MetaSys: A Practical Open-source Metadata Management System to Implement and Evaluate Cross-layer Optimizations

NANDITA VIJAYKUMAR, University of Toronto, Canada
ATABERK OLGUN, ETH Zurich, TOBB ETU, Turkey
KONSTANTINOS KANELLOPOULOS, ETH Zurich, Switzerland
F. NISA BOSTANCI, ETH Zurich, TOBB ETU, Turkey
HASAN HASSAN, ETH Zurich, Switzerland
MEHRSHAD LOTFI, Max Plank Institute, Germany
PHILLIP B. GIBBONS, Carnegie Mellon University, USA
ONUR MUTLU, ETH Zurich, Switzerland

This article introduces the first open-source FPGA-based infrastructure, MetaSys, with a prototype in a RISC-V system, to enable the rapid implementation and evaluation of a wide range of cross-layer techniques in real hardware. Hardware-software cooperative techniques are powerful approaches to improving the performance, quality of service, and security of general-purpose processors. They are, however, typically challenging to rapidly implement and evaluate in real hardware as they require full-stack changes to the hardware, system software, and instruction-set architecture (ISA).

MetaSys implements a rich hardware-software interface and lightweight metadata support that can be used as a common basis to rapidly implement and evaluate new cross-layer techniques. We demonstrate MetaSys's versatility and ease-of-use by implementing and evaluating three cross-layer techniques for: (i) prefetching in graph analytics; (ii) bounds checking in memory unsafe languages, and (iii) return address protection in stack frames; each technique requiring only ~100 lines of Chisel code over MetaSys.

Using MetaSys, we perform the first detailed experimental study to quantify the performance overheads of using a *single* metadata management system to enable multiple cross-layer optimizations in CPUs. We identify the key sources of bottlenecks and system inefficiency of a general metadata management system. We design MetaSys to minimize these inefficiencies and provide increased versatility compared to previously proposed metadata systems. Using three use cases and a detailed characterization, we demonstrate that a common metadata management system can be used to efficiently support diverse cross-layer techniques in CPUs. MetaSys is completely and freely available at https://github.com/CMU-SAFARI/MetaSys.

We acknowledge the generous gifts provided by our industrial partners: Google, Huawei, Intel, Microsoft, VMware, and the ETH Future Computing Laboratory.

Authors' addresses: N. Vijaykumar, University of Toronto, 40 St. George St., Toronto, Canada; email: nandita@cs.toronto.edu; A. Olgun and F. N. Bostanci, ETH Zürich, ETZ H 61.2, Gloriastrasse 35, 8092 Zürich, Switzerland; email: ataberk.olgun@safari.ethz.ch; K. Kanellopoulos, H. Hassan, and O. Mutlu, ETH Zürich, ETZ H 64, Gloriastrasse 35, 8092 Zürich, Switzerland; emails: konkanello@gmail.com, hasan.hasan@safari.ethz.ch, omutlu@gmail.com; M. Lotfi, Campus E1 5, 66123 Saarbrücken, Germany; email: lotfimehrshad@gmail.com; P. B. Gibbons, Carnegie Mellon University, Gates-Hillman Center, Office: 7221, Pittsburgh, PA United States; email: gibbons@cs.cmu.edu.

1544-3566/2022/03-ART26 $15.00
https://doi.org/10.1145/3505250

CCS Concepts: • **General and reference** → *Performance*; Experimentation; *Design*;; • **Computer systems organization** → **Architectures**

Additional Key Words and Phrases: Hardware-software cooperation, metadata, memory, RISC-V, open-source

ACM Reference format:
Nandita Vijaykumar, Ataberk Olgun, Konstantinos Kanellopoulos, F. Nisa Bostanci, Hasan Hassan, Mehrshad Lotfi, Phillip B. Gibbons, and Onur Mutlu. 2022. MetaSys: A Practical Open-source Metadata Management System to Implement and Evaluate Cross-layer Optimizations. *ACM Trans. Arch. Code Optim.* 19, 2, Article 26 (March 2022), 29 pages.
https://doi.org/10.1145/3505250

1 INTRODUCTION

Hardware-software cooperative techniques offer a powerful approach to improving the performance and efficiency of general-purpose processors. These techniques involve communicating key application and semantic information from the software to the architecture to enable more powerful optimizations and resource management in hardware. Recent research proposes many such cross-layer techniques for various purposes, e.g., performance, **quality of service (QoS)**, memory protection, programmability, and security. For example, Whirlpool [104] identifies and communicates regions of memory that have similar properties (i.e., data structures) in the program to the hardware, which uses this information to more intelligently place data in a **non-uniform cache architecture (NUCA)** system. RADAR [96] and EvictMe [166] communicate which cache blocks will no longer be used in the program, such that cache policies can evict them. These are just a few examples in an increasingly large space of cross-layer techniques proposed in the form of hints implemented as new ISA instructions to aid cache replacement, prefetching, memory management, and so on [22, 26, 58, 66, 96, 116, 117, 126, 136, 137, 159, 166, 179], program annotations/directives to convey program semantics [3, 47, 58, 87, 104, 163], or interfaces to communicate an application's QoS requirements for efficient partitioning and prioritization of shared hardware resources [62, 93].

While cross-layer approaches have been demonstrated to be highly effective, such proposals are challenging to evaluate on real hardware as they require cross-layer changes to the hardware, **operating system (OS)**, application software, and **instruction-set architecture (ISA)**. Existing open-source infrastructures for implementing cross-layer techniques in real hardware include PARD [62, 93] for QoS and Cheri [174] for fine-grained memory protection and security. Unfortunately, these open-source infrastructures are not designed to provide key features required for *performance* optimizations: (i) rich dynamic hardware-software interfaces, (ii) low-overhead metadata management, and (iii) interfaces to numerous hardware components such as prefetchers, caches, memory controllers, and so on.

In this work, we introduce **MetaSys** (**Meta**data Management **Sys**tem for Cross-layer Performance Optimization), a full-system FPGA-based infrastructure, with a prototype in the RISC-V Rocket Chip system [10], to enable rapid implementation and evaluation of diverse cross-layer techniques in real hardware. MetaSys comprises three key components: (1) A rich **hardware-software interface** to communicate a *general* and extensible set of application information to the hardware architecture at runtime. We refer to this additional application information as *metadata*. Examples of metadata include memory access pattern information for prefetching, data reuse information for cache management, address bounds for hardware bounds checking, and so on. The interface is implemented as new instructions in the RISC-V ISA and is wrapped with easy-to-use software library abstractions. (2) **Metadata management** support in the OS and hardware to store and access the communicated metadata. Hardware components performing optimizations

can then efficiently query for the metadata. We use a *tagged memory-based* design for metadata management where each memory address is tagged with an ID. This ID points to metadata that describes the data contained in the location specified by the memory address. (3) **Modularized components** to quickly implement various cross-layer optimizations with interfaces to the metadata management support, OS, core, and memory system. Our FPGA-based infrastructure provides flexible modules that can be easily extended to implement different cross-layer optimizations.

The closest work to our proposed system is XMem [164]. XMem proposes a general metadata management system that can communicate semantic information at compile time. This limits the use cases supported by XMem. MetaSys has the following benefits over XMem: First, MetaSys offers a richer interface that communicates a flexible amount of metadata at *runtime*, rather than being limited to statically available program information. This enables a wider set of use cases and more powerful cross-layer techniques (as explained in Section 3.8). Second, MetaSys has a more optimized system design that is designed to be *lightweight* in terms of the hardware complexity and changes to the ISA, without sacrificing versatility (Section 3.8). MetaSys incurs only a small area overhead of 0.02% (including 17 KB of additional SRAM), 0.2% memory overhead in DRAM, and adds only eight new instructions to the RISC-V ISA. Third, MetaSys is open-source and freely available, whereas XMem is neither implemented nor evaluated in real hardware with full-system support.

Use cases. Cross-layer techniques that can be implemented with MetaSys include performance optimizations such as cache management, prefetching, memory scheduling, data compression, and data placement; cross-layer techniques for QoS; and lightweight techniques for memory protection (see Section 7). To demonstrate the versatility and ease-of-use of MetaSys in implementing new cross-layer techniques, we implement and evaluate three hardware-software cooperative techniques: (i) prefetching for graph analytics applications; (ii) bounds checking in memory unsafe languages, and (iii) return address protection in stack frames. These techniques were quick to implement with MetaSys, each requiring only an additional ~100 lines of Chisel [13] code on top of MetaSys's hardware codebase (~1,800 lines of code).

Characterizing a general metadata management system. Using MetaSys, we perform the first detailed experimental characterization and limit study of the performance overheads of using a *single* common metadata management system to enable multiple diverse cross-layer techniques in a general-purpose processor. We make four new observations from our characterization across 24 applications and four microbenchmarks that were designed to stress MetaSys.

First, the performance overheads from the cross-layer interface and metadata system itself are on average very low (2.7% on average, up to 27% for the most intensive microbenchmark). Second, there is no performance loss from supporting *multiple* techniques that simultaneously query the shared metadata system. This indicates that MetaSys can be designed to be a scalable substrate. Third, the most critical factor in determining the performance overhead is the fundamental spatial and temporal locality in the accesses to the metadata itself. This determines the effectiveness of the metadata caches and the additional memory accesses to retrieve metadata. Fourth, we identify TLB misses from the required address translation when metadata is retrieved from memory as an important factor in performance overhead.

Conclusions from characterization. From our detailed characterization and implemented use cases on real hardware, we make the following conclusions: First, using a *single* general metadata management system is a promising low-overhead approach to implement *multiple* cross-layer techniques in future general-purpose processors. The significance of using a single framework is in enabling a wide range of cross-layer techniques with a single change to the hardware-software interface [93, 164] and *consolidating* common metadata management support; thus, making the adoption of new cross-layer techniques in future processors significantly easier. Second, we demon-

strate that a common framework can simultaneously and scalably support multiple cross-layer optimizations. For our implemented use cases, we observe low performance overheads from using the general MetaSys system: 0.2% for prefetching, 14% for bounds checking, and 1.2% for return address protection.

MetaSys is fully open-source and freely available at https://www.github.com/CMU-SAFARI/MetaSys.

This work makes the following major contributions.

- We introduce MetaSys, the first full-system open-source FPGA-based infrastructure of a lightweight metadata management system. MetaSys provides a rich hardware-software interface that can be used to implement a diverse set of cross-layer techniques. We implement a prototype of MetaSys in a RISC-V system providing the required support in the hardware, OS, and the ISA to enable quick implementation and evaluation of new hardware-software cooperative techniques in real hardware.
- We propose a new hardware-software interface that enables *dynamically* communicating information and a more streamlined system design that can support a richer set of cross-layer optimizations than prior work [164].
- We present the first detailed experimental characterization of the performance and area overheads of a *general* hardware-software interface and lightweight metadata management system designed to enable *multiple* and diverse cross-layer performance optimizations. We identify key sources of inefficiencies and bottlenecks of a general metadata system on real hardware, and we demonstrate its effectiveness as a common substrate for enabling cross-layer techniques in CPUs.
- We demonstrate the versatility and ease-of-use of the MetaSys infrastructure by implementing and evaluating three hardware-software cooperative techniques: (i) prefetching for graph analytics applications; (ii) efficient bounds checking for memory-unsafe languages; and (iii) return address protection for stack frames. We highlight other use cases that can be implemented with MetaSys.

2 BACKGROUND AND RELATED WORK

Hardware-software cooperative techniques in CPUs. Cross-layer performance optimizations communicate *additional* information across the application-system boundary. We refer to this information as *metadata*. Metadata that is typically useful for performance optimization include program properties such as access patterns, read-write characteristics, data locality/reuse, data types/layouts, data "hotness," and working set size. This metadata enables more intelligent hardware/system optimizations such as cache management, data placement, thread scheduling, memory scheduling, data compression, and approximation [162–164]. For QoS optimizations, metadata includes application priorities and prioritization rules for allocation of resources such as memory bandwidth and cache space [48, 62, 76, 93, 106, 107, 152, 153]. Memory safety optimizations may communicate base/bounds addresses of data structures [43, 45].

A *general* framework is a promising approach as it enables many cross-layer techniques with a single change to the hardware-software interface and enables *reusing* the metadata management support across multiple optimizations. Such systems were recently proposed for performance [163, 164], memory protection and security [45, 174], and QoS [62, 93].

A general framework to support a wide range of cross-layer optimizations—specifically for *performance*—requires (i) a rich and dynamic hardware-software interface to communicate a diverse set of metadata at runtime, (ii) lightweight and *low-overhead* metadata management [164], and (iii) interfaces to numerous hardware components. Even small overheads imposed as a result of the system's generality may overshadow the performance benefits of a cross-layer technique.

General metadata systems may also impose significant complexity, performance, and power overheads to the processor. While prior work has demonstrated the significant benefits of cross-layer approaches, no previous work has characterized the efficiency and capacity limits of a general metadata system for cross-layer optimizations in CPUs.

Tagged architectures. MetaSys is inspired by the metadata management and interfaces proposed in XMem [164] and the large body of work on tagged memory [45, 53, 68, 173, 182] and capability-based systems [27, 85, 168, 174]. We compare against the closest prior work, XMem, qualitatively in Section 3.8 and quantitatively in Section 5. Unlike all above works, our goal is to provide an open-source framework to implement and these prior cross-layer approaches in real hardware and to perform a detailed real-system characterization of such metadata systems for performance optimization.

Infrastructure for evaluating cross-layer techniques. Evaluating the overheads and feasibility of a newly proposed cross-layer technique is non-trivial. Fully characterizing the performance and area overheads either with a full-system cycle-accurate simulator or an FPGA implementation requires implementing: (i) *Hardware support* to implement the mechanism; (ii) *OS support* for OS-based cross-layer optimizations and to characterize the context-switch and system overheads of saving and handling a process' metadata; and (iii) *Compiler support and ISA modifications* to add and recognize new instructions to communicate metadata.

Recent works propose general systems that are designed to enable cross-layer techniques for QoS (PARD [62, 93]) or fine-grained memory protection and security (Cheri [174]). PARD enables tagging of components and applications with IDs that are propagated with memory requests and enforcing QoS requirements in hardware. Cheri [174] is a capability-based system that provides hardware support and ISA extensions to enable fine-grained memory protection. Neither system supports the (i) communication of diverse metadata at runtime, (ii) flexible granularity tagging of memory to enable efficient metadata lookups from multiple components, or (iii) interfaces to numerous hardware components (such as the prefetcher, caches, memory controllers) that are needed for *performance* optimization.

Our Goal. Our goal in this work is twofold. First, we aim to develop an efficient and flexible *open-source* framework that enables rapid implementation of new cross-layer techniques to evaluate the associated performance, area, and power overheads, and thus their benefits and feasibility, in real hardware.

Second, we aim to perform the first detailed limit study to characterize and experimentally quantify the overheads associated with *general* metadata systems to determine their practicality for performance optimization in future CPUs.

3 METASYS: ENABLING AND EVALUATING CROSS-LAYER OPTIMIZATIONS

To this end, we develop MetaSys , an open-source full-system FPGA-based infrastructure to implement and evaluate new cross-layer techniques in real hardware. MetaSys includes: (i) a rich hardware-software interface to dynamically communicate a flexible amount of metadata at runtime from the application to the hardware, using new RISC-V instructions; (ii) a tagged memory-based [45, 53, 68, 173, 182] implementation of metadata management in the system and OS; and (iii) flexible modules to add new hardware optimizations with interfaces to the metadata, processor, memory, and OS. We build a prototype of MetaSys in the RISC-V Rocket Chip [10] system.

We choose an FGPA implementation as opposed to a full-system simulator as: (i) This enables us to focus on feasibility as all components need to be fully implemented (e.g., ports, wires, buffers) and their impact on area, cycle time, power, and scalability is quickly visible. (ii) FPGAs are much faster, running full application simulations in a few minutes/hours as opposed to many days on a full-system simulator, making FPGAs a better fit for quick experimentation. (iii) The RTL generated

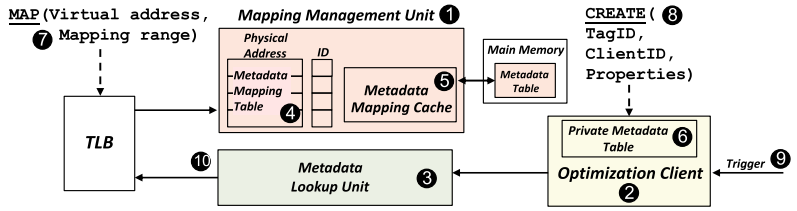

Fig. 1. MetaSys hardware components and operation. MetaSys' structures are highlighted.

can be used for more accurate area and power calculation and potential future synthesis on other systems.

Figure 1 depicts an overview of the major hardware components in MetaSys and their operation: The mapping management unit ❶, the optimization client ❷, and the metadata lookup unit ❸.

3.1 Tagged Memory-based Metadata Management

Similar to prior systems for taint-tracking, security, and performance optimization, MetaSys implements *tagged memory*-based [53, 68, 173, 182] metadata management. MetaSys associates metadata with memory address ranges of arbitrary sizes by tagging each memory address with an 8-bit (configurable) ID or tag. Each tag is a unique pointer to metadata that describes the data at the memory address. Hardware optimizations (e.g., in the cache, memory controller, or core) can query for the tag associated with any memory address and the metadata associated with the tag.

The mapping between each memory address and the corresponding ID is saved in a table in memory referred to as Metadata Mapping Table (MMT): ❹ in Figure 1. This table is allocated by the OS for each process and is saved in memory. In MetaSys (similar to XMem [164] and Cheri [174]), we tag *physical addresses*. As a result, any virtual address has to be translated before indexing the MMT to retrieve the tag ID. To enable fast retrieval of IDs, we implement a cache for the MMT in hardware that stores frequently accessed mappings, referred to as the **Metadata Mapping Cache (MMC)** ❺. MMC misses lead to memory accesses to retrieve mappings from the MMT in memory.

MetaSys can be configured to tag memory at flexible granularities. In Section 9.1, we evaluate the performance impact of the tagging granularity. The size of the MMT depends on the tagging granularity. For a 512 B mapping granularity, the MMT requires 0.2% of physical memory (16 MB in a 8 GB system). The MMC holds 128 entries, where each entry stores a physical-address-to-tag mapping, and is 608 B in size (8 bit entry and 30 bit tag).

We implement dedicated mapping tables for tag IDs rather than use the page table or TLBs for the following reasons: First, doing so obviates the need to modify the latency-critical address translation structures. Second, MetaSys associates physical addresses with Tag IDs rather than virtual addresses (to enable the memory controller and LLCs to look up metadata). Thus, a page table or TLB cannot be directly used to save Tag IDs as they are indexed with virtual addresses.

The actual metadata associated with any ID is saved in special SRAM caches that are private to each hardware component or optimization. For example, the prefetcher would separately save access pattern information, while a hardware bounds checker would privately save data structure boundary information. We refer to these stores as Private Metadata Tables (PMTs) ❻. The PMTs are saved near each component (private to each component) and are loaded/updated by MetaSys. The metadata (e.g., locality/"hotnes") is encoded such that it can be directly interpreted by the component, e.g., a prefetcher.

Table 1. MetaSys Instructions

MetaSys Operator	MetaSys ISA Instructions
CREATE	CREATEClientID, TagID, Metadata
(UN)MAP	(UN)MAP TagID, start_addr, size (UN)MAP2D TagID, start_addr, lenX, sizeX, sizeY (UN)MAP3D TagID, start_addr, lenX, lenY, sizeX, sizeY, sizeZ;

Table 2. MetaSys Software Library Function Calls

Library Function Call	Description
CREATE(*ClientID, TagID, *meta*)	*ClientID* -> PMT[*TagID*] = **metadata*
MAP(*start*, end*, TagID*)	MMT[*start...end*] = *TagID*
UNMAP(*start*, end**)	MMT[*start...end*] = 0

3.2 The Hardware-Software Interface

Communicating application information with MetaSys requires (i) associating memory address ranges with a tag or ID of configurable size (8 bits by default) and (ii) associating each ID with the relevant metadata. The metadata could include program properties that describe the memory range, such as data locality/reuse, access patterns, read-write characteristics, data "hotness," and data types/layouts. We use two operators (described below) that can be called in programs to dynamically communicate metadata.

To associate memory address ranges with an ID, we provide the MAP/UNMAP interface ❼ (similar to XMem [164]). MAP and UNMAP are implemented as new RISC-V instructions that are interpreted by the **Mapping Management Unit (MMU)** to map a range of memory addresses (from a given virtual address up to a certain length) to the provided ID. These mappings are saved by the MMU in the MMT. We also implement 2D and 3D versions of MAP to efficiently map two-/three-dimensional address ranges in a multi-dimensional data structure with a single instruction.

To associate each ID with metadata, we provide the CREATE interface. CREATE ❽ takes three inputs from the application: the tag ID, the 8-bit ID for the *hardware component* (i.e., prefetcher, bounds checker, etc., called Module ID), and 512 B of metadata. CREATE directly populates the PMT of the appropriate hardware component with 512 B of (or less) metadata. Each PMT (private to the optimization client) has 256 entries assuming 8-bit tag IDs. The CREATE operator overwrites the metadata at the entry indexed by the tag ID at the PMT specified by the module ID. All CREATE and MAP instructions are associated with the *next* load/store instruction in program order to avoid inaccuracies due to out-of-order execution. In other words, an implicit dependence is created in hardware between these instructions and the next load/store, and they are committed together. This enables associating information with the next load/store and not just the memory region associated with it, e.g., in the bounds checking use case described in Section 6.1.

Table 1 lists the new instructions along with their arguments.

3.3 Metadata Lookup

Each optimization component is triggered by a hardware event ❾ (e.g., a cache miss). A component then retrieves the physical address corresponding to the virtual address associated with the event (e.g., the virtual address that misses in the cache) from the TLB ❿ (in case of L1 optimizations) and queries the MMC with the physical address to retrieve the associated tag ID. On a miss in the MMC, the mapping is retrieved from the MMT in memory. The optimization client uses the retrieved tag ID to obtain the appropriate metadata from the PMT. The optimization client is designed to flexibly implement a wide range of use cases and can be designed based on the optimization at hand. For example, the optimization client used to build the prefetcher use case in Section 5 has interfaces to the prefetcher, caches, memory controller, and TLBs to make implementing optimizations easier. Each client has a static ID (clientID) and a PMT that is updated by the CREATE operator.

3.4 Operating System Support

We add OS support for metadata management in the RISC-V proxy kernel [128], which can be booted on our Rocket RISC-V prototype: First, we add support to manage the MMT in memory,

where the OS allocates the MMT in the physical address space and communicates the pointer to the MAP hardware support. Second, we add support to flush the PMTs during a context switch (similar to how the TLB is flushed). Third, if the OS changes the virtual to physical address mapping of a page, then to ensure consistency of the metadata, the MMT is updated by the OS to reflect the correct physical-address-to-tag-ID mapping and the corresponding MMC entries are invalidated. We modify the page allocation mechanism in the OS to do this. In addition, we also provide support to implement optimizations performed by the OS or with OS cooperation. To do so, MetaSys enables trapping into the OS to perform customized checks or optimizations (e.g., protection checks or altering virtual-to-physical mappings) based on specific hardware trigger events (using interrupt routines). We describe one such use case in Section 6.

3.5 Coherence/Consistency of Metadata in Multicore Systems

MetaSys can be flexibly extended to multicore processors. Metadata is maintained at a process-level, therefore, threads within the same process cannot have different metadata for the same data structure. The MMC is a per-core structure, while the **Private Metadata Tables (PMTs)** are per-component structures (e.g., at the memory controller, LLC, prefetcher). The two dynamic operators (`CREATE` and `MAP`) may cause challenges in coherence and consistency of metadata in multicore systems. `CREATE` directly updates metadata associated with the *per-process* tag ID, which is saved at the per-component PMTs. The PMTs are shared by all cores when the optimization component is also shared (and thus any updates by `CREATE` are automatically coherent). The PMTs for private components (e.g., L1 cache) are not coherent and can only be updated by the corresponding thread. `MAP` updates the mapping in the MMC, which is private to each core. To ensure coherence of the MMC mappings, a `MAP` update invalidates the corresponding MMC entry (if present) in other MMCs by broadcasting updates with a snoopy protocol. If the use case requires consistency of the metadata, i.e., ordering between a `CREATE`/`MAP` instruction and when it is visible to other cores, then barriers and fence instructions are used to enforce any required ordering between threads for updates to metadata.

3.6 Timing Sensitivity of Metadata

MetaSys supports three modes: (i) *Force stall*, where the instruction triggering a metadata lookup cannot commit until the optimization completes (e.g., for security use cases); (ii) *No stall*, where metadata lookups do not stall the core but are always resolved (e.g., for page placement, cache replacement); and (iii) *Best effort*, where lookups may be dropped to minimize performance overheads (e.g., for prefetcher training).

3.7 Software Library

We develop a software library that can be included in user programs to facilitate the use of MetaSys primitives `CREATE` and `MAP` (Table 2). The library exposes three functions: (i) `CREATE` populates an entry indexed by the tag ID (*TagID*) in the PMT of a hardware optimization client (*ClientID*) with the corresponding metadata; (ii) `MAP` updates the MMT by assigning tag IDs to memory addresses of the range (*start, end*); (iii) `UNMAP` resets the tag IDs of the corresponding address range in the MMT. While the operators can be directly used via the provided software library, their use can be simplified by using wrapper libraries that abstract away the need to directly manage tag IDs and their mappings.

3.8 Comparison to the XMem Framework [164]

MetaSys implements a tagged-memory-based system with a metadata cache similar to XMem [164]. MetaSys however has three major benefits over XMem. First, MetaSys enables communicating

Table 3. Comparison between MetaSys and XMem Interfaces

Operator	XMem [164]	MetaSys
CREATE	Compiler pragma to communicate static metadata at program load time.	Selects a hardware optimization, dynamically associates metadata with an ID, and communicates both to hardware at runtime (implemented as a new instruction).
(UN)MAP	Associate memory ranges with tag IDs (implemented as new instructions).	Same semantics and implementation as XMem.
(DE)ACTIVATE	Enable/disable optimizations associated with a tag ID (implemented as new instructions).	**Does not exist** as the same functionality can now be done with CREATE.

metadata at *runtime* using a more powerful CREATE operator that is implemented as a new instruction. In XMem, metadata is communicated only *statically* at compile time (CREATE is hence a compiler pragma). MetaSys thus enables a wider set of optimizations including fine-grained memory safety, protection, prefetching, and so on, and enables communicating metadata that is dependent on program input and metadata that can be accurately known only at runtime (e.g., access patterns, data "hotness," etc.). MetaSys was designed to efficiently handle these dynamic metadata updates. Second, the dynamic and more expressive CREATE operator obviates the need for additional interfaces (ACTIVATE/DEACTIVATE) to track the validity of statically communicated metadata. This enables a more streamlined metadata system in MetaSys with fewer new instructions, tables, and lookups. Third, MetaSys allows the application programmer to directly select which cross-layer optimization to enable/disable and communicate metadata to, via the CREATE operator. XMem, however, does not allow control of hardware optimizations from the application. Table 3 summarizes the MetaSys operators and compares to the corresponding operators in XMem. *Of the three MetaSys use cases we evaluate in this article, only return address protection (Section 6.2) can be implemented with XMem.*

3.9 FPGA-based Infrastructure

We build a full system prototype of MetaSys on an FPGA with the Rocket Chip RISC-V system [10] and add the necessary support in the compiler, libraries, OS, ISA, and hardware. The modularized MetaSys components can also be ported to other RISC-V cores. We used the RoCC accelerator [10] in the Rocket chip to implement the metadata management system. RoCC is a customizable module that enables interfacing with the core and memory. The hardware support implemented in ROCC comprises (i) the control logic to handle MAPs and CREATEs, (ii) control logic to perform metadata lookups by components that implement optimizations, and (iii) the memory for metadata caches (MMC and PMTs). We extended the RISC-V ISA with eight instructions (six for MAP/CREATE and two for OS operations). To implement all the hardware modules of MetaSys, we modified/added 1,781 lines of Chisel code in the Rocket Chip. As we demonstrate later, since the MetaSys hardware modules can be flexibly reused across multiple hardware-software optimizations, the techniques in our use cases only required 87–103 additional lines of Chisel code. The full MetaSys infrastructure is open-sourced [57] including the Chisel code for the MetaSys hardware support, the RISC-V OS with the required modification, and the software libraries to expose the MetaSys primitives.

3.10 Implementing a Hardware-Software Cooperative Technique with MetaSys

To implement a new hardware technique with the baseline MetaSys code, we provide a flexible module (❶ in Figure 2) with a PMT and interfaces to the metadata lookup unit, to the core (to receive triggers), and interfaces to the cache controller. The interface to the lookup unit ❷ provides dynamic access to the metadata communicated by the CREATE and MAP operators. The interfaces to the core ❸ and the memory system ❹ can be used as *trigger* events for optimization and lookups

Fig. 2. MetaSys Optimization Client.

(e.g., a cache miss). The different components within the MetaSys logic itself (i.e., the metadata caches, logic to access the Metadata Mapping Table in memory, and the lookup logic) can be flexibly reconfigured.

3.11 Dynamically Typed or Managed Languages

MetaSys relies heavily on function calls/libraries that abstract away low-level details that call the MetaSys instructions even in C/C++. With managed and dynamically typed languages, the metadata associated with data structures/objects would be provided by the user with additional class/object member functions. The metadata could also be directly embedded within object/class definitions (e.g., a list or map in Python would by definition have certain access properties). Other properties (e.g., data types) would be provided by the interpreter (in the case of dynamically typed languages) and the mapping/remapping calls to memory addresses would be handled by the runtime during memory (de)allocation.

3.12 Comparison to Specialized Cross-layer Solutions

In comparison to specialized cross-layer solutions, MetaSys offers the following benefits: (i) Generality: toward implementing a large number of use cases, including more complex use cases such as specialized prefetching (Section 5), which amortizes the overall hardware cost; (ii) Flexibility and versatility in the implemented instructions: A challenge with specialized cross-layer solutions is the need to add new instructions that create challenges in forward/backward compatibility and also require changes across the stack for each new optimization. With MetaSys, the instructions are designed to be agnostic to the optimization and only require a one-off change to the hardware-software interface; (iii) Infrastructure for evaluation: MetaSys can be used to implement many specialized cross-layer techniques in real hardware, which would otherwise be a challenging programming task (as demonstrated in Sections 5 and 6). In Sections 5 and 6, we evaluate MetaSys's ability to implement several cross-layer techniques.

4 METHODOLOGY

Baseline system. We use the in-order Rocket core [10] as the baseline CPU and conduct our experiments on the ZedBoard Zynq-7000 [12] FPGA board. Table 4 lists the parameters of the core and memory system as well as evaluated workloads.[1] MetaSys does not require any changes to support an L2/LLC and optimization modules for an L2/LLC can be flexibly implemented similar to the L1. The cost of an MMC miss may be further alleviated with an L2/LLC that reduces access to memory.

5 USE CASE 1: HW-SW COOPERATIVE PREFETCHING

Hardware-software cooperative prefetching techniques have been widely proposed to handle challenging access patterns such as in graph processing [4–6, 18, 103, 113, 157, 181, 185],

[1]Since DRAM is disproportionately faster than the CPU clock rate on FPGAs, we added logic in the memory controller to scale the rate at which memory requests are issued. The resulting average memory latency and bandwidth in core cycles were validated with microbenchmarks against a real CPU.

Table 4. Parameters of the Evaluated Real FPGA-based System

CPU: 25 MHz; in-order Rocket core [10]; **TLB** 16 entries DTLB; LRU policy
L1 Data + Inst. Cache: 16 KB, 4-way; 4-cycle; 64 B line; LRU policy; MSHR size: 2
MMC: NMRU Policy; 128 entries; 38 bits/entry; **Tagging Granularity:** 512 B
Private Metadata Table: 256 entries; 64 B/entry; **DRAM:** 533 MHz; V_{dd}: 1.5 V
Workloads: Ligra [145]: PageRank (PR), Shortest Path (SSSP), Collaborative Filtering (CF), Teenage Follower (TF), Triangle Counting (TC), Breadth-First Search (BFS) Radius Estimation (Radii), Connected Components (CC); **Polybench** [124]; **μBenchmarks**

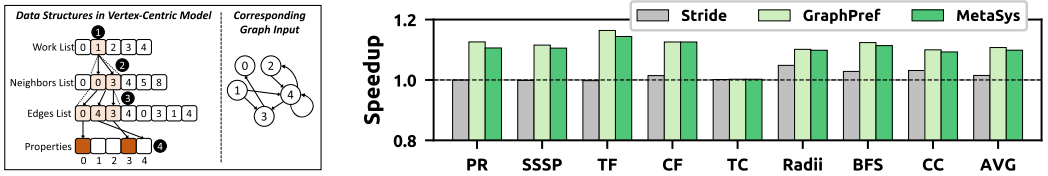

Fig. 3. Data-dependent accesses in vertex-centric graph processing model (left), speedup with the MetaSys prefetcher (right).

pointer-chasing [7, 30, 49, 131, 132, 187], linear algebra computation [32], and other applications [9, 120, 165, 167]. In this section, we demonstrate how MetaSys can be flexibly used to implement and evaluate such prefetching techniques. We design a new prefetcher for graph applications that leverages knowledge of the semantics of graph data structures using MetaSys. Graph applications typically involve irregular pointer-chasing-based memory access patterns. The data-dependent non-sequential accesses in these workloads are challenging for spatial [14, 21, 23, 52, 54, 55, 64, 71, 75, 80, 82, 99, 115, 125, 139, 140, 148, 150], temporal [15, 19, 31, 33, 35, 51, 61, 65, 70, 73, 147, 169–171, 175, 176], and learning-based hardware prefetchers [20, 59, 121, 122, 141–143, 184] that rely either on (i) program context information (e.g., program counter, cache line address) or (ii) memorizing long sequences of cache line addressses to generate accurate prefetch requests.

To implement the hardware support for our prefetcher, we only needed to add 87 lines of Chisel code to the baseline MetaSys codebase, all within the provided module for new optimization components.

5.1 Hardware-Software Cooperative Prefetching for Graph Analytics with MetaSys

Vertex-centric graph analytics typically involves first traversing a *work list* containing vertices to be visited (❶ in Figure 3, left). For each vertex, the application accesses the *vertex list* ❷ to retrieve the neighboring vertex IDs from the *edge list* ❸. To perform computation on the graph, the application then operates on the properties of these neighboring vertices (retrieved from the property list ❹). Graph processing thus involves a series of memory accesses that depend on the contents of the work, vertex and edge lists.

In this use case, we design a prefetcher that can interpret the contents of each of the above data structures and appropriately compute the next data-dependent memory address to prefetch. To capture the required application information for each data structure, we use MetaSys's `CREATE` interface to communicate the following metadata: (i) base address of the data structure that is indexed using the current data structure's contents (64 bits); (ii) base address of the current data structure (64 bits); (iii) data type (32 bits) and size (32 bits) to determine the index of the next access; and (iv) the prefetching stride (6 bits). `MAP` then associates the address range of each data structure with the appropriate tag.

```
1  /* Additional Code in BFS */
2  metadata_create(0, 1, WorkList, { sizeof(Worklist), VertexList, Stride }); // Create metadata (arguments: ClientID=0, tag ID, metadata)
3  metadata_create(0, 2,VertexList, { sizeof(VertexList), EdgeList, Stride});
4  metadata_create(0, 3,EdgeList, { sizeof(Worklist), VertexList, Stride});
5  metadata_create(0, 4,Property, {sizeof(Property), NULL, Stride});
6
7  metadata_map((void*) (WorkList), mapSize, 1); // Map tag 1 to Worklist
8  metadata_map((void*) (VertexList), mapSize, 2);
9  metadata_map((void*) (EdgeList), mapSize, 3);
10 metadata_map((void*) (Property), mapSize, 4);
11
12 /* Hardware Prefetcher Functionality */
13 void snoop_mem_request ( address ): // snoop every memory request
14       (Valid,Base,Bounds,PointerToNextDS,Stride) = metadata_lookup (address); // Access MetaSys using the address
15
16        while (Valid && PointerToNext != NULL) // While the data structure traversal is not complete
17         if (Base < address && address > Bounds) // If the memory request comes from a tracked data structure
18              initiate_prefetch(address+stride); // Initiate a stride prefetch request
19              Value = wait_for_value(address+stride); // Wait for the prefetch request to return data
20              address = &PointerToNextDS[value]; // Discover the address of the next data structure (DS)
21              (Valid,Base,Bounds,PointerToNextDS,Stride) = metadata_lookup (PointerToNextDS[value]); // Look up metadata for the next DS
```

Listing 1. Metasys-based Graph Prefetcher. Available online [57].

Listing 1 shows a detailed end-to-end example of how metadata is created in the application (BFS), how metadata tags are associated with the data structures of BFS, and how the prefetcher operates. Lines 2-10 (incorporated into the code of the BFS application) use the MetaSys software libraries to create metadata (with `CREATE`) and associate it with the corresponding data structures (using `MAP`). `CREATE` saves the metadata in the PMT and `MAP` updates the MMT. Lines 13-21 (incorporated into the hardware optimization client responsible for prefetching) describe the algorithm behind the MetaSys-based prefetcher. The prefetcher is implemented with an optimization client (ClientID = 0). The prefetcher essentially: (i) snoops every memory request from the core and retrieves the associated tag ID using MetaSys; (ii) queries the PMT to retrieve the communicated metadata (listed above); and (iii) uses the metadata to identify dependencies between the data structures of the application.

We describe a detailed walkthrough of how the prefetcher operates during the execution of the BFS application using Figure 3 and Listing 1. In Figure 3 (left), when the prefetcher snoops a memory request that targets the work list at index 0, it looks ahead (depending on the prefetching stride) to retrieve the contents of the work list at index 1. At this point, it also prefetches the contents of the vertex, edge, and property lists based on the computed index at each level. In graph applications where the work list is ordered, the prefetcher is configured to simply stream through the contents of the vertex and edge lists to prefetch the data dependent memory locations in the property list. The *snoop_mem_request*(*address*) (Line 13) function is executed for each request sent by the core to the memory hierarchy. For every memory request, the prefetcher accesses the MMC using the address to receive the tag ID (using MetaSys's lookup functionality). Next, it indexes the PMT using the tag ID to retrieve the metadata associated with the memory request. Using the metadata, the prefetcher determines if the request comes from one of the data structures of the application (Line 17). In this case, the prefetcher first prefetches ahead (Line 18) according to the stride and waits until it receives the value of the prefetched request (Line 19). Using the value, it calculates the address of the data-dependent data structure (e.g., value of WorkList used as an index for VertexList) and looks up the metadata for the newly formed address. The same procedure happens until no further data-dependency is found (Line 16).

The prefetcher can be flexibly configured (by associating metadata to data structures, Lines 2–10 in Listing 1) by the user based on the specific properties associated with any data structure, algorithm, and the desired aggressiveness of prefetcher.

5.2 Evaluation and Methodology

We evaluate the MetaSys-based prefetcher using eight graph analytics workloads from the Ligra framework [145] using the Rocket Chip prototype of MetaSys with the system parameters listed in Table 4. We evaluate three configurations: (i) the baseline system with a hardware stride prefetcher [55]; (ii) *GraphPref*, a customized hardware prefetcher that implements the same idea described above without the generalized MetaSys support (similar to prior work [5, 157]); and (iii) the MetaSys-based graph prefetcher. In the case of *GraphPref*, all the required metadata (e.g., base and bound addresses, stride) are directly provided to the prefetcher using specialized instructions. Thus, *GraphPref* is able to access metadata at low latency and does not access the memory hierarchy. The prefetcher works in the same way as the MetaSys-based prefetcher, however, in the case of MetaSys, the general CREATE/MAP instructions are used to communicate information and the metadata lookups access the MMC (which may lead to additional memory accesses when there is an MMC miss). Figure 3 (right) depicts the corresponding speedups, normalized to the baseline. We observe that the MetaSys graph prefetcher improves performance by 11.2% on average (up to 14.3%) over the baseline by accurately prefetching data-dependent memory accesses. It also significantly outperforms the stride prefetcher, which is unable to capture the irregular access patterns in graph workloads. Compared to *GraphPref*, the MetaSys-based prefetcher performs almost as well: within 0.2% on average (within 0.8% for `BFS`). The additional overheads of MetaSys come from the MMC misses and the larger number of instructions used. In terms of area, MetaSys requires 17 KB of SRAM (1 KB for the MMC and 16 KB for the Private Metadata Table) compared to the custom hardware prefetcher, which requires 8 KB of SRAM for the metadata. The custom prefetcher requires two additional instructions and additional logic to perform metadata lookups and create/update metadata. We found the area complexity to be slightly less for the custom solution as the SRAM requirements are lower (~0.01% for custom hardware versus ~0.02% for MetaSys, compared to a 22 nm Intel CPU Core [144]). However, MetaSys's overhead can be amortized over multiple use cases, whereas a custom solution is specific to a single use case.

We conclude that MetaSys can be used to flexibly implement and evaluate hardware-software cooperative techniques for prefetching by leveraging MetaSys's metadata support and interfaces, incurring only small overheads from MetaSys's general metadata management.

6 USE CASE 2: MEMORY SAFETY AND PROTECTION

We describe two hardware-software cooperative mechanisms for memory safety and protection that can be directly implemented with MetaSys. *To implement both use cases, we only add 103 lines of Chisel code to the baseline MetaSys code, all within the new optimization client that is used in both use cases.*

6.1 Hardware Bounds Checking

Unmanaged languages such as C/C++ provide great flexibility in memory management but an important challenge with these languages is *memory safety*. The pointer casting and pointer arithmetic supported by these languages allow buffer overflows and potentially hazardous writes to arbitrary memory locations. Prior work has demonstrated a range of software approaches [11, 34, 46, 50, 60, 67, 79, 100, 111, 112, 114, 119, 133, 173, 177, 180] to increase memory safety in the form of static or dynamic checks, such as CCured [112], Cyclone [67], and Softbound [111]. These approaches are known to incur significant runtime overheads in performing numerous checks in software [156]. Hardware-based approaches offer a promising opportunity to alleviate these overheads. Prior work [42, 43, 84, 98, 109, 110, 161, 174], including HardBound [43], ShaktiT [98], and Cheri [174] investigate enabling hardware-software cooperative bounds checking. These

```
1   /* Example bounds checking software */
2   metadata_map((void*) (array1), mapSize, 1); // Map TagID 1 to array "array1"
3   metadata_map((void*) (array2), mapSize, 2); // Map TagID 2 to array "array2"
4   metadata_map((void*) (array3), mapSize, 3); // Map TagID 3 to array "array3"
5
6   // Access every element of each array with a stride of one element
7   for(i = 0 ; i < array_size ; i += 1)
8   {
9       metadata_create(0,1,1); // Create metadata with TagID=1 and Metadata=1 to ClientID=0
10      int elem1 = array1[i];
11      metadata_create(0,2,2); // Create metadata with TagID=2 and Metadata=2 to ClientID=0
12      int elem2 = array2[i];
13      int result = elem1 + elem2;
14      metadata_create(0,3,3); // Create metadata with TagID=3 and Metadata=3 to ClientID=0
15      array3[i] = result;
16  }
17
18  /* Hardware Bounds Checker Functionality */
19  HardwareBoundsChecker(CreateTagID, Address):
20      TagIDRegister <= CreateTagID // Software communicates TagID using CREATE
21      MetadataTagID <= PerformMetadataLookup(Address) // Bounds checker client performs a metadata lookup to find the TagID of address
22      if MetadataTagID != TagIDRegister: // Interrupt rocket core if TagIDs do not match (i.e., access is out of bounds)
23          InterruptRocketCore()
```

Listing 2. MetaSys-based bounds checking example. Full source code is available online [57].

approaches require architectures that are entirely specialized for bounds checking [43, 84, 98] or more heavyweight metadata management systems tailored for memory security and protection [42, 109, 110, 161, 174]. In this section, we demonstrate how MetaSys can be used to implement hardware-based bounds checking at low overhead using a lightweight and general metadata system.

6.1.1 Implementing bounds checking with MetaSys. We use Listing 2, where two arrays A and B are traversed with a stride of one element, to illustrate the mechanism. To implement bounds checking with MetaSys, we use the `MAP` operator to tag each data structure to be protected with a unique ID (lines 2–4). For dynamically allocated nodes (which may not be contiguously located), each node is tagged with the *same* ID as other nodes in the same data structure. Every memory access in the program then needs to be verified in hardware to be going to the correct data structure. To do this, we add the `CREATE` operator before every load or store to a protected data structure (lines 9, 11, and 14). The `CREATE` operator in this case communicates the tag ID of the desired data structure as metadata and the ClientID of the bounds checking hardware. In hardware, we simply check whether there is a match between `CREATE`'s tag ID and the ID of the load/store address that follows the `CREATE` instruction. To perform this check, the bounds check optimization client (ClientID=0), performs a lookup to the MMC to retrieve the tag ID associated with the load/store address. This ID is compared with the value stored in the PMT by the previous `CREATE` instruction. If there is a mismatch, then this indicates a buffer overflow or an access to data that is not part of the intended data structure as the load is accessing data that was not mapped to the same tag ID and using its interface to the OS, MetaSys terminates the program.

6.1.2 Methodology and Evaluation. We evaluate MetaSys-based bounds checking on our prototype with the parameters listed in Table 4 (tagging granularity is set to 64 B). We use the Olden [130] benchmarks (commonly used for bounds checking and stack protection research [43, 50, 111, 146, 174] due to its focus on pointer-based data structures). We only compare against a prior software solution [177] as custom hardware solutions for bounds checking require intrusive and significant changes to the microarchitecture, ISA, and application, which are difficult to reasonably implement on a full-system simulator or an FPGA. For example, implementing Hardbound [43], the closest custom hardware solution for bounds checking, requires compiler support, extending every register and word of memory with "sidecar" shadow registers for base and bound addresses,

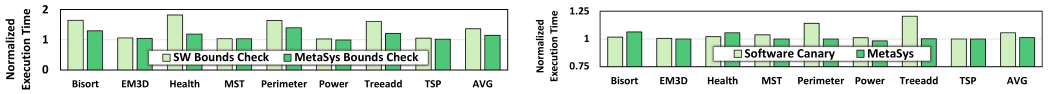

Fig. 4. Performance overheads for (left) bounds checking, (right) return address protection.

compression/decompression engines to compress these base/bound addresses, tag caches, and bounds checking logic.

We evaluate three designs: (i) the *Baseline* system without MetaSys; (ii) software bounds checking, based on prior work [177]; and (iii) MetaSys-based bounds checking. Figure 4 (left) depicts the execution time normalized to the Baseline. We observe that the software bounds checking design incurs a high average performance overhead of 36% (up to 82%). This overhead comes from executing more instructions to check bounds (64% on average). In contrast, MetaSys-based bounds checking incurs only an average performance overhead of 14% (up to 40%). MetaSys requires only a 32% increase in the number of executed instructions. Workloads such as `em3d`, `power`, and `mst`, are highly compute intensive and hence do not incur significant overheads with either bounds checking technique.

We conclude that MetaSys provides a lightweight substrate to implement and evaluate hardware-software cooperative bounds checking. MetaSys can be flexibly extended to implement more sophisticated memory protection techniques.

6.2 Return Address Protection

The program's call stack is a known source of many security vulnerabilities in low-level, memory-unsafe languages such as C/C++. For example, the control flow in the program can be hijacked by overwriting the *return addresses* saved in the stack [8, 36]. Existing defenses such as ExecShield [160] and stack canaries [36] do not protect against sophisticated attack techniques [24, 28, 158]. Stack canary protection [36] is a software check that involves writing an additional randomly generated value in the stack and a duplicate is saved separately in memory. The stack canary checks the randomly generated value in the stack against its duplicate to detect stack overwriting before a function returns to the return address saved in the stack. Protecting return addresses with more powerful software checks [1, 16, 83, 97, 101] incurs significant runtime overheads and are hence difficult to use in practice [39, 156]. Prior work has proposed a range of hardware techniques [45, 63, 86, 129, 174, 186] to enable return address protection more efficiently. These approaches either require dedicated hardware support for stack protection (e.g., RAGuard [186], PAC-it-up [86], CET [63]) or more heavy-weight metadata systems for memory protection (e.g., SDMP [129], Cheri [174], PUMP [45]). In this section, we implement and evaluate return address protection with MetaSys's lightweight metadata support and cross-layer interfaces.

6.2.1 Return Address Protection with MetaSys. To enable return address protection with MetaSys, we first tag each return address using `MAP` as id="1." This is done automatically with compiler support and no programmer intervention is required. No `CREATE` instruction is used. In hardware, we add support to simply disallow writes to any address tagged with id="1." To do this, we implement a simple hardware optimization client, which is triggered by store instructions. For each store instruction, the client performs a lookup to the MMC to determine the tag ID associated with the address. If the tag="1," then the location is a return address and the store is not allowed to complete. Any store to a tagged memory address causes the hardware to invoke the OS to terminate the program. The application can then unmap the return address when it is retrieved again from the stack. This ensures that once a memory location within the stack has a return address saved, it cannot be overwritten via attacks that hijack control flow such as buffer overflow attacks.

6.2.2 Evaluation and Methodology. We evaluate MetaSys-based return address protection using our FPGA prototype with system parameters listed in Table 4 (the tagging granularity set to 64 B). We evaluate three designs using the Olden [130] benchmarks: (i) the *Baseline* system with no overheads; (ii) canary stack protection [36] in the GCC RISC-V compiler; and (iii) MetaSys-based return address protection.

Figure 4 (right) depicts the execution time normalized to the *Baseline*. We observe that the canary approach incurs a performance overhead of 5.5% (up to 20%), while MetaSys incurs a diminished overhead of 1.2% (up to 6.2%). The major overheads for the stack canaries come from executing extra instructions (5.5% on average) to perform software checks. The overheads for MetaSys are low due to the high MMC hit rate, which leads to few additional memory accesses. In addition to providing less overhead, MetaSys-based return address protection can also protect against more sophisticated attacks that exploit write-what-where gadgets [38] and, unlike canaries, are immune to information leaks [151]. Protecting additional memory locations beyond return addresses (e.g., function pointers) with software approaches would incur even higher instruction overhead. However, the observed MetaSys overhead would largely remain the same as it already involves checking each store. In addition, MetaSys-based return address protection utilizes the Metasys support and interfaces, whose cost is amortized across many use cases, without requiring specialized ISA and hardware support.

We conclude that MetaSys enables easy implementation and evaluation of lightweight memory protection mechanisms with low performance overhead.

7 OTHER USE CASES OF METASYS

We briefly discuss various other cross-layer techniques that can be implemented with MetaSys (but would be challenging to implement with prior approaches like XMem [164]).

Performance optimization techniques. MetaSys provides a low-overhead framework and a rich cross-layer interface to implement a diverse set of performance optimizations including cache management, prefetching, page placement in memory, approximation, data compression, DRAM cache management, and memory management in NUMA and NUCA systems [2, 22, 26, 40, 41, 58, 66, 96, 116, 117, 126, 136, 137, 159, 166, 179]. MetaSys can flexibly implement the range of cross-layer optimizations supported by XMem [164], and the Locality Descriptor [163]. MetaSys's *dynamic* interface for metadata communication enables even more powerful optimizations than XMem including memory optimizations for dynamic data structures such as graphs. We already demonstrate one performance optimization in Section 5.

Techniques to enforce cross-layer quality of service (QoS). MetaSys can be used to implement cross-layer techniques to enforce QoS requirements of applications in shared environments [48, 62, 76, 93, 106, 107, 152, 153]. MetaSys allows communicating an applications' QoS requirements to hardware components (e.g., the last-level cache, memory controllers) to enable optimizations for partitioning and allocating shared resources such as cache space and memory bandwidth.

Hardware support for debugging and monitoring. MetaSys can be used to implement cross-layer techniques for performance debugging and bug detection by providing efficient mechanisms to track memory access patterns using its memory tagging and metadata lookup support. This includes efficient detection of memory safety violations [123, 161] or concurrency bugs [88–91, 108, 188] such as data races, deadlocks, or atomicity violations.

Security and protection. MetaSys provides a substrate to implement low-overhead hardware techniques for security/protection: the tagged memory support can be used to implement protection for spatial memory safety [42, 127, 172, 183], cache timing side-channels [78] and stack protection [86, 129]. For example, using MetaSys, software can tag memory accesses as

security-critical or safe. Based on the metadata received for every access, MetaSys can activate/deactivate (for the specific access) the corresponding side-channel defense technique at runtime (e.g., protect from or undo speculation [17, 25, 74, 134, 178]). We already demonstrate two security techniques in Section 6.

Garbage collection. MetaSys offers an efficient mechanism to track dead memory regions, unreachable objects, or young objects in managed languages. MetaSys is hence a natural substrate to implement hardware-software cooperative approaches (such as prior work [69, 94, 95]) for garbage collection. For example, HAMM [69], a hardware-software cooperative technique for reference counting, tracks the number of references to any object in hardware. It has many of the same metadata management components as MetaSys. HAMM uses a multi-level metadata cache to manage the large amounts of metadata associated with reference counting for each object. MetaSys was designed with modular interfaces that enable adding more levels to the metadata cache for such use cases.

OS optimizations. MetaSys can be used to implement OS optimizations that require hardware performance monitoring of memory access patterns, contention, reuse, and so on [29, 47, 105, 118, 147, 148]. The metadata support in MetaSys can be used to implement this monitoring and then inform OS optimizations like thread scheduling, I/O scheduling, and page allocation/mapping [40, 56, 77, 102, 105].

Cache optimizations. MetaSys enables various cache optimizations such as cache scrubbing [136, 166] and cache prioritization [22, 26, 58, 66, 96, 116, 117, 126, 136, 137, 159, 166, 179]. To implement such optimizations with MetaSys, the `CREATE` operator is used to specify the expected *reuse* of a data object at runtime. For example, objects can be tagged as having no reuse (e.g., once all threads have completed operations on it). Thus, upon encountering a cache miss (the trigger event), the cache controller can look up the expected reuse of different cache lines using MetaSys's lookup mechanism and then evict the dead cache line. A similar mechanism can be used to retain cache lines that have high expected cache reuse.

Compressing sparse data structures. MetaSys can be used to support techniques that efficiently compress sparse data structures and accelerate sparse workloads [72, 138]. For example, SMASH [72] is a hardware-software cooperative technique that efficiently compresses sparse matrices using a hierarchy of bitmaps to encode non-zero cache lines and accelerates the discovery of the non-zero elements of the sparse matrix. Instead of using specialized hardware, SMASH could access the hierarchy of bitmaps and identify non-zero elements with MetaSys' metadata support.

Heterogeneous reliability memory optimizations. MetaSys' metadata support can be used by techniques that exploit heterogeneous reliability characteristics of memory devices to improve performance, power consumption, and system cost [81, 87, 92, 135]. These techniques typically require support for dynamically looking up the error tolerance characteristics of data structures to place them in memory to satisfy a target bit error rate. MetaSys' metadata support is a natural candidate for providing these techniques with a means to query reliability characteristics of data structures.

8 LIMITATIONS OF METASYS

Our goal of providing a low-overhead and general system largely tailored for cross-layer *performance* optimization leads to several major limitations in MetaSys. These limitations can be mitigated by future work.

Instruction and register tagging are not supported. MetaSys does not currently support tagging of instructions or registers and thus cannot easily support techniques such as taint-tracking [37, 149, 154] and security mechanisms that require rule-checking at the instruction/register level [44, 45].

Overheads of fine-granularity memory tagging. While MetaSys supports memory tagging at flexible granularity, the system is optimized for the *larger* granularities typically required for performance optimization (>=64 B) or fine granularities for only *some* data (e.g., return addresses). Byte/word granularity tagging for the entire program data may lead to high MMC miss rates and may thus incur higher overheads with MetaSys.

Limitations on using private metadata tables (PMTs) for runtime profiling. With MetaSys's existing interfaces, the PMTs cannot be used to collect program information and supply it back to the application. The PMTs can only be updated by the `CREATE` operator. This issue can be mitigated relatively easily in future MetaSys versions.

9 CHARACTERIZING GENERAL METADATA MANAGEMENT SYSTEMS FOR CROSS-LAYER OPTIMIZATIONS

Our goal in this section is to perform a detailed characterization of the overheads of using a *single* common metadata system and interface for multiple cross-layer techniques. Three major challenges and sources of system overhead include:

(1) *Handling dynamic metadata:* Communicating metadata at runtime requires execution of additional instructions in the program. This incurs performance overheads in the form of CPU processing cycles and data movement to communicate the metadata to hardware components or to save them in memory.
(2) *Efficient metadata management and lookups:* The communicated metadata must be saved in memory or specialized caches (the MMC in MetaSys) that overflow to memory. Different components in the system must then be able to efficiently look up the metadata for performance optimization. Storing and retrieving metadata may incur expensive memory accesses and consume memory bandwidth.
(3) *Scaling to multiple components:* A *general* cross-layer interface and metadata system must be able to serve multiple client components implementing different optimizations in the caches, the prefetchers, the memory controller, and so on. Multiple components accessing shared metadata support during program execution poses significant scalability challenges.

The above challenges may impose significant area and performance overheads in the CPU, making the feasibility of a common metadata system and interface (as opposed to per-use-case specialized interfaces and optimizations) for cross-layer techniques questionable. In this section, we set out to experimentally quantify these overheads, identify key bottlenecks, and discover and provide insights on how these challenges affect different workloads and how they can be alleviated.

9.1 Analysis

We perform our characterization using the Polybench [124] and Ligra [145] benchmark suites along with a set of microbenchmarks (available on Github [57]). Polybench contains building block kernels frequently used in linear algebra, scientific computation, and machine learning. Ligra contains widely used graph analytics workloads. The microbenchmarks are designed to *intensely* stress the MetaSys system and identify *worst-case* overheads.

- *Stream.* This memory-bandwidth-intensive microbenchmark streams through a large amount of data, accessing it only once. It hence has high spatial locality and no data reuse.
- *Linked List Traversal.* The microbenchmark mimics typical linked list creation, insertion, and traversal and emulates the widely seen memory-intensive *pointer-chasing* operation.
- *Random Access.* This microbenchmark accesses memory locations within a large array at random indices and is designed to test an extremely rare worst-case scenario: no pattern in accesses, no reuse, and no spatial locality.

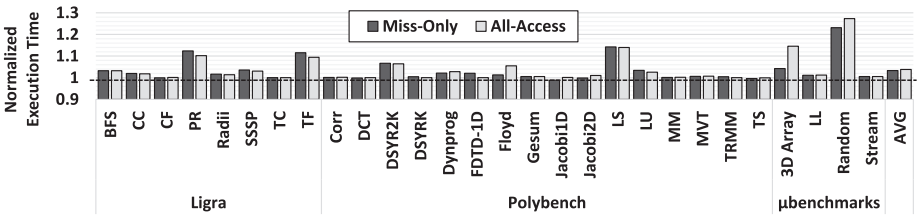

Fig. 5. Normalized performance overhead of MetaSys.

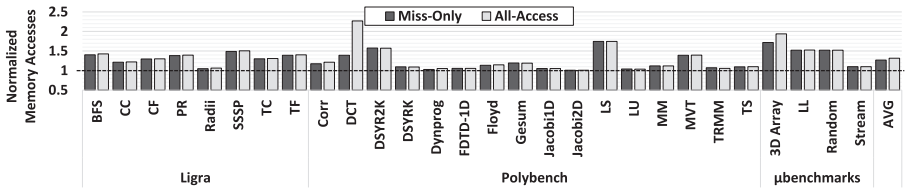

Fig. 6. Additional memory accesses introduced by MetaSys metadata lookups.

- *Three-dimensional array traversal (3D array).* This microbenchmark mimics the access pattern and locality seen in applications with multi-dimensional arrays. It traverses a 3D array first along the third dimension, and then along the second and first (data is contiguously placed in the first dimension). The access pattern is highly regular but exhibits no spatial locality.

Section 4 describes the parameters of our baseline system. We summarize our key findings in Section 9.3. In all evaluations in this section, since we aim to characterize the overheads of the system itself, we do *not* implement any cross-layer optimization that improves performance. We simply implement *lookups* to the metadata system that an optimization could potentially make. Since our goal is to stress the system and understand the worst-case overheads, we perform metadata lookups for *every memory access*. In typical use cases, the lookup triggers would be much less frequent than for every memory access, e.g., lookups on only cache misses for prefetching or only stores for return address protection.

9.1.1 Performance Overhead Analysis. The performance overheads in MetaSys come from two major sources: (i) dynamic instructions (`MAP` and `CREATE`) and (ii) metadata *lookups* when a component retrieves the tag ID associated with any memory address (from the MMT, cached in the MMC) and then the corresponding metadata (in the PMT).

Figure 5 depicts the execution time normalized to the baseline system (without MetaSys) for two scenarios: (i) when performing metadata lookups for *every* access to the L1 cache (*All-Accesses*) and (ii) when performing metadata lookups only on every L1 cache miss (*Miss-Only*). These studies were conducted with our baseline 128-entry MMC with a tagging granularity of 512 B (as in XMem [164]). Figure 6 plots the number of memory accesses, normalized to baseline (the additional memory accesses come from misses in the MMC). Figure 7 plots the corresponding MMC hit rates.

We make two major observations from the three figures. First, the overall performance overheads from the metadata management system for both designs are low in most workloads with an average performance overhead of 2.7% (ranging from ~0% up to 14%), excluding the microbenchmarks. The highest overheads observed in the microbenchmarks is 27% for `Random` and represents the worst-case overhead. Workloads with the highest overheads (`Random`, `GS`, `PR`, `TF`) are highly

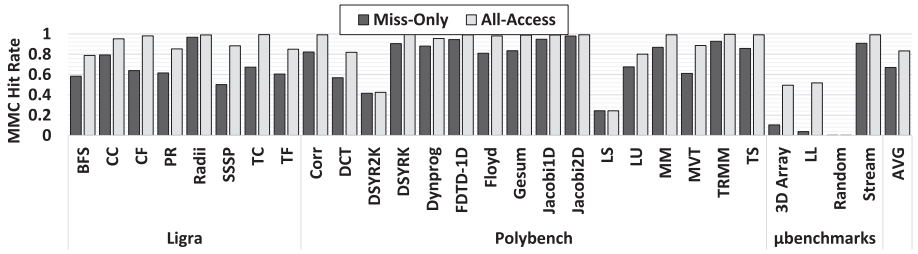

Fig. 7. Metadata Mapping Cache (MMC) hit rate.

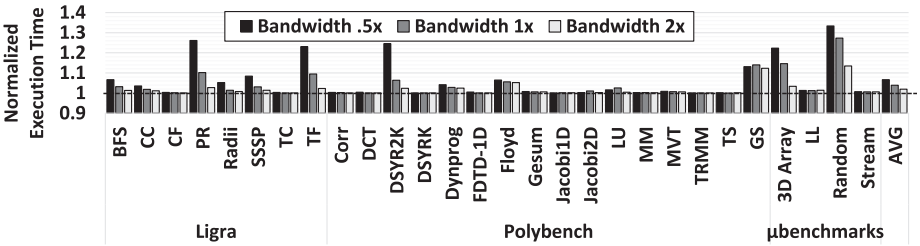

Fig. 8. MetaSys performance overhead on systems with varying amounts of memory bandwidth.

memory-intensive and have low spatial and temporal locality, which leads to low hit rates in the MMC (e.g., ~0% in Random and 24% in GS). This causes a significant increase in accesses to memory and thus higher performance overheads.

Second, the *number* of metadata lookups (not shown in the figure) does not have a direct impact on the overall performance overhead. *All-access* performs on average 75.2% more lookups than *Miss-only*, but incurs an additional overhead of only 0.05%. *Miss-only* has lower MMC hit rates due to lower locality in lookups than *All-access*. Thus, the number of *additional* memory accesses is largely the same for both designs, as shown in Figure 6.

Since the major overheads are from additional memory accesses, we evaluate the impact of available memory bandwidth. Figure 8 depicts the performance overhead of *All-access* on two different systems with 0.5× and 2× the memory bandwidth of the baseline system. We observe that, except for GS, more memory bandwidth significantly reduces any lookup overheads: the average overhead is only 0.5% on a system with 2× the memory bandwidth. Conversely, in workloads with higher MMC miss rates (e.g., DSYR2K), performance overheads increase with a reduction in available memory bandwidth.

We conclude that (i) performance overheads are correlated to the MMC hit rates; (ii) metadata lookup hardware can be frequently queried with no direct observable impact on performance; and (iii) overall performance overheads are small when the MMC provides high hit rates.

9.1.2 Effect of the Metadata Mapping Cache (MMC).. Figure 9 shows the impact of the size of the MMC on performance overhead. We evaluate six sizes for *All-access* and Figure 9 presents the resulting execution time (normalized to the baseline system). We make two observations. First, in most workloads, 128 entries is sufficient to obtain small overheads. This is because with a 512 B tagging granularity, we can hold tag IDs for 64 KB of memory in the MMC (compared to 16 KB of L1 cache space). Second, workloads with poor spatial and temporal locality (e.g., Random, GS), are largely insensitive to the MMC sizes evaluated. Thus, overheads in such workloads cannot be easily addressed by increasing the MMC size.

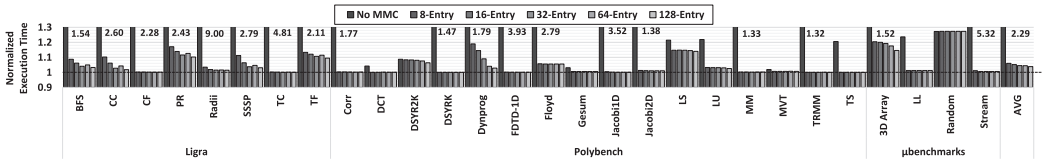

Fig. 9. Impact of Metadata Mapping Cache (MMC) size on MetaSys performance overhead.

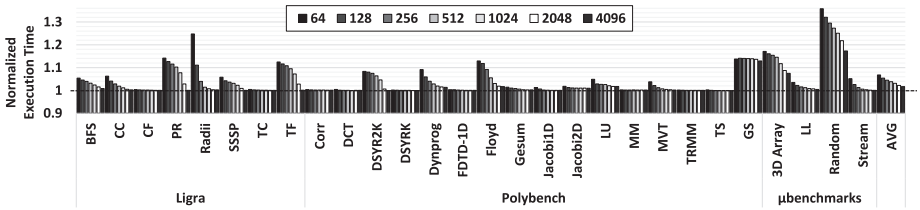

Fig. 10. Impact of tagging granularity on MetaSys performance overhead.

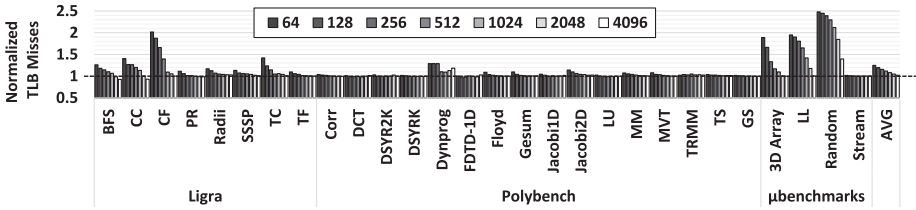

Fig. 11. Impact of tagging granularity on additional TLB misses.

9.1.3 Effect of Metadata Granularity. The *granularity* at which memory is tagged plays a critical role in determining the reach of the MMC. Tagging memory at smaller granularities requires more MMC entries to tag the same amount of memory, but it enables more optimizations (e.g., bounds checking). Figure 10 presents execution time for different granularities of tagging, normalized to the baseline system without MetaSys. For most workloads, even the smallest granularity we evaluated (64 B) has a small impact on performance. Large granularities reduce overheads for all but Random and GS by significantly increasing the MMC hit rate. A secondary effect in irregular workloads, such as PR and SSSP, is that small granularities increase the number of *TLB* misses (by 11% and 13%, respectively), as depicted in Figure 11. The additional MMC misses cause accesses to the MMT in memory, which requires address translation.

To evaluate the effect of the TLB, we implement a MetaSys design that does *not* require address translation to access the MMT (i.e., MMT entries are accessed directly using physical addresses). Figure 12 presents the resulting normalized execution time without address translation. We observe a decrease in overhead with this design in the irregular workloads: BFS, CC, Random, and LL (by 1.9%, 1.8%, 14%, and 1%, respectively).

9.1.4 Effect of Contention for Metadata. To evaluate the *scalability* of MetaSys with multiple clients accessing the same metadata support, we evaluate the overheads of two clients performing frequent metadata lookups: one client on every *memory access* (with the corresponding memory address) and another on every *TLB miss* (with the page table entry address). Since each design performs lookups with *different* memory addresses, they do not share entries in the MMC and this creates a difficult scenario for the shared MMC.

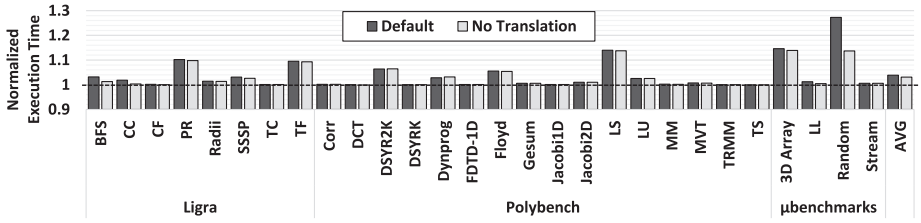

Fig. 12. Performance overhead with no address translation overhead for metadata.

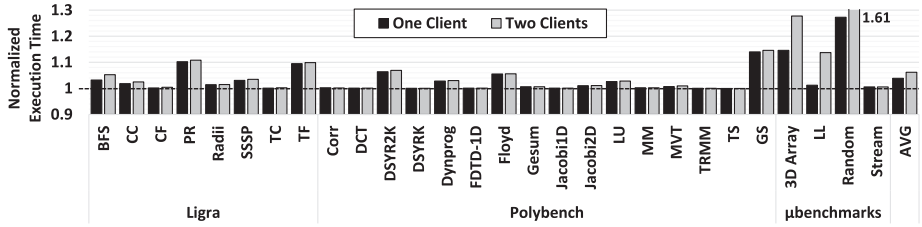

Fig. 13. Performance overhead with multiple clients.

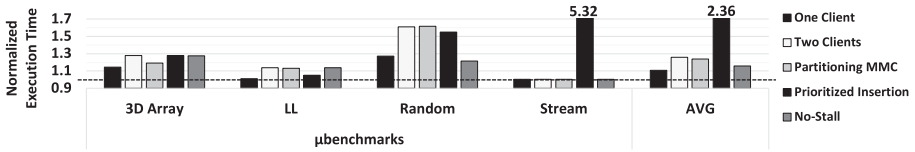

Fig. 14. Alleviating MMC contention in microbenchmarks.

Figure 13 depicts the resulting execution time for two designs, normalized to the baseline system: (i) One Client performs metadata lookups on every memory access and (ii) Two Clients performs metadata lookups on every memory access as well as on every TLB miss. We observe that for all workloads except the microbenchmarks, increasing the number of clients leads to a small additional performance overhead (on average 0.3%). This is because the MMC can sufficiently capture the tag ID working set for both clients. The microbenchmarks designed to stress the system experience a significant additional performance overhead (up to 34% for Random) as a result of more misses in the MMC due to more clients.

To investigate mechanisms to alleviate the MMC contention overheads seen in the microbenchmarks, we evaluate three designs in Figure 14: (i) *Partitioning* the MMC equally between the two clients; (ii) *Prioritized Insertion*, where we insert mappings for the client with better locality at a higher priority in the MMC (such that such mappings are evicted last); and (iii) *No stall*, where we do not stall the core on an MMC miss (instead, the optimization performed by the client is delayed). We observe that *Partitioning* reduces the overhead for `3D Array` and `LL` by 9% and 4% by avoiding cache thrashing. *Prioritized Insertion* helps reduce the overheads in `LL` (by 8.5%) and `Random` (by 6%), where one client has more locality than the other in lookups. *No Stall* significantly reduces the overhead in `Random` (by 40%) by mitigating the latency overhead of additional memory accesses.

9.1.5 Evaluating Instruction Overheads. To evaluate the instruction overheads of the dynamic `MAP/CREATE` instructions, we present an analysis in Figure 15, where we intensively use these instructions: (i) for every eight memory instructions, we add one `MAP` and one `CREATE` instructions

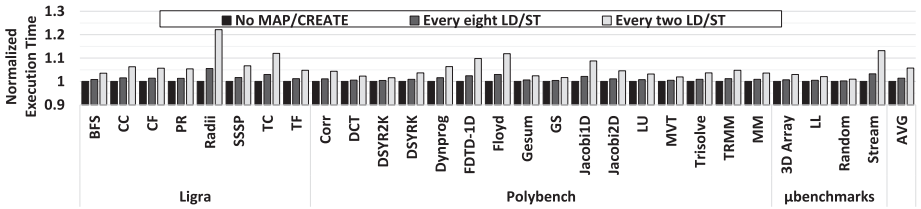

Fig. 15. Performance overhead of MAP/CREATE instructions.

and (ii) for every two memory instructions, we add one `MAP` and one `CREATE` instruction. We observe an average slowdown of 1.4% and 5.7%, respectively, over a baseline without MetaSys. This indicates that while excessive use of `MAP`/`CREATE` instructions can lead to slowdowns, the primary overheads are still from the metadata lookups, which may lead to additional memory accesses.

We conclude that MetaSys' metadata support is scalable to multiple components with small impact on performance overheads (except in microbenchmarks). The overheads seen in microbenchmarks are a result of poor MMC hit rates that can be mitigated via techniques such as partitioning, prioritized insertion, and by not stalling the core on an MMC miss. Since optimizations are triggered by loads/stores in MetaSys, MetaSys can be expected to gracefully scale to more than two clients as most clients are expected to query the MMC using the same addresses, which are aggregated and thus would not lead to additional lookups.

9.2 Hardware Area Overhead

We synthesized the baseline MetaSys system using the Synopsys DC [155] at 22 nm process technology to estimate the area overhead. MetaSys incurs small area overhead: 0.03 mm^2 (0.02% of a 22 nm Intel Ivy Bridge CPU Core [144]).

9.3 Summary of Findings

(1) Despite stressing the metadata support, the overall performance overheads of MetaSys are very low (2.7% on average, excluding the microbenchmarks). This indicates that using metadata systems that are *general* enough to support a range of use cases is a promising approach to enabling cross-layer performance optimizations in a general-purpose manner in real-world applications. The higher overheads seen in microbenchmarks indicate that the worst-case overheads are however substantially higher (up to 27%).

(2) Our studies indicate that MetaSys' metadata management is *scalable* to support multiple client components that have high rates of metadata access requirements. Performance overhead is dependent on locality of metadata accesses as opposed to number of metadata accesses, indicating that the same system can support multiple cross-layer optimizations at the same time. We propose simple techniques to alleviate metadata contention generated by multiple clients.

(3) The most critical factor that impacts MetaSys' performance overhead is the effectiveness of the **Metadata Mapping Cache (MMC)**. Workloads with low locality in metadata lookups incur performance overheads from additional memory accesses. The *reach* of the MMC is also affected by the *granularity* at which memory is tagged and hence the MMC hit rate can be improved with larger granularities. Thus, efficient caching of metadata tags is critical.

(4) Accesses to metadata mappings in memory require address translation and cause TLB misses, leading to high performance overhead especially in irregular workloads (e.g., Random). We find that this overhead can be mitigated by using physical addresses to access metadata

mappings or by storing address translations required to access metadata mappings in a separate TLB.

10 CONCLUSION

This work introduces MetaSys, the first open-source full-system FPGA-based infrastructure to rapidly implement and evaluate diverse cross-layer optimizations in real hardware. We demonstrate MetaSys's versatility and ease-of-use by implementing and evaluating three new cross-layer techniques. We believe and hope MetaSys can enable new ideas and their rigorous evaluation on real hardware.

Using MetaSys, we present the first detailed experimental characterization to evaluate the efficiency and practicality of a single metadata system for cross-layer performance optimization. We demonstrate that the associated performance and area overheads are small, identify key performance bottlenecks, and propose simple techniques to alleviate them. Our characterization thus indicates that a *general* hardware-software interface with lightweight metadata management support offers a promising approach toward enabling general-purpose cross-layer techniques in CPUs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of TACO, MICRO 2020, ISCA 2020/2021, and SIGMETRICS 2020 for feedback. We thank the SAFARI group members for feedback and the stimulating intellectual environment they provide.

REFERENCES

[1] Martín Abadi et al. 2005. Control-flow integrity. In *Proceedings of the CCS*.
[2] Neha Agarwal et al. 2015. Unlocking bandwidth for GPUs in CC-NUMA systems. In *Proceedings of the HPCA*.
[3] Neha Agarwal et al. 2015. Page placement strategies for GPUs within heterogeneous memory systems. In *Proceedings of the ASPLOS*.
[4] Junwhan Ahn et al. 2015 A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the ISCA*.
[5] Sam Ainsworth et al. 2016. Graph prefetching using data structure knowledge. In *Proceedings of the ICS*.
[6] Sam Ainsworth et al. 2018 An event-triggered programmable prefetcher for irregular workloads. In *Proceedings of the ASPLOS*.
[7] Hassan Al-Sukhni et al. 2003. Compiler-directed content-aware prefetching for dynamic data structures. In *Proceedings of the PACT*.
[8] Aleph One. 1996. Smashing The Stack For Fun And Profit. Retrieved from https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf.
[9] Murali Annavaram et al. 2001. Data prefetching by dependence graph precomputation. In *Proceedings of the ISCA*.
[10] Krste Asanović et al. 2016. The Rocket Chip Generator. Technical Report: UCB/EECS-2016-17.
[11] Todd M. Austin et al. 1994. Efficient detection of all pointer and array access errors. In *Proceedings of the PLDI*.
[12] AVNET. 2021. Zynq-7000 Zedboard. Retrieved from http://zedboard.org/product/zedboard.
[13] Jonathan Bachrach et al. 2012. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the DAC*.
[14] Jean-Loup Baer et al. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the SC*.
[15] Mohammad Bakhshalipour et al. 2018. Domino temporal data prefetcher. In *Proceedings of the HPCA*.
[16] Arash Baratloo et al. 2000. Transparent run-time defense against stack smashing attacks. In *Proceedings of the ATEC*.
[17] Kristin Barber et al. 2019. SpecShield: Shielding speculative data from microarchitectural covert channels. In *Proceedings of the PACT*.
[18] Abanti Basak et al. 2019. Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proceedings of the HPCA*.
[19] Michael Bekerman et al. 1999. Correlated load-address predictors. In *Proceedings of the ISCA*.
[20] Rahul Bera et al. 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *Proceedings of the MICRO*.
[21] Rahul Bera et al. 2019. DSPatch: Dual spatial pattern prefetcher. In *Proceedings of the MICRO*.

[22] Kristof Beyls et al. 2005. Generating cache hints for improved program efficiency. In *Proceedings of the JSA*.
[23] Eshan Bhatia et al. 2019. Perceptron-based prefetch filtering. In *Proceedings of the ISCA*.
[24] Tyler Bletsch et al. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the ASIA CCS*.
[25] Thomas Bourgeat et al. 2009. MI6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the MICRO*.
[26] Jacob Brock et al. 2013. Pacman: Program-assisted cache management. In *Proceedings of the ISMM*.
[27] Nicholas P. Carter et al. 1994. Hardware support for fast capability-based addressing. In *Proceedings of the ASPLOS*.
[28] Stephen Checkoway et al. 2010. Return-oriented programming without returns. In *Proceedings of the CCS*.
[29] Tien-Fu Chen et al. 1995. Effective hardware-based data prefetching for high-performance processors. In *Proceedings of the TC*.
[30] Chi-Keung Luk et al. 1998. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the MICRO*.
[31] Trishul M. Chilimbi et al. 2002. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the PLDI*.
[32] Tzicker Chiueh. 1994. Sunder: A programmable hardware prefetch architecture for numerical loops. In *Proceedings of the SC*.
[33] Yuan Chou. 2007. Low-cost epoch-based correlation prefetching for commercial applications. In *Proceedings of the MICRO*.
[34] Jeremy Condit et al. 2007. Dependent types for low-level programming. In *Proceedings of the ESOP*.
[35] Robert Cooksey et al. 2002. A stateless, content-directed data prefetching mechanism. In *Proceedings of the ASPLOS*.
[36] Crispin Cowan et al. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security*.
[37] Jedidiah R. Crandall et al. 2006. Minos: Architectural support for protecting control data. In *Proceedings of the TACO*.
[38] CWE MITRE. 2019. CWE-123: Write-what-where Condition. Retrieved from https://cwe.mitre.org/data/definitions/123.html.
[39] Thurston H. Y. Dang et al. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the ASIA CCS*.
[40] Reetuparna Das et al. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Proceedings of the HPCA*.
[41] Mohammad Dashti et al. 2013. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the ASPLOS*.
[42] Daniel Y. Deng et al. 2012. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Proceedings of the DSN*.
[43] Joe Devietti et al. 2008. Hardbound: Architectural support for spatial safety of the C programming language. In *Proceedings of the ASPLOS*.
[44] Udit Dhawan et al. 2015. Architectural support for software-defined metadata processing. In *Proceedings of the ASPLOS*.
[45] Udit Dhawan et al. 2014. PUMP: A programmable unit for metadata processing. In *Proceedings of the HASP*.
[46] Dinakar Dhurjati et al. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the ICSE*.
[47] Subramanya R. Dulloor et al. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the EuroSys*.
[48] Eiman Ebrahimi et al. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the ASPLOS*.
[49] Eiman Ebrahimi et al. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proceedings of the HPCA*.
[50] Archibald Samuel Elliott et al. 2018. Checked C: Making C safe by extension. In *Proceedings of the SecDev*.
[51] Michael Ferdman et al. 2007. Last-touch correlated data streaming. In *Proceedings of the ISPASS*.
[52] Michael Ferdman et al. 2009. Spatial memory streaming with rotated patterns. In *Proceedings of the 1st JILP Data Prefetching Championship*.
[53] E. A. Feustel. 1973. On the advantages of tagged architecture. In *Proceedings of the TC*.
[54] John W. C. Fu et al. 1991. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the ISCA*.
[55] John W. C. Fu et al. 1992. Stride directed prefetching in scalar processors. In *Proceedings of the MICRO*.
[56] Mel Gorman. 2007. Physical Page Allocation. Retrieved from https://www.kernel.org/doc/gorman/html/understand/understand009.html.
[57] SAFARI Research Group. 2021. MetaSys. Retrieved from https://github.com/CMU-SAFARI/MetaSys.
[58] Xiaoming Gu et al. 2008. P-OPT: Program-directed optimal cache management. In *Proceedings of the LCPC*.

[59] Milad Hashemi et al. 2018. Learning memory access patterns. In *Proceedings of the ICML*.
[60] Reed Hastings et al. 1991. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX*.
[61] Zhigang Hu et al. 2003. TCP: Tag correlating prefetchers. In *Proceedings of the HPCA*.
[62] Bowen Huang et al. 2017. Labeled RISC-V: A new perspective on software-defined architecture. In *Proceedings of the CARVV*.
[63] Intel. 2019. Control-flow Enforcement Technology Specification. Retrieved from www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf
[64] Yasuo Ishii et al. 2009. Access map pattern matching for data cache prefetch. In *Proceedings of the ISC*.
[65] Akanksha Jain et al. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the MICRO*.
[66] Prabhat Jain et al. 2001. Software-assisted cache replacement mechanisms for embedded systems. In *Proceedings of the ICCAD*.
[67] Trevor Jim et al. 2002. Cyclone: A safe dialect of C. In *Proceedings of the ATEC*.
[68] A. Joannou et al. 2017. Efficient tagged memory. In *Proceedings of the ICCD*.
[69] José A. Joao et al. 2009. Flexible reference-counting-based hardware acceleration for garbage collection. In *Proceedings of the ISCA*.
[70] Doug Joseph et al. 1997. Prefetching using Markov predictors. In *Proceedings of the ISCA*.
[71] Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the ISCA*.
[72] Konstantinos Kanellopoulos et al. 2019. SMASH: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the MICRO*.
[73] Magnus Karlsson et al. 2000. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the HPCA*.
[74] Khaled N. Khasawneh et al. 2019. SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation. In *Proceedings of the DAC*.
[75] Jinchun Kim et al. 2016. Path confidence based lookahead prefetching. In *Proceedings of the MICRO*.
[76] Yoongu Kim et al. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the MICRO*.
[77] Kenneth C. Knowlton. 1965. A fast storage allocator. In *Proceedings of the CACM*.
[78] Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the CRYPTO*.
[79] Clemens Kolbitsch et al. 2011. Extending Mondrian Memory Protection. Retrieved from https://www.sto.nato.int/publications/STO%20Meeting%20Proceedings/RTO-MP-IST-091/MP-IST-091-10.pdf.
[80] Sushant Kondguli et al. 2018. Division of labor: A more effective approach to prefetching. In *Proceedings of the ISCA*.
[81] Skanda Koppula et al. 2019. EDEN: Enabling energy-efficient, high-performance deep neural network inference using approximate DRAM. In *Proceedings of the MICRO*.
[82] Sanjeev Kumar et al. 1998. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the ISCA*.
[83] Volodymyr Kuznetsov et al. 2014. Code-pointer integrity. In *Proceedings of the OSDI*.
[84] Albert Kwon et al. 2103. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the CCS*.
[85] Henry M. Levy. 1984. *Capability-based Computer Systems*.
[86] Hans Liljestrand et al. 2018. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the USENIX Security*.
[87] Song Liu et al. 2011. Flikker: Saving DRAM refresh-power through critical data partitioning. In *Proceedings of the ASPLOS*.
[88] Shan Lu et al. 2006. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the ASPLOS*.
[89] Brandon Lucia et al. 2010. ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the ISCA*.
[90] Brandon Lucia et al. 2010. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the ISCA*.
[91] Brandon Lucia et al. 2008. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the ISCA*.
[92] Yixin Luo et al. 2014. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *Proceedings of the DSN*.

[93] Jiuyue Ma et al. 2015. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *Proceedings of the ASPLOS*.
[94] Martin Maas et al. 2016. Grail quest: A new proposal for hardware-assisted garbage collection. In *Proceedings of the Workshop on Architectures and Systems for Big Data*.
[95] Martin Maas et al. 2018. A hardware accelerator for tracing garbage collection. In *Proceedings of the ISCA*.
[96] M. Manivannan et al. 2016. RADAR: Runtime-assisted dead region management for last-level caches. In *Proceedings of the HPCA*.
[97] Ali Jose Mashtizadeh et al. 2015. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the CCS*.
[98] Arjun Menon et al. 2017. Shakti-T: A RISC-V processor with light weight security extensions. In *Proceedings of the HASP*.
[99] Pierre Michaud. 2016. Best-offset hardware prefetching. In *Proceedings of the HPCA*.
[100] Daniele Midi et al. 2017. Memory safety for embedded devices with nesCheck. In *Proceedings of the ASIA CCS*.
[101] Vishwath Mohan et al. 2015. Opaque control-flow integrity. In *Proceedings of the NDSS*.
[102] Ingo Molnar. 2007. Modular Scheduler Core and Completely Fair Scheduler. Retrieved from https://web.archive.org/web/20070419102054/http://kerneltrap.org/node/8059.
[103] Anurag Mukkara et al. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the MICRO*.
[104] Anurag Mukkara et al. 2016. Whirlpool: Improving dynamic cache management with static data classification. In *Proceedings of the ASPLOS*.
[105] Muralidhara et al. 2011. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the MICRO*.
[106] Onur Mutlu et al. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the MICRO*.
[107] Onur Mutlu et al. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the ISCA*.
[108] A. Muzahid et al. 2010. AtomTracker: A comprehensive approach to atomic region inference and violation detection. In *Proceedings of the MICRO*.
[109] Santosh Nagarakatte et al. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the ISCA*.
[110] Santosh Nagarakatte et al. 2014. WatchdogLite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of the CGO*.
[111] Santosh Nagarakatte et al. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the PLDI*.
[112] George C. Necula et al. 2005. CCured: Type-safe retrofitting of legacy software. In *Proceedings of the TOPLAS*.
[113] Karthik Nilakant et al. 2014. PrefEdge: SSD prefetcher for large-scale graph traversal. In *Proceedings of the SYSTOR*.
[114] Oleksii Oleksenko et al. 2018. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack. In *Proceedings of the SIGMETRICS*.
[115] Samuel Pakalapati et al. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *Proceedings of the ISCA*.
[116] Abhisek Pan et al. 2015. Runtime-driven shared last-level cache management for task-parallel programs. In *Proceedings of the SC*.
[117] Vassilis Papaefstathiou et al. 2013. Prefetching and cache management using task lifetimes. In *Proceedings of the ICS*.
[118] Heekwon Park et al. 2013. Regularities considered harmful: Forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems. In *Proceedings of the ASPLOS*.
[119] Harish Patil et al. 1995. Efficient run-time monitoring using shadow processing. In *Proceedings of the AADEBUG*.
[120] Leeor Peled et al. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the ISCA*.
[121] L. Peled et al. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the ISCA*.
[122] Leeor Peled et al. 2018. A Neural Network Memory Prefetcher Using Semantic Locality. Retrieved from https://arXiv:1804.00478.
[123] Pin Zhou et al. 2004. iWatcher: Efficient architectural support for software debugging. In *Proceedings of the ISCA*.
[124] Louis Pouchet. 2015. Polybench: The polyhedral benchmark suite. Retrieved from http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.
[125] Seth H. Pugsley et al. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *Proceedings of the HPCA*.
[126] Rajiv Ravindran et al. 2007. Compiler-managed partitioned data caches for low power. In *Proceedings of the LCTES*.

[127] Steven K. Reinhardt et al. 1994. Tempest and typhoon: User-level shared memory. In *Proceedings of the ISCA*.
[128] RISC-V. 2019. RISC-V Proxy Kernel. Retrieved from https://github.com/riscv/riscv-pk.
[129] Nick Roessler et al. 2018. Protecting the stack with metadata policies and tagged hardware. In *Proceedings of the SP*.
[130] Anne Rogers et al. 1995. Supporting dynamic data structures on distributed-memory machines. In *Proceedings of the TOPLAS*.
[131] Amir Roth et al. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the ASPLOS*.
[132] Amir Roth et al. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the ISCA*.
[133] Olatunji Ruwase et al. 2004. A practical dynamic buffer overflow detector. In *Proceedings of the NDSS*.
[134] Gururaj Saileshwar et al. 2019. CleanupSpec: An "Undo" approach to safe speculation. In *Proceedings of the MICRO*.
[135] Adrian Sampson et al. 2014. Approximate storage in solid-state memories. In *Proceedings of the TOCS*.
[136] Jennifer B. Sartor et al. 2014. Cooperative cache scrubbing. In *Proceedings of the PACT*.
[137] Jennifer B. Sartor et al. 2005. Cooperative caching with keep-me and evict-me. In *Proceedings of the INTERACT*.
[138] Vivek Seshadri et al. 2015. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *Proceedings of the ISCA*.
[139] Mehran Shakerinava et al. 2019. Multi-Lookahead Offset Prefetching. In *Proceedings of the 3rd Data Prefetching Championship*.
[140] Manjunath Shevgoor et al. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the MICRO*.
[141] Zhan Shi et al. 2019. A neural hierarchical sequence model for irregular data prefetching. In *Proceedings of the NeurIPS*.
[142] Zhan Shi et al. 2021. A hierarchical neural model of data prefetching. In *Proceedings of the ASPLOS*.
[143] Zhan Shi et al. 2019. Learning Execution Through Neural Code Fusion. Retrieved from https://arXiv:1906.07181.
[144] Anand Lal Shimpi. 2012. Dual Core/GT2 Ivy Bridge Die Measured: 121mm2. Retrieved from https://www.anandtech.com/show/5875/dual-coregt2-ivy-bridge-die-measured-121mm2.
[145] Julian Shun et al. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the PPoPP*.
[146] Matthew S. Simpson et al. 2010. MemSafe: Ensuring the spatial and temporal memory safety of C at runtime. In *Proceedings of the SCAM*.
[147] Stephen Somogyi et al. 2009. Spatio-temporal memory streaming. In *Proceedings of the ISCA*.
[148] Stephen Somogyi et al. 2006. Spatial memory streaming. In *Proceedings of the ISCA*.
[149] Chengyu Song et al. 2016. HDFI: Hardware-assisted data-flow isolation. In *Proceedings of the SP*.
[150] Santhosh Srinath et al. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the HPCA*.
[151] Raoul Strackx et al. 2009. Breaking the memory secrecy assumption. In *Proceedings of the EUROSEC*.
[152] Lavanya Subramanian et al. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the MICRO*.
[153] Lavanya Subramanian et al. 2013. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *Proceedings of the HPCA*.
[154] G. Edward Suh et al. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the ASPLOS*.
[155] Synopsys. 2021. Synopsys Design Compiler. Retrieved from https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.
[156] Laszlo Szekeres et al. 2013. SoK: Eternal war in memory. In *Proceedings of the SP*.
[157] Nishil Talati et al. 2021. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *Proceedings of the HPCA*.
[158] Minh Tran et al. 2011. On the expressiveness of return-into-libc attacks. In *Proceedings of the RAID*.
[159] Gary Tyson et al. 1995. A modified approach to data cache management. In *Proceedings of the MICRO*.
[160] Arjan van de Ven. 2004. New Security Enhancements in Red Hat Enterprise Linux: v.3 update 3. Retrieved from https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.
[161] Guru Venkataramani et al. 2007. MemTracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the HPCA*.
[162] Nandita Vijaykumar. 2019. *Enhancing Programmability, Portability, and Performance with Rich Cross-layer Abstractions*. Ph. D. Dissertation.
[163] Nandita Vijaykumar et al. 2018. The locality descriptor: A holistic cross-layer abstraction to express data locality in GPUs. In *Proceedings of the ISCA*.
[164] Nandita Vijaykumar et al. 2018. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *Proceedings of the ISCA*.

[165] Zhenlin Wang et al. 2004. Combining cooperative software/hardware prefetching and cache replacement. In *Proceedings of the IBM Austin CAS Center for Advanced Studies Conference*.
[166] Zhenlin Wang et al. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of the PACT*.
[167] Zhengrong Wang et al. 2019. Stream-based memory access specialization for general purpose processors. In *Proceedings of the ISCA*.
[168] Robert N. M. Watson et al. 2010. Capsicum: Practical capabilities for UNIX. In *Proceedings of the USENIX Security*.
[169] Thomas F. Wenisch et al. 2009. Practical off-chip meta-data for temporal memory streaming. In *Proceedings of the HPCA*.
[170] Thomas F. Wenisch et al. 2010. Making address-correlated prefetching practical. In *Proceedings of the IEEE Micro*.
[171] Thomas F. Wenisch et al. 2005. Temporal streaming of shared memory. In *Proceedings of the ISCA*.
[172] Adam Wiggins et al. 2003. Legba: Fast hardware support for fine-grained protection. In *Proceedings of the ASIA CCS*.
[173] Emmett Witchel et al. 2002. Mondrian memory protection. In *Proceedings of the ASPLOS*.
[174] Jonathan Woodruff et al. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the ISCA*.
[175] Hao Wu et al. 2019. Temporal prefetching without the off-chip metadata. In *Proceedings of the MICRO*.
[176] Hao Wu et al. 2019. Efficient metadata management for irregular data prefetching. In *Proceedings of the ISCA*.
[177] Wei Xu et al. 2004. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the SIGSOFT*.
[178] Mengjia Yan et al. 2018. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the MICRO*.
[179] Hongbo Yang et al. 2003. Compiler-assisted cache replacement: Problem formulation and performance evaluation. In *Proceedings of the LCPC*.
[180] Suan Hsi Yong et al. 2003. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the ESEC*.
[181] Xiangyao Yu et al. IMP: Indirect memory prefetcher. In *Proceedings of the MICRO*.
[182] Nickolai Zeldovich et al. 2008. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the OSDI*.
[183] Nickolai Zeldovich et al. 2008. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the OSDI*.
[184] Yuan Zeng et al. 2007. Long short-term memory based hardware prefetcher: A case study. In *Proceedings of the MEMSYS*.
[185] Dan Zhang et al. 2018. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. In *Proceedings of the ASPLOS*.
[186] Jun Zhang et al. 2017. RAGuard: A hardware based mechanism for backward-edge control-flow integrity. In *Proceedings of the CF*.
[187] Zhenlin Wang et al. 2003. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the ISCA*.
[188] Pin Zhou et al. 2007. HARD: Hardware-assisted lockset-based race detection. In *Proceedings of the HPCA*.

Received June 2021; revised September 2021; accepted December 2021