Inherently Deterministic Operating Systems

Fachgruppe Betriebssysteme, Herbsttreffen 2021, 2021-09-22

Stefan Reif[1], Timo Hönig[2], Wolfgang Schröder-Preikschat[1]

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg
[2] Ruhr-Universität Bochum

Scalability and Predictability

Scalable systems

- $>$ 100 cores
- communicating partitions

(!) system noise

(!) tail latencies

Scalability and Predictability

Scalable systems

- $>$ 100 cores
- communicating partitions

(!) system noise

(!) tail latencies

Predictable systems

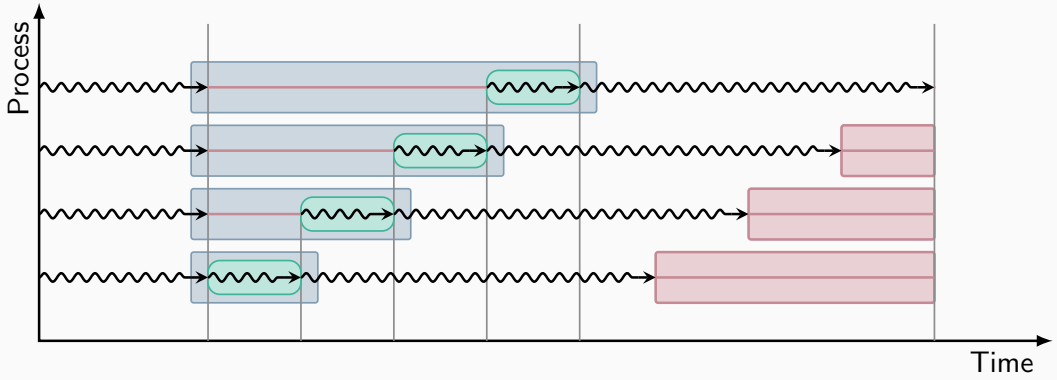
- $\approx$ 1 core (isolated partitions)

(!) low performance

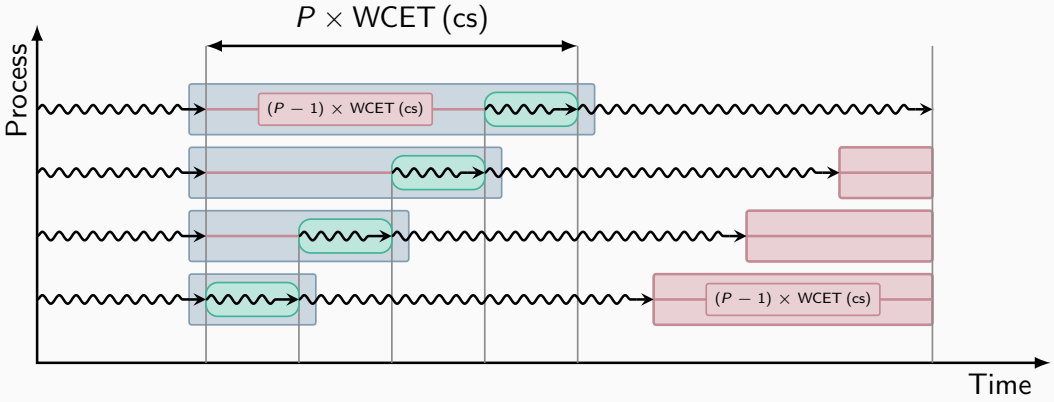
(!) bad energy efficiency

(!) poor utilisation

Interferences – Toy Example



Interferences – Toy Example



Problem Formulation

- OSs orchestrate communication
  - ↯ Interferences due to shared data structures
- OSs manage shared stateful data
  - ↯ Need for synchronisation
- How to build operating systems that are …
  - ? scalable
  - ? predictable
  - ? and analysable?

Approach

The IDOS Approach

- Inherently Deterministic Operating Systems (“IDOS”)
  - → IDOS := costs are independent of...
    - ...system states & input data
    - ...the number of resources
    - ...concurrent operations
- Predictability & concurrency as ~~afterthoughts~~ first-class citizens

The IDOS Approach

- Inherently Deterministic Operating Systems (“IDOS”)
  - → IDOS := costs are independent of...
    - ...system states & input data ⇒ single-path code
    - ...the number of resources ⇒ $\mathcal{O}(1)$ algorithms
    - ...concurrent operations ⇒ wait-freedom
- Predictability & concurrency as ~~afterthoughts~~ first-class citizens

The IDOS Approach

- Inherently Deterministic Operating Systems (“IDOS”)
  - → IDOS := costs are independent of...
    - ...system states & input data ⇒ single-path code
    - ...the number of resources ⇒ $\mathcal{O}(1)$ algorithms if possible
    - ...concurrent operations ⇒ wait-freedom but no universal $\mathcal{O}(1)$ approach
- Predictability & concurrency as ~~afterthoughts~~ first-class citizens

The IDOS Approach

- Inherently Deterministic Operating Systems (“IDOS”)
  - → IDOS := costs are independent of...
    - ...system states & input data ⇒ single-path code
    - ...the number of resources ⇒ $\mathcal{O}(1)$ algorithms if possible
    - ...concurrent operations ⇒ wait-freedom but no universal $\mathcal{O}(1)$ approach
- Predictability & concurrency as ~~afterthoughts~~ first-class citizens
  - ⊕ Scalability: $\mathcal{O}(1) < \mathcal{O}(P)$ for large P
  - ⊕ Predictability: costs are constant
  - ⊕ Analysability: trivial, no overestimation, incremental
  - ? Implementability?
  - ? Efficiency?

Implementability

Implementation of an IDOS prototype

- DetOX—the **Det**erministic **O**peratin**g** **S**ystem
- Identify …
  1. …hardware requirements
     - ⚠ Constant-time multi-core hardware does not exist
     - ⇒ {Functional, non-functional} hardware features?
     - ⇒ {Necessary, desirable} hardware features?
  2. …suitable algorithms & data structures
     - ⚠ Non-universality
     - ⇒ Concurrent $\mathcal{O}(1)$ operations
  3. …implementable application-level interface
     - ⇒ Deterministic coordination & communication

The Road to Determinism

1. Approach
2. Design
3. Implementation
4. Constant-Time Programming
5. Discussion
6. Evaluation
7. Conclusion

Design

Design of DetOX | Strategies

System behaviour

1. Explicit thread placement
   - Deterministic control flow location
   - Thread migration supported
2. Cooperative scheduling
   - No involuntary preemptions
   - Voluntary release of CPU supported
3. Analysable memory demand
   - Constant memory demand per object
   - No need for run-time allocation
4. No resource limitations
   - Dynamic object creation supported

Design of DetOX | Strategies

System behaviour

1. Explicit thread placement
   - Deterministic control flow location
   - Thread migration supported
2. Cooperative scheduling
   - No involuntary preemptions
   - Voluntary release of CPU supported
3. Analysable memory demand
   - Constant memory demand per object
   - No need for run-time allocation
4. No resource limitations
   - Dynamic object creation supported

Design guidelines

- Worst-case minimisation
  - Worst-case $=$ average-case
- Data access restrictions
  - Data-structure partitioning
  - Implicit ownership model
- Redundancy elimination
  - Avoid inconsistency correction overhead
  - Consider concurrent updates
- Corner case elimination
  - Invariants for Fail-safe operations
  - Unconditional branch execution

Novel Concepts

- Synchronising queues
  - Communication & coordination
  - Efficient wait-free signalling
- Core management threads
  - Thread state changes as events
  - Synchronisation simplification
- Migration-based synchronisation
  - Mutual exclusion by thread migration
- Companion cubes
  - Fail-safe concurrent memory management
  - Support for preallocation
- DETOX semaphor interface
  - Thread interaction unification
  - Corner case elimination
- Concurrent ownership model
  - Restrict & control data access

Implementation

Implementation

- Dynamic memory management
  - `memory alloc`, `free`
- Dynamic thread management
  - `thread create`, `exit`, `join`, ...
- Actor-based synchronisation
  - `actor create`, `run`
  - (SPSC) `reply create`, `set`, `get`, ...
- Message-based coordination
  - (MPSC) `pipe create`, `recv`, `send`, ...
- Migration-based synchronisation
  - `thread migrate`, `pause`
- Lock-based synchronisation
  - `mutex create`, `enter`, `leave`
  - `condition create`, `wait`, `wake`

Constant-Time Programming

Code Transformation for IDOS | Overview

High-level Algorithm

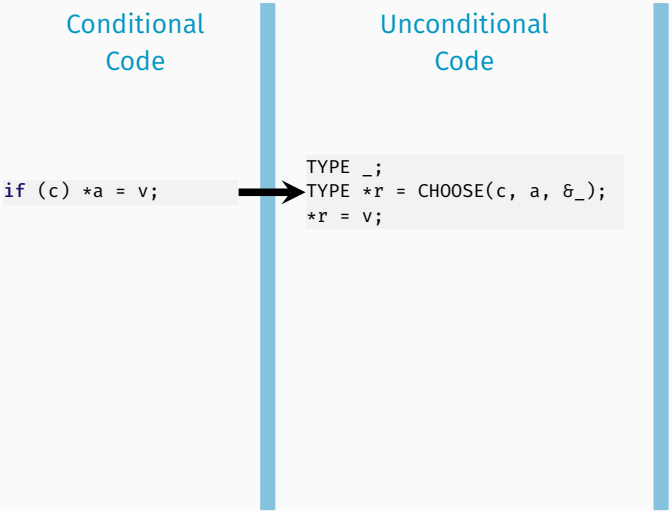
1. Start with wait-free code (with $\mathcal{O}(1)$ complexity)
2. Transform loops
   - Discussed in literature
   - Practical solution: unrolling
   - Little relevance due to $\mathcal{O}(1)$ code
3. Transform conditions
   - Discussed in literature
   - Problem: atomic memory operations
   - “interim & choose”
4. Avoid run-time errors
   - Division by zero, invalid pointer dereference, …
   - “interim & choose” again

Code Transformation for IDOS | Conditional Statements

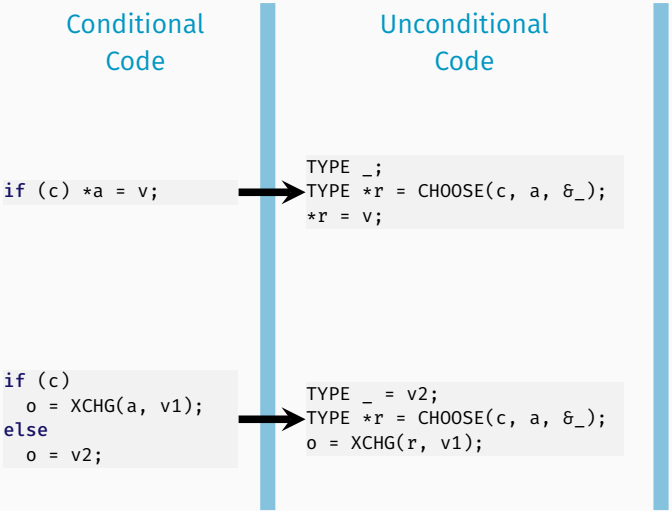
Conditional Code

```
if (c) *a = v;
```

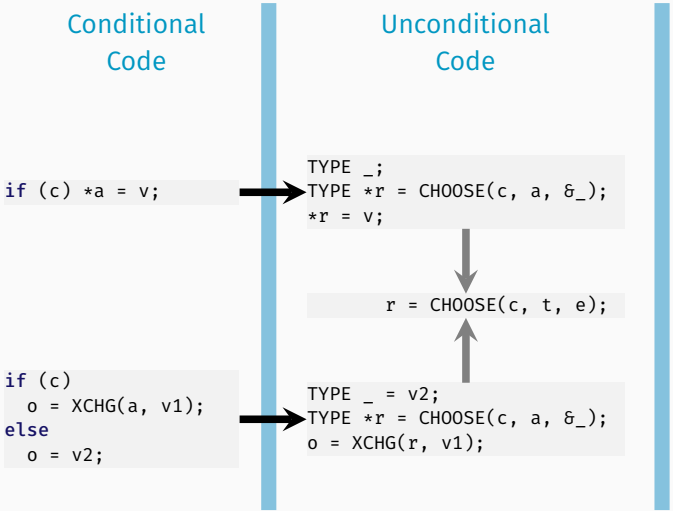
Code Transformation for IDOS | Conditional Statements



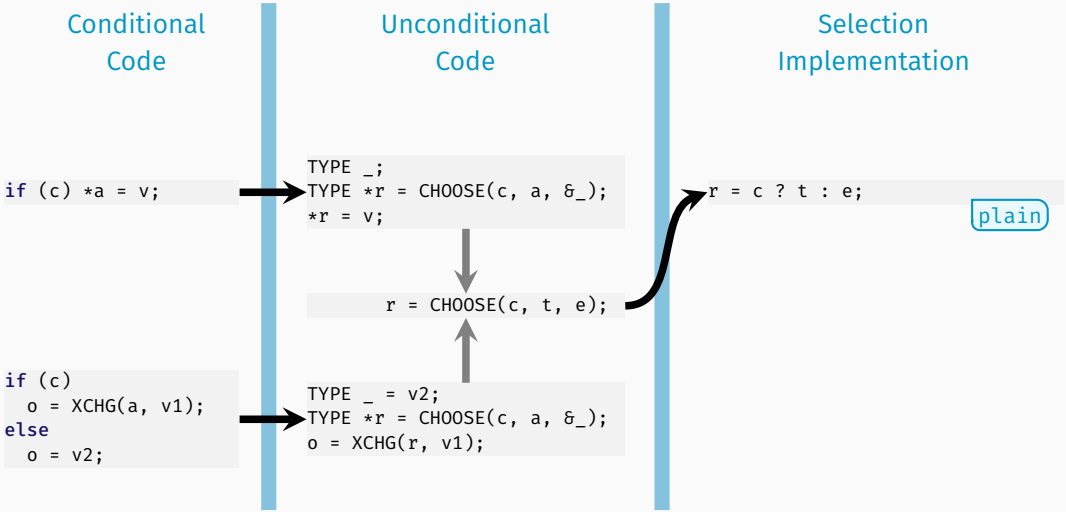
Code Transformation for IDOS | Conditional Statements



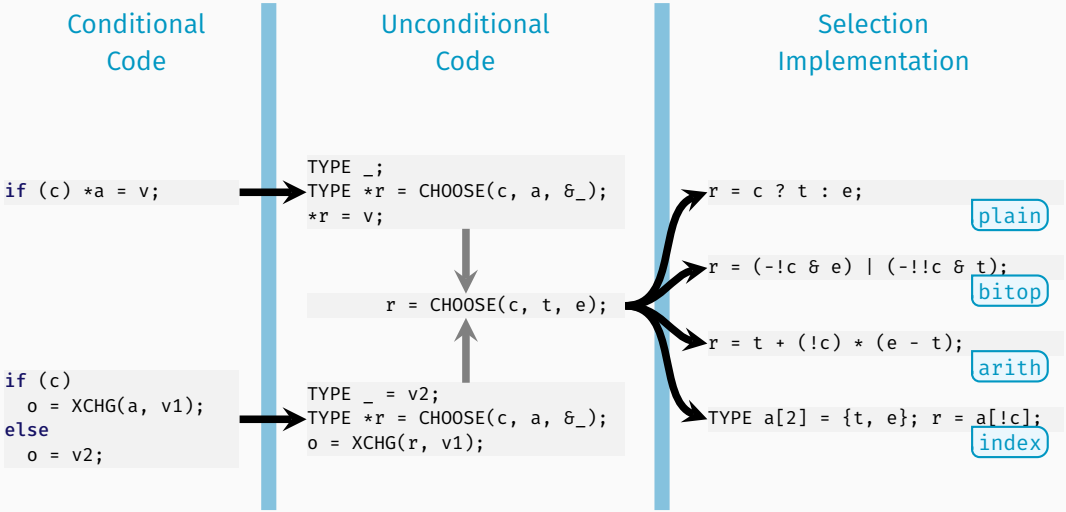
Code Transformation for IDOS | Conditional Statements



Code Transformation for IDOS | Conditional Statements



Code Transformation for IDOS | Conditional Statements



Code Transformation for IDOS | Example

```
void enq_if(bool cond,
            queue_t *list, chain_t *item)
{
  if (cond) {
    chain_t **addr = &list->tail;
    chain_t  *prev = XCHG(addr, item);
    prev->next = item;
  }
}
```

- Address calculation adapted to consider condition
- Unconditional execution of functional part
- Code more difficult to write & understand

Code Transformation for IDOS | Example

```
void enq_if(bool cond,
            queue_t *list, chain_t *item)
{
  if (cond) {
    chain_t **addr = &list->tail;
    chain_t  *prev = XCHG(addr, item);
    prev->next = item;
  }
}
```

```
void enq_if(bool cond,
            queue_t *list, chain_t *item)
{
  chain_t   temp = { .next = &temp };
  chain_t **addr = cond ? &list->tail
                        : &temp.next;
  chain_t  *prev = XCHG(addr, item);
  prev->next = item;
}
```

- Address calculation adapted to consider condition
- Unconditional execution of functional part
- Code more difficult to write & understand

Code Transformation for IDOS | Example

```
void enq_if(bool cond,
            queue_t *list, chain_t *item)
{
  if (cond) {
    chain_t **addr = &list->tail;
    chain_t  *prev = XCHG(addr, item);
    prev->next = item;
  }
}
```

```
void enq_if(bool cond,
            queue_t *list, chain_t *item)
{
  chain_t   temp = { .next = &temp };
  chain_t **addr = cond ? &list->tail
                        : &temp.next;
  chain_t  *prev = XCHG(addr, item);
  prev->next = item;
}
```

- Address calculation adapted to consider condition
- Unconditional execution of functional part
- Code more difficult to write & understand

Discussion

Discussion of DetOX

- Efficiency
  - Worst-case costs apply unconditionally
  - No overly pessimistic analysis needed
- Implementation issues
  - Device drivers, energy efficiency (Hardware-software interface)
  - Unimplemented features (Memory Protection, ...)

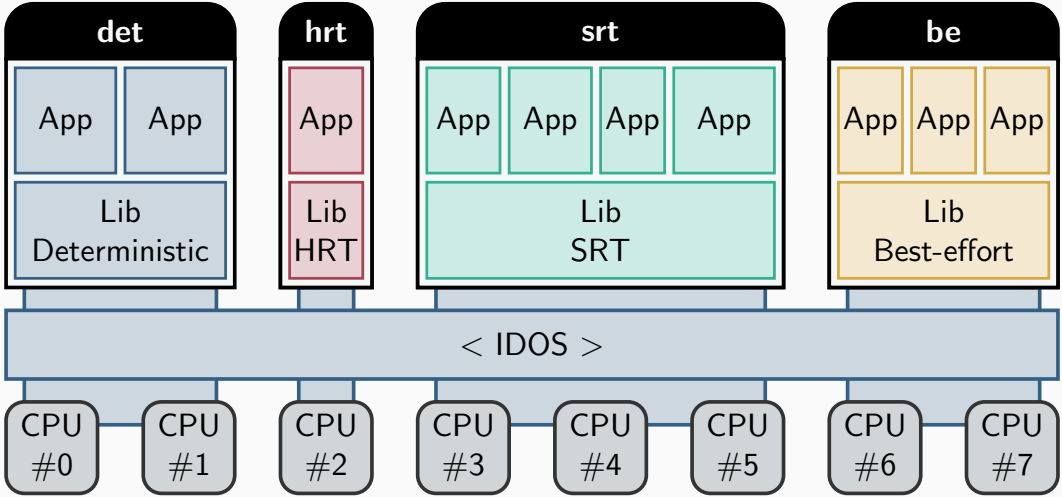
Discussion of DetOX

- Efficiency
  - Worst-case costs apply unconditionally
  - No overly pessimistic analysis needed
- Implementation issues
  - Device drivers, energy efficiency (Hardware-software interface)
  - Unimplemented features (Memory Protection, ...)
- Fundamental issues
  - Functional timing dependencies
    - Communication (e.g., `reply.get`)
    - Blocking operations (e.g., `thread.join`)
  - Approach is not universal

Discussion of DetOX

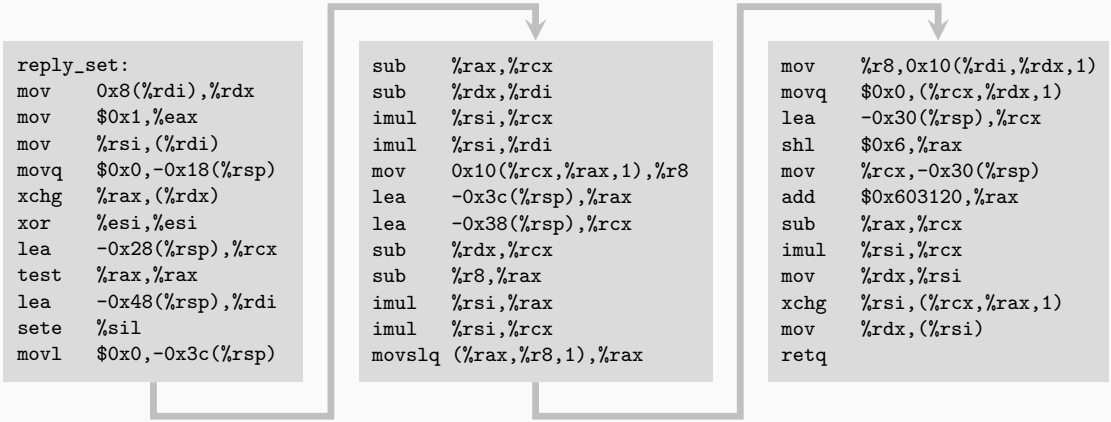
- Efficiency
  - Worst-case costs apply unconditionally
  - No overly pessimistic analysis needed
- Implementation issues
  - Device drivers, energy efficiency (Hardware-software interface)
  - Unimplemented features (Memory Protection, ...)
- Fundamental issues
  - Functional timing dependencies
    - Communication (e.g., `reply.get`)
    - Blocking operations (e.g., `thread.join`)
  - Approach is not universal
- Time to say goodbye to...
  - ...interrupts, traps & preemption
  - ...non-$\mathcal{O}(1)$ scheduling strategies
  - ...

Co-existence of Nondeterministic Subsystems



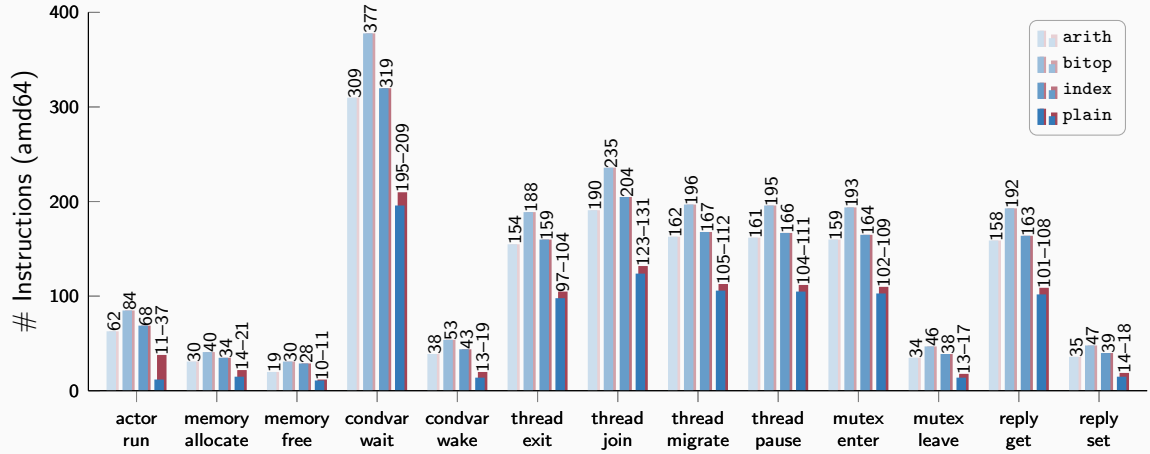
Evaluation

Control-Flow Graphs



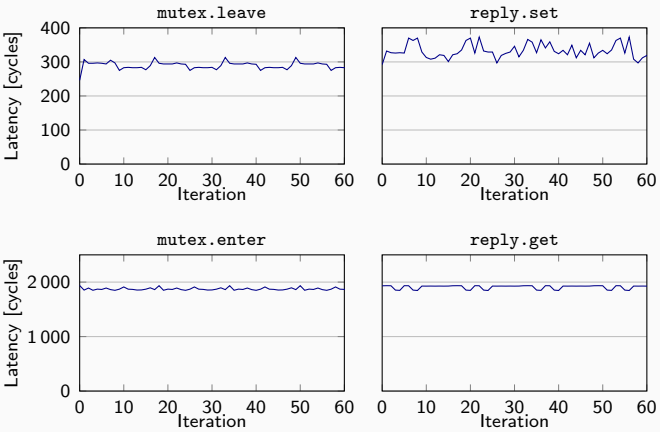
⇒ No branches, no control-flow dependency on system state or size

Control-Flow Analysis



⇒ Software-level analysis is almost trivial, only in-hardware timing variance remains

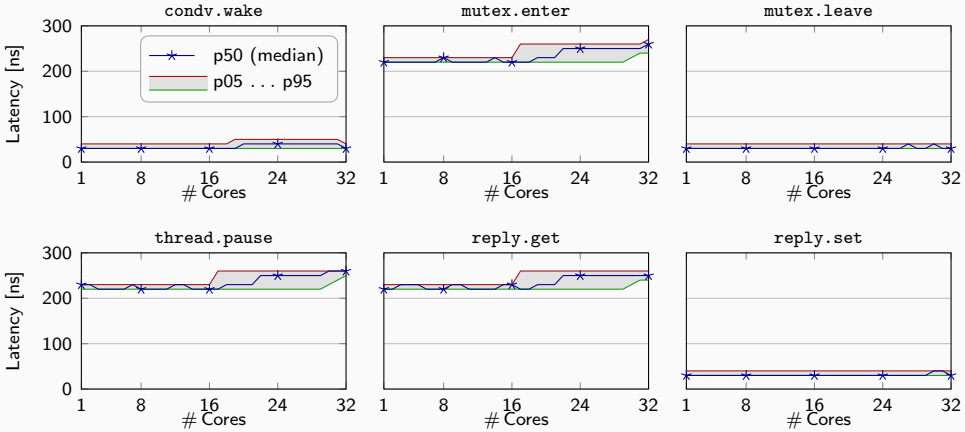
Hardware Predictability



- Cycle-accurate RISC-V emulator
- Rocket-chip configuration
  - 2 “small” cores
  - L1D$.nsets = 2
  - L1D$.nways = 1
  - L2D$.nsets = 2
  - L2D$.nways = 4
  - L2D$.replacement: TrueLRU
  - Bus arbitration: LowestIndexFirst
  - Cycle measurement at core # 0
  - Measurement overhead = 1 cy
- Calls with debug interrupt ignored

⇒ Some observable patterns in latency variation, some noise

Scalability



⇒ Latency independent of parallelism, except hyper-threading (amd64)

Conclusion

Summary & Conclusion

- Need for scalable, predictable, and analysable systems
  - Main issue: transitive interferences
  - System structure: interacting subsystems
  - Focus on predictable communication
- Approach: inherently deterministic operating systems (IDOSs)
  - ✓ Scalability: $\mathcal{O}(1)$
  - ✓ Analysability: almost trivial, no overestimation, incremental
  - ✓ Predictability: constant costs
  - ✗ ...above ISA-level

Summary & Conclusion

- Need for scalable, predictable, and analysable systems
  - Main issue: transitive interferences
  - System structure: interacting subsystems
  - Focus on predictable communication
- Approach: inherently deterministic operating systems (IDOSs)
  - ✓ Scalability: $\mathcal{O}(1)$
  - ✓ Analysability: almost trivial, no overestimation, incremental
  - ✓ Predictability: constant costs
  - ✗ ...above ISA-level
- Co-existence with non-deterministic subsystems
  - Various means for coordination
  - Constraints for functional timing dependencies
- Determinism still has some open issues
  - ✗ constant-time hardware
  - ✗ Fundamental limitations (e.g., interrupts)
  - ? Hardware-software interfaces
  - ? Suitable $\mathcal{O}(1)$ strategies (e.g., scheduler)
  - ? Concurrent $\mathcal{O}(1)$ algorithms