INRIA

Exploiting the cache capacity of a single-chip multicore processor with execution migration

Pierre Michaud

February 2004

Multicore processor

- A.k.a. chip-multiprocessor (CMP)
- Main motivation: run multiple tasks
- Integration on a single chip → **new possibilities**

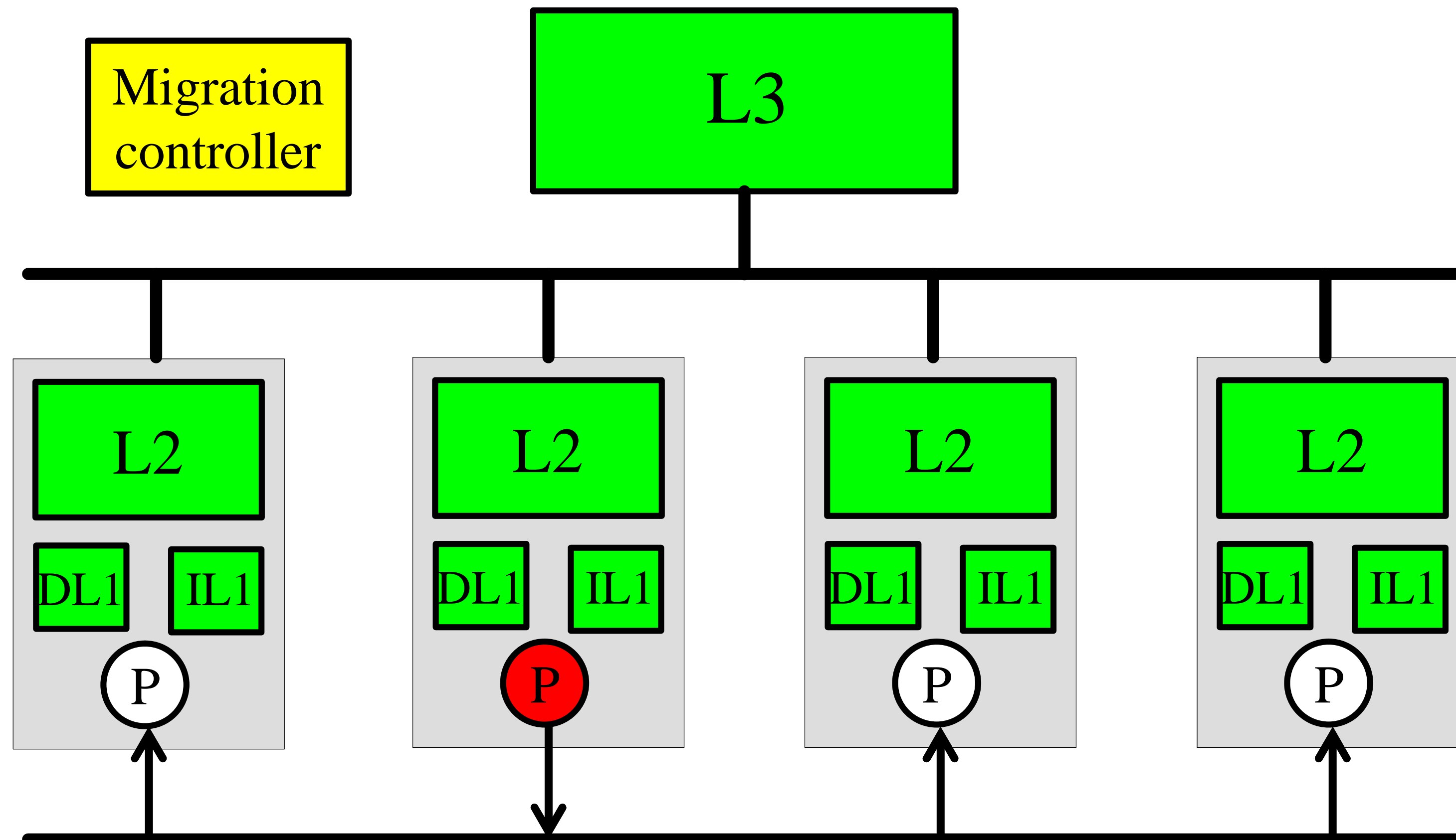
Plausible scenario for 2014

- Multicore = general purpose processor
- L2 cache per core, L3 shared between cores
- *Fast Sequential* mode
  - several mechanisms that allow a task to run faster when it is the only task running on a multicore
  - **Execution migration**: (perhaps) one of these mechanisms

Execution migration

- Assumption: **single task currently running**
- Other cores = (temporarily) wasted silicon
- Idea: **exploit the overall L2 capacity**. Problem: latency to remote L2 caches.
- *“If we cannot bring the data to the processor as fast as we would like, we could instead take the processor to the data”*
  - Garcia-Molina, Lipton, Valdes, “A Massive Memory Machine”, IEEE Transactions on Computers, May 1984.
- Execution migration = possibility to migrate the execution of a sequential task **quickly** from one core to another

Hardware



Active core broadcasts architectural updates to inactive cores

Trading L2 misses for migrations

- Why the L2 and not the L1 ?
  - Cost of a migration: probably several tens of cycles
  - L2 miss can take tens of cycles, easier to trade L2 misses for migrations
- Want to minimize the number of lines that are replicated
  - would like to "see" a bigger L2
- Want to keep the frequency of migrations under control

“Splittability”

- 2 cores
- Example: random accesses
  - Assign one half of the working set to a core, the other half to the other core
  - Frequency of migrations = ½ ➔ working-set not “splittable”
- Example: circular accesses
  - 0,1,2,…,99, 0,1,2,…,99, 0,1,2,…,99…
  - Assign [0,..49] to a core, [50,..99] to the other core
  - Frequency of migrations = 1/50 ➔ working set “splittable”

Targeted applications

- Working set does not fit in single L2 but fits in overall L2
- Working set **splittable**
- Enough working set **reuse** for learning pattern of requests

- "Bad" applications
  - Random accesses, no splittability
  - Or not enough working-set reuse
    - Example: O(N log N) algorithm
  - ➔ nothing to expect from migrations, try to limit them

Graph partitioning problem

- Node = static cache line
- Edge between nodes A and B labeled with frequency of occurrence of line A immediately followed by line B

- Classic min-cut problem
  - split the working-set in two equally sized subsets so that the frequency of transitions between subsets is minimum

- Our problem:
  - Subset sizes need not be strictly equal, rough balance is OK
  - Frequency of transitions need not be minimum, but small enough
  - Heuristic must be **simple enough to be implemented in hardware**

The affinity algorithm

Affinity =signed integer $A_e(t)$ associated with each cache line e at time t

Working set bipartitioning according to sign of affinity

N fixed, set R = N lines most recently requested

$$A_R(t) = \sum_{e \in R} A_e(t)$$

For all e,

$$A_e(t+1) = \begin{cases} A_e(t) + \text{sign}(A_R(t)) & \text{if } e \in R \\ A_e(t) - \text{sign}(A_R(t)) & \text{if } e \notin R \end{cases}$$

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Local positive feeback

Groups of lines that are often referenced together tend to have the same sign of affinity

Global negative feeback

N much smaller than working set

➔ Lines spend more time out of R than in R

If a majority of lines have positive affinity, affinity decreases

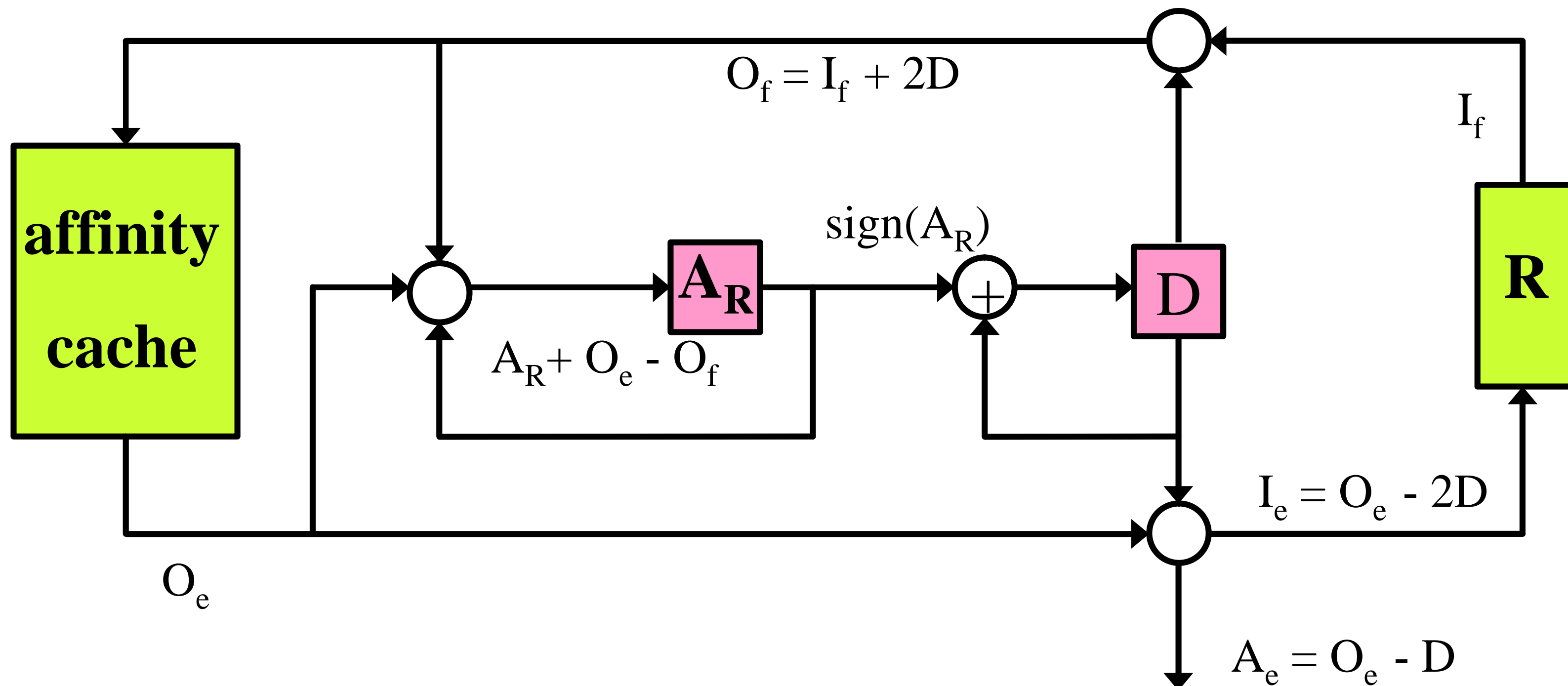
➔ Converges toward balanced partition

2-way splitting mechanism

define $D(t+1) = D(t) + sign(A_R(t))$

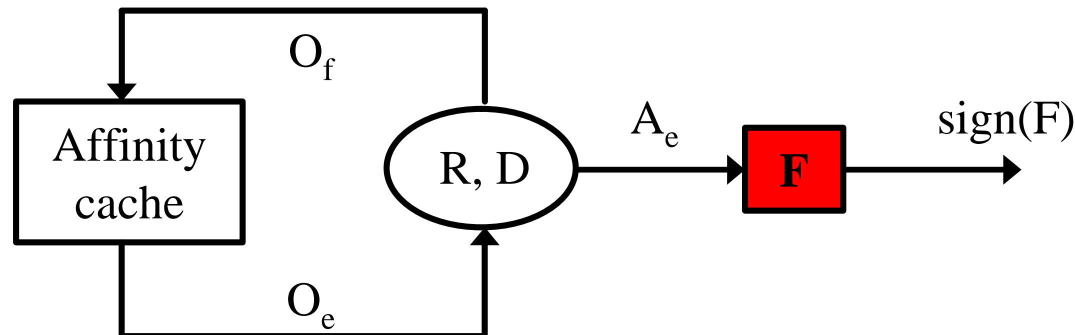
$I_e(t) = A_e(t) - D(t)$

$O_e(t) = A_e(t) + D(t)$



Filter

- Random accesses: we want as few migrations as possible
- Filter = up-down saturating counter
  - $F(t+1) = F(t) + A_e(t)$
- Partition the working set according to the sign of F



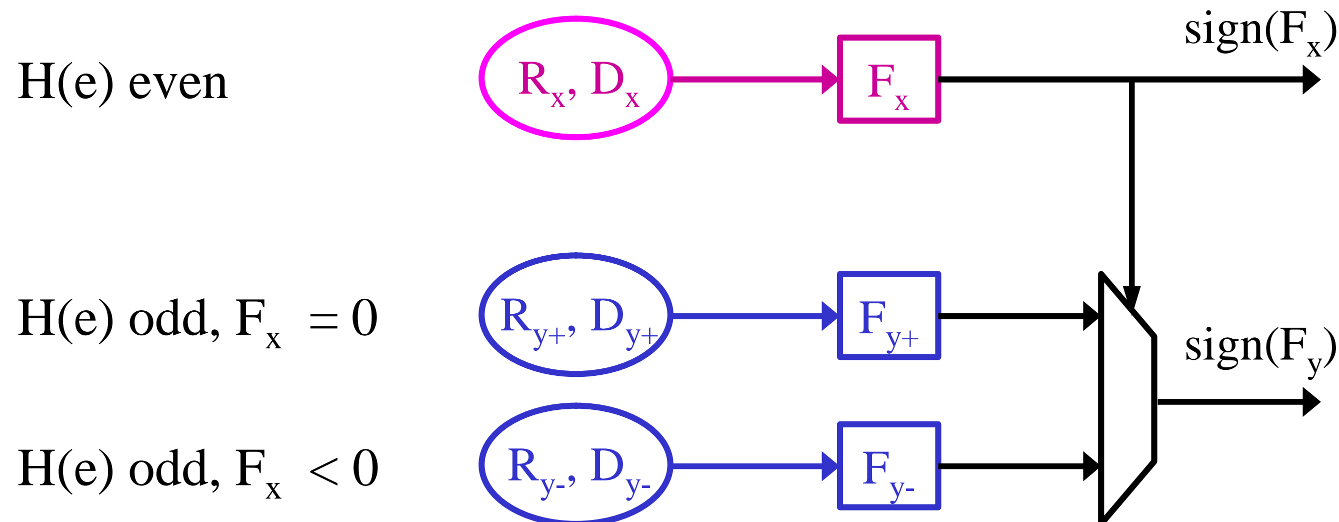
Working-set sampling

- Size of affinity cache proportional to overall L2 capacity
- Example
  - 2 cores
  - 512 KB L2 cache per core, 64-byte lines
  - overall L2 = 1 MB
  - ➔ 16k lines ➔ 16k-entry affinity cache
- Can reduce affinity cache size by sampling the working-set
  - hash function H(e) = e mod 31
  - if H(e)<8, allocate entry in affinity cache and update affinities
  - otherwise, just read filter F
  - ➔ 4k-entry affinity cache

4-way working-set splitting

Split in 2 subsets, then split each subset

No need for second affinity cache, just use sampling



Experiments

- Benchmarks: subset of SPEC2000 and Olden
  - those with significant L1 misses (instructions and/or data)
  - 18 benchmarks
- Is "splittability" rare or not ?
- Can I keep the migration frequency under control ?
- How many L2 misses can I remove with one migration ?

“Splittability” is the common case

- Simulated four LRU stacks and a 4-way splitting mechanism
  - Measured per-stack reuse distance and frequency of transitions between stacks
  - Decreased reuse distances implies “splittability”
- **“Splittability”observed on a majority of benchmarks**
  - More or less pronounced depending on the application
  - 4 benchmarks on 18 with no apparent “splittability” at all
- Circular accesses frequent in L1 miss requests
  - Not necessarily constant stride (Olden)

Migrations are kept under control

- Filter is efficient
- But also
  - Do not update the filter upon a L2 hit
  - Set $A_e = 0$ upon miss in affinity cache ➔ filter value not changed
  - ➔ For working sets fitting in single L2, or not fitting in overall L2, one migration for at least hundreds of thousands of instructions
- **Migrations frequent only when it can reduce L2 misses**

How many L2 misses removed ?

- Simulated four 512-Kbyte L2 caches
- **6 benchmarks** on 18 see an important reduction of L2 misses
- 1 migration removes X misses
  - health: X=2500
  - 179.art : X=1800
  - em3d : X=1700
  - 256.bzip2 : X=1300
  - 188.ammp: X=500
  - 181.mcf : X=60
    - performance gain if migration penalty less than 60 times L2 miss penalty

Conclusion

- Potential for improving significantly the performance of **some** applications in Fast Sequential mode

- Main hardware cost = register write broadcast bandwidth
  - Affinity cache = secondary cost

- Possible ways to decrease bandwidth requirement
  - Register write cache
  - Prepare migration when filter absolute value falls below threshold

Open questions…

- How to combine with prefetching ?
- Orthogonal to prefetching : to what extent ?
- Is there more to "splittability" than circular accesses ?
  - In theory yes, in practice not sure
    - Can programmers take advantage of execution migration ?
- Other reasons why we would want to migrate ?
  - Temperature ?