A Case for Richer Cross-layer Abstractions: Bridging the Semantic Gap with Expressive Memory

Nandita Vijaykumar   Abhilasha Jain   Diptesh Majumdar   Kevin Hsieh   Gennady Pekhimenko
Eiman Ebrahimi   Nastaran Hajinazar   Phillip B. Gibbons   Onur Mutlu

Published in ISCA 2018

Presented by Philippe Voinov

Background

- ISAs traditionally only convey program functionality
- High-level program semantics never reach hardware

Background - Cache tiling

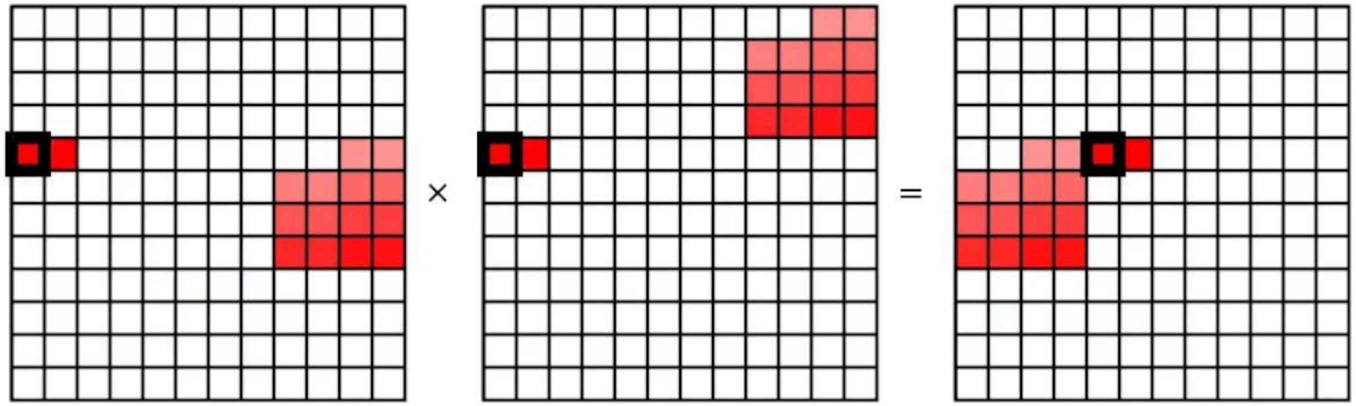


Image from video “Matrix multiplication: tiled implementation” https://www.youtube.com/watch?v=aMvCEEBIBto

Background - DRAM system structure

- DRAM systems have a very hierarchical structure
- Distributing load well throughout this structure can have performance benefits

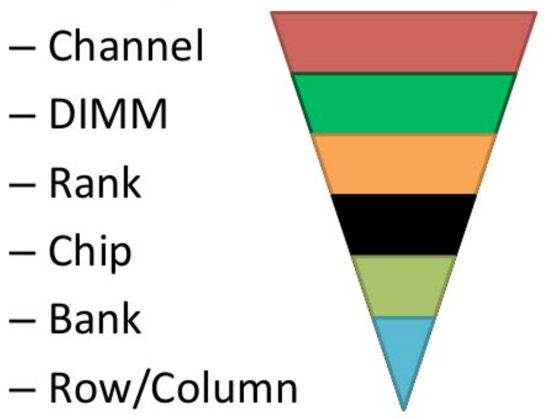


Diagram: Computer Architecture Lecture - Onur Mutlu, Yoongu Kim - Carnegie Mellon University

Background - Implications

- OS and hardware try to predict program behavior
- Compilers and programmers may try to optimize for architecture

Previous work

- Fine-grained hints as ISA instructions
- Program annotations to convey semantics
- Hardware-software co-designs

Problem

- Optimizing program execution is difficult without hints
- Fine-grained hints require large changes for each optimization
- Platform-specific directives are not portable

Goal

Create a

general cross-layer interface

to

communicate higher-level program semantics

to various system components

Novelty of XMem

- Can pass information used for multiple optimizations
- Describes properties of data, rather than directive for hardware
- Is highly extensible

Key approach - Example

```
A = malloc(size);
Atom1 = CreateAtom(“INT’, “Regular”, …);
MapAtom(Atom1, A, size);
ActivateAtom(Atom1);
…
Atom2 = CreateAtom(“INT”, “Irregular”, …);
UnMapAtom(Atom1, A, size);
MapAtom(Atom2, A, size);
ActivateAtom(Atom2);
```

Key approach - Atoms

- Atoms describe data which is semantically similar
- Programs explicitly specify atoms
- Atoms are immutable
- Atoms can be mapped to memory or deactivated
- Each virtual address maps to at most one atom

Key approach - Attributes of an atom

- Paper defines a specific set of attributes, but this can be extended
- *Data value properties* (eg. float32, sparse)
- *Access properties* (eg. accessed with specific stride, read only)
- *Data locality properties* (working set size and reuse for caching)

Mechanisms - Overview

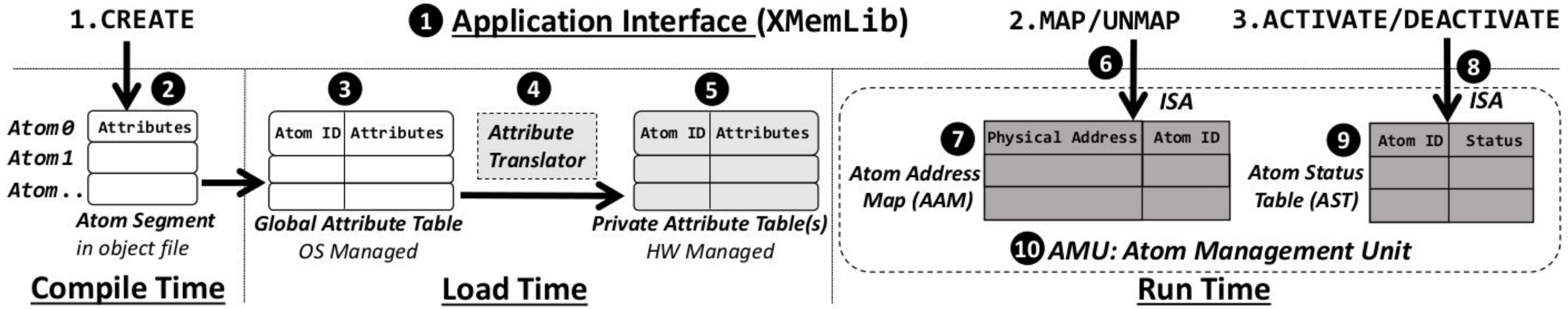


Figure 3: XMem: Overview of the components.

Key approach - Design choices

- Minimize runtime overhead of tracking and retrieving semantics
- Summarize atoms in software, track in hardware
- *Centralized tracking:* Atoms have an ID that the entire system recognizes
- *Attribute translation:* OS simplifies attributes for each hardware component

Mechanisms - XMemLib

- CreateAtom
  - Compiler populates the atom segment with passed attributes
  - OS loads the atom segment
- MapAtom and UnMapAtom
  - Translated to dedicated ISA instructions
  - AMU modifies the Atom Address Map
- ActivateAtom and DeactivateAtom
  - Translated to dedicated ISA instructions
  - AMU modifies the Atom Status Table

Mechanisms - malloc

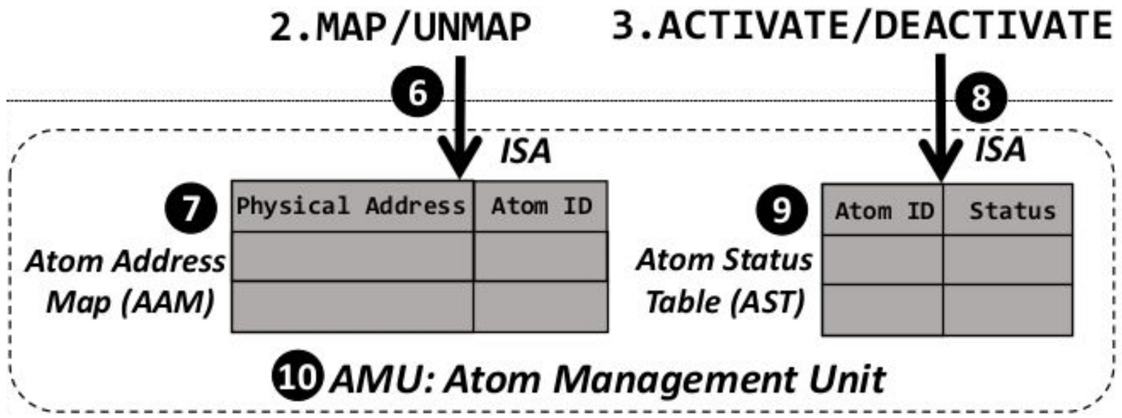
- Optimizations may require data placed at specific location in physical memory
- OS must know about atoms when allocating memory
- Atom ID is passed by compiler to malloc, and by malloc to the OS

```
A = malloc(size);
AtomMap(atomID, A, size)
```

$\longrightarrow$

```
A = malloc(size, atomID);
AtomMap(atomID, A, size)
```

Mechanisms - Atom Address Map (AAM)



- Uses PA instead of VA to simplify table design
- 512 byte granularity be default (~0.2% storage overhead)
- Continuous list of atom IDs indexed by physical address

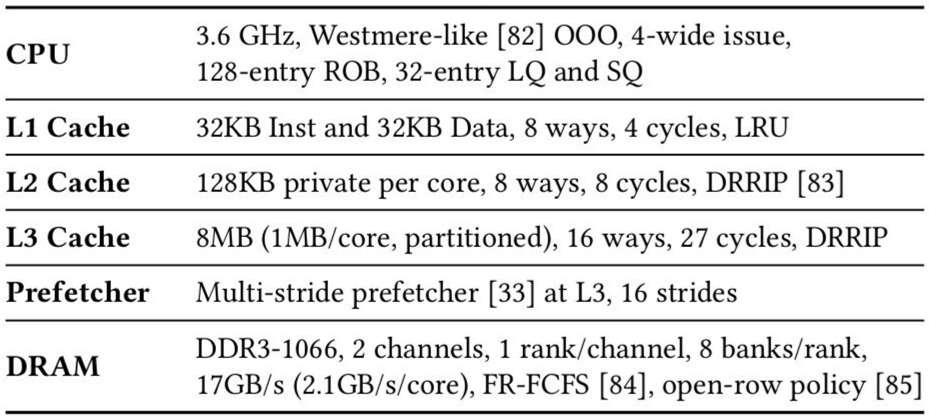
Mechanisms - Atom Management Unit (AMU)

- Hardware unit which manages the AAM and AST
- Handles ATOM_(UN)MAP and ATOM_(DE)ACTIVATE
- Handles ATOM_LOOKUP and has a lookaside buffer

Key results - Methodology

- XMem modelled in zsim and evaluated with DRAMSim2
- Two separate use cases evaluated

Table 3: Simulation configuration for Use Case 1.



CPU	3.6 GHz, Westmere-like [82] OOO, 4-wide issue, 128-entry ROB, 32-entry LQ and SQ
L1 Cache	32KB Inst and 32KB Data, 8 ways, 4 cycles, LRU
L2 Cache	128KB private per core, 8 ways, 8 cycles, DRRIP [83]
L3 Cache	8MB (1MB/core, partitioned), 16 ways, 27 cycles, DRRIP
Prefetcher	Multi-stride prefetcher [33] at L3, 16 strides
DRAM	DDR3-1066, 2 channels, 1 rank/channel, 8 banks/rank, 17GB/s (2.1GB/s/core), FR-FCFS [84], open-row policy [85]

Key results - Case 1 - Cache management

- Tests run against the Polybench suite
- Cache tiling optimization performed by PLUTO (polyhedral locality optimizer)
- XMem provides information on
  - Access pattern and intensity
  - Data reuse and working set size
- The hardware cache will
  - Prioritize keeping high-reuse data in the cache
  - Pin part of the working set if it doesn't fully fit in cache
- The prefetcher will
  - Prefetch data based on the provided access patterns

Key results - Case 1 - Cache management

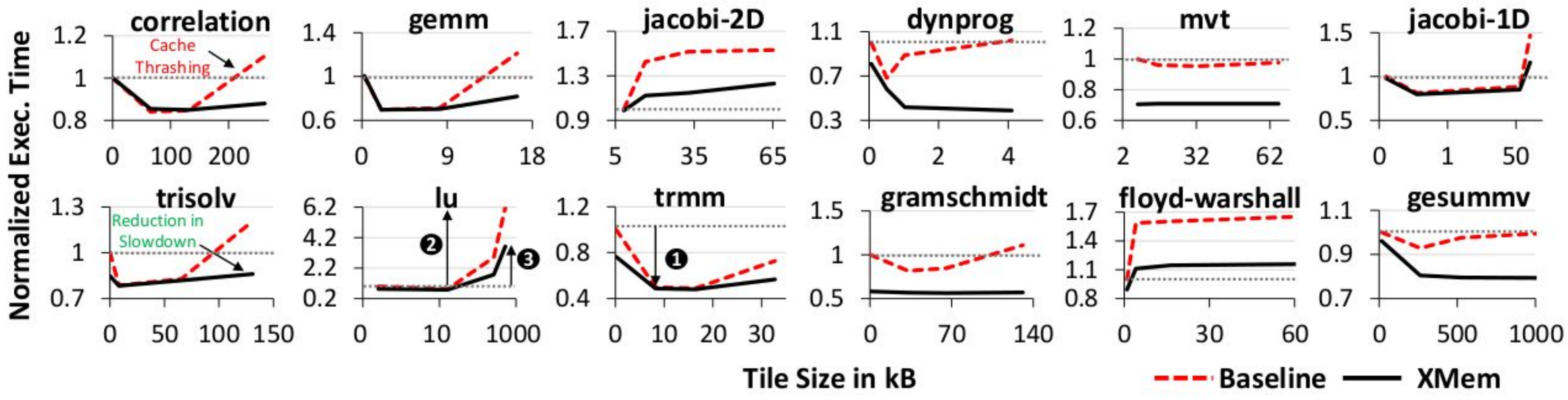


Figure 4: Execution time across different tile sizes (normalized to Baseline with the smallest tile size).

- Choosing too small tile size causes ~30% slowdown on average
- Choosing too large tile size causes thrashing (~65% slowdown)
- XMem reduces thrashing for ~25% slowdown

Key results - Case 1 - Cache management

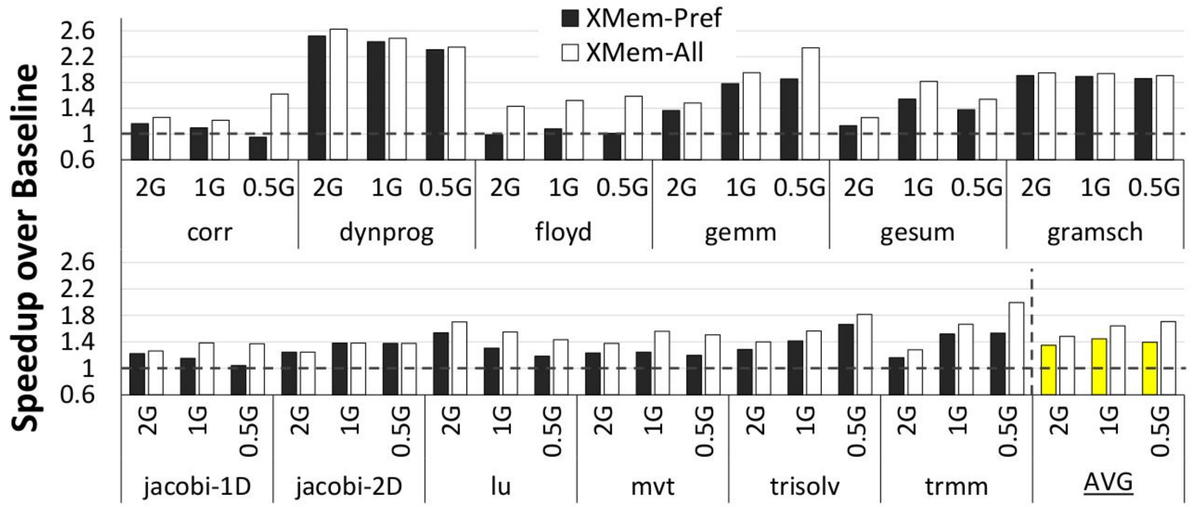


Figure 6: XMem's speedup over Baseline with different memory bandwidth availability.

- Both prefetching and pinning improvements contribute to performance

Key results - Case 2 - Data placement in DRAM

- Different set of workloads than in case 1
- XMem provides information on access pattern and intensity
- System provides information on DRAM configuration
- Goal is to improve RBL and MLP
- The OS will
  - Isolate high RBL data structures in their own banks
  - Spread out other data structures evenly
- Baseline system uses randomized virtual-to-physical address mapping

Key results - Case 2 - Data placement in DRAM

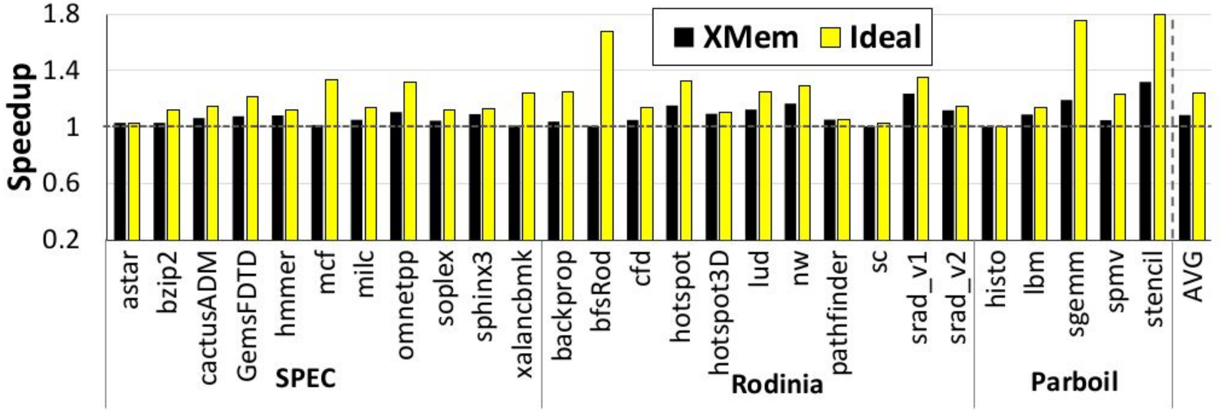


Figure 7: Speedup w/ XMem-based DRAM placement.

- XMem based DRAM placement improves runtime by ~8.5% on average
- Reduces read latency by ~12% on average
- Works by improving row buffer locality and memory level parallelism

Summary - XMem

- Problem
  - Optimizing program execution is difficult without hints
  - Fine-grained hints require large changes for each optimization
  - Platform-specific directives are not portable
- Goal
  - Create a general cross-layer interface to communicate higher-level program semantics
- Result - XMem
  - Enables performant memory optimizations using high-level information
  - Uses the atom abstraction to describe semantics of data
  - More general and versatile than past work
  - Low overhead by pre-processing in software and tracking in hardware

Strengths

- Simple and well explained concept
- Low overhead implementation with significant benefits
- Adopting XMem in future systems seems realistic

Weaknesses

- Unclear why both MAP and ACTIVATE are necessary
- Unclear which cache setup was used in tests
- XMem tightly couples guest and host in virtualized environments
- Effects of remapping atoms on malloc-integration not explored

Questions and discussion

Thoughts and discussion

- Could a similarly general and declarative approach be used for non-memory-related optimizations?
- Could the ACTIVATE/DEACTIVATE concept be removed entirely?
- Which XMem attributes could be inferred by a compiler? How effective would that be compared to a programmer specifying attributes?