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
Software

Functionality

Performance?

Hardware

SW Optimization

HW-SW Interfaces
(ISA & Virtual Memory)

HW Optimization
Higher-level information is not visible to HW

Data Structures

Code Optimizations

Access Patterns

Data Type

Hardware

Software

Instructions

Memory Addresses
Virtual Memory

Higher-level Program Semantics

Expressive Memory “XMem”

ISA
Virtual Memory

Performance

Functionality

Software

Hardware

Software
Hardware
Outline

Why do we need a richer cross-layer abstraction?

Designing Expressive Memory (XMem)

Evaluation (with a focus on one use case)
Performance optimization in hardware
What we do today: We design hardware to infer and predict program behavior to optimize for performance.
With a richer abstraction: SW can provide program information can significantly help hardware.
Benefits of a richer abstraction:

**Express:**
- Data structures
- Access semantics
- Data types
- Working set
- Reuse
- Access frequency
  
**Optimizations:**
- Cache Management
- Data Placement in DRAM
- Data Compression
- Approximation
- DRAM Cache Management
- NVM Management
- NUCA/NUMA Optimizations
  
....
Optimizing for performance in software
What we do today: Use **platform-specific optimizations** to tune SW

Example: SW-based cache optimizations

SW optimizations make assumptions regarding HW resources

Significant portability and programmability challenges
With a richer interface: HW can alleviate burden on SW

Working set

Reuse

SW only expresses program information

System/HW handles optimizing for specific system details

e.g. exact cache size, memory organization, NUMA organization
Benefits of a richer abstraction:

**Express:**
- Data structures
- Access semantics
- Data types
- Working set
- Reuse
- Access frequency

**Optimizations:**
- Cache Management
- Data Placement in DRAM
- Data Compression
- Approximation
- DRAM Cache Management
- NVM Management
- NUCA/NUMA Optimizations

**HW optimizations**
- ✓ Performance

**SW optimizations**
- ✓ Programmability
- ✓ Portability
SW information in HW is proven to be useful, but..

Lots of research on hints/directives to hardware and HW-SW co-designs
  Cache hints, Prefetcher hints, Annotations for data placement, ...

Downsides:
  Not scalable – can’t add new instructions for each optimization
  Not portable – make assumptions about underlying resources

These downsides significantly limit adoption of otherwise useful approaches
Our Goal:
Design a rich, general, unifying abstraction between HW and SW for performance
Outline

Why do we need a richer cross-layer abstraction?

Designing Expressive Memory (XMem)

Evaluation (with a focus on one use case)
Key design goals

Supplemental and hint-based only

General and extensible

Architecture-agnostic

Low overhead
An Overview of Expressive Memory

System to summarize and save program information

Application

Expressive Memory

Interface to Application

Interface to System/Architecture

OS
Cache

Memory Controller

Prefetcher
DRAM
Cache

...
Challenge 1: Generality and architecture-agnosticism

Data structures, data type, access patterns, …

What to prefetch? Which data to cache?

Application

Interface to Application

Expressive Memory

Interface to System/Architecture

OS Cache Memory Controller Prefetcher DRAM Cache …

General, High-level

Architecture-specific, Low-level
Challenge 2: Tracking changing program properties with low overhead

Program behavior keeps changing:
- Data structures are accessed differently in different phases
- New data structures are allocated

Dynamic interface that continually tracks program behavior

We want to convey lots of information!

Potentially very high storage/communication overhead at run time.
A new HW-SW abstraction

Software

Atom:1

Memory Region

Application

Atom: 2

Memory Region

Hardware

Atom: 3

Memory Region

System/Architecture
The Atom: A closer look

A hardware-software abstraction to convey program semantics

1) Data Value Properties:
   INT, FLOAT, CHAR,…
   COMRESSIBLE, APPROXIMABLE

2) Access Properties:
   Read-Write Characteristics
   Access Pattern
   Access Intensity (“Hotness”)

3) Data Locality:
   Working Set
   Reuse

4) ….
The three Atom operators

1) CREATE
2) MAP/UNMAP
3) ACTIVATE/DEACTIVATE
Using Atoms to express program semantics

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);

Attributes cannot be changed
Implementing the Atom

Compile Time (CREATE)

Load Time (CREATE)

Run Time (MAP and ACTIVATE)
Compile Time (CREATE)

A = malloc ( size );
Atom1 = CreateAtom("INT", "Regular", ...);

High overhead operations are handled at compile time

Atom2 = CreateAtom("INT", "Irregular", ...);
UnMapAtom( Atom1, A, size );
MapAtom( Atom2, A, size );
ActivateAtom(Atom2);
Load Time (CREATE)

Architecture-agnostic, general

Architecture-specific, low-level

Atom Segment in Object File

Attribute Translator

OS

Atom ID
Attributes

Cache Controllers
Prefetcher
Memory Controller

Architecture-agnostic, general

Architecture-specific, low-level
Run Time (MAP and ACTIVATE)

$A = \text{malloc} \ (\text{size});$

$\text{Atom1} = \text{CreateAtom} ("\text{INT}", "\text{Regular}", \ldots);$  

$\text{MapAtom}(\text{Atom1}, A, \text{size});$

$\text{ActivateAtom}(\text{Atom1});$

$\ldots$

$\ldots$

$\text{Atom2} = \text{CreateAtom} ("\text{INT}", "\text{Irregular}", \ldots);$  

$\text{UnMapAtom}(\text{Atom1}, A, \text{size});$

$\text{MapAtom}(\text{Atom2}, A, \text{size});$

$\text{ActivateAtom}(\text{Atom2});$

Design challenge: How to do this with low overhead?
Architectural support

For a performance abstraction: low overhead is critical

No impact on functionality (hint-based): We can heavily optimize for low overhead
Outline

Why do we need a richer cross-layer abstraction?

Designing Expressive Memory (XMem)

Evaluation (with a focus on one use case)
A fresh approach to traditional optimizations

Express:
- Data structures
- Access semantics
- Data types
- Working set
- Reuse
- Access frequency
- ...

Optimizations:
- Cache Management
- Data Placement in DRAM
- Data Compression
- Approximation
- DRAM Cache Management
- NVM Management
- NUCA/NUMA Optimizations
- ...

HW optimizations
- ✔ Performance

SW optimizations
- ✔ Programmability
- ✔ Portability
Use Case 1: Improving portability of SW cache optimization

SW-based cache optimizations try to fit the working set in the cache

Examples: hash-join partitioning, cache tiling, stencil pipelining
Methodology (Use Case 1)

**Evaluation Infrastructure:** Zsim, DRAMSim2  
**Workloads:** Polybench  
**System Parameters:**

- **Core:** 3.6 GHz, Westmere-like OOO, 4-wide issue, 128-entry ROB  
- **L1 Cache:** 32KB Inst and 32KB Data, 8 ways, 4 cycles, LRU  
- **L2 Cache:** 128KB private per core, 8 ways, 8 cycles, DRRIP  
- **L3 Cache:** 8MB (1MB/core, partitioned), 16 ways, 27 cycles, DRRIP  
- **Prefetcher:** Multi-stride prefetcher at L3, 16 strides  
- **Memory:** DRAM DDR3-1066, 2 channels, 1 rank/channel, 8 banks/rank, 17GB/s (2.1GB/s/core), FR-FCFS, open-row policy
Correctly sizing the working set is critical

Optimal tile size depends on available cache space:
This causes portability and programmability challenges
Leveraging Expressive Memory for cache tiling

Map tile to an atom, specifying **high reuse** and **tile size**

If tile size < available cache space: **default policy**
If tile size > available cache space: **pin a part of the tile, prefetch the rest** (avoid thrashing)

SW expresses program-level semantic information
HW manages cache space to optimize for performance
Cache tiling with Expressive Memory

Knowledge of locality semantics enables more intelligent cache management. Improves portability and programmability.
Results across more workloads

Normalized Exec. Time

Tile Size in kB

correlation

gemm

jacobi-2D

dynprog

mvt

jacobi-1D

trisolv

lu

trmm

gramschmidt

floyd-warshall

gesummv

Baseline

Expressive Memory
More in the paper

Use Case 2: Leveraging data structure semantics to enable more intelligent OS-based page placement

More details on the implementation

Overhead analysis

Other use cases of XMem
Conclusion

General and architecture-agnostic interface to SW to express program semantics

Expressive Memory “XMem”

Higher-level Program Semantics

Key program information to aid system/HW components in optimization
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
MetaSys
A Practical Open-Source Metadata Management System
to Implement and Evaluate Cross-Layer Optimizations

Nandita Vijaykumar

Ataberk Olgun, Konstantinos Kanellopoulos, F. Nisa Bostanci
Hasan Hassan, Mehrshad Lotfi, Phillip B. Gibbons, Onur Mutlu

SAFARI
Executive Summary

Problem

• Cross-layer techniques are challenging to implement because they require full-stack changes
• Existing open-source infrastructures for implementing cross-layer techniques are not designed to provide key features:

Key Idea – Provide:

• Rich dynamic HW/SW interfaces
• Low-overhead metadata management
• Interfaces to key hardware components (e.g., prefetcher)

Our goal is twofold:

1. Develop an efficient and flexible framework to enable rapid implementation of new cross-layer techniques
2. Perform a detailed limit study to quantify the overheads associated with general metadata systems
FPGA Prototype

Prototype on Xilinx Zedboard within a real RISC-V system (Rocket Chip)
MetaSys in Rocket Chip

Implement two main components:

1. Atom Controller
   • Manages the attribute table (CREATE – (DE)ACTIVATE)
   • Performs atom mapping (MAP/UNMAP)
     • Physical address → Atom ID

2. Metadata Lookup Unit
   • Responds to clients:
     • Provides atom attributes
     • Contains the metadata mapping cache
Changes in Rocket Chip

- Core
- Co-Processor
- TLB
- L1 Cache

- Atom Controller
  - Attribute Table
- Metadata Lookup Unit
  - Metadata Mapping Cache

- Co-processor

- User Program
- Optimization Client
- Lookup IO
  - I: Address
  - O: Attributes

- Instructions
Source on Github

https://github.com/CMU-SAFARI/MetaSys

MetaSys

We refer the developers of the MetaSys repository to metasys_readme.md, where we describe our modifications to the existing rocket-chip code base, and present a walkthrough of an implementation of the prefetching use case described in our paper.
Evaluation Methodology

Run workloads on MetaSys prototype (Zedboard):

**Microbenchmarks**: Represent a variety of memory access patterns  
**Polybench**: Scientific computation kernels  
**Ligra**: Graph workloads
Performance Overhead

Metadata lookups occur low performance overheads
2.7% on average
MMC can cover ~81% of all memory requests on average
Impact of Tagging Granularity

Performance impact increases with finer granularity
Effect of Contention

**One Client:** All memory requests originating from rocket core

**Two Clients:** One client + all memory requests originating from the page table walker

Multiple clients do not significantly affect performance
(0.3% overhead on average)
MetaSys: A Practical Open-Source Metadata Management System to Implement and Evaluate Cross-Layer Optimizations

Nandita Vijaykumar
Ataberk Olgun, Konstantinos Kanellopoulos, F. Nisa Bostanci
Hasan Hassan, Mehrshad Lotfi, Phillip B. Gibbons, Onur Mutlu

SAFARI
Backup
Table 1. MetaSys instructions.

<table>
<thead>
<tr>
<th>MetaSys Operator</th>
<th>MetaSys ISA Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>CREATE(ClientID, TagID, Metadata)</td>
</tr>
<tr>
<td>(UN)MAP</td>
<td>(UN)MAP TagID, start_addr, size</td>
</tr>
<tr>
<td></td>
<td>(UN)MAP2D TagID, start_addr, lenX, sizeX, sizeY</td>
</tr>
<tr>
<td></td>
<td>(UN)MAP3D TagID, start_addr, lenX, lenY, sizeX, sizeY, sizeZ</td>
</tr>
</tbody>
</table>

Table 2. The MetaSys software library function calls.

<table>
<thead>
<tr>
<th>Library Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE(ClientID, TagID, &quot;meta&quot;)</td>
<td>ClientID -&gt; PMT[TagID] = &quot;metadata&quot;</td>
</tr>
<tr>
<td>MAP(start*, end*, TagID)</td>
<td>MMT[start...end] = TagID</td>
</tr>
<tr>
<td>UNMAP(start*, end*)</td>
<td>MMT[start...end] = 0</td>
</tr>
</tbody>
</table>
Table 3. Comparison between MetaSys and XMem interfaces.

<table>
<thead>
<tr>
<th>Operator</th>
<th>XMem [164]</th>
<th>MetaSys</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>Compiler pragma to communicate static metadata at program load time.</td>
<td>Selects a hardware optimization, dynamically associates metadata with an ID, and communicates both to hardware at runtime (implemented as a new instruction).</td>
</tr>
<tr>
<td>(UN)MAP</td>
<td>Associate memory ranges with tag IDs (implemented as new instructions).</td>
<td>Same semantics and implementation as XMem.</td>
</tr>
<tr>
<td>(DE)ACTIVATE</td>
<td>Enable/disable optimizations associated with a tag ID (implemented as new instructions).</td>
<td><strong>Does not exist</strong> as the same functionality can now be done with CREATE.</td>
</tr>
</tbody>
</table>
Fig. 3. Data-dependent accesses in vertex-centric graph processing model (left), speedup with the MetaSys prefetcher (right).
Backup

```c
/* Additional Code in BFS */
metadata_create(0, 1, WorkList, sizeof(WorkList), VertexList,Stride); // Create metadata (arguments: ClientID=0, tag ID, metadata)
metadata_create(0, 2, VertexList, sizeof(VertexList), EdgeList, Stride);
metadata_create(0, 3, EdgeList, sizeof(EdgeList), VertexList, Stride);
metadata_create(0, 4, Property, sizeof(Property), NULL, Stride);
metadata_map(WorkList, mapSize, 1); // Map tag 1 to Worklist
metadata_map(VertexList, mapSize, 2);
metadata_map(EdgeList, mapSize, 3);
metadata_map(Property, mapSize, 4);

/* Hardware Prefetcher Functionality */
void snoop_mem_request ( address ); // snoop every memory request
    (Valid,Base,Bounds,PointerToNextDS,Stride) = metadata_lookup (address); // Access MetaSys using the address

while (Valid && PointerToNext != NULL) // While the data structure traversal is not complete
    if (Base < address && address > Bounds) // If the memory request comes from a tracked data structure
        initiate_prefetch(address+stride); // Initiate a stride prefetch request
        Value = wait_for_value(address+stride); // Wait for the prefetch request to return data
        address = &PointerToNextDS[value]; // Discover the address of the next data structure (DS)
    (Valid,Base,Bounds,PointerToNextDS,Stride) = metadata_lookup (PointerToNextDS[value]); // Look up metadata for the next DS

Listing 1. Metasys-based Graph Prefetcher. Available online [57].
Listing 2. MetaSys-based bounds checking example. Full source code is available online [57].
Fig. 4. Performance overheads for (left) bounds checking, (right) return address protection.
Fig. 6. Additional memory accesses introduced by MetaSys metadata lookups.
Backup

Fig. 8. MetaSys performance overhead on systems with varying amounts of memory bandwidth.
Fig. 12. Performance overhead with no address translation overhead for metadata.
Fig. 14. Alleviating MMC contention in microbenchmarks.
Fig. 15. Performance overhead of MAP/CREATE instructions.