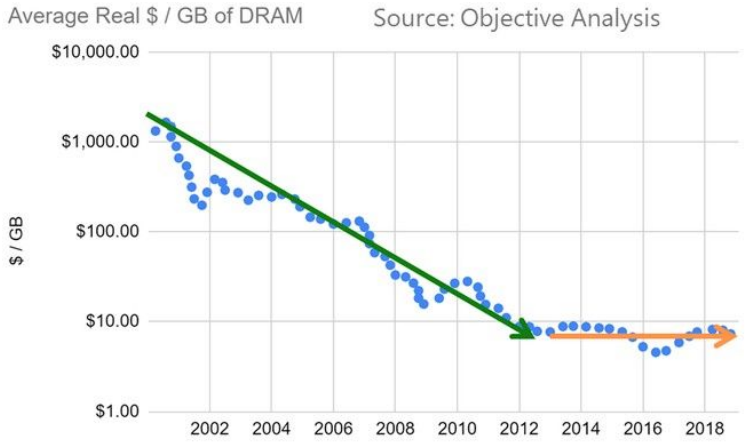
Heterogeneous Memory Management in Servers with fitd

Chun Deng, Anna Eaton, Evan Laufer, Philip Levis

DAM Summer Retreat
July 2, 2025

Stanford | ENGINEERING

DRAM costs are driving new memory approaches



- DRAM costs dominate in servers
  - Microsoft claims that 50% of their server costs are from memory, Meta claims 40%[1]
- New memory types
  - 3D crosspoint
- New interconnects
  - CXL
- Software approaches
  - zswap

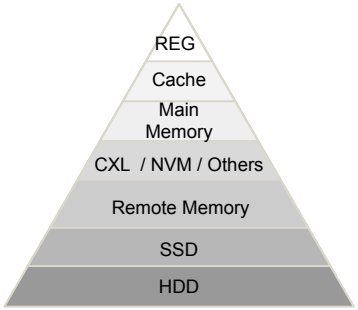
[1] Maruf et al. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory

Systems today assume a hierarchy

- Research into adding a new “tier” of memory
  - HeMem (SOSP 21’)
  - Memtis (SOSP 23’)
  - TPP (ASPLOS 23’)
  - Colloid (SOSP 24’)
  - Nomad, Memstrata (OSDI 24’)
- Use “temperature” (hot/cold) to decide where to place data
- Placing data in tiers based on temperature implies a hierarchy
  - Faster to slower (latency, bandwidth, etc.)
  - Expensive to cheap

Memory is not always a strict hierarchy

Memory	Read Latency	Read Bandwidth	Write Bandwidth
DDR5 DRAM	114ns	38 GB/s	38 GB/s
Optane DC	172ns	32 GB/s	11 GB/s
CXL	239ns	52 GB/s	52 GB/s



- Optane read bandwidth similar to DRAM, write bandwidth is ⅓ of DRAM
- DDR5 over CXL is higher bandwidth *and* higher latency than Optane
- CXL and Optane don’t have a strict ordering

[1] Liu et al. Dissecting CXL Memory Performance at Scale: Analysis, Modeling, and Optimization
[2] Raybuck et al. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM

Deciding where to put data

- Suppose we have a server with multiple types of memory, each with different tradeoffs (not a hierarchy)
  - Capacity/cost
  - Latency
  - Read bandwidth
  - Write bandwidth
- How can an operating system decide which pages to put in what memory type?
  - Needs to understand all of the tradeoffs
  - Needs to observe access patterns at extremely high speed

Proposal: fitd, the memory fitness daemon

Add a small amount of hardware support in the MMU to score memory access patterns for different memories (“fitness”)

> Challenge: how can we very inexpensively score access patterns across a diverse set of tradeoffs?

Operating system decides when to move pages between memory types based on fitness scores

> Challenge: how can we scale to a large amount of pages, while maintaining the quality of assignments?

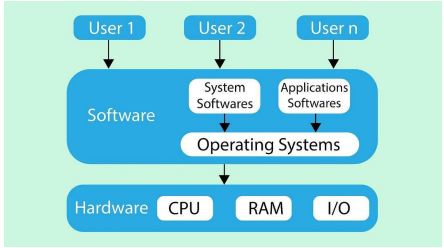
Outline of this talk

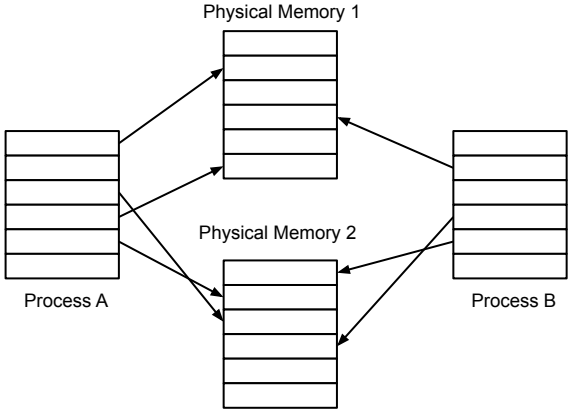
1. Background
    a. Operating systems and memory management
    b. OS page placement
    c. OS-Level system example

1. Heterogeneous memory management with fitd
    b. Estimating fit
    c. Score-based decision-making
    d. Implementation
    e. Preliminary results

Background: Operating System and Memory

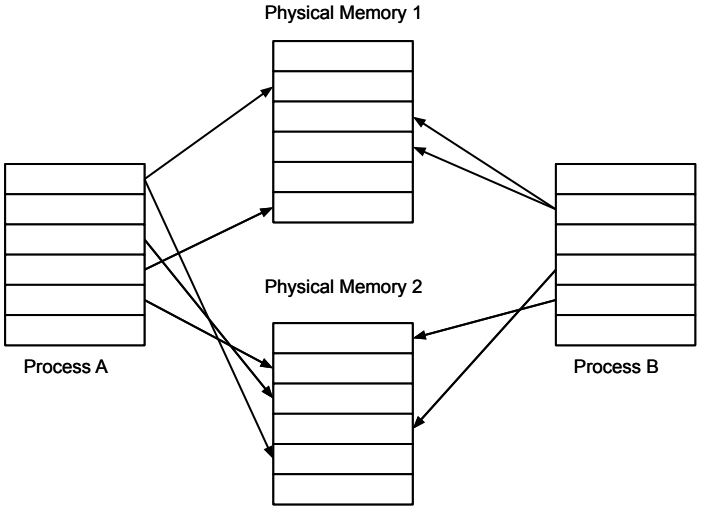
- Operating system is the lowest software layer, it can directly access all of the hardware
- Provides useful abstractions to applications (e.g., files instead of disk blocks)
- Virtualizes memory, mapping application addresses to actual physical memory
- Operating system gets to decide which physical memory a given part of an application's data lives in





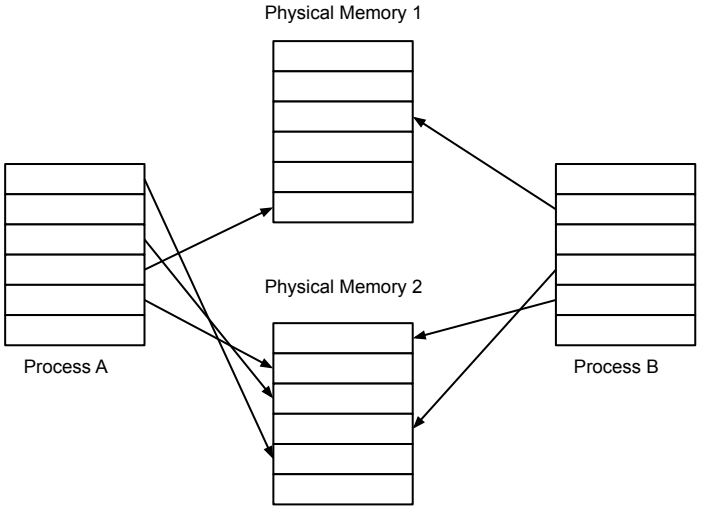
Background: OS-Level Page Placement Policies

- The OS decides where a page should reside in physical memory



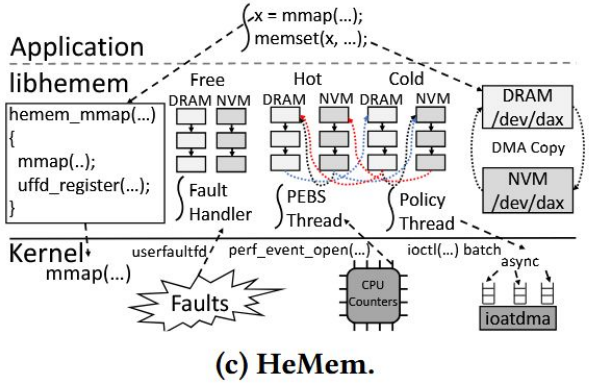
Background: OS-Level Page Placement Policies

- The OS decides where a page should reside in physical memory
- It can change this decision and *migrate* a page from one memory to another, updating the virtual memory pointers
- Migration hidden from application



Example Prior Work: HeMem *[1]

- HeMem is a system that chooses between DRAM and Optane
- The OS tracks the number of access counts to pages
- Once the counter hits 8 reads or 4 writes, it is considered “hot” and HeMem promotes it to DRAM
- Periodically halve access counts to “cool” pages, demote pages with low counter values
- Every few seconds, decide on promotions and demotions, move pages



(c) HeMem.

[1] Raybuck et al. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM

Outline of this talk

1. Background
   a. Operating systems and memory management
   b. OS page placement
   c. OS-Level system example

1. Heterogeneous memory management with fitd
   b. Estimating fit
   c. Score-based decision-making

1. Implementation
2. Preliminary results

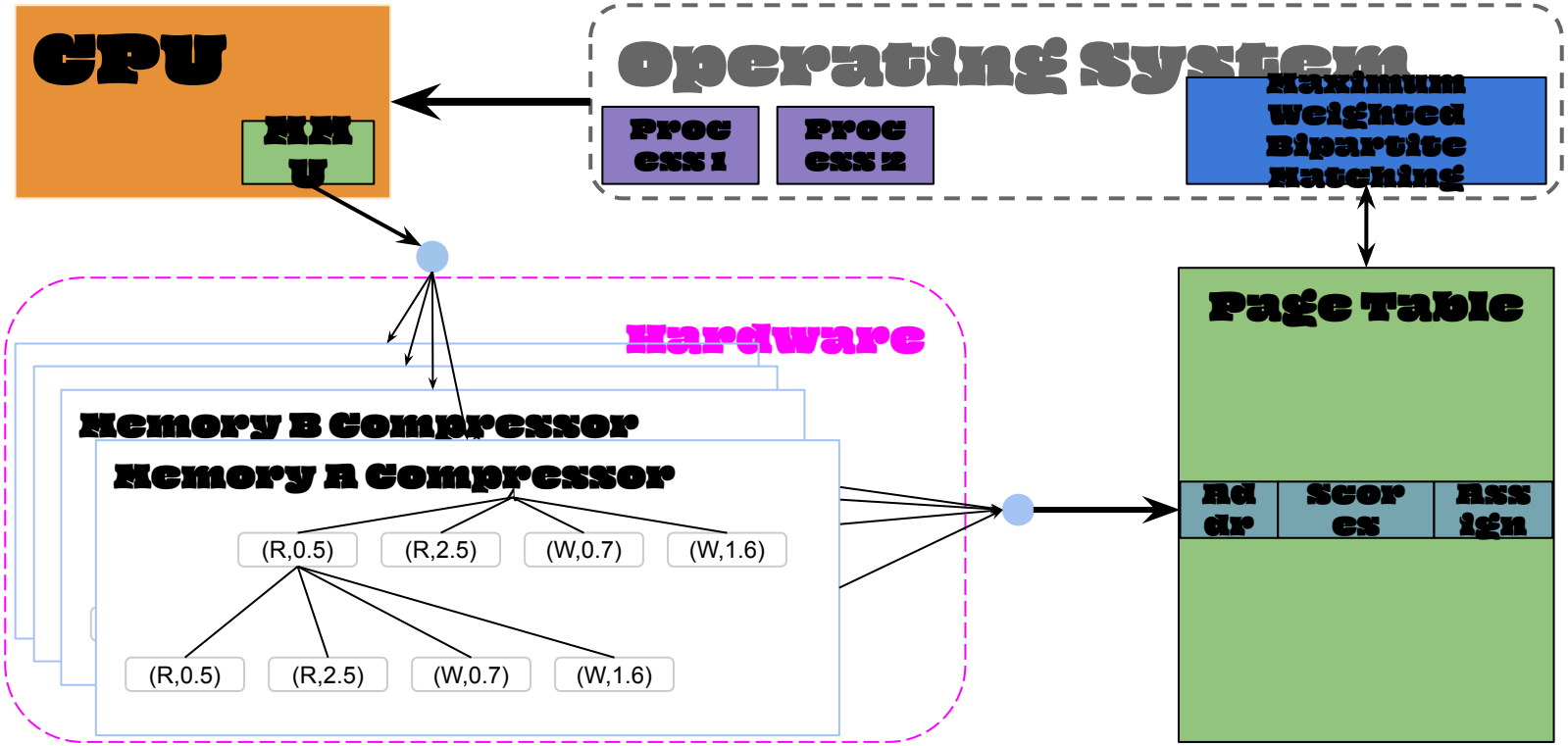
fitd: System for Heterogeneous Memory Management

1. fitd is composed of two parts - a hardware scoring component, and an operating system daemon.

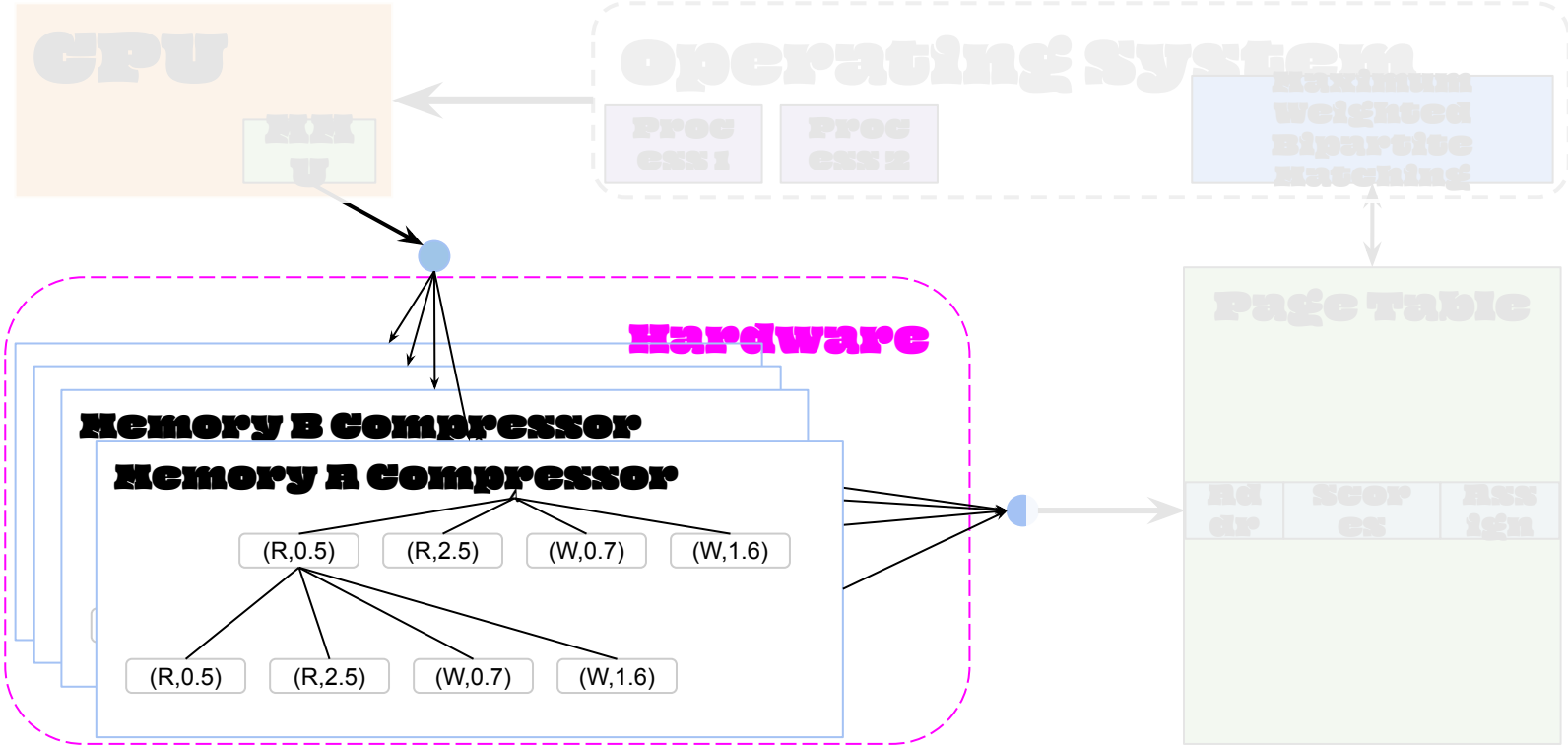
1. The hardware component creates “fit” scores indicating how much a memory prefers a particular page’s access patterns
   a. Key insight: Instead of counting patterns, treat the problem as matching the two distributions.
   b. Use a novel lightweight compression algorithm to estimate match

1. The operating system daemon uses these “fit” scores to determine placement.

fitd design: Overview



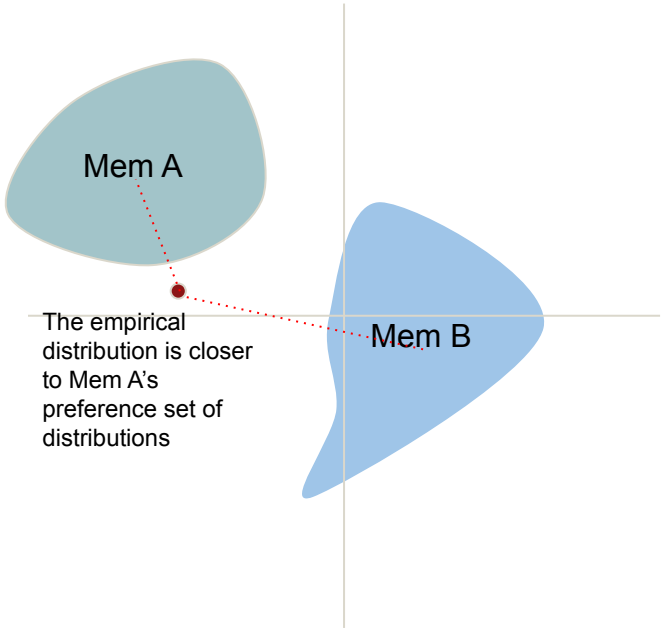
fitd Design: Hardware Component



Compression as a distribution learner

- We want to be able to estimate how well a pattern “fits” a memory.
- First step: model what “good” patterns are for each memory

- **This can be framed as a compression task**
  - Compressors learn an expected distribution of values (patterns)
  - If it can compress an access pattern well, the accesses are closer to the “good” patterns



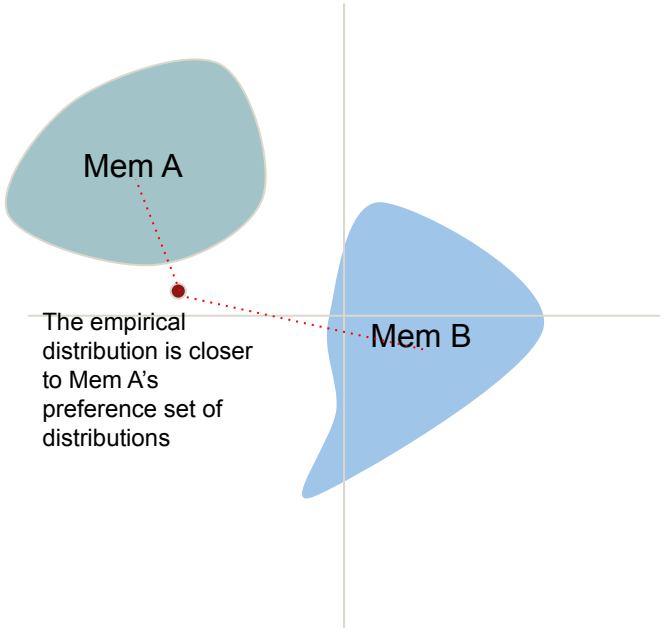
distribution “simplex” - each point is a distribution

Compression as a distribution learner

Offline: Build a compressor, trained on “good” patterns for that memory, incorporated into MMU

Hardware: Compress the access pattern of each page with each compressor

Software: See which compressor works best! High compression means good fit.



distribution “simplex” - each point is a distribution

Need a new lossy compression algorithm

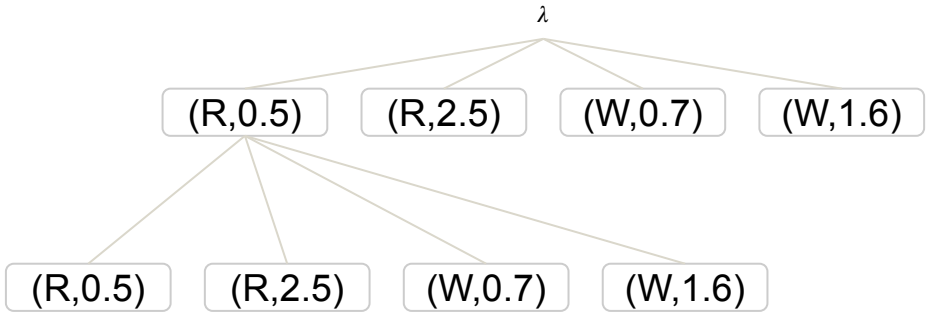
- Standard lossless compressors (LZ77, LZ78) are poor for memory traces

- Use exact phrase matching
  - Encode “nop nop write” separately from “nop nop nop write”
  - Dictionary sizes grows rapidly

- Limiting dictionary/window sizes performs poorly
  - Around 60-70% in a binary classification setting, over simple traces
  - Compressor A trained on 7 0’s followed by a 1
  - Compressor B trained on 14 0’s followed by a 1
  - Evaluated on 9 0’s followed by a 1, Compressor B compressed better (this isn’t what we want)

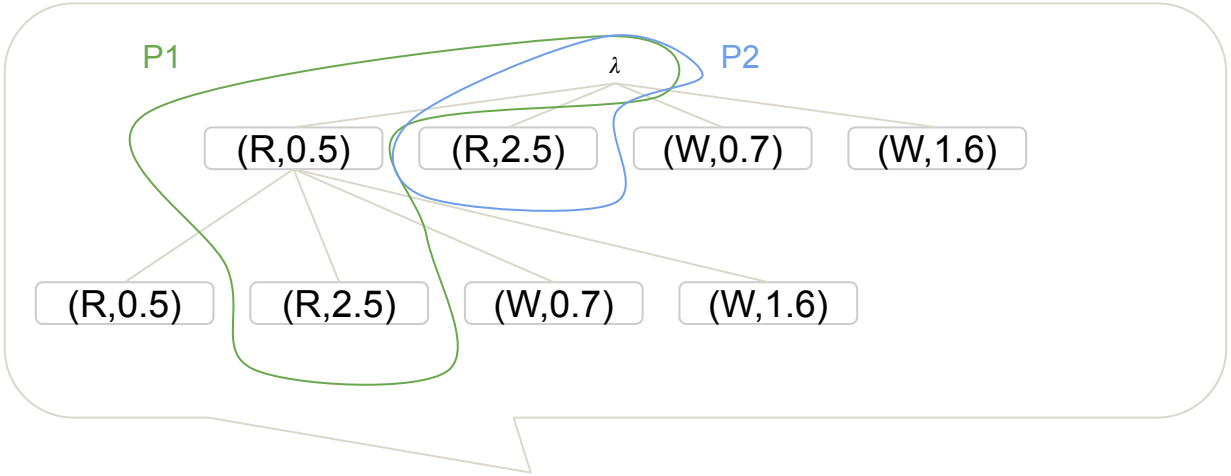
Soft Context Tree Weighting

- Adapt Context Tree Weighting (CTW) algorithm to work with run-length encoded data
- Cluster branches of the context tree based on access type (read/write) and sparsity of access (high = sparse)
- Use KDE estimation to blend probabilities from the KT-estimator



$$P(i) = \frac{\sum_i \omega_i * KT(i)}{\sum_i w_i}$$

A Running Example



Access Trace: (R, 1), (R, 2.5), (R, 0.9), (W, 1.4), (R, 0.5)

Cost: C1 + C2

Where C2 = log(blending of P1 & P2)

Picking out traces to train our compressor model

- What traces are considered “good” for each memory?
  - Depends on the objective function that is being optimized. Is it average read latency, endurance, cost? or a mixture?
  - Can also be manually defined
  - Ideally, these traces pulls the preference distributions apart. But are still in-distribution of real workloads

- We use a memory simulator to evaluate traces for each memory, currently optimizing for total simulation time

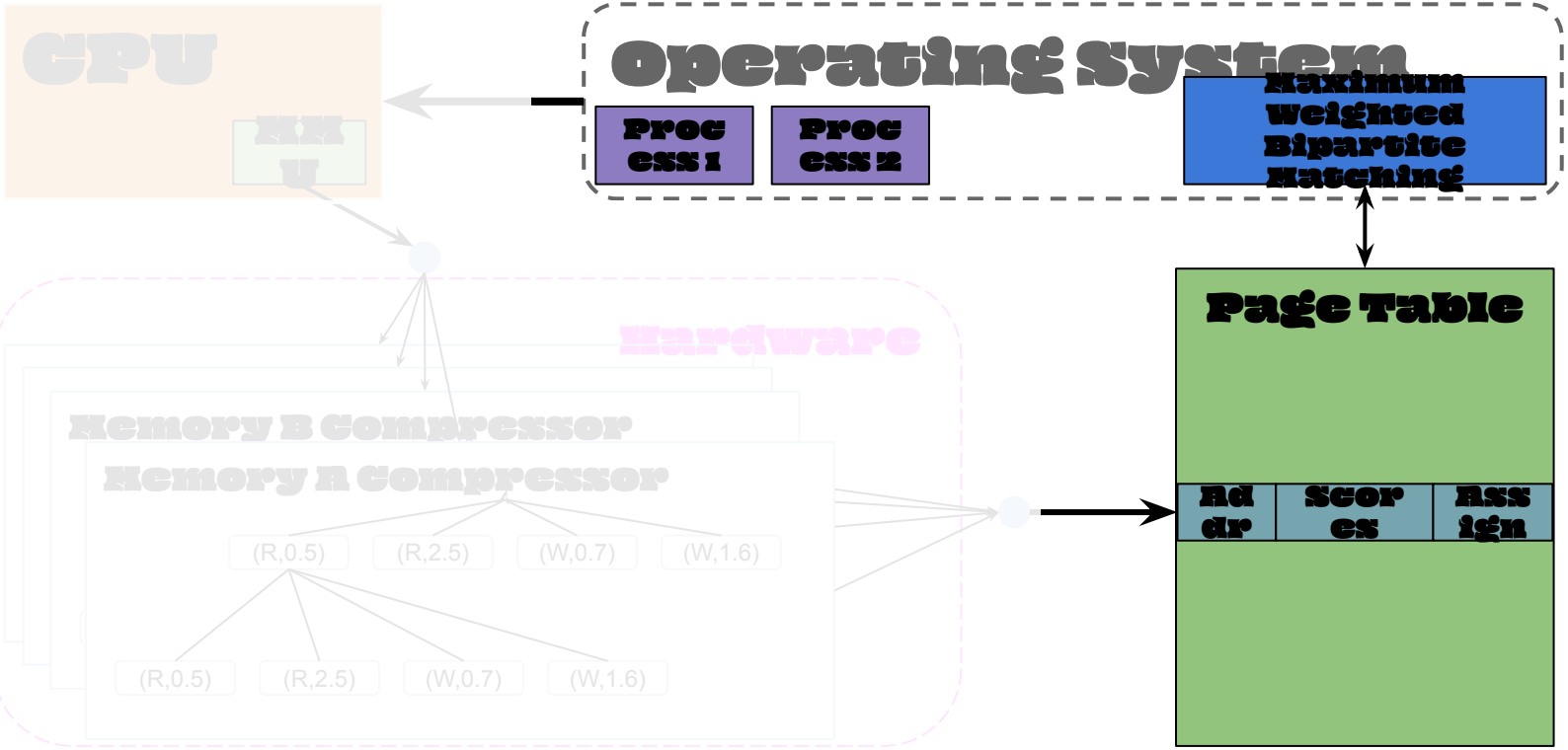
fitd: System for Heterogeneous Memory Management

1. fitd is composed of two parts - a hardware scoring component, and an operating system daemon

1. The hardware component creates “fit” scores indicating how much a memory prefers a particular page’s access patterns
   a. Key insight: Instead of counting patterns, treat the problem as matching the two distributions.
   b. Use a novel lightweight compression algorithm to estimate match

1. The operating system daemon uses these “fit” scores to determine placement

fitd design: Software Component



Assignment and Placement

	Mem A	Mem B	Mem C
Page 1	**1.4**	2.7	3.9
Page 2	2.3	3.1	**1.2**
Page 3	5.1	**0.5**	1.3

fitness scores from each page to each memory

- Standard algorithms can find the global optimum, but are too slow ($O(N^3)$, N is # of pages)
- Solution: use an incremental algorithm that converges to a local optimum
- Swap operator incrementally swaps pages that improve fitness

The swap operator

- Each memory keeps k - 1 ordered lists on the willingness of it's assigned pages to go to a memory elsewhere, where k is the number of memories

- To check for swap candidates, we check the most willing to leave element from each memory to other memory
  - At most there are k(k-1) / 2 checks

- If there are no swap candidates, by definition we are at a local optima

fitd: System for Heterogeneous Memory Management

1. fitd is composed of two parts - a hardware scoring component, and an operating system daemon

1. The hardware component creates “fit” scores indicating how much a memory prefers a particular page’s access patterns
   a. Key insight: Instead of counting patterns, treat the problem as matching the two distributions
   b. Use a novel lightweight compression algorithm to estimate match

1. The operating system daemon uses these “fit” scores to determine placement

Outline of this talk

1. Background
   a. Operating systems and memory management
   b. OS page placement
   c. OS-Level system example

1. Heterogeneous memory management with fitd
   b. Estimating fit
   c. Score-based decision-making

1. Implementation
2. Preliminary results

SCTW Hardware Implementation Optimization

- KT-probabilities are retrieved from a lookup table

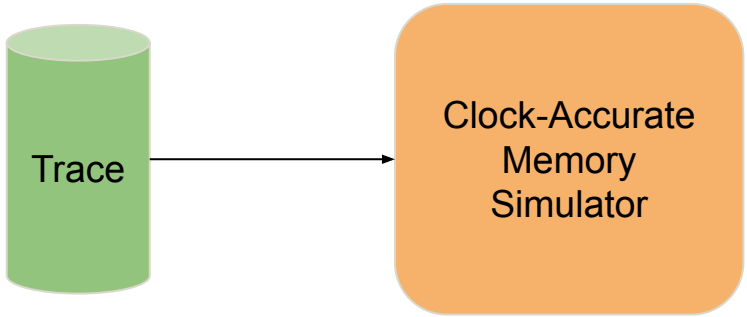
- Blending KT-probabilities at specific depth d involves a linear combination of these looked-up probabilities.
  - since addition operators are associative, we can do them in parallel

$$P(i) = \frac{\sum_i \omega_i * KT(i)}{\sum_i w_i}$$

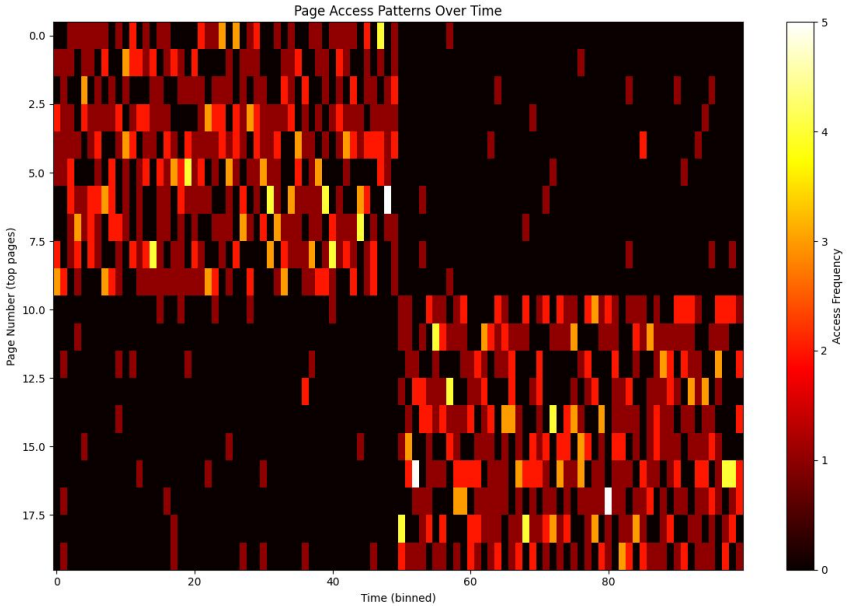
- Blending probabilities across depths also involves a linear combination, which we can also do in parallel if needed.
  - In practice, a model of depth 2 is sufficient to capture first order read write dependencies

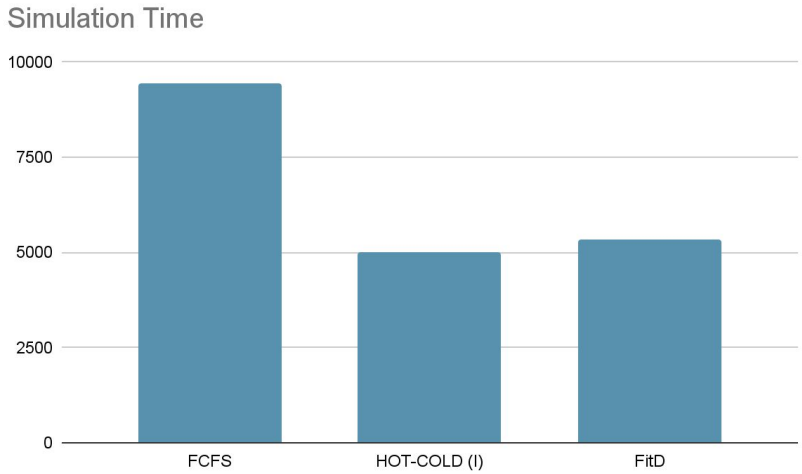
Simulator

- Trained two context trees with a generated set of microbenchmark traces
  - Optane DC and DRAM
- Evaluate fitd with those two trained context trees on microbenchmarks
  - regenerated traces for validation



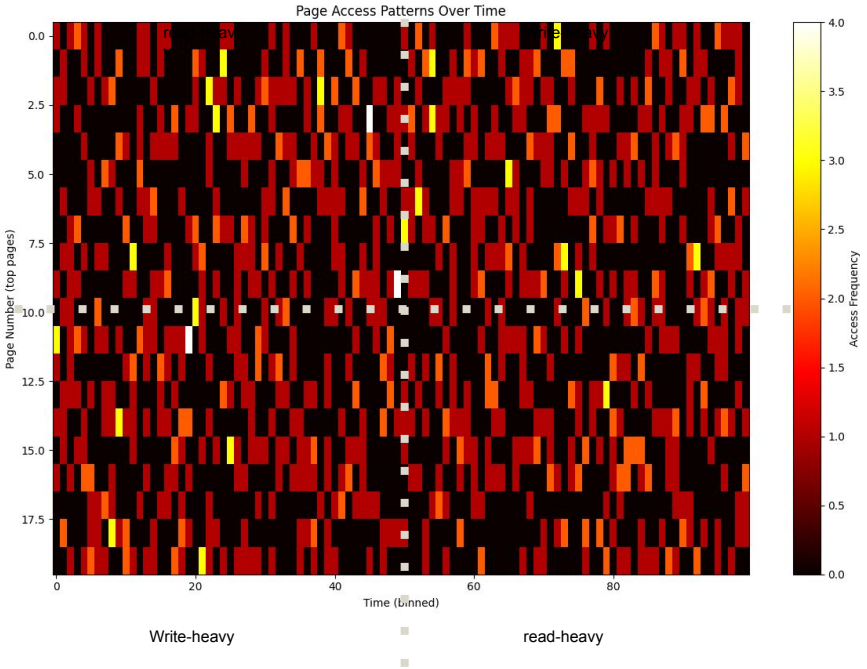
Results - Hot-cold workloads (GUPS)

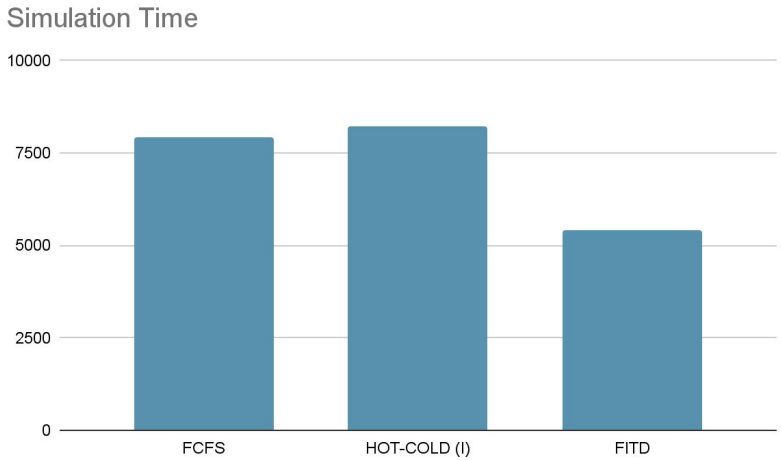




fitd can effectively represent hot-cold policies when facing hot-cold workloads

Results - Read heavy vs Write Heavy workloads





fitd significantly outperforms by exploiting patterns beyond hot-cold
i.e. read-heavy vs write-heavy patterns

Thanks