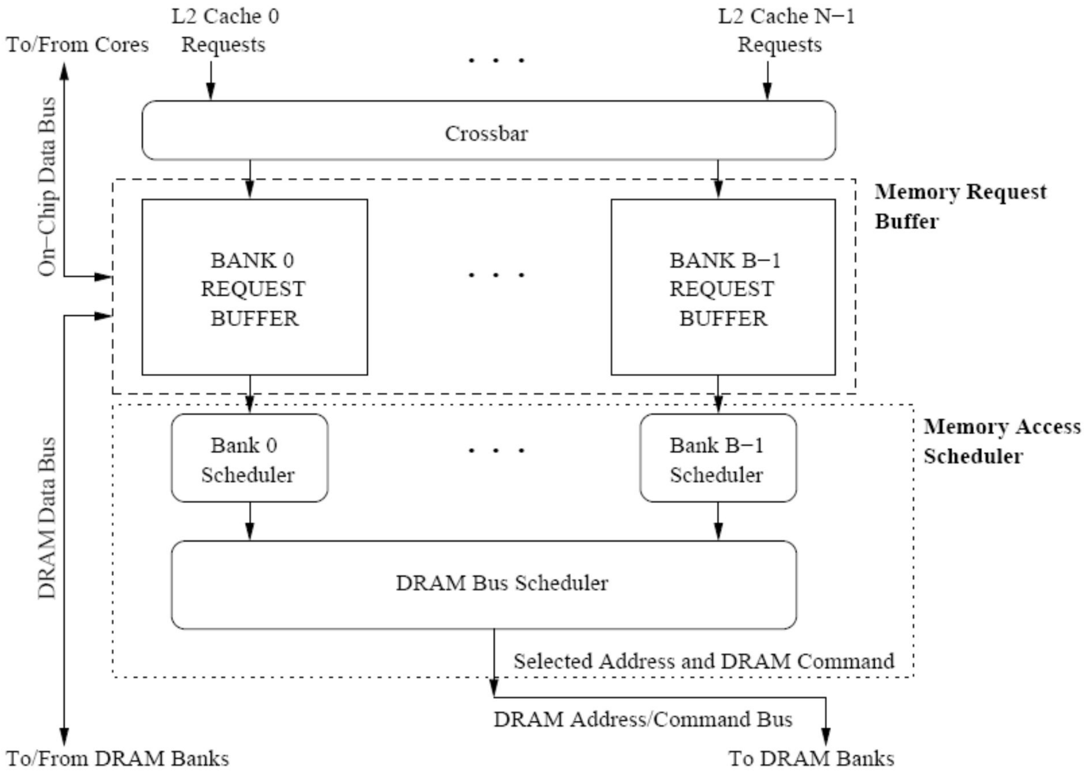
Accelerating Dependent Cache Misses with an Enhanced Memory Controller

Yunhao Lan, Ying Meng

Memory Controller Review

- Multi-core LLC miss requests
- On-chip interconnect
- DRAM bus scheduler
- #cores increases contention



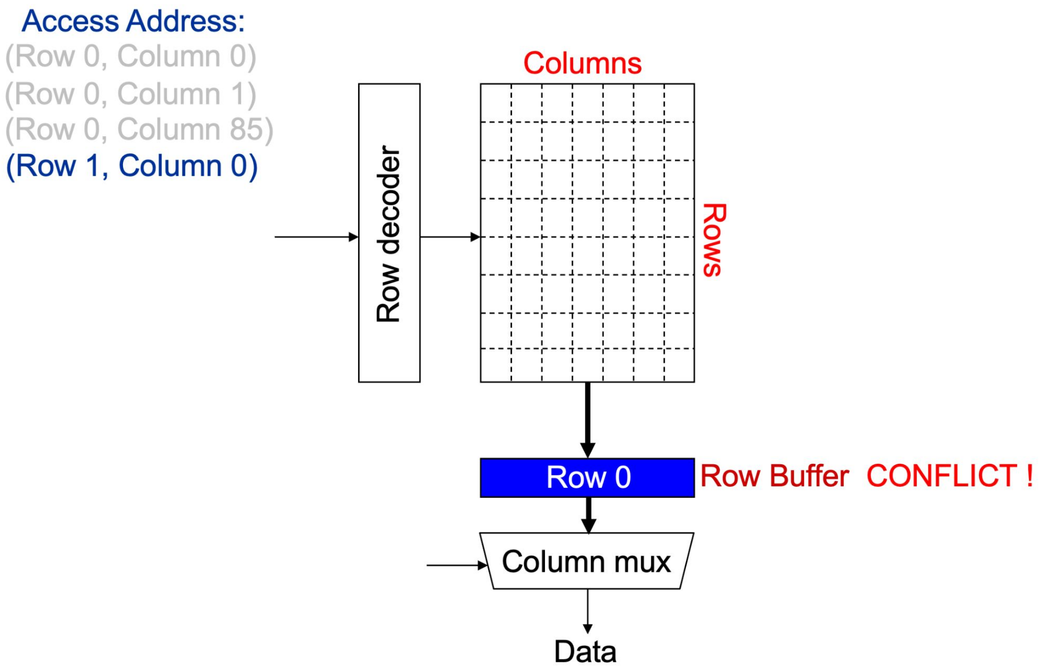
DRAM

FR-FCFS (first ready, first come first served) Scheduling Policy

1. Row-hit first

2. Oldest first

Goal: Maximize row buffer hit rate → maximize DRAM throughput



Dependent Cache Miss

- Result in cache miss & address depend on data from a prior cache miss
- Pointer-chasing (create linked-list)
- LLC miss → DRAM → core compute address → DRAM …

```
struct Node {
    int data;
    Node* next;
};
```

```
for (int i = 0; i < size - 1; i++) {
    nodes[i]->next = nodes[i + 1];
}
```

Accelerating Dependent Cache Misses with an Enhanced Memory Controller

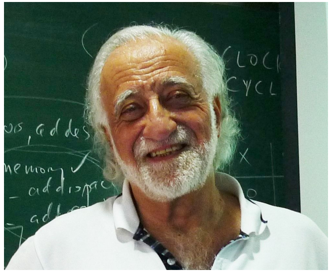
Milad Hashemi: PhD at UT Austin, now Research Scientist at Google

Khubaib: PhD at UT Austin, now at CPU Design Group in Apple Austin

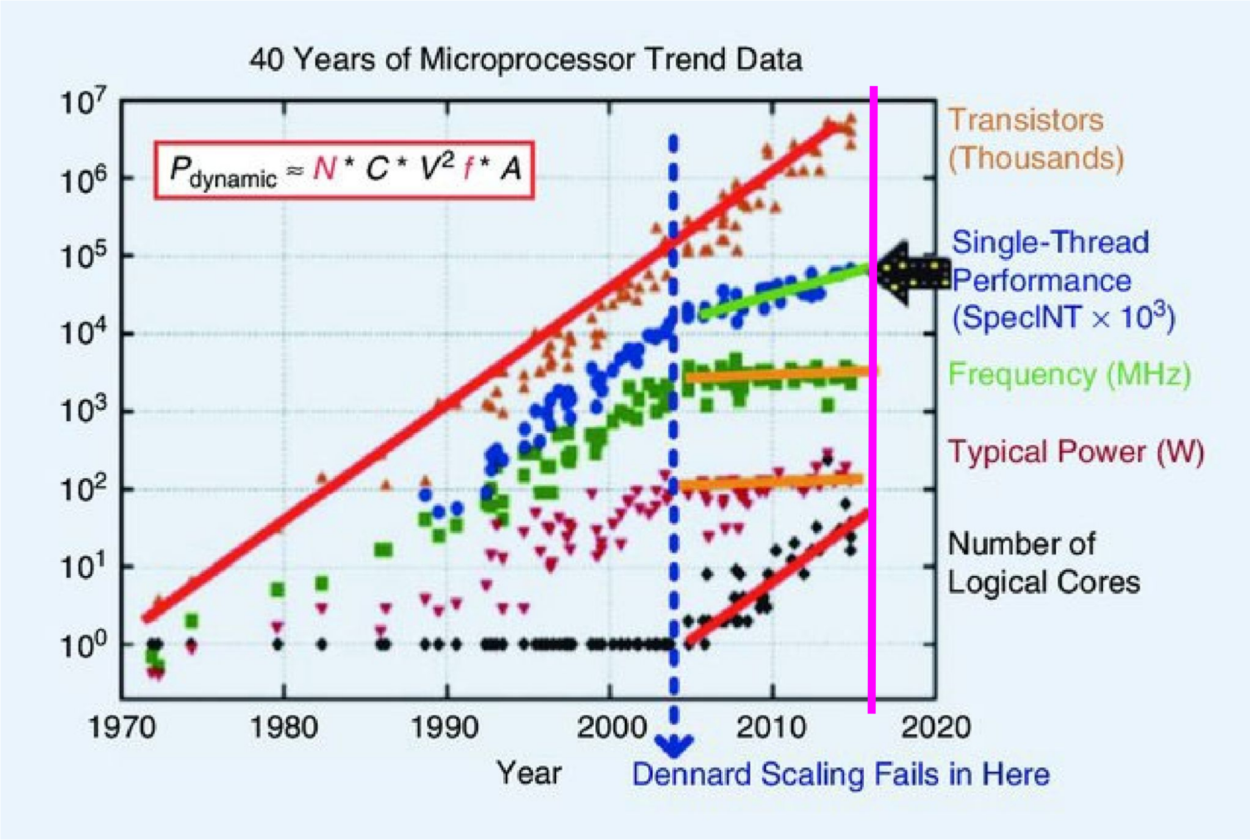
Eiman Ebrahimi: PhD at UT Austin, worked at Nvidia, now CEO at Protopia AI

Onur Mutlu: PhD at UT Austin, CMU prof, now Professor at ETH

Yale N. Patt: Professor at UT Austin, PhD advisor for all authors, lead HPS Research Group (High Performance Systems)



Scaling at 2016



Motivation

On-chip contention is a **substantial** portion of memory access latency in multi-core systems

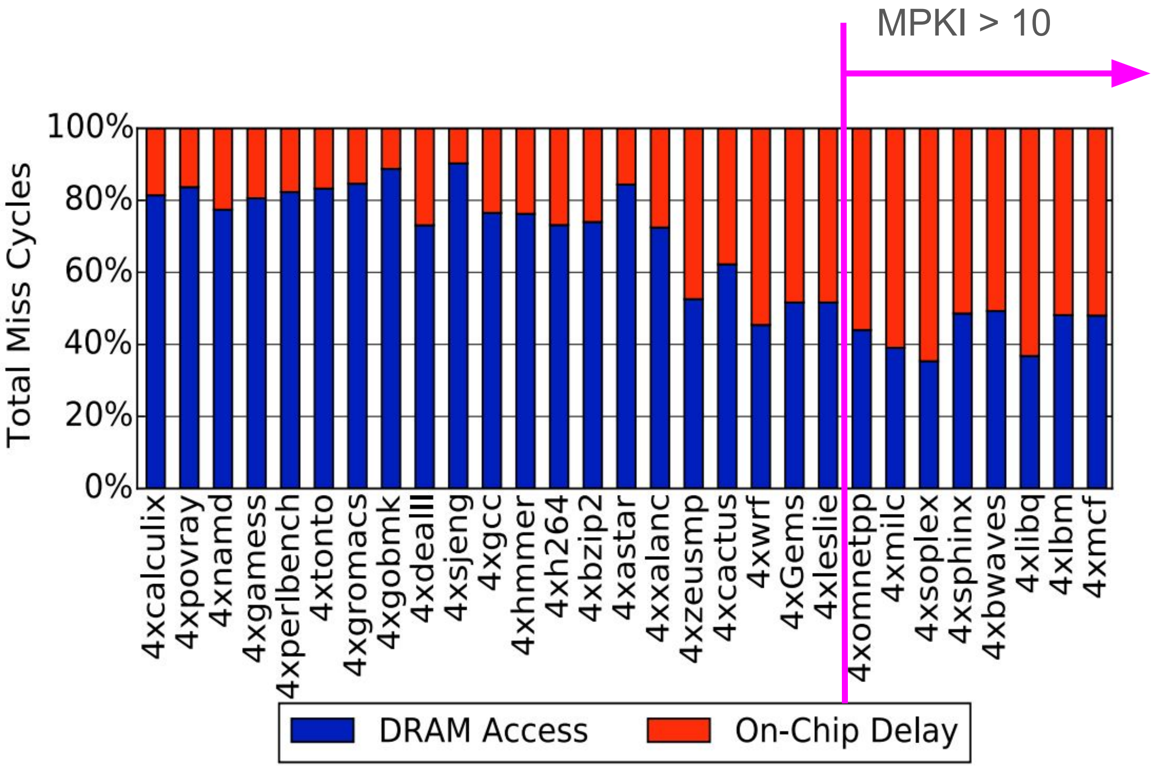


Figure 1: Breakdown of total memory access latency into DRAM latency and on-chip delay.

Motivation

Dependent cache misses are latency-critical operations that are **hard** to **prefetch**.

Prefetch ~**20%** of dependent miss

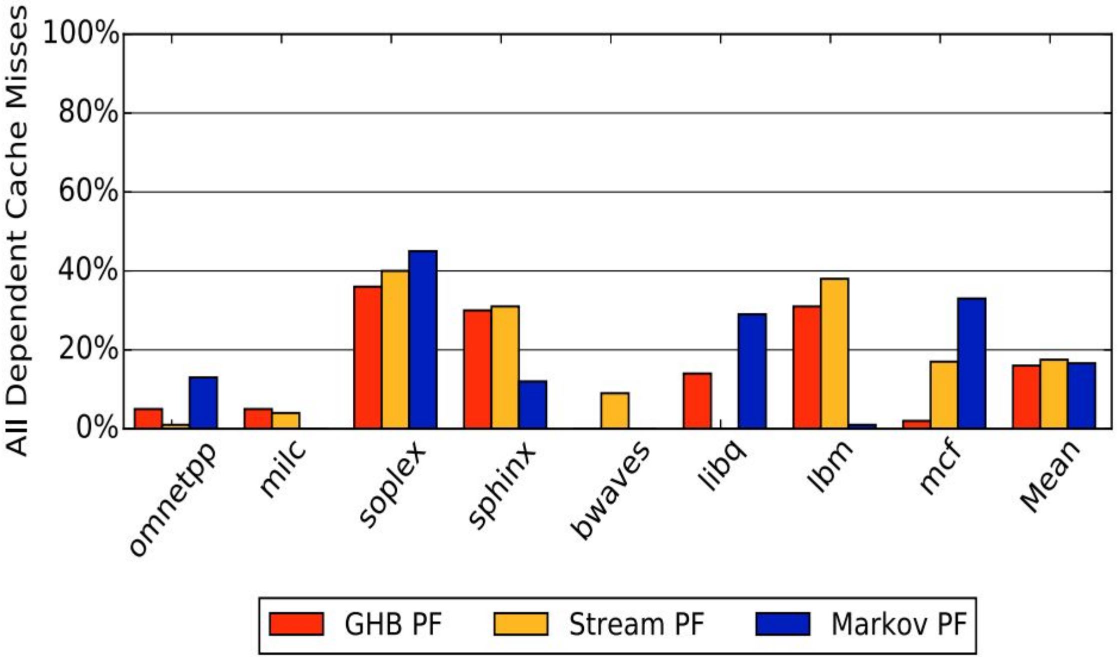


Figure 3: Percentage of dependent cache misses that are prefetched with a GHB, stream, and Markov prefetcher.

Motivation

The number of instructions between a source cache miss and a dependent cache miss is often **small**

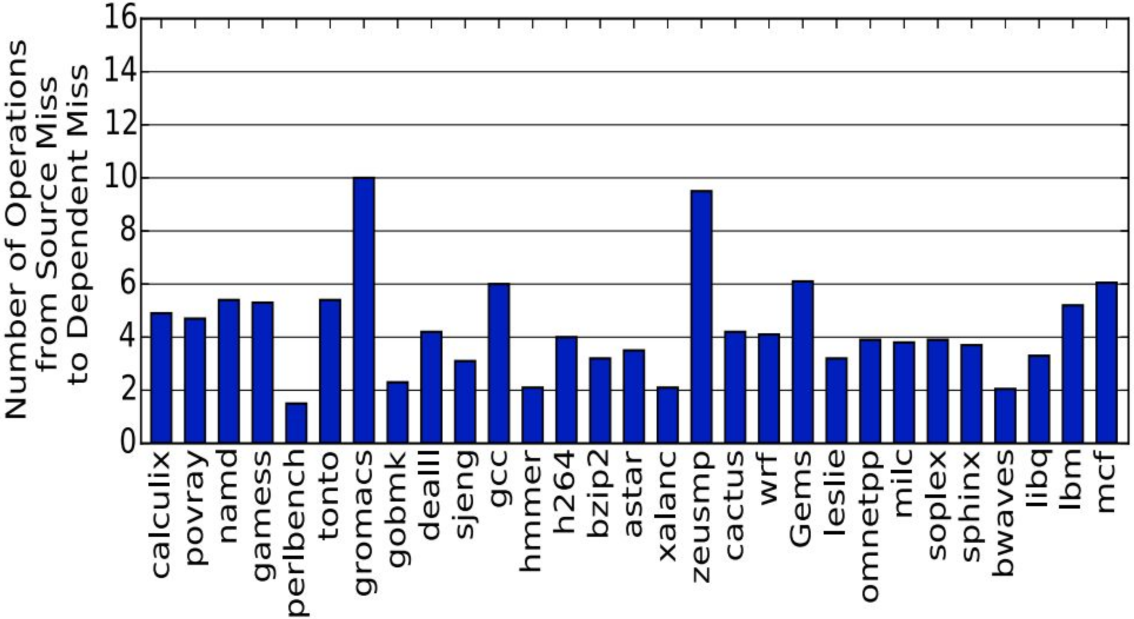
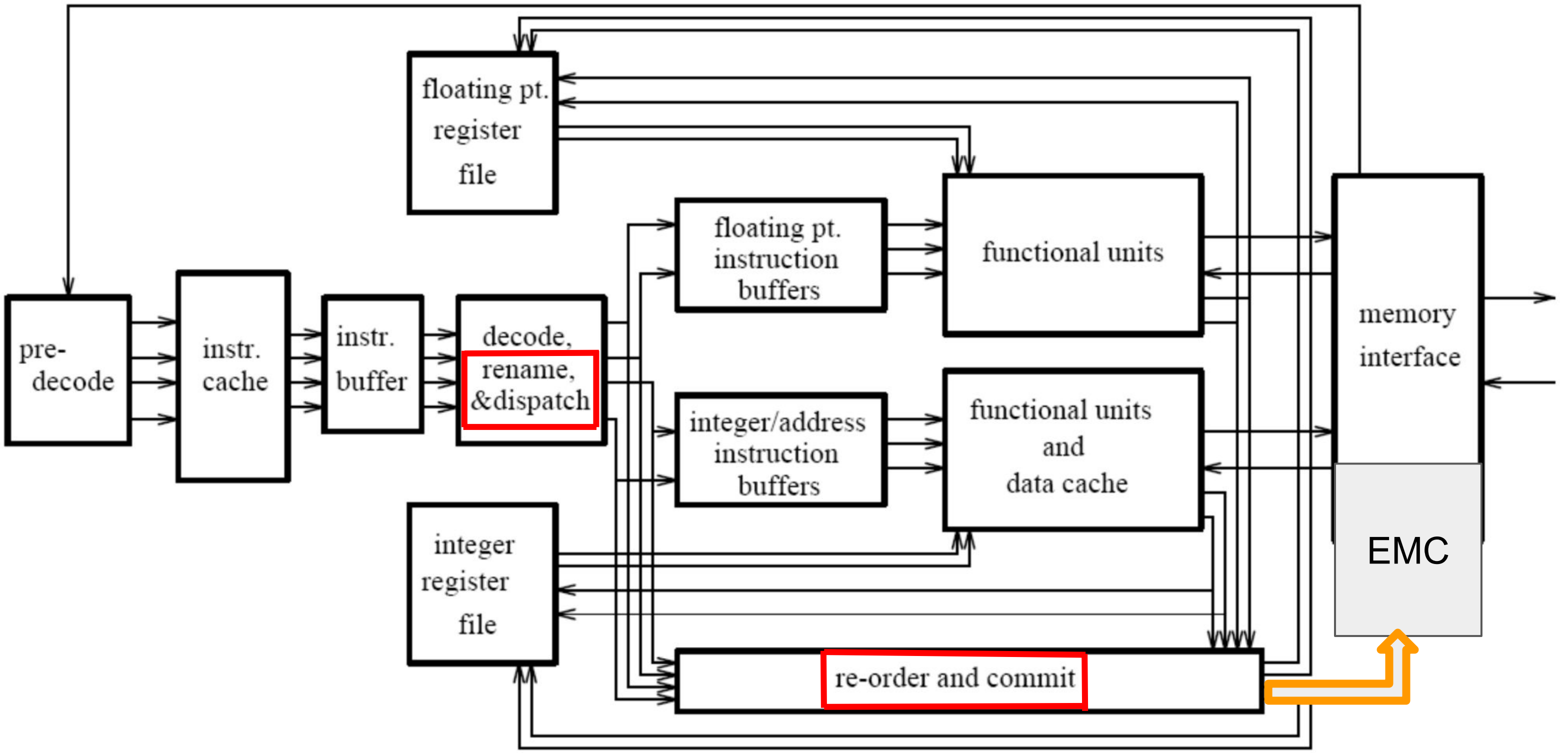


Figure 6: Average number of dependent operations between a source miss and dependent miss.

Related Work

- Independent cache misses
  - Correlation prefetching: stream/stride prefetcher & temporal prefetcher
    - oblivious to control flow, bandwidth limited
  - Content-directed prefetching
    - greedily prefetches by dereferencing values that could be memory addresses
  - Runahead execution & continual flow pipelines
    - execute ahead of the demand access stream, generating independent cache misses
- Enhancing memory controller
  - Move computation close to memory
  - 3D-stacked DRAM with computation
- This paper
  - Target dependent cache misses
  - Add compute capability to memory controller

Out-of-Order Processor Review



Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proc. IEEE, Dec. 1995.

Implementation

EMC **sits in the memory controller**, close to DRAM.

What it does:

- Offloads dependent instructions from the core.
- Reduces memory latency.
- Issues requests **faster** than the core.

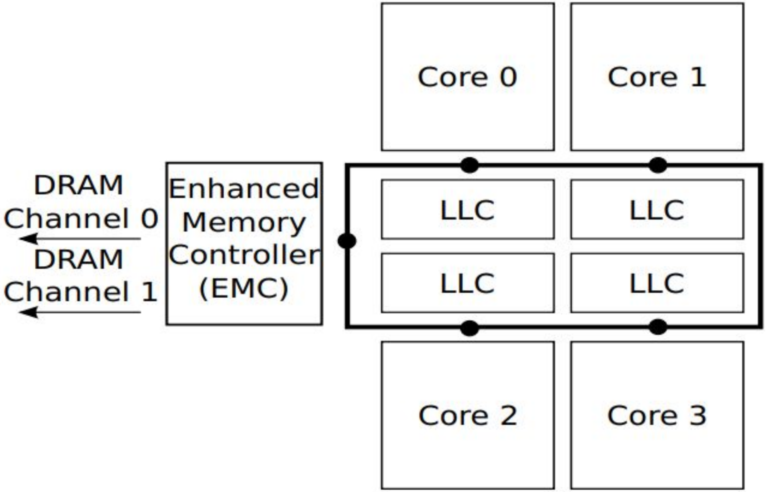


Figure 7: A high level view of a quad-core processor with an Enhanced Memory Controller. Each core has a ring stop, denoted by a dot, which is also connected to a slice of the shared last level cache.

Implementation

EMC **has just enough hardware** to process dependent instructions.

Key parts:

- **Front-end:** Holds small buffers for instructions.
- **Back-end:** Two ALUs and a small data cache.
- **No fetch/decode logic** (saves area & power).

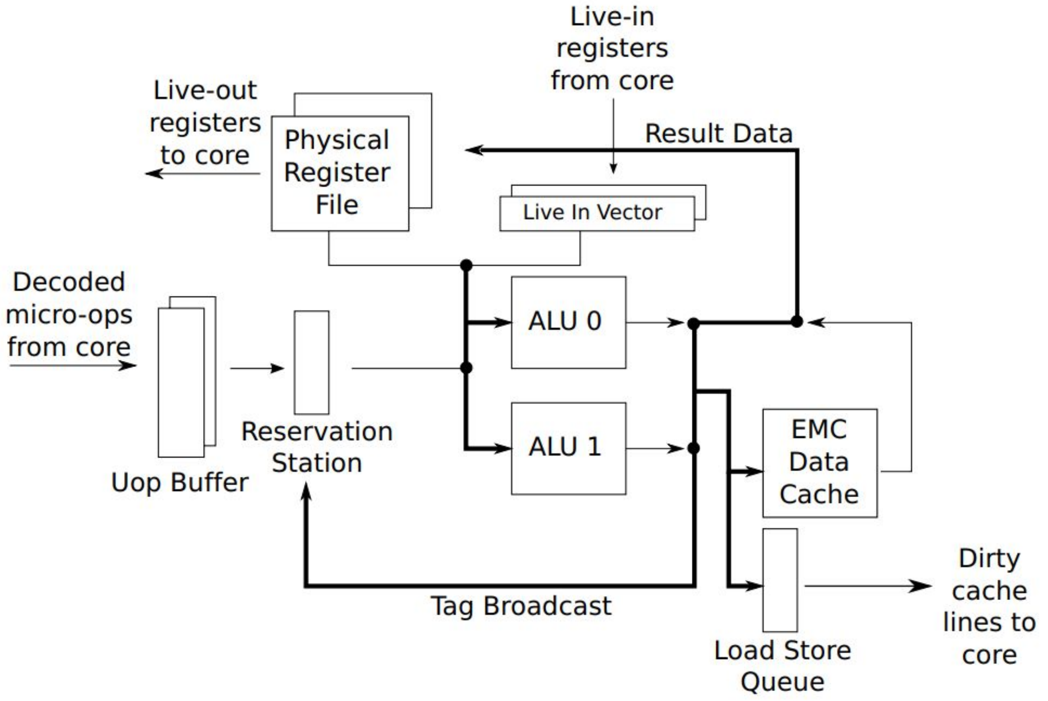


Figure 8: The microarchitecture of the EMC.

Generate Dependence Chain

- Full-window ROB stall due to an LLC miss blocking retirement
- 3-bit saturating counter to determine if a dependent cache miss is likely
- Dataflow walk to track dependencies until end or micro-ops reaches 16 (the size of EMC register file)

Algorithm 1: Dependence Chain Generation

```
//Process the source uop at ROB full stall;
Allocate EPR for destination CPR of uop in RRT;
Add uop to chain and broadcast destination CPR tag;
for each dependent uop do
    if uop Allowed and (all source CPRs ready or in RRT) then
        //Prepare the dependent uop to send to EMC;
        for each source operand do
            if CPR ready then
                Read data from PRF into live-in vector;
            else
                EPR = RRT[CPR];
            end
        end
        Allocate EPR for destination CPR in RRT;
        Add uop to chain and broadcast destination CPR tag;
        if Total uops in Chain == 16 then
            break;
        end
    end
end
Send filtered chain of uops and live-in data to EMC;
```

Figure 10: Dependence chain generation. CPR: Core Physical Register. EPR: EMC Physical Register. RRT: Register Remapping Table.

Cycle 1

ROB	
Core instruction	**EMC micro-ops**
MEM_LD C8 -> C1	MEM_LD C8 -> E0
(independent instruction)	
MOV C1 -> C9	MOV E0 -> E1
(independent instruction)	
ADD C9, 0x18 -> C12	
MEM_LD C12 -> C10	
ADD C10, C3 -> C16	
MEM_LD C16 -> C19	

RRT					
C1	C9				
E0	E1				

Live-In					

Cycle 2

ROB	
Core instruction	**EMC micro-ops**
MEM_LD C8 -> C1	MEM_LD C8 -> E0
(independent instruction)	
MOV C1 -> C9	MOV E0 -> E1
(independent instruction)	
ADD C9, 0x18 -> C12	ADD E1, L0 -> E2
MEM_LD C12 -> C10	
ADD C10, C3 -> C16	
MEM_LD C16 -> C19	

RRT					
C1	C9	C12			
E0	E1	E2			

Live-In					
0x18					

Cycle 3

ROB	
Core instruction	**EMC micro-ops**
MEM_LD C8 -> C1	MEM_LD C8 -> E0
(independent instruction)	
MOV C1 -> C9	MOV E0 -> E1
(independent instruction)	
ADD C9, 0x18 -> C12	ADD E1, L0 -> E2
MEM_LD C12 -> C10	MEM_LD E2 -> E3
ADD C10, C3 -> C16	
MEM_LD C16 -> C19	

RRT					
C1	C9	C12	C10		
E0	E1	E2	E3		

Live-In					
0x18					

Cycle 4

ROB	
Core instruction	**EMC micro-ops**
MEM_LD C8 -> C1	MEM_LD C8 -> E0
(independent instruction)	
MOV C1 -> C9	MOV E0 -> E1
(independent instruction)	
ADD C9, 0x18 -> C12	ADD E1, L0 -> E2
MEM_LD C12 -> C10	MEM_LD E2 -> E3
ADD C10, C3 -> C16	ADD E3, L1 -> E4
MEM_LD C16 -> C19	

RRT					
C1	C9	C12	C10	C16	
E0	E1	E2	E3	E4	

Live-In					
0x18	C3				

Cycle 5

ROB	
Core instruction	**EMC micro-ops**
MEM_LD C8 -> C1	MEM_LD C8 -> E0
(independent instruction)	
MOV C1 -> C9	MOV E0 -> E1
(independent instruction)	
ADD C9, 0x18 -> C12	ADD E1, L0 -> E2
MEM_LD C12 -> C10	MEM_LD E2 -> E3
ADD C10, C3 -> C16	ADD E3, L1 -> E4
MEM_LD C16 -> C19	MEM_LD E4 -> E5

RRT					
C1	C9	C12	C10	C16	C19
E0	E1	E2	E3	E4	E5

Live-In					
0x18	C3				

EMC Execution

- Dependence chain (micro-ops) + live-in → live-outs
- Core sends branch information, cannot resolve mispredicted branch
- Loads to EMC cache, predicts if cache miss by 3-bit counters
- In-order retirements in core

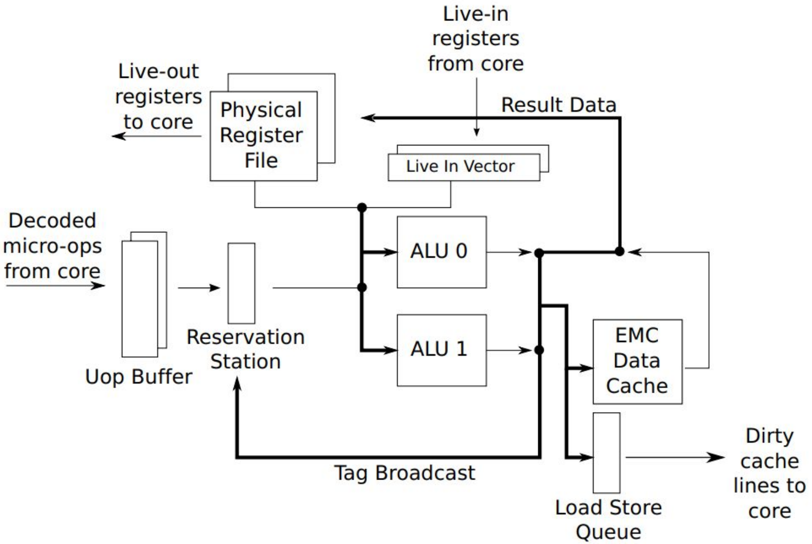


Figure 8: The microarchitecture of the EMC.

Methodology

H1	bwaves+lbm+milc+omnetpp
H2	soplex+omnetpp+bwaves+libq
H3	sphinx3+mcf+omnetpp+milc
H4	mcf+sphinx3+soplex+libq
H5	lbm+mcf+libq+bwaves
H6	lbm+soplex+mcf+milc
H7	bwaves+libq+sphinx3+omnetpp
H8	omnetpp+soplex+mcf+bwaves
H9	lbm+mcf+libq+soplex
H10	libq+bwaves+soplex+omentpp

Table 3: Quad-Core workloads.

High Intensity (MPKI >= 10)	omnetpp, milc, soplex, sphinx3, bwaves, libquantum, lbm, mcf
Low Intensity (MPKI <10)	calculix, povray, namd, gamess, perlbench, tonto, gromacs, gobmk, dealII, sjeng, gcc, hmmer, h264ref, bzip2, astar, xalancbmk, zeusmp, cactusADM, wrf, GemsFDTD, leslie3d

Table 2: *SPEC CPU2006* classification by memory intensity.

Results

Performance Improvement:

- Works well with memory-intensive workloads.
- Best with prefetchers
- Up to **15%** speedup over baseline, **9–11%** over prefetchers.

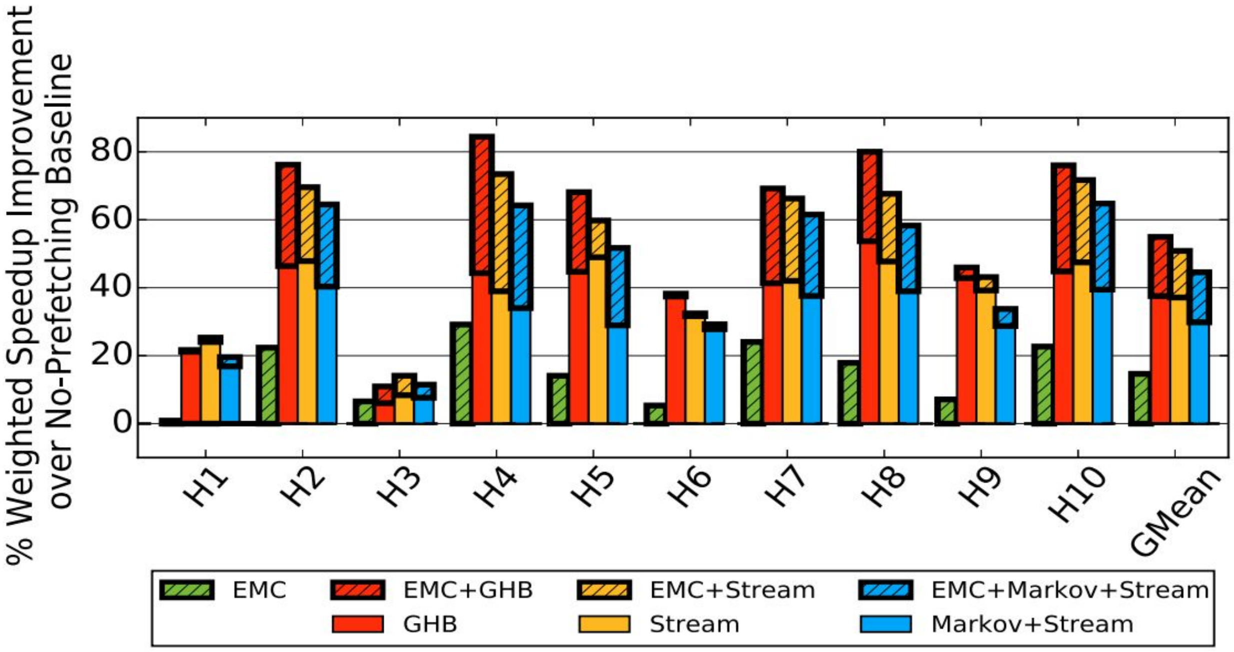


Figure 12: Quad-Core performance for workloads H1-H10.

Results

Latency Reduction:

- EMC reduces cache miss latency by **~20%**.
- **Why?** EMC **bypasses** on-chip delays.
- Faster dependent cache misses = faster execution.

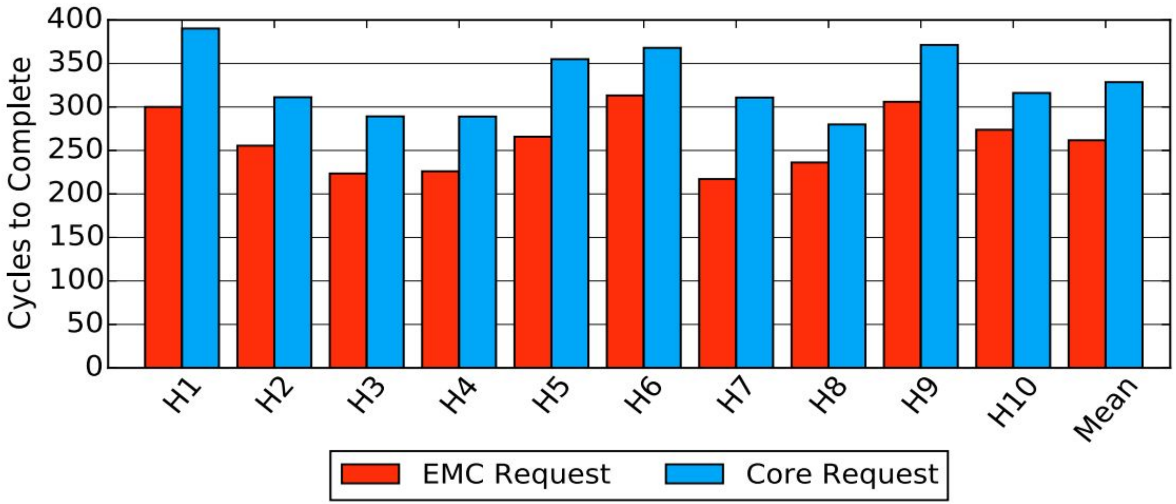


Figure 18: Latency observed by an LLC miss generated by the EMC vs. an LLC miss generated by the core for H1-H10.

Results

DRAM Contention:

- EMC **lowers row-buffer conflicts** in DRAM (**~19%**).
- **Why?** It issues dependent requests **faster & in groups**.
- Explore the **first-ready-first-serve** scheduling policy
- **Result:** More row-buffer hits, fewer delays.

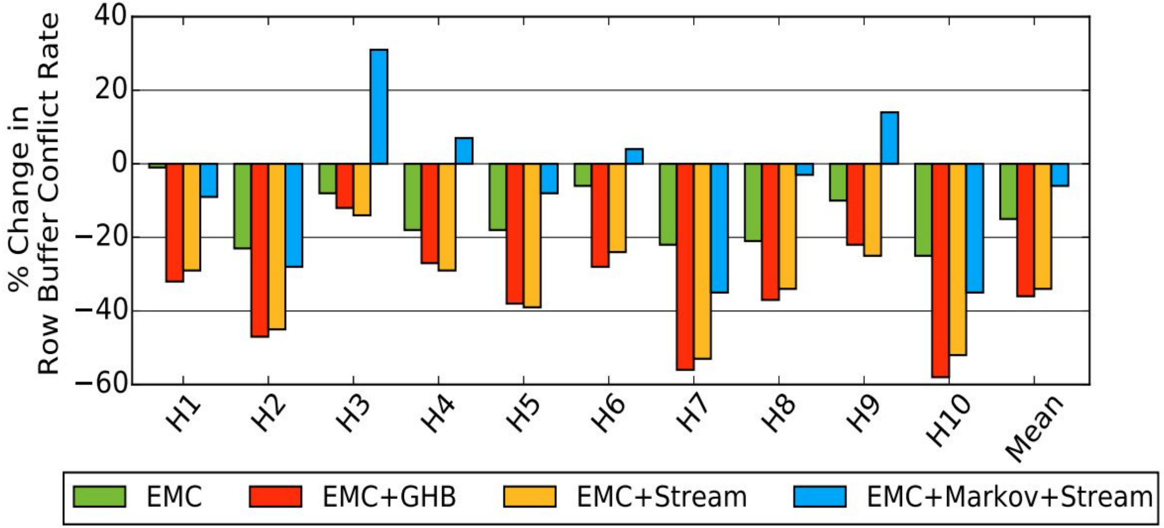


Figure 16: Change in row-buffer conflict rate with the EMC over a no-prefetching baseline.

Results

Energy Efficiency:

- Prefetching alone **increases energy** use (useless prefetches).
- EMC lowers energy by **11%** (*less execution time + reduced row-buffer conflict rate*)
- EMC + prefetching = better efficiency.

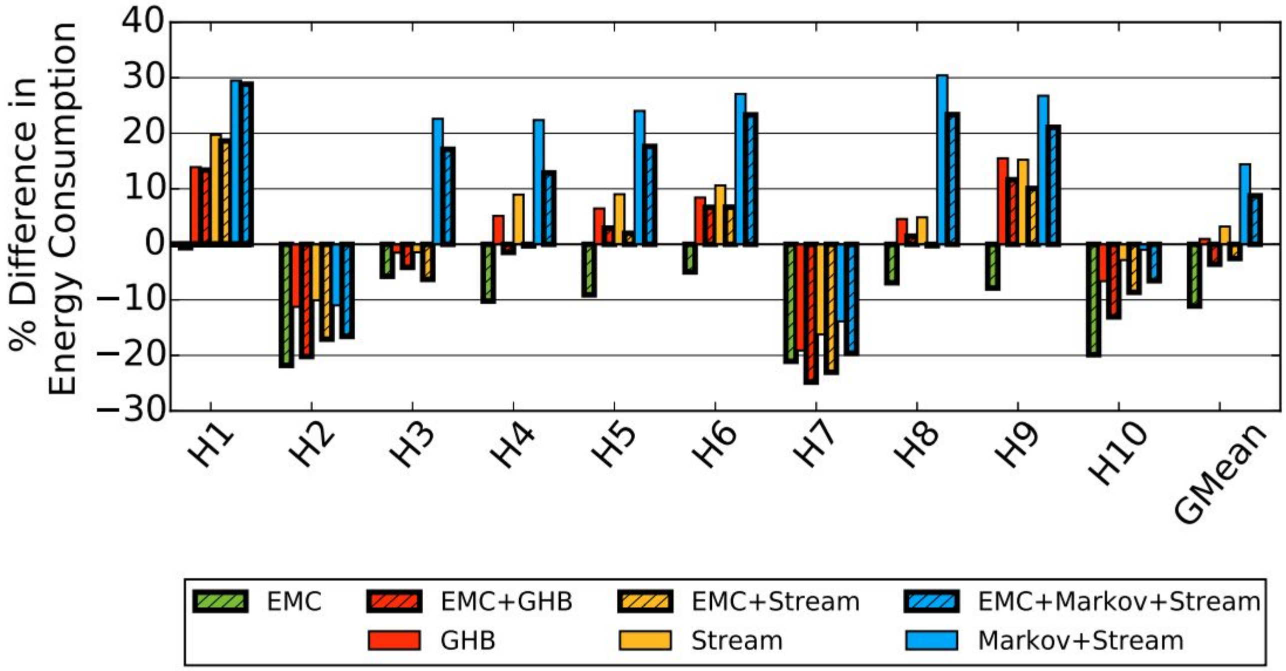


Figure 23: Energy consumption for workloads H1-H10.

Thoughts

- Pros
  - Novel idea to reduce dependent cache miss delay
  - Bypass on-chip delay
  - Increase DRAM row buffer hit
- Cons
  - Performance improvement only on high memory intensity workloads (MPKI >= 10)
  - Redundancy cycles in generating dependence chain
  - Large area overhead (2% of chip area)
  - No evidence used by commercial chip?