TECHNISCHE UNIVERSITÄT DRESDEN

CAN WE PUT CONCURRENCY BACK INTO REDUNDANT MULTITHREADING?

Björn Döbel and Hermann Härtig (TU Dresden)

New Delhi, 14.10.2014

Motivation: Transient Hardware Faults

- Radiation-induced soft errors
  - Mainly an issue in avionics+space
- DRAM errors in large data centers
  - Google Study: $> 2\%$ failing DRAM DIMMs per year[1]
  - ECC insufficient[2]
- Decreasing transistor sizes $\rightarrow$ higher rate of errors in CPU functional units[3]

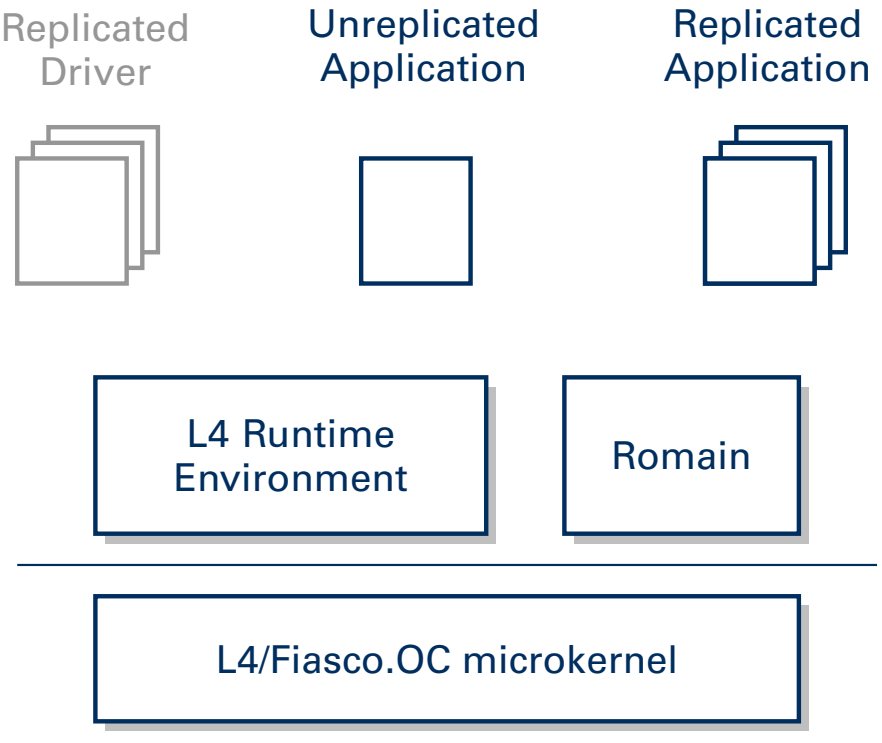
---

[1] Schroeder, Pinheiro, Weber: **DRAM Errors in the Wild: A Large-Scale Field Study**, SIGMETRICS 2009

[2] Hwang, Stefanovici, Schroeder: **Cosmic Rays Don't Strike Twice**, ASPLOS 2012

[3] Dixit, Wood: **The Impact of New Technology on Soft Error Rates**, IRPS 2011

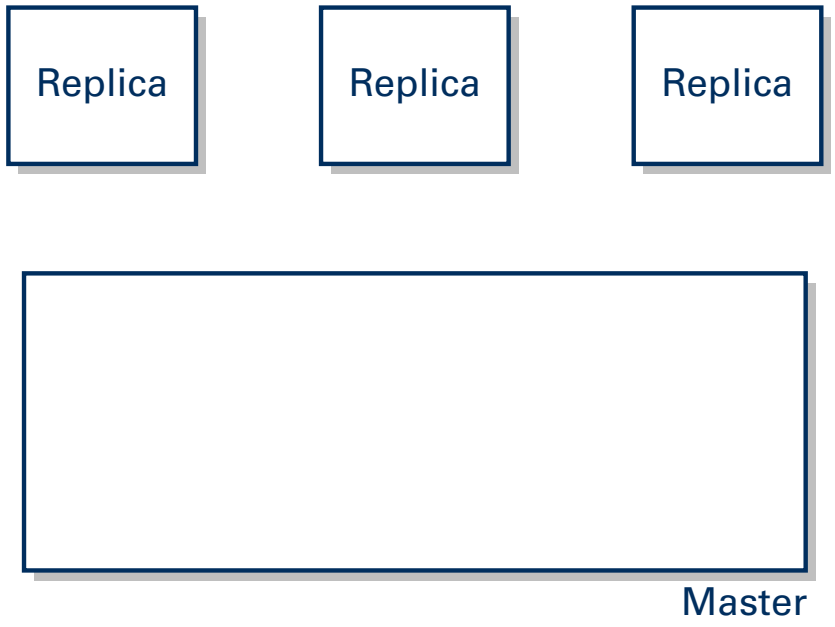
ASTEROID Operating System



Romain: Structure

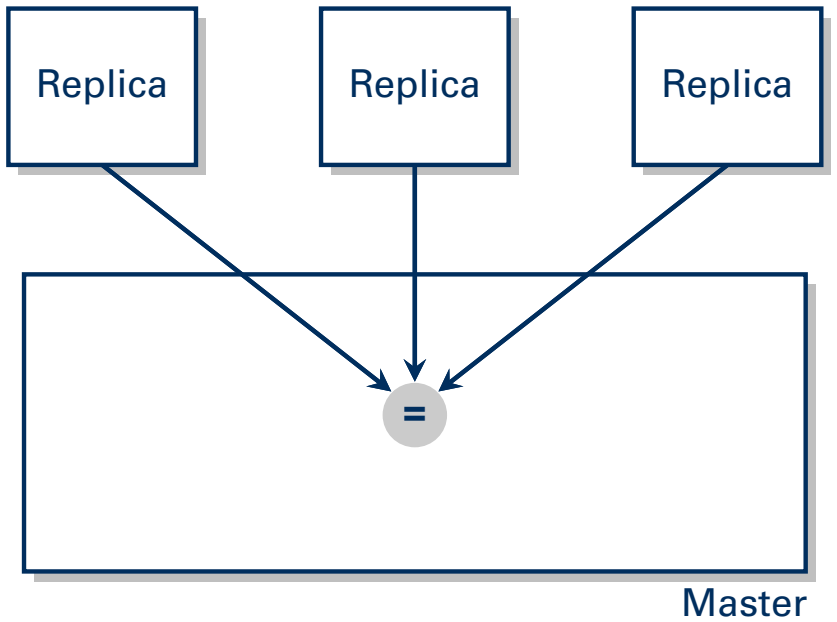
Details: Döbel, Härtig, Engel: **Operating System Support for Redundant Multithreading**, EMSOFT 2012

Romain: Structure



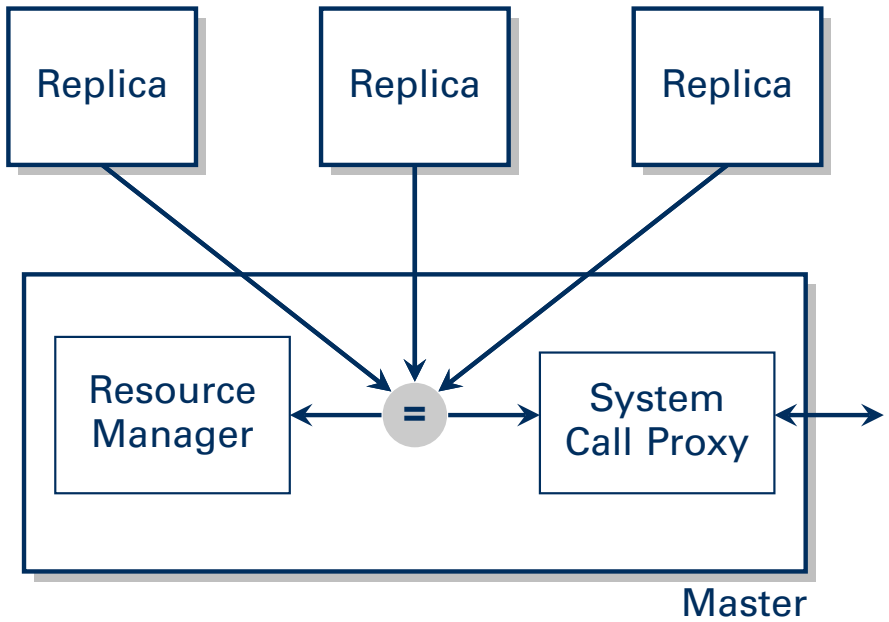
Details: Döbel, Härtig, Engel: **Operating System Support for Redundant Multithreading**, EMSOFT 2012

Romain: Structure



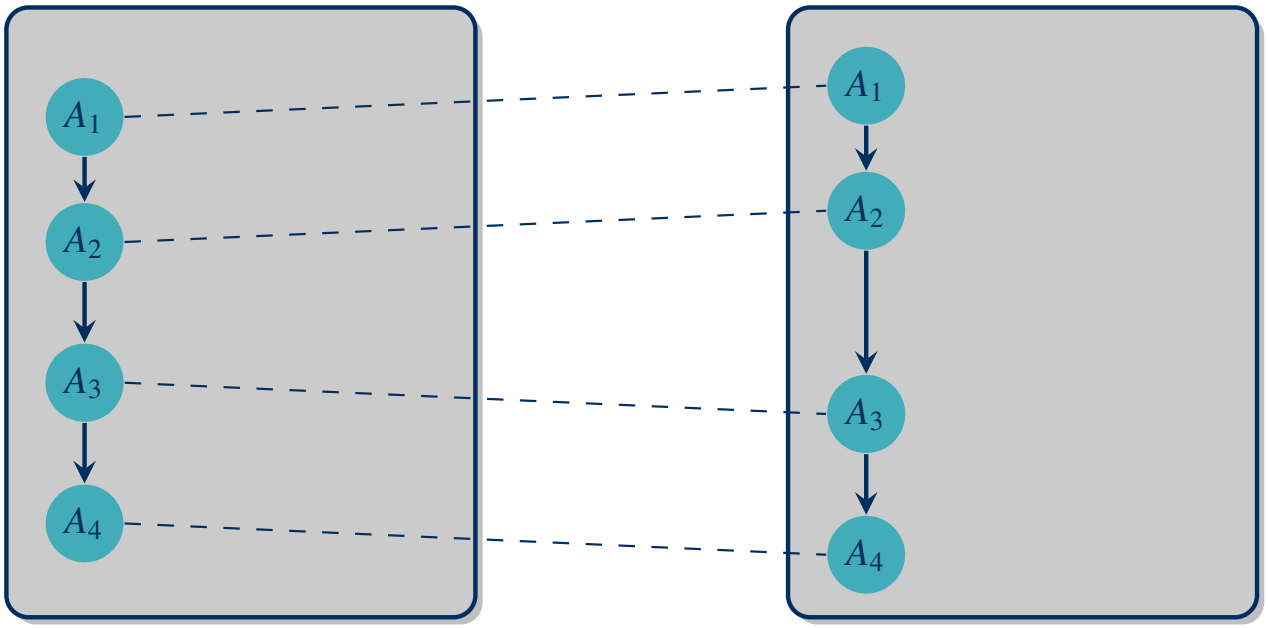
Details: Döbel, Härtig, Engel: **Operating System Support for Redundant Multithreading**, EMSOFT 2012

Romain: Structure

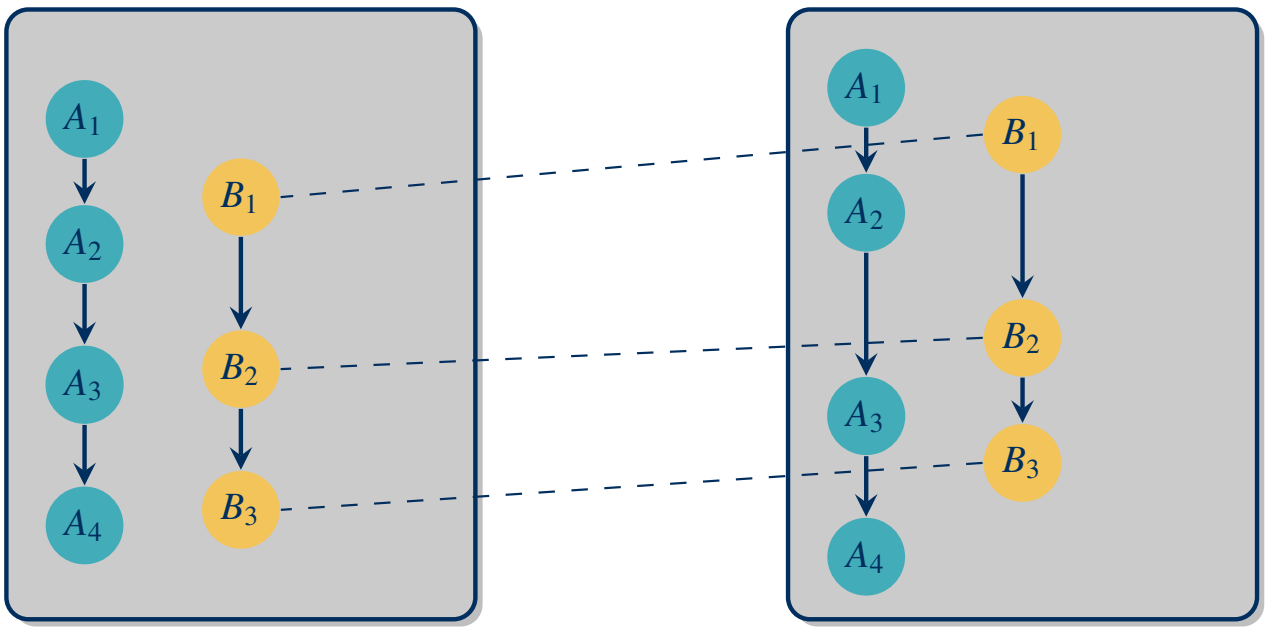


Details: Döbel, Härtig, Engel: **Operating System Support for Redundant Multithreading**, EMSOFT 2012

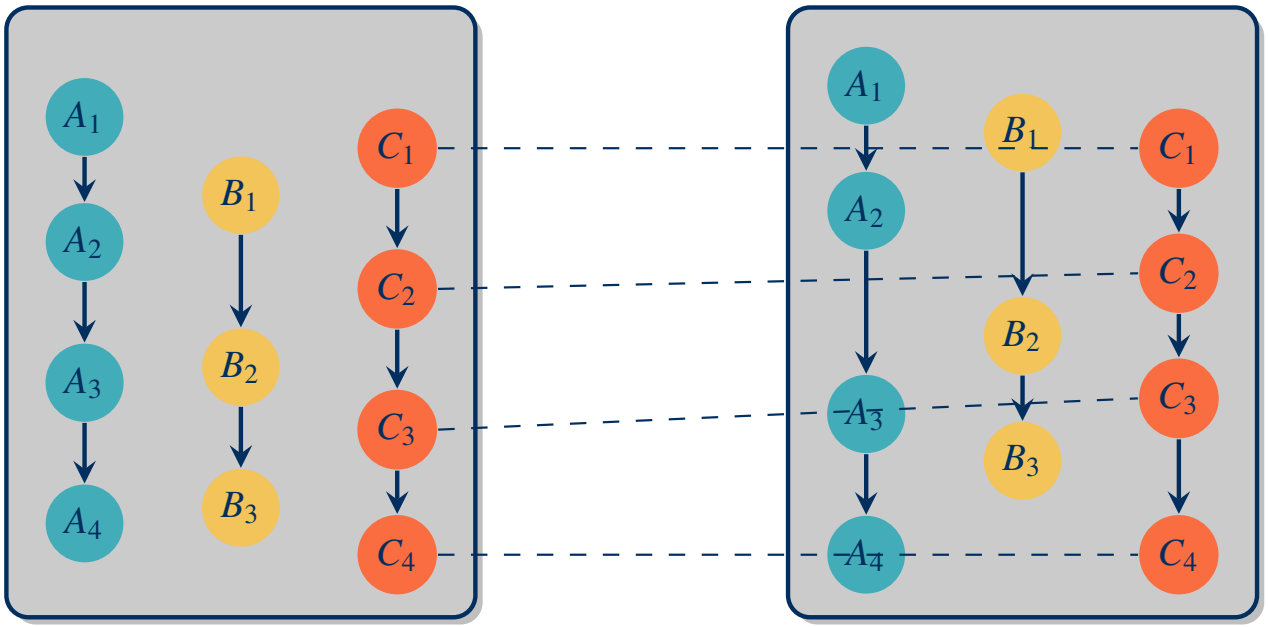
How About Multithreading?



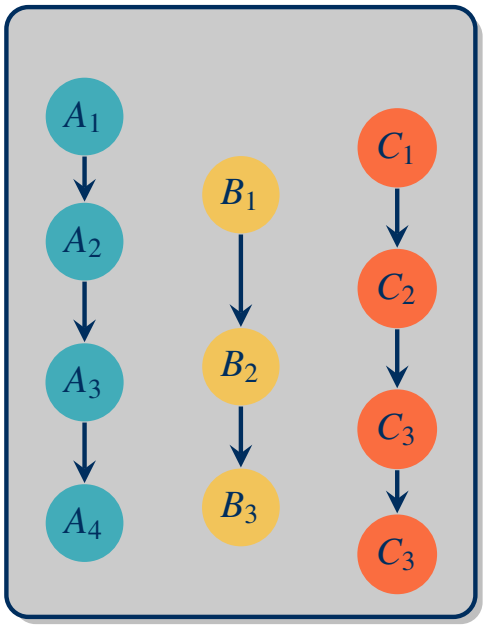
How About Multithreading?

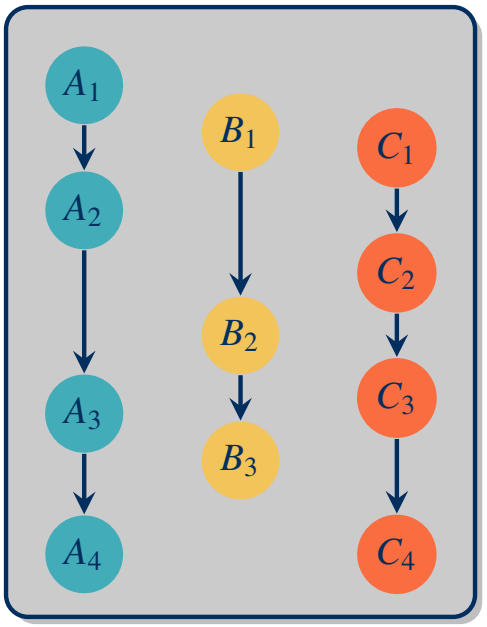


How About Multithreading?

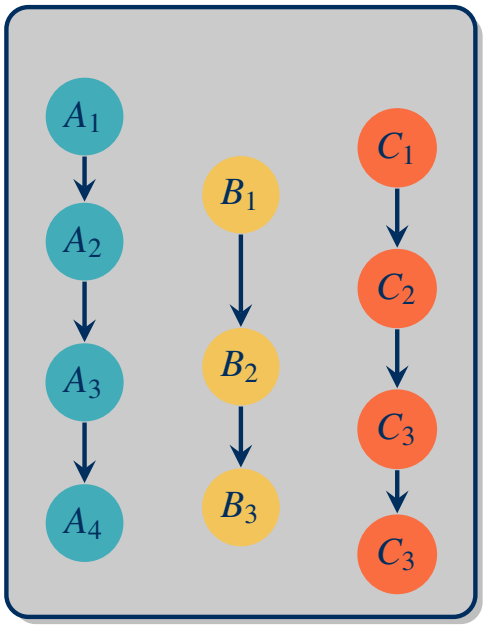


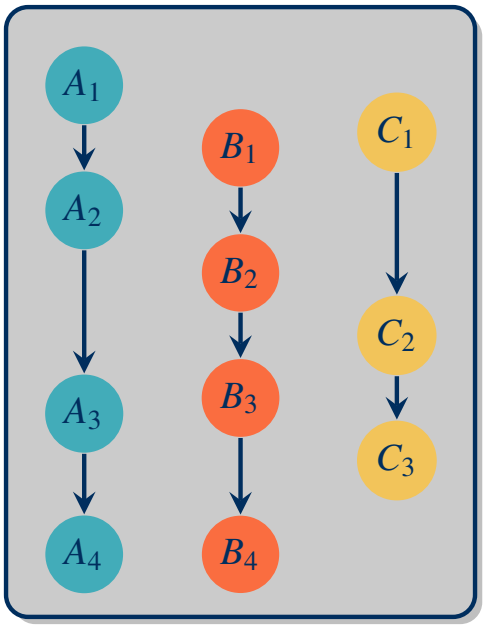
Problem: Nondeterminism





Problem: Nondeterminism





Nondeterminism Example

```
int x = 1;
pthread_mutex_t m =
    PTHREAD_MUTEX_DEFAULT;

void *thread_A(void *data)
{
  pthread_mutex_lock(&m);
  x = x + 1;
  pthread_mutex_unlock(&m);
  return NULL;
}

void *thread_B(void *data)
{
  pthread_mutex_lock(&m);
  x = x * 2;
  pthread_mutex_unlock(&m);
  return NULL;
}
```

Nondeterminism Example

```
int x = 1;
pthread_mutex_t m =
    PTHREAD_MUTEX_DEFAULT;

void *thread_A(void *data)
{
  pthread_mutex_lock(&m);
  x = x + 1;
  pthread_mutex_unlock(&m);
  return NULL;
}

void *thread_B(void *data)
{
  pthread_mutex_lock(&m);
  x = x * 2;
  pthread_mutex_unlock(&m);
  return NULL;
}
```

- race-free (locks)
- (A;B) $\rightarrow$ x = 4
- (B;A) $\rightarrow$ x = 3

Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions:[4] no support for binary-only software

[4] Bergan et al.: **Core-Det: A Compiler and Runtime System for Deterministic Multithreaded Execution**, ASPLOS 2010

[5] Aviram et al.: **Efficient System-enforced Deterministic Parallelism**, OSDI 2010

[6] Mushtaq et al.: **Efficient Software Based Fault Tolerance Approach on Multicore Platforms**, DATE 2013

[7] Olszewski et al: **Kendo: Efficient Deterministic Multithreading in Software**, ASPLOS 2009

Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions:[4] no support for binary-only software
- Workspace-Consistent Memory:[5] Requires per-replica and per-thread memory copies

[4] Bergan et al.: **Core-Det: A Compiler and Runtime System for Deterministic Multithreaded Execution**, ASPLOS 2010

[5] Aviram et al.: **Efficient System-enforced Deterministic Parallelism**, OSDI 2010

[6] Mushtaq et al.: **Efficient Software Based Fault Tolerance Approach on Multicore Platforms**, DATE 2013

[7] Olszewski et al: **Kendo: Efficient Deterministic Multithreading in Software**, ASPLOS 2009

Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions:[4] no support for binary-only software
- Workspace-Consistent Memory:[5] Requires per-replica and per-thread memory copies
- Lock-Based Determinism
  - Reliance on ECC-protected memory[6]
  - Our work reuses ideas from Kendo.[7]

---

4 Bergan et al.: **Core-Det: A Compiler and Runtime System for Deterministic Multithreaded Execution**, ASPLOS 2010

5 Aviram et al.: **Efficient System-enforced Deterministic Parallelism**, OSDI 2010

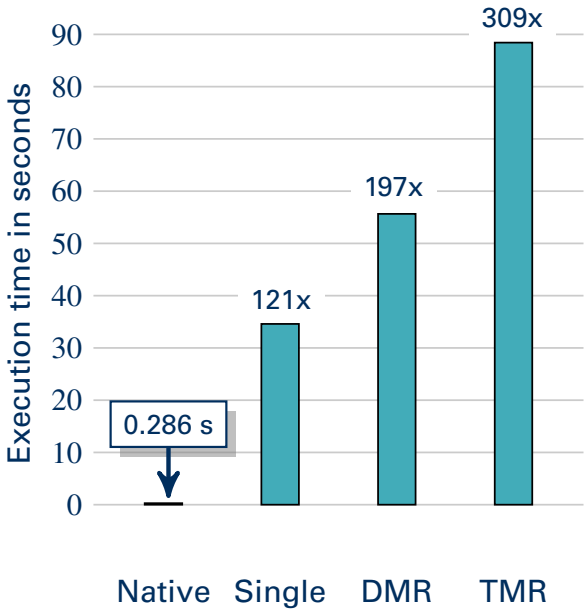
6 Mushtaq et al.: **Efficient Software Based Fault Tolerance Approach on Multicore Platforms**, DATE 2013

7 Olszewski et al: **Kendo: Efficient Deterministic Multithreading in Software**, ASPLOS 2009

Enforced Determinism

- Adapt `libpthread`: place `INT3` into four functions
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`
  - `__pthread_lock`
  - `__pthread_unlock`
- Lock operations reflected to *RomainMT* master
- Master enforces lock ordering

Enforced Determinism: Microbenchmark

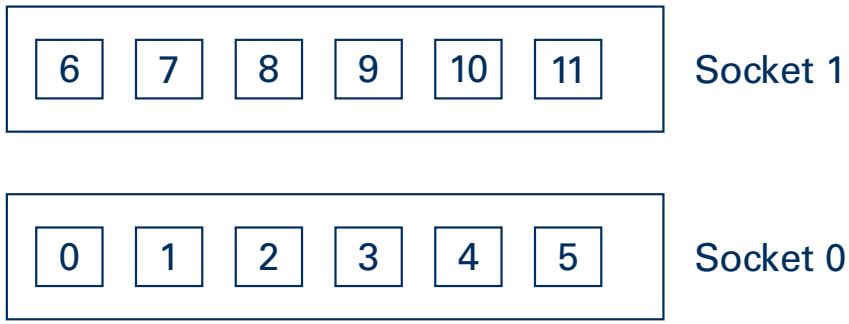


2 Threads:

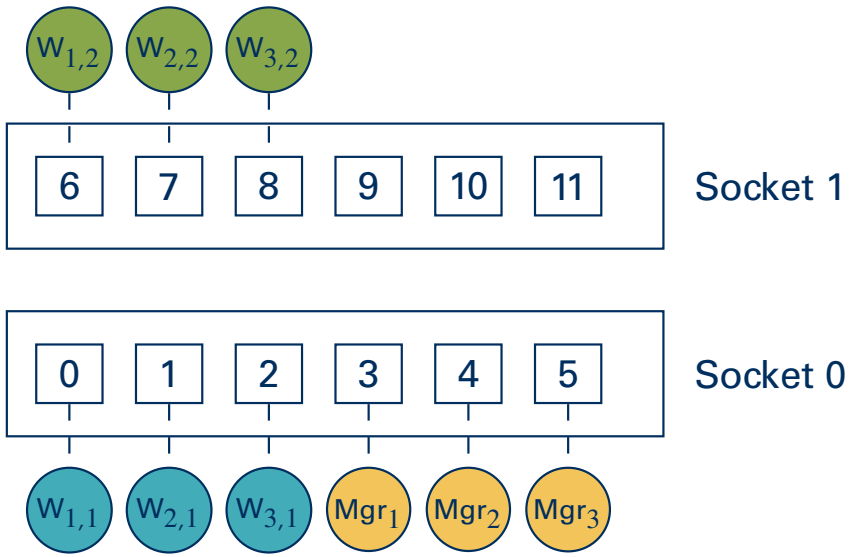
```
int x = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_DEFAULT;

void *thread(void *data)
{
  for (int i = 0; i < 5000000; ++i)
  {
    pthread_mutex_lock(&m);
    x = x + 1;
    pthread_mutex_unlock(&m);
  }
  return NULL;
}
```

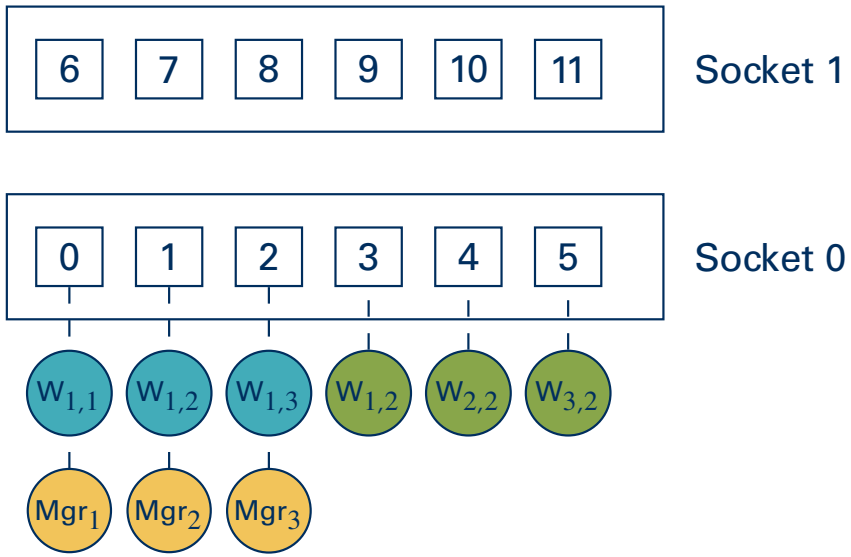
Optimization Opportunities?



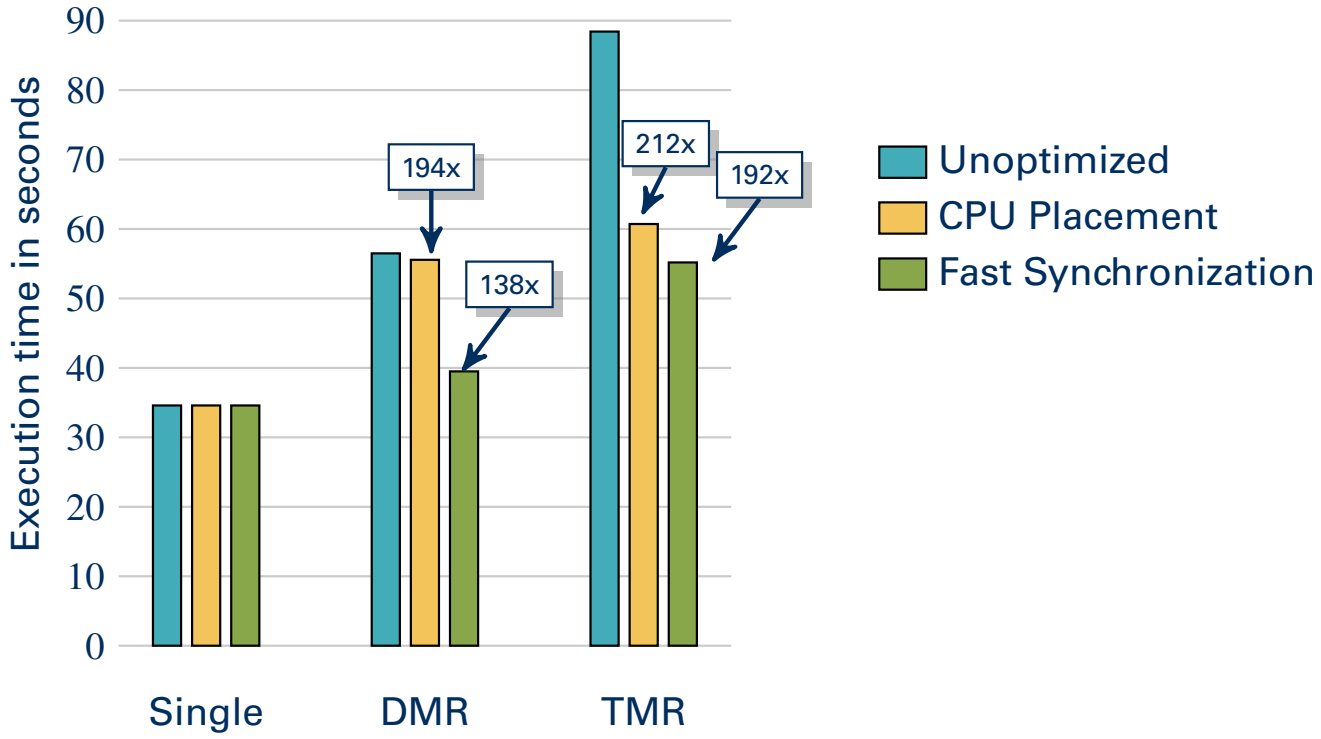
Optimization Opportunities?



Optimization Opportunities?

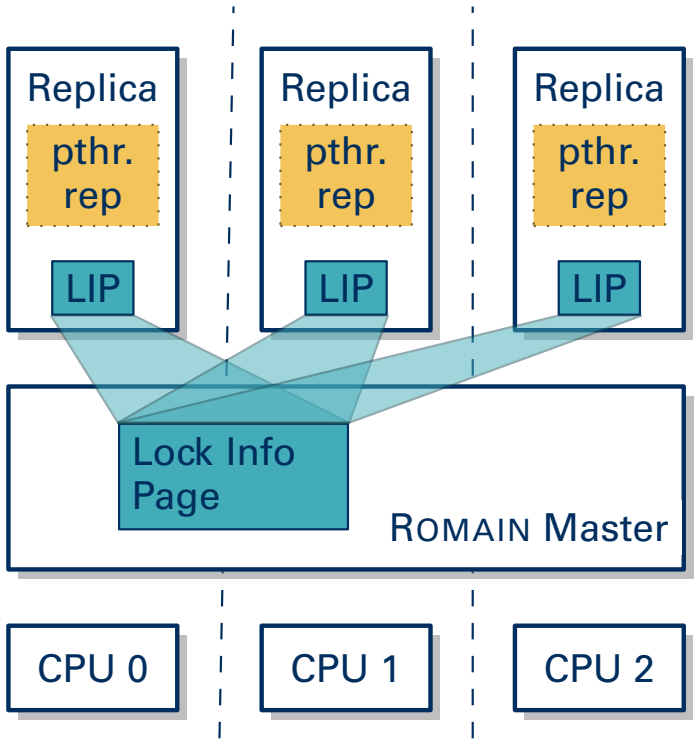


Optimized Enforced Determinism

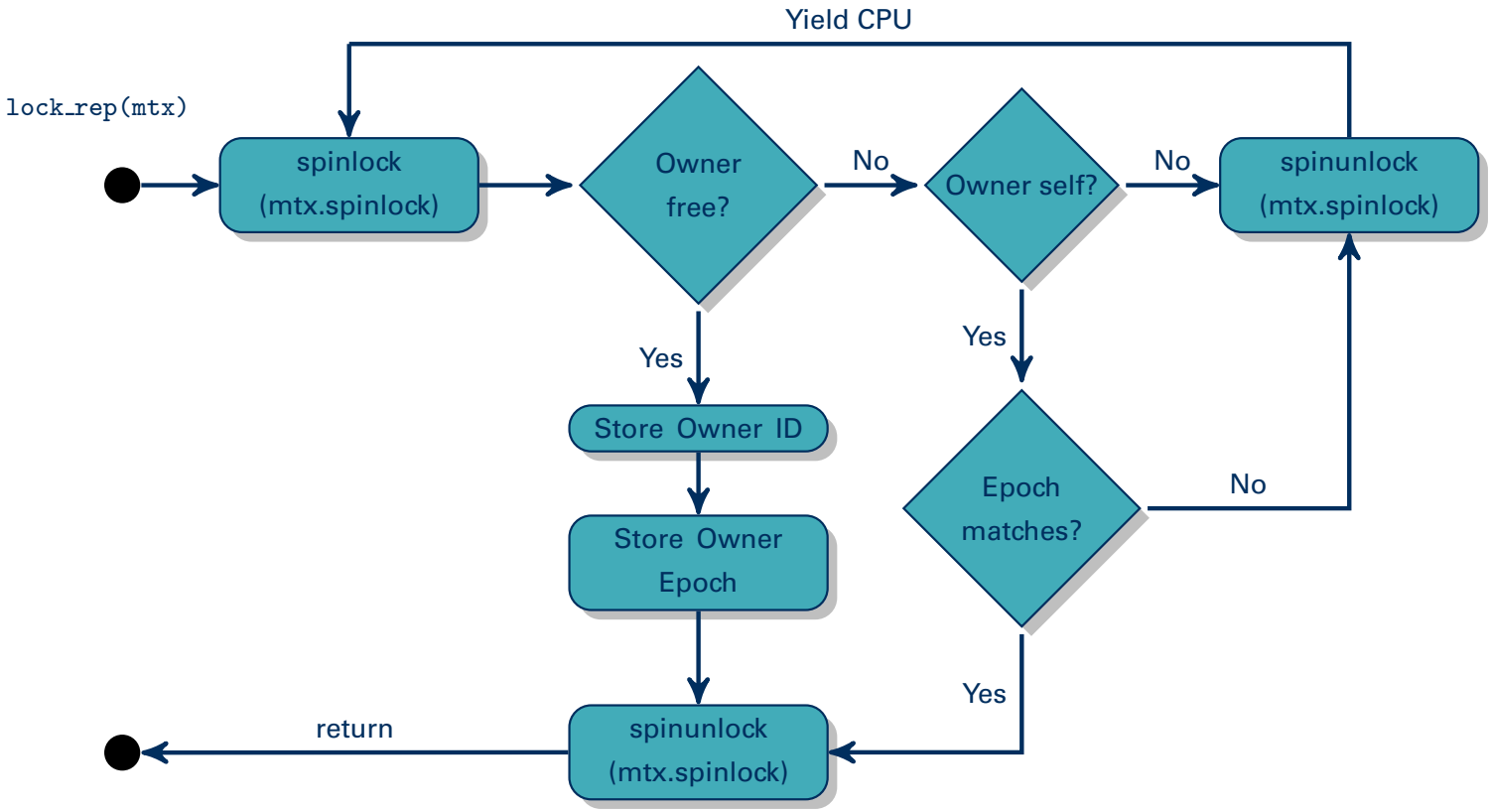


Cooperative Determinism

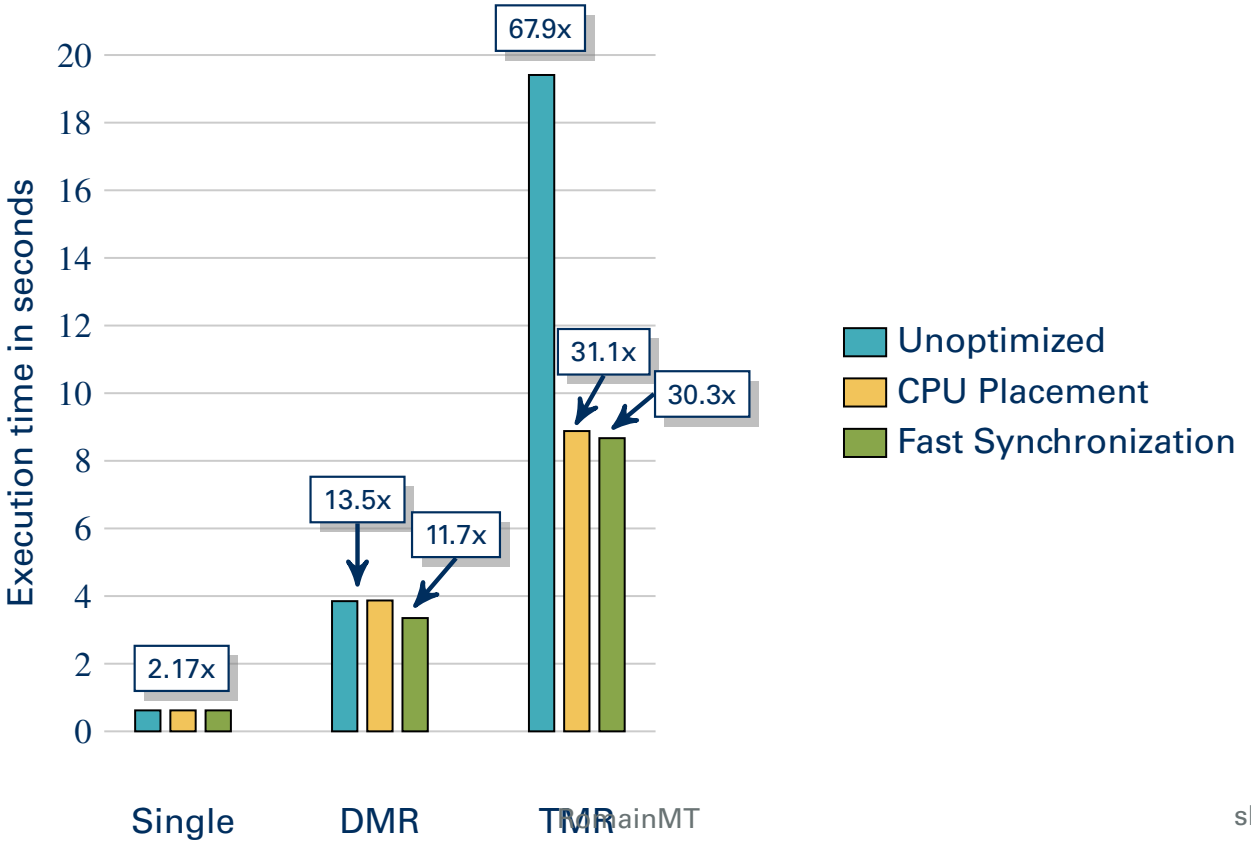
- Replication-aware `libpthread`
- Replicas agree on acquisition order w/o master invocation
- Trade-off: `libpthread` becomes single point of failure



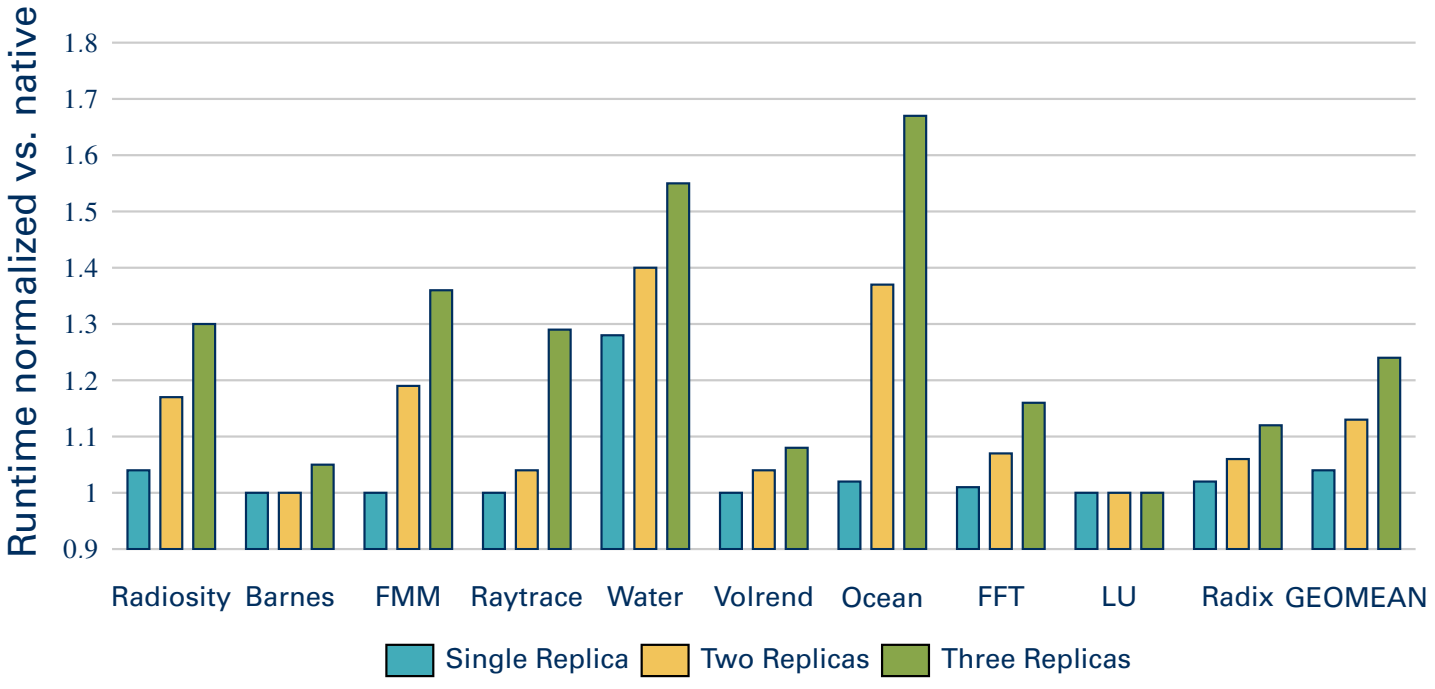
Cooperation: Lock Acquisition



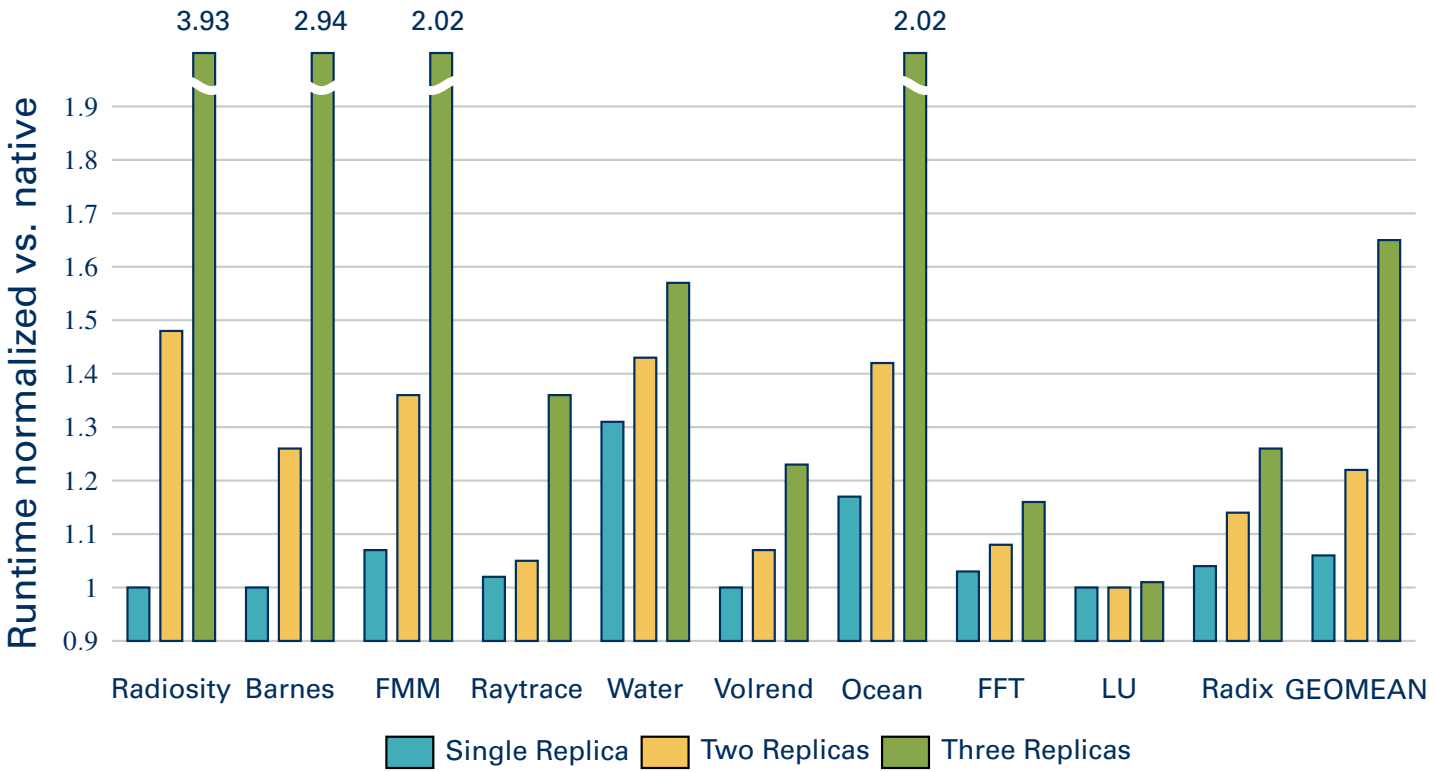
Cooperative Determinism: Microbenchmark



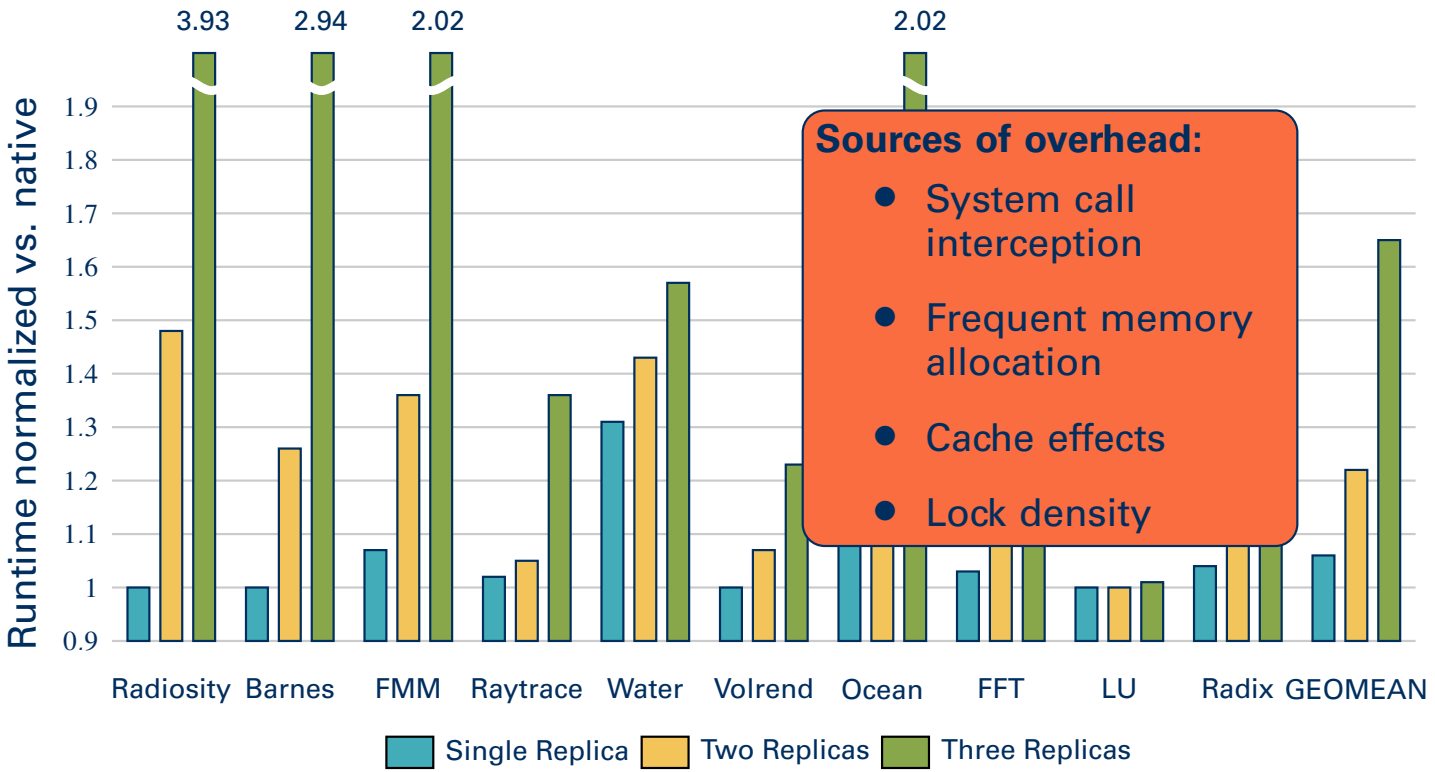
Overhead: SPLASH2, 2 workers



Overhead: SPLASH2, 4 workers



Overhead: SPLASH2, 4 workers



Summary

- Redundant Multithreading as an OS Service
- Binary-only applications
- Multithreaded Replication using lock-based determinism
  - Enforced Determinism
  - Cooperative Determinism
  - Lock Density limits performance
- Application overhead for TMR: 24% (2 workers), 65% (4 workers)

http://spp1500.itec.kit.edu