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

HW-SW Interfaces (ISA & Virtual Memory)

Performance?

Hardware

SW Optimization

HW Optimization
Higher-level information is not visible to HW

Software

Hardware

Code Optimizations

Access Patterns

Data Structures

Data Type

Integer

Float

Char

Instructions

Memory Addresses

100011111...

101010011...
Software

Performance

Functionality

ISA
Virtual Memory

Higher-level Program Semantics

Expressive Memory “XMem”

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

Tune working set → 8MB cache

→ SW optimizations make assumptions regarding HW resources

→ Significant portability and programmability challenges

→ 6MB cache
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
Challenge 1: Generality and architecture-agnosticism

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

What to prefetch? Which data to cache?

Application

Expressive Memory

Interface to Application

Interface to System/Architecture

General, High-level

Architecture-specific, Low-level

OS  Caches  Memory Controller  Prefetcher  DRAM Cache
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

Application

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) ....

Unique Atom ID

Atom: X

Program Attributes: Valid/invalid at current execution point

Mapping

State
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);
....
....
Attributes cannot be changed
Atom2 = CreateAtom("INT", "Irregular", ...);
UnMapAtom(Atom1, A, size);
MapAtom(Atom2, A, size);
 ActivateAtom(Atom2);
```
Implementing the Atom

Compile Time (CREATE)

Load Time (CREATE)

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

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

Atom1 = \textcolor{red}{\text{CreateAtom}}("\text{INT}\), "\text{Regular}\), ...);\]

---

High overhead operations are handled at compile time

Atom2 = \textcolor{red}{\text{CreateAtom}}("\text{INT}\), "\text{Irregular}\), ...);

\text{UnMapAtom}(\text{Atom1}, \text{A}, \text{size});
\text{MapAtom}(\text{Atom2}, \text{A}, \text{size});
\text{ActivateAtom}(\text{Atom2});

Atom Segment in Object File
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)

\[ A = \text{malloc}\,(\text{size}); \]
\[ \text{Atom1} = \text{CreateAtom}(\text{"INT"}, \text{"Regular"}, \ldots); \]
\[ \text{MapAtom}(\text{Atom1}, A, \text{size}); \]
\[ \text{ActivateAtom}(\text{Atom1}); \]
\[ \ldots \]
\[ \ldots \]
\[ \text{Atom2} = \text{CreateAtom}(\text{"INT"}, \text{"Irregular"}, \ldots); \]
\[ \text{UnMapAtom}(\text{Atom1}, A, \text{size}); \]
\[ \text{MapAtom}(\text{Atom2}, A, \text{size}); \]
\[ \text{ActivateAtom}(\text{Atom2}); \]
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 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

Execution Time

- Bigger is better (more reuse)

Tile Size (kB)

Gemm

- 1.4
- 1
- 0.6

Cache Thrashing

- 71%

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

SW expresses program-level semantic information

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)

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

<table>
<thead>
<tr>
<th>Workload</th>
<th>Normalized Exec. Time</th>
<th>Tile Size in kB</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlation</td>
<td>0.7 - 1.2</td>
<td>0 - 200</td>
</tr>
<tr>
<td>trisolv</td>
<td>0.7 - 1.3</td>
<td>0 - 150</td>
</tr>
<tr>
<td>lu</td>
<td>0.7 - 1.3</td>
<td>0 - 1000</td>
</tr>
<tr>
<td>trmm</td>
<td>0.8 - 1.2</td>
<td>0 - 100</td>
</tr>
<tr>
<td>gramshmidt</td>
<td>0.7 - 1.1</td>
<td>0 - 150</td>
</tr>
<tr>
<td>floyd-warshall</td>
<td>0.5 - 1.5</td>
<td>0 - 50</td>
</tr>
<tr>
<td>gesummv</td>
<td>0.2 - 2.2</td>
<td>0 - 1000</td>
</tr>
<tr>
<td>mvt</td>
<td>0.5 - 1.5</td>
<td>0 - 62</td>
</tr>
<tr>
<td>jacobi-1D</td>
<td>0.3 - 1.1</td>
<td>0 - 32</td>
</tr>
<tr>
<td>jacobi-2D</td>
<td>0.7 - 1.7</td>
<td>0 - 65</td>
</tr>
</tbody>
</table>

Baseline vs 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

Higher-level Program Semantics

Expressive Memory “XMem”

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

Carnegie Mellon University

University of Toronto

SFU

NVIDIA

ETH Zürich