

Hardware-level thread migration in a 110-core shared-memory multiprocessor



Mieszko Lis Keun Sup Shim
Brandon Cho Ilia Lebedev
Srinivas Devadas

Execution Migration Machine: Highlights

- **110-core chip multiprocessor**
 - unified shared memory, general-purpose
- **Fast, autonomous thread migration**
 - fast: migration entirely in hardware
 - fine-grained: instruction granularity
 - autonomous: hardware decides when to migrate
- **Reduces on-chip traffic up to 14-fold**
 - less interconnect traffic → lower dynamic power

Outline

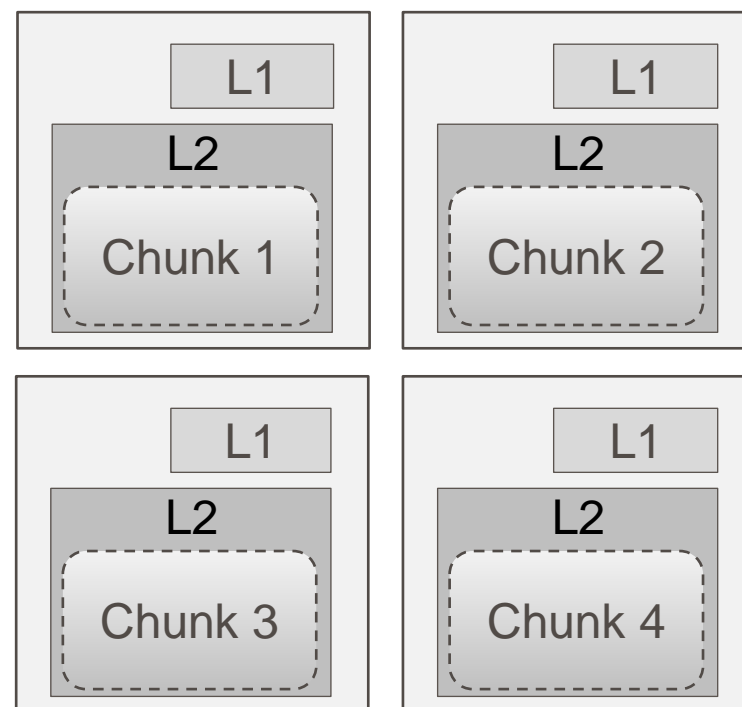
- **Problem statement**
- **The Execution Migration Machine**
- **Costs and performance**

Problem? On-chip traffic

- **On-chip interconnect power already significant**
 - MIT RAW (16 RISCy cores): **39%** of tile [ISPLED 2003]
 - Intel TeraFLOPS (80 double-MAC cores): **28%** of tile power [JSSC 2008]
- **This is getting worse...**
 - transistor dimensions continue to scale
 - but shrinking wires makes them **slow** and **hot**:
higher RC-delay, power, crosstalk
 - relatively shorter and shorter wires feasible
- **...and worse**
 - scaling technology nodes + short wires → lots (100s–1000s) of cores
 - more cores need more data to process
 - so interconnects have to carry more and more data

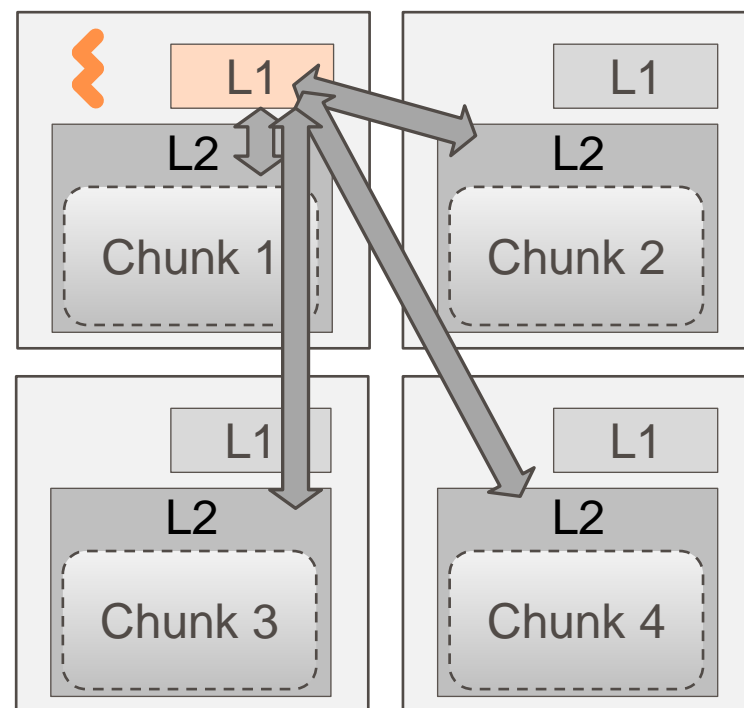
What causes this traffic?

- **In a typical multicore CPU...**
 - fast private caches (say L1)
 - slower shared caches (say L2)
 - shared caches in per-tile shards



What causes this traffic?

- In a typical multicore CPU...
 - fast private caches (say L1)
 - slower shared caches (say L2)
 - shared caches in per-tile shards
- **Data fetched to threads**
 - threads generally pinned to cores
 - interconnect brings data to the locus of computation



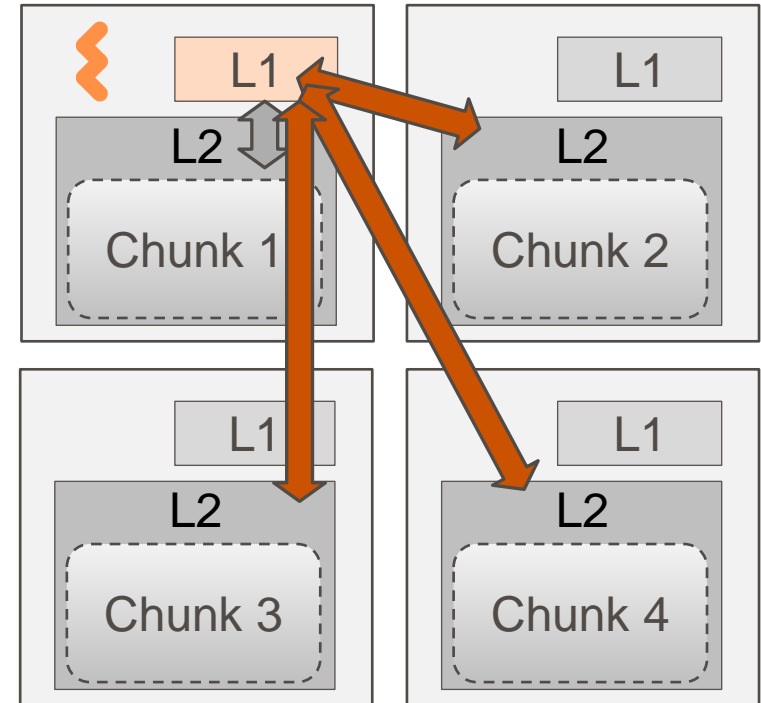
What causes this traffic?

- **In a typical multicore CPU...**

- fast private caches (say L1)
- slower shared caches (say L2)
- shared caches in per-tile shards

- **Data fetched to threads**

- threads generally pinned to cores
- interconnect brings data to the locus of computation

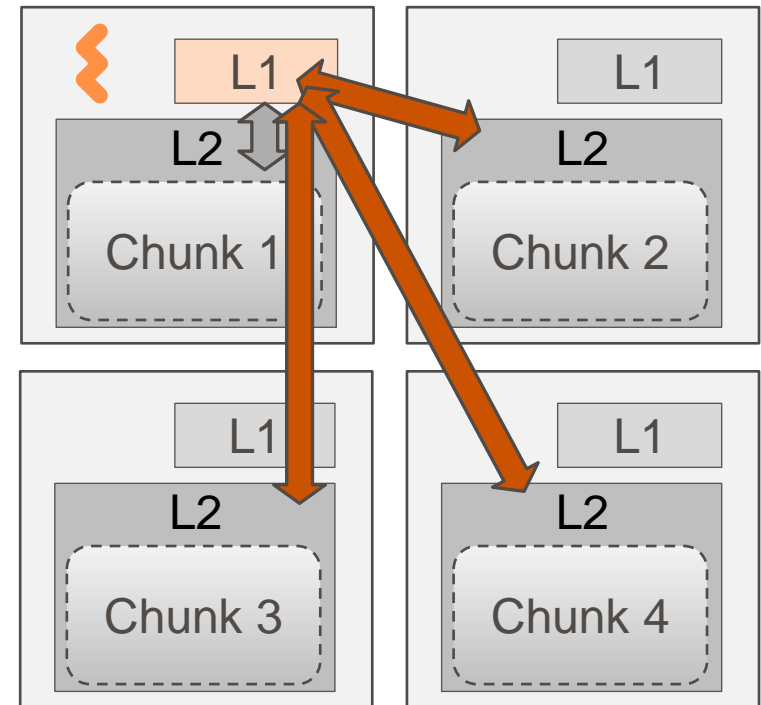


- **If workloads exceed L1 capacity...**

- repeated L1 fetches from remote L2 chunks → lots of traffic
- common with big datasets: db apps, machine learning, etc.

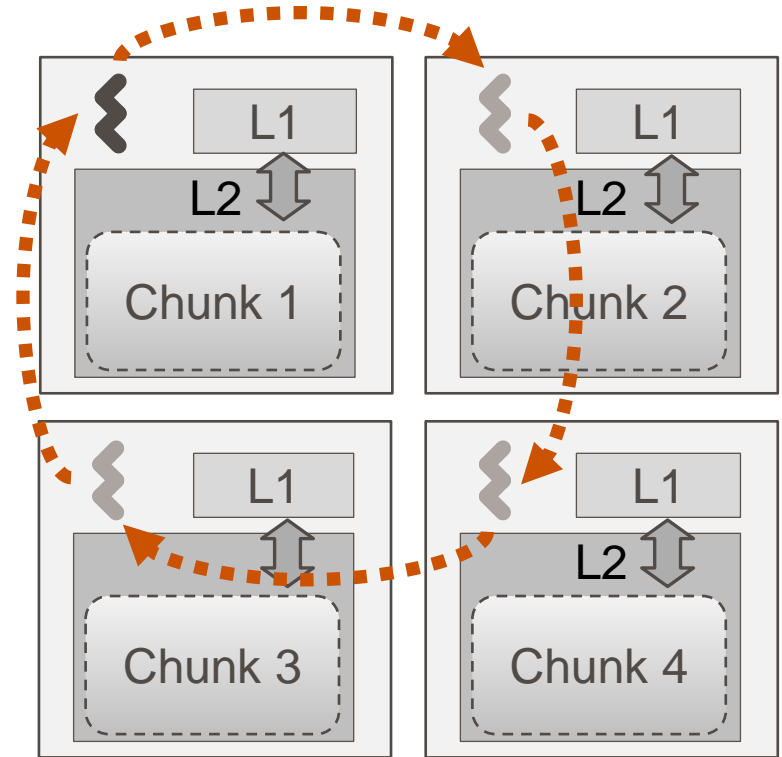
How can we reduce this traffic?

- **Locality is everything**
 - many remote L2 requests = bad
 - would prefer to access **local** L2



How can we reduce this traffic?

- **Locality is everything**
 - many remote L2 requests = bad
 - would prefer to access **local** L2
- **Move the threads around!**
 - threads follow data
 - makes remote accesses local
 - one-off accesses can be remote
- **...but this requires an efficient hardware solution**
 - even high-locality apps can't amortize slow migrations



Thread migration desiderata

- **Frequent use requires *fast* migration**
 - should not involve software (e.g., OS)
 - should not involve round-trips (e.g., via cache coherence)
- **Frequent use requires *fine-grained* migration**
 - data accesses are dynamic, need to respond quickly
 - should not involve centralized scheduling
- **Reducing on-chip traffic requires *small* migrations**
 - either keep thread context small
 - or migrate only a useful subset
- **Realistic evaluation requires a *large core count***
 - simple core and scalable memory model

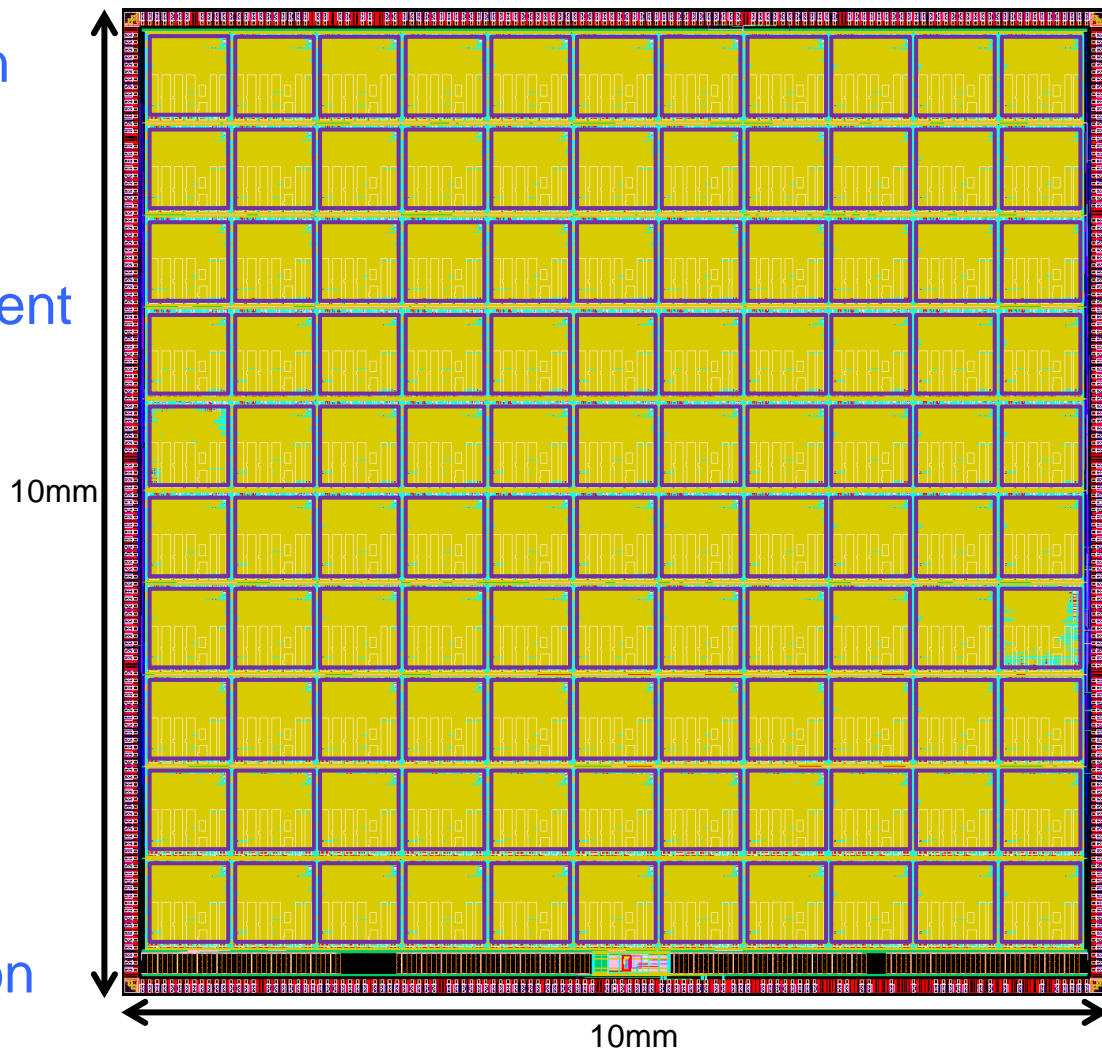
The Execution Migration Machine

- **Goals:**

- most efficient thread migration
- simple & scalable design
- explore what is possible
- focus on on-chip data movement

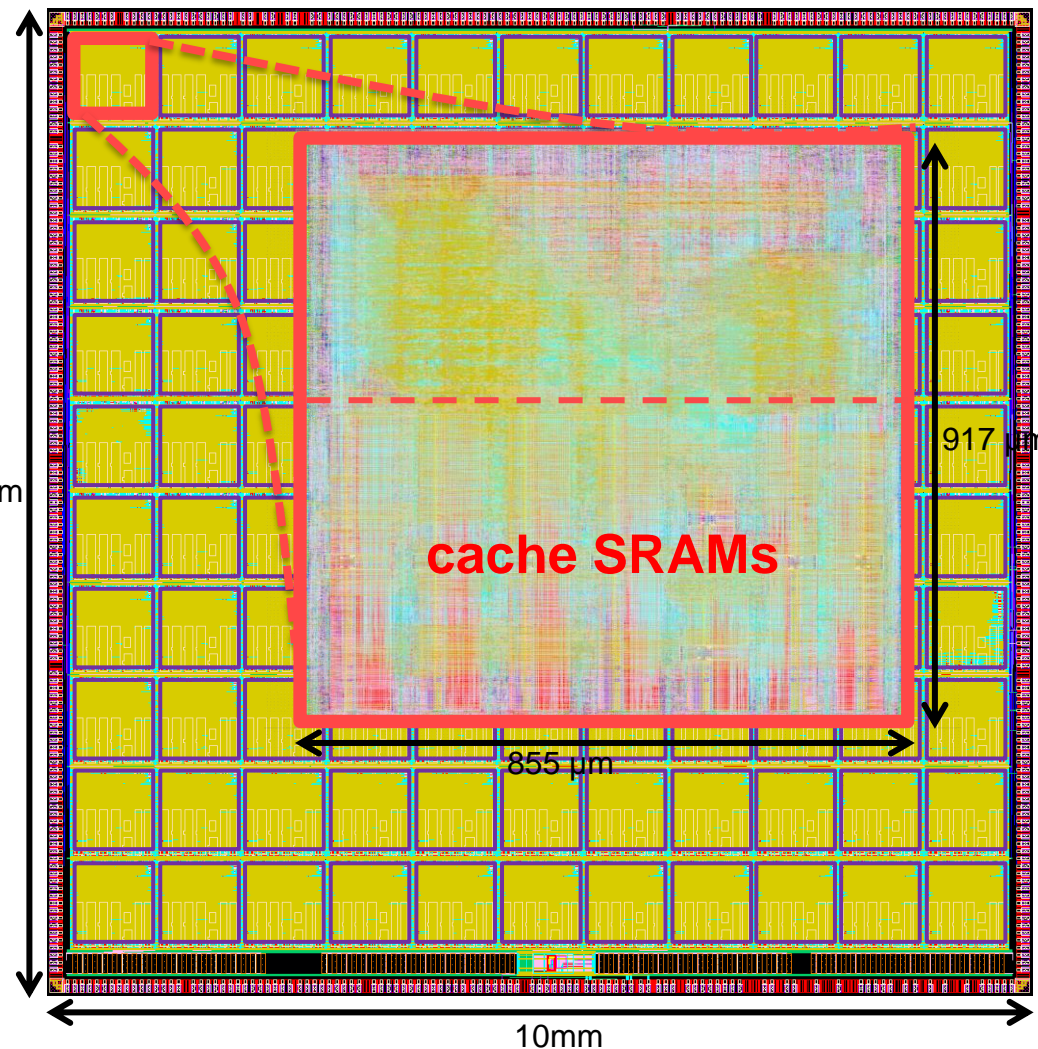
- **ASIC:**

- 10mm x 10mm in 45nm
- 110 homogeneous cores
- single level of cache
- 2D mesh interconnect
- 2 off-chip memory interfaces
- optimized for efficient migration

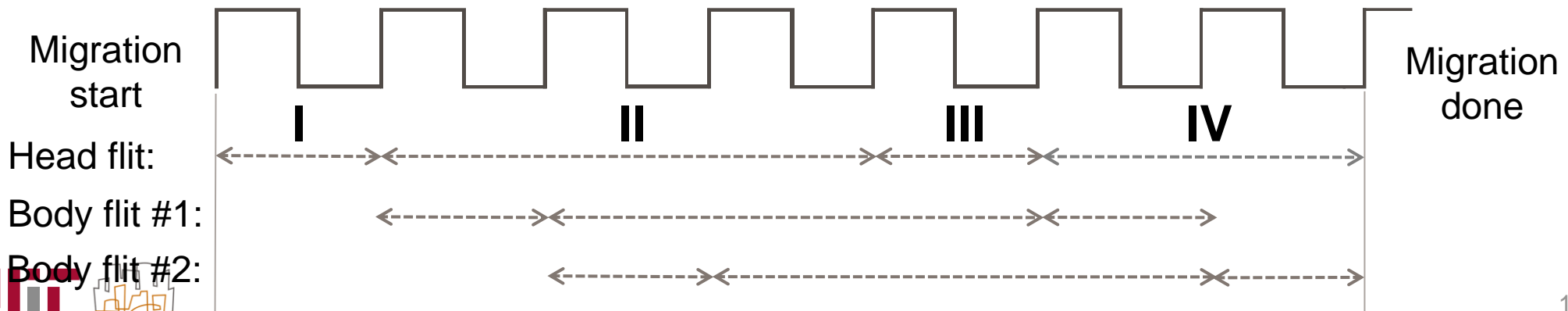
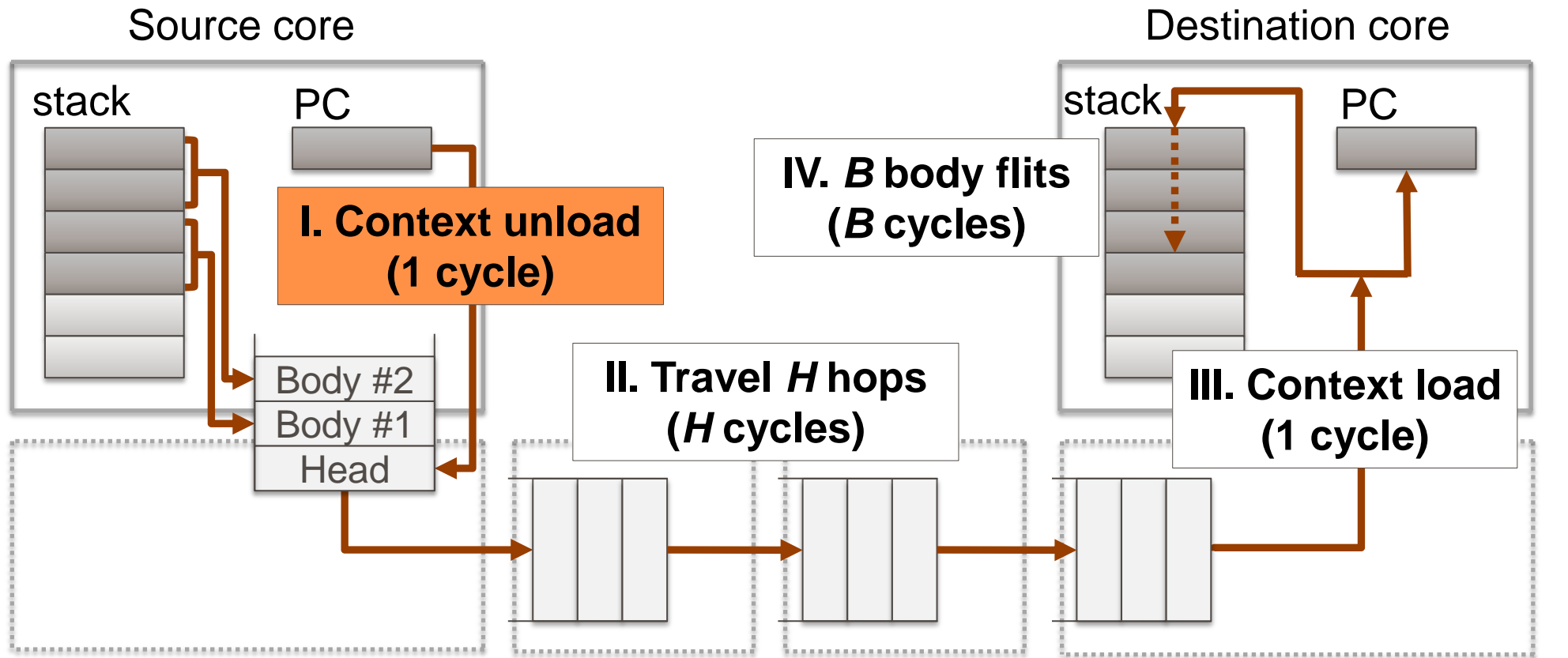


Tile architecture

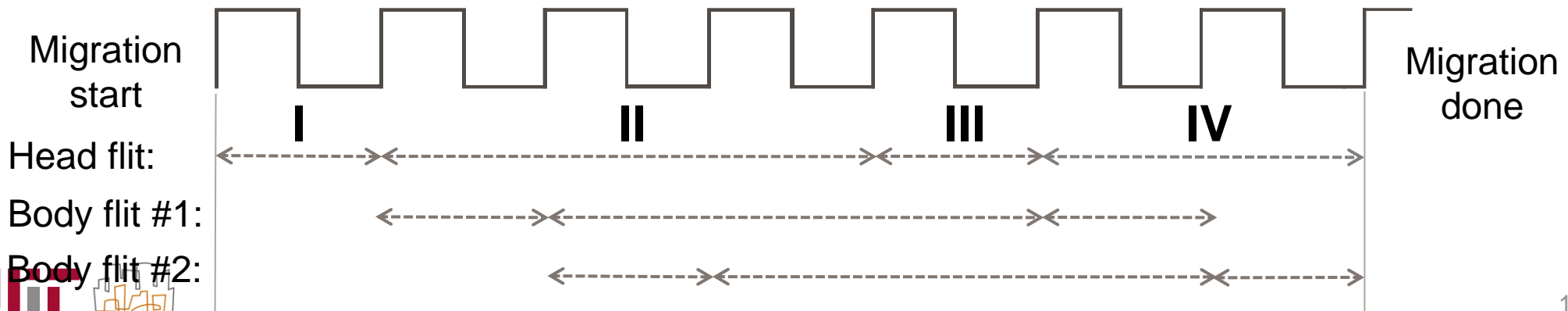
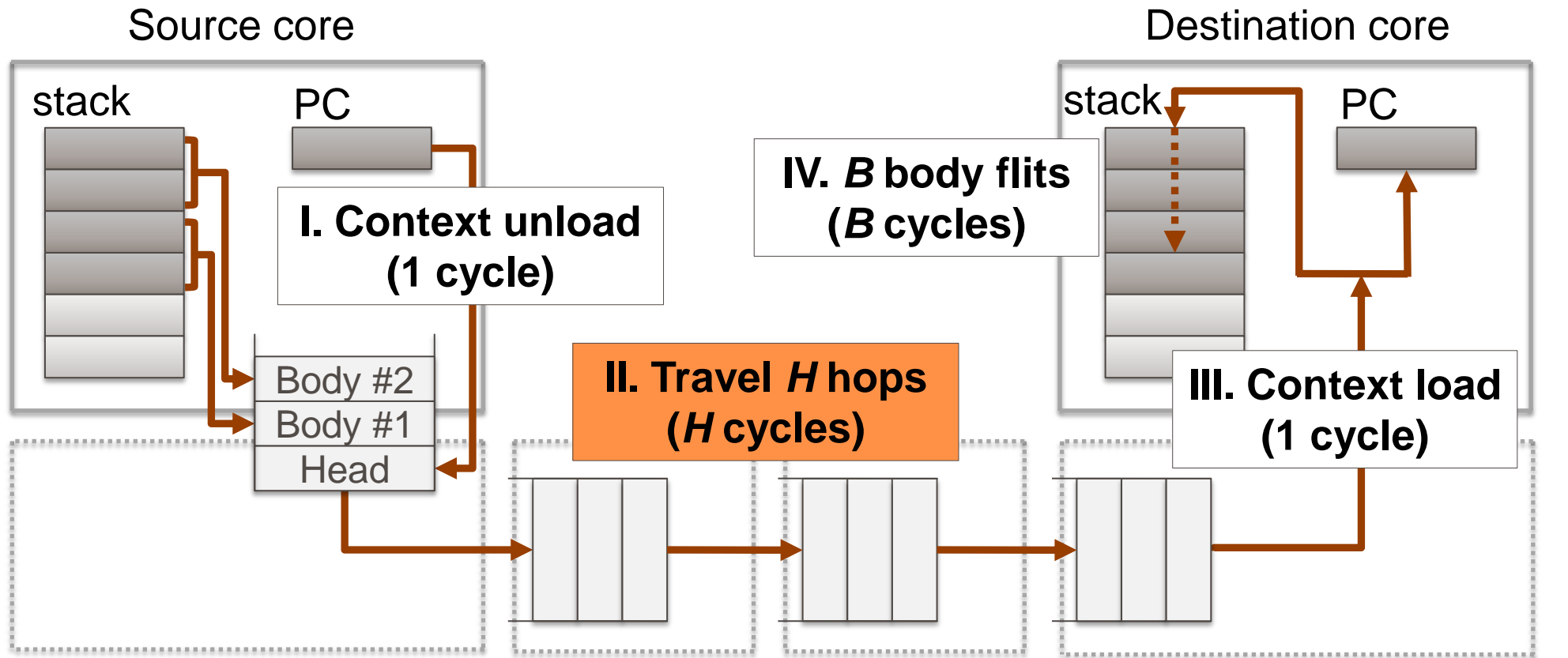
- **Caches: over half of each tile**
 - 32KB data cache
 - 8KB instruction cache
- **Six on-chip network routers**
 - 64-bit flits, wormhole DoR
 - single-cycle if no congestion
 - six ensure deadlock freedom
- **Custom stack-based core**
 - instruction-granularity migrations
 - allows partial-context xfers (min. 128 bits)
 - two stacks, automatically spilled/refilled via the data cache
 - **two SMT contexts** ensure deadlock-free migrations



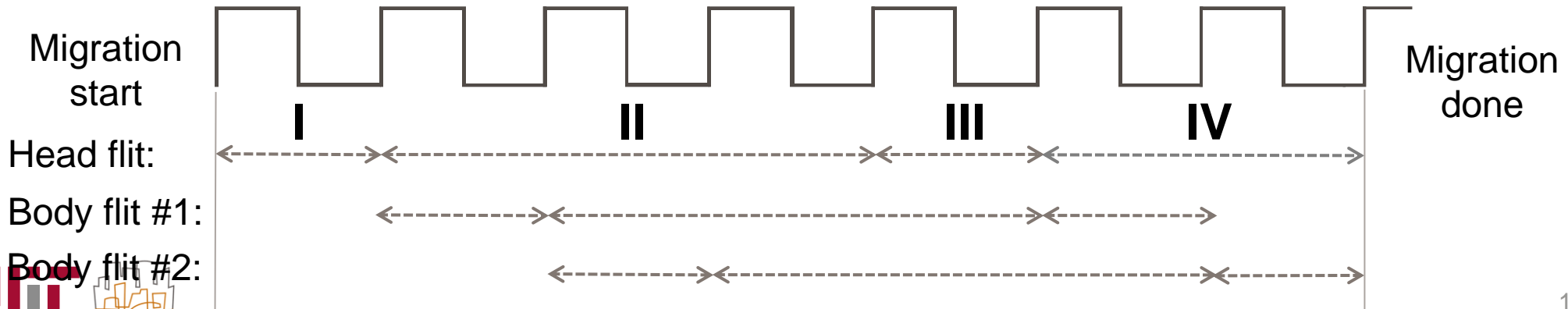
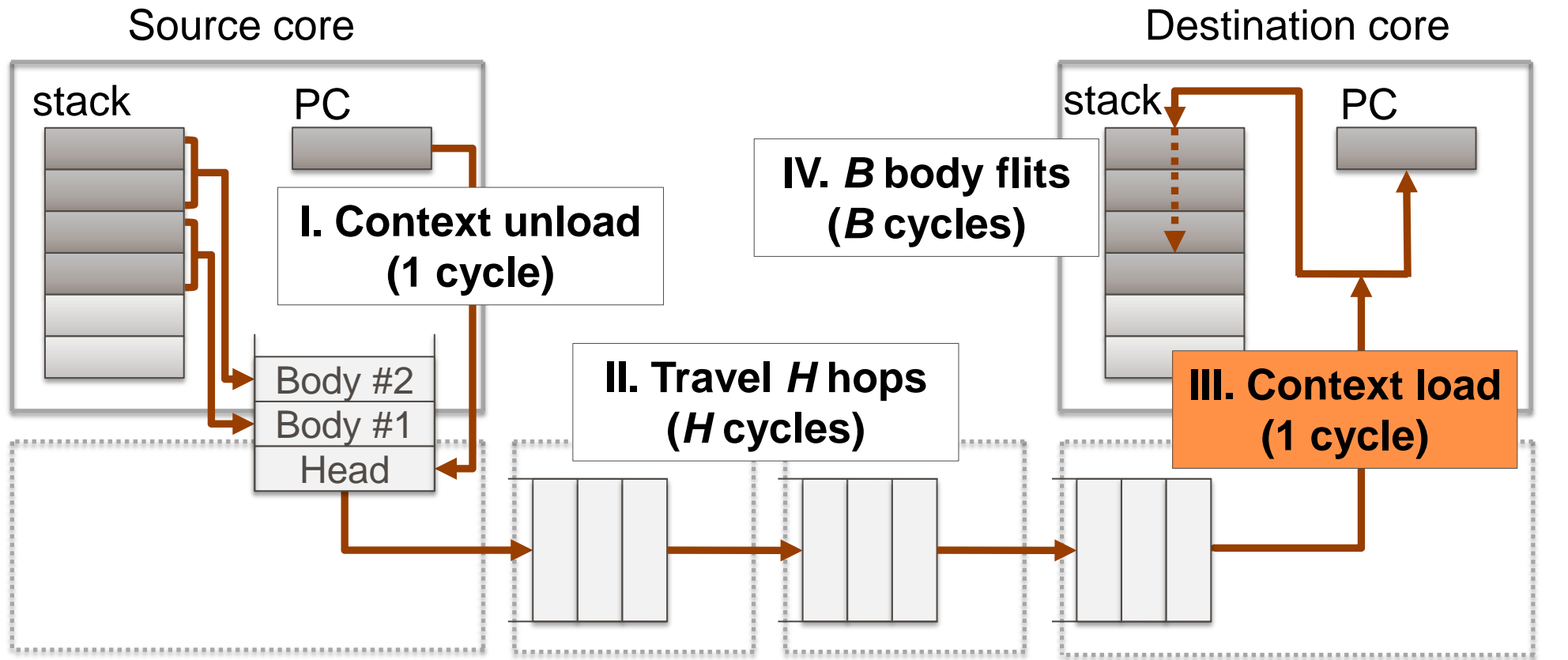
Superfast migration: min 4 cycles



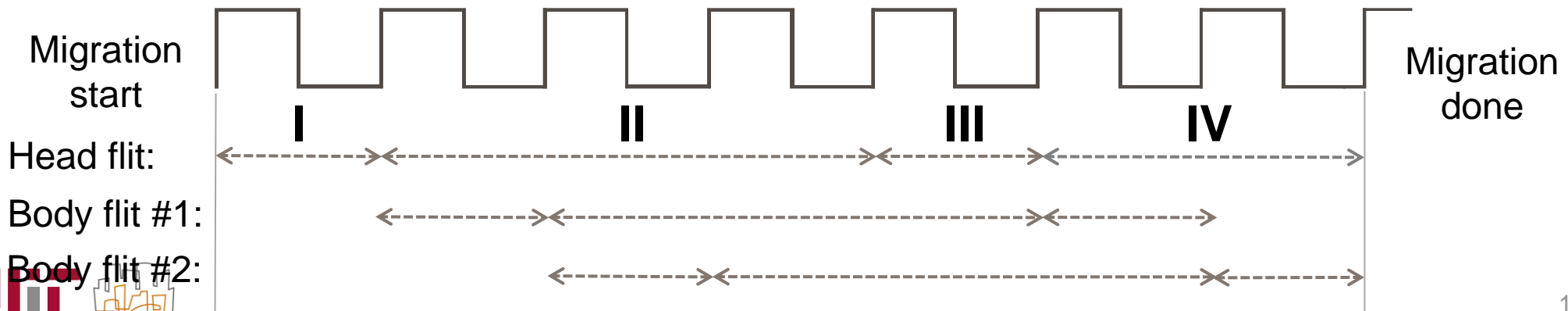
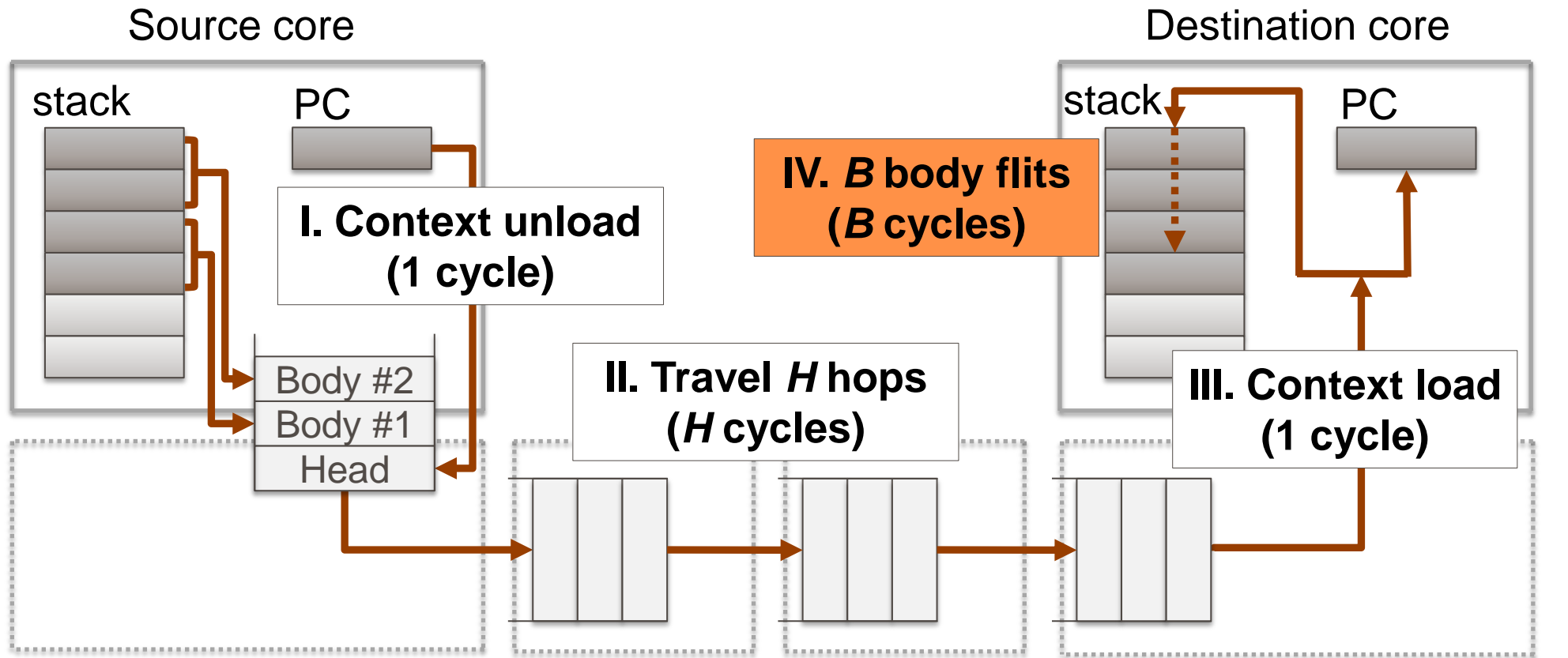
Superfast migration: min 4 cycles



Superfast migration: min 4 cycles



Superfast migration: min 4 cycles



Shared memory model

- **Shared D\$ architecture**

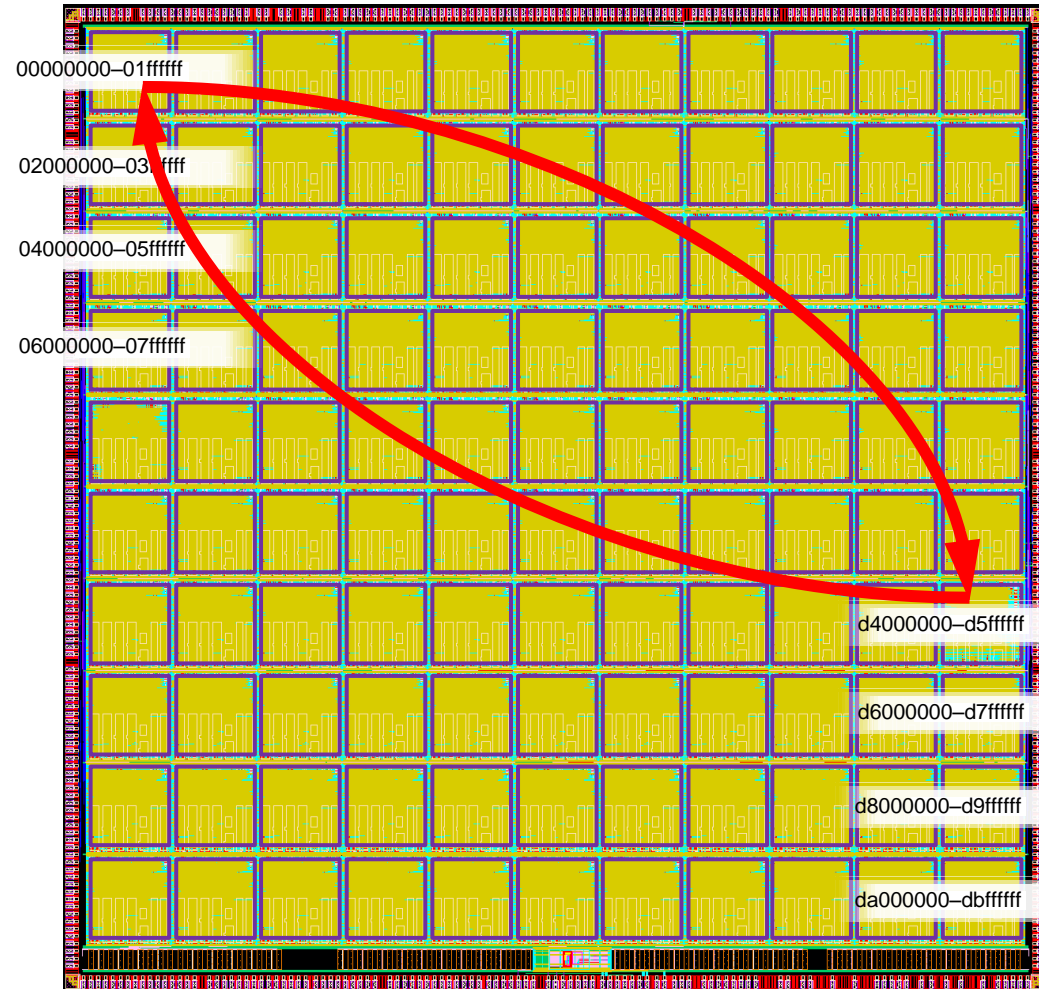
- one D\$ slice in each core
- top 7 bits of address \rightarrow core ID
- a specific address can only be cached in a specific core
- memory consistency trivial

- **LD/ST: via remote D\$ access**

- LD/ST/LD_RSV/ST_CND
- word request to remote D\$
- result / ack from remote D\$
- cannot cache data locally
 \rightarrow round-trip for every access

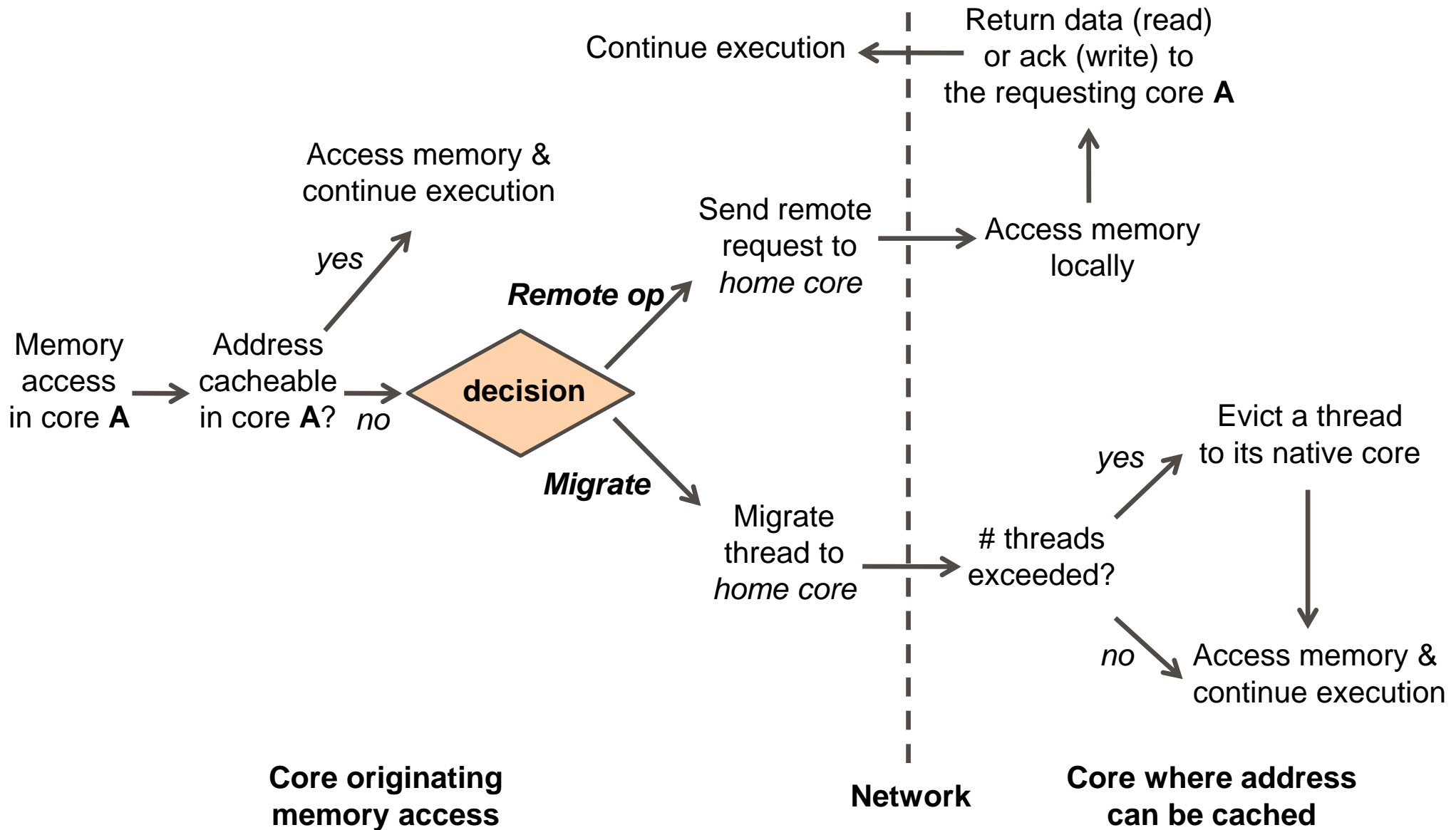
- **Migration accelerates this**

- turns multiple round-trips into one trip + local accesses



+ d6000000-ffffff cacheable in all cores

Shared memory & migration

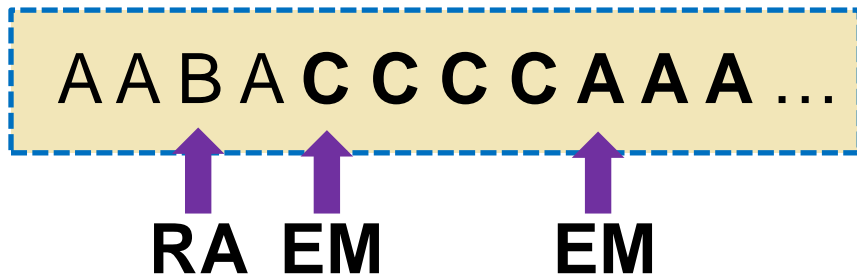


Supported modes of migration

- **Instruction-based**
 - e.g., migrate to core 10
- **LD/ST-triggered, static**
 - determine if effective address cached in remote D\$
 - if remote, **memory instruction** specifies whether to migrate
 - suitable for access patterns amenable to static/profiling analysis (or for the particularly determined programmer)
- **LD/ST-triggered, fully automatic**
 - determine if effective address cached in remote D\$
 - if remote, **learning predictor** decides whether to migrate
 - suitable for dynamically changing access patterns (or the not-so-determined programmer)

Learning migration predictors

- Per-tile predictors trigger migrations when advantageous
 - detect **long runs** of accesses to the same core



Learning migration predictors

- Per-tile predictors trigger migrations when advantageous
 - detect long runs of accesses to the same core
 - enter **sequence start PC** into predictor table, migrate on this PC next time

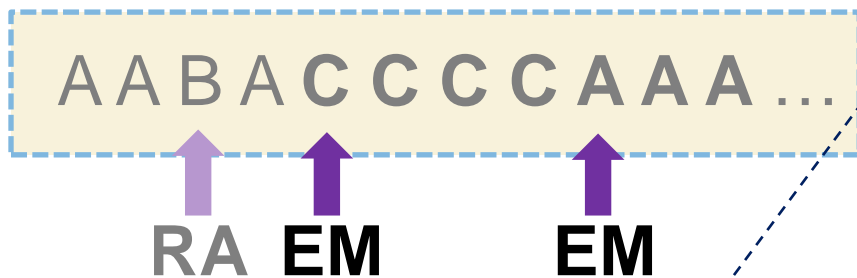
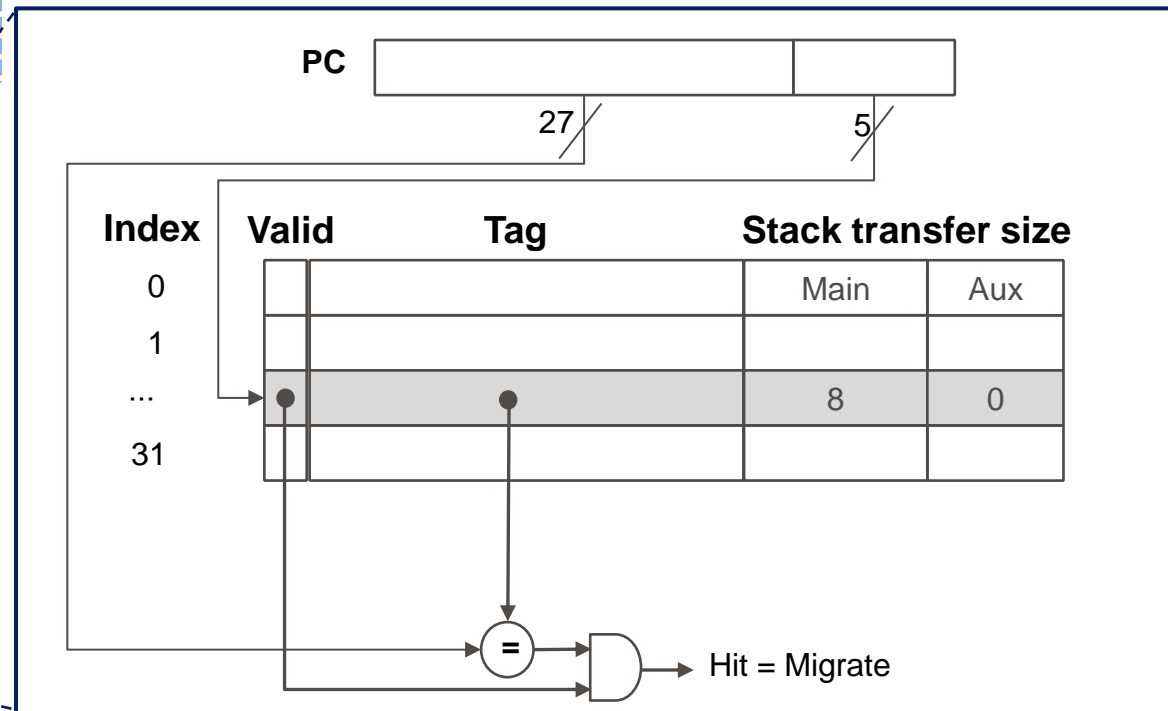


table indexed by seq. start PC



Learning migration predictors

- **Per-tile predictors trigger migrations when advantageous**
 - detect **long runs** of accesses to the same core
 - enter **sequence start PC** into predictor table, migrate on this PC next time
 - **adjust migrated stack size** if migrated back on stack over/underflow

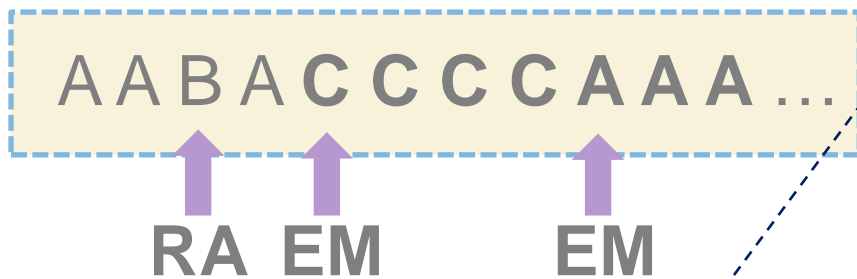
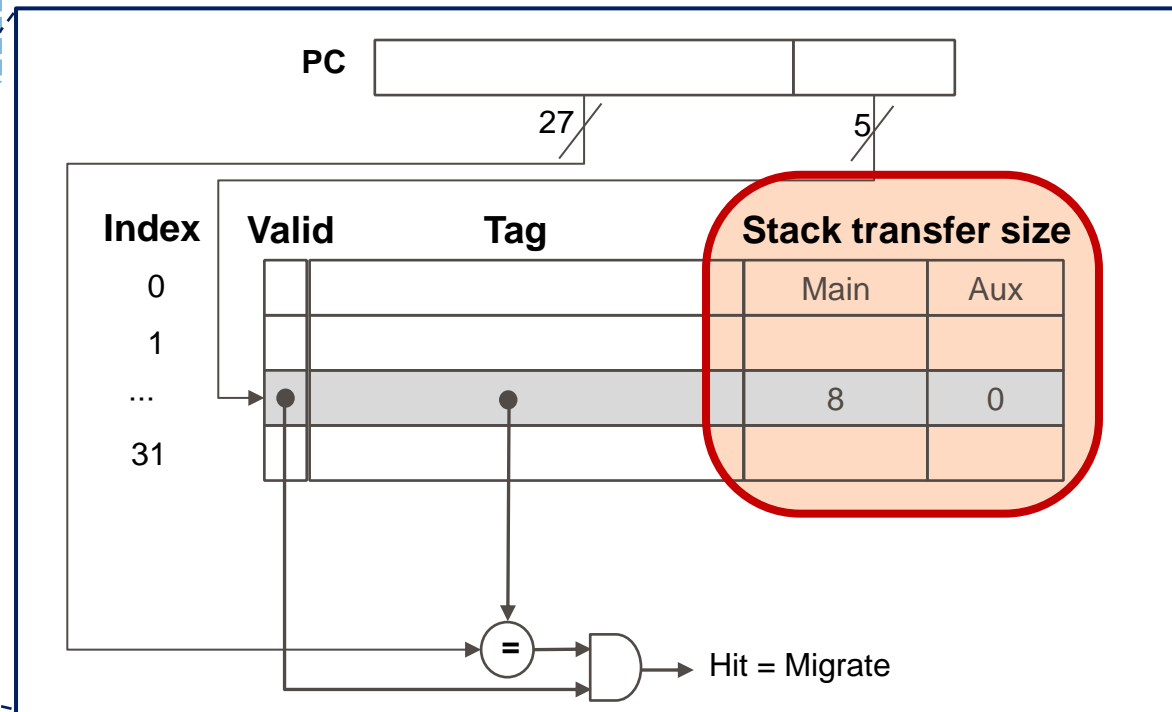


table indexed by seq. start PC



Learning migration predictors

- **Per-tile predictors trigger migrations when advantageous**
 - detect **long runs** of accesses to the same core
 - enter **sequence start PC** into predictor table, migrate on this PC next time
 - **adjust migrated stack size** if migrated back on stack over/underflow
 - **remove start PC** from table if too few accesses after migration

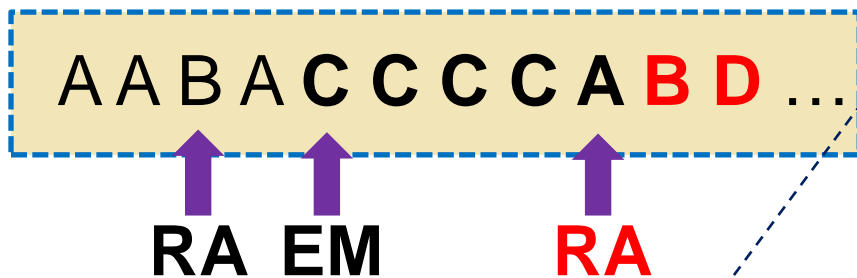
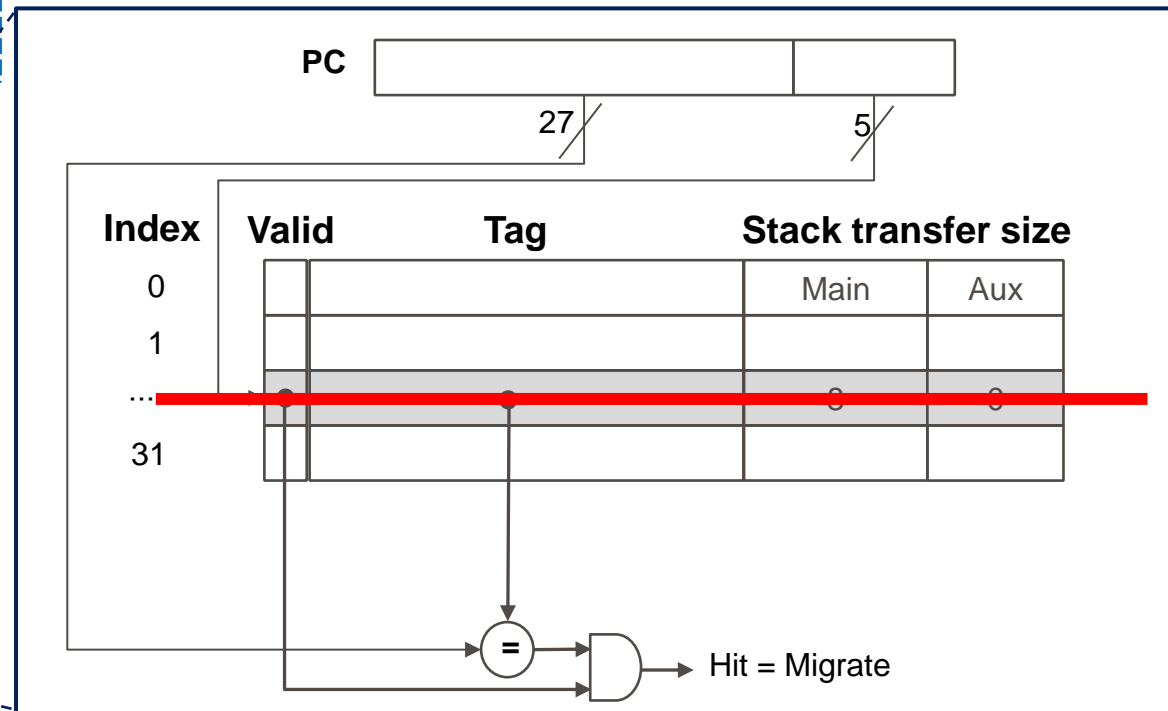


table indexed by seq. start PC



How much does this cost?

- **Arch. overhead**

- extra core context
- extra routers
- predictor (tiny)

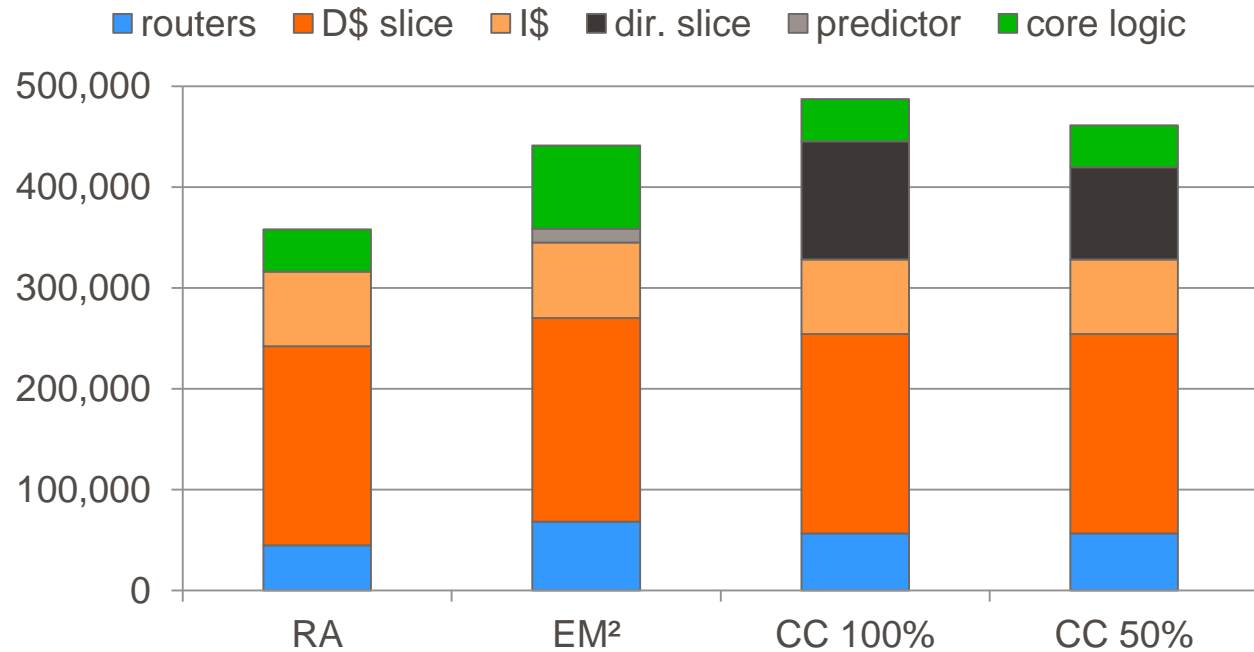
- **Area & leakage**

- total +23% area
- leakage similar

- **Dynamic power overhead**

- dominates but highly benchmark-dependent
- cache accesses same for EM and RA, **on-chip traffic** different

Single tile area contributions, 45nm [μm^2]

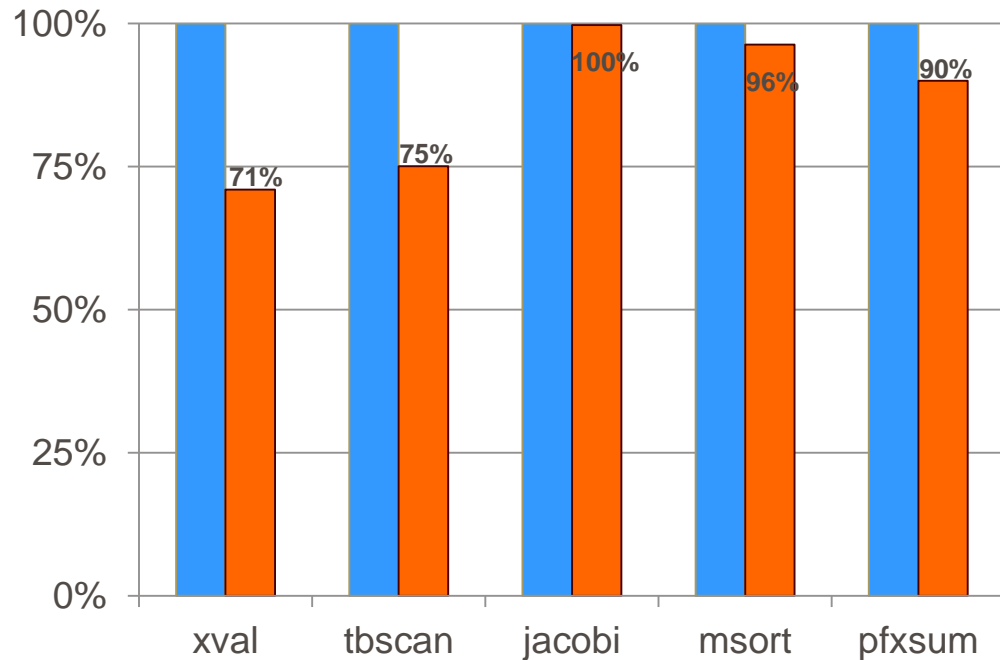


(synthesized ASIC cell area, 800MHz; CC areas estimated)

How well does it perform?

Completion time vs baseline

■ RA-only baseline ■ EM²

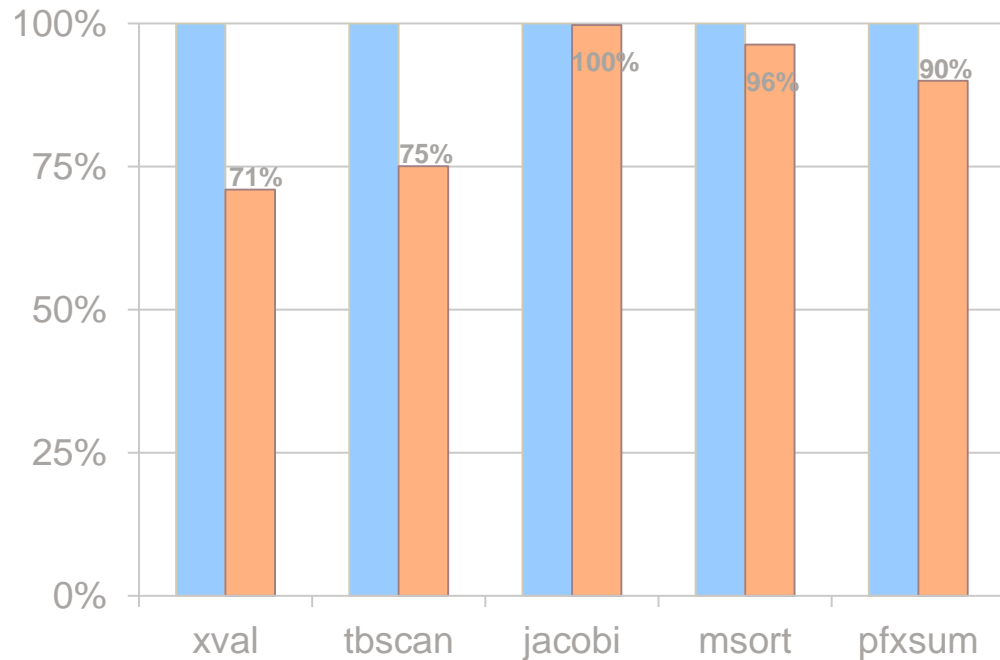


- **Benchmarks optimized for migration-friendly access patterns**
- **Significant improvements in performance (up to 25%)**

How well does it perform?

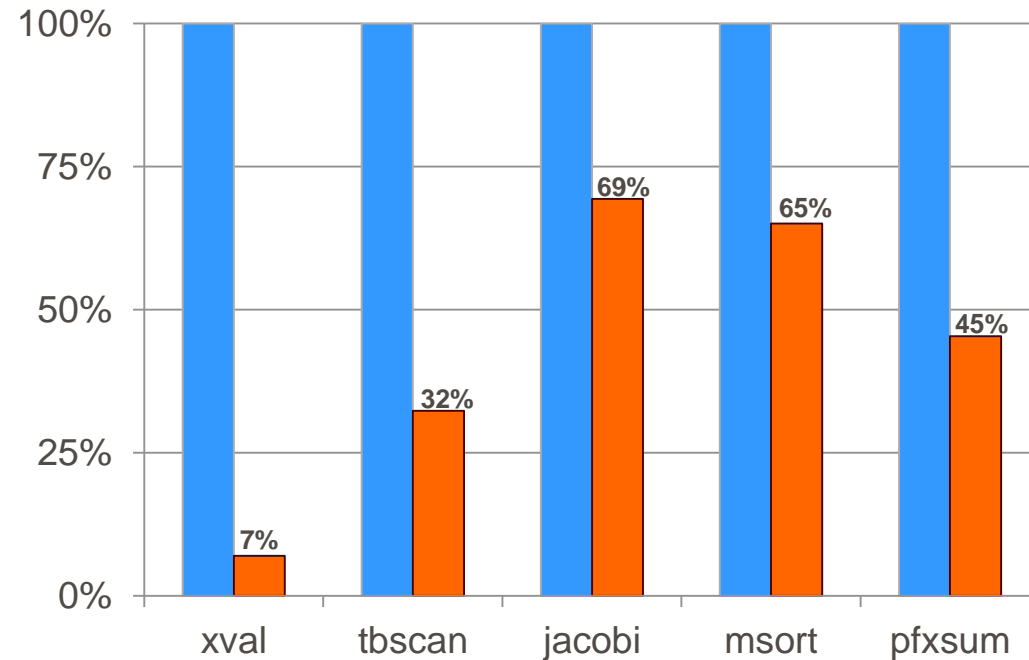
Completion time vs baseline

■ RA-only baseline ■ EM²



On-chip traffic vs baseline

■ RA-only baseline ■ EM²



- Benchmarks optimized for migration-friendly access patterns
- Significant improvements in performance (up to 25%)
- **Huge improvements in on-chip traffic reduction (up to 14x!) (→ significant reduction in dynamic power dissipation)**

Summary

- **Advantages**

- significantly reduces traffic on high-locality workloads up to **14x** reduction in traffic in some benchmarks
- simple to implement and verify (indep. of core count, no transient states)
- decentralized & trivially scalable (only # core ID bits, addr ↔ core mapping)

- **Challenges**

- workloads should be optimized with memory model in mind (like allocating data on cache line boundaries but more coarse-grained)
- automatically mapping allocation over cores not a trivial problem

- **Opportunities**

- fine-grained migration is an **enabling technology**
- since it's cheap and responsive, can be used for almost anything
- e.g., if only some cores have FPUs, migrate to access FPU