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[ϰ] Nastaran Hajinazar[┼] Phillip B. Gibbons[†] Onur Mutlu[§†]

[†]Carnegie Mellon University [‡]University of Toronto [ϰ]NVIDIA
[┼]Simon Fraser University [§]ETH Zürich

Abstract

This paper makes a case for a new cross-layer interface, Expressive Memory (XMem), to communicate higher-level program semantics from the application to the system software and hardware architecture. XMem provides (i) a flexible and extensible abstraction, called an Atom*, enabling the application to express key program semantics in terms of how the program accesses data and the attributes of the data itself, and (ii) new cross-layer interfaces to make the expressed higher-level information available to the underlying OS and architecture. By providing key information that is otherwise unavailable, XMem exposes a new, rich* view *of the program data to the OS and the different architectural components that optimize memory system performance (e.g., caches, memory controllers).*

By bridging the semantic gap *between the application and the underlying memory resources, XMem provides two key benefits. First, it enables architectural/system-level techniques to leverage key program semantics that are challenging to predict or infer. Second, it improves the efficacy and portability of software optimizations by alleviating the need to tune code for specific hardware resources (e.g., cache space). While XMem is designed to enhance and enable a wide range of memory optimizations, we demonstrate the benefits of XMem using two use cases: (i) improving the performance portability of software-based cache optimization by expressing the semantics of data locality in the optimization and (ii) improving the performance of OS-based page placement in DRAM by leveraging the semantics of data structures and their access properties.*

1. Introduction

Traditionally, the key interfaces between the software stack and the architecture (the ISA and virtual memory) have been primarily designed to convey program *functionality* to ensure the program is executed as required by software. An application is converted into ISA instructions and a series of accesses to virtual memory for execution in hardware. The application is, hence, stripped down to the basics of what is necessary to execute the program correctly, and the *higher-level semantics* of the program are lost. For example, even the simple higher-level notion of different *data structures* in a program is *not* available to the OS or hardware architecture, which deal only with virtual/physical pages and addresses. While the higher-level semantics of data structures may be irrelevant for correct execution, these semantics could prove very useful to the system for *performance optimization.*

There is, in other words, a *disconnect* or *semantic gap* between the levels of the computing stack when it comes to conveying higher-level program semantics from the application to the wide range of system-level and architectural components that aim to improve performance. While the implications of the disconnect are far-reaching, in this work, we narrow the focus to a critical component in determining the overall system efficiency, *the memory subsystem.* Modern systems employ a large variety of components to optimize memory performance (e.g., prefetchers, caches, memory controllers). The semantic gap has two important implications:

Implication 1. The OS and hardware memory subsystem components are forced to *predict* or *infer* program behavior when optimizing for performance. This is challenging because: *(i)* each component (e.g., L1 cache, memory controller) sees only a *localized* view of the data accesses made by the application and misses the bigger picture, *(ii) specialized* hardware may be required for each component optimizing for memory, and *(iii)* the optimizations are typically only *reactive* as the program behavior is *not* known a priori.

Implication 2. Software is forced to optimize code to the specifics of the underlying architecture (e.g., by tuning *tile size* to fit a specific cache size). Memory resource availability, however, can change or be unknown (e.g., in virtualized environments or in the presence of co-running applications). As a result, software optimizations are often unable to make accurate assumptions regarding memory resource availability, leading to significant challenges in *performance portability.*

The challenges of predicting program behavior and hence the benefits of knowledge from software in memory system optimization are well known [1–22]. There have been numerous hardware-software cooperative techniques proposed in the form of fine-grain hints implemented as new ISA instructions (to aid cache replacement, prefetching, etc.) [1–13], program annotations or directives to convey program semantics and programmer intent [3, 14–17], or hardware-software co-designs for specific optimizations [18–22]. These approaches, however, have two significant shortcomings. First, they are designed for a *specific memory optimization* and are limited in their implementation to address only challenges specific to that optimization. As a result, they require changes across the stack for a *single optimization* (e.g., cache replacement, prefetching, or data placement). Second, they are often very specific directives to instruct a particular component to behave in a certain manner (e.g., instructions to prefetch specific data or prioritize certain cache lines). These specific

directives create portability and programmability concerns because these optimizations may *not* apply across different architectures and they require significant effort to understand the hardware architecture to ensure the directives are useful.

Our Goal. In this work, we ask the question: *can we design a unifying general abstraction and a cohesive set of interfaces between the levels of the system stack to communicate key program semantics from the application to all the system-level and architectural components?* In response, we present Expressive Memory (XMem), a rich cross-layer interface that provides a new *view* of the program data to the entire system. Designing XMem in a *low-overhead, extensible, and general* manner requires addressing several non-trivial challenges involving conflicting tradeoffs between generality and overhead, programmability and effectiveness (§2.2). In this paper, we provide a first attempt at designing a new end-to-end system to achieve our goal while addressing these challenges.

Expressive Memory comprises two key components:

(1) The Atom. We introduce a new hardware-software abstraction, the *atom*, which is a region of virtual memory with a set of well-defined properties (§3.1). Each atom maps to data that is *semantically similar*, e.g., a data structure, a *tile* in an array, or any pool of data with similar properties. Programs explicitly specify atoms that are communicated to the OS and hardware. Atoms carry program information such as: *(i)* data properties (e.g., data type, sparsity, approximability), *(ii)* access properties (e.g., access pattern, read-write characteristics), and *(iii)* data locality (e.g., data reuse, working set). The atom can also track properties of data that *change* during program execution.

(2) System and Cross-layer Interfaces. Figure 1 presents an overview of this component: *(i)* The interface to the application enables software to explicitly express atoms via program annotation, static compiler analysis, or dynamic profiling ❶; *(ii)* The XMem system enables summarizing, conveying, and storing the expressed atoms ❷; *(iii)* The interface to the OS and architectural components (e.g., caches, prefetchers) provides key supplemental information to aid optimization ❸. This interface enables any system/architectural component to simply *query* the XMem system for the higher-level semantics attached to a memory address ❹.

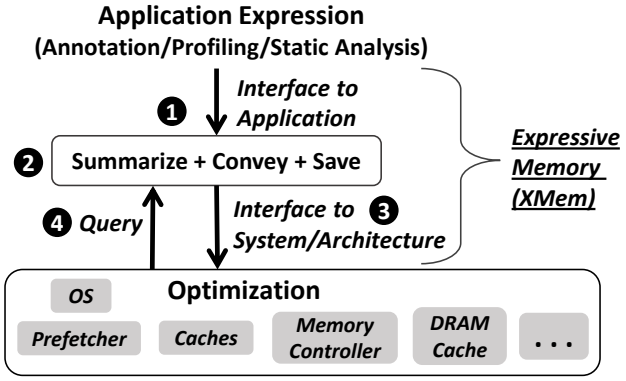

Figure 1: XMem: the system and interfaces.

Use Cases. XMem is designed to be a *general* interface to aid a wide range of memory optimizations. In this paper, we first demonstrate the benefits of XMem in enhancing the portability of *software-based cache optimizations* (§5). The effectiveness of such optimizations (e.g., *cache tiling* [60–66]) is highly susceptible to changes in cache space availability: If the available cache space at runtime is *less* than what the program was optimized for, *cache thrashing* often ensues. We demonstrate that by leveraging data locality semantics (working set size and data reuse), we can enhance and coordinate the cache management and prefetching policies to avoid cache thrashing and ensure high hit rates are retained, thereby improving the portability of the optimization. We demonstrate that when software optimizations inaccurately assume available cache space, XMem reduces the loss in performance from 55% in the baseline system to 6% on average. Second, we demonstrate the performance benefits of *intelligent OS-based page placement in DRAM* by leveraging knowledge of data structures and their access semantics (§6). XMem improves performance by *(i)* isolating regular data structures with high row buffer locality in separate banks and *(ii)* spreading out irregular data structures across many banks/channels to maximize parallelism. Our experimental evaluation demonstrates an 8.5% average performance improvement (up to 31.9%) over state-of-the-art techniques.

More generally, Table 1 presents nine example memory optimizations and the benefits XMem can provide over prior works that propose these optimizations in a specialized manner. XMem's benefits arise in three ways. First, it provides a *unifying*, central interface for a wide range of optimizations that use many of the *same* semantics. Second, it partitions data into pools of semantically-similar data. This enables using different policies (e.g., cache policies, compression algorithms) for different pools of data. Third, it enhances optimizations by providing higher-level semantics that *(i)* are unavailable locally to each component at runtime (e.g., distinguishing between data structures, data properties), *(ii)* are challenging to accurately infer (e.g., working set size, data reuse) or *(iii)* require profiling/monitoring to determine (e.g., read-only/read-write, private/shared data characteristics).

This paper makes the following **contributions**:

- This work is the first attempt to design a *holistic and general* cross-layer interface to communicate higher-level program semantics to the different system and architectural components in order to enable more effective memory optimizations in modern CPUs.
- To this end, we introduce XMem, which comprises a new software-hardware abstraction—the Atom—and a full end-to-end system design. XMem *(i)* is general and flexible enough to cover a wide range of program semantics and use cases, *(ii)* is completely decoupled from system functionality and affects only performance not correctness, *(iii)* can react to phase changes in data properties during execution, and *(iv)* has a low-overhead implementation.
- We quantitatively demonstrate the benefits of using XMem to *(i)* improve the portability of software-based cache optimizations by leveraging *data locality* semantics and *(ii)* enhance OS-based DRAM placement by leveraging semantics of data structures and their access properties. We highlight seven other use cases (Table 1).

Table 1: Summary of the example memory optimizations that XMem aids.

Memory optimization	Example semantics provided by XMem (described in §3.3)	Example Benefits of XMem
Cache management	*(i)* Distinguishing between data structures or pools of similar data; *(ii)* Working set size; *(iii)* Data reuse	Enables: *(i)* applying different caching policies to different data structures or pools of data; *(ii)* avoiding cache thrashing by *knowing* the active working set size; *(iii)* bypassing/prioritizing data that has no/high reuse. (§5)
Page placement in DRAM e.g., [23, 24]	*(i)* Distinguishing between data structures; *(ii)* Access pattern; *(iii)* Access intensity	Enables page placement at the *data structure* granularity to *(i)* isolate data structures that have high row buffer locality and *(ii)* spread out concurrently-accessed irregular data structures across banks and channels to improve parallelism. (§6)
Cache/memory compression e.g., [25–32]	*(i)* Data type: integer, float, char; *(ii)* Data properties: sparse, pointer, data index	Enables using a *different compression algorithm* for each data structure based on data type and data properties, e.g., sparse data encodings, FP-specific compression, delta-based compression for pointers [27].
Data prefetching e.g., [33–36]	*(i)* Access pattern: strided, irregular, irregular but repeated (e.g., graphs), access stride; *(ii)* Data type: index, pointer	Enables *(i) highly accurate* software-driven prefetching while leveraging the benefits of hardware prefetching (e.g., by being memory bandwidth-aware, avoiding cache thrashing); *(ii)* using different prefetcher *types* for different data structures: e.g., stride [33], tile-based [20], pattern-based [34–37], data-based for indices/pointers [38, 39], etc.
DRAM cache management e.g., [40–46]	*(i)* Access intensity; *(ii)* Data reuse; *(iii)* Working set size	*(i)* Helps avoid cache thrashing by knowing working set size [44]; *(ii)* Better DRAM cache management via reuse behavior and access intensity information.
Approximation in memory e.g., [47–53]	*(i)* Distinguishing between pools of similar data; *(ii)* Data properties: tolerance towards approximation	Enables *(i)* each memory component to track how approximable data is (at a fine granularity) to inform approximation techniques; *(ii)* data placement in heterogeneous reliability memories [54].
Data placement: NUMA systems e.g., [55, 56]	*(i)* Data partitioning across threads (i.e., relating data to threads that access it); *(ii)* Read-Write properties	Reduces the need for profiling or data migration *(i)* to co-locate data with threads that access it and *(ii)* to identify Read-Only data, thereby enabling techniques such as replication.
Data placement: hybrid memories e.g., [16, 57, 58]	*(i)* Read-Write properties (Read-Only/Read-Write); *(ii)* Access intensity; *(iii)* Data structure size; *(iv)* Access pattern	Avoids the need for profiling/migration of data in hybrid memories to *(i)* effectively manage the asymmetric read-write properties in NVM (e.g., placing Read-Only data in the NVM) [16, 57]; *(ii)* make tradeoffs between data structure "hotness" and size to allocate fast/high bandwidth memory [14]; and *(iii)* leverage row-buffer locality in placement based on access pattern [45].
Managing NUCA systems e.g., [15, 59]	*(i)* Distinguishing pools of similar data; *(ii)* Access intensity; *(iii)* Read-Write or Private-Shared properties	*(i)* Enables using different cache policies for different data pools (similar to [15]); *(ii)* Reduces the need for reactive mechanisms that detect sharing and read-write characteristics to inform cache policies.

2. Goals and Challenges

2.1. Key Requirements

There are several key requirements and invariants that drive the design of the proposed system:

***(i)* Supplemental and hint-based.** The new interface should *not* affect functionality or correctness of the program in any way—it provides only *supplemental* information to help improve *performance*. This reduces the necessity of obtaining precise or detailed hints, and system implementation can be *simpler* as information can be conveyed/stored imprecisely.

***(ii)* Architecture agnosticism.** The abstraction for expressing semantics must be based on the *application* characteristics rather than the specifics of the system, e.g., cache size, memory banks available. This means that the programmer/software need *not* be aware of the precise workings of the memory system resources, and it significantly alleviates the portability challenges when the programmer/software optimizes for performance.

***(iii)* Generality and extensibility.** The interface should be general enough to flexibly express a wide range of program semantics that could be used to aid many system-level and architectural (memory) optimizations, and extensible to support more semantics and optimizations.

***(iv)* Low overhead.** The interface must be amenable to an implementation with low storage area and performance overheads, while preserving the semantics of existing interfaces.

2.2. Challenges

Current system and architectural components see only a description of the program's data in terms of *virtual/physical addresses*. To provide higher-level program-related semantics, we need to associate each address with much more information than is available to the entire system today, addressing the following three challenges:

Challenge 1: Granularity of expression. The granularity of associating program semantics with program data is challenging because the best granularity for *expressing* program semantics is program dependent. Semantics could be available at the granularity of an entire data structure, or at much smaller granularities, such as a *tile* in an array. We cannot simply map program semantics to *every* individual virtual address as that would incur too much overhead, and the fixed granularity of a *virtual page* may be too coarse-grained, inflexible and challenging for programmers to reason about.

Challenge 2: Generality vs. specialization. Our architecture-agnosticism requirement implies that we express higher-level information from the application's or the programmer's point of view—without any knowledge/assumptions of the memory resources or specific directives to a hard-

ware component. As a consequence, much of the conveyed information may be either irrelevant, too costly to manage effectively, or *too complex* for different hardware components to easily use. For example, hardware components like prefetchers are operated by simple hardware structures and need only know prefetchable access patterns. Hence, the abstraction must be *(i) high-level and architecture-agnostic*, so it can be easily expressed by the programmer and *(ii) general*, in order to express a range of information useful to many components. At the same time, it should be still amenable to translation into simple directives for each component.

Challenge 3: Changing data semantics. As the program executes, the semantics of the data structures and the way they are accessed can change. Hence, we need to be able to express *dynamic* data attributes in *static* code, and these changing attributes need to be conveyed to the running system at the appropriate time. This ensures that the data attributes seen by the memory components are accurate any time during execution. Continual updates to data attributes at runtime can impose significant overhead that must be properly managed to make the approach practical.

3. Our Approach: Expressive Memory

We design Expressive Memory (XMem), a new rich cross-layer interface that enables *explicit* expression and availability of key program semantics. XMem comprises two key components: *(i)* a new hardware-software *abstraction* with a well-defined set of properties to convey program semantics (§3.1-§3.3) and *(ii)* a rich set of *interfaces* to convey and store that information at different system/memory components (§3.4-§3.5). We describe how our key design choices for both components address the above challenges in §3.2 and §3.4.

3.1. The Atom Abstraction

We define a new hardware-software abstraction, called an *Atom*, that serves as the *basic unit* of expressing and conveying program semantics to the system and architecture. An atom forms both an *abstraction* for information expressed as well as a *handle* for communicating, storing, and retrieving the conveyed information across different levels of the stack. Application programs can dynamically create atoms in the program code, each of which describes a specific range of program data at any given time during execution. The OS and hardware architecture can then interpret atoms specified in the program when the program is executed.

There are three key components to an atom: *(i) Attributes:* higher-level data semantics that it conveys; *(ii) Mapping:* the virtual address range that it describes; and *(iii) State:* whether the atom is currently active or inactive.

3.2. Semantics and Invariants of an Atom

We define the invariants of the atom abstraction and then describe the operators that realize the semantics of the atom.

- **Homogeneity:** All the data that maps to the same atom has the *same* set of attributes.
- **Many-to-One VA-Atom Mapping:** At any given time, any virtual address (VA) can map to *at most* one atom. Hence, the system/architecture can query for the atom (if any) associated with a VA and thereby obtain any attributes associated with the VA, at any given time.
- **Immutable Attributes:** While atoms are dynamically created, the attributes of an atom *cannot* be changed once created. To express different attributes for the same data, a new atom should be created. Atom attributes can, hence, be specified *statically* in the program code. Because any atom's attributes *cannot* be updated during execution, these attributes can be summarized and conveyed to the system/architecture at compile/load time *before execution*. This minimizes expensive communication at runtime (Challenge 3).
- **Flexible mapping to data:** Any atom can be flexibly and dynamically mapped/unmapped to any set of data of any size. By selectively mapping and/or unmapping data to the same atom, an atom can be mapped to non-contiguous data (Challenge 1).
- **Activation/Deactivation:** While the attributes of an atom are immutable and statically specified, an atom itself can be *dynamically* activated and deactivated in the program. The attributes of an atom are recognized by the system *only* when the atom is currently active. This enables updating the attributes of any data region as the program executes: when the atom no longer accurately describes the data, it is deactivated and a new atom is mapped to the data. This ensures that the system always sees the correct data attributes during runtime, even though the attributes themselves are communicated earlier at load time (Challenge 3).

To manipulate the three components (*Attributes*, *Mapping*, and *State*), there are three corresponding operations that can be performed on an atom via a corresponding library call (§4.1.1): *(i)* **`CREATE`** an atom, providing it with immutable statically-specified attributes; *(ii)* **`MAP/UNMAP`** an atom to/from a range of data; and *(iii)* **`ACTIVATE/DEACTIVATE`** an atom to dynamically tell the memory system that the attributes of the atom are now (in)valid for the data the atom is mapped to.

Figure 2 depicts an overview of the atom operators. Atoms are first created in the program with statically-specified attributes ❶. During memory allocation (e.g., malloc), data structures are allocated ranges of virtual memory. After allocation, atoms with the appropriate attributes can be flexibly *mapped* to the corresponding VA range that each atom describes ❷. Once the mapped atom is *activated*, all the system components recognize the attributes as valid ❸. Data can be easily remapped to a different atom that describes it better as the program moves into a different phase of execution (using the `MAP` operator ❷), or just unmapped ② when the atom no longer accurately describes the data. The `MAP/UNMAP` operator can be flexibly called to selectively map/unmap multiple

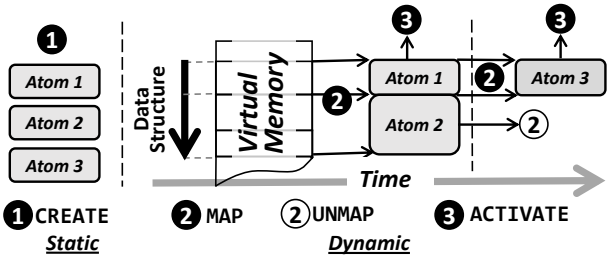

Figure 2: Overview of the three atom operators.

data ranges to/from the same atom at any granularity. The ACTIVATE/DEACTIVATE operator dynamically validates/invalidates the atom attributes relating to *all* data the atom is mapped to. §3.5.2 describes XMem's low-overhead implementation of these operators.

3.3. Attributes of an Atom and Use Cases

Each atom contains an extensible set of attributes that convey the key program semantics to the rest of the system. Table 1 lists example use cases for these attributes. The three classes of attributes (to date) are:

(1) Data Value Properties: An expression of the attributes of the data values contained in the data pool mapped to an atom. It is implemented as an extensible list using a single bit for each attribute. These attributes include data type (e.g., INT32, FLOAT32, CHAR8) and data properties (e.g., SPARSE, APPROXIMABLE, POINTER, INDEX).

(2) Access Properties: This describes three key characteristics of the data the atom is mapped to:

- **AccessPattern**: This attribute defines the PatternType, currently either REGULAR (with a specific stride that is also specified), IRREGULAR (when the access pattern is *repeatable* within the data range, but with no repeated stride, e.g., graphs), or NON_DET (when there is no repeated pattern).
- **RWChar**: This attribute describes the read-write characteristics of data at any given time, currently either READ_ONLY, READ_WRITE, or WRITE_ONLY. It could also be extended to include varying degrees of read-write intensity, and include shared/private information.
- **AccessIntensity**: This attribute conveys the access frequency or "hotness" of the data relative to other data at any given time. This attribute can be provided by the programmer, compiler, or profiler. It is represented using an 8-bit integer, with 0 representing the lowest frequency. Higher values imply an increasing amount of intensity *relative* to other data. Hence, this attribute conveys an access intensity ranking between different data mapped to different atoms.

(3) Data Locality: This attribute serves to express software optimizations for cache locality explicitly (e.g., cache tiling, stream buffers, partitions, etc.). The key attributes include *working set size* (which is inferred from the size of data the atom is mapped to) and *reuse*, for which we use a simple 8-bit integer (0 implying no reuse and higher values implying a higher amount of reuse *relative* to other data).

Note that the atom abstraction and its interface do *not* a priori limit the program attributes that an atom can express. This makes the interface flexible and forward-compatible in terms of extending and changing the expressed program semantics. The above attributes have been selected for their memory optimization benefits (Table 1) and ready translation into simple directives for the OS and hardware components.

3.4. The XMem System: Key Design Choices

Before we describe XMem's system implementation, we explain the rationale behind the key design choices.

- **Leveraging Hardware Support:** For the XMem design, we leverage hardware support for two major reasons. First, a key design goal for XMem is to minimize the runtime *overhead* of tracking and retrieving semantics at a fine granularity (even semantics that change as the program executes). We hence leverage support in *hardware* to efficiently perform several key functionalities of the XMem system—mapping/unmapping of semantics to atoms and activating/deactivating atoms at runtime. This avoids the high overhead of frequent system calls, memory updates, etc. Second, we aim to enable the many hardware-managed components in the memory hierarchy to leverage XMem. To this end, we use hardware support to efficiently transmit key semantics to the different hardware components.
- **Software Summarization and Hardware Tracking:** Because the potential atoms and their attributes are known statically (by examining the application program's CREATE calls), the compiler can summarize them at compile time, and the OS can load them into kernel memory at load time. The program directly communicates an atom's active status and address mapping(s) at runtime (via MAP and ACTIVATE calls) with the help of new instructions in the ISA (§4) and hardware support. This minimizes expensive software intervention/overhead at runtime. In other words, the static CREATE operator is handled in software before program execution and the dynamic MAP and ACTIVATE operators are handled by hardware at runtime.
- **Centralized Global Tracking and Management:** All the statically-defined atoms in the program are assigned a global *Atom ID* (within a process) that the *entire* system recognizes. Tracking which atoms are active at runtime and the inverse mapping between a VA and atom ID is also centralized at a *global* hardware table, to minimize storage and communication cost (i.e., all architectural components access the same table to identify the active atom for a VA).
- **Private Attributes and Attribute Translation:** The atom attributes provided by the application may be too complex and excessive for easy interpretation by components like the cache or prefetcher. To address this challenge (Challenge 2), when the program is loaded for execution or after a context switch, the OS invokes a *hardware translator* that converts the higher-level attributes to sets of specific primitives relevant to each hardware component and the optimization the component performs. Such specific primitives are then saved *privately* at each component, e.g., the prefetcher saves only the access pattern for each atom.

3.5. XMem: Interfaces and Mechanisms

Figure 3 depicts an overview of the system components to implement the semantics of XMem.

3.5.1. Programmer/Application Interface. The application interface to XMem is via a library, XMemLib (❶). An atom is defined by a class data structure (§4.1.1) that defines the attributes of the atom and the operator functions (CREATE, MAP/UNMAP, and ACTIVATE/DEACTIVATE). An atom and its static attributes can be instantiated in the program code (CREATE) by the programmer, autotuner, or compiler.

3.5.2. System/Architecture Interface. XMemLib communicates with the OS and architecture in the following two ways.

First, at compile time, the compiler summarizes all the atoms in the program statically and creates a table for atom

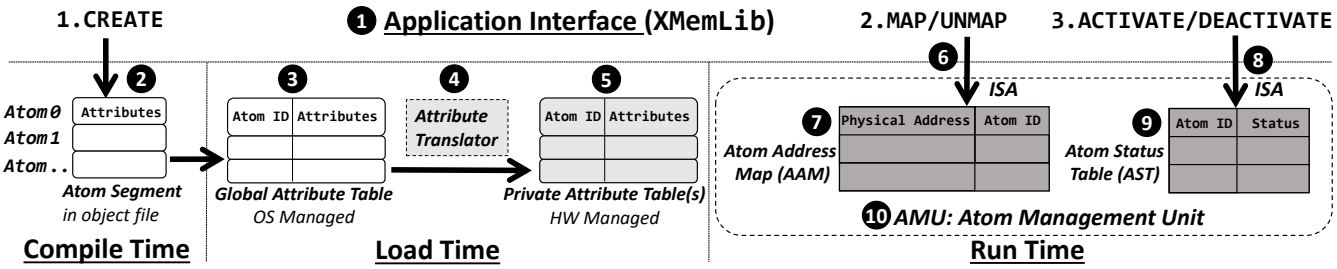

Figure 3: XMem: Overview of the components.

attributes, indexed by atom ID. During run time, the same static atom can have many instances (e.g., within a `for` loop or across multiple function calls). All of the calls to create the same atom will, however, be mapped to the same static atom (and Atom ID). This is possible because atom attributes are immutable. However, the address mapping of each atom is typically not known at compile time because virtual address ranges are only resolved at runtime. The compiler creates a table of all the atoms in the program along with the atom attributes. This table is placed in the *atom segment* of the program object file (❷). When the program is loaded into memory for execution by the OS, the OS also reads the atom segment and saves the attributes for each atom in the GLOBAL ATTRIBUTE TABLE (GAT ❸), which is managed by the OS in kernel space. The OS also invokes a *hardware translator* (❹) that converts the higher-level attributes saved in the GAT to sets of specific hardware primitives relevant to each hardware component, and saves them in a per-component PRIVATE ATTRIBUTE TABLE (PAT ❺), managed in hardware by each component.

Second, at run time, XMem operators, in the form of function calls in XMemLib, are translated into *ISA instructions* that inform the system and architecture of the atoms' activation/deactivation and mapping. Conceptually, the MAP/UNMAP operator (❻) is converted into ISA instructions that update the ATOM ADDRESS MAP (AAM ❼), which enables looking up the atom ID associated with a physical address (PA). We use the PA to index the AAM instead of the VA to simplify the table design (§4.2). The ACTIVATE/DEACTIVATE operator (❽) is converted into ISA instructions that update an atom's active status in the ATOM STATUS TABLE (AST ❾). The AST and AAM are managed by the Atom Management Unit (AMU ❿). Because tables with entries for each PA are infeasible, we use simple mechanisms to avoid them. These mechanisms, and the functioning and implementation of these hardware and software components, are described in §4.2.

Flexibility and Extensibility. The system/architecture interface ensures that the ISA and the microarchitecture need only implement the three operators, but does *not* dictate what application attributes can be conveyed. The attributes are stored in the binary as a separate metadata segment with a version identifier to identify the information format. The information format can be enhanced across architecture generations, ensuring flexibility and extensibility, while the version identifier ensures forward/backward compatibility. Any future architecture can interpret the semantics and older XMem architectures can simply ignore unknown formats.

4. XMem: Detailed Design

We now detail the design and implementation of the interfaces and components in XMem. We describe the application, OS, and architecture interfaces (§4.1), the key components of XMem (§4.2), the use of XMem in virtualized environments (§4.3), and the overheads of our design (§4.4).

4.1. The Interfaces

4.1.1. Application Interface. The primary interface between XMem and the application is XMemLib, a library that provides type definitions and function calls for atoms. XMemLib includes an atom class definition with the attributes described in §3.3. XMemLib provides three types of XMem operations on atoms in the form of function calls. These operations are the interface to manipulate the attributes, mappings and state of an atom. Table 2 summarizes the definition of all the functions (also discussed below):

(1) CREATE: The function CreateAtom creates an atom with the attributes specified by the input parameters, and returns an *Atom ID*. Multiple invocations of CreateAtom at the same place in the program code always return the same Atom ID (without reinvoking the function).

(2) MAP/UNMAP: These functions take an Atom ID and an address range as parameters, and invoke corresponding ISA instructions to tell the Atom Management Unit (AMU) to update the Atom Address Map (§4.2). We create multiple functions so that we can easily map or unmap multi-dimensional data structures (e.g., 2D/3D arrays). For example, Atom2DMap maps/unmaps a 2D block of data of width sizeX and height sizeY, in a 2D data structure that has a row length lenX.

(3) ACTIVATE/DEACTIVATE: The functions AtomActivate and AtomDeactivate serve to (de)activate the specified atom at any given time. They invoke corresponding ISA instructions that update the Atom Status Table (§4.2) at run time.

4.1.2. Operating System Interface. XMem interfaces with the OS in two ways. First, the OS manages the Global Attribute Table (GAT) (§4.2), which holds the attributes of all the atoms in each application. Second, the OS can optionally query for the *static* mapping between VA ranges and atoms through an interface to the memory allocator. This interface ensures that the OS knows the mapping *before* the virtual pages are mapped to physical pages, so that the OS can perform static optimizations, such as memory placement based on program semantics. Specifically, we augment the memory allocation APIs (e.g., malloc) to take Atom ID as a parameter. The memory allocator, in turn, passes the Atom ID to the OS via augmented system calls that request virtual

Table 2: The XMem operators and corresponding XMemLib functions and ISA instructions (sizes/lengths in bytes).

XMem Op	XMemLib Functions (Application Interface)	XMem ISA Insts (Architecture Interface)
CREATE	`AtomID CreateAtom(data_prop, access_pattern, reuse, rw_characteristics)`	No ISA instruction required

MAP/UNMAP	`AtomMap(atom_id, start_addr, size, map_or_unmap)`
	`Atom2DMap(atom_id, start_addr, lenX, sizeX, sizeY, map_or_unmap)`
	`Atom3DMap(atom_id, start_addr, lenX, lenY, sizeX, sizeY, sizeZ, map_or_unmap)`
		`ATOM_MAP AtomID, Dimensionality`
		`ATOM_UNMAP AtomID, Dimensionality`
		Address ranges specified in AMU-specific registers

ACTIVATE/ DEACTIVATE	`AtomActivate(atom_id)` `AtomDeactivate(atom_id)`	`ATOM_ACTIVATE AtomID` `ATOM_DEACTIVATE AtomID`

pages. The memory allocator maintains the static mapping between atoms and virtual pages by returning virtual pages that match the requesting Atom ID. The compiler converts the pair `A=malloc(size);` `AtomMap(atomID,A,size);` into this augmented API: `A=malloc(size,atomID);` `AtomMap(atomID,A,size);`. This interface enables the OS to manipulate the virtual-to-physical address mapping *without* extra system call overheads.

4.1.3. Architecture Interface. We add two new ISA instructions to enable XMem to talk to the hardware at run time: *(i)* ATOM_MAP/ATOM_UNMAP tells the Atom Management Unit (AMU) to update the address ranges of an atom. When this instruction is executed, the parameters required to convey the address mapping for the different mapping types (Table 2) are implicitly saved in AMU-specific registers and accessed by the AMU. To map or unmap the address range to/from the specified atom, the AMU asks the Memory Management Unit (MMU) to translate the virtual address ranges specified by ATOM_MAP to physical address ranges, and updates the Atom Address Map (AAM) (§4.2). *(ii)* ATOM_ACTIVATE/ATOM_DEACTIVATE causes the AMU to update the Atom Status Table (AST) to activate/deactivate the specified atom.

4.2. System Design: Key Components

The system/architecture retrieves the data semantics associated with each memory address in three steps: *(i)* determine to which atom (if any) a given address maps; *(ii)* determine whether the atom is *active*; and *(iii)* retrieve the atom attributes. XMem enables this with four key components:

(1) Atom Address Map (AAM): This component determines the latest atom (if any) associated with any PA. Because the storage overhead of maintaining a mapping table between *each address* and Atom ID would be prohibitively large, we employ an *approximate mapping* between atoms and address ranges at a configurable granularity. The system decides the smallest *address range unit* the AAM stores for each address-range-to-atom mapping. The default granularity is 8 cache lines (512B), which means each consecutive 512B can map only to one atom. This design significantly reduces the storage overhead as we need only store one Atom ID for each 512B (0.2% storage overhead assuming an 8-bit Atom ID). We can reduce this overhead further by increasing the granularity or limiting the number of atoms in each application. For instance, if we support only 6-bit Atom IDs with a 1KB address range unit, the storage overhead becomes 0.07%. Note that because XMem provides only *hints* to the system, our approximate mapping may cause optimization inaccuracy but it has no impact on functionality and correctness.

To make it easy to look up the Atom ID for each address, the AAM stores the Atom IDs *consecutively* for *all* the physical pages. The index of the table is the physical page index and each entry stores all Atoms IDs in each page. In the default configuration, each of the Atom IDs require 8B of storage per page (8 bits times 8 subpages). With this design, the OS or the hardware architecture can simply use the physical address that is queried as the table index to find the Atom ID.

We use the PA instead of the VA to index this table because *(i)* there are far fewer PAs compared to VAs and *(ii)* this enables the simplified lookup scheme discussed above.

(2) Atom Status Table (AST): We use a bitmap to store the status (active or inactive) of all atoms in each application. Because CreateAtom assigns atom IDs consecutively starting at 0, this table is efficiently accessed using the atom ID as index. Assuming up to 256 atoms per application (all benchmarks in our experiments had under 10 atoms, all in performance-critical sections), the AST is only 32B per application. The Atom Management Unit (AMU) updates the bitmap when an ATOM_(DE)ACTIVATE instruction is executed.

(3) Attribute Tables (GAT and PAT) and the Attribute Translator: As discussed in §3.4, we store the attributes of atoms in a Global Attribute Table (GAT) and multiple Private Attribute Tables (PAT). GAT is managed by the OS in kernel space. Each hardware component that benefits from XMem maintains its own PAT, which stores a *translated* version of the attributes (an example of this is in §5). This translation is done by the *Attribute Translator*, a hardware runtime system that translates attributes for each component at program load time and during a context switch.

(4) Atom Management Unit (AMU): This is a hardware unit that is responsible for *(i)* managing the AAM and AST and *(ii)* looking up the Atom ID given a physical address. When the CPU executes an XMem ISA instruction, the CPU sends the associated command to the AMU to update the AAM (for ATOM_MAP or ATOM_UNMAP) or the AST (for ATOM_ACTIVATE). For higher-dimensional data mappings, the AMU converts the mapping to a linear mapping at the AAM granularity and broadcasts this mapping to all the hardware components that require accurate information of higher-dimensional address mappings (see §5 for an example).

A hardware component determines the Atom ID of a specific physical address (PA) by sending an ATOM_LOOKUP request to the AMU, which uses the PA as the index into the AAM. To avoid memory accesses for all the ATOM_LOOKUP requests, each AMU has an atom lookaside buffer (ALB), which caches the results of recent ATOM_LOOKUP requests. The functionality of an ALB is similar to a TLB in an MMU, so the AMU accesses

the AAM *only* on ALB misses. The tags for the ALB are the physical page indexes, while the data are the Atom IDs in the physical pages. In our evaluation, we find that a 256-entry ALB can cover 98.9% of the ATOM_LOOKUP requests.

4.3. XMem in Virtualized Environments

Virtualized environments employ virtual machines (VMs) or containers that execute applications over layers of operating systems and hypervisors. The existence of multiple address spaces that are seen by the guest and host operating systems, along with more levels of abstraction between the application and the underlying hardware resources, makes the design of hardware-software mechanisms challenging. XMem is, however, designed to seamlessly function in these virtualized environments, as we describe next.

XMem Components. The primary components of XMem include the AAM, AST, the PATs, and the GAT. Each of these components function with no changes in virtualized environments: *(i)* AAM: The hardware-managed AAM, which maps physical addresses to atom IDs, is indexed by the *host* physical address. As a result, this table is *globally shared* across all processes running on the system irrespective of the presence of multiple levels of virtualization. *(ii)* AST and PATs: All atoms are tracked at the *process level* (irrespective of whether the processes belong to the same or different VMs). The per-process hardware-managed tables (AST and PATs) are reloaded during a context switch to contain the state and attributes of the atoms that belong to the currently-executing process. Hence, the functioning of these tables remains the same in the presence of VMs or containers. *(iii)* GAT: The GAT is software-managed and is maintained by each *guest* OS. During context switches, a register is loaded with a host physical address that points to the new process' GAT and AST.

XMem Interfaces. The three major interfaces (CREATE, MAP/UNMAP, and ACTIVATE/DEACTIVATE) require no changes for operation in virtualized environments. The CREATE operator is handled in software at compile time by the guest OS and all created atoms are loaded into the GAT by the guest OS at program load time. The MAP/UNMAP operator communicates directly with the MMU to map the host physical address to the corresponding atom ID using the XMem ISA instructions. The ACTIVATE/DEACTIVATE operator simply updates the AST, which contains the executing process' state.

Optimizations. OS-based software optimizations (e.g., DRAM placement in §6) require that the OS have visibility into the available physical resources. The physical resources may however be abstracted away from the guest OS in the presence of virtualization. In this case, the resource allocation and its optimization needs to be handled by the hypervisor or host OS for all the VMs that it is hosting. To enable the hypervisor/host OS to make resource allocation decisions, the guest OS also communicates the attributes of the application's atoms to the hypervisor. For hardware optimizations (e.g., caching policies, data compression), the hardware components (e.g., caches, prefetchers) retrieve the atom attributes for each process using the AAM and PATs. This is the same mechanism irrespective of the presence of virtualization. These components use application/VM IDs to distinguish between addresses from different applications/VMs (similar to prior work [67] or modern commercial virtualization schemes [68, 69]).

4.4. Overhead Analysis

The overheads of XMem fall into four categories: memory storage overhead, instruction overhead, hardware area overhead, and context switch overhead, all of which are small (the technical report has a more detailed discussion [70]):

(1) Memory storage overhead. The storage overhead comes from the tables that maintain the attributes, status, and mappings of atoms (AAM, AST, GAT, and PAT). As §4.2 discusses, the AST is very small (32B). The GAT and PAT are also small as the attributes of each atom need 19B, so each GAT needs only 2.8KB assuming 256 atoms per application. AAM is the largest table in XMem, but it is still insignificant as it takes only 0.2% of the physical memory (e.g., 16MB on a 8GB system), and it can be made even smaller by increasing the granularity of the address range unit (§4.2).

(2) Instruction overhead. There are instruction overheads when applications invoke the XMemLib functions to create, map/unmap, activate/deactivate atoms, which execute XMem instructions. We find this overhead negligible because: *(i)* XMem does *not* use extra system calls to communicate with the OS, so these operations are very lightweight; *(ii)* the program semantics or data mapping do *not* change very frequently. Among the workloads we evaluate, an additional 0.014% instructions on average (at most, 0.2%) are executed.

(3) Hardware area overhead. XMem introduces two major hardware components, Attribute Translator and AMU. We evaluate the storage overhead of these two components (including the AMU-specific registers) using CACTI 6.5 [71] at 14 *nm* process technology, and find that their area is 0.144 mm^2, or 0.03% of a modern Xeon E5-2698 CPU.

(4) Context switch overhead. XMem introduces one extra register for context switches—it stores the pointer to AST and GAT (stored consecutively for each application) in the AMU. AAM does not need a context-based register because it is a global table. The OS does not save the AMU-specific registers for MAP/UNMAP (Table 2) as the information is saved in the AAM. One more register adds very small overhead (two instructions, $\leq$ 1 ns) to the OS context switch (typically 3-5 μs). Context switches also require flushing the ALBs and PATs. Because these structures are small, the overhead is also commensurately small (~700 ns).

5. Use Case 1: Cache Management

5.1. Overview

Cache management is a well-known complex optimization problem with substantial prior work in both software (e.g., code/compiler optimizations, auto tuners, reuse hints) and hardware (advanced replacement/insertion/partitioning policies). XMem seeks to *supplement* both software and hardware approaches by providing key program semantics that are challenging to infer at runtime. We discuss XMem's benefits for cache management in Table 1. As a concrete end-to-end example, we describe and evaluate how XMem enhances *dy-*

namic policies to improve the portability and effectiveness of *static software-based* cache optimizations under *varying* cache space availability (as a result of co-running applications or unknown cache size in virtualized environments).

Many software techniques *statically* tune code by sizing the active working set in the application to maximize cache locality and reuse—e.g., hash-join partitioning [72] in databases, cache tiling [60–66] in linear algebra and stencils, cache-conscious data layout [73] for similarity search [74]. Static code optimization, however, makes assumptions regarding cache space availability when tuning code and hence imposes significant challenges in portability when the cache space availability is unknown or dynamically changing. Dynamic approaches to address this problem [75–78] have three major shortcomings. They *(i)* require using sophisticated software frameworks that use runtime systems to change code during execution, *(ii)* are highly specialized towards certain application classes, and *(ii)* are limited to codes that can be easily altered at runtime without losing correctness.

XMem improves the portability and effectiveness of static optimizations when resource availability is unknown by conveying the *optimization intent* to hardware—i.e., XMem conveys which high-reuse working set (e.g., tile) should be kept in the cache. It does *not* dictate exactly what caching policy to use to do this. The *hardware cache* leverages the conveyed information to keep the high-reuse working set of each application in the cache by prioritizing such data over other low-reuse data. In cases where the active working set does *not* fit in the available cache space, the cache *mitigates thrashing* by *pinning* part of the working set and then *prefetches* the rest based on the expressed access pattern.

5.2. Mechanism

(1) Expressing key working sets. As cache optimizations typically partition key data structures (e.g., the hash table in hash-join, tiles in arrays) to fit the caches, they simply need to use `XMemLib` to *map* the active high-reuse partitions (e.g., tiles) of key data structures to an atom that specifies a high reuse value and the access pattern. When the program is done with one partition, it *unmaps* the current partition and *maps* the next partition to the same atom.

(2) Optimization algorithm. Based on each active atom's *size* and *reuse* value, the cache decides the *insertion* and *prefetching* policy for each atom. We use a simple and practical greedy algorithm that the cache control logic runs, to decide which data to pin in the cache and what data to prefetch. The algorithm takes the active atoms in all the cores (each time there is a change in active atoms), and sorts the atoms based on the reuse values. Starting from the atom with the highest reuse, the cache decides if it has enough space to keep the data specified by each atom. When the total data size kept in the cache reaches the *pinning size limit* (we use 75% of the cache size so the cache still has space to handle other data), the algorithm stops and returns the list of atoms to be pinned. The cache inserts these atoms with the highest priority, and uses the default low-priority insertion policy for all other data. Any cache miss for a pinned atom (e.g., due to interference by a co-running application) triggers *prefetching* for the atom according to the expressed access pattern.

(3) Support in cache controllers. To keep the pinned atoms in the cache as long as possible, the cache controller inserts these atoms with the highest priority, and inserts all other data with the default insertion policy. When the highest-priority cache lines fill *75% of the ways*, all cache lines are inserted with the default insertion policy. When the list of active atoms changes (e.g., as a result of an `UNMAP`), only then does the cache *age* (reduce priority of) the high-priority lines so they can be evicted by the default replacement policy.

(4) Support in prefetchers. The prefetcher uses a PAT (§4.2) to keep the access pattern (stride) and address ranges for all pinned atoms. When an access to one of these atoms misses the cache, it prefetches the next cache line(s) based on the access pattern.

5.3. Evaluation Methodology

We model and evaluate XMem using zsim [79] with a DRAMSim2 [80] DRAM model. Table 3 summarizes the main simulation parameters. We use the Polybench suite [81], a collection of linear algebra, stencil, and data mining kernels. We use PLUTO [64], a polyhedral locality optimizer that uses *cache tiling* to statically optimize the kernels. We evaluate kernels that can be tiled within three dimensions and a wide range of tile sizes (from 64B to 8MB), ensuring the total work is always the same. We run the kernels to completion.

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]

5.4. Evaluation Results

Overall performance. To understand the cache tiling challenge, in Figure 4, we plot the execution time of 12 kernels, which are statically compiled with different tile sizes. For each workload, we show the results of two systems: *(i)* `Baseline`, the baseline system with a high-performance cache replacement policy (DRRIP [83]) and a multi-stride prefetcher [33] at L3; and *(ii)* `XMem`, the system with the aforementioned cache management and prefetching mechanisms.

For the `Baseline` system, execution time varies significantly with tile size, making tile size selection a challenging task. Small tiles significantly reduce the reuse in the application and can be on average 28.7% (up to 2× ❶) *slower* than the *best* tile size. Many optimizations, hence, typically size the tile to be as big as what can fit in the available cache space [65,78]. However, when an optimization makes incorrect assumptions regarding available cache space (e.g., in virtualized environments or due to co-running applications), the tile size may *exceed* the available cache space. We find that this can lead to *cache thrashing* and severe slowdown (64.8% on average, up to 7.6× ❷), compared to the performance with an optimized tile size. `XMem`, however, significantly reduces this slowdown

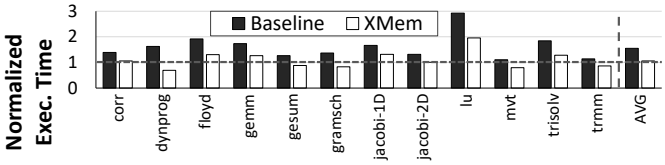

Figure 5: Maximum execution time with different cache sizes when code is optimized for a 2MB cache.

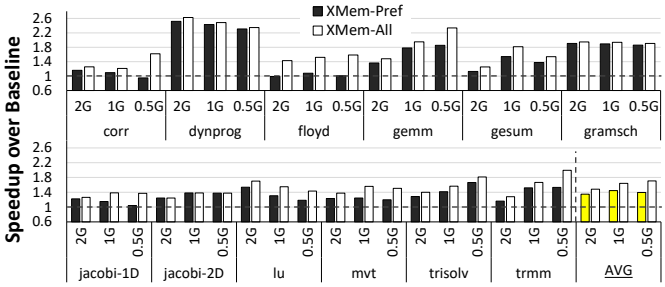

Figure 6: XMem's speedup over Baseline with different memory bandwidth availability.

from cache thrashing in the largest tile sizes to 26.9% on average (up to 4.6× ❸). XMem's large improvement comes from accurate *pinning* (that retains part of the high-reuse working set in the cache) and more accurate prefetching (that fetches the remaining working set).

Performance portability. To evaluate portability benefits from the reduced impact of cache thrashing in large tile sizes, we run the following experiment. For each workload, we pick a tile size optimized for a 2MB cache, and evaluate *the same program binary* on a 2MB cache and 2 smaller caches (1MB and 512KB). Figure 5 depicts the *maximum* execution time among these three cache sizes for both Baseline and XMem, normalized to Baseline with a 2MB cache. When executing with less cache space, we find that XMem increases the execution time by only 6%, compared to the Baseline's 55%. Hence, we conclude that by leveraging the program semantics, XMem greatly enhances the performance portability of applications by reducing the impact of having less cache space than what the program is optimized for.

Effect of prefetching and cache management. We evaluate two design points of XMem to see the effect of different components: *(i)* XMem-Pref, the system that employs XMem-based prefetching (using DRRIP for cache management) and achieves similar performance to software prefetching [86]; *(ii)* XMem, the system that employs XMem for both cache management and prefetching. Figure 6 shows the speedup of these two design points over the *respective* Baseline for the largest tile sizes across three memory bandwidth configurations (2 GB/s, 1 GB/s, 0.5 GB/s per core). We see that *both* cache management and prefetching contribute to performance improvement. On average, XMem performs better than XMem-Pref by 13%, 19.5%, 31% with different memory bandwidth availability (2 GB/s, 1 GB/s and 0.5 GB/s, respectively). When the available memory bandwidth decreases, the gap between these two designs increases because reducing memory traffic is vital when memory bandwidth is scarce. XMem's program-semantics-aware data pinning reduces memory traffic by keeping high-reuse data in the caches. We conclude that coordinating prefetching and cache management, by leveraging knowledge of program semantics, enables efficient use of memory resources to improve overall system performance.

6. Use Case 2: Data Placement in DRAM

We describe a software-only use case that can be implemented *without* any of XMem's hardware components. XMem enables more intelligent data placement in DRAM by leveraging knowledge of: *(i)* the key *data structures* in the program and their access semantics, obtained from the application and *(ii)* the underlying resources (e.g., number of banks, channels), obtained from the system.

6.1. Overview

Off-chip main memory (DRAM) latency is a major performance bottleneck in modern CPUs [87–91]. The performance impact of this latency is in large part determined by two factors: *(i) Row Buffer Locality (RBL)* [92, 93]: how often requests access the *same* DRAM row in a bank *consecutively*. Consecutive accesses to the same open row saves the long latency required to *close* the already-open row and *open* the requested row. *(ii) Memory Level Parallelism (MLP)* [91, 94]: the number of concurrent accesses to different memory banks or channels. Serving requests in parallel *overlaps* the latency of the different requests. A key factor that determines the DRAM access pattern—and thus RBL and MLP—is how the program's data is mapped to the DRAM channels, banks, and rows. This data placement is controlled by *(i)* the OS (via the virtual-to-physical address mapping) and *(ii)* the memory controller (via the mapping of physical addresses to DRAM channels/banks/rows).

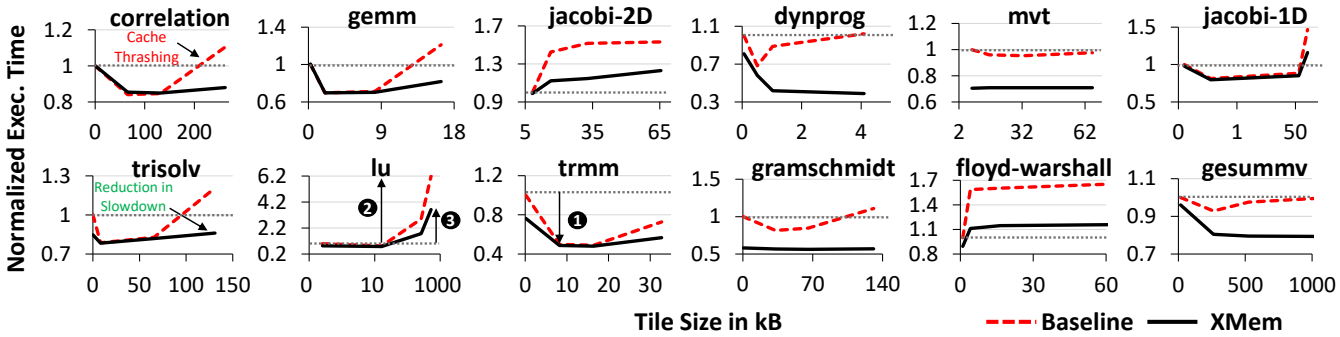

Figure 4: Execution time across different tile sizes (normalized to Baseline with the smallest tile size).

To improve RBL and MLP, prior works use both the OS (e.g., [23, 24, 95–105]) and the memory controller (e.g., [106–112]) to introduce *randomness* in how data is mapped to the DRAM channels/banks/rows or *partition* banks/channels between different threads or applications. While effective, these techniques are *unaware* of the different semantics of data structures in an application, and hence suffer from two shortcomings. First, to determine the properties of data, an application needs to be *profiled* before execution or pages need to be migrated *reactively* based on runtime behavior. Second, these techniques apply the *same* mapping for *all* data structures within the application, even if RBL and MLP vary significantly across different data structures.

XMem enables distinguishing between data structures and provides key access semantics to the OS. Together with the knowledge of the underlying banks, channels, ranks, etc. and other co-running applications, the OS can create an intelligent mapping at the *data structure* granularity. Based on the data structure access patterns, the OS can *(i)* improve RBL by *isolating* data structures with high RBL from data structures that could cause interference if placed in the same bank and *(ii)* improve MLP by *spreading* out accesses to concurrently-accessed data structures across multiple banks and channels.

6.2. Mechanism & Algorithm

The mechanism to support this use case involve three steps. First, the OS obtains the attributes of program data structures by reading the *atom segment* when loading the program (§3.5.2). Second, based on the program semantics of all co-running applications, the OS decides how to map atoms to DRAM channels and banks (described next). Third, the OS leverages XMem's OS interface to obtain the Atom ID mapped to different virtual address ranges (§4.1.2), and manipulates the virtual-to-physical address mapping to place data at specific DRAM banks and channels.

The OS employs an algorithm (described in detail in our technical report [70]) that takes the atom attributes as input, and outputs the bank and channel mapping for each atom. The algorithm first isolates the data structures with high RBL, while ensuring that their access frequencies are high enough that allocating a bank for them does not reduce the overall MLP. The algorithm then spreads out all other data structures to the unallocated banks to maximize MLP.

6.3. Evaluation Methodology

We use zsim [79] and DRAMSim2 [80] with the parameters summarized in Table 3. We strengthen our baseline system in three ways: *(i)* We use the *best*-performing physical DRAM mapping, among all the seven mapping schemes in DRAMSim2 and the two proposed in [106, 107], as our baseline; *(ii)* We *randomize* virtual-to-physical address mapping, which is shown to perform better than the Buddy algorithm [23]; *(iii)* For each workload, we enable the L3 prefetcher *only if* it improves performance. We evaluate a wide range of workloads from SPEC CPU2006 [113], Rodinia [114], and Parboil [115] and show results for 27 memory intensive workloads (with L3 MPKI > 1). We run all applications to completion or for 10 billion instructions, whichever comes first.

6.4. Evaluation Results

We evaluate three systems: *(i)* `Baseline`, the strengthened baseline system (§6.3); *(ii)* `XMem`, DRAM placement using XMem (§6.2); *(iii)* an *ideal* system that has *perfect* RBL, which represents the best performance possible by improving RBL. Figure 7 shows the speedup of the last two systems over `Baseline`. Figure 8 shows the corresponding memory read latency, normalized to `Baseline`. We make two observations.

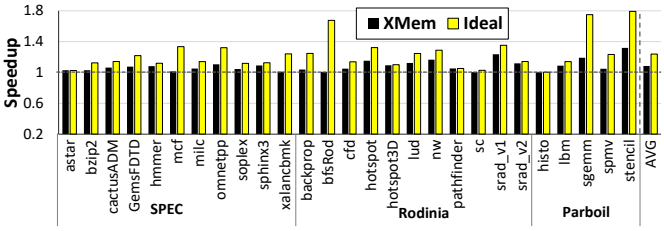

Figure 7: Speedup w/ XMem-based DRAM placement.

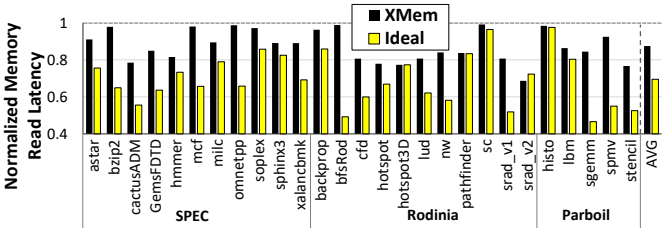

Figure 8: Normalized read latency with XMem-based DRAM placement.

First, XMem-based DRAM placement improves performance across a range of workloads: by 8.5% on average over `Baseline`, up to 31.9%. It is a significant improvement as the absolute upper-bound for *any* DRAM row-buffer optimization is 24.4% (`Ideal`). Among the 27 workloads, only 5 workloads do not see much improvement—they either *(i)* have less than 3% headroom to begin with (`sc` and `histo`) or *(ii)* are dominated by random accesses (`mcf`, `xalancbmk`, and `bfsRod`).

Second, the performance improvement of XMem-based DRAM placement comes from the significant reduction in average memory latency, especially read latency, which is usually on the critical path. On average, `XMem` reduces read latency by 12.6%, up to 31.4%. Write latency is reduced by 6.2% (not shown).

We conclude that leveraging both the program semantics provided by XMem and knowledge of the underlying DRAM organization enables the OS to create intelligent DRAM mappings at a fine (data structure) granularity, thereby reducing memory latency and improving performance.

7. Related Work

To our knowledge, this is the first work to design a holistic and general cross-layer interface to enable the entire system and architecture to be aware of key higher-level program semantics that can be leveraged in memory optimization. We now briefly discuss closely related prior work.

Expressive programming models and runtime systems. Numerous software-only approaches tackle the disconnect between an application, the OS, and the underlying

memory resources via programming models and runtime systems that allow explicit expression of data locality and independence [116–127] in the programming model. This explicit expression enables the programmer and/or runtime system to make effective memory placement decisions in a NUMA system or produce code that is optimized to effectively leverage the cache hierarchy. These approaches have several shortcomings. First, they are entirely software-based and are hence limited to using the *existing* interfaces to the architectural resources. Second, unlike XMem, which is general and only hint-based, programming model-based approaches *require* rewriting applications to suit the model, while ensuring that program correctness is retained. Third, these systems are specific to an application type (e.g., operations on tiles, arrays). XMem is a general interface that is *not* limited to any programming language, application, or architecture. These approaches are orthogonal to XMem, and XMem can be built into them to enable a wider range of memory.

The Locality Descriptor [128] is a cross-layer abstraction to express data locality in GPUs. This abstraction is similar in spirit to XMem in bridging the semantic gap between hardware and software. However, the Locality Descriptor is primarily designed to convey *locality semantics* to leverage cache and NUMA locality in GPUs. XMem aims to convey *general* program semantics to aid memory optimization. This goal imposes different design challenges, requires describing a different set of semantics, and requires optimizing a different set of architectural techniques, leading to a very different cross-layer design for the abstraction.

Leveraging hints, annotations, and software management for memory optimization. A large body of prior work aims to leverage the benefits of static program information in the form of hints, annotations, or directives in memory optimization. These include *(i)* hint-based approaches, such as software prefetch instructions [129] and cache bypass/insertion/eviction hints [1–13, 130]; *(ii)* hardware-software cooperative prefetch techniques [18–22, 38, 131–134] that use compiler analysis or annotations to inform a hardware prefetch engine; and *(iii)* program annotations to place data in heterogeneous memories (e.g., [14, 17, 54]). XMem differs from these works in several ways. First, many of these approaches seek to inform hardware components with *specific directives* that override dynamic policies by enforcing *static* policies. This loss in dynamism introduces challenges when the workload behavior changes, the underlying architecture changes or is unknown (portability), or in the presence of co-running applications [15, 135, 136]. XMem does *not direct* policy at any component but only provides higher-level program semantics. The memory components can use this information to *supplement* their dynamic management policies. Second, the approaches are *specific* to an optimization (e.g., prefetching, cache insertion/eviction). XMem provides a *general* interface to communicate program semantics that can be leveraged by many system/architectural components.

The closest work to ours is Whirlpool [15], which provides a memory allocator to *statically* classify data into pools. Each pool is then managed differently at runtime to place data efficiently in NUCA caches. Whirlpool is similar to XMem in the ability to classify data into similar types and in retaining the benefits of dynamic management. However, XMem is *(i)* more versatile, as it enables *dynamically* classifying/reclassifying data and expressing more powerful program semantics than just static data classification and *(ii)* a general and holistic interface that can be used for a wide range of use cases, including Whirlpool itself. Several prior works [137–143] use runtime systems or the OS to aid in management of the cache. These approaches are largely orthogonal to XMem and can be used in conjunction with XMem to provide more benefit.

Tagged Architectures. Prior work associate software-defined metadata with each memory location in the form of tagged/typed memory [144–147]. These approaches are typically used for fine-grained memory access protection, debugging, etc., and usually incur *non-trivial* performance/storage overhead. In contrast, XMem aims to deliver *general* program semantics to many system/hardware components to aid in performance optimization with *low overhead*. To this end, XMem is designed to enable a number of features and benefits that cannot be obtained from tagged/typed architectures: *(i)* a flexible abstraction to dynamically describe program behavior with `XMemLib`; and *(ii)* low-overhead interfaces to many hardware components to access the expressed semantics. PARD [67] and Labeled RISC-V [148] are tagged architectures that enable labeling memory requests with tags to applications, VMs, etc. These tags are used to convey an application's QoS, security requirements, etc., to hardware. XMem is similar in that it provides an interface to hardware to convey information from software. However, unlike these works, we design a new abstraction (the atom) to flexibly express program semantics that can be seamlessly integrated into programming languages, systems, and modern ISAs. The atom has a low-overhead implementation to convey software semantics to hardware components *dynamically* and at flexible granularities. XMem can *leverage* tagged architectures to communicate atom IDs to hardware components. Hence, PARD and Labeled RISC-V are complementary to XMem.

8. Conclusion

This paper makes the case for richer cross-layer interfaces to bridge the semantic gap between the application and the underlying system and architecture. To this end, we introduce Expressive Memory (XMem), a holistic cross-layer interface that communicates higher-level program semantics from the application to different system-level and architectural components (such as caches, prefetchers, and memory controllers) to aid in memory optimization. XMem improves the performance and portability of a wide range of software and hardware memory optimization techniques by enabling them to leverage key semantic information that is otherwise unavailable. We evaluate and demonstrate XMem's benefits for two use cases: *(i)* static software cache optimization, by leveraging data locality semantics, and *(ii)* OS-based page placement in DRAM, by leveraging the ability to distinguish between data structures and their access patterns. We conclude that XMem provides a versatile, rich, and low overhead interface to bridge the semantic gap in order to enhance memory system optimization. We hope XMem encourages future

work to explore re-architecting the traditional interfaces to enable many other benefits that are not possible today.

Acknowledgments

We thank the ISCA 2018 reviewers and our shepherd, Yungang Bao, for their valuable suggestions. Special thanks go to Vivek Seshadri, Mike Kozuch, and Nathan Beckmann. We acknowledge the support of our industrial partners, especially Google, Huawei, Intel, Microsoft, and VMware. This work was supported in part by SRC, NSF, and ETH Zürich.

References

[1] P. Jain *et al.*, "Software-assisted cache replacement mechanisms for embedded systems," in *ICCAD*, 2001.
[2] Ravindran *et al.*, "Compiler-managed partitioned data caches for low power," in *LCTES*, 2007.
[3] X. Gu *et al.*, "P-OPT: program-directed optimal cache management," in *LCPC*, 2008.
[4] J. Brock *et al.*, "Pacman: Program-assisted cache management," in *ISMM*, 2013.
[5] Z. Wang *et al.*, "Using the compiler to improve cache replacement decisions," in *PACT*, 2002.
[6] K. Beyls *et al.*, "Generating cache hints for improved program efficiency," *JSA*, 2005.
[7] J. B. Sartor *et al.*, "Cooperative caching with keep-me and evict-me," in *INTERACT-9*, 2005.
[8] H. Yang *et al.*, "Compiler-assisted cache replacement: Problem formulation and performance evaluation," *LCPC*, 2004.
[9] J. B. Sartor *et al.*, "Cooperative cache scrubbing," in *PACT*, 2014.
[10] A. Pan *et al.*, "Runtime-driven shared last-level cache management for task-parallel programs," in *SC*, 2015.
[11] V. Papaefstathiou *et al.*, "Prefetching and cache management using task lifetimes," in *ICS*, 2013.
[12] M. Manivannan *et al.*, "RADAR: Runtime-assisted dead region management for last-level caches," in *HPCA*, 2016.
[13] G. Tyson *et al.*, "A modified approach to data cache management," in *MICRO*, 1995.
[14] N. Agarwal *et al.*, "Page placement strategies for GPUs within heterogeneous memory systems," in *ASPLOS*, 2015.
[15] A. Mukkara *et al.*, "Whirlpool: Improving dynamic cache management with static data classification," in *ASPLOS*, 2016.
[16] S. R. Dulloor *et al.*, "Data tiering in heterogeneous memory systems," in *EuroSys*, 2016.
[17] S. Liu *et al.*, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *ASPLOS*, 2011.
[18] T. F. Chen, "An effective programmable prefetch engine for on-chip caches," in *MICRO*, 1995.
[19] S. P. Vander Wiel *et al.*, "A compiler-assisted data prefetch controller," in *ICCD*, 1999.
[20] T. Chiueh, "Sunder: a programmable hardware prefetch architecture for numerical loops," in *SC*, 1994.
[21] E. H. Gornish *et al.*, "An integrated hardware/software data prefetching scheme for shared-memory multiprocessors," in *ICPP*, 1994.
[22] Z. Wang *et al.*, "Guided region prefetching: a cooperative hardware/-software approach," in *ISCA*, 2003.
[23] H. Park *et al.*, "Regularities considered harmful: Forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems," in *ASPLOS*, 2013.
[24] S. P. Muralidhara *et al.*, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *MICRO*, 2011.
[25] G. Pekhimenko *et al.*, "Linearly compressed pages: a low-complexity, low-latency main memory compression framework," in *MICRO*, 2013.
[26] N. Vijaykumar *et al.*, "A case for core-assisted bottleneck acceleration in GPUs: enabling flexible data compression with assist warps," in *ISCA*, 2015.
[27] G. Pekhimenko *et al.*, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT*, 2012.
[28] J. Dusser *et al.*, "Zero-content augmented caches," in *SC*, 2009.
[29] M. Ekman *et al.*, "A robust main-memory compression scheme," in *ISCA*, 2005.
[30] A. Arelakis *et al.*, "SC2: A statistical compression cache scheme," in *ISCA*, 2014.
[31] G. Pekhimenko *et al.*, "Exploiting compressed block size as an indicator of future reuse," in *HPCA*, 2015.
[32] G. Pekhimenko *et al.*, "A case for toggle-aware compression for GPU systems," in *HPCA*, 2016.
[33] T. F. Chen *et al.*, "Effective hardware-based data prefetching for high-performance processors," *TC*, 1995.
[34] S. Somogyi *et al.*, "Spatial memory streaming," in *ISCA*, 2006.
[35] S. Somogyi *et al.*, "Spatio-temporal memory streaming," in *ISCA*, 2009.
[36] S. Volos *et al.*, "Bump: Bulk memory access prediction and streaming," in *MICRO*, 2014.
[37] L. Peled *et al.*, "Semantic locality and context-based prefetching using reinforcement learning," in *ISCA*, 2015.
[38] E. Ebrahimi *et al.*, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *HPCA*, 2009.
[39] R. Cooksey *et al.*, "A stateless, content-directed data prefetching mechanism," in *ASPLOS*, 2002.
[40] M. R. Meswani *et al.*, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *HPCA*, 2015.
[41] X. Jiang *et al.*, "CHOP: Adaptive filter-based DRAM caching for CMP server platforms," in *HPCA*, 2010.
[42] D. Jevdjic *et al.*, "Unison cache: A scalable and effective die-stacked DRAM cache," in *MICRO*, 2014.
[43] D. Jevdjic *et al.*, "Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA*, 2013.
[44] X. Yu *et al.*, "Banshee: Bandwidth-efficient DRAM caching via software/hardware cooperation," in *MICRO*, 2017.
[45] H. Yoon *et al.*, "Row buffer locality-aware data placement in hybrid memories," in *ICCD*, 2012.
[46] J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," in *CAL*, 2012.
[47] J. S. Miguel *et al.*, "Load value approximation," in *MICRO*, 2014.
[48] J. S. Miguel *et al.*, "Doppelgänger: a cache for approximate computing," in *MICRO*, 2015.
[49] B. Thwaites *et al.*, "Rollback-free value prediction with approximate loads," in *PACT*, 2014.
[50] H. Esmaeilzadeh *et al.*, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.
[51] A. Sampson *et al.*, "Approximate storage in solid-state memories," *TOCS*, 2014.
[52] A. Sampson *et al.*, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI*, 2011.
[53] A. Yazdanbakhsh *et al.*, "Rfvp: Rollback-free value prediction with safe-to-approximate loads," *TACO*, 2016.
[54] Y. Luo *et al.*, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *DSN*, 2014.
[55] N. Agarwal *et al.*, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *HPCA*, 2015.
[56] M. Dashti *et al.*, "Traffic management: A holistic approach to memory placement on NUMA systems," in *ASPLOS*, 2013.
[57] B. Wang *et al.*, "Exploring hybrid memory for GPU energy efficiency through software-hardware co-design," in *PACT*, 2013.
[58] Y. Li *et al.*, "Utility-based hybrid memory management," in *CLUSTER*, 2017.
[59] N. Hardavellas *et al.*, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009.
[60] S. Mehta *et al.*, "Tile size selection revisited," *TACO*, 2013.
[61] T. Yuki *et al.*, "Automatic creation of tile size selection models," in *CGO*, 2010.
[62] J. Xiong *et al.*, "SPL: A language and compiler for DSP algorithms," in *PLDI*, 2001.
[63] R. C. Whaley *et al.*, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, 2001.
[64] U. Bondhugula *et al.*, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI*, 2008.
[65] S. Coleman *et al.*, "Tile size selection using cache organization and data layout," in *PLDI*, 1995.
[66] T. Henretty *et al.*, "A stencil compiler for short-vector simd architectures," in *SC*, 2013.
[67] J. Ma *et al.*, "Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD)," in *ASPLOS*, 2015.
[68] Intel Corporation, "Enabling Intel virtualization technology features and benefits." https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf
[69] AMD, "AMD-V nested paging." http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf

[70] N. Vijaykumar *et al.*, "A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory," in *CMU SAFARI Technical Report No. 2018-001*, 2018.
[71] S. Thoziyoor *et al.*, "CACTI 5.1," HP Laboratories, Tech. Rep. HPL-2008-20, 2008.
[72] P. A. Boncz *et al.*, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB*, 1999.
[73] X. Tang *et al.*, "Partitioned similarity search with cache-conscious data traversal," *TKDD*, 2017.
[74] R. J. Bayardo *et al.*, "Scaling up all pairs similarity search," in *WWW*, 2007.
[75] J. Srinivas *et al.*, "Reactive tiling," in *CGO*, 2015.
[76] A. Jain *et al.*, "Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting," in *MICRO*, 2016.
[77] S. Tavarageri *et al.*, "Dynamic selection of tile sizes," in *HiPC*, 2011.
[78] B. Bao *et al.*, "Defensive loop tiling for shared cache," in *CGO*, 2013.
[79] D. Sánchez *et al.*, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA*, 2013.
[80] P. Rosenfeld *et al.*, "DRAMSim2: A cycle accurate memory system simulator," *CAL*, 2011.
[81] L. Pouchet, "Polybench: The polyhedral benchmark suite." http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
[82] N. A. Kurd *et al.*, "Westmere: A family of 32nm IA processors," in *ISSCC*, 2010.
[83] A. Jaleel *et al.*, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *ISCA*, 2010.
[84] S. Rixner *et al.*, "Memory access scheduling," in *ISCA*, 2000.
[85] Y. Kim *et al.*, "ATLAS: a scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.
[86] T. C. Mowry *et al.*, "Design and evaluation of a compiler algorithm for prefetching," in *ASPLOS*, 1992.
[87] J. Dean *et al.*, "The tail at scale," *Commun. ACM*, 2013.
[88] S. Kanev *et al.*, "Profiling a warehouse-scale computer," in *ISCA*, 2015.
[89] M. V. Wilkes, "The memory gap and the future of high performance memories," *CAN*, 2001.
[90] O. Mutlu, "Memory scaling: A systems architecture perspective," in *IMW*, 2013.
[91] O. Mutlu *et al.*, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA*, 2003.
[92] O. Mutlu *et al.*, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.
[93] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.
[94] A. Glew, "MLP yes! ILP no," *ASPLOS WACI*, 1998.
[95] K. Sudan *et al.*, "Micro-pages: increasing DRAM efficiency with locality-aware data placement," *ASPLOS*, 2010.
[96] L. Liu *et al.*, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT*, 2012.
[97] L. Liu *et al.*, "BPM/BPM+: Software-based dynamic memory partitioning mechanisms for mitigating DRAM bank-/channel-level interferences in multicore systems," *TACO*, February 2014.
[98] W. Ding *et al.*, "Compiler support for optimizing memory bank-level parallelism," in *MICRO*, 2014.
[99] R. A. Bheda *et al.*, "Improving DRAM bandwidth utilization with MLP-Aware OS paging," in *MEMSYS*, 2016.
[100] M. Xie *et al.*, "Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning," in *HPCA*, 2014.
[101] W. Mi *et al.*, "Software-hardware cooperative DRAM bank partitioning for chip multiprocessors," *NPC*, 2010.
[102] M. K. Jeong *et al.*, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *HPCA*, 2012.
[103] H. Yun *et al.*, "Palloc: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *RTAS*, 2014.
[104] L. Liu *et al.*, "Going vertical in memory management: Handling multiplicity by multi-policy," in *ISCA*, 2014.
[105] Y. Liu *et al.*, "Locality-aware bank partitioning for shared DRAM MPSoCs," in *ASP-DAC*, 2017.
[106] Z. Zhang *et al.*, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO*, 2000.
[107] D. Kaseridis *et al.*, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *MICRO*, 2011.
[108] H. Vandierendonck *et al.*, "XOR-based hash functions," *TC*, 2005.
[109] Z. Zhang *et al.*, "Breaking address mapping symmetry at multi-levels of memory hierarchy to reduce DRAM row-buffer conflicts," *JILP*, 2001.
[110] M. Ghasempour *et al.*, "Dream: Dynamic re-arrangement of address mapping to improve the performance of DRAMs," in *MemSys*, 2016.
[111] M. Hillenbrand *et al.*, "Multiple physical mappings: Dynamic DRAM channel sharing and partitioning," in *APSys*, 2017.
[112] J. Carter *et al.*, "Impulse: Building a smarter memory controller," in *HPCA*, 1999.
[113] SPEC CPU2006 Benchmarks, "http://www.spec.org/."
[114] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[115] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," UIUC, Tech. Rep. IMPACT-12-01, 2012.
[116] P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," *OOPSLA*, 2005.
[117] B. Chamberlain *et al.*, "Parallel programmability and the chapel language," *IJHPCA*, 2007.
[118] K. Fatahalian *et al.*, "Sequoia: Programming the memory hierarchy," in *SC*, 2006.
[119] S. Treichler *et al.*, "Dependent partitioning," in *OOPSLA*, 2016.
[120] M. Bauer *et al.*, "Structure slicing: Extending logical regions with fields," in *SC*, 2014.
[121] S. Treichler *et al.*, "Realm: An event-based low-level runtime for distributed memory architectures," in *PACT*, 2014.
[122] E. Slaughter *et al.*, "Regent: A high-productivity programming language for HPC with logical regions," in *SC*, 2015.
[123] Y. Yan *et al.*, "Hierarchical place trees: A portable abstraction for task parallelism and data movement." in *LCPC*, 2009.
[124] G. Bikshandi *et al.*, "Programming for parallelism and locality with hierarchically tiled arrays," in *PPoPP*, 2006.
[125] M. Bauer *et al.*, "Legion: Expressing locality and independence with logical regions," in *SC*, 2012.
[126] J. Guo *et al.*, "Programming with tiles," in *PPoPP*, 2008.
[127] D. Unat *et al.*, "Tida: High-level programming abstractions for data locality management," in *HiPC*, 2016.
[128] N. Vijaykumar *et al.*, "The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs," in *ISCA*, 2018.
[129] "Memory management optimizations on the Intel® Xeon Phi™ coprocessor," in *https://software.intel.com/sites/default/files/managed/b4/24/mem_management_dgemm.pdf*. Intel Compiler Lab, 2015.
[130] K. Beyls *et al.*, "Compile-time cache hint generation for EPIC architectures," in *EPIC*, 2002.
[131] P. Jain *et al.*, "Controlling cache pollution in prefetching with software-assisted cache replacement," *Comptation Structures Group, Laboratory for Computer Science CSG Memo*, 2001.
[132] J. Skeppstedt *et al.*, "Hybrid compiler/hardware prefetching for multiprocessors using low-overhead cache miss traps," in *ICPP*, 1997.
[133] M. Karlsson *et al.*, "A prefetching technique for irregular accesses to linked data structures," in *HPCA*, 2000.
[134] A. Fuchs *et al.*, "Loop-aware memory prefetching using code block working sets," in *MICRO*, 2014.
[135] J. Leverich *et al.*, "Comparing memory systems for chip multiprocessors," in *ISCA*, 2007.
[136] N. Vijaykumar *et al.*, "Zorua: A Holistic Approach to Resource Virtualization in GPUs," in *MICRO*, 2016.
[137] D. Chiou *et al.*, "Application-specific memory management for embedded systems using software-controlled caches," in *DAC*, 2000.
[138] N. Beckmann *et al.*, "Jigsaw: Scalable software-defined caches," in *PACT*, 2013.
[139] P. Tsai *et al.*, "Jenga: Software-defined cache hierarchies," in *ISCA*, 2017.
[140] H. Cook *et al.*, "Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments," EECS Department, University of California, Berkeley, Tech. Rep., Sep 2009.
[141] J. Cong *et al.*, "An energy-efficient adaptive hybrid cache," in *ISLPED*, 2011.
[142] C. F. Fajardo *et al.*, "Buffer-integrated-cache: A cost-effective SRAM architecture for handheld and embedded platforms," in *DAC*, 2011.
[143] N. Beckmann *et al.*, "Talus: A simple way to remove cliffs in cache performance," in *HPCA*, 2015.
[144] U. Dhawan *et al.*, "Architectural support for software-defined metadata processing," *ASPLOS*, 2015.
[145] E. A. Feustel, "On the advantages of tagged architecture," *TC*, 1973.
[146] N. Zeldovich *et al.*, "Hardware enforcement of application security policies using tagged memory." in *OSDI*, 2008.
[147] E. Witchel *et al.*, "Mondrian memory protection," in *ASPLOS*, 2002.
[148] Z. Yu *et al.*, "Labeled RISC-V: A new perspective on software-defined architecture," in *CARVV*, 2017.