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
Functionality

HW-SW Interfaces
(ISA & Virtual Memory)

Software

Hardware
Software

Functionality

HW-SW Interfaces
(ISA & Virtual Memory)

Hardware

SW Optimization

HW Optimization
Higher-level information is not visible to HW

Software

Data Structures

Code Optimizations

Access Patterns

Hardware

Data Type

Integer

Float

Char

100011111...

101010011...

Instructions

Memory Addresses
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

Data Structures

Access Patterns

Data Type/Layout

Integer

Float

Char

Software

Hardware

Data Placement

Prefetcher

Data Compression

Software

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

Tune working set

8MB cache
What we do today: Use platform-specific optimizations to tune SW

Example: SW-based cache optimizations

Tune working set → 8MB cache
What we do today: **Use platform-specific optimizations to tune SW**

Example: **SW-based cache optimizations**

- Tune working set
- SW optimizations make assumptions regarding HW resources
- Significant portability and programmability challenges
What we do today: Use platform-specific optimizations to tune SW

Example: SW-based cache optimizations

Tune working set

8MB cache

6MB cache

SW optimizations make assumptions regarding HW resources

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

SW only expresses program information

System/HW handles optimizing for specific system details e.g. exact cache size, memory organization, NUMA organization

Working set Reuse
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

Interface to Application

Expressive Memory

Interface to System/Architecture

OS
Caches
Memory Controller
Prefetcher
DRAM Cache

...
Challenge 1: Generality and architecture-agnosticism

Data structures, data type, access patterns, ...

What to prefetch? Which data to cache?

OS Caches Memory Controller Prefetcher DRAM Cache ...

Application

Interface to Application

Expressive Memory

Interface to System/Accructure

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 → Atom: 2 → Memory Region → Atom: 3 → Memory Region

Hardware

System/Architecture
The Atom: A closer look

A hardware-software abstraction to convey program semantics

1) Data Value Properties:
   - INT, FLOAT, CHAR, ...
   - COMPRESSIBLE, APPROXIMABLE

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

3) Data Locality:
   - Working Set
   - Reuse

4) ....

Valid/invalid at current execution point
The three Atom operators

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

\[
\begin{align*}
A & = \text{malloc}\ (\text{size})
\end{align*}
\]

\[
\begin{align*}
\text{Atom1} & = \text{CreateAtom}(\text{"INT"}, \text{"Regular"}, \ldots) \\
\text{MapAtom}(\text{Atom1, A, size}) & \\
\text{ActivateAtom}(&\text{Atom1})
\end{align*}
\]
Using Atoms to express program semantics

```c
A = malloc( size );
Atom1 = CreateAtom( "INT", "Regular", ...);
MapAtom( Atom1, A, size );
ActivateAtom(Atom1);
....
....
```

Attributes cannot be changed
Using Atoms to express program semantics

\[ A = \text{malloc}\(\text{ size}\); \]
\[ \text{Atom1} = \text{CreateAtom}(\text{“INT”, “Regular”, ...}); \]
\[ \text{MapAtom} (\text{Atom1, A, size}); \]
\[ \text{ActivateAtom(Atom1)}; \]
\[ \ldots \]
\[ \ldots \]
\[ \text{Attributes cannot be changed} \]
\[ \text{Atom2} = \text{CreateAtom}(\text{“INT”, “Irregular”, ...}); \]
\[ \text{UnMapAtom(Atom1, A, size);} \]
\[ \text{MapAtom(Atom2, A, size);} \]
\[ \text{ActivateAtom(Atom2);} \]
Implementing the Atom

Compile Time (CREATE)

Load Time (CREATE)

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

\[ A = \text{malloc (size)}; \]
\[ \text{Atom1} = \text{CreateAtom(“INT”, “Regular”, …);} \]
\[ \text{MapAtom(Atom1, A, size);} \]
\[ \text{ActivateAtom(Atom1);} \]
\[ \ldots \]
\[ \ldots \]
\[ \text{Atom2} = \text{CreateAtom(“INT”, “Irregular”, …);} \]
\[ \text{UnMapAtom(Atom1, A, size);} \]
\[ \text{MapAtom(Atom2, A, size);} \]
\[ \text{ActivateAtom(Atom2);} \]
Compile Time (CREATE)

\[
A = \text{malloc}(\text{size});
\]

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

High overhead operations are handled at compile time

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

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

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

\[
\text{ActivateAtom(Atom2);}
\]
Load Time (CREATE)

Architecture-agnostic, general

Atom Segment in Object File

Architecture-specific, low-level

OS

Attribute Translator

Atom ID | Attributes
--- | ---

Cache Controllers
Prefetcher
Memory Controller

...
Run Time (MAP and ACTIVATE)

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);

Design challenge: How to do this with low overhead?

New insts in ISA

Application

Express in Memory

OS
Caches
Memory Controller
Prefetcher
DRAM Cache

...
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

39
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 000, 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

![Graph showing the relationship between tile size and cache thrashing](image)

Bigger is better (more reuse)

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)
Leveraging Expressive Memory for cache tiling

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

*SW expresses program-level semantic information*

*HW manages cache space to optimize for performance*

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)
Cache tiling with Expressive Memory

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

- **correlation**
- **gemm**
- **jacobi-2D**
- **dynprog**
- **mvt**
- **jacobi-1D**
- **trisolv**
- **lu**
- **trmm**
- **gramschmidt**
- **floyd-warshall**
- **gesummv**

---

**Normalized Exec. Time**

- **Tile Size in kB**

---

**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

Software

Higher-level Program Semantics

Expressive Memory “XMem”

Hardware
Conclusion

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

Software

Higher-level Program Semantics

Expressive Memory “XMem”

Virtual Memory

ISA

Key program information to aid system/HW components in optimization

Hardware
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

Carnegie Mellon University
UNIVERSITY OF TORONTO
SFU
NVIDIA
ETH Zürich