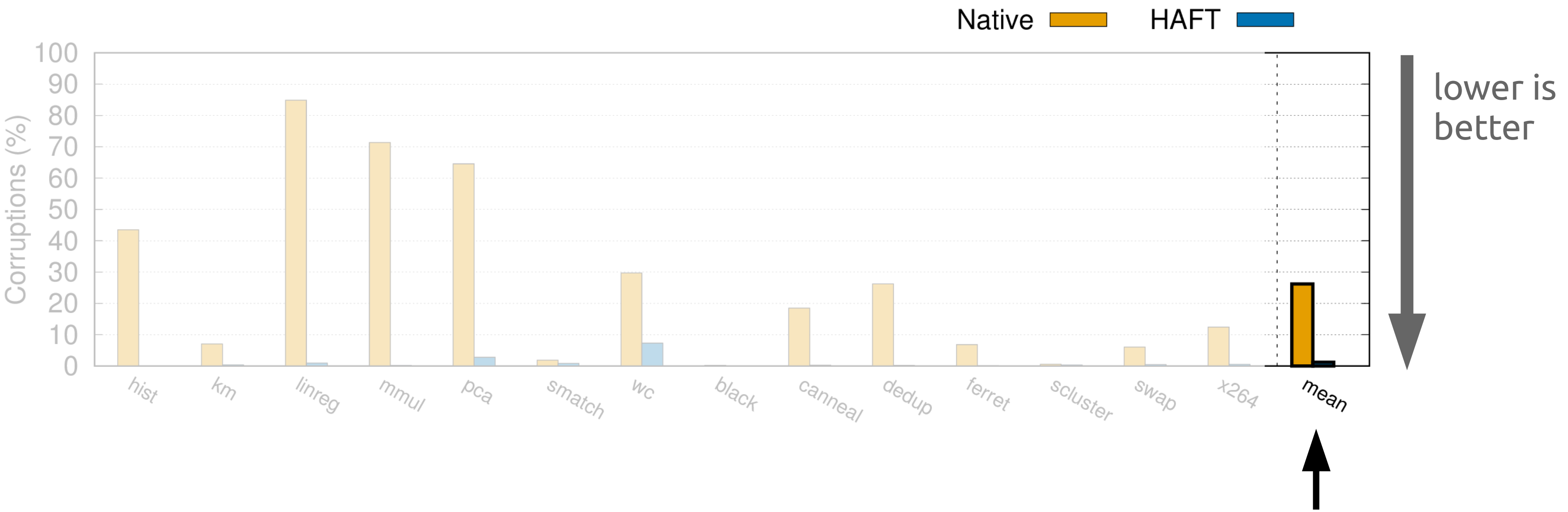
Data Corruptions

2500 fault injections per benchmark



Data Corruptions

2500 fault injections per benchmark



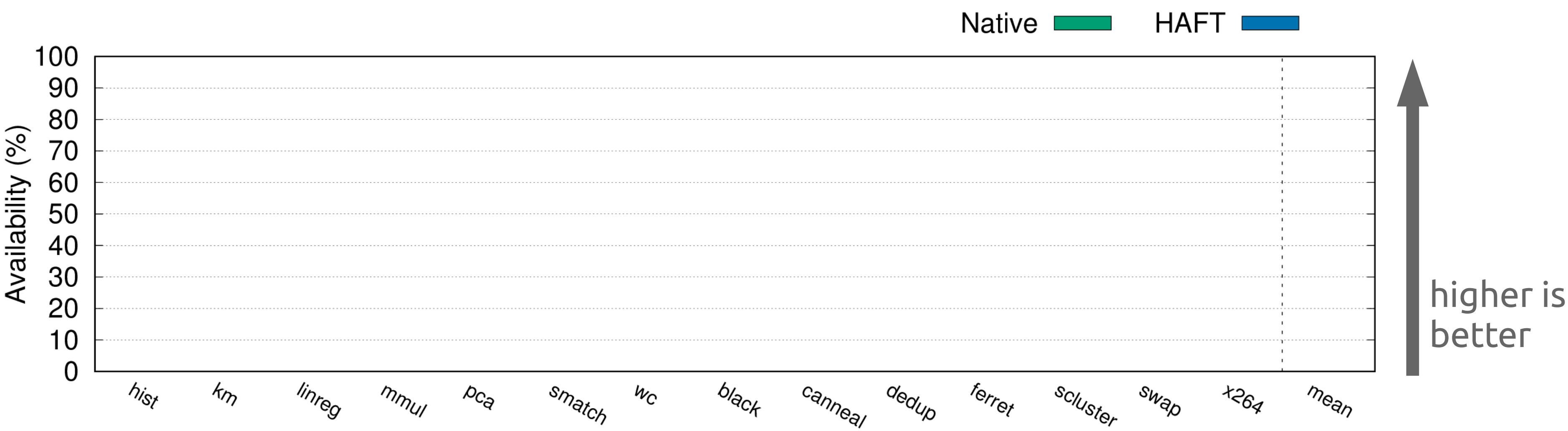
Injected faults: only **1.1**% undetected

Availability

2500 fault injections per benchmark

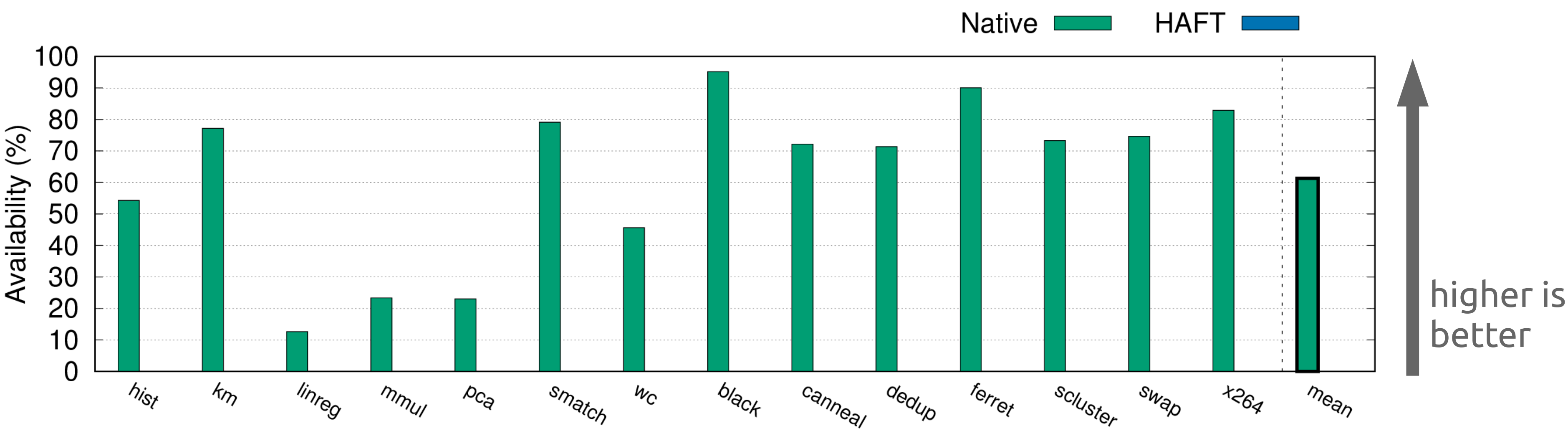
Availability

2500 fault injections per benchmark



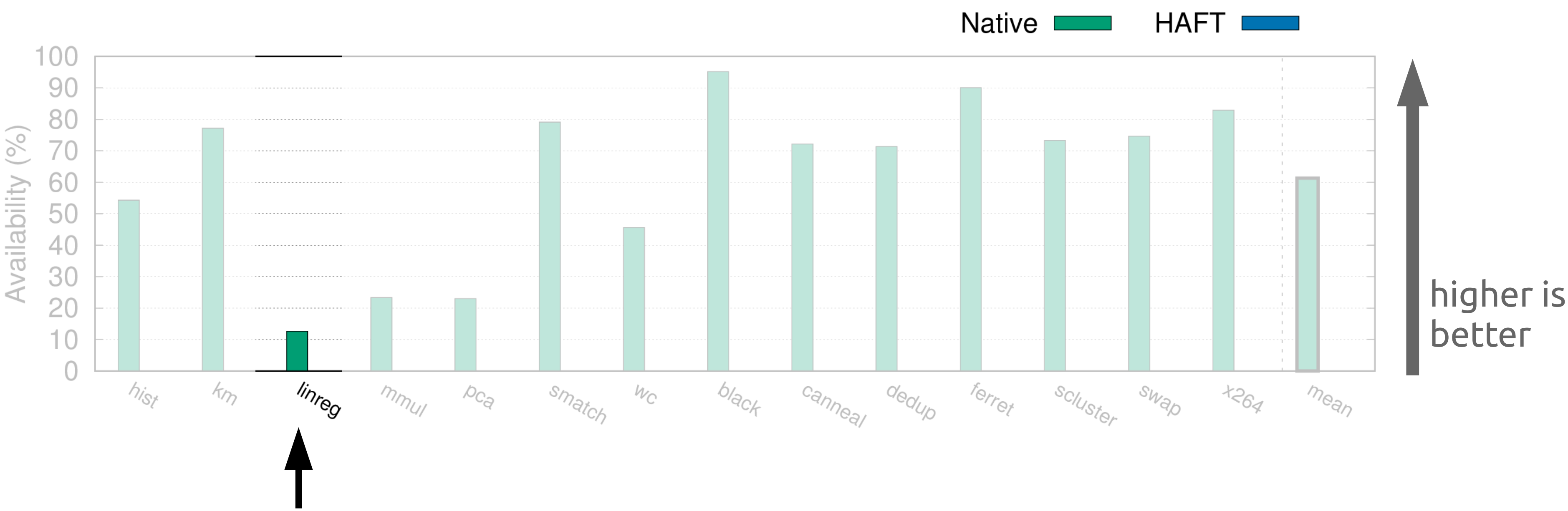
Availability

2500 fault injections per benchmark



Availability

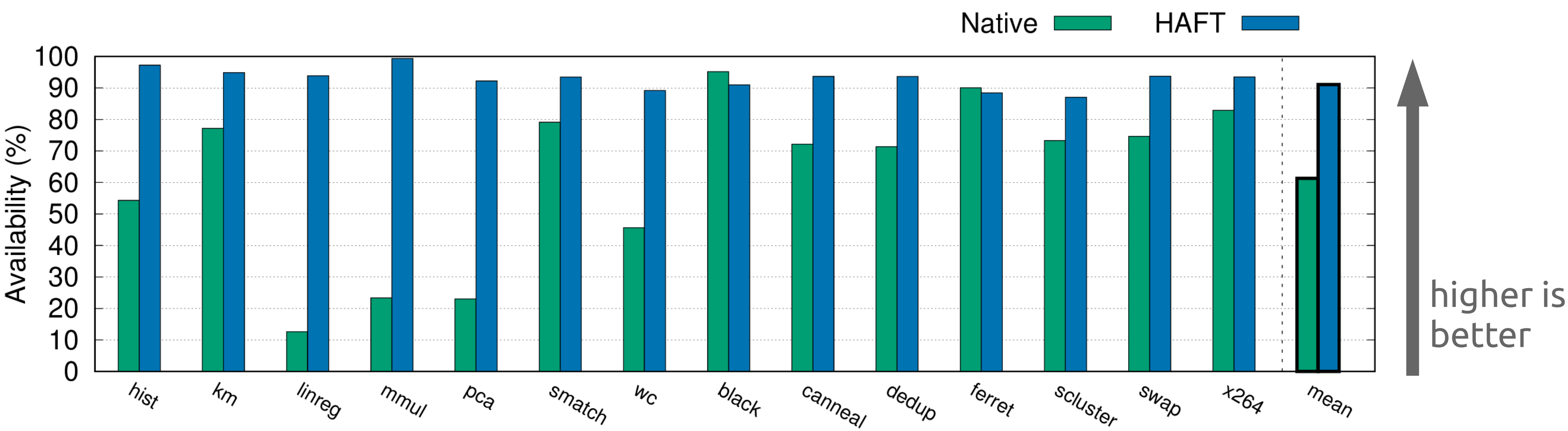
2500 fault injections per benchmark



Out of 2500 fault injections, **12%** resulted in correct execution

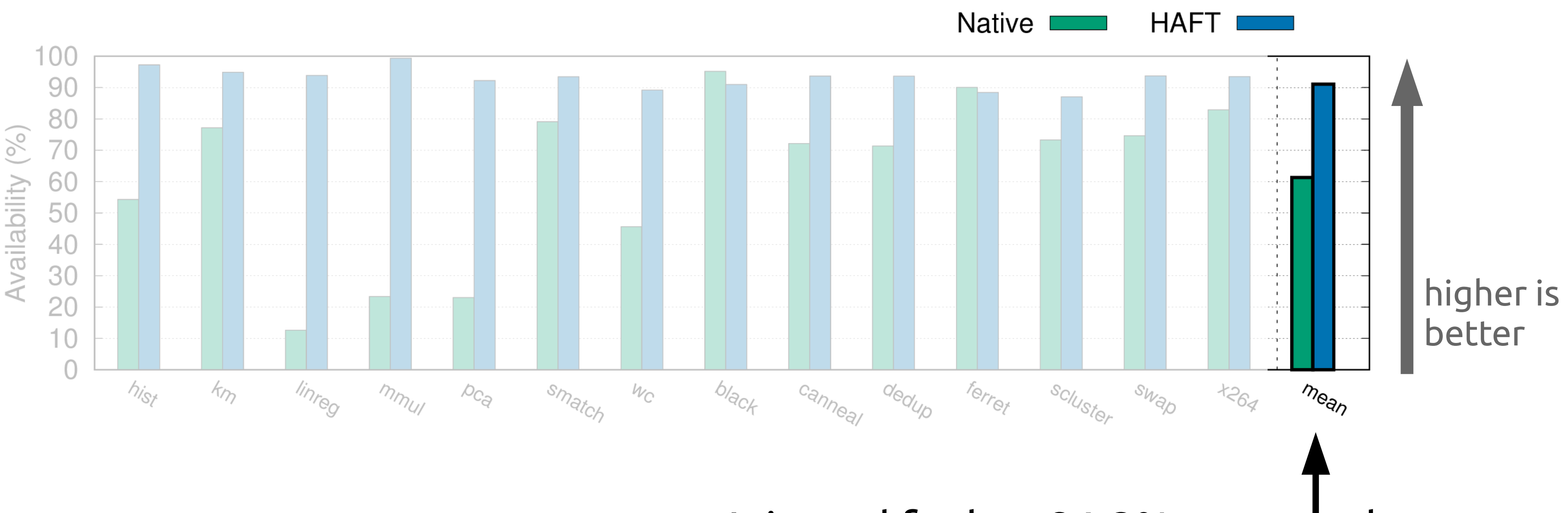
Availability

2500 fault injections per benchmark



Availability

2500 fault injections per benchmark



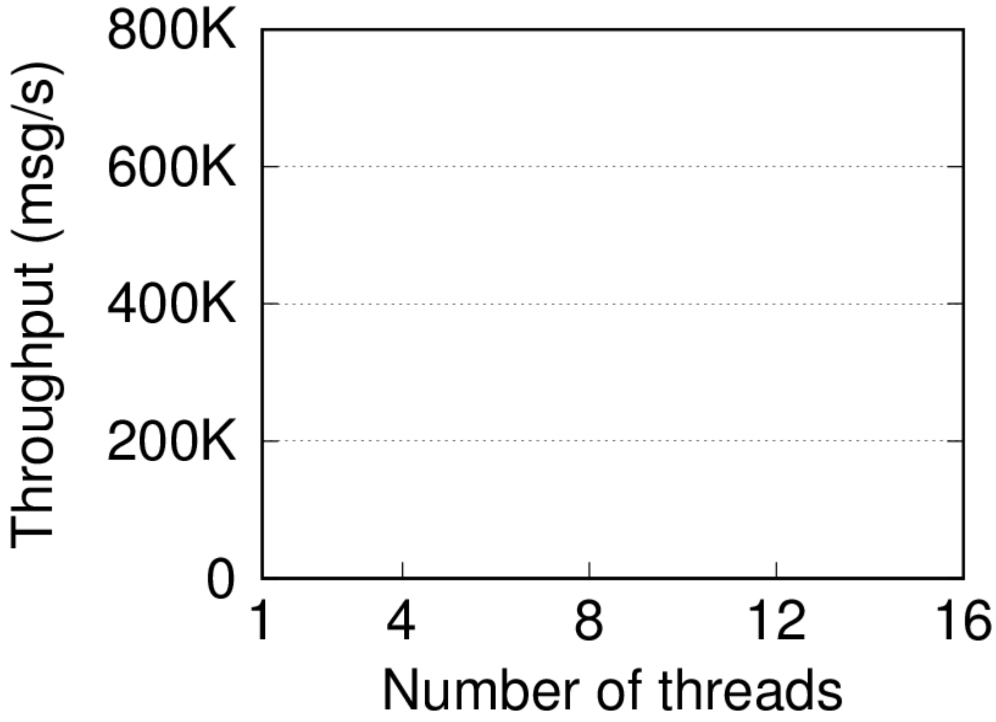
Injected faults: **91.2%** corrected
(best-effort nature of Intel TSX)

Memcached evaluation

YCSB with **95% reads, 5% writes, latest**

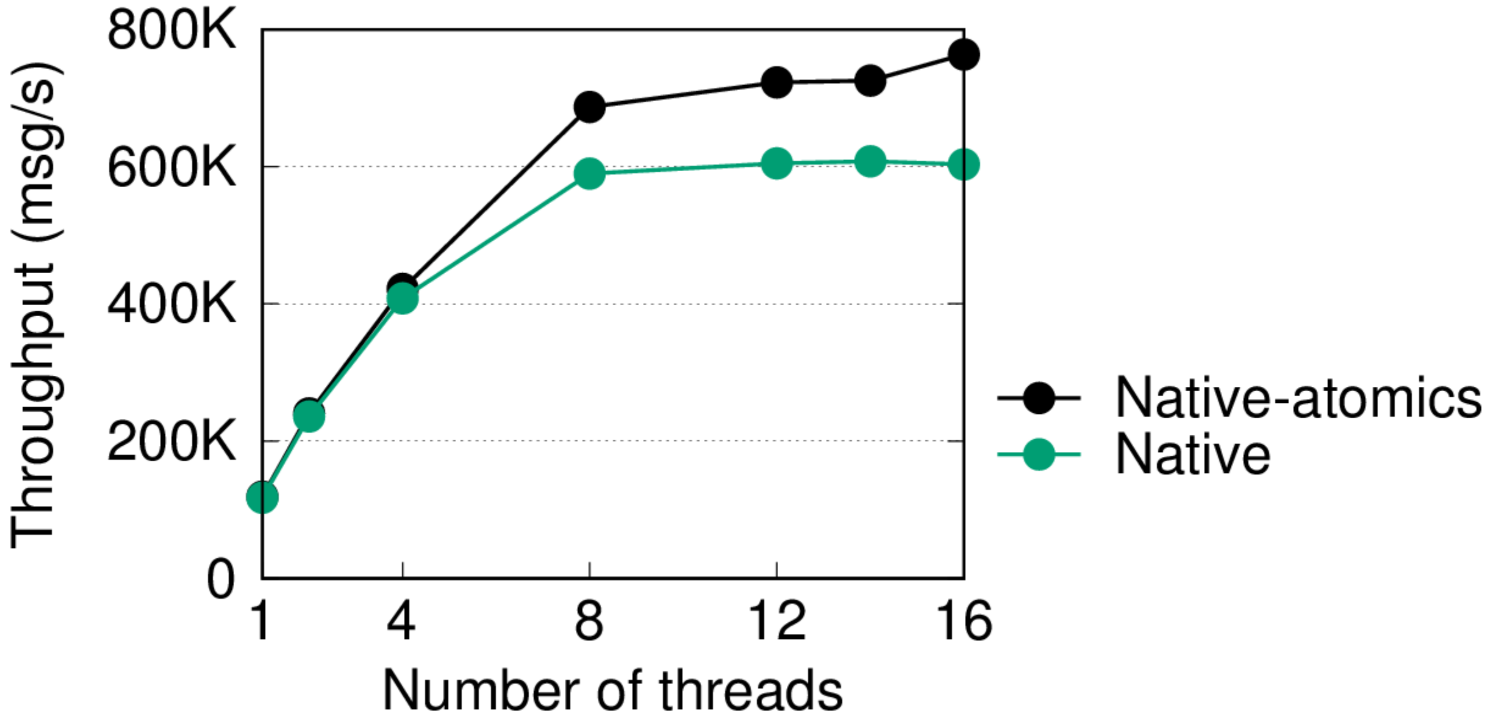
Memcached evaluation

YCSB with **95% reads, 5% writes, latest**



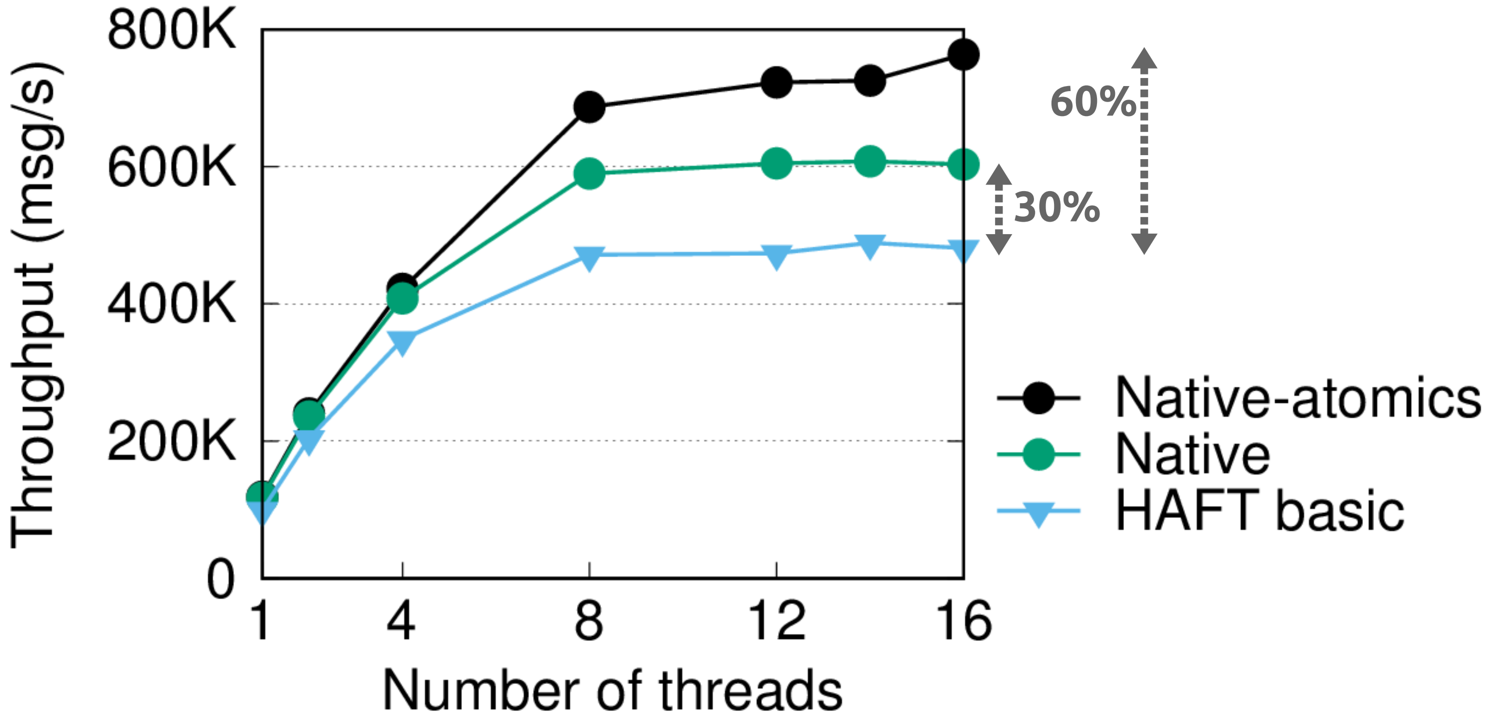
Memcached evaluation

YCSB with **95% reads, 5% writes, latest**



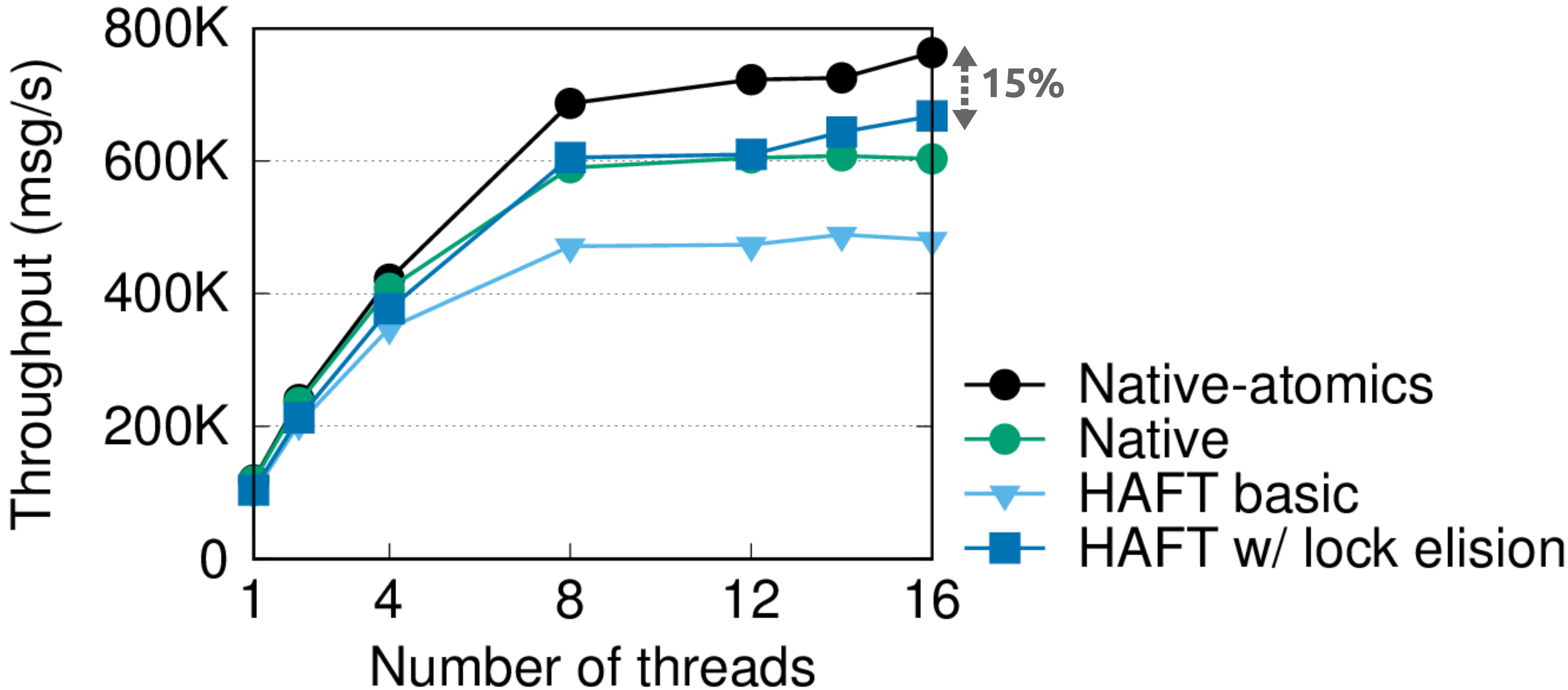
Memcached evaluation

YCSB with **95% reads, 5% writes, latest**



Memcached evaluation

YCSB with **95% reads, 5% writes, latest**



Summary

Summary

HAFT provides **fault tolerance** against arbitrary **data corruptions** caused by transient **CPU faults**

Summary

HAFT provides **fault tolerance** against arbitrary **data corruptions** caused by transient **CPU faults**

✔ Transparent

- no changes in source code
- general programming model

Summary

HAFT provides **fault tolerance** against arbitrary **data corruptions** caused by transient **CPU faults**

✔ Transparent

- no changes in source code
- general programming model

✔ Practical

- Shared-memory multithreaded programs
- Fault detection *and* fault recovery

Summary

HAFT provides **fault tolerance** against arbitrary **data corruptions** caused by transient **CPU faults**

✔ Transparent

- no changes in source code
- general programming model

✔ Practical

- Shared-memory multithreaded programs
- Fault detection *and* fault recovery

✔ Efficient

- Low performance overhead
- Relies on commodity-hardware HTM (Intel TSX)

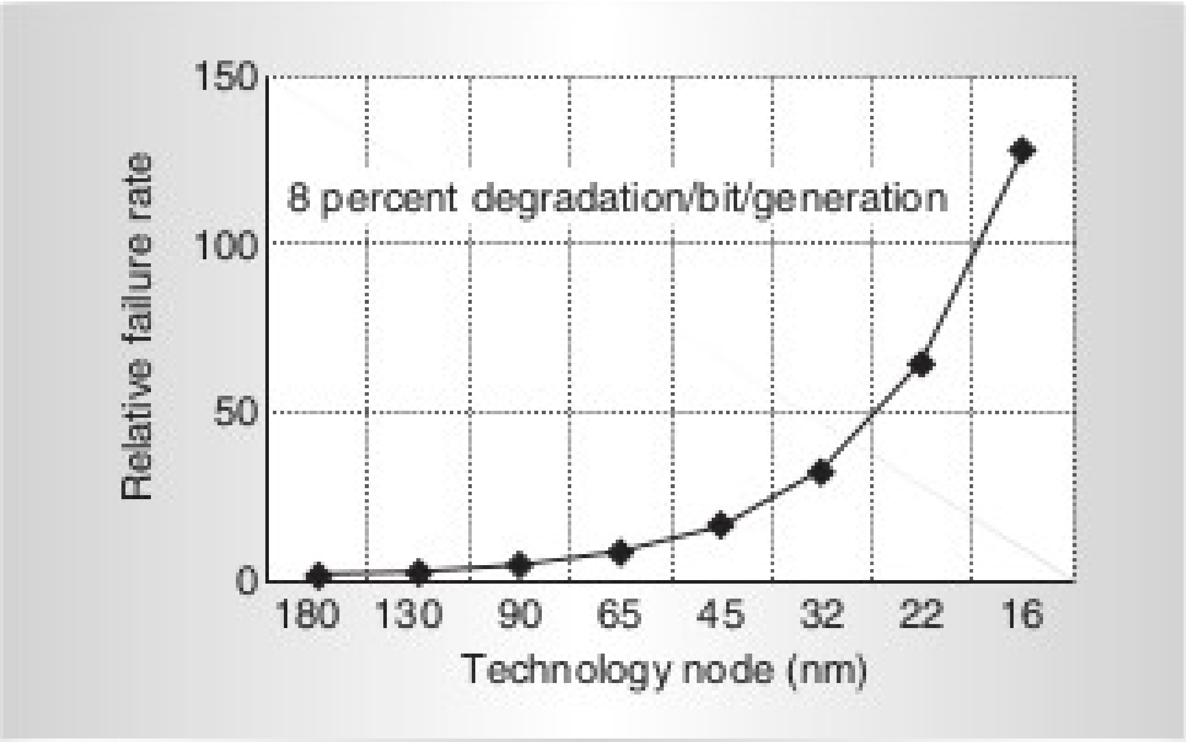
Thank you!

dmitrii.kuvaiskii@tu-dresden.de

Source code available: **https://github.com/tudinfse/haft**

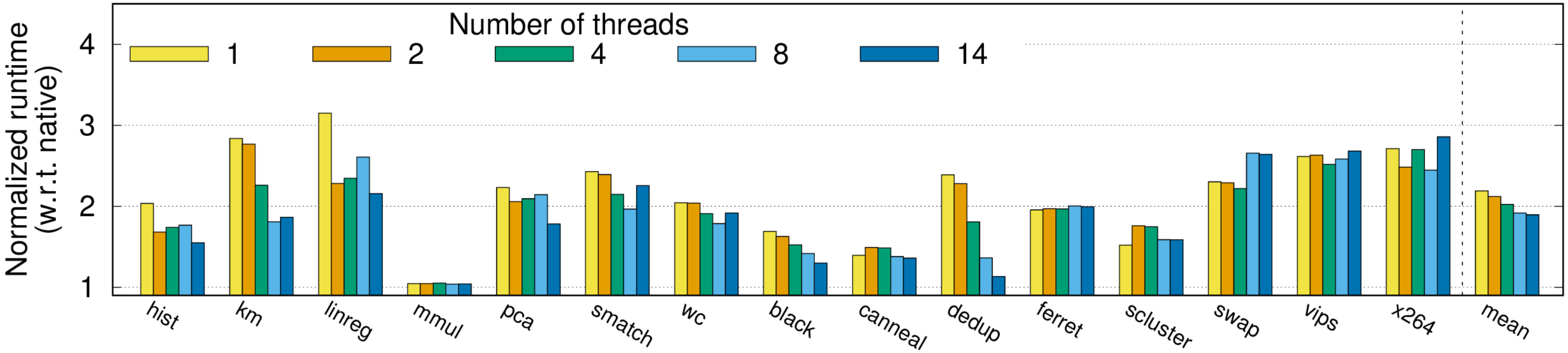
Backup slides

(Un)Reliability of CPUs



[Designing Reliable Systems from Unreliable Components, S. Borkar, Micro'05]

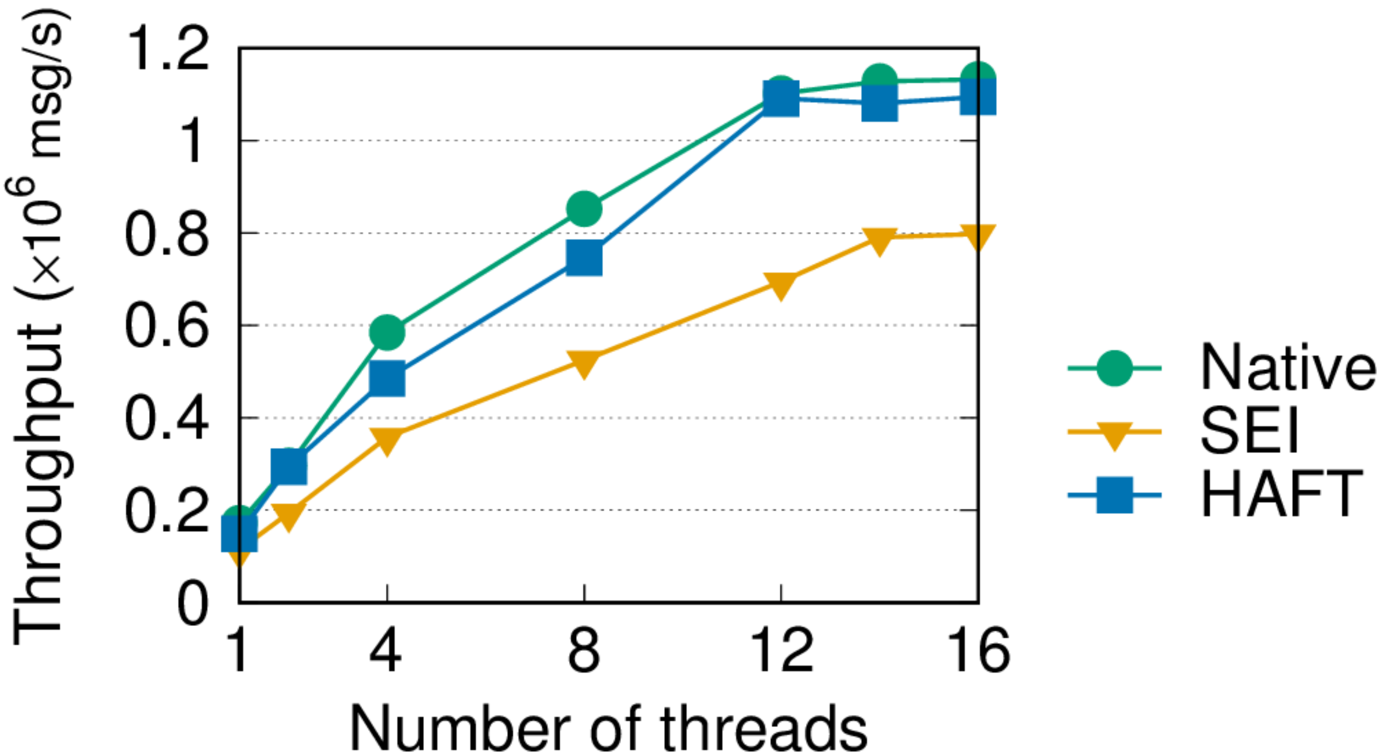
Performance evaluation



Average performance overhead is **2×** (less with more threads)

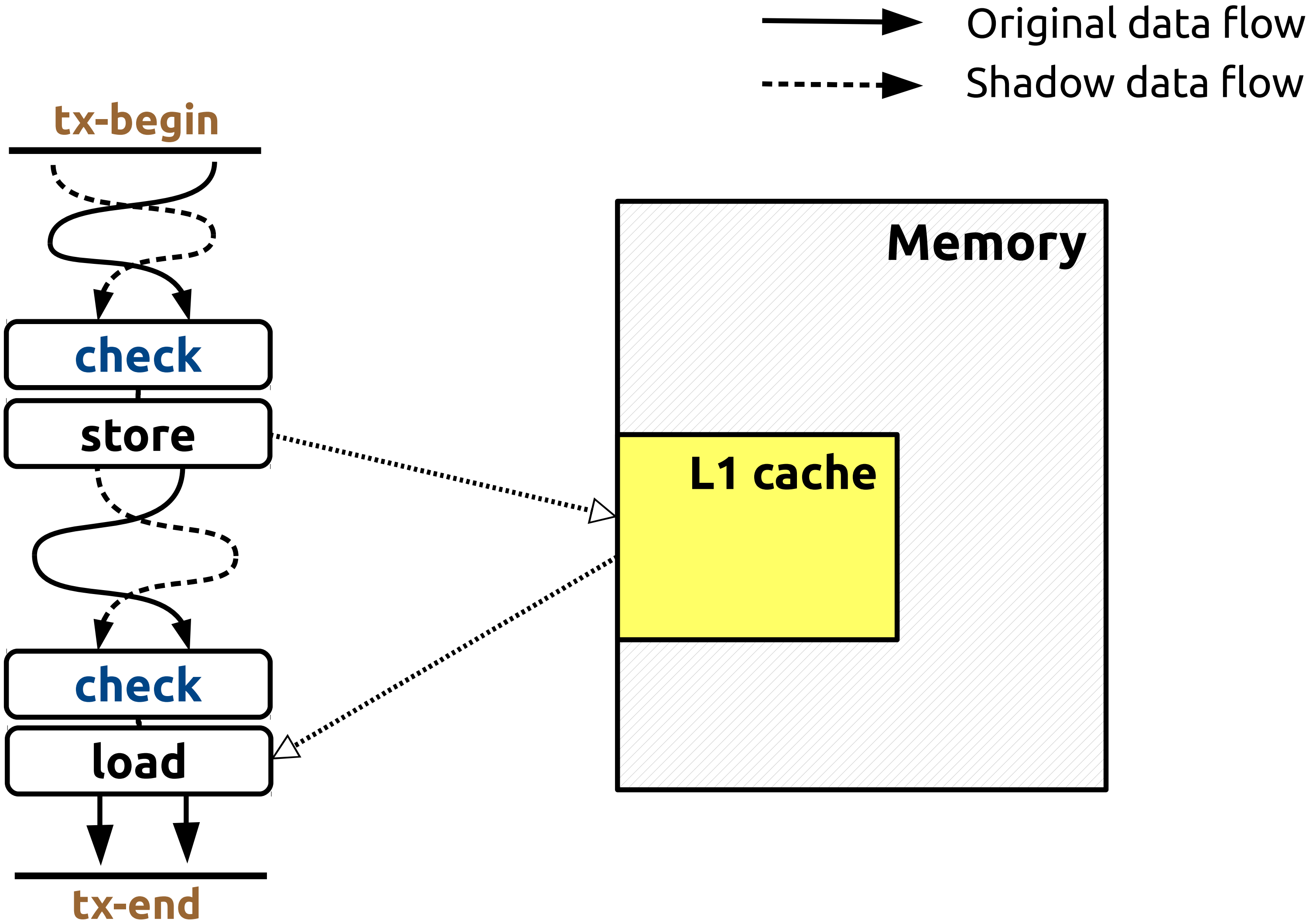
Memcached comparison

SEI (Behrens et al, NSDI'2015)



HAFT outperforms SEI by **30-40%**

HAFT: Run-time Execution



Comparison with State-of-the-Art

Approach	Resources	Multith.	Perf overhead	Fault coverage
PLR [8] DSN'07	2-3× memory usage 2-3× spare cores	No	Detection: 16.9% Recovery: 41.1%	Detection: very high Recovery: N/A
SWIFT [6] CGO'05	–	No	Detection: 41% Recovery: N/A	Detection: high Recovery: N/A
Shoestring [5] ASPLOS'10	–	No	Detection: 15-30% Recovery: N/A	Detection: medium Recovery: N/A
DAFT [10] PACT'10	2× spare cores	No	Detection: 38% Recovery: N/A	Detection: high Recovery: N/A
RAFT [9] CGO'12	2× memory usage 2× spare cores	No	Detection: 3% Recovery: N/A	Detection: very high Recovery: N/A
RomainMT [4] EMSOFT'14	2-3× memory usage 2-3× spare cores	Yes	Detection: 13-22% Recovery: 24-65%	Detection: N/A Recovery: N/A
SEI [2] NSDI'15	– (manual code changes)	Yes	Detection: 20-50% Recovery: N/A	Detection: very high Recovery: N/A
HAFT (this work)	–	Yes	Detection: **52%** Recovery: **89%**	Detection: **high** Recovery: **high**

Hardening

Limitations:

✘ Non-transparent
- Manual changes in source code [1] [2]
- Specific languages / programming models [3]

✘ Impractical
- Only single-threaded programs [1] [5-10]
- Only fail-stop execution [1] [2] [5] [6] [8-10]

✘ Inefficient
- Requires spare cores / deterministic execution [4] [8-10]
- Memory overhead [8] [9]

References

[1] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini.
Practical hardening of crash-tolerant systems. ATC'12

[2] D. Behrens, M. Serafini, S. Arnautov, F. P. Junqueira, and C. Fetzer.
Scalable error isolation for distributed systems. NSDI'15

[3] P. Bhatotia, A. Wieder, R. Rodrigues, F. Junqueira, and B. Reed.
Reliable data-center scale computations. LADIS'10

[4] B. Döbel and H. Härtig.
Can we put concurrency back into redundant multithreading? EMSOFT'14

[5] S. Feng, S. Gupta, A. Ansari, and S. Mahlke.
Shoestring: Probabilistic soft error reliability on the cheap. ASPLOS'10

[6] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August.
SWIFT: Software implemented fault tolerance. CGO'05

[7] G. A. Reis, J. Chang, and D. I. August.
Automatic instruction-level software-only recovery. Micro'07

[8] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors.
Using process-level redundancy to exploit multiple cores for transient fault tolerance. DSN'07

[9] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August.
Runtime asynchronous fault tolerance via speculation. CGO'12

[10] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August.
DAFT: Decoupled acyclic fault tolerance. PACT'10