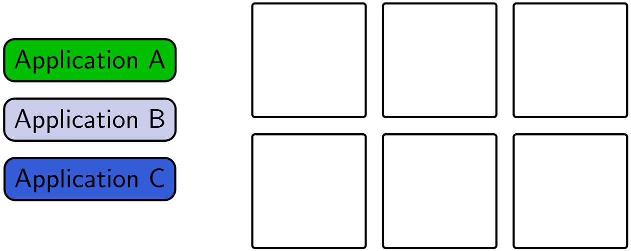
OctoPOS: An Operating System for Invasive Computing

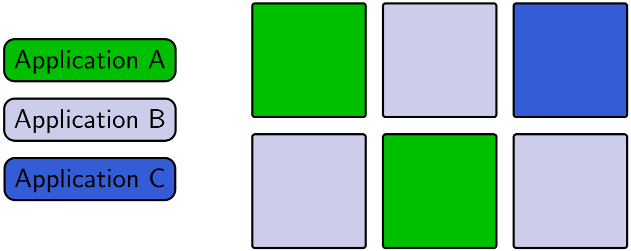
Benjamin Oechslein

8. November 2013

Traditional Application Model

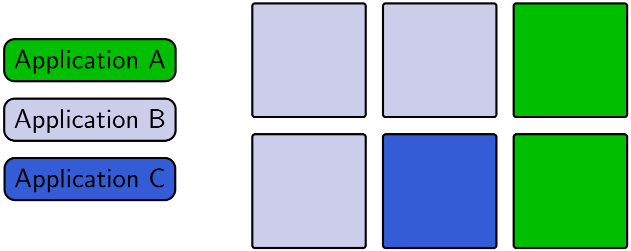


Traditional Application Model



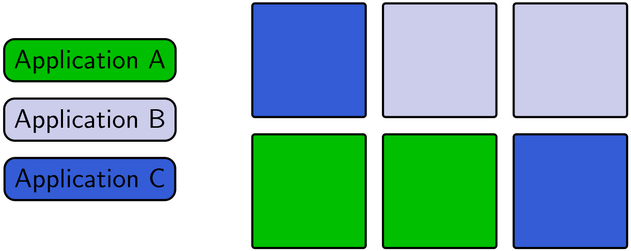
- Multiplexing of CPU cores

Traditional Application Model



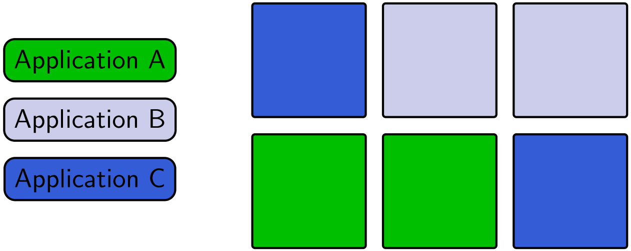
- Multiplexing of CPU cores

Traditional Application Model



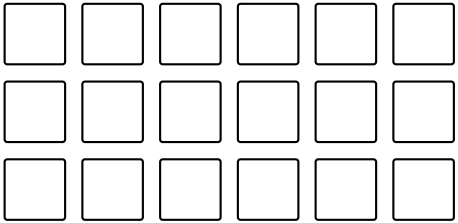
- Multiplexing of CPU cores
- Applications are unaware of this

Traditional Application Model

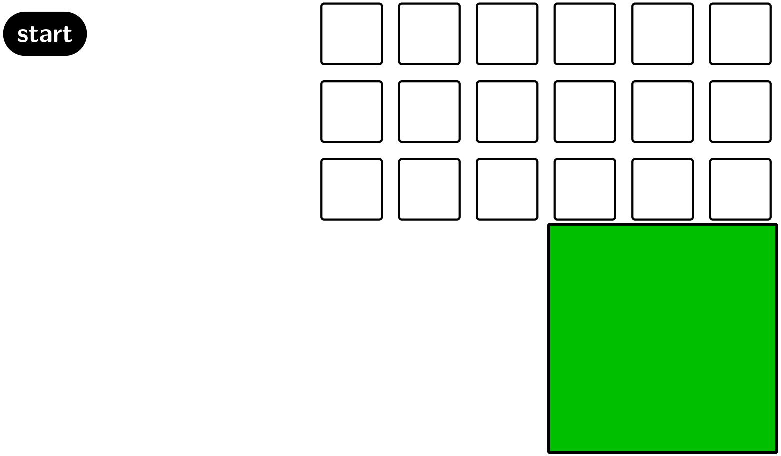


- Multiplexing of CPU cores
- Applications are unaware of this
- Can we do better?

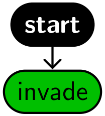
Invasive Computing: Application Model

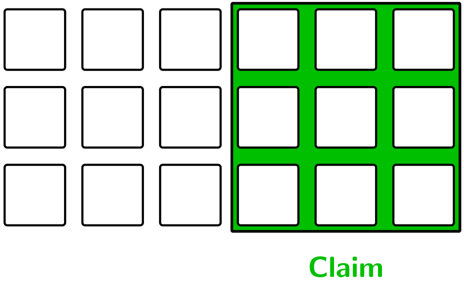


Invasive Computing: Application Model

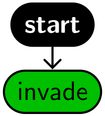


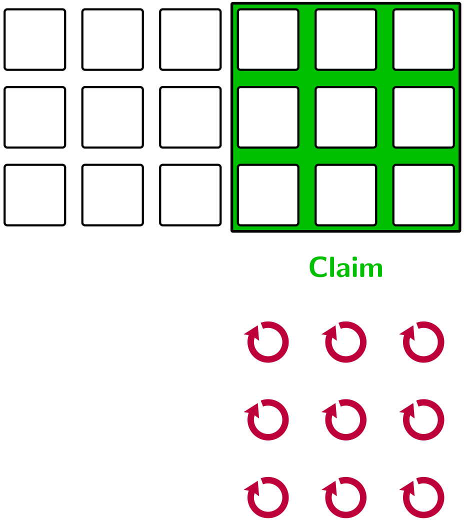
Invasive Computing: Application Model



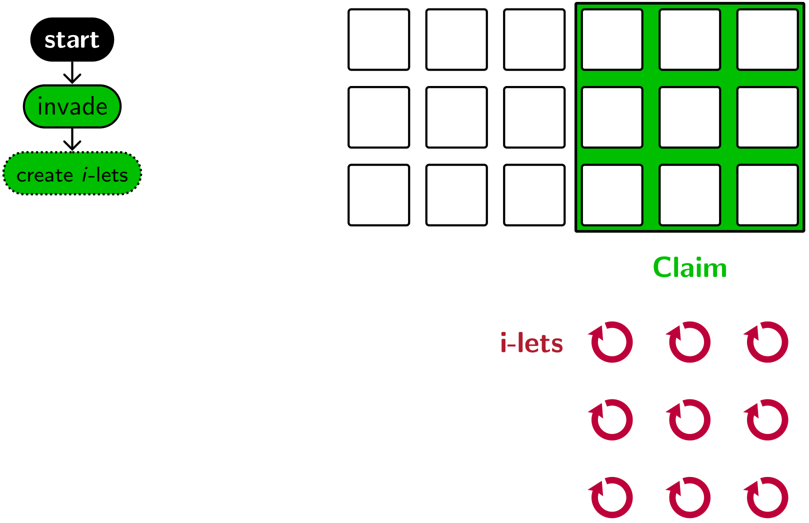


Invasive Computing: Application Model

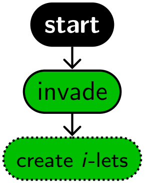


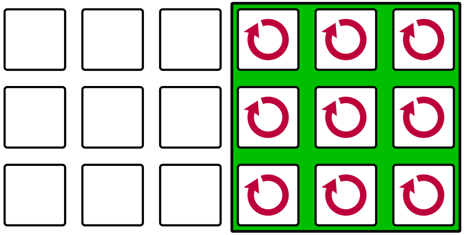


Invasive Computing: Application Model

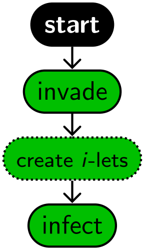


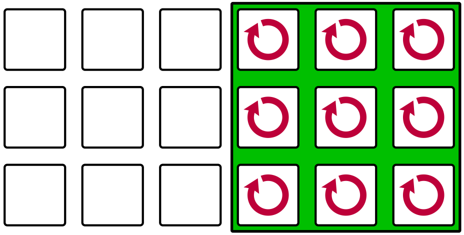
Invasive Computing: Application Model



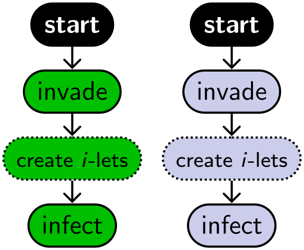


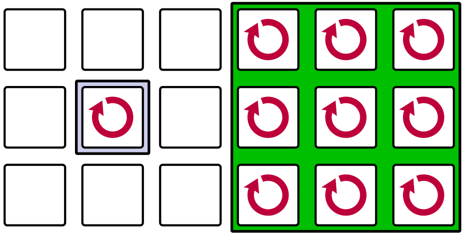
Invasive Computing: Application Model



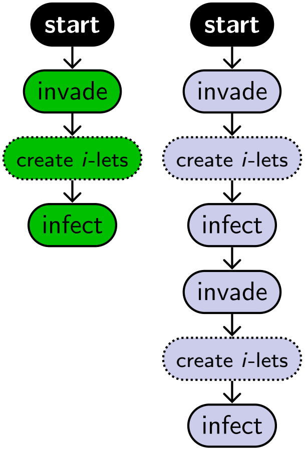


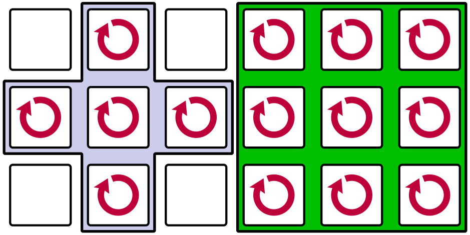
Invasive Computing: Application Model



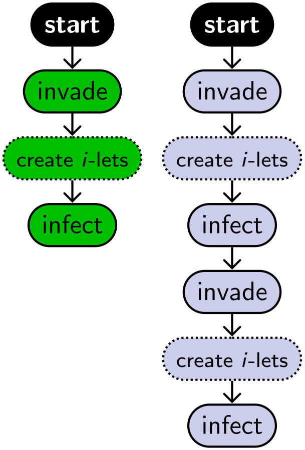


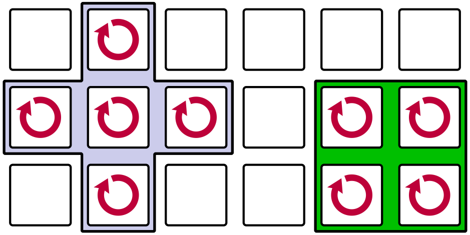
Invasive Computing: Application Model



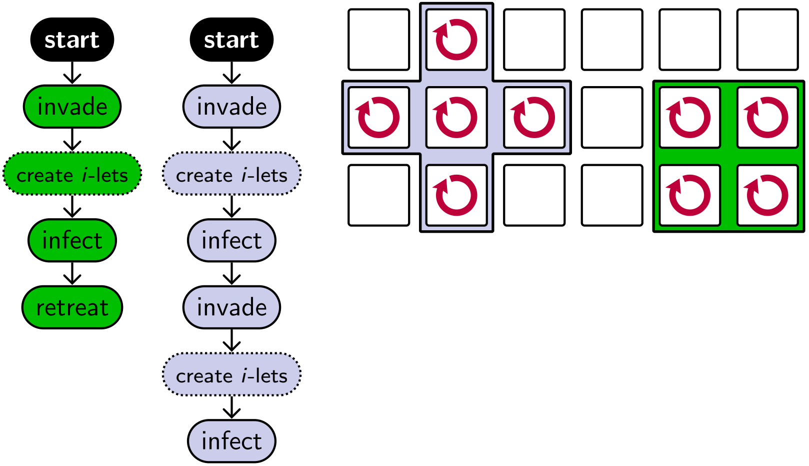


Invasive Computing: Application Model

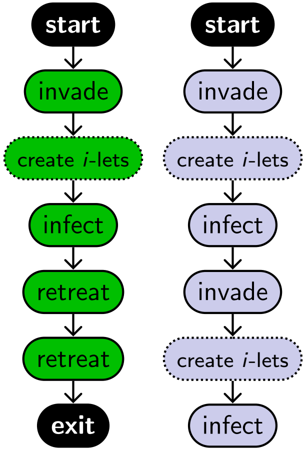


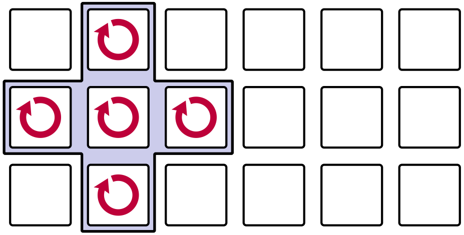


Invasive Computing: Application Model



Invasive Computing: Application Model





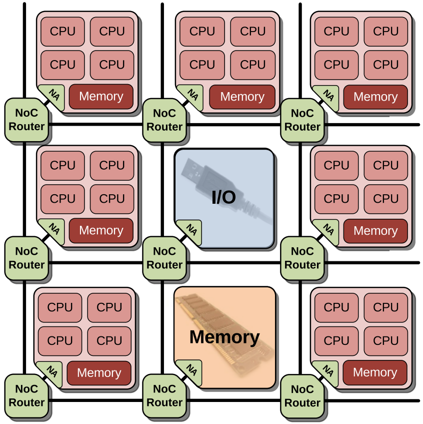
Invasive Computing

- Collaborative Research Center/Transregio
- Research on all levels
  - Application
    - Resource-aware algorithms: Robotics and HPC
    - Dynamic adaptation of algorithms to varying amount of resources
  - Compiler/runtime system
    - Abstract description of resource demand
    - Distribution of large, parallel systems between applications
  - Operating system
    - Support for novel application model: explicit hardware allocation
    - Lightweight and scalable execution model for applications
  - Hardware
    - Resource-aware hardware components
    - ”Allocatable” hardware

Hardware Model

- Tiled many-core architecture
- Limited cache coherence

  ⇒ Confined to the cores of a tile
- Similar to distributed systems



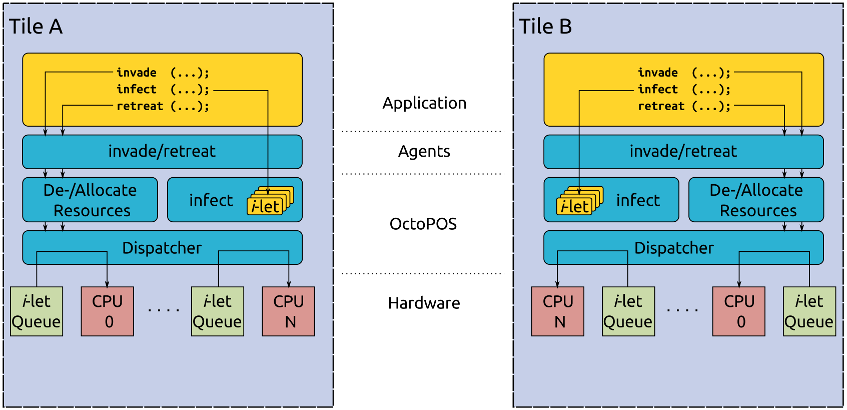
Requirements for the Operating System

- Novel mechanism supporting resource-aware programming
  - Allocate parts of the machine (invade/retreat)
  - Guarantee access to allocated resources
- Scalability
  - Support for highly parallel applications
  - Support fine-grained parallelism
  - Support for highly parallel many-core systems
- Adapt to non-uniform hardware architecture
  - Operation without cache coherency
  - Exploit hardware architecture for efficiency

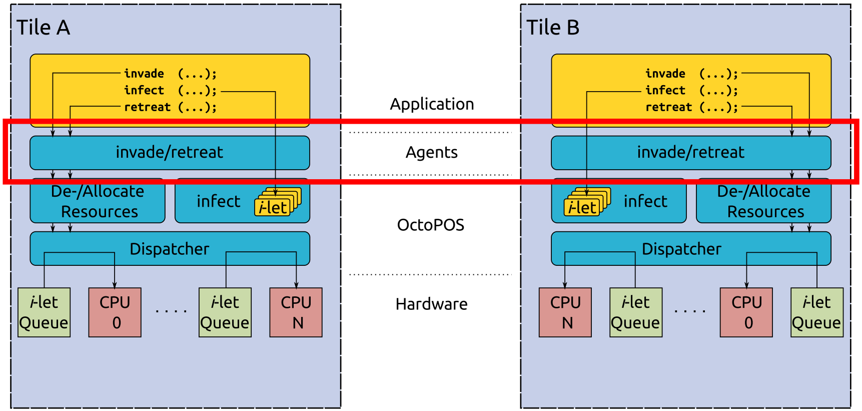
Agenda

1. Motivation
2. Design
   - Architectural overview
   - Mechanisms
   - Scalability aspects
3. Preliminary results
4. Conclusion & Outlook

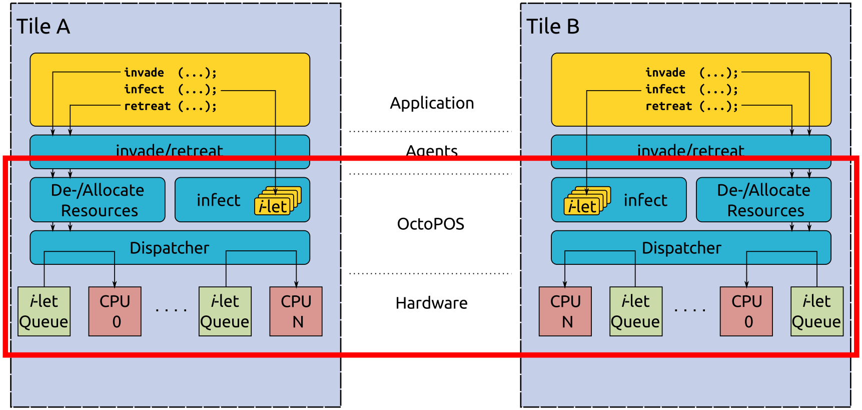
OctoPOS Architecture



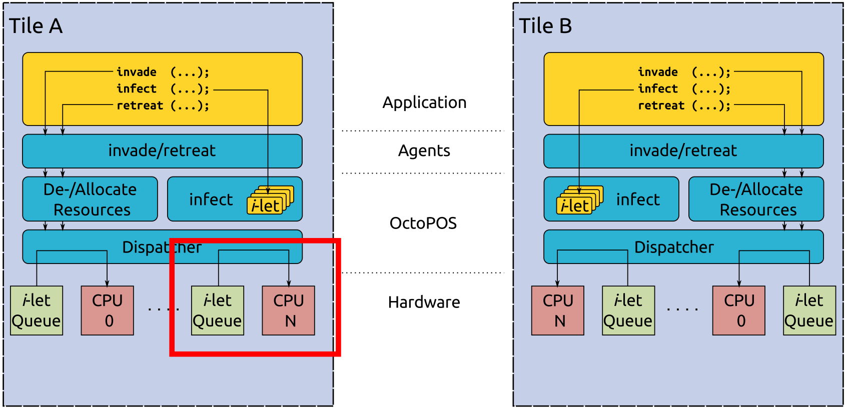
OctoPOS Architecture



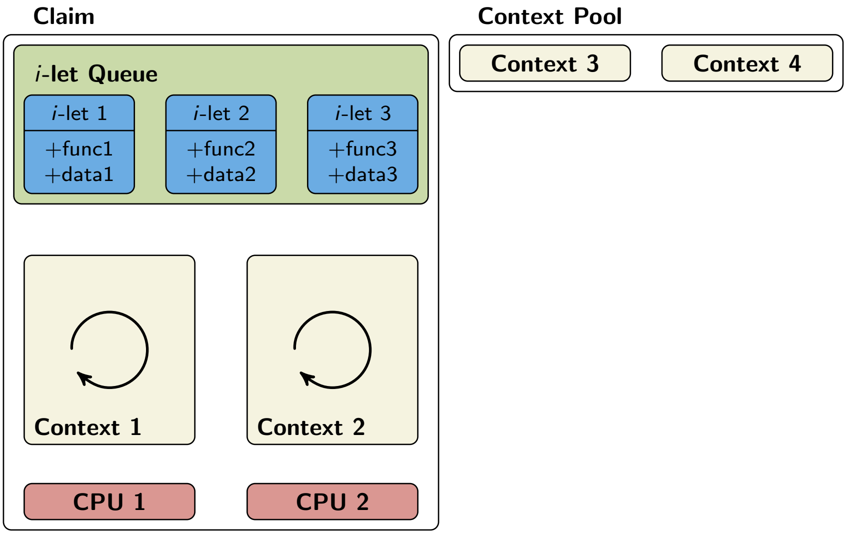
OctoPOS Architecture



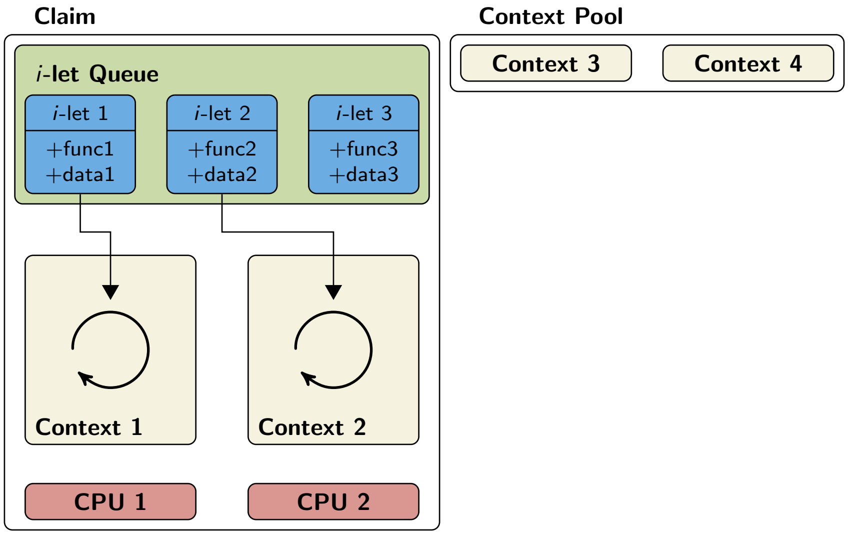
OctoPOS Architecture



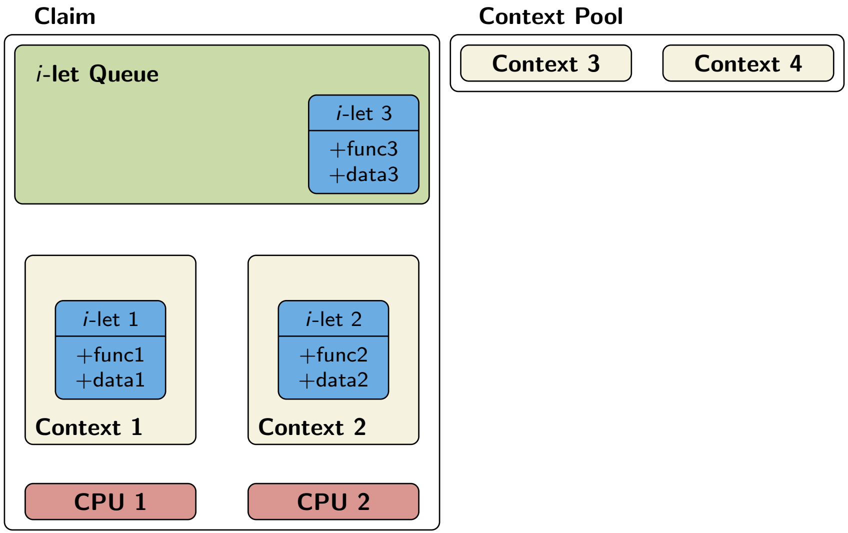
OS Architecture: Local Execution Model



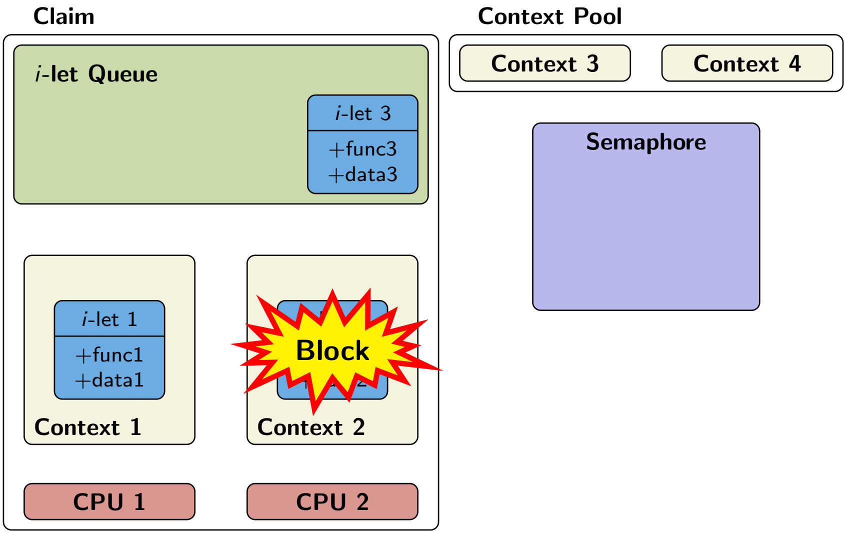
OS Architecture: Local Execution Model



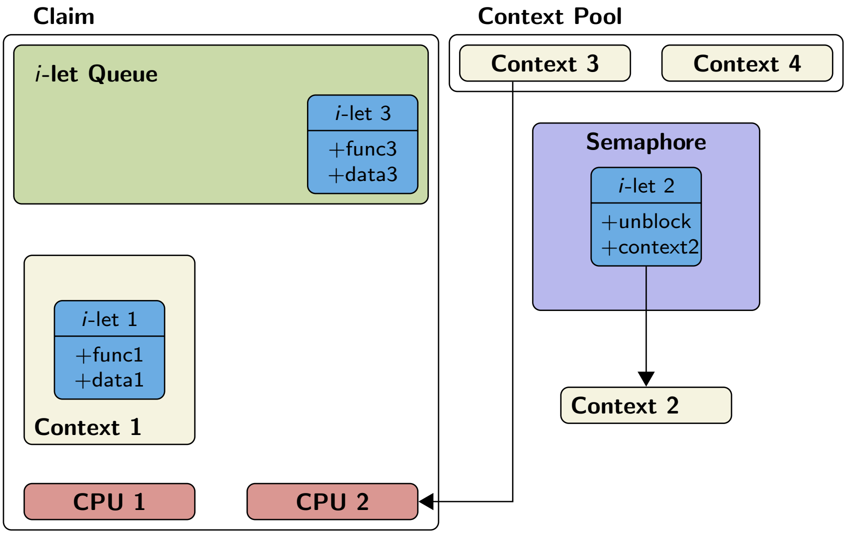
OS Architecture: Local Execution Model



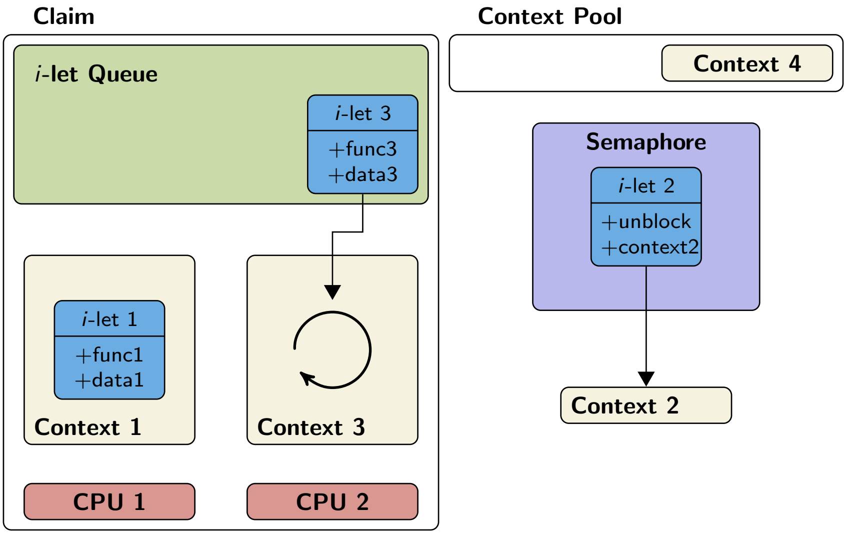
OS Architecture: Local Execution Model



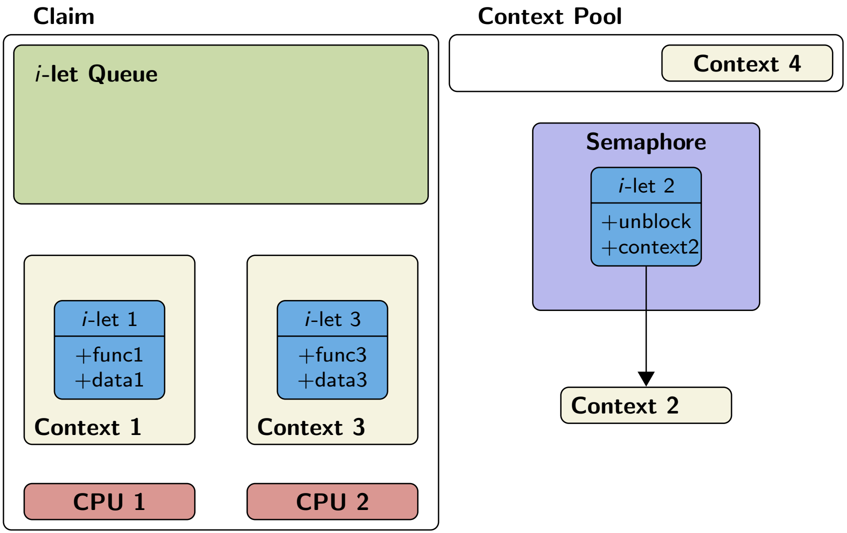
OS Architecture: Local Execution Model



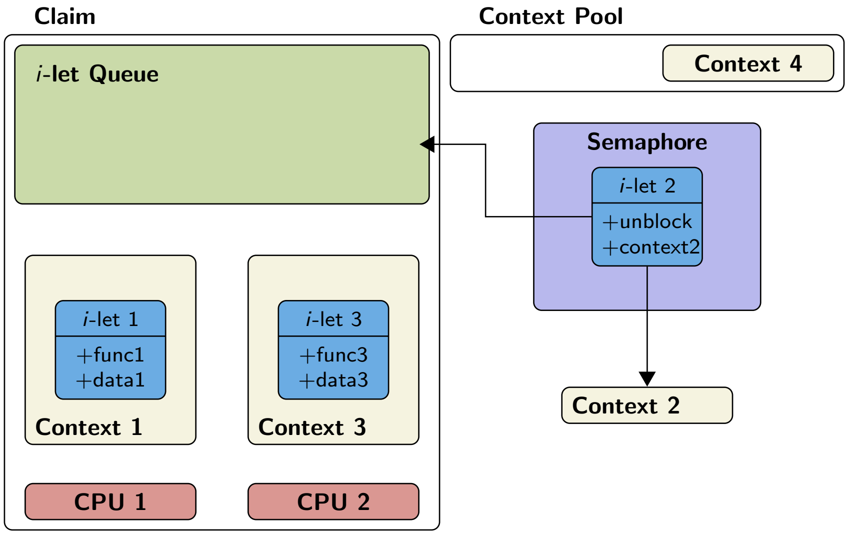
OS Architecture: Local Execution Model



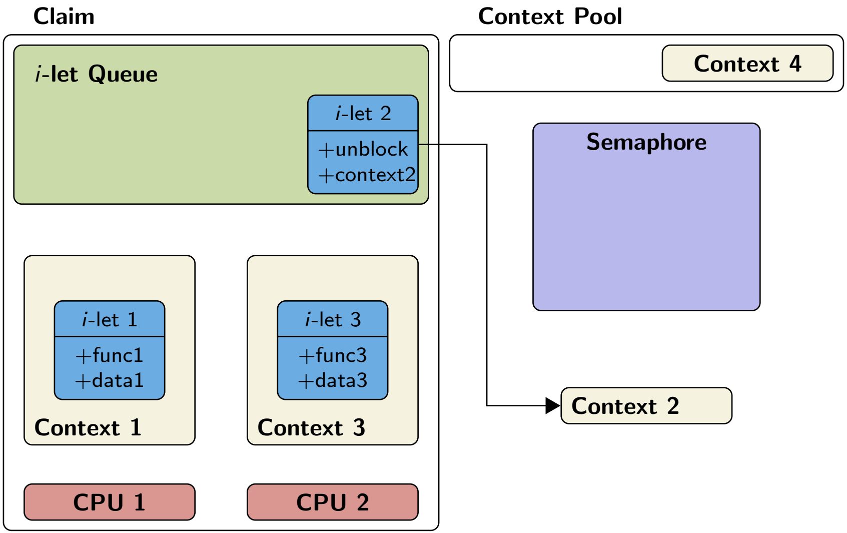
OS Architecture: Local Execution Model



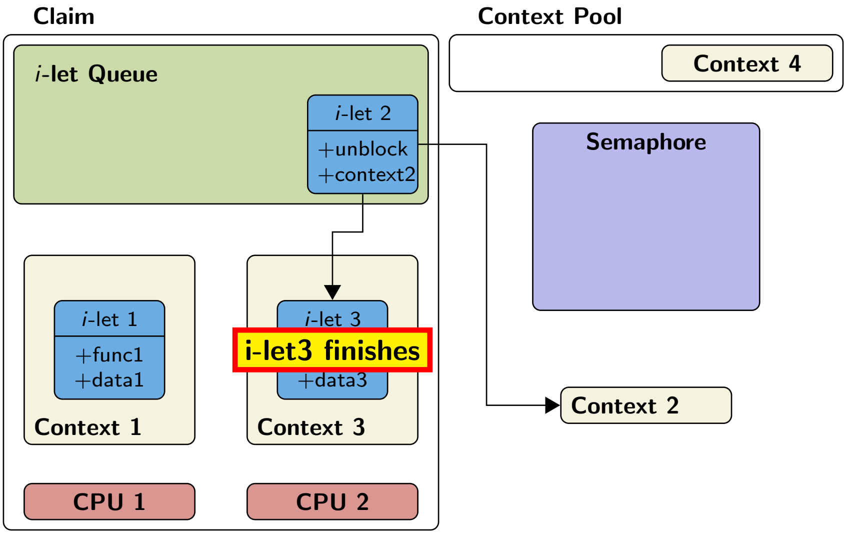
OS Architecture: Local Execution Model



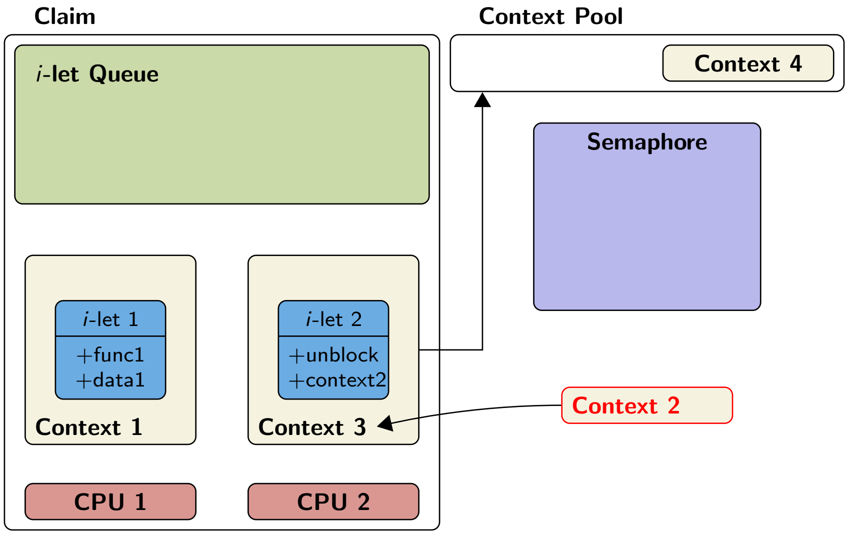
OS Architecture: Local Execution Model



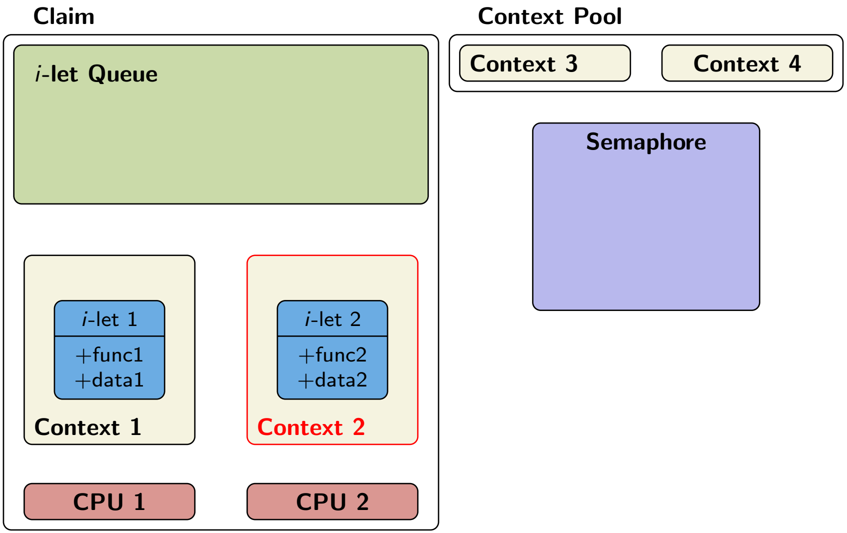
OS Architecture: Local Execution Model



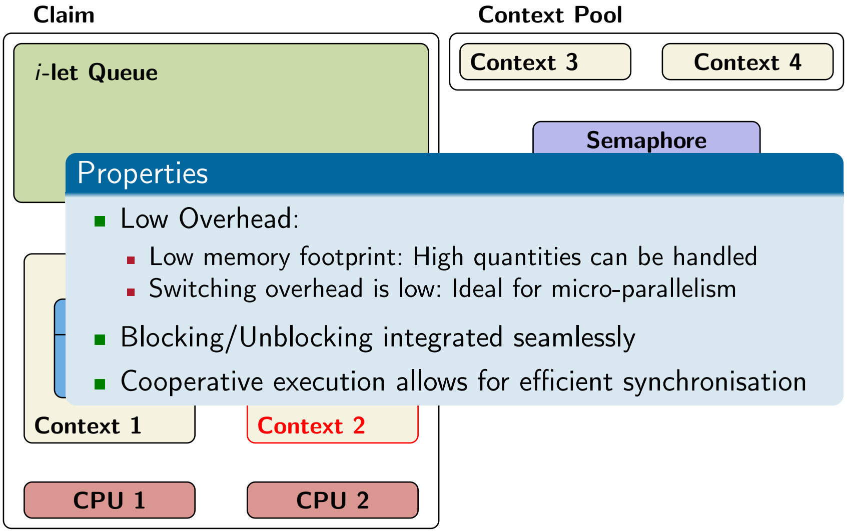
OS Architecture: Local Execution Model



OS Architecture: Local Execution Model



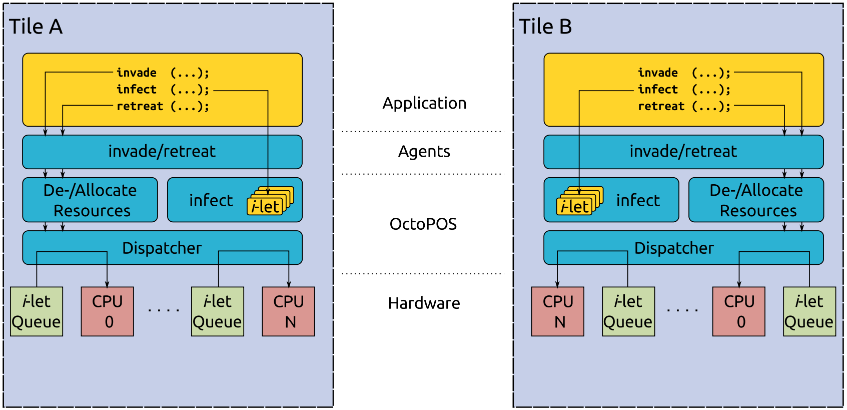
OS Architecture: Local Execution Model



Properties

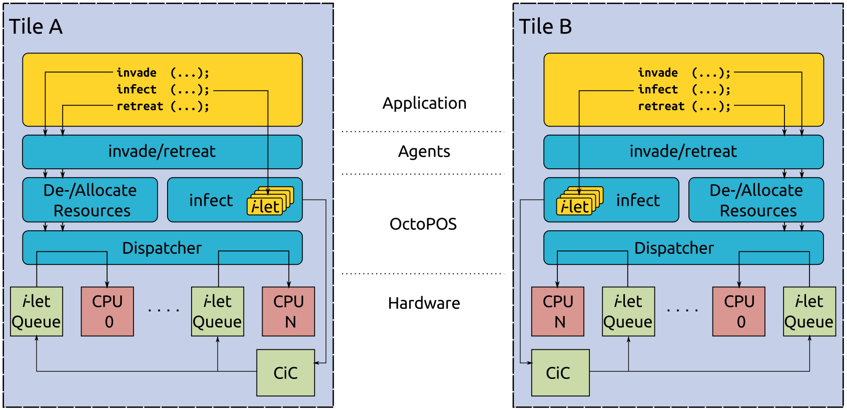
- Low Overhead:
  - Low memory footprint: High quantities can be handled
  - Switching overhead is low: Ideal for micro-parallelism
- Blocking/Unblocking integrated seamlessly
- Cooperative execution allows for efficient synchronisation

Scalability



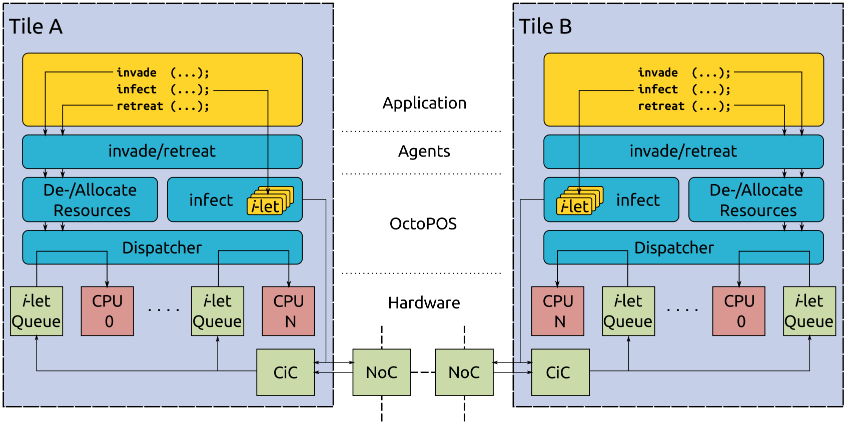
- Multi-tile execution model
  - Transfer *i*-lets
  - Allocate remote resources

Scalability



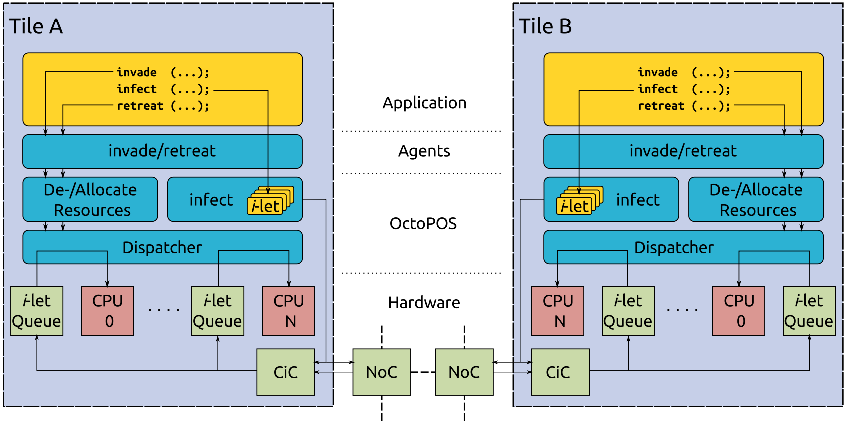
- Multi-tile execution model
  - Transfer *i*-lets
  - Allocate remote resources

Scalability



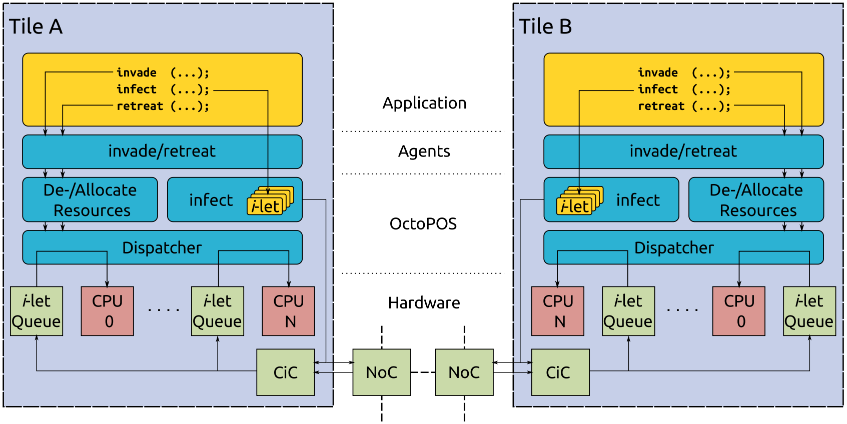
- Multi-tile execution model
  - Transfer *i*-lets
  - Allocate remote resources

Scalability



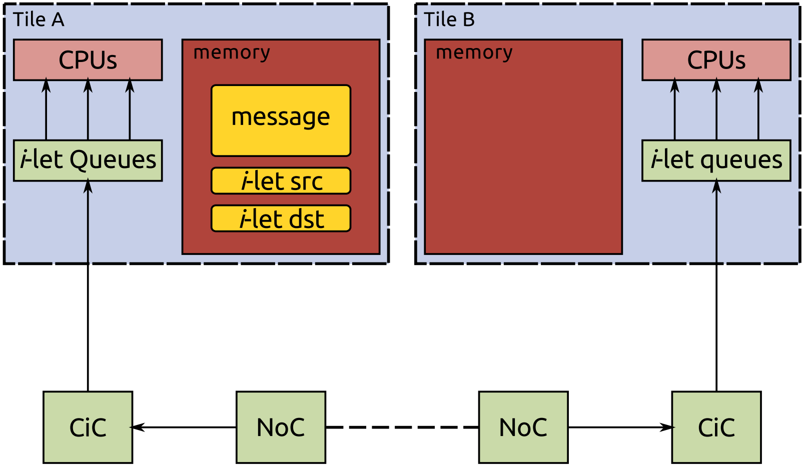
- HW/SW-Codesign
  - Increase scalability for common operations
  - Transfer *i*-lets directly in hardware

Scalability

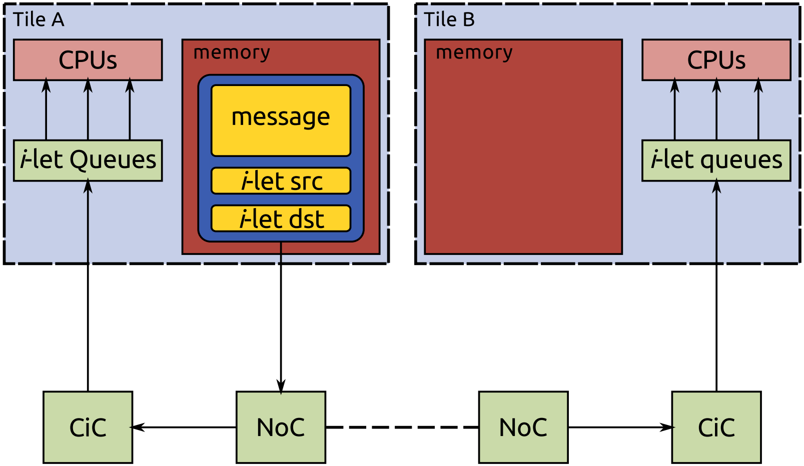


- Data transfer between cache-coherency domains
  - Necessary for applications running in multiple cache-coherency domains
  - Notification mechanism between data transfer and application

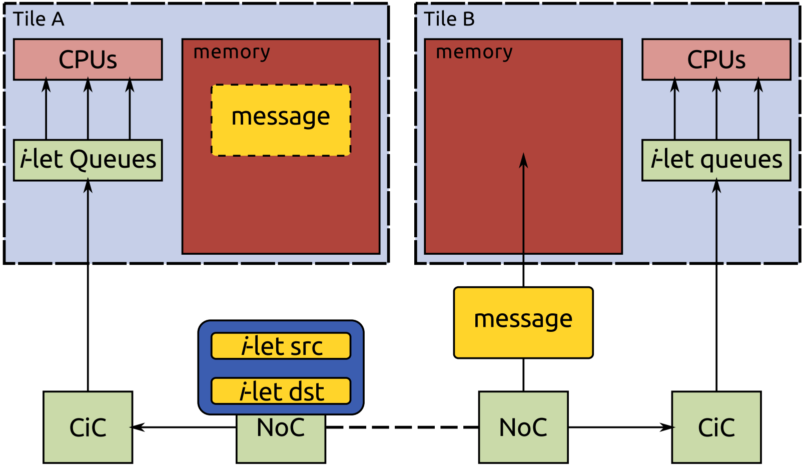
Codesign between HW and SW: Data transfer



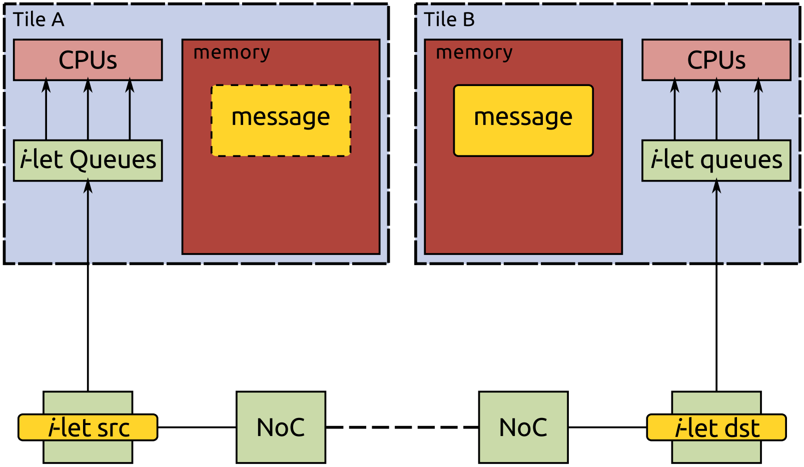
Codesign between HW and SW: Data transfer



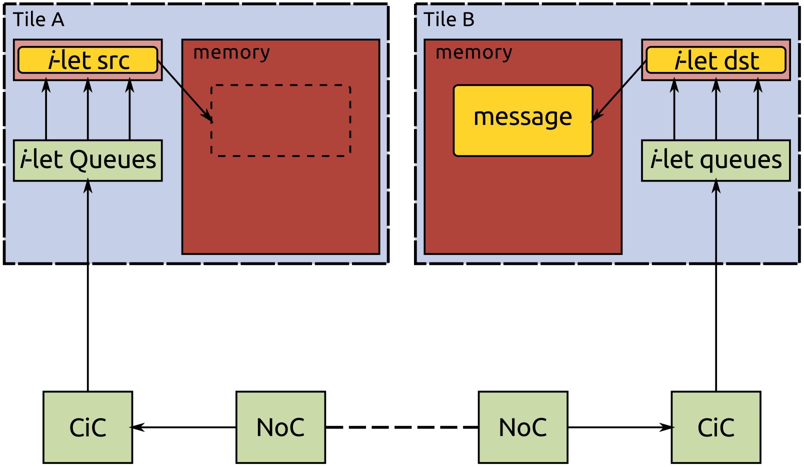
Codesign between HW and SW: Data transfer



Codesign between HW and SW: Data transfer



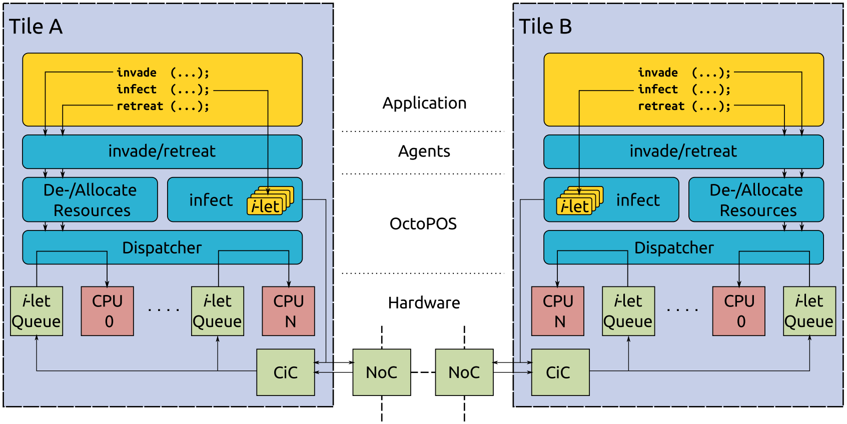
Codesign between HW and SW: Data transfer



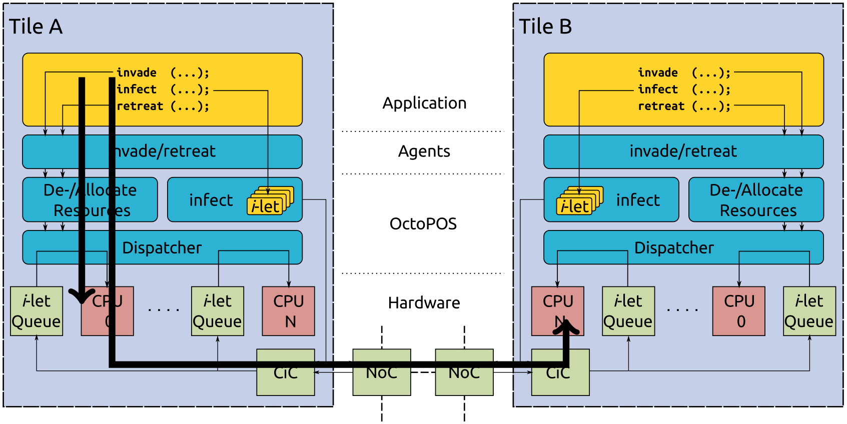
Preliminary Results

- Evaluation on an FPGA-based prototype with SPARC LEON CPUs

Preliminary Results

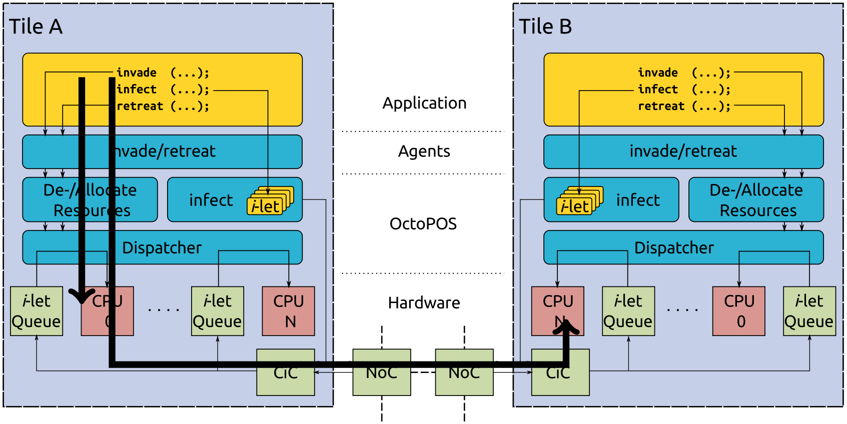


Preliminary Results



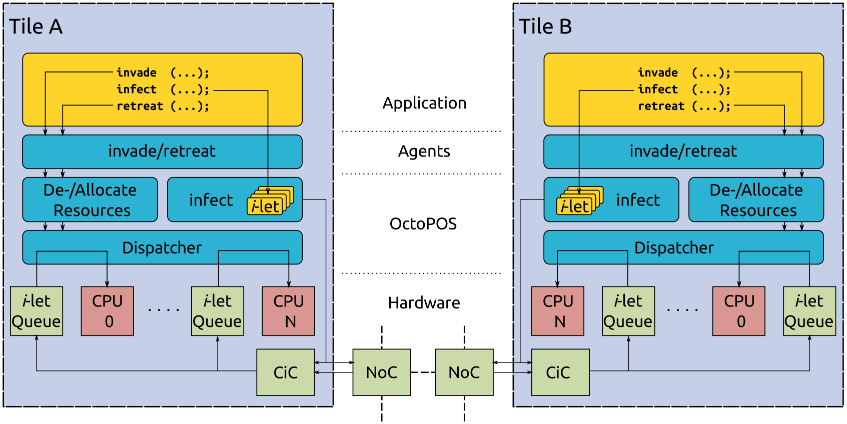
Local Invade:	355 cycles
Remote Invade:	2162 cycles

Preliminary Results



Local Invade:	355 cycles	Local Infect:	86 cycles
Remote Invade:	2162 cycles	Remote Infect:	99 cycles

Preliminary Results



Local Invade:	355 cycles	Local Infect:	86 cycles
Remote Invade:	2162 cycles	Remote Infect:	99 cycles
Local Infect Latency:	168 cycles		
Remote Infect Latency:	255 cycles		

Conclusion & Future Work

- Fast and scalable execution model
  - Provides lightweight primitives for executing *i*-lets
  - Can cope with large, parallel applications
- Codesign between hardware and system software
  - Helps scalability
  - Avoids unnecessary overhead for common operations
  - Efficiently supports large systems by not relying on cache coherency

Conclusion & Future Work

- Fast and scalable execution model
  - Provides lightweight primitives for executing *i*-lets
  - Can cope with large, parallel applications
- Codesign between hardware and system software
  - Helps scalability
  - Avoids unnecessary overhead for common operations
  - Efficiently supports large systems by not relying on cache coherency
- Custom-built hardware entails lots of engineering to get it running

Conclusion & Future Work

- Fast and scalable execution model
  - Provides lightweight primitives for executing *i*-lets
  - Can cope with large, parallel applications
- Codesign between hardware and system software
  - Helps scalability
  - Avoids unnecessary overhead for common operations
  - Efficiently supports large systems by not relying on cache coherency
- Custom-built hardware entails lots of engineering to get it running
- Future Work
  - Port execution model to standard hardware
  - x86/Xeon Phi
  - Protection