Invasive Computing: A Systems-Programming Perspective

Wolfgang Schröder-Preikschat

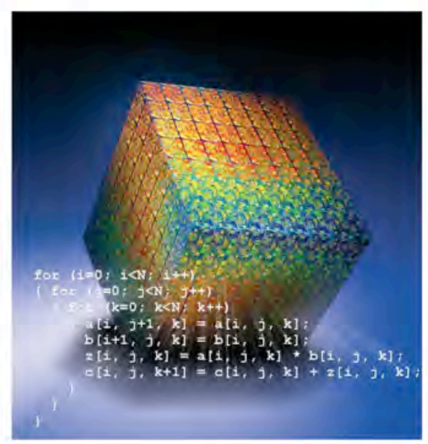
Embedded Systems Institute
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Collaborative Research Centre/Transregio

*Distributed Systems & Operating Systems
Hardware-Software-Codesign

12 chairs
+60 scientists
22.5 funded thereof
term: 3 x 4 years
phase 1: €9m
started Q3/2010

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG



KIT Karlsruhe Institute of Technology

TRR 89

TUM

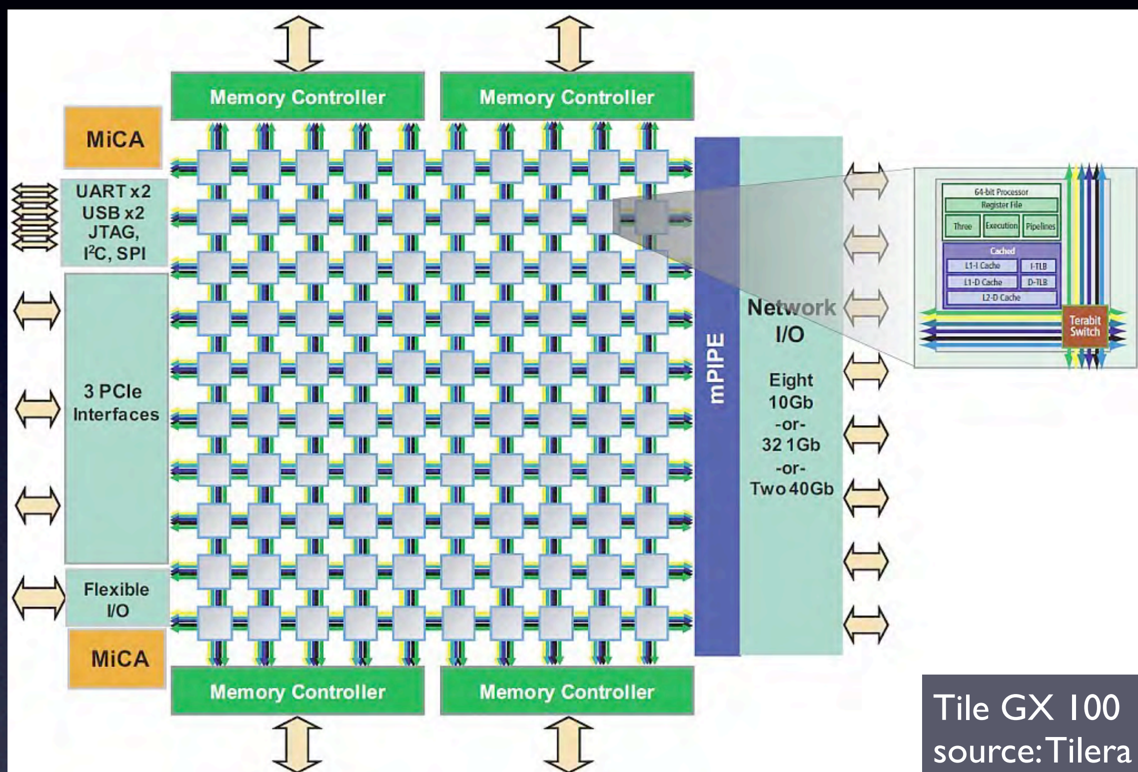
*** System software:**
5 scientists (full-time equiv.)
1 system programmer
4 funded thereof

Embedded Systems*
Theoretical Informatics
Programming Paradigms
Information Processing Technologies
Humanoids and Intelligent Systems

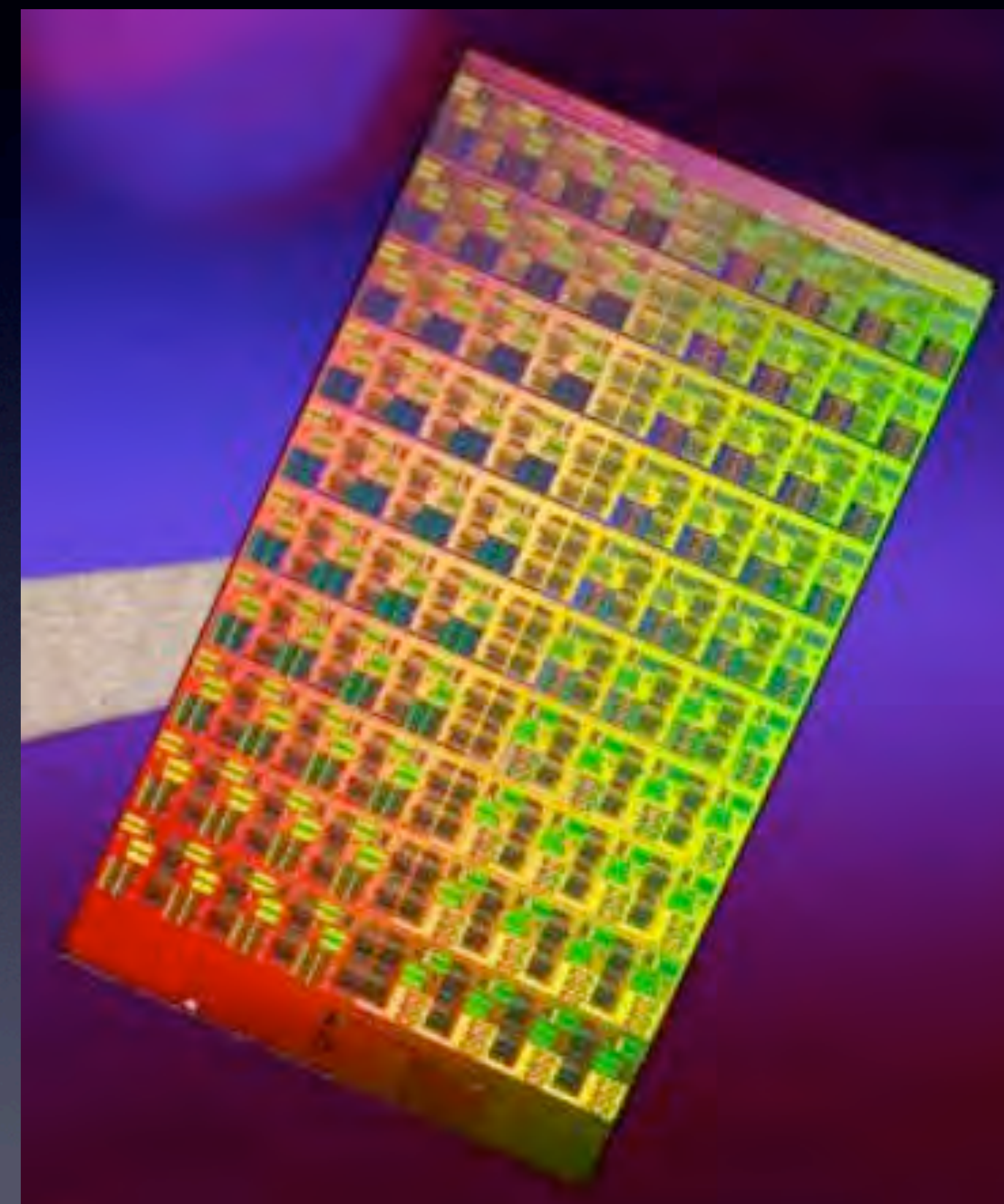
Scientific Computing
Integrated Systems
Technical Electronics
Electronic Design Automation
Computer Engineering & Organization

Parallel Processing

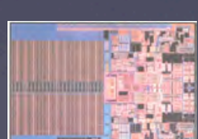

Parallel Systems



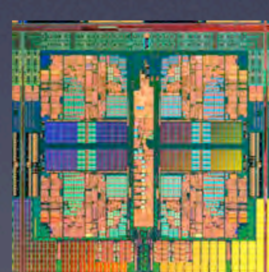
Tile GX 100
source: Tilera



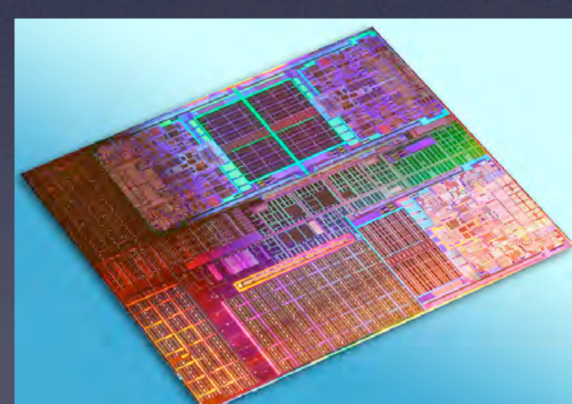
80



2

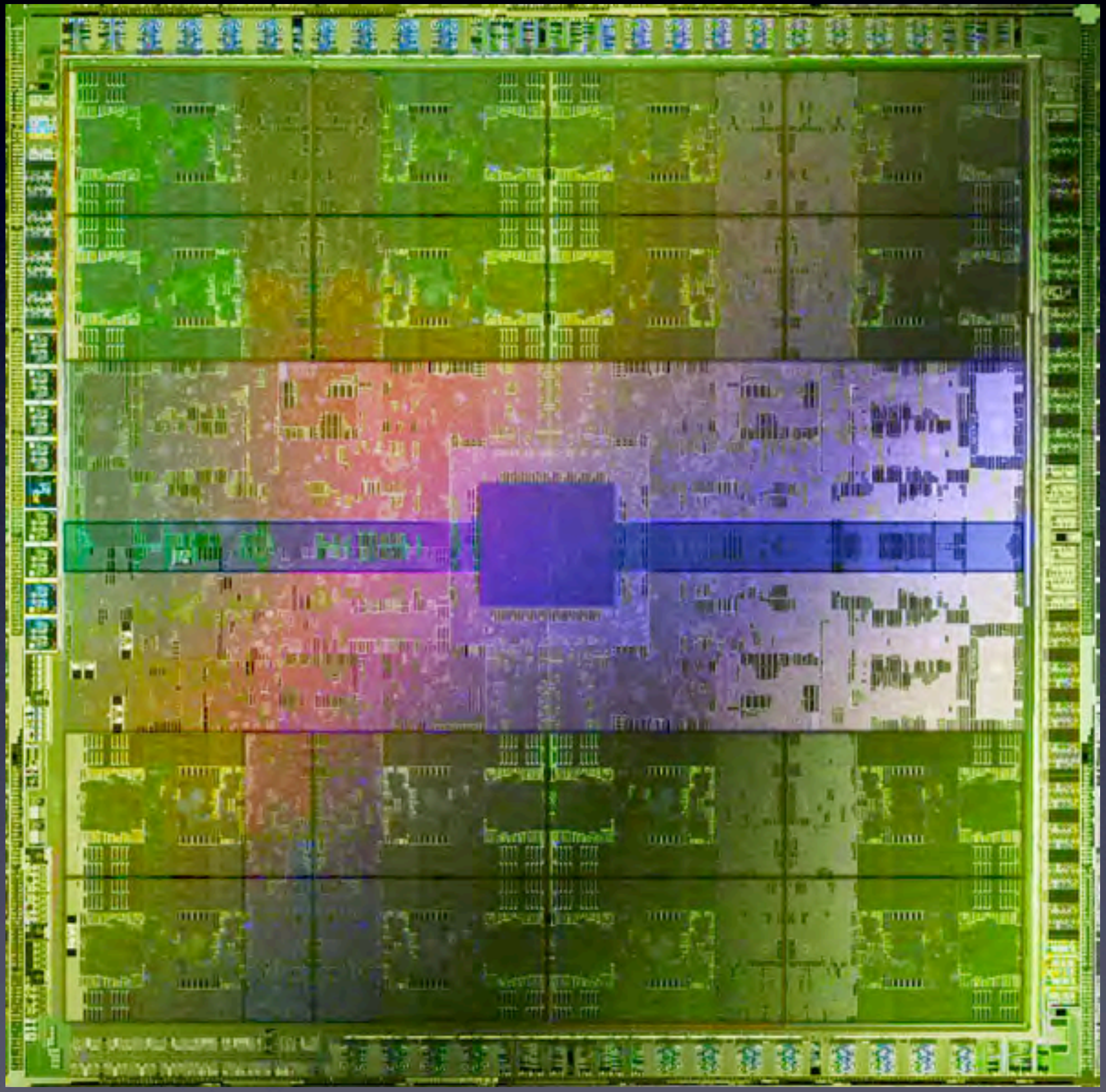


4

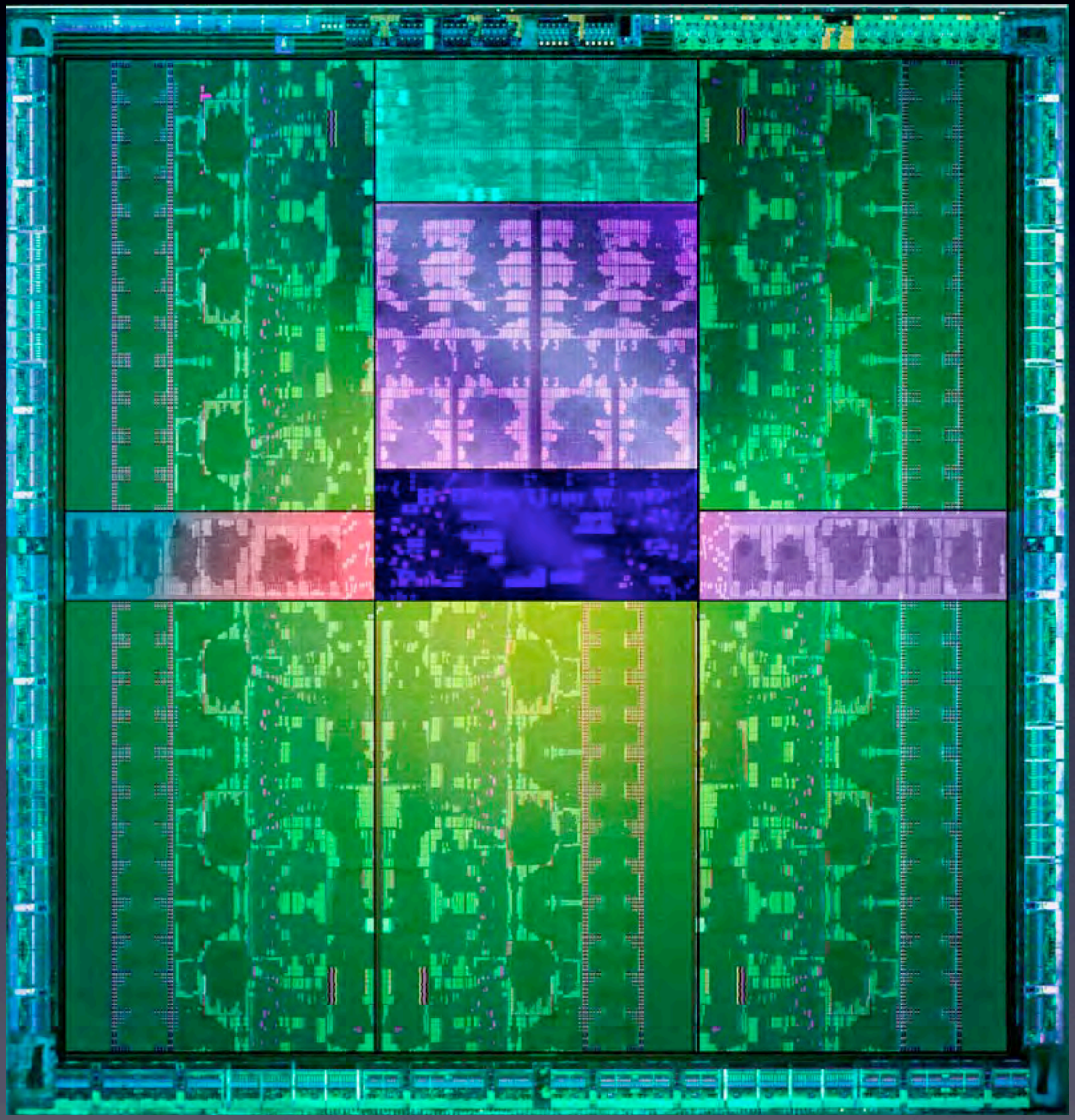


8

Parallel Systems(cont.)



512

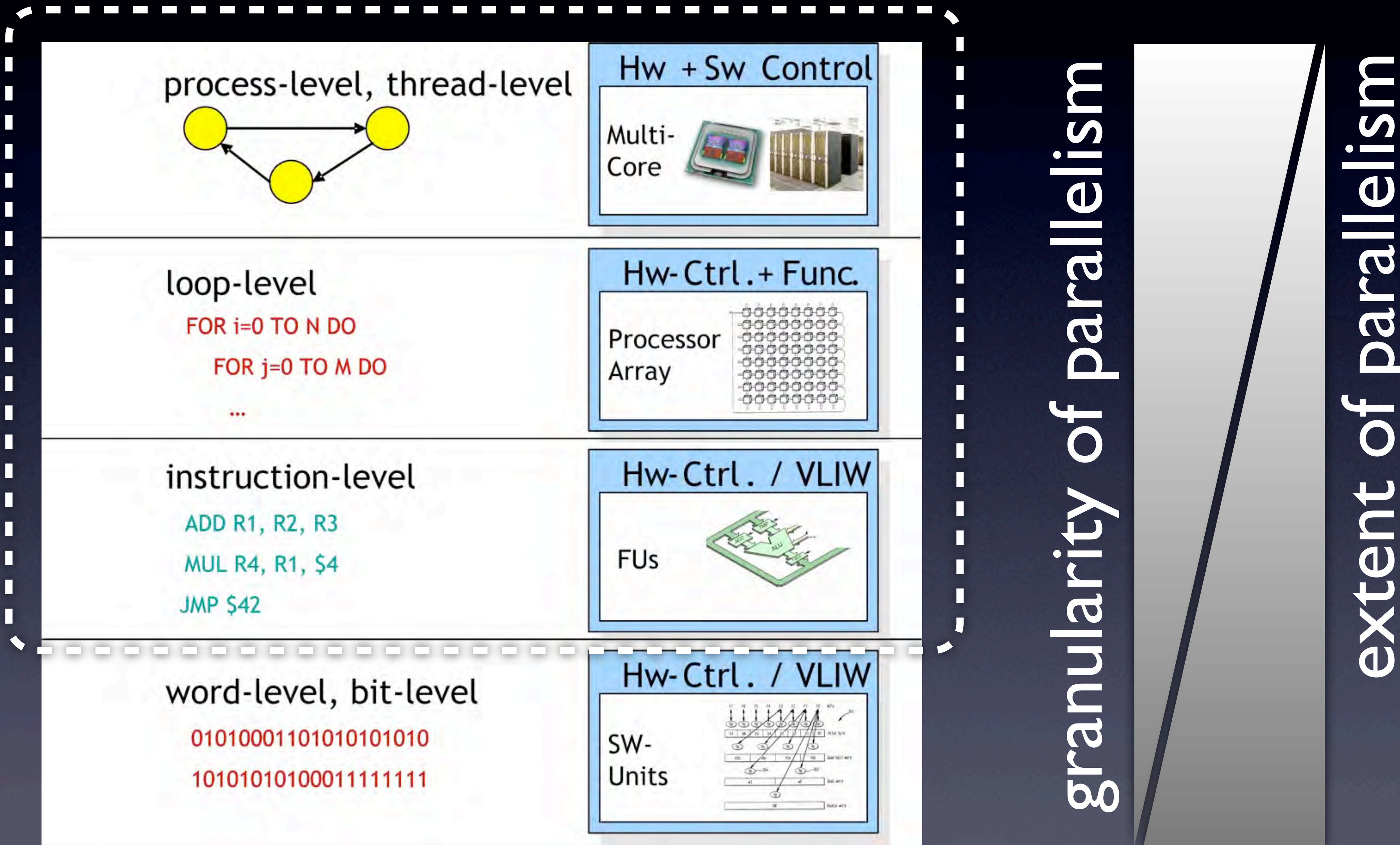


1536

Parallel Systems(cont.)

1572864

Levels of Parallelism



Invasive Computing

Acronym: InvasIC

Investigation of a new paradigm of parallel computing

- *through introduction of resource-aware language, programming and operating-system support*
- *and through dynamic and distributed allocation and reconfiguration of processor, interconnect, and memory resources*
- *with particular focus on Multiprocessor System-on-Chip (SoC) systems for the years 2020 and beyond.*

Resource-Aware

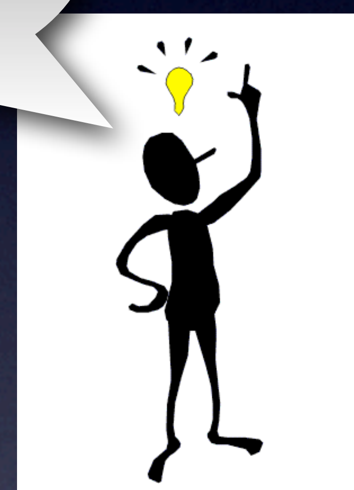
(Trinity of requisition, usage, and restoration)

1. invade
   - request/reserve computing resources
2. infect
   - deal with the resources received
3. retreat
   - release the resources received

Units of Invasion

„invasive-let“
i-let

Program section being aware of potential parallel execution



1. candidate
2. instance
3. incarnation
4. execution

operating-system entities { 3. incarnation, 4. execution

System Abstractions

1. claim
   - of (hardware/software) resources
   - for the execution of *i-let* incarnations
2. team
   - of (related/dependent) *i-let* instances
   - for the coordination of invasive processes

Claim Attributes (excerpt)

- coherent/incoherent
- heterogeneous/homogeneous
- preemptive/nonpreemptive
  - temporal, spatial
  - interceptive
- interactive/passive

} *operation mode*

Team Attributes (excerpt)

scheduling & dispatching

- sequential/nonsequential
- coordinated/uncoordinated
- dependent/independent
- clocked/freewheeling
- run to completion/surrender

Principle of Operation

```
claim = invade(type, quantity, properties);
if (!useful(claim)) {
   /* Retry with alternate claim setting or algorithm. */
}

team = assort(claim, code, data);
if (!viable(team)) {
   /* Retry with alternate team setting or fail. */
}

if (!infect(claim, team)) {    /* employ resource(s) */
   /* Should not happen: Retry later or fail. */
}
retreat(claim);                /* clean-up of resource(s) */
```

i-let instance

Invasive Sorting

```
claim = invade(SMP, 42, COHERENT);
if (sizeof(claim) == 1)
   /* only one processing element: sort serial */
else {
   /* 1 < n <= 42 processing elements */

   team = assort(claim, code, data);    /* create workload */
   if (viable(team))
      infect(claim, team);              /* sort in parallel */

   retreat(claim);            /* await join, clean-up */
}
```

Invasive Ray Tracing

```
if (all pixels see same object) {
   claim = invade(SIMD, ∞, COHERENT);
   if (useful(claim))   /* n > 1 processing elements */
      team = assort(claim, code(SIMD), data(SIMD));
} else {
   claim = invade(MIMD, 42, COHERENT|HOMOGENEOUS);
   if (useful(claim))   /* 1 < n <= 42 processing elements */
      team = assort(claim, code(MIMD), data(MIMD));
}

if (viable(team))
   infect(claim, team);    /* run in parallel */

retreat(claim);            /* clean-up */
```

Invasive Resourcing

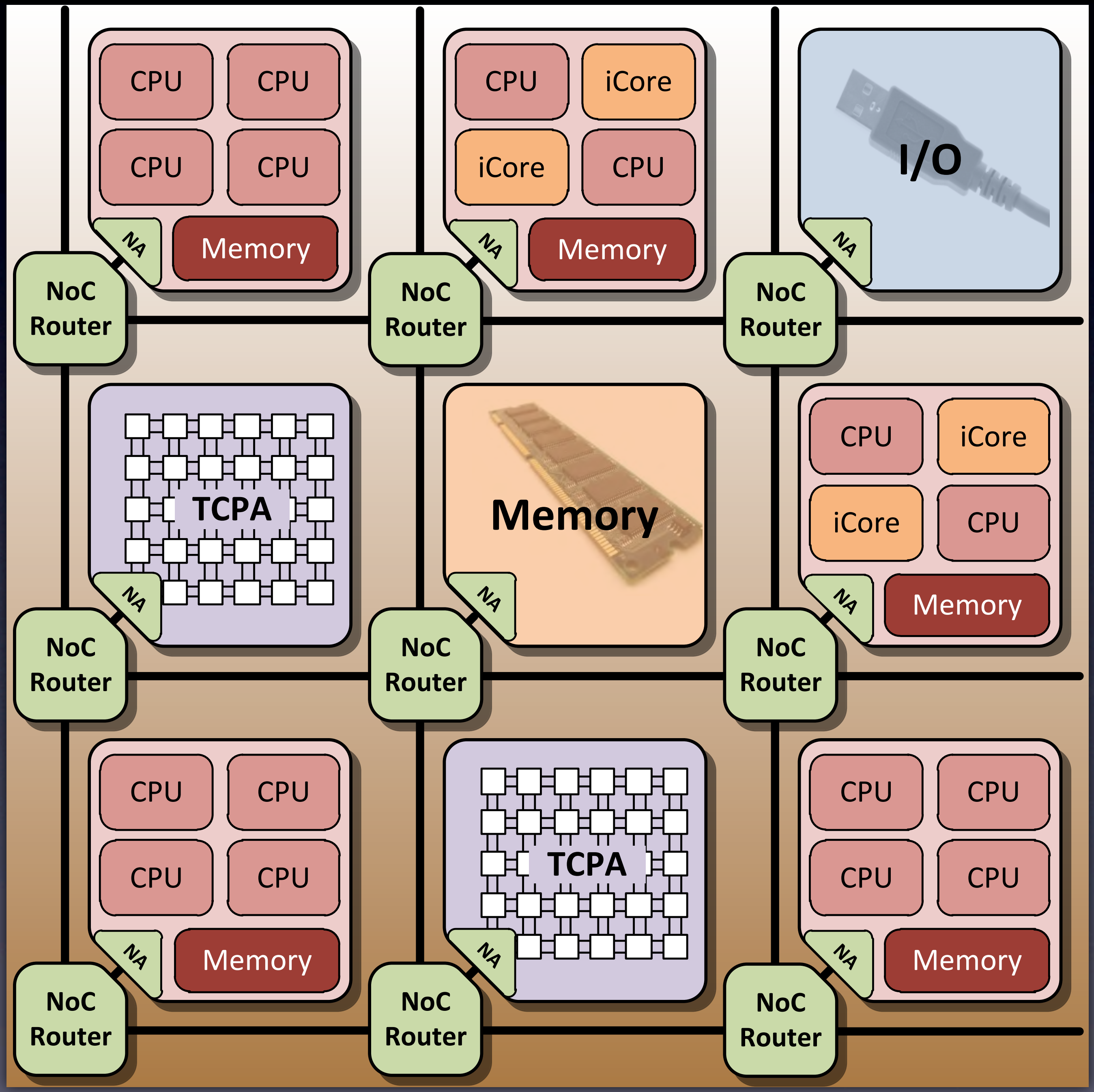
```
try {
   claim = invade(SMP, ∞, PREEMPTIVE|INTERCEPTIVE);
   /* Virtual claim: assort, infect, and retreat. */
}

catch(int what) {
   switch(what) {
      case CLAIM_CORE:  /* Handle request to release core. */
         if (this i-let gets finished shortly)
            allow(remaining period of this i-let);
         else
            yield();
         break;
      /* Other: CLAIM_PAGE, CLAIM_TILE, CLAIM_AREA... */
   }
}
```

X10: Language Support

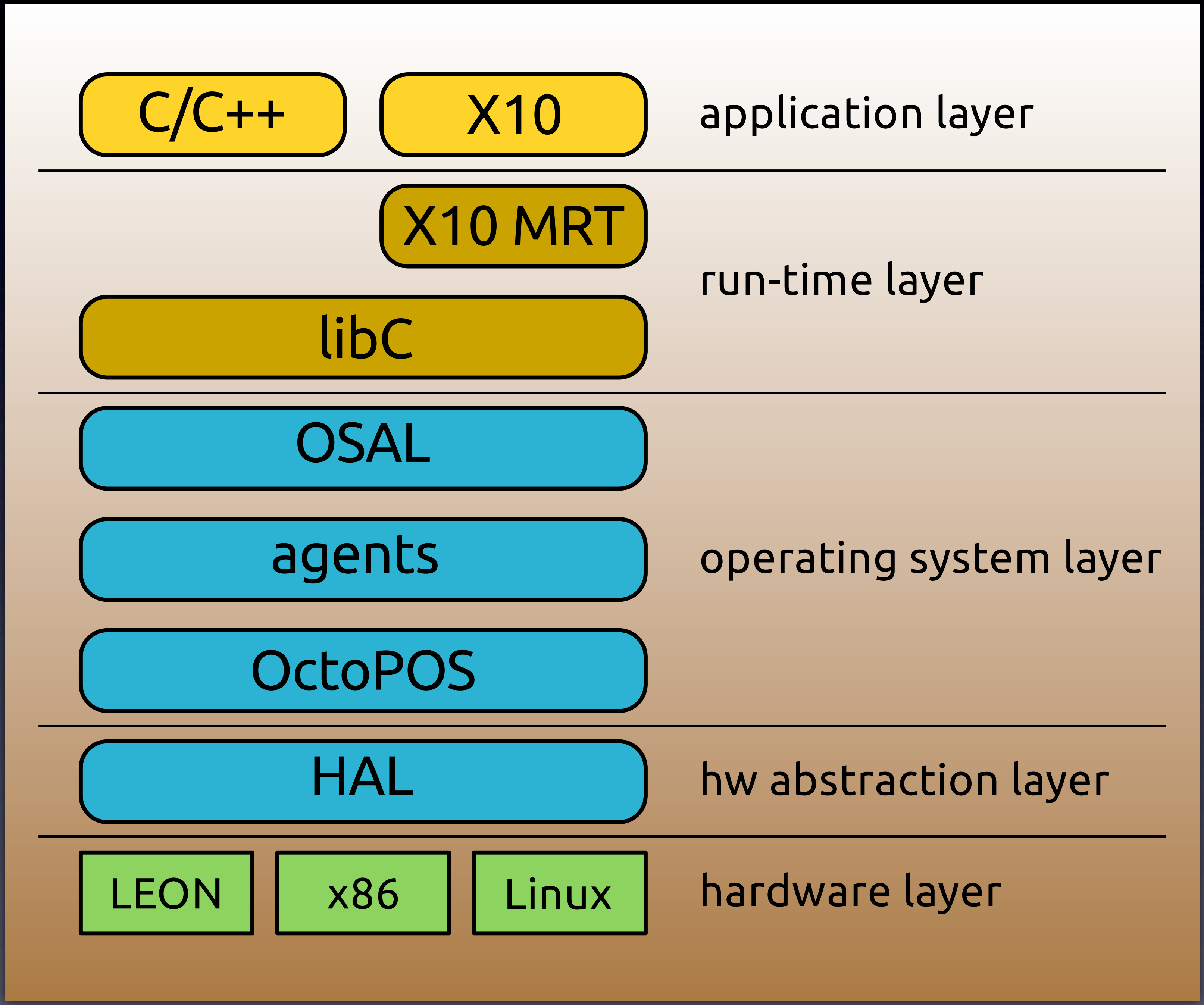
	C/C++	invasive X10	
Attributes	argument vector	class hierarchy	
Resource request	`invade(...)`	`claim.invade(...)`	
Resource use	`infect(...)`	*intra-place* *inter-place*	`async {...}` `at (...){...}`
	fan-in	`finish {...}`	
Resource release	`retreat(...)`	`claim.retreat(...)`	
Critical section	*blocking* *non-blocking*	`atomic {...}`	
Shared memory	*tile*	*place*	

System Organization and Operation

MPSoC „InvasIC“



Software Stack



Operating-System Abstraction Layer

- provides an API to an abstract operating system dedicated to invasive computing
- forwards service requests beyond invasive computing to some host operating system
- eases the embedding of invasive application programs into the InvasIC simulation system
- and enhances portability of these programs

Agent System

- takes care of resource allocation on behalf of an invading application program
- different applications compete and cooperate by means of dedicated agents
- agents bargain over incentives and strategies for competition and cooperation
- focus is on latency, energy and utilization...

OctoPOS

Octo — reference to a nature that is:

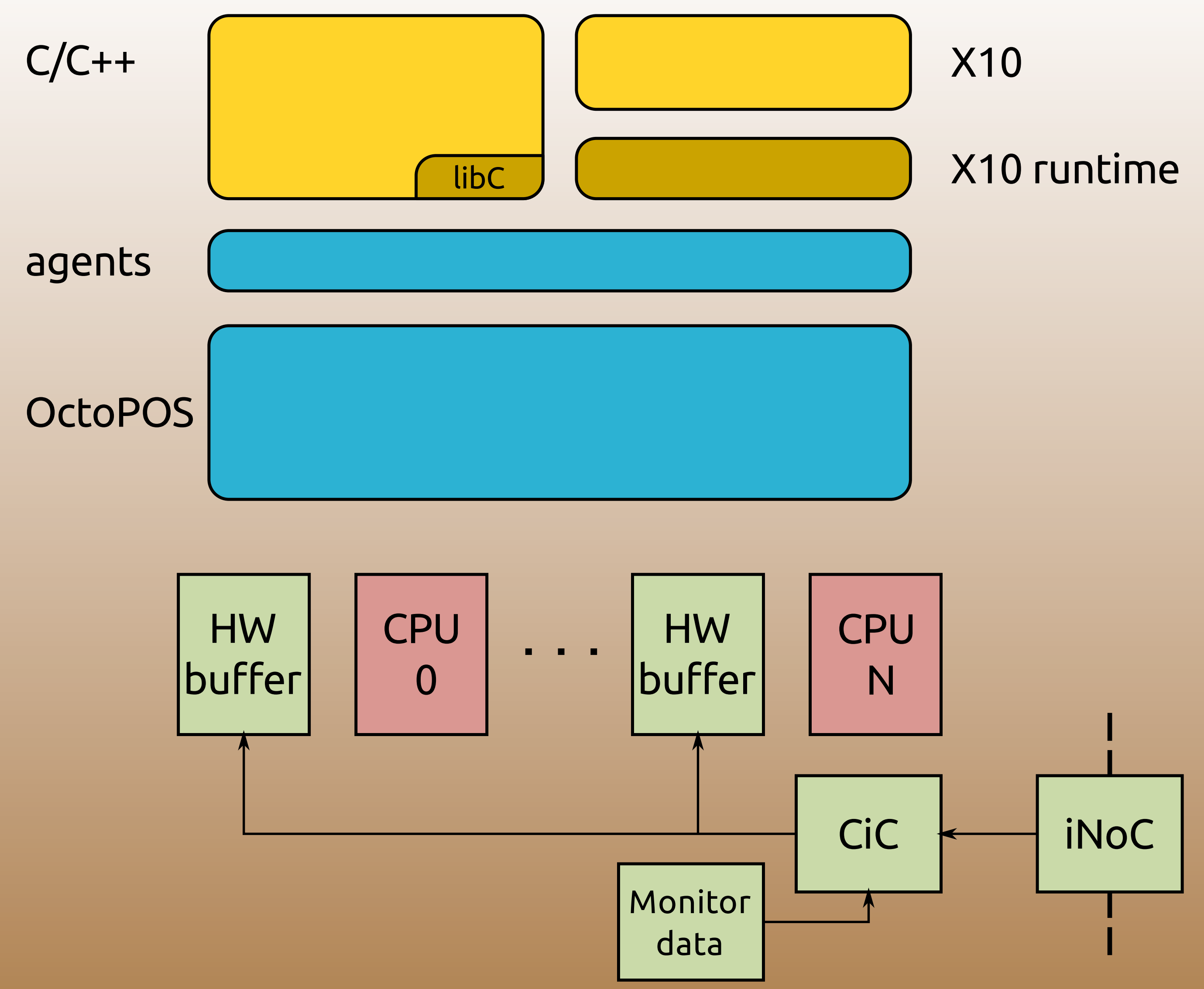
- highly parallel in its action and
- adaptable to its particular environment

POS (abbr.) for Parallel Operating System:

- an OS that not only aids parallel processing
- but also operates inherently in parallel

OctoPOS Internals

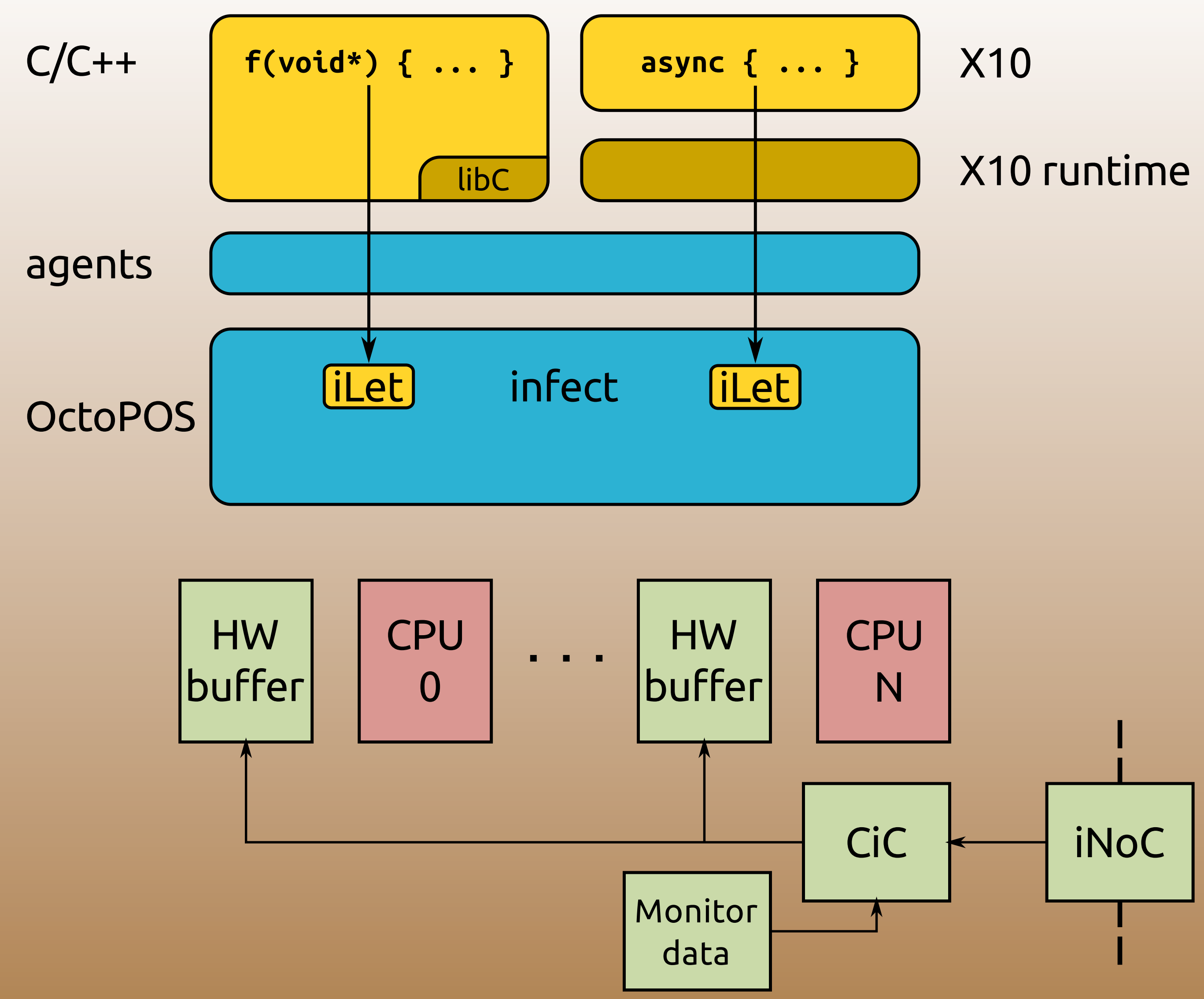
i-let Dispatching



CiC: *dynamic core i-let controller*

i-let Dispatching

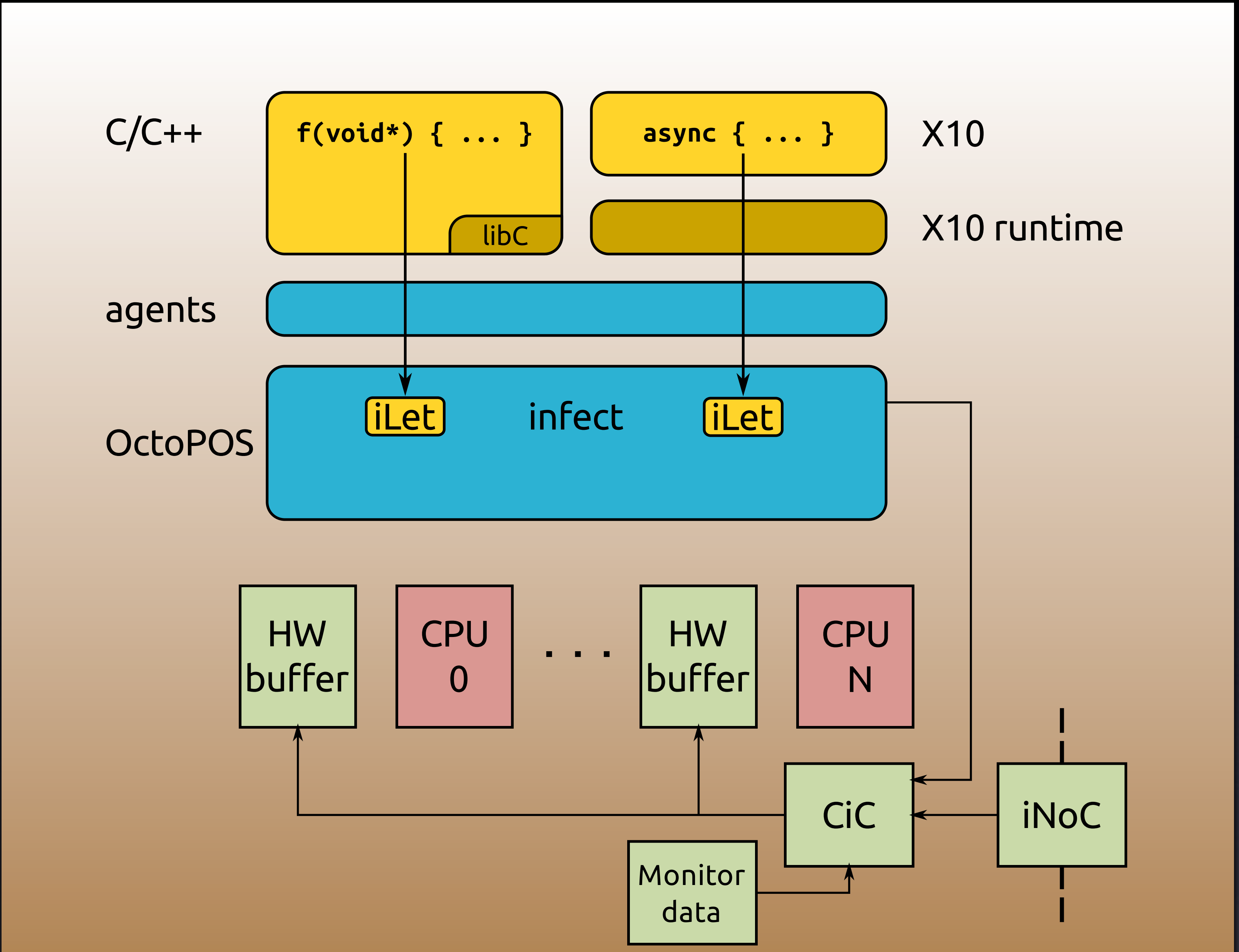
1. create *i-let* incarnation



CiC: *dynamic core i-let controller*

i-let Dispatching

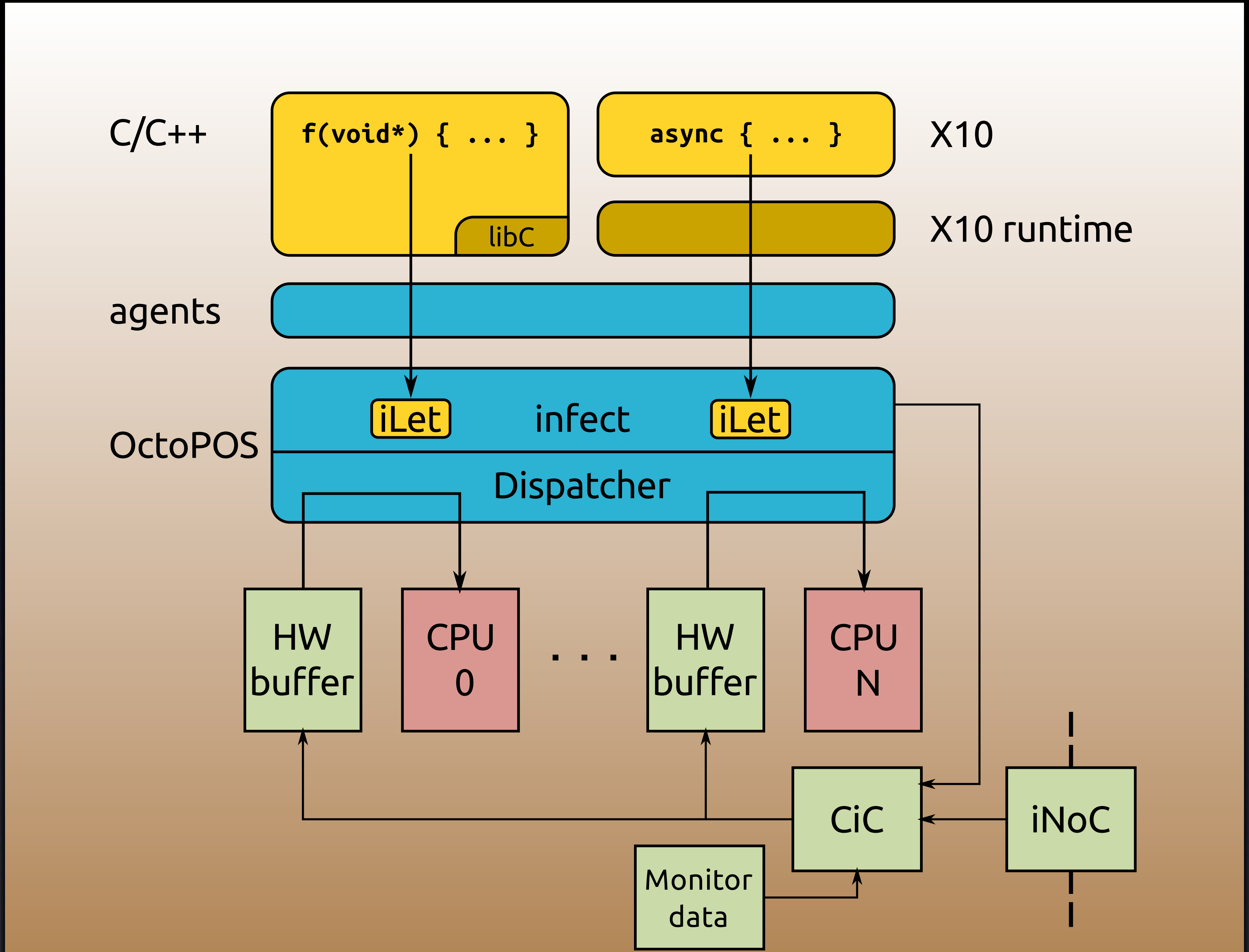
1. create *i-let* incarnation
2. trigger *i-let* dissemination



CiC: *dynamic core i-let controller*

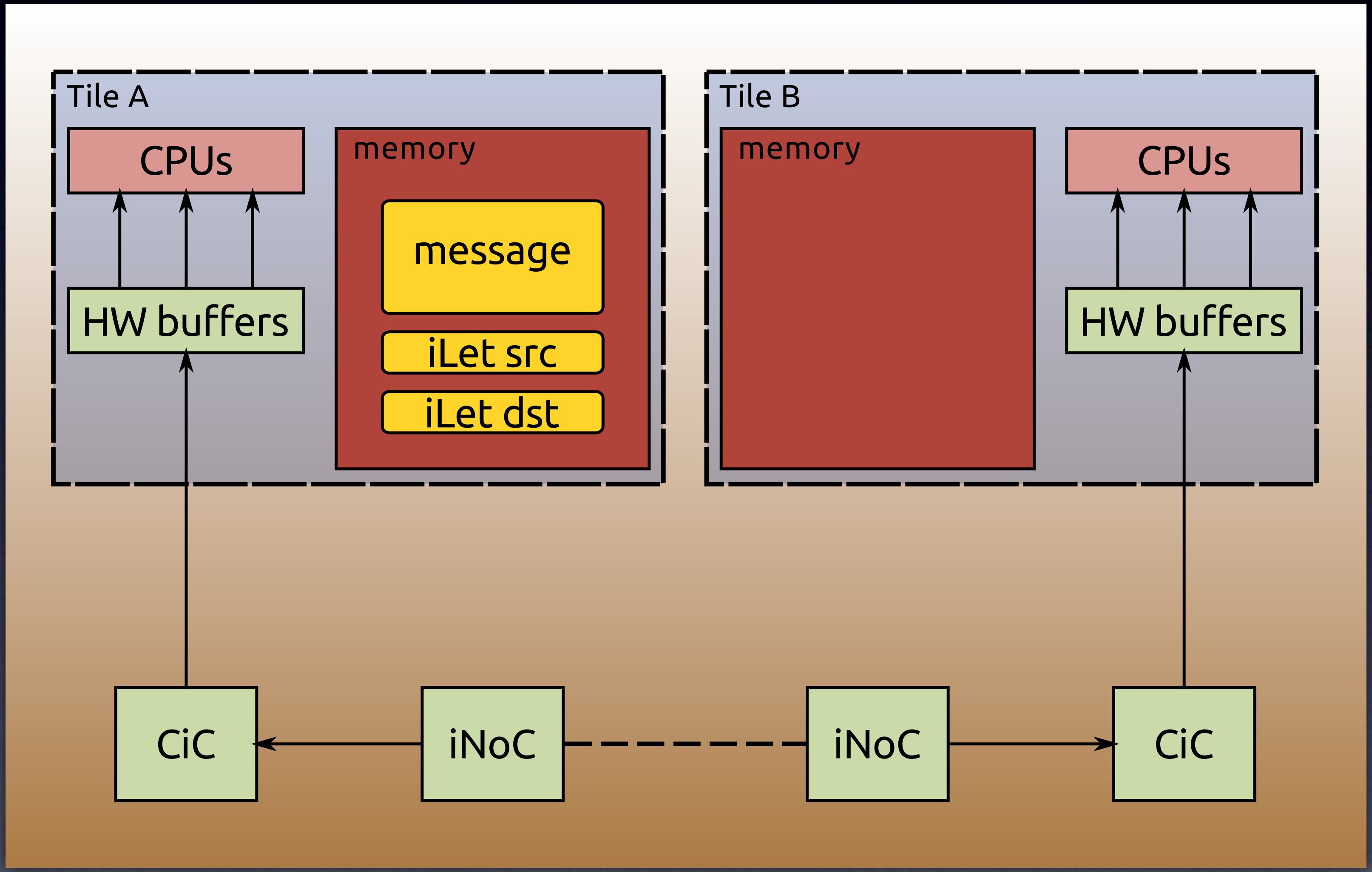
i-let Dispatching

1. create *i-let* incarnation
2. trigger *i-let* dissemination
3. dispatch *i-let* to selected core



CiC: *dynamic core i-let controller*

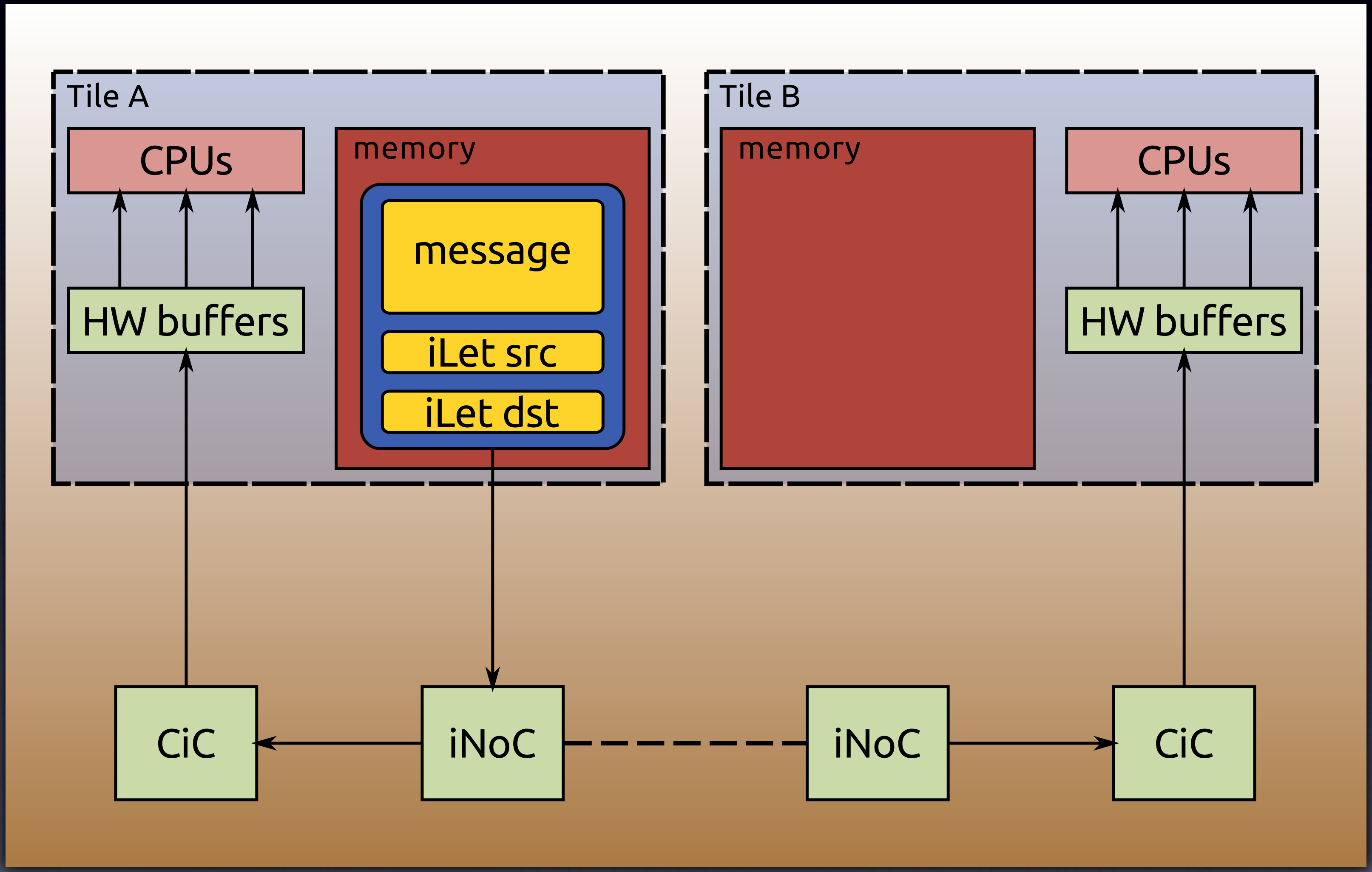
Data Transfers



active messages resembled

Data Transfers

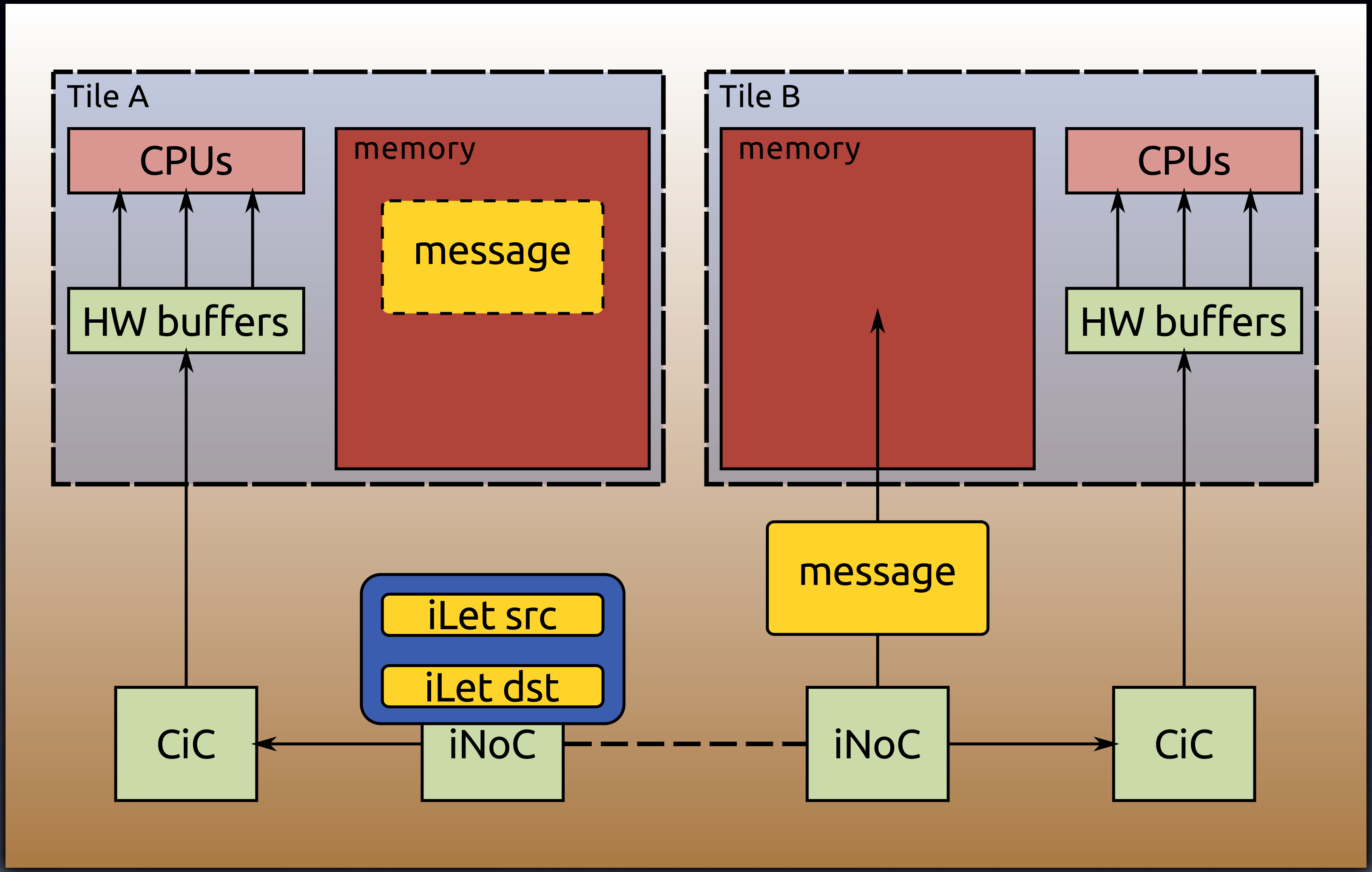
1. create & pass transfer order



active messages resembled

Data Transfers

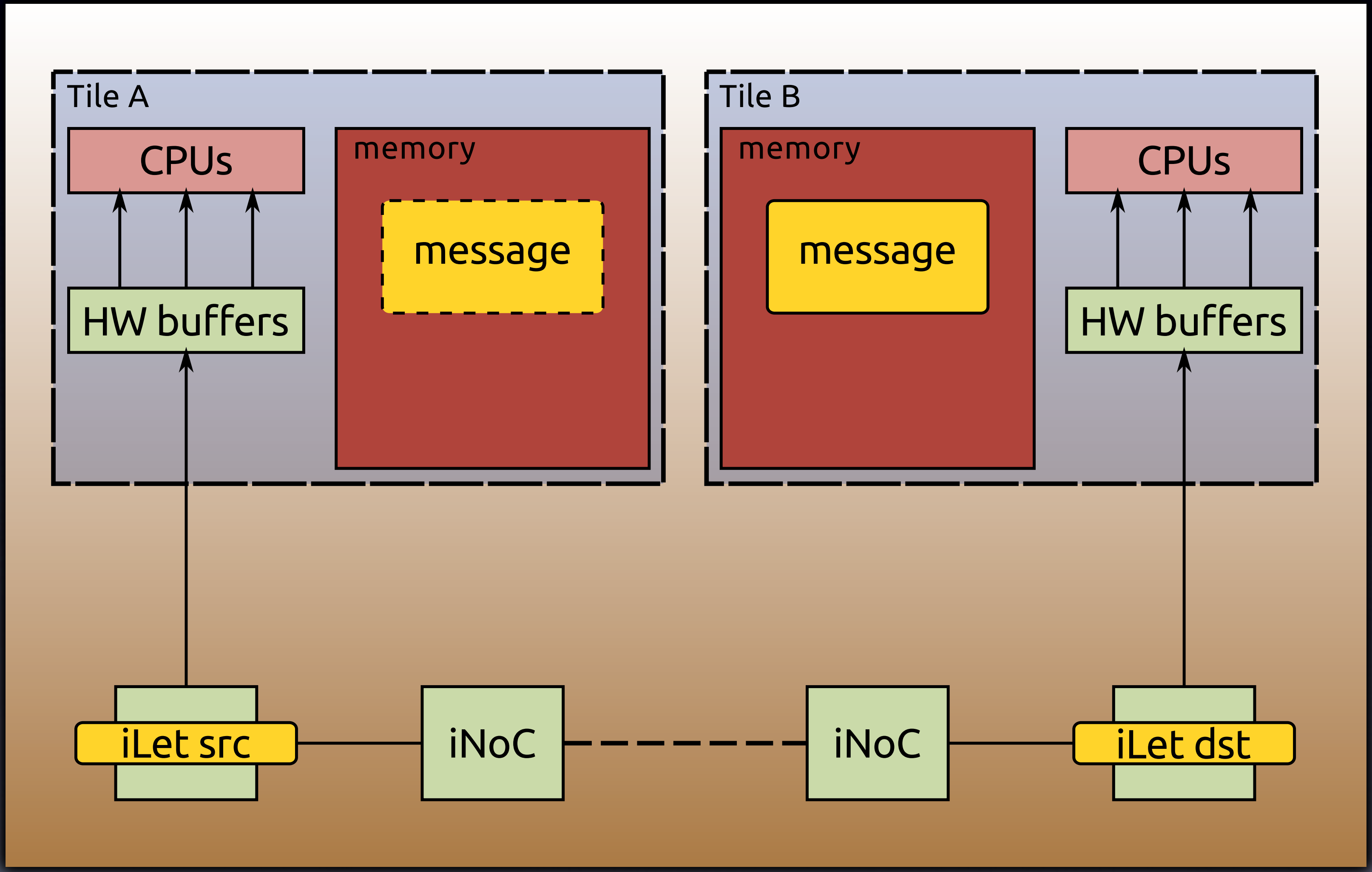
1. create & pass transfer order
2. copy (DMA) data to destination



active messages resembled

Data Transfers

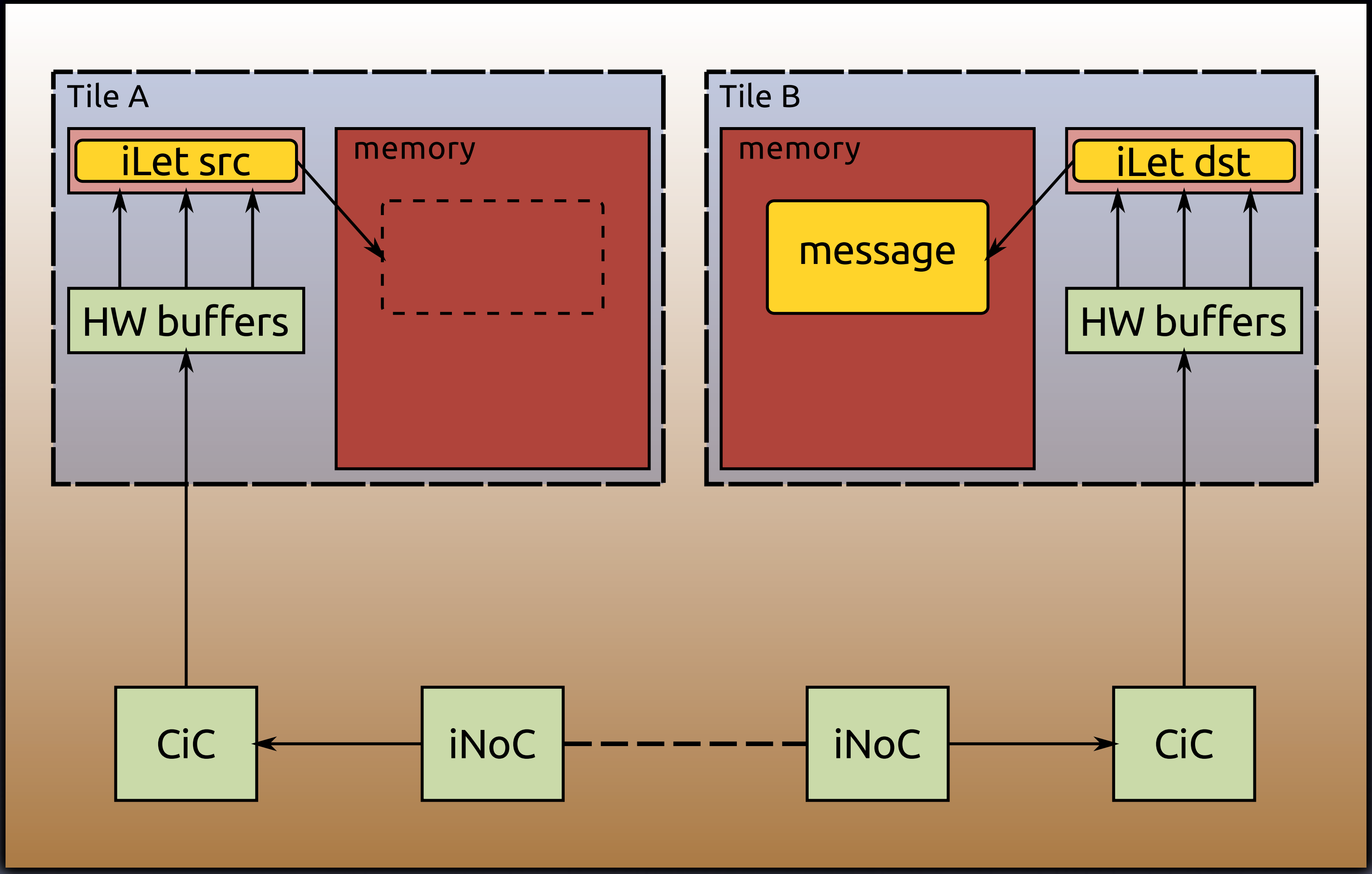
1. create & pass transfer order
2. copy (DMA) data to destination
3. disseminate & dispatch *i-lets*



active messages resembled

Data Transfers

1. create & pass transfer order
2. copy (DMA) data to destination
3. disseminate & dispatch *i-lets*
4. reuse source buffer & process data



active messages resembled

State of Work

Gaisler LEON3 FPGA-based prototyping system:

- coherent, homogeneous, nonpreemptive, passive claim and run-to-completion teams
- 1 compute tile of max. 6 cores

Synopsis CHIPit demonstration system:

- 3x3 array of (compute, memory, I/O) tiles
- max. 24 LEON3 cores

AMD 48-core experimental system: porting...

Systems-Programming Perspective

Field of Interest

- more optimistic – and less pessimistic – concurrency control
- latency hiding through relocation of system activities to available cores
- control of system-level resource sharing among user-level processes
- re-engineering of legacy operating systems for time-sharing or real-time mode

Synchronization

Agile Critical Sections

Self-organizing and use-case dependent protection of non-sequential programs against concurrent processes

- procedures vary with degree of contention
- coexisting variants of the same critical region
- ``synchronization on the fly´´
- hardware-supported (e.g. ASF) or -based

Exploitation of Cores

- many-core processor
  - universal cores — *load balancing*
  - special cores — *latency hiding (system-inherent activities)*
    - changing core — *system call handling as a whole*
    - fixed core — *system call handling in pieces (pipelined)*
      - driver core — *off-line I/O*
        - device core — *peripheral control*
        - interrupt core — *interrupt handling*
      - service core — *off-line service*
        - scheduling core — *task delivery*
        - I/O core — *file operation*
        - communication core — *protocol stack processing*
        - paging core — *external pager (replacement policy)*
        - synchronization core — *semaphore/ticket server, RCL*
        - ⋮

The Devil is in the Details

Caches!

Closing

Beyond InvasIC

LAOS (Latency-Aware Operating System)

- latency-awareness in operating systems for massively-parallel processors
- 2 scientists, 2 student research assistants

COKE (Coherency Kernel)

- software-controlled consistency and coherence for many-core architectures
- 2 scientists, 2 student research assistants

Summary

- thousands of processing elements of clustered heterogeneous CPUs need to be engaged
- multiplexing of single cores (then) becomes more and more insignifcant
- programming of massively-parallel systems calls for abstraction *and* resource-awareness
- operating systems have to reflect on these issues and leave old ways behind!