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[ℵ] Nastaran Hajinazar[∂] Phillip B. Gibbons[†] Onur Mutlu[§†]

[†]Carnegie Mellon University [‡]University of Toronto [ℵ]NVIDIA
[∂]Simon Fraser University [§]ETH Zürich

ISCA June 2018.

Presentation by Max Striebel

Executive Summary

Motivation

Memory is the most performance critical part of most systems / applications

Problem

There is a semantic gap between higher-level program semantic and ISA

Observation

There are a lot of memory optimizations that could be enabled by knowing how the memory is used

Key Idea

Tag memory regions with properties that describe how the memory is being used

Evaluation

1) 31% average performance improvement when used for prefetching and cache management on low memory bandwidth system
2) 8.5% average performance improvement with intelligent DRAM placement

Conclusion

XMem provides a low overhead interface to bridge the semantic gap in order to enhance memory optimizations

Outline

- Background
  - Semantic gap
  - Current Situation
  - Caches
  - Prefetcher
- Observation
- Key Idea
- Implementation
- Evaluation

Background Semantic gap

A lot of knowledge about **memory usage** is lost during translation to machine code

- Programmer → High level language
  - Access **frequency**
  - Access **pattern**
- High level language → Machine code
  - Data types
  - Read-Write properties

Background Semantic gap Example

```
int sum(int *array, int length)
{
    int result = 0;
    for(size_t i = 0; i < length; ++i)
        result += array[i];
    return result;
}
```

```
loop:
        add     eax, [rdi]
        add     rdi, 4
        cmp     rdi, rdx
        jne     loop
        ret
```

Background Current situation

- ISA is almost exclusively concerned with **correctness**
  - Flexible microarchitecture
- Memory hierarchy almost completely abstracted away
  - Exceptions: OS, Prefetch instructions
- **Caches, prefetching**, branch predictions speculate about future

Background Current situation

- ISA is almost exclusively concerned with **correctness**
  - Flexible microarchitecture

Programmer has to know details about microarchitecture
in order to write optimal code

Caches, prefetching, branch predictions speculate about future

Background Current situation

- ISA is almost exclusively concerned with **correctness**
  - Flexible microarchitecture

Microarchitecture has to analyze behavior in real time

Caches, prefetching, branch predictions speculate about future

Background **Caches**

Fast but small memory on chip for caching **recently** and/or **frequently** used data

- Reduces memory access latency significantly
- Has to have a strategy for what data to **evict** (i.e. replacement policy)
- Makes use of spatial and temporal locality
- Size of cache can have a huge impact on performance
  - Cache trashing

Background **Prefetcher**

Tries to fetch memory before it is requested in order to reduce access latency

1) Analyses memory **access patterns**
2) Tries to **predict** next accessed memory
3) Loads this predicted memory into caches

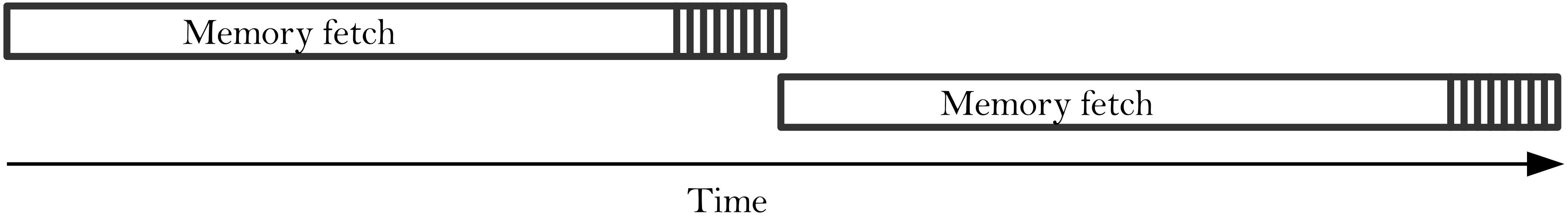
Outline

- Background
- Observation
  - Prefetcher
- Key Idea
- Implementation
- Evaluation

Observation Prefetcher Example

```
int sum(int *array, int length)
{
    int result = 0;
    for(size_t i = 0; i < length; ++i)
        result += array[i];
    return result;
}
```

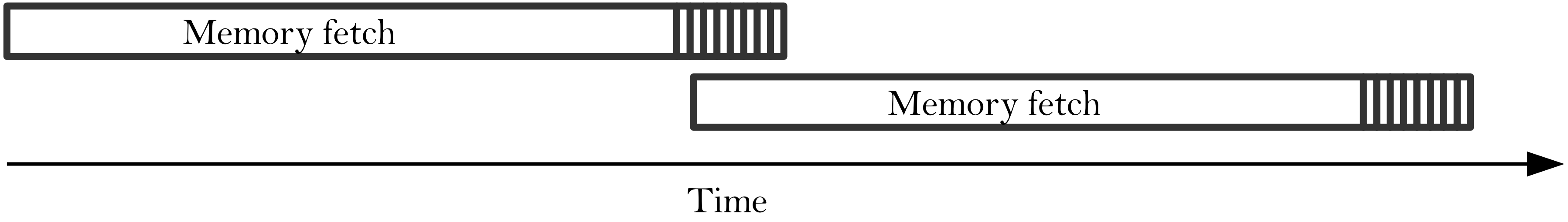
Without prefetcher



Observation Prefetcher Example

```
int sum(int *array, int length)
{
    int result = 0;
    for(size_t i = 0; i < length; ++i)
        result += array[i];
    return result;
}
```

With prefetcher



Memory fetch

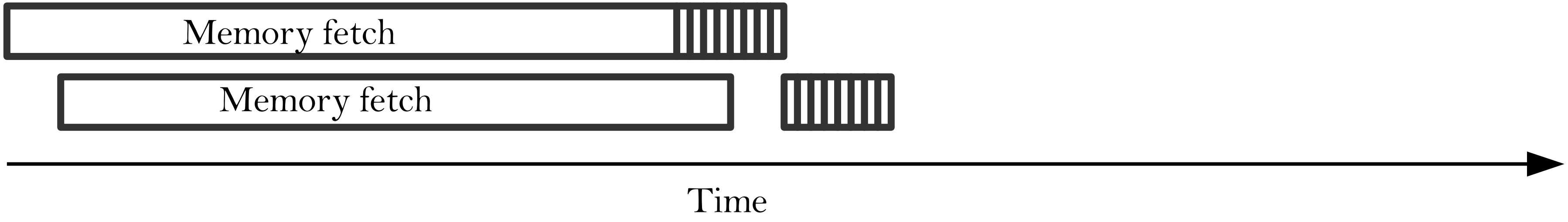
Memory fetch

Time

Observation Prefetcher Example

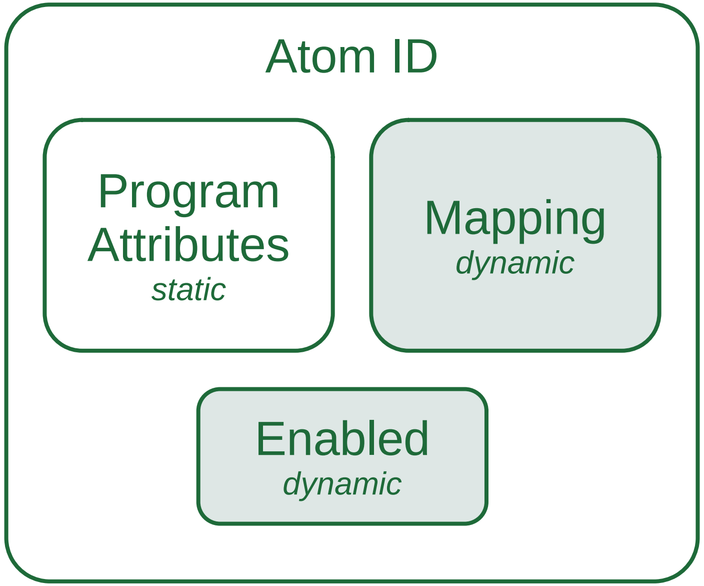
```
int sum(int *array, int length)
{
    int result = 0;
    for(size_t i = 0; i < length; ++i)
        result += array[i];
    return result;
}
```

Optimal (knowledge about access pattern)



Key idea

- Provide the OS and Hardware with more detailed information about intended memory usage
- Expressive Memory (XMem)
  - Create **atoms** that describe Program Attributes
  - Dynamically **map** and unmap memory regions
  - Have **hardware** support for keeping track of this mapping
- Create OS and Hardware optimizations that make use of this information



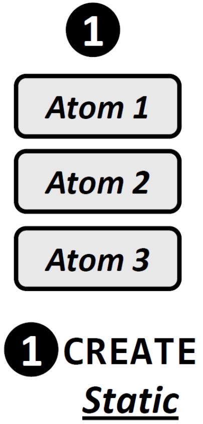
Outline

- Background
- Observation
- Key Idea
- Implementation
  - Key requirements
  - Atom
  - XmemLib
  - System
- Evaluation

Implementation Design goals

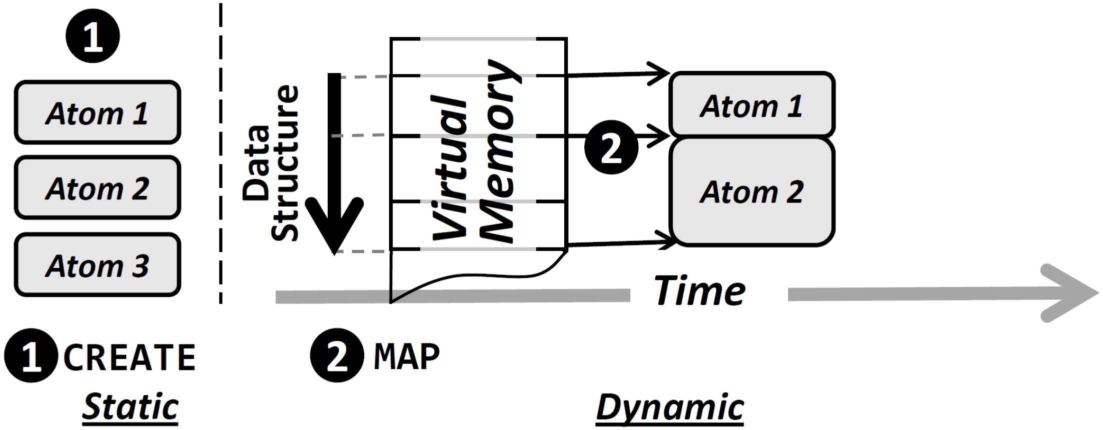
- No effect on functionality or **correctness**
  - Simpler implementation because information can be conveyed/stored imprecisely
- Architecture agnostic
  - Should improve performance on different platforms without knowledge about the specific microarchitecture
- General and extensible
  - Should work for a wide range of applications
  - Should allow for future extensions
- Low overhead

Implementation Atom



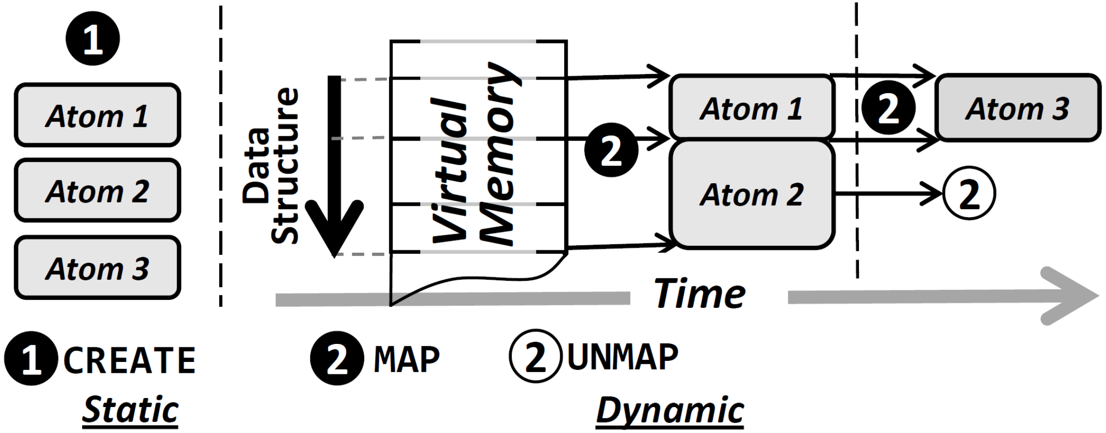
- Immutable Attributes
  - Atoms are **created** statically and can not change during run time
- Homogeneity
  - All data that maps to a specific atom has the **same** attributes

Implementation Atom



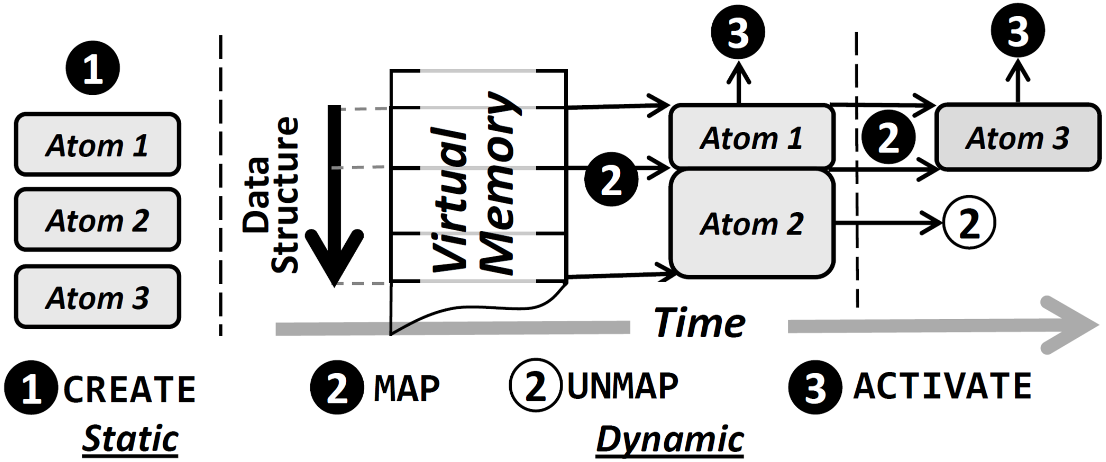
- Many-to-One PA-Atom Mapping
  - Each physical address can be **associated** with at most one atom
  - Fixed sized **granularity** of PAs that have the same atom assigned

Implementation Atom



- Dynamic mapping
  - Atoms can be **mapped** and unmapped dynamically to any (non-contiguous) memory regions

Implementation Atom



- Dynamic activation
  - Activate and deactivate atoms dynamically to effect all memory regions that are assigned to one atom at once

Implementation Atom attributes

- Data Value Properties [**compression**]
  - Data type (e.g., INT32, FLOAT32, CHAR)
  - Data properties (e.g., sparse, approximable, pointer, index)
- Access Properties [**prefetching**]
  - Regular, irregular, non-determent
- RWChar [**data placement**]
  - Read-only, write-only, read-write
- Access Intensity [**cache management**]
  - Access frequency relative to other data (0-255)
- Data Locality [**cache management**]
  - Working set size, reuse relative to other data

Implementation **XMemLib**

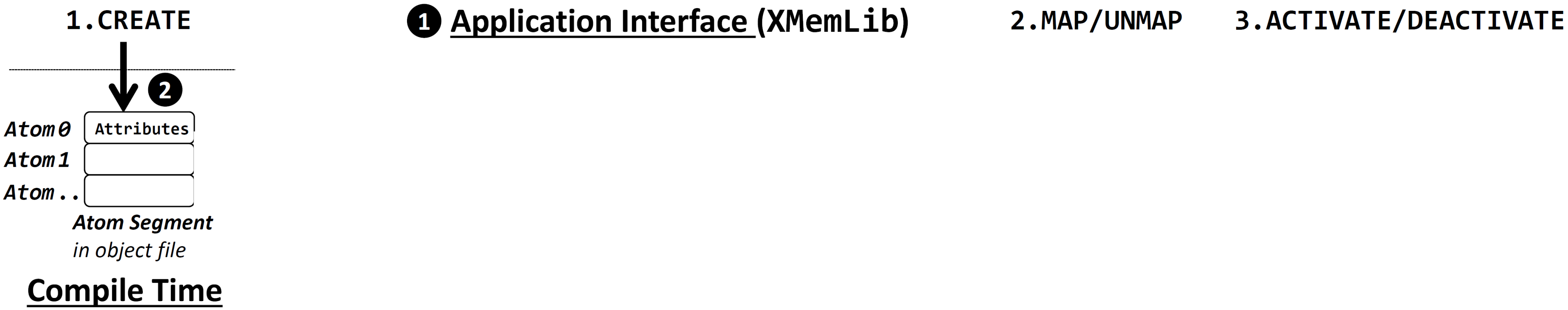
```
AtomID CreateAtom(data_prop, access_pattern, reuse, rw_characteristics);
void   AtomMap(atom_id, start_addr, size, map_or_unmap);
void   AtomActivate(atom_id);
void   AtomDeactivate(atom_id);
```

- **Library** that provides interface between XMem and application
- CreateAtom return **AtomID** (0-255) that uniquely identifies an **atom** (per process)
- Translates map and activation calls to direct **machine instructions**

Implementation System

`1.CREATE` ❶ **Application Interface (`XMemLib`)** `2.MAP/UNMAP` `3.ACTIVATE/DEACTIVATE`

Implementation System



1.CREATE

❶ Application Interface (XMemLib)

2.MAP/UNMAP

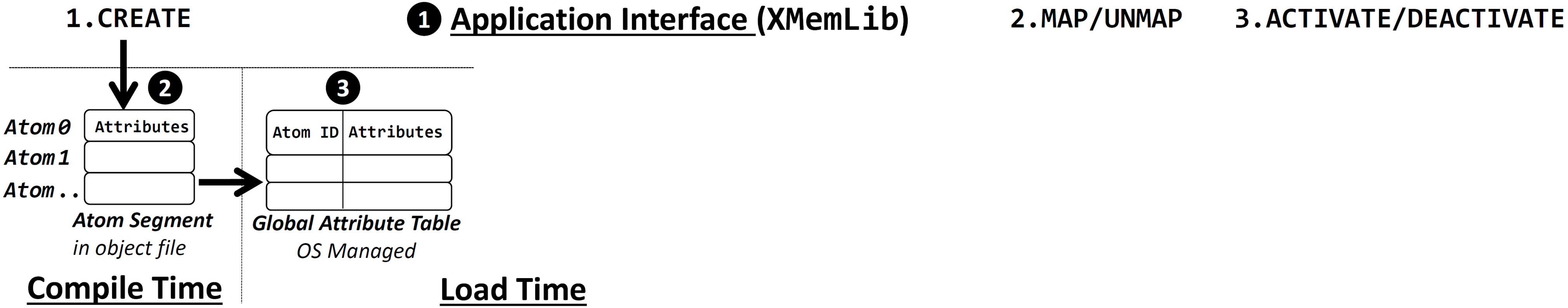
3.ACTIVATE/DEACTIVATE

❷

Compile Time

- Evaluate all CreateAtom call sites at **compile time**
- Create Atom Segment in object file

Implementation System



1.CREATE

❶ Application Interface (XMemLib)

2.MAP/UNMAP

3.ACTIVATE/DEACTIVATE

❷

Atom0

Atom1

Atom..

Attributes

Atom Segment
in object file

Compile Time

❸

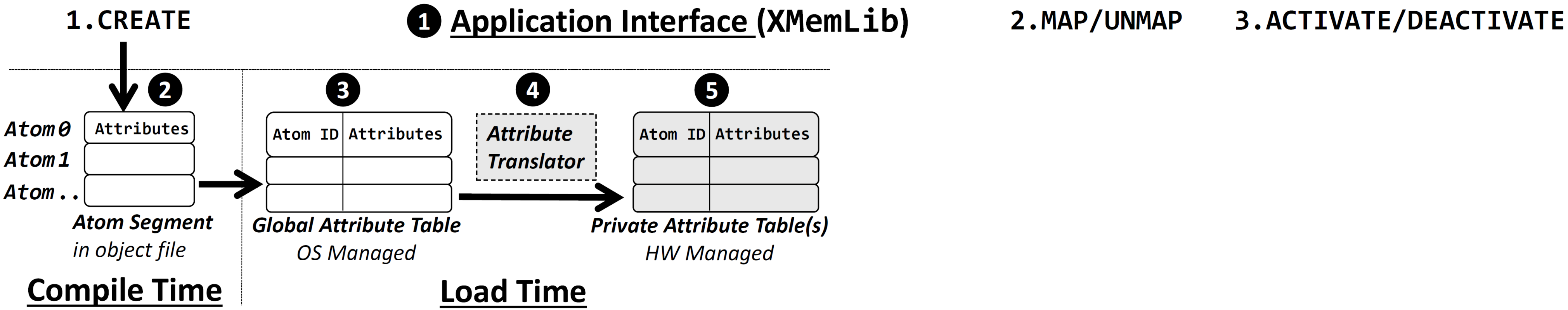
Atom ID	Attributes

Global Attribute Table
OS Managed

Load Time

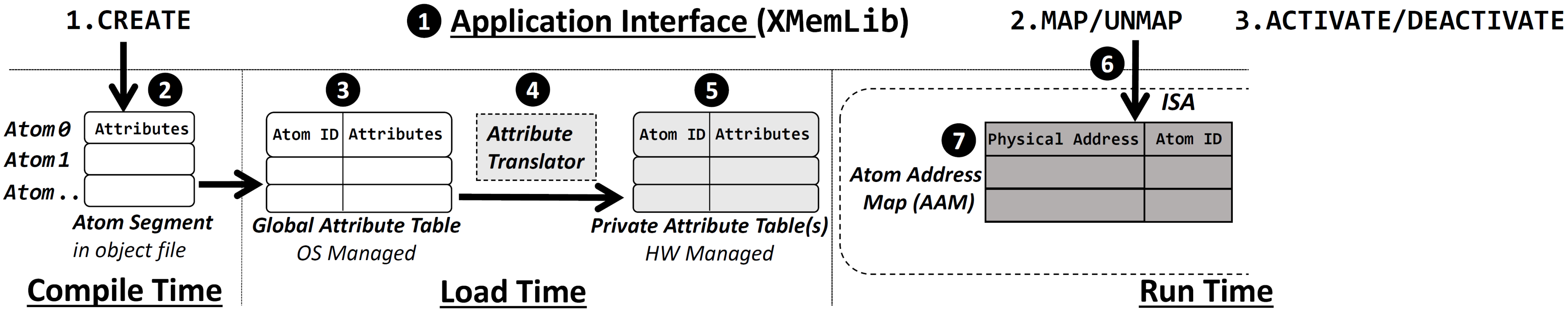
- During **load time** OS reads Atom Segment and creates Global Attribute Table in memory

Implementation System



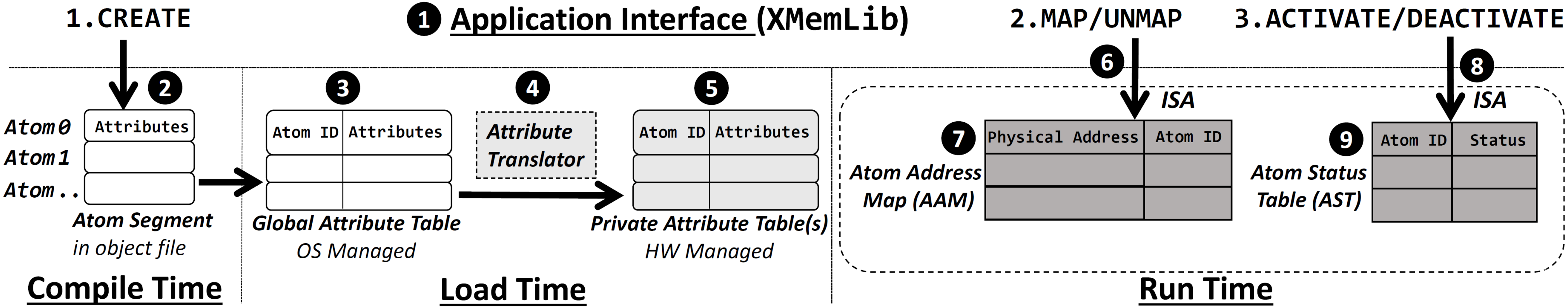
- OS **invokes** Attribute Translator during each context switch
- This supplies the relevant components with the needed attributes
  - Can be **tailored** for each microarchitecture
  - Version number provides backward and forward **compatibility**

Implementation System



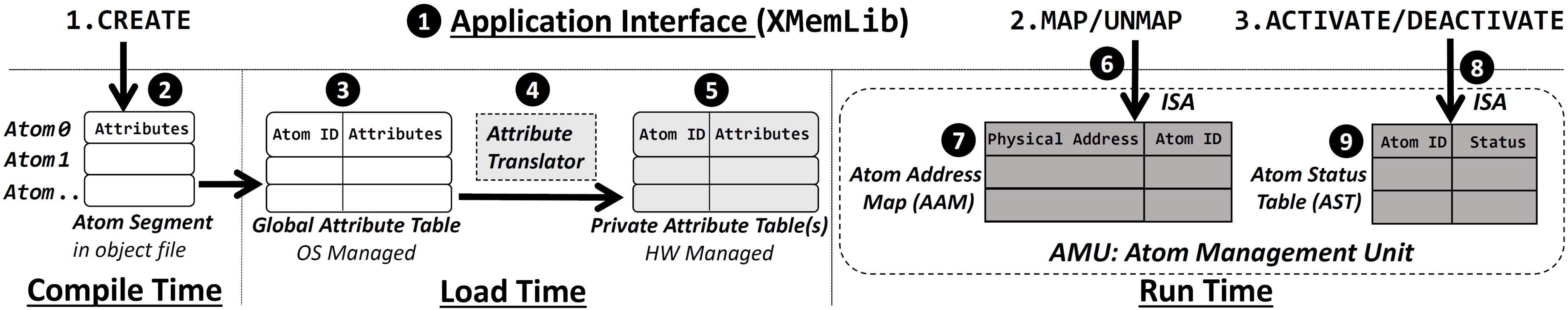
- Atom Address Map (AAM) maps each PA to an Atom
  - Proposed **resolution** of 8 cache lines (512 bytes)
  - Stored in memory
- Atom Lookaside Buffer (ALB) to **cache** AAM entries
  - 256-entry cover 98.9% of requests

Implementation System



- Atom Status Table (AST) stores active **status** for each Atom
  - For 256 possible AtomIDs only needs 32 bytes

Implementation System



- Atom Management Unit (AMU) handles atom **lookup requests**
  - Directly for hardware lookups
  - Indirectly through the MMU for OS requests

Outline

- Background
- Observation
- Key Idea
- Implementation
- Evaluation
  - Changes to HW/SW Stack
  - Overhead
  - Setup
  - Result

Evaluation Changes to HW/SW Stack

- Program / Library
- Compiler
- Linker / Object file specification
- OS
  - Program load
  - Context switch
  - (Memory layout)
- ISA
- Microarchitecture / (Memory controller)

Evaluation **Overhead**

- Memory **storage** overhead
  - Global Attribute Table (GAT)
    - 2.8KB per application assuming 256 atoms
  - Atom Address Map (AAM)
    - 0.2% physical memory assuming 512 byte granularity
    - Can be reduced by increasing granularity or reducing the number of atoms
- Hardware **area** overhead
  - Attribute Translator and Attribute Management Unit (AMU)
    - around 0.03% on modern chips

Evaluation **Overhead**

- Memory **storage** overhead
  - Global Attribute Table (GAT)

Small memory overhead of 0.2% that can be reduced further

- Hardware **area** overhead
  - Attribute Translator and Attribute Management Unit (AMU)
    - around 0.03% on modern chips

Evaluation **Overhead**

- Instruction overhead
  - Map/unmap and activate/deactivate **instructions**
    - 0.014% on average
    - 0.2% at most
- Context switch overhead
  - Extra **register** for storing address of Global Attribute Table
    - ≤1 nano seconds
  - Attribute Lookaside Buffer (ALB) **flushing** and **invoking** the Attribute Translator
    - ~700 nano seconds

Evaluation **Overhead**

- Instruction overhead
  - Map/unmap and activate/deactivate **instructions**

Small but noticable context switch overhead of around 15%

    - ≤1 nano seconds
  - Attribute Lookaside Buffer (ALB) **flushing** and **invoking** the Attribute Translator
    - ~700 nano seconds

Evaluation **Setup**

- Expressing key working sets
  - By mapping to atom with high **reuse** value
- Optimization algorithm
  - Greedy **insertion** and **prefetching** logic for deciding what data to **pin** and prefetch based on reuse value
- Support in cache controllers
  - 25% of the cache is **reserved** for default insertion policy
- Support in prefetchers
  - Uses Private Attribute Table (PAT) to keep track of access pattern (**stride**) and address ranges of pinned atoms

Evaluation **Setup**

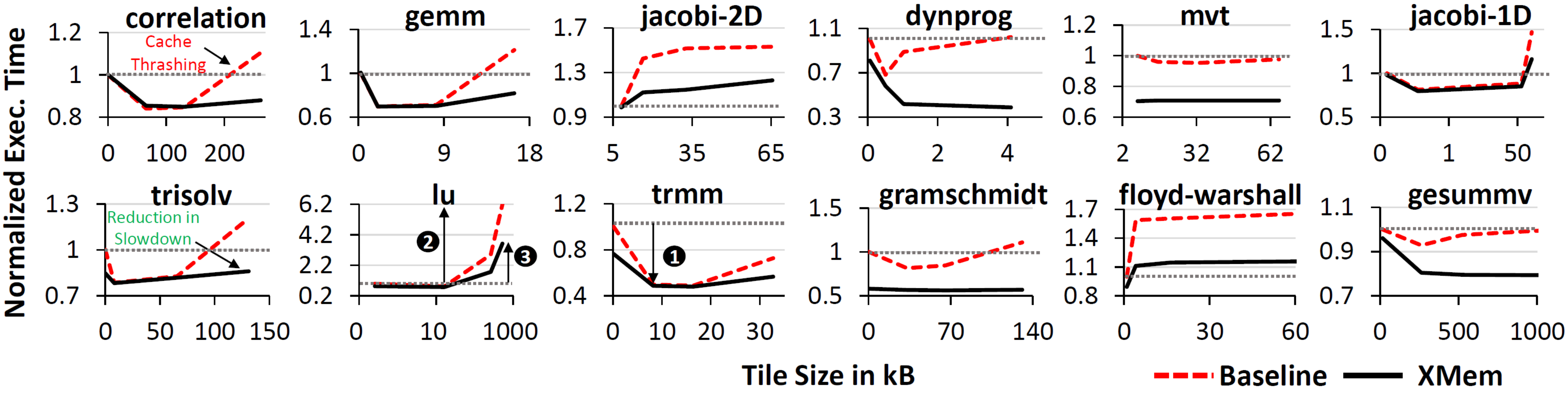
- Expressing key working sets
  - By mapping to atom with high **reuse** value
- Optimization algorithm
  - Greedy **insertion** and **prefetching** logic for deciding what data to **pin** and prefetch based on reuse value
- Support in cache controllers
  - 25% of the cache is **reserved** for default insertion policy
- Support in prefetchers
  - Uses Private Attribute Table (PAT) to keep track of access pattern (**stride**) and address ranges of pinned atoms

Evaluation Setup

- Evaluated different programs from the **Polybench** suit
  - Linear algebra, graph calculations
- Modeling and simulation using **zsim** and **DRAMSim2**
- Baseline uses high-performance cache replacement policy and a multi-stride prefetcher at L3

1) Test versions of the test programs that use different sized **tiles**
   - Expect to see drop in performance for suboptimal sizes (cache trashing)
2) Test under different memory bandwidth speeds
   - Expect to see a drop in performance the lower the **bandwidth**

Results Tile Sizes



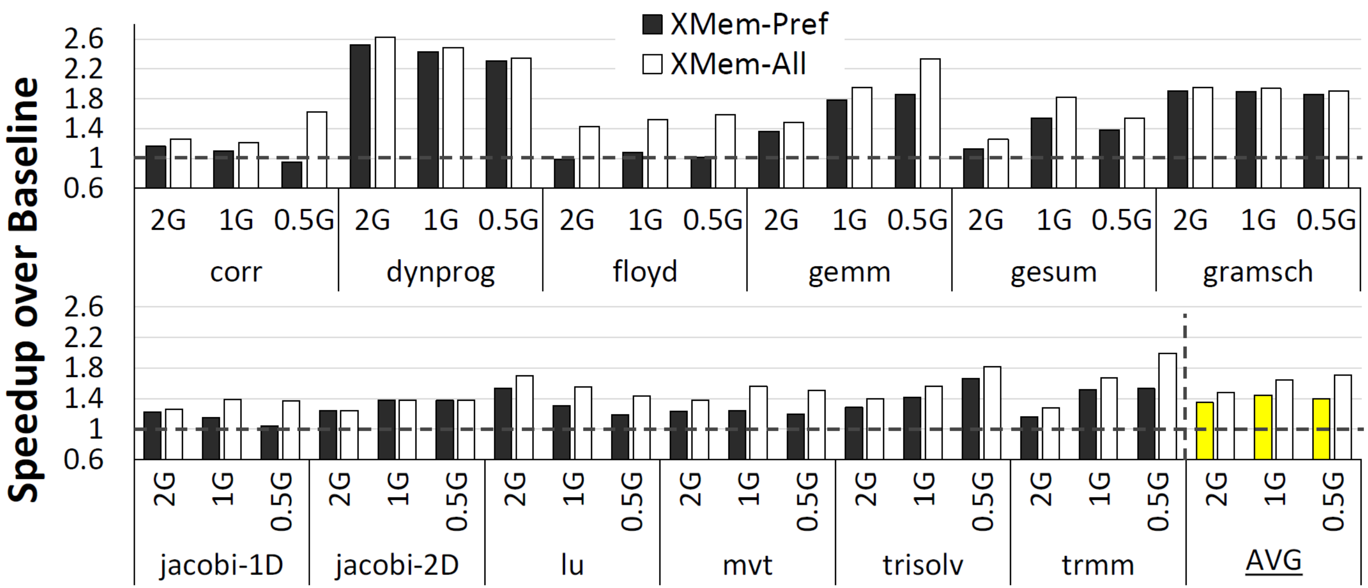
Execution time across different tile sizes (normalized to Baseline with the smallest tile size).

❶ Small tiles reduce **reuse** and result in an avg of 28.7% slowdown

❷ Cache **trashing** can lead to severe slowdowns (64.8 % avg up to 7.6x)

❸ XMem reduces cache trashing to 26.9% avg up to 4.6x

Results Memory bandwidth



- XMem-Pref similar to software based prefetching
- XMem can reduce memory traffic through data **pinning** and thus achieves higher speedups compared to Xmem-Pref in low bandwidth situations

Executive Summary

Motivation
Memory is the most performance critical part of most systems / applications

Problem
There is a semantic gap between higher-level program semantic and ISA

Observation
There are a lot of memory optimizations that could be enabled by knowing how the memory is used

Key Idea
Tag memory regions with properties that describe how the memory is being used

Evaluation

1) 31% average performance improvement when used for prefetching and cache management on low memory bandwidth system
2) 8.5% average performance improvement with intelligent DRAM placement

Conclusion
XMem provides a low overhead interface to bridge the semantic gap in order to enhance memory optimizations

Questions?

Strengths

- Tackles a major performance bottleneck in a novel way
- Enables **multiple** optimizations
- Allows for **portable** performance optimizations
- Minimal or even negative chip area overhead
- Has many clever details
  - PA-Atom mapping so that only one **global** AAM is required
  - Versioning and Attribute Translator to enable forward and backwards **compatibility**

Weaknesses

- A lot of **components** a cross the hierarchy have to be changed
- The create function in XMemLib gets evaluated at **compile time**
  - Unexpected function behavior
  - Requires compiler changes
- Allows for dynamic PA-Atom mapping, but some optimizations like DRAM placement require static mapping
- Source code / raw data are not publicly available

Ideas

- Create and submit GAT at run time by the library
  - Enables more dynamic **usages**
    - e.g. automatic testing of different atoms configurations
  - Library can be a pure library without changing the **compiler**
  - No changes to the object file specification
  - Only minimal OS support needed (context switch)
- Add atom attributes that indicates if memory region changes it's atom dynamically
  - Can be used by malloc and OS to make more informed decision on how to allocate memory

Discussion starters

How likely is it that this gets adopted?

- Where should we start? (chicken and egg problem)
- What incentives do the different decision makers have for adopting this?

Discussion starters

How likely is it that this gets adopted?

- Where should we start? (chicken and egg problem)
- What incentives do the different decision makers have for adopting this?

How would this influence the way software gets written/optimized?

- How could programming languages support this feature?
- What types of programs would benefit the most?

Discussion starters

How likely is it that this gets adopted?

- Where should we start? (chicken and egg problem)
- What incentives do the different decision makers have for adopting this?

How would this influence the way software gets written/optimized?

- How could programming languages support this feature?
- What types of programs would benefit the most?

Do we need new kinds of **diagnostics**?

Discussion starters

How likely is it that this gets adopted?

- Where should we start? (chicken and egg problem)
- What incentives do the different decision makers have for adopting this?

How would this influence the way software gets written/optimized?

- How could programming languages support this feature?
- What types of programs would benefit the most?

Do we need new kinds of **diagnostics**?

Ideas for improving on the presented ideas