Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks

A. Boroumand et al. 2018

speaker: Julian Pszczołowski
Seminar: Advances in Computer Systems
University of Wrocław

Introduction

- This paper focuses on *consumer* devices
- Smartphones, tablets, wearable devices have become increasingly ubiquitous in recent years
- There were 2.3 billion smartphone users in 2017, and 1.2 billion tablet users in 2016

Energy consumption

- A first-class concern for consumer devices
- Devices have many power-hungry components such as powerful CPU, GPU, special-purpose accelerators, sensors, high-resolution screen
- Performance requirements increase every year to support things like 4K video, VR, AR, ...
- Lithium-ion battery capacity has only doubled in the last 20 years

Identifying sources of energy consumption

The authors analysed the most popular Google consumer workloads:

- Google Chrome
- TensorFlow Mobile (used by Google Translate, Google Now, and Google Photos, …)
- video playback and capture using VP9 codec (used by YouTube, Skype, Google Hangouts, …)

Some results of the analysis (teaser)

- Data movement between the main memory system and computation units is a major contributor to the total system energy
- While scrolling through a Google Docs web page, moving data between memory and computation units causes **77%** of the total system energy consumption
- On average: **63%** of the total energy is consumed by data movement
- Notice: Wi-Fi turned off, the lowest display brightness used

How to reduce the energy consumption

- Let’s execute data-movement-heavy portions of the application close to the data!
- Recent advances in 3D-stacked memory technology have enabled processing-in-memory (PIM), a.k.a near-data processing
- 3D-stacked architectures include a dedicated logic layer (with high-bandwidth low-latency connectivity to DRAM layers)
- Challenges:
  - area for PIM is limited
  - additional energy needed by PIM
  - additional cost of the device

Let’s switch papers for a while...

…to understand the processing-in-memory (PIM) better.

Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. S. Ghose et al. 2018.

PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. J. Ahn et al. 2015.

Problems with DRAM

- Performance improvements from DRAM technology scaling lag behind the improvements from logic technology scaling
- DRAM-based main memory is increasingly becoming a larger bottleneck in terms of performance and energy consumption
- Data stored within DRAM must be moved into the CPU before any computation can take place

Problems with PIM

- No low-latency access to some CPU structures:
  - translation lookaside buffer (TLB),
  - page table walker,
  - cache coherence mechanisms,
  - etc.
- Forcing PIM processing logic to send queries to the CPU is very inefficient

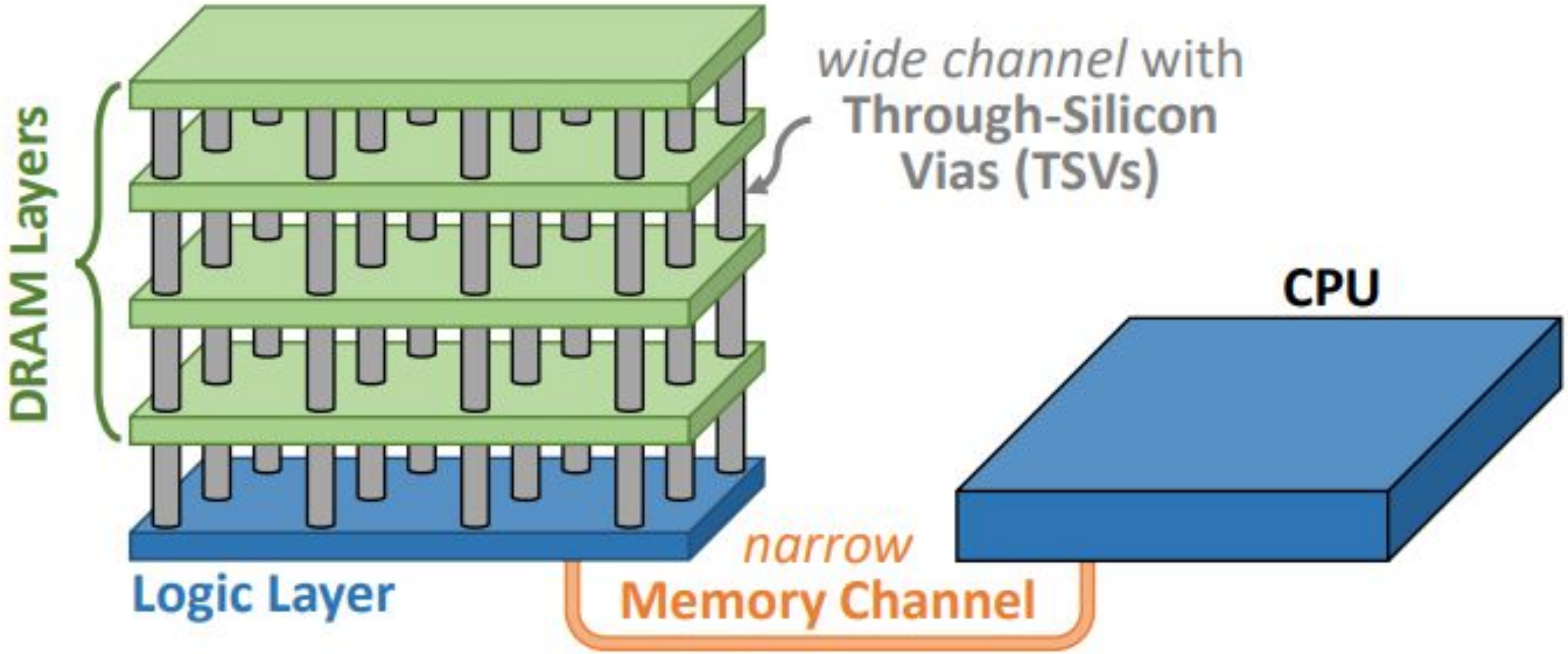

Fig. 1. High-level overview of a 3D-stacked DRAM based architecture.

Possible PIM layers

- Fixed-function accelerator
- Simple in-order core
- Simple reconfigurable logic
- ~~Out-of-order core with large cache and sophisticated instruction-level parallelism~~

The complexity is limited by the manufacturing process and thermal design (and cost and area for consumer devices)!

Examples of 3D-stacked DRAM in 2018

- Hybrid Memory Cube (HMC), first CPU using HMC was Fujitsu SPARC64 XIfx in 2015
- High Bandwidth Memory (HBM), first GPU using HBM was AMD Fiji in 2015

They make limited use of the logic layer!

- HMC implements command scheduling logic there

Using PIM logic in applications

- PIM architecture exposes an interface to the CPU
- No standardization of this interface, PIM typically treated as a coprocessor
- PIM used to execute:
    a. entire application
    b. single function
    c. single instruction

Different ideas in different papers! Let’s look at an example of (c), and then (b).

PIM-Enabled Instructions

- PIM-Enabled Instructions (PEI) added to CPU’s ISA
  - memory accessible by PEI is limited to a single LLC block
- PEI Computation Unit (PCU) - executes PEIs
- PEI Management Unit (PMU) - coordinates all PCUs in terms of:
  - atomicity management (e.g. PEI atomic add)
  - cache coherence (so that all operations access the latest data)
  - data locality profiling for locality-aware execution

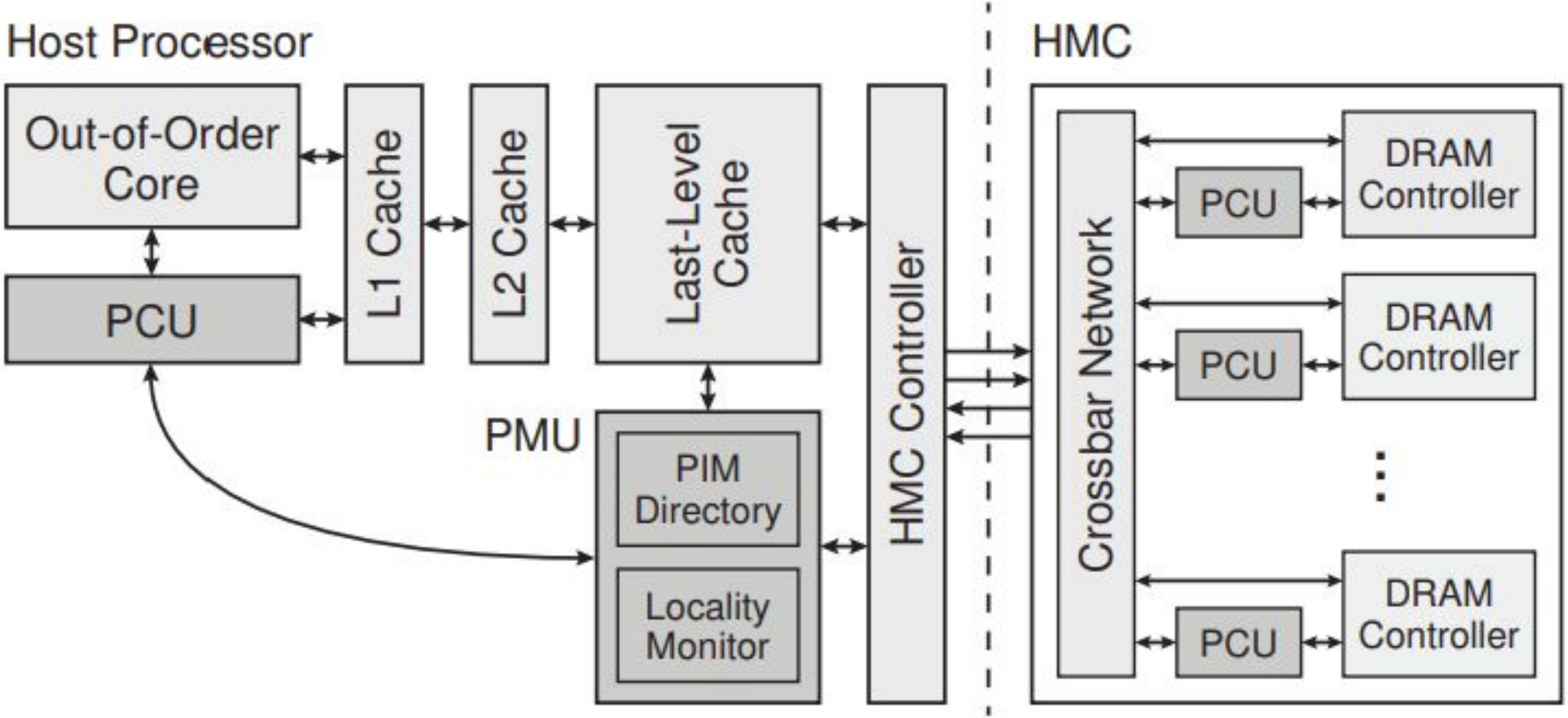

Figure 3: Overview of the proposed architecture.

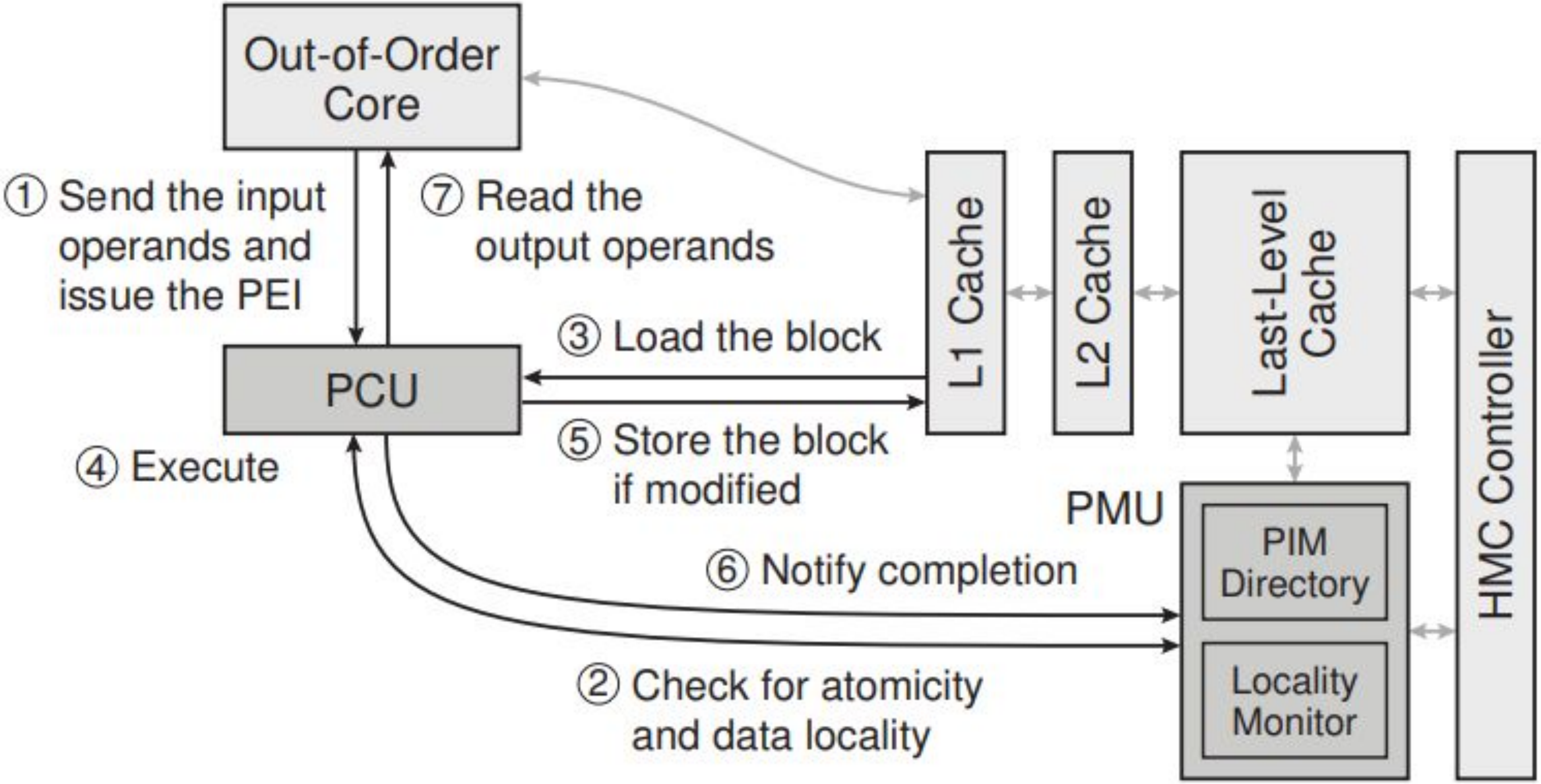

Figure 4: Host-side PEI execution.

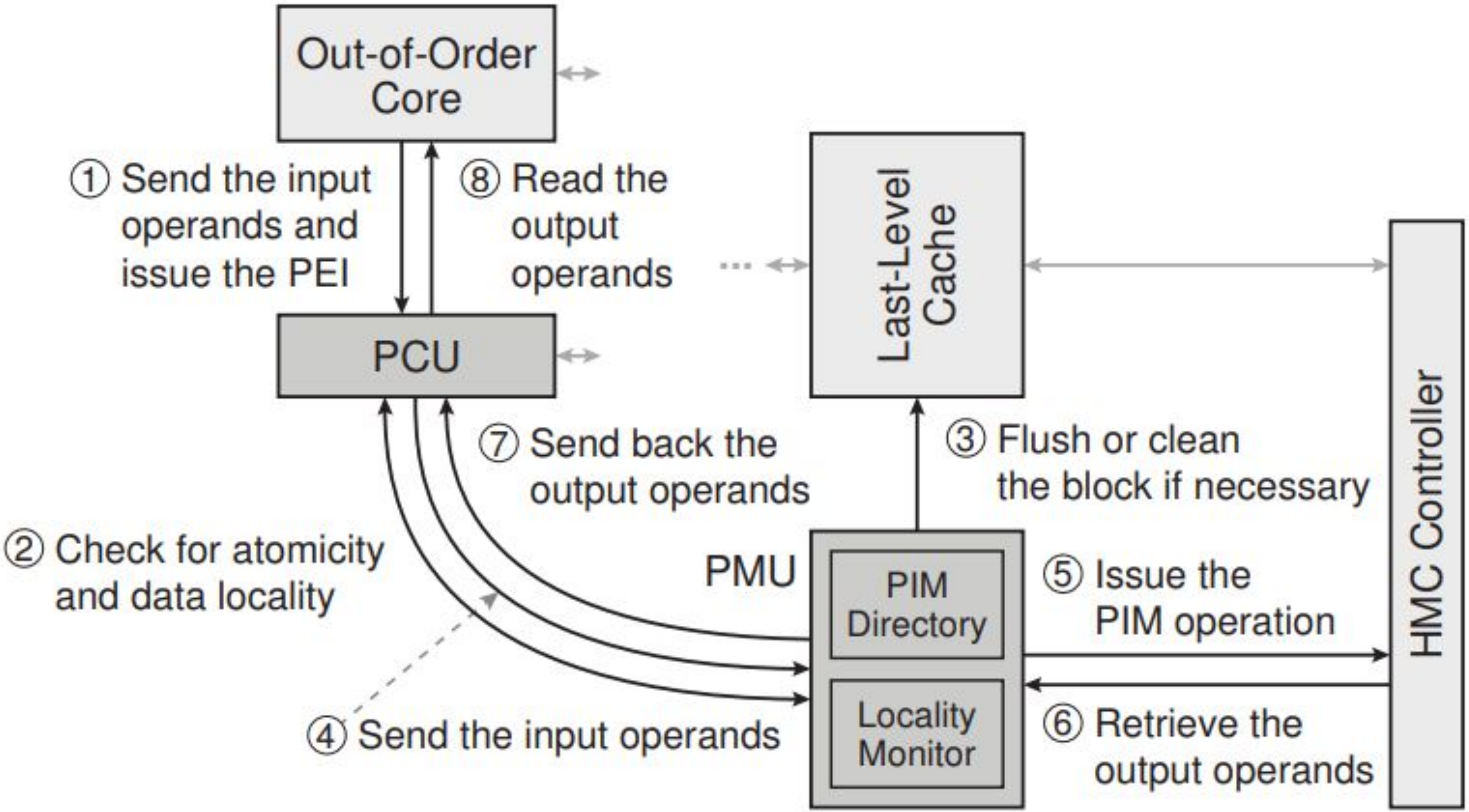

Figure 5: Memory-side PEI execution.

PIM-Enabled Instructions: evaluation

- Simulation using an in-house x86-64 simulator that models:
  - out-of-order cores,
  - caches,
  - DRAM controllers inside HMC,
  - MESI cache coherence protocol,
  - etc.
- Benchmarking using i.a.:
  - graph: Breadth-First Search (BFS), Single-Source Shortest Path (SP),
  - data analytics: Hash Join (HJ), Histogram (HG),
  - ML/DM: Streamcluster (SC), Support Vector Machine Recursive Feature Elimination (SVM)
- Three input set sizes

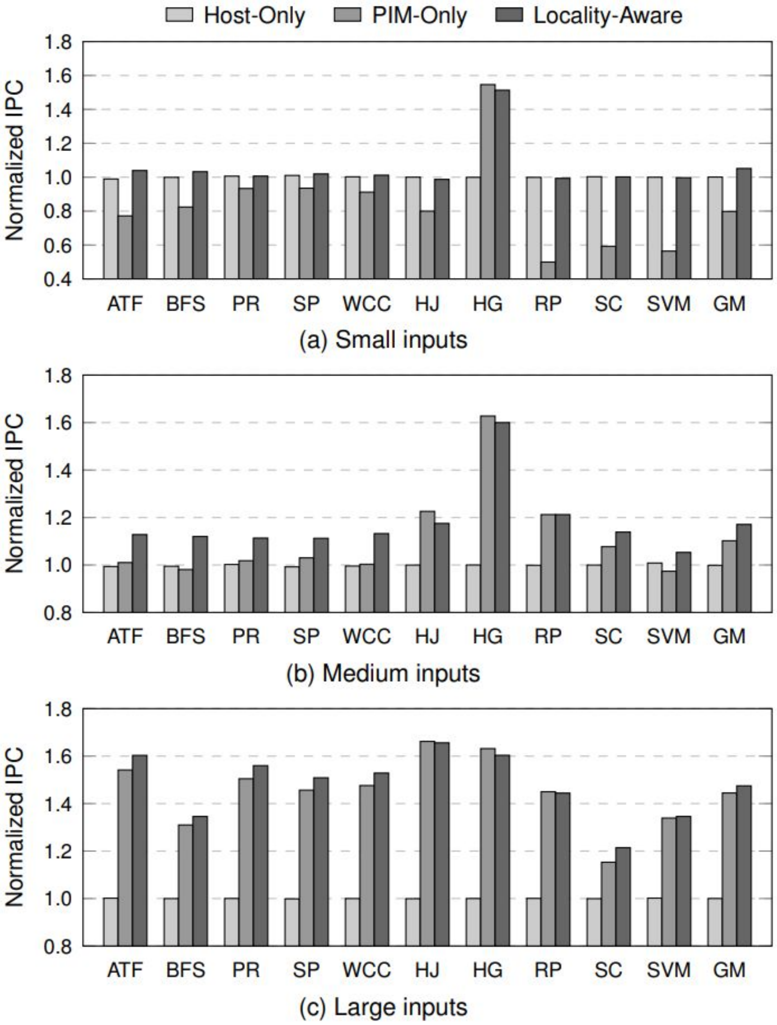

Figure 6: Speedup comparison under different input sizes.

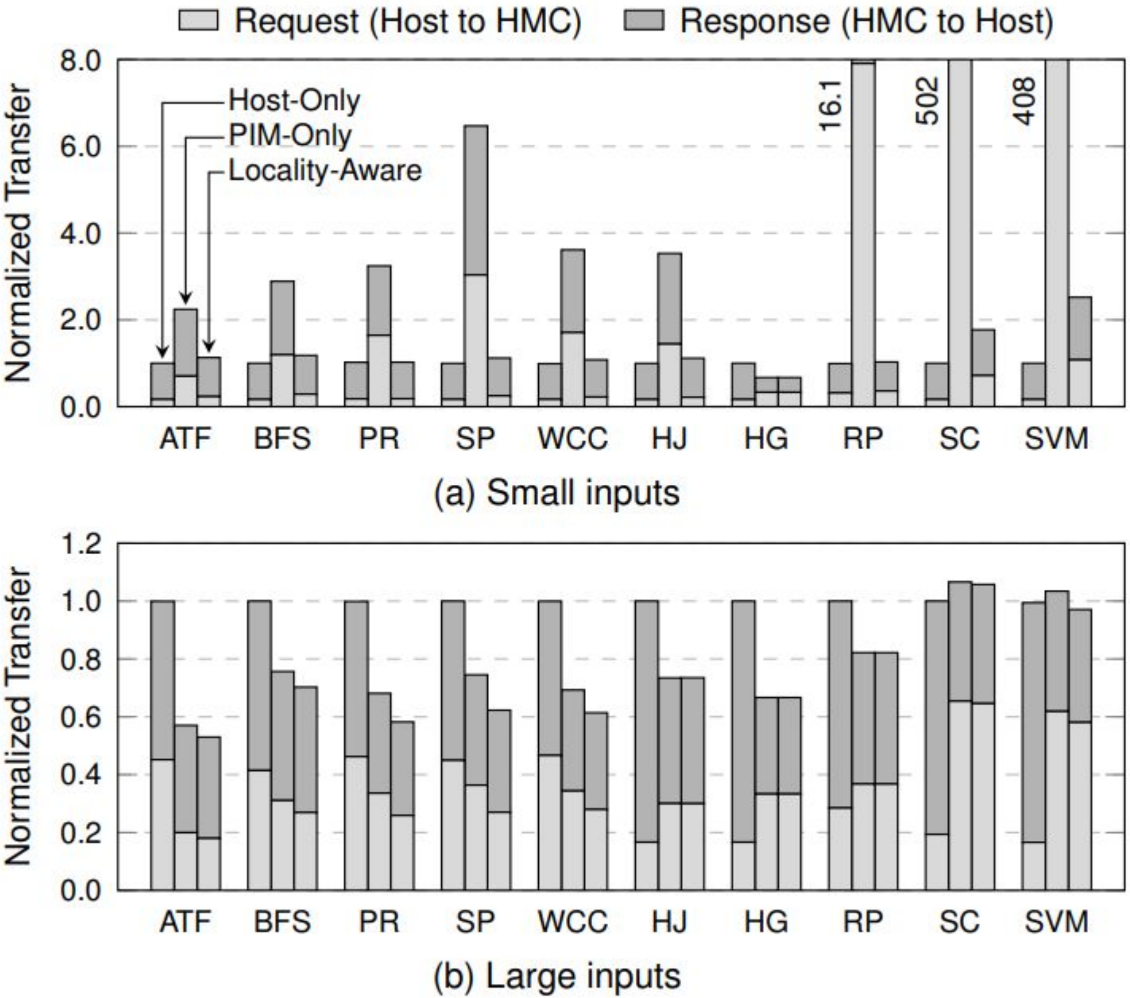

Figure 7: Normalized amount of off-chip transfer.

Another PIM example: pointer chasing

- Memory access pattern where previous memory access is required to determine the address of next memory access
- Used heavily in: databases and file systems, graph processing, garbage collectors, video games (binary space-partitioning trees for rendering), routing tables
- Very inefficient in general-purpose CPU

Solution: In-Memory PoInter Chasing Accelerator

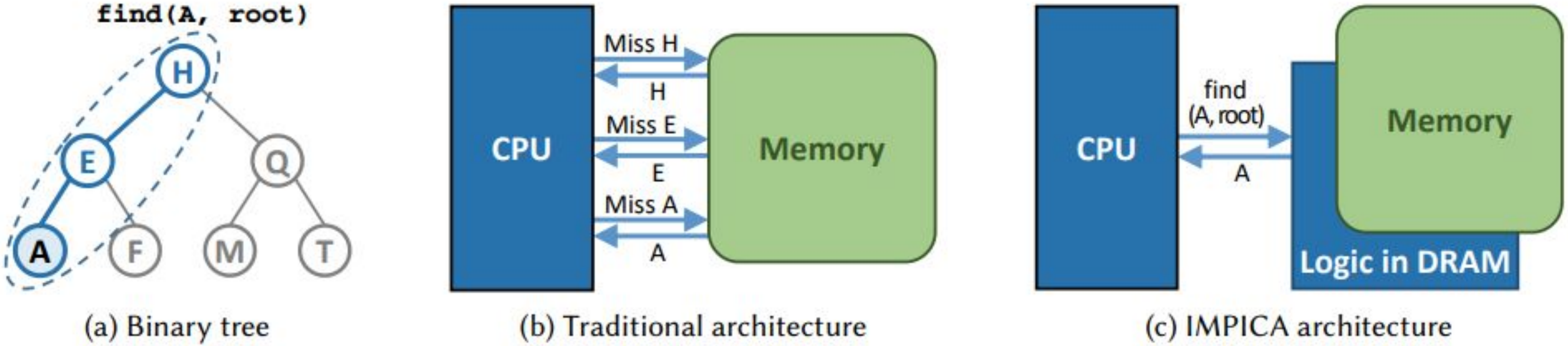

Fig. 3. Pointer chasing (a) in a traditional architecture (b) and in IMPICA with 3D-stacked memory (c). Figure adapted from [67].

Solution: In-Memory PoInter Chasing Accelerator

- Not that easy:
  - how to handle parallel chasing for multiple CPU cores?
  - how to handle virtual-physical address translation?
  - …
- We could spend another seminar discussing IMPICA!
- It’s also only a proof of concept (as PIM-Enabled Instructions), evaluated using a simulation

Let’s come back to the original paper

- Now we have some background in processing-in-memory (PIM)
- The authors of *Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks* analysed most popular Google consumer workloads
- 63% of the total energy is consumed by data movement, so let’s move some parts of the applications (PIM targets) to PIM logic!
- Is it feasible and reasonable, given the limited area and power constraints of a *consumer* device?

But wait…

- How can we measure energy consumed by data movement? Or even by basic components such as CPU / L1 / interconnect / memory controller / …
- The authors used a memory model of a different processor created in a prior work, and scaled it to fit their Intel Celeron
- The model is driven by hardware performance counters

Operation	Energy cost(nJ)	ΔEnergy(nJ)
NOP	0.105	-
ADD	0.105	-
LOAD L1→Reg	0.192	0.192
LOAD L2→Reg	0.803	0.611
LOAD RAM→Reg	12.032	11.228
Stall cycle	0.068	-

A fragment of the original memory model created for Samsung Galaxy S3 (Exynos SMDK 4412 Quad with 4 ARM Cortex A9 cores). Source: *Quantifying the Energy Cost of Data Movement for Emerging Smartphone Workloads on Mobile Platforms.* D. Pandiyan et al. 2014.

Identifying ideal PIM target

A function is a good candidate if:

- it consumes the most energy out of the all functions in the workload,
- its data movement consumes a significant fraction of the total workload energy,
- its LLC misses per kilo instruction (MPKI) is greater than 10,
- it doesn’t require more area than available in the logic layer,
- etc.

Google Chrome: case study

- One of the most commonly-used applications by consumer device users with over *a billion active users*
- What happens while using the browser?
- Which functions in the browser use most energy due to data movement?
- Which functions in the browser are good PIM targets?
- Would it be better to implement them using a PIM-Core or PIM-Accelerator?

User perception of the browser

Based on three main factors:

1. page load time,
2. smooth page scrolling,
3. quick switching between browser tabs.

We'll focus on (2) and (3).

What happens when a web page is downloaded?

- The rendering engine, Blink, parses HTML and produces DOM tree; it also parses CSS
- *render tree* = DOM tree + style rules, a visual representation of the page
- *render object* = node of the render tree
- *layout* = the process of calculating the position and size of each render object
- *rasterization* = the process of creating a bitmap per each render object
- *texture upload* = the process of sending the rasterized bitmap (also known as a *texture*) to the GPU
- *compositing* = the process of painting the pixels onto the screen (by GPU)

What happens while we scroll a page?

Scrolling triggers:

- layout,
- rasterization,
- compositing.

All three operations must happen within the mobile screen refresh time (e.g. 60 FPS / 16.7 ms) to avoid frame dropping.

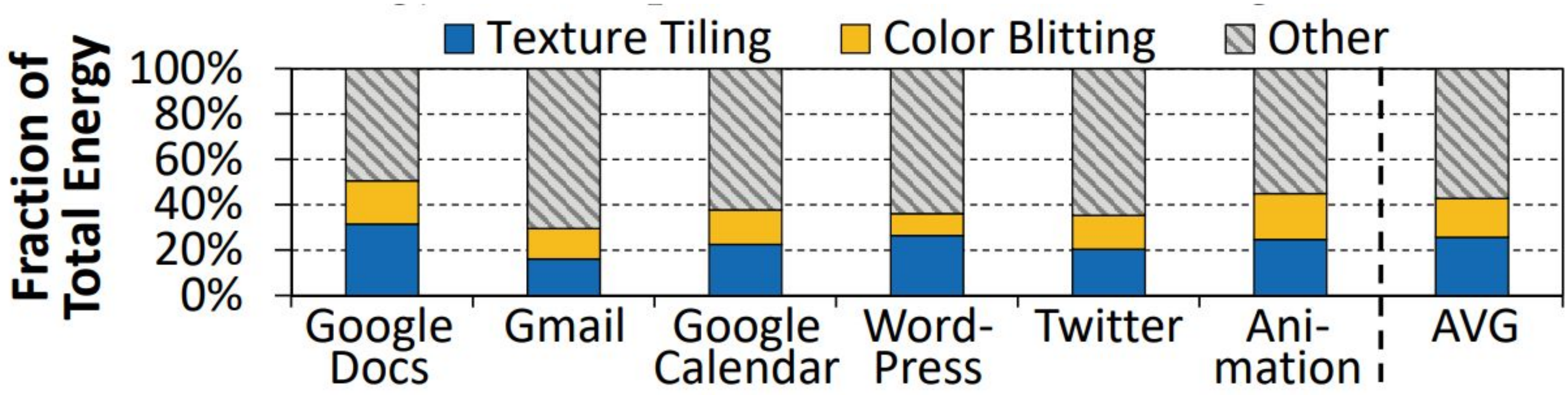

Figure 1. Energy breakdown for page scrolling.

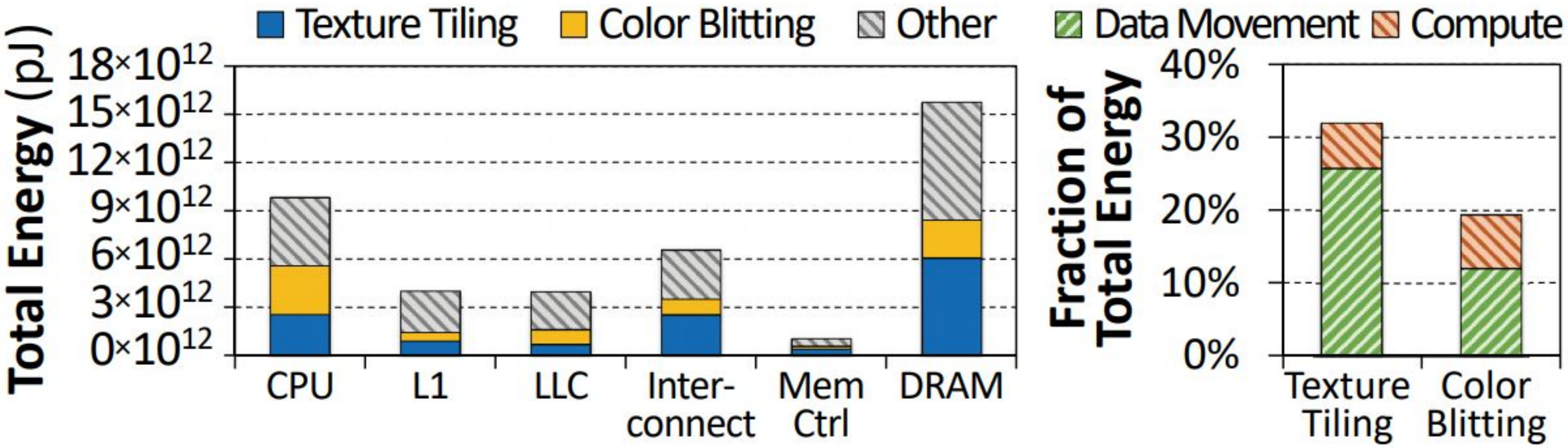

Figure 2. Energy breakdown when scrolling through a Google Docs web page.

The most data-intensive components

- Texture tiling:
  - Rasterization generates a bitmap, which is written using a linear access pattern to memory
  - Compositing accesses each texture in both the horizontal and vertical directions
  - To minimize cache misses during compositing, the graphics driver converts the bitmap into a tiled layout, e.g. Intel HD Graphics driver breaks down each rasterized bitmap into multiple 4 kB texture tiles
  - Notice: GPU’s highly-parallel architecture is not a good fit for rasterizing fonts and other small shapes, so by default rasterization is CPU-based
- Color blitting:
  - Chrome draws basic primitives (lines, text, …) for each render object
  - The browser users color blitter, which converts the primitives into the bitmaps
  - Blitting is mainly copying a block of pixels from one location to another

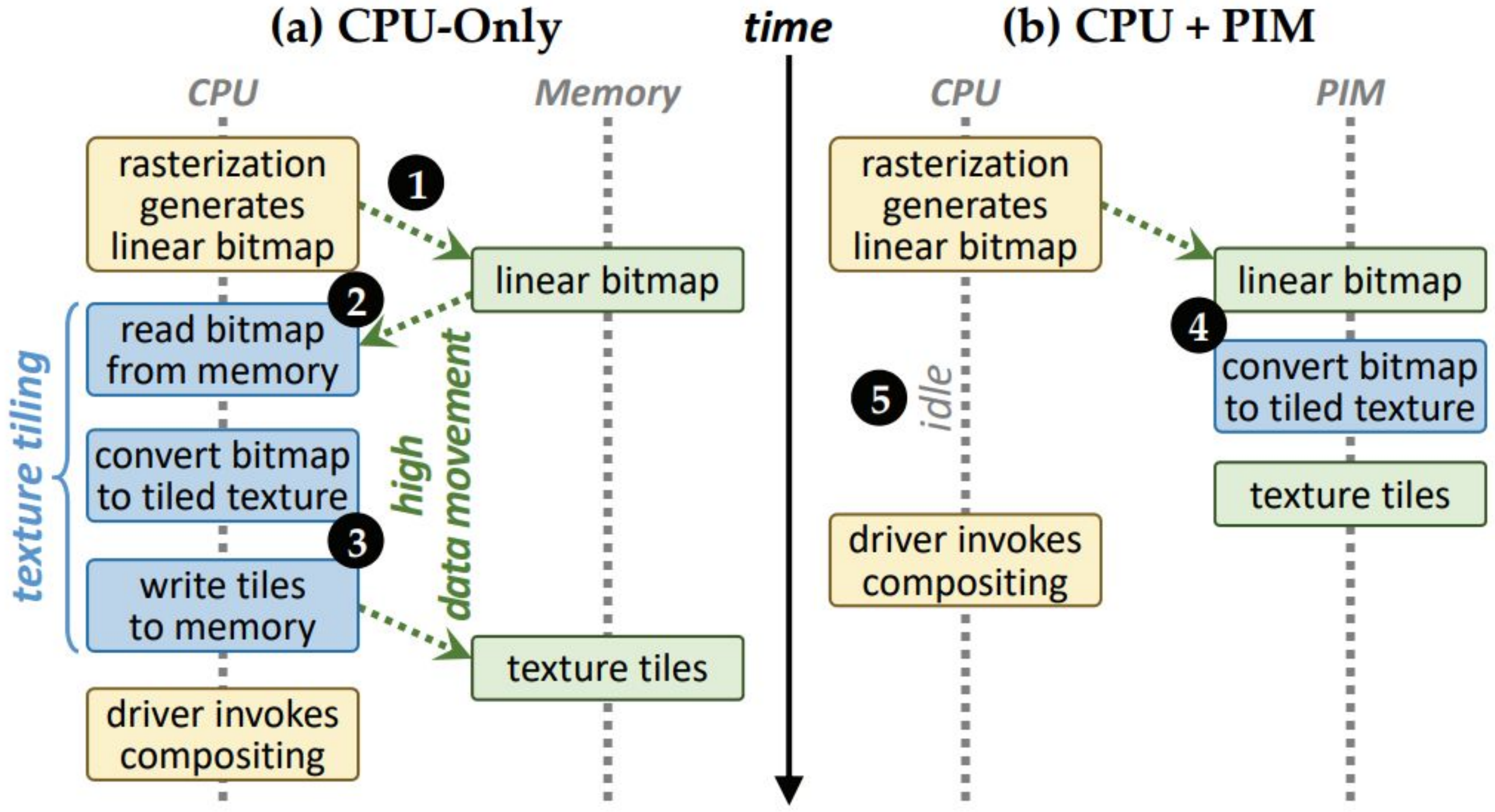

Figure 3. Texture tiling on (a) CPU vs. (b) PIM.

Texture tiling and color blitting: PIM effectiveness

- Only require bitwise operations, arithmetic operations, memcpy and memset
- These operations can be performed at high performance on PIM core or PIM accelerator
- Little area needed, so they're feasible to implement in a consumer device

Tab switching

- Each tab has its own process
- Switching between tabs triggers:
  - a context switch,
  - a load operation for the new page
- Fast tab loading is important, but the memory consumption is a major concern:
  - average memory footprint of a web page increases on a yearly basis,
  - users tend to open multiple tabs at a time,
  - consumer devices have lower memory capacity than server / desktop systems
- Chrome compresses inactive tabs and places them into a DRAM-based memory pool, called ZRAM

Tab switching energy analysis

- An experiment was made:
  - user opens 50 tabs,
  - scrolls through each tab for a few seconds,
  - switches to the next tab
- In total 12 GB of data swapped out to ZRAM, 8 GB of data swapped in
- Compression and decompression contributed to 18% of the total system energy

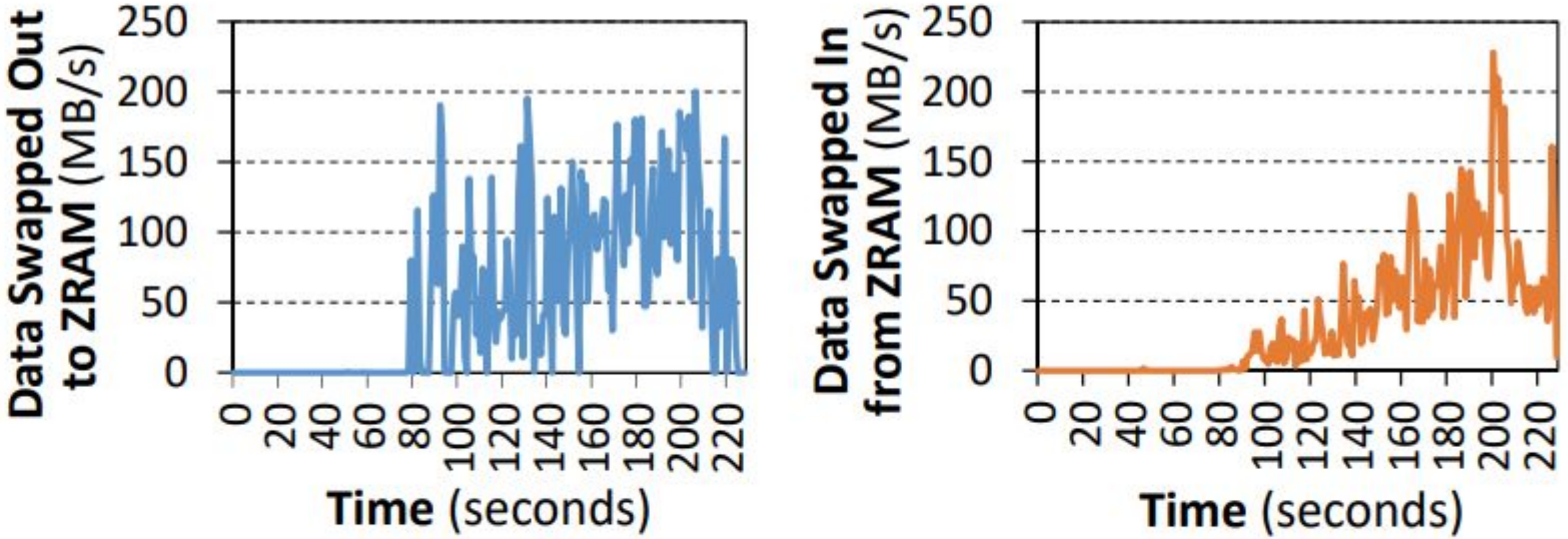

Figure 4. Number of bytes per second swapped out to ZRAM (left) and in from ZRAM (right), while switching between 50 tabs.

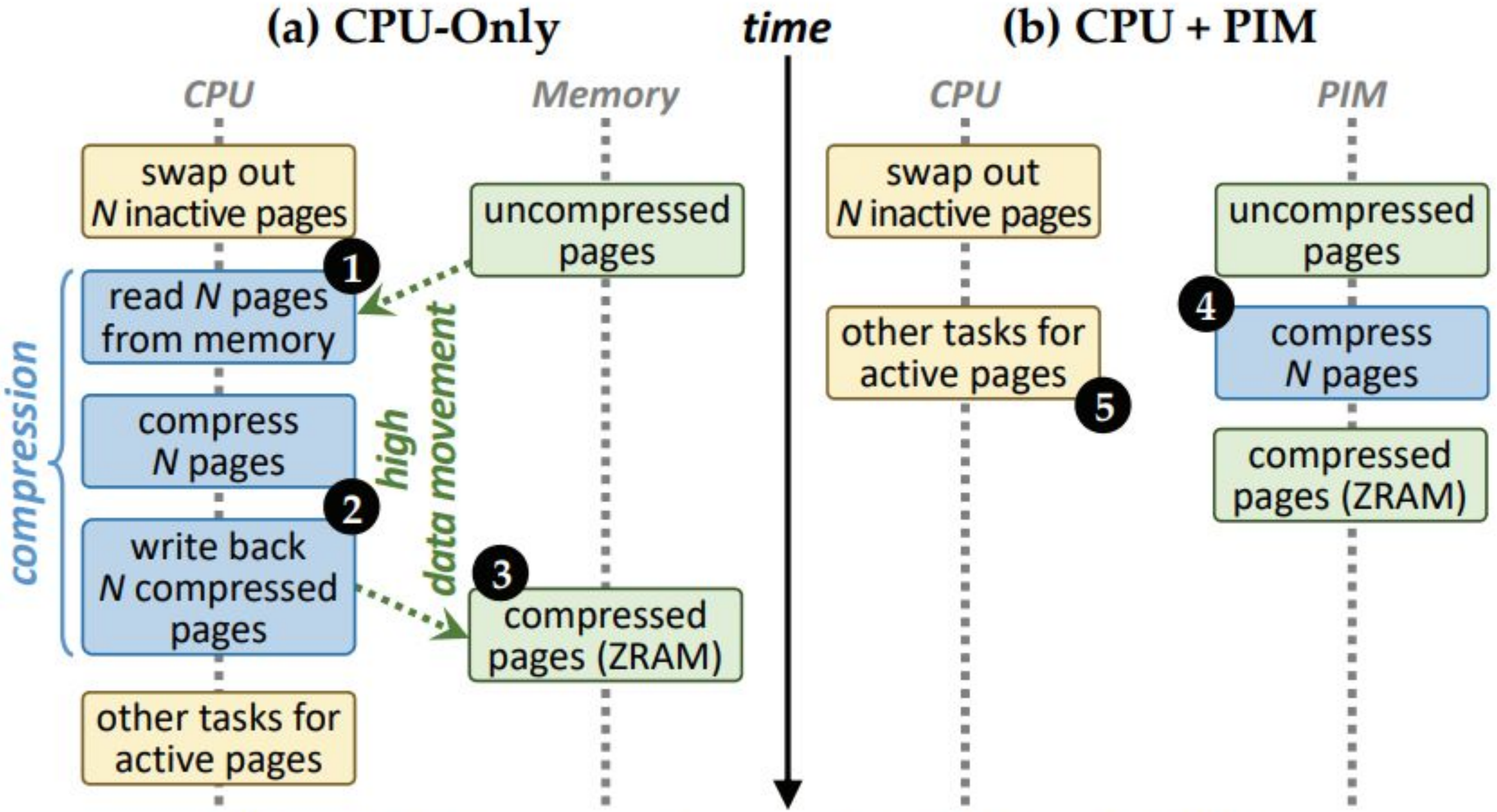

Figure 5. Compression on (a) CPU vs. (b) PIM.

Tab switching: PIM effectiveness

- Good fit for PIM execution
- Compression can be handled in the background
- ZRAM uses LZO algorithm, which uses simple operations and favors speed over compression ratio
- LZO can be efficiently implemented as a PIM core or a PIM accelerator
- In-memory compression/decompression can benefit other use cases:
    - e.g. BTRFS or ZFS, not yet widely supported in mobile OSes

Other workloads

- In the paper you can find similar analyses for:
  - TensorFlow Mobile
  - Video playback using VP9 decoder
  - Video capture using VP9 encoder
- They’re not really related to our seminar
- We’re fine with just Google Chrome

Evaluation

- Done using gem5 full-system simulator
- Many methodology details described in the paper, if you are interested

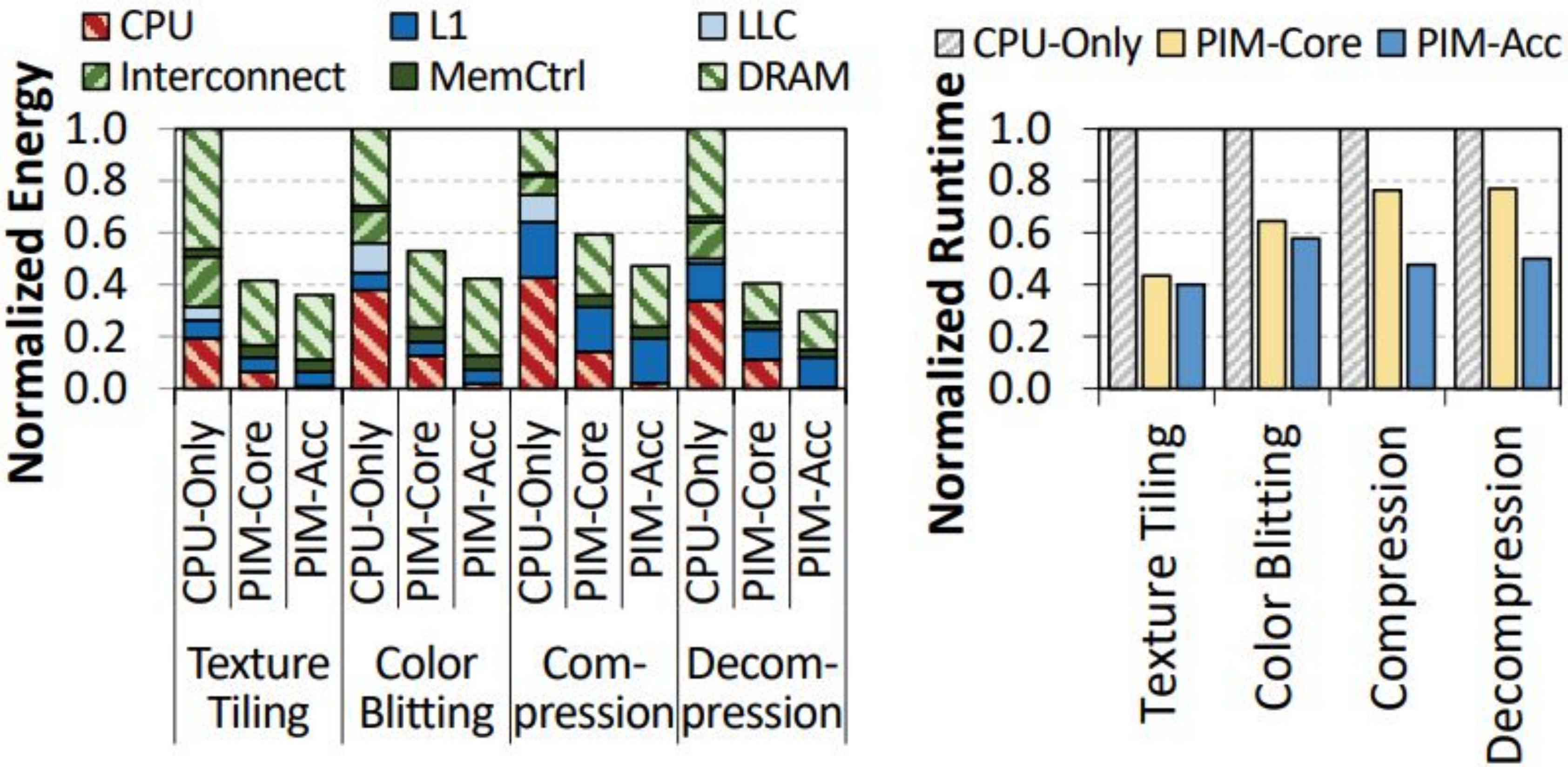

Figure 18. Energy (left) and runtime (right) for all browser kernels, normalized to CPU-Only, for kernel inputs listed in Section 9.

Conclusions

- Data movement contributes to a significant portion (**62.7%**) of widely-used Google consumer workloads
- Majority of this data movement comes from a number of simple functions
- Offloading these functions to PIM logic reduces (in all workloads, on average):
  - energy consumption by **55%**
  - execution time by **54%**
- Very promising results!

Bibliography

- **Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions.** S. Ghose et al. 2018.
- **Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks.** A. Boroumand et al. 2018.
- **PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture.** J. Ahn et al. 2015.
- **Quantifying the Energy Cost of Data Movement for Emerging Smartphone Workloads on Mobile Platforms.** D. Pandiyan et al. 2014.
- https://en.wikipedia.org/wiki/Hybrid_Memory_Cube
- https://en.wikipedia.org/wiki/High_Bandwidth_Memory
- https://en.wikichip.org/wiki/pointer_chasing