MetaSys: A Practical Open-Source Metadata Management System to Implement and Evaluate Cross-Layer Optimizations

Nandita Vijaykumar[⋆] Ataberk Olgun[§ ℵ] Konstantinos Kanellopoulos[§] Nisa Bostanci[§ ℵ]
Hasan Hassan[§] Mehrshad Lotfi[‡] Phillip B. Gibbons[†] Onur Mutlu[§]
[⋆]*University of Toronto* [†]*Carnegie Mellon University* [§]*ETH Zürich*
[ℵ]*TOBB ETÜ* [‡]*Max Plank Institute*

Abstract — **This paper introduces the first open-source FPGA-based infrastructure, MetaSys, with a prototype in a RISC-V core, 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 improve 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, OS, 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 for graph analytics; *(ii)* bounds checking in memory unsafe languages, and *(iii)* return address protection in stack frames; each technique only requiring ~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.

1. Introduction

Hardware-software cooperative techniques offer a powerful approach to improve the performance and efficiency of general-purpose processors. These techniques involve communicating key application information from the software to the architecture to enable more powerful optimizations and resource management in hardware. Recent research proposes many such cross-layer approaches for various purposes, e.g., performance, quality of service (QoS), memory protection, programmability, security. For example, Whirlpool [1] identifies and communicates regions of memory that have similar properties (i.e., data structures) in the program to the hardware, where it is used to more intelligently place data in a non-uniform cache architecture (NUCA) system. RADAR [2] and EvictMe [3] communicate which data 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, etc. [2–14], program annotations/directives to convey program semantics [1, 6, 15–17], or interfaces to communicate an application's QoS requirements for efficient partitioning and prioritization of shared hardware resources [18, 19].

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), the instruction-set architecture (ISA), and application software. Existing open-source infrastructure for implementing cross-layer techniques in real hardware include PARD [18, 19] for QoS and Cheri [20] 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, etc.

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 Rocket RISC-V core [21], 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 (metadata) to the hardware architecture at runtime. The metadata that can be communicated with this interface include memory access pattern information for prefetching, data reuse information for cache management, address bounds for hardware bounds checking, etc. 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 save 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 memory address location. (3) **Modularized components** to quickly implement various cross-layer optimizations with interfaces to the metadata management support, OS, core, and memory system.

Compared to the closest prior-work, XMem [22], MetaSys uses a similar tagged memory-based metadata management system. MetaSys however offers two key 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 §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 (§3.8). MetaSys incurs only a small area overhead of 0.02% (including 17KB of additional SRAM), 0.2% memory overhead in DRAM, and adds only 8 new instructions to the RISC-V ISA. Furthermore, XMem is not an open-source infrastructure and was not 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 §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 only requiring an additional ~100 lines of Chisel [23] code. In comparison, the hardware components of MetaSys required ~1800 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 the following new observations from our characterization across 24 applications and 4 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 the system 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. Finally, an important previously unidentified factor in performance overhead is the additional *TLB* misses from the required address translation when metadata is retrieved from memory.

Conclusions from characterization. From our detailed characterization and implemented use cases on real hardware, we make the following conclusions: 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 [18, 22] and *consolidating* common metadata management support; thus, making the adoption of new cross-layer techniques in future processors significantly easier. We demonstrate that a common framework can simultaneously and scalably support multiple 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.

This work makes the following major contributions.

• We introduce MetaSys, the first full-system open-source FPGA-based infrastructure, with a prototype in a RISC-V core, of a lightweight metadata management system with a rich hardware-software interface that can be used to implement a diverse set of cross-layer techniques. MetaSys provides 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.

• MetaSys comprises a new expressive hardware-software interface with a streamlined system design that enables a richer set of cross-layer optimizations than prior work.

• 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 and 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 [22, 24]. For QoS optimizations, metadata includes application priorities and prioritization rules for allocation of resources such as memory bandwidth and cache space [18, 19]. Memory safety optimizations may communicate base/bounds addresses of data structures [25, 26].

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 [22, 24], memory protection and security [20, 25], and QoS [18, 19].

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 and *(ii)* lightweight and *low-overhead* metadata management [22]. 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 [22] and the large body of work on tagged memory [25, 27–30] and

capability-based systems [20, 31–33]. We qualitatively compare against the closest work, XMem, in §3.8 and quanti
tively in §5. Unlike all above works, our goal is to provide
open-source framework to implement and evaluate cross-lay
approaches in real hardware and to perform a characterizati
of such metadata systems for performance optimizations.

Infrastructure for evaluating cross-layer techniqu Evaluating the overheads and feasibility of a newly-propos
cross-layer technique is non-trivial. Fully characterizing t
performance and area overheads either with a full-syste
cycle-accurate simulator or an FPGA implementation requi
implementing: *(i) Hardware support* to implement the mec
nism; *(ii) OS support* for OS-based cross-layer optimizatio
and to characterize the context-switch and system overhea
of saving and handling a process' metadata; and *(iii) Co
piler support and ISA modifications* to add and recognize n
instructions to communicate metadata.

Recent works propose general systems that are designed
enable cross-layer techniques for QoS (PARD [18, 19]) or fi
grained memory protection and security (Cheri [20]). PAR
enables tagging of components and applications with IDs t
are propagated with memory requests and enforce QoS requi
ments in hardware. Cheri [20] is a capability-based syste
that provides hardware support and ISA extensions to enab
fine-grained memory protection. Neither system supports t
(i) communication of diverse metadata at runtime, *(ii)* flexi
granularity tagging of memory to enable efficient metadata lookups from multiple components, or *(iii)* interfaces to numerous hardware components such as the prefetcher, caches, etc., 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 overheads performance, area, and power overheads, and thus their 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 (**Meta**data Management **Sys**tem for Cross-Layer Performance Optimization), 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 [27–30] implementation of metadata management in the system and OS; and *(iii)* flexible modules to add new hardware optimizations with interfaces to the metadata, core, memory, and OS. We build a prototype of MetaSys in the RISC-V Rocket [21] core.

We choose an FGPA implementation as opposed to a full-system simulator as: *(i)* This enables focus on feasibility as all components need to be fully implemented (e.g., ports, wires, buffers) and its 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 it a better

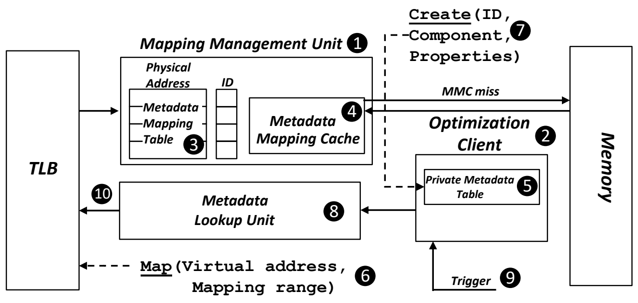


Figure 1: MetaSys hardware components and operation.

3.1. Tagged memory-based metadata management

Similar to prior systems for taint-tracking, security, and performance optimization, MetaSys implements *tagged memory*-based [27–30] metadata management. MetaSys associates metadata with memory address ranges of any size 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): ❸ in Fig. 1. This table is allocated by the OS for each process and is saved in memory. In MetaSys (similar to XMem [22] and Cheri [20]), 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) ❹. 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 §9.1, we evaluate the performance impact of the tagging granularity. The size of the MMT depends on the tagging granularity. For a 512B mapping granularity, the MMT requires 0.2% of physical memory (16MB in a 8GB system). The MMC holds 128 entries, where each entry stores a physical-address-to-tag mapping, and is 608B in size (8 bit entry and 30 bit tag).

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 bounds information. We refer to these stores as Private Metadata Tables (PMTs) ❺. The PMTs are saved near each component (private to each component) and are loaded/updated by MetaSys. The metadata (e.g., locality/"hotness") is encoded such that it can be directly interpreted by the component, e.g., prefetcher.

3.2. The Hardware-Software Interface

Communicating application information with MetaSys requires *(i)* associating memory address ranges with a tag or ID of configurable bit width (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 ❻ (similar to XMem [22]). MAP and UNMAP are implemented as new RISC-V instructions that are interpreted by the MetaSys hardware support 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 hardware in the MMT. We also implement 2D and 3D versions of MAP to efficiently map 2/3-dimensional address ranges in a multi-dimensional data structure with a single instruction.

To associate each ID with metadata, we provide the CREATE interface. CREATE❼ takes 3 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 512B of metadata. CREATE directly populates the PMT of the appropriate hardware component with the 512 bytes (or less) of metadata. 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.

Table 1 lists the new instructions along with their arguments.

Table 1: MetaSys instructions.

MetaSys Operator	MetaSys ISA Instructions
CREATE	CREATE ModuleID, 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 ;

3.3. Metadata Lookup ❽

Each optimization component is triggered by a hardware event ❾ (e.g., a cache miss). It then retrieves the physical address from the TLB ❿ 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 component uses the retrieved tag ID to obtain the appropriate metadata from the PMT. The optimization module is designed to flexibly implement a wide range of use cases and can be designed based on the optimization in question. This module has the required interfaces to the prefetcher, caches, memory controller, and TLBs to make this easier.

3.4. Operating System Support

We add OS support for metadata management in the RISC-V proxy kernel [34], which can be booted on our Rocket RISC-V prototype. This includes support to manage the MMT in memory and flush the PMTs during a context switch. 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. If the application changes metadata before a virtual address has a page mapping, the OS is forced to allocate a physical page to it. In addition, we also provide support to implement optimizations performed by the OS or with OS cooperation. 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. We describe one such use case in §6.

3.5. Coherence/Consistency of Metadata in Multicore

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 address to tag ID 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. 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, 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).

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

Table 2: The MetaSys software library.

3.7. Software Library

We develop a software library which can be included in user programs to facilitate the use of MetaSys primitives CREATE and MAP (summarized in 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 module (*ModuleID*) 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.

3.8. Comparison to the XMem [22] framework

MetaSys implements a tagged-memory-based system with a metadata cache similar to XMem [22]. MetaSys however has three major benefits. First, MetaSys enables communicating 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 in memory safety, protection, prefetching, etc., and enables communicating metadata that is input-data dependent and metadata that can only be known 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 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, on the other hand, does not allow control of hardware optimizations from the application. Table 3 summarizes the operators and compares to the corresponding operators in XMem. *Of the three evaluated use cases, only return address protection (§6.2) can be implemented with XMem.*

Table 3: Comparison between MetaSys and XMem's interfaces.

Operator	XMem [22]	MetaSys
CREATE	Compiler pragma to communicate static metadata at load time.	Selects a hardware optimization, dynamically associates metadata with an ID, and communicates both to hardware at runtime (implemented as new instructions).
(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).	Removed as the same functionality can now be done with CREATE.

3.9. FPGA-based infrastructure

We build a full system prototype of MetaSys on an FPGA with the Rocket [21] RISC-V core 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 [21] in the Rocket chip to implement the metadata management system. The RoCC is a customizable module that enables interfacing with the core and memory. The hardware support implemented in the 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 SRAM metadata caches (MMC and PMTs). We extended the RISC-V ISA with 8 instructions (6 for MAP/CREATE and 2 for OS operations). To implement all the hardware modules of MetaSys, we modified/added 1781 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.

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 Fig. 2) with a PMT and interfaces to the metadata lookup unit, to the core (to receive triggers), and templated interfaces to the cache controller, memory controller, etc. 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 (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 additional 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 prop-

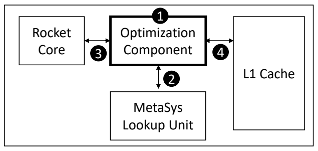


Figure 2: MetaSys Optimization Module.

4. Methodology

Baseline system. We use the in-order Rocket core [21] as the baseline CPU and conduct our experiments on the ZedBoard Zynq-7000 [35] FPGA board. Table 4 lists the parameters of the core and memory system. [1]

CPU: 25 MHz; in-order Rocket core [21]; **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; 38bits/entry; **Tagging Granularity:** 512B;
Private Metadata Table: 256 entries; 64B/entry; **DRAM:** 533MHz; V_{dd}: 1.5V;
Workloads: Ligra [36]: 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** [37]; μ**Benchmarks**

Table 4: Parameters of evaluated system.

[1] Since DRAM is disproportionately faster than the CPU clock rate possible on FPGAs, we added logic in the memory controller to proportionately 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.

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 [38–45], pointer-chasing [46–49], linear algebra computation [50] and other applications [51–54]. 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 works are challenging for traditional heuristic-based hardware prefetchers, e.g., stride [55] and stream [56], or even sophisticated prefetchers that rely on repeated patterns [57–59].

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 Fig. 3). For each vertex, the *vertex list* ❷ is indexed to retrieve the neighboring vertex IDs from the *edge list* ❸. To perform computation on the graph, the application then operates on the data 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.

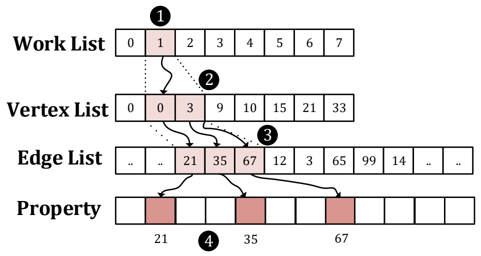


Figure 3: Data-dependent accesses in graph processing.

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 memory address range of each data structure with the appropriate tag.

We then design a hardware prefetcher that: *(i)* snoops every memory request from the core and retrieves the associated tag ID using MetaSys; *(ii)* queries the Private Metadata Table to retrieve the communicated metadata (listed above); and *(iii)* directly interprets the metadata to compute the data-dependent memory address of the data that is indexed by the current memory address content. In Fig. 3, when the prefetcher sees a memory request to 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 prefetcher can hence be flexibly programmed based on the specific properties associated with any data structure, algorithm, as well as the desired aggressiveness of prefetcher.

5.2 Evaluation and Methodology. We evaluate the MetaSys-based prefetcher using 8 graph workloads from the Ligra framework [36] using the Rocket-based prototype of MetaSys with the system parameters listed in Table 4. We evaluate three configurations: *(i)* the baseline system with a hardware stride prefetcher [60]; *(ii) GraphPref*, a customized hardware prefetcher that implements the same idea described above without the generalized MetaSys support; and *(iii)* the MetaSys-based graph prefetcher. Fig. 4 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 a similar prefetcher implemented as customized hardware, *GraphPref*, the MetaSys-based prefetcher performs almost as well: within 0.2% on average (only up to 0.8% for `BFS`).

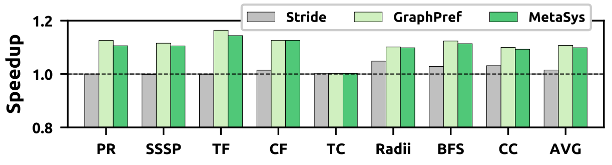


Figure 4: Performance speedup with the MetaSys prefetcher.

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/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 provided module for new optimization components.*

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 [61–75] to increase memory safety in the form of static or dynamic checks, such as CCured [63], Cyclone [67], and Softbound [69]. These approaches are however known to incur significant runtime overheads in performing numerous checks in software [76]. Hardware-based approaches offer a promising opportunity to alleviate these overheads. Prior work [20, 26, 77–82] including HardBound [26], ShaktiT [82] and Cheri [20] investigate enabling hardware-software cooperative bounds checking. These approaches however require architectures that are entirely specialized for bounds checking [26, 77, 82] or more heavyweight metadata management systems tailored for memory security and protection [20, 78–81]. In this section, we demonstrate how MetaSys can be used to implement hardware-based bounds checking in a lightweight and general metadata system.

6.1.1. Implementing bounds checking with MetaSys. To implement bounds checking with MetaSys, we use the `MAP` operator to tag each data structure to be protected with a unique ID. 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 then 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. The `CREATE` operator in this case only communicates the tag ID of the desired data structure. These instructions are added using software libraries for pointers. 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, we perform a lookup to the Metadata Management Cache to retrieve the associated tag ID. On a mismatch, 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 64B). We use the Olden [83] benchmarks (commonly used for bounds checking and stack protection research [20, 26, 69, 75, 84] due to its focus on pointer-based data structures) for both use cases.

We evaluate 3 designs: *(i)* the *Baseline* system without MetaSys; *(ii)* software bounds checking, based on prior work [61]; and *(iii)* MetaSys-based bounds checking. Fig. 5 depicts the execution time normalized to the Baseline. We observe that on average the software bounds checking design incurs a significantly high 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 only incurs a 14% overhead on average (up to 40%). MetaSys only requires 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. Existing defenses such as ExecShield [85] and

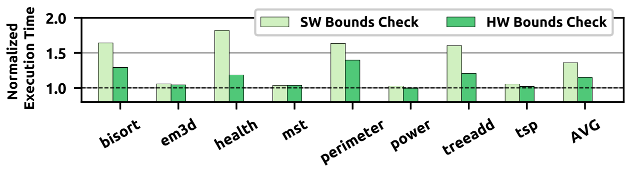


Figure 5: Performance overheads with bounds checking.

stack canaries [86] do not protect against sophisticated attack techniques [87–89]. Stack canary protection is a software check that involves writing an additional randomly generated value in the stack and a duplicate is saved separately in memory. This value in the stack is checked against the duplicate to detect stack overwriting by the canary code before returning to the saved return address. Protecting return addresses with more powerful software checks [90–94] incurs significant runtime overheads and are hence difficult to use in practice [76, 95]. Prior work has proposed a range of hardware techniques [20, 25, 96–99] to enable return address protection more efficiently. These approaches however require dedicated hardware support for stack protection (e.g., RAGuard [96], PAC-it-up [97], CET [99]) or more heavy-weight metadata systems for memory protection (e.g., SDMP [98], Cheri [20], PUMP [25]). 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". In hardware, we add support to simply disallow writes to any address tagged with id="1". 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. To check in hardware whether any memory address is tagged as "1" (i.e., contains a return address), we simply issue a lookup to the Metadata Mapping Cache. Any store to a tagged memory address causes the hardware to invoke the OS to terminate the program.

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 64B). We evaluate 3 designs using the Olden [83] benchmarks: *(i)* the *Baseline* system with no overheads; *(ii)* canary [86] stack protection in the GCC RISC-V compiler; and *(iii)* MetaSys-based return address protection.

Fig. 6 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 [100] gadgets and, unlike canaries, is immune to information leaks [101]. Protecting additional memory locations beyond return addresses (e.g., function pointers) with software approaches would incur even higher instruc-

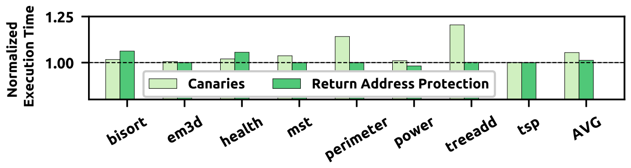


Figure 6: Performance overheads for return address protection.

tion overhead. However, the above MetaSys overheads would largely remain the same as it already involves checking each store.

We conclude that MetaSys enables easy implementation and evaluation of lightweight memory protection mechanisms using the provided interfaces and metadata management with low performance overhead.

7. Other Use Cases of MetaSys

We briefly discuss other cross-layer techniques that can be implemented with MetaSys (but would be challenging to implement with XMem).

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. MetaSys can flexibly implement the range of cross-layer optimizations supported by XMem [22]. MetaSys's *dynamic* interface for metadata communication enables even more powerful optimizations than XMem including memory optimizations for dynamic data structures such as graphs.

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 [18, 19, 102, 103]. 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 the cache 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 the memory tagging and metadata lookup support. This includes efficient detection of memory safety violations [81, 104] or concurrency bugs [105–110] 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 [80], cache timing side-channels [111] and stack protection [98]. 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/undo speculation [112–114]). We demonstrate two security techniques in §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 [115–117]) for garbage collection. For example, HAMM [115], a hardware-software 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, etc. The metadata support in MetaSys can used to implement this monitoring and then inform OS optimizations like thread scheduling, I/O scheduling, and page allocation/mapping.

8. Limitations of MetaSys

Our goal of providing a low-overhead and general system largely tailored for cross-layer *performance* optimization leads to two major limitations in MetaSys:

Instruction and register tagging. MetaSys does not currently support tagging of instructions or registers and thus cannot easily support techniques such as taint-tracking and security mechanisms that require rule-checking at the instruction/register level.

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 (>=64B) 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.

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, etc. Multiple components accessing shared metadata support during program execution poses significant scalability challenges.

All 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 systems) for cross-layer techniques questionable. In this section, we set out to experimentally quantify these overheads and identify the key bottlenecks and insights on how these challenges affect different workloads and how they can be alleviated.

9.1. Analysis

We perform our characterization using the Polybench [37] and Ligra [36] benchmark suites along with a set of microbenchmarks. 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 the worst-case (and a largely unrealistic) scenario—no pattern in accesses, no reuse, and no spatial locality.
• *3-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 with no spatial locality.

§4 describes the parameters of our baseline system. We summarize our key findings in §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 would make. Since our goal is to stress the system, we perform lookups for *every memory access*. In typical use cases, the lookup trigger would be less frequent, e.g., lookups on every cache miss for prefetching or on every store for return address protection.

9.1.1. Performance Overhead Analysis. The performance overheads in MetaSys come from two major sources: *(i)* the 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).

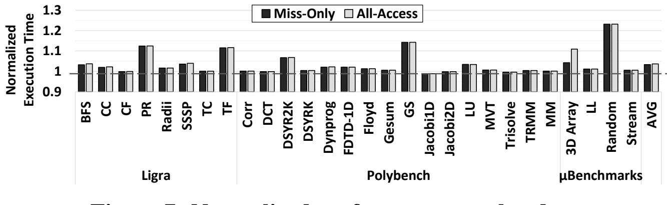


Figure 7: Normalized performance overheads.

Fig. 7 depicts the execution time normalized to the baseline system (without MetaSys) for two scenarios: *(i)* when

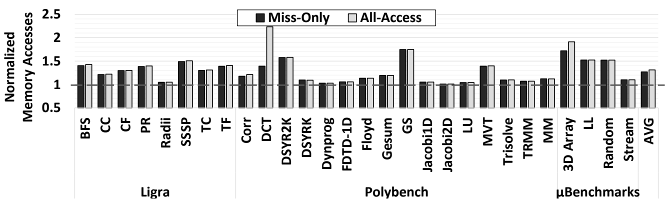
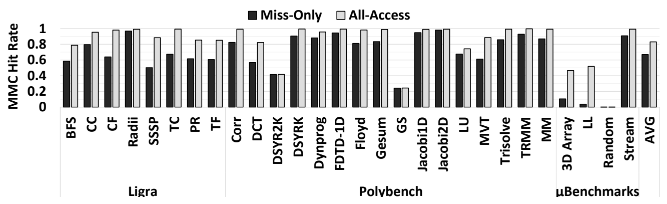


Figure 8: Additional memory accesses introduced by lookups.



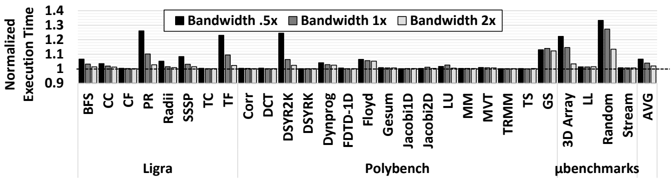


Figure 10: Impact of memory bandwidth on overhead.

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 512B (as in XMem [22]). Fig. 9 plots the corresponding MMC hit rates and Fig. 8 plots the number of memory accesses, normalized to baseline (the additional memory accesses come from misses in the MMC).

We make two major observations from the figure. First, the overall overheads from the metadata management system for both designs are low in most workloads with an average 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 absolute worst-case overhead. Workloads with the highest overheads (`Random`, `GS`, `PR`, `TF`) are highly 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 has a minimal 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.

Since the major overheads are from additional memory accesses, we evaluate the impact of available memory bandwidth. Fig. 10 depicts the performance overhead of *All-Access* with 0.5× and 2× the memory bandwidth as the baseline system. We observe that, except for `GS`, more memory bandwidth significantly reduces any lookup overheads: the average overhead is only 0.5% with 2× the bandwidth. Conversely, in workloads with higher MMC miss rates (e.g, `DSYR2K`), the overheads increase with a reduction in available memory bandwidth.

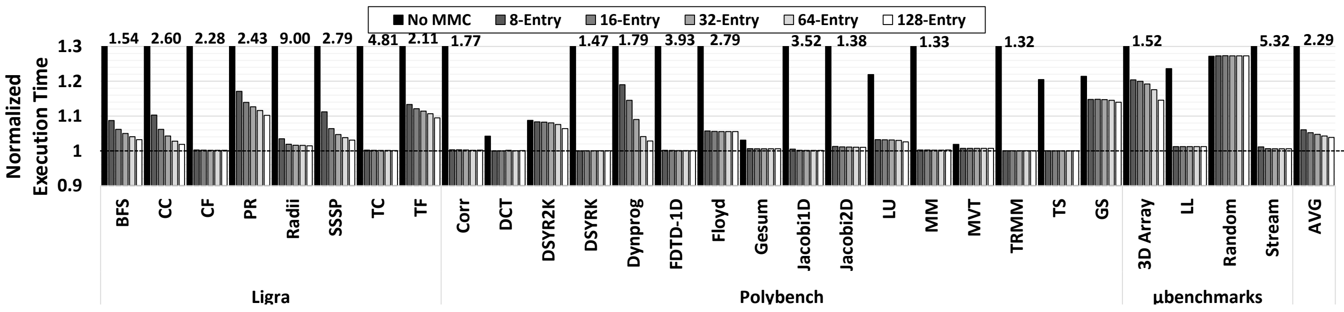


Figure 11: Impact of Metadata Mapping Cache (MMC) size on performance overhead.

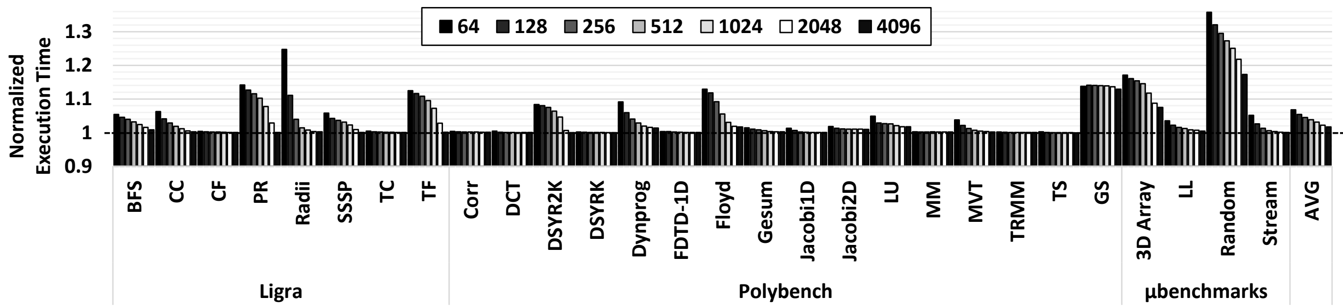


Figure 12: Impact of tagging granularity on performance overhead.

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

9.1.2. Effect of the Metadata Mapping Cache (MMC). In Fig. 11, we evaluate the impact of the size of the MMC on performance overhead. We evaluate 6 sizes for *All-Access* and Fig. 11 presents the resulting execution time (normalized to the baseline system). We make two observations. First, in most workloads 128 entries is sufficient to obtain minimal overheads. This is because with a 512B tagging granularity, we can hold tag IDs for 64KB of memory in the MMC (compared to 16KB of L1 cache space). Second, workloads with poor spatial and temporal locality (e.g., `Random`, `GS`), are largely insensitive to the MMC sizes we evaluated. These overheads cannot hence be easily addressed by increasing its size.

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. More MMC entries are needed when memory is tagged at small granularities to hold tag IDs for the same amount of memory, but enables more optimizations (e.g., bounds checking). Fig. 12 presents execution time for different granularities of tagging, normalized to the baseline system without MetaSys. For most workloads, even the smallest granularity we evaluated (64B) has a minimal impact on performance. High granularities minimize 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 low granularities increase the number of *TLB* misses (by 11% and 13% respectively), depicted in Fig. 13. The additional MMC misses cause accesses to the Metadata Mapping Table in memory which requires address translation.

To evaluate the effect of the TLB, we implement a design which does *not* require address translation to access the MMT (i.e., it is addressed directly in physical memory). Fig. 14 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. To evaluate the *scalability* of the system with multiple components accessing the same metadata support, we evaluate the overheads of two components performing frequent metadata lookups: one component at every *memory access* (with the corresponding memory address) and another at 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 worst-case scenario for the shared MMC.

Fig. 15 depicts the resulting execution time for both designs normalized to the baseline system. We observe that for all workloads except the microbenchmarks, increasing the number of client components only trivially impacts the performance overhead (on average the increase is 0.3%). This is because the MMC can sufficiently capture the tag ID working set for both components. The microbenchmarks designed to stress the system have a significant performance degradation (up to 60%) as a result of more misses in the MMC.

To investigate mechanisms to alleviate the contention overheads seen in the microbenchmarks we evaluate three designs in Fig. 16: *(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 (they are evicted last); and *(iii) No stall*, where we do not stall the core on an MMC miss (the optimization performed by the client will be 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* signifi-

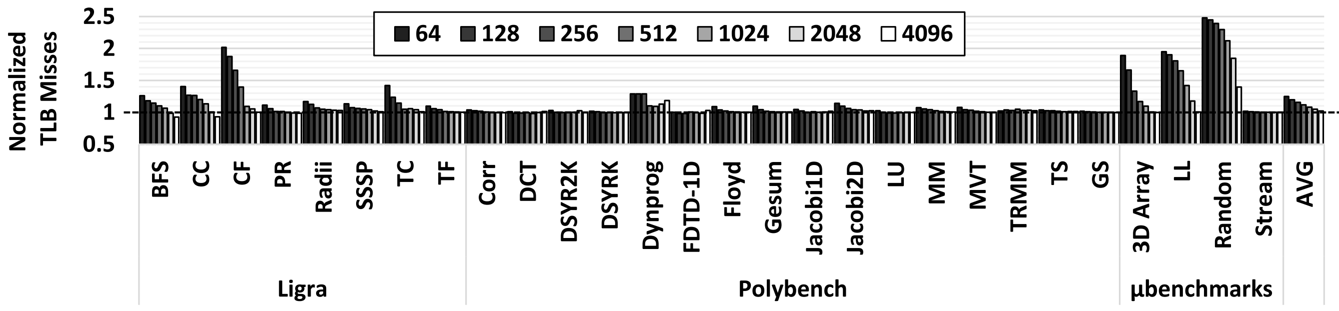


Figure 13: Impact of tagging

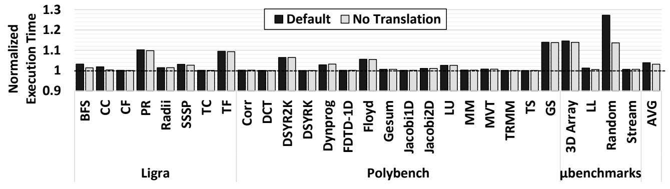


Figure 14: Performance overhead with no address translation for metadata.

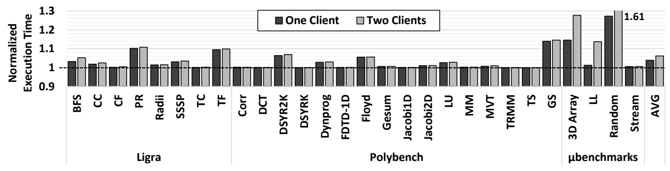


Figure 15: Performance overhead with multiple clients.

cantly reduces the overhead in Random (by 40%) by mitigating the latency overhead of additional memory accesses.

We conclude that the metadata support is scalable to multiple components with no observable 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, it can be expected to scale to more than two clients as most clients will query the MMC with the same addresses, which are aggregated, and will not lead to more lookups.

9.2. Hardware Area Overhead

We synthesized the baseline MetaSys system using the Synopsys DC [118] at 22nm process technology to estimate the area overhead. MetaSys incurs small area overhead: $0.03mm^2$ (0.02% of a 22nm Intel CPU Core [119]).

9.3. Summary of Findings

(1) Despite stressing the metadata support, the overall overheads of the system 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 for cross-layer performance optimizations 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 the metadata management is *scalable* in supporting multiple client components that have high frequencies of querying. There was no observable impact of the *number* of queries, indicating that the same system can

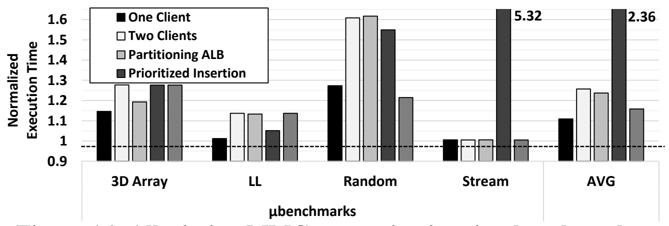


Figure 16: Alleviating MMC contention in microbenchmarks.

support multiple cross-layer optimizations at the same time. We proposed simple techniques to alleviate contention in the MMC from multiple clients.

(3) The most critical factor that impacted performance overhead was the effectiveness of the 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) In irregular workloads, accesses to tag ID mappings in memory require address translation and cause additional TLB misses, leading to higher performance overhead. We demonstrate that this can be mitigated by addressing mappings directly in physical memory or by using a separate TLB for mappings.

10. Conclusion

This work introduces MetaSys, an 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 towards enabling cross-layer performance optimization in CPUs.

References

[1] A. Mukkara, N. Beckmann, and D. Sanchez, "Whirlpool: Improving dynamic cache management with static data classification," in *ASPLOS*, 2016.

[2] M. Manivannan, V. Papaefstathiou, M. Pericas, and P. Stenstrom, "RADAR: Runtime-assisted dead region management for last-level caches," in *HPCA*, 2016.

[3] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *PACT*, 2002.

[4] P. Jain, S. Devadas, D. Engels, and L. Rudolph, "Software-assisted cache replacement mechanisms for embedded systems," in *ICCAD*, 2001.

[5] R. Ravindran, M. Chu, and S. Mahlke, "Compiler-managed partitioned data caches for low power," in *LCTES*, 2007.

[6] X. Gu, T. Bai, Y. Gao, C. Zhang, R. Archambault, and C. Ding, "P-opt: Program-directed optimal cache management," in *LCPC*, 2008.

[7] J. Brock, X. Gu, B. Bao, and C. Ding, "Pacman: Program-assisted cache management," in *ISMM*, 2013.

[8] K. Beyls and E. H. D'Hollander, "Generating cache hints for improved program efficiency," *JSA*, 2005.

[9] J. B. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang, "Cooperative caching with keep-me and evict-me," in *INTERACT-9*, 2005.

[10] H. Yang, R. Govindarajan, G. R. Gao, and Z. Hu, "Compiler-assisted cache replacement: Problem formulation and performance evaluation," in *LCPC*, 2003.

[11] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley, "Cooperative cache scrubbing," in *PACT*, 2014.

[12] A. Pan and V. S. Pai, "Runtime-driven shared last-level cache management for task-parallel programs," in *SC*, 2015.

[13] V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos, "Prefetching and cache management using task lifetimes," in *ICS*, 2013.

[14] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *MICRO*, 1995.

[15] N. Agarwal, D. W. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for GPUs within heterogeneous memory systems," in *ASPLOS*, 2015.

[16] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *EuroSys*, 2016.

[17] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *ASPLOS*, 2011.

[18] J. Ma, X. Sui, N. Sun, Y. Li, Z. Yu, B. Huang, T. Xu, Z. Yao, Y. Chen, H. Wang, L. Zhang, and Y. Bao, "Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD)," in *ASPLOS*, 2015.

[19] B. Huang, X. Jin, H. Wang, Y. Zhou, Z. Chang, Y. Cao, and Y. Bao, "Labeled RISC-V: A new perspective on software-defined architecture," in *CARVV*, 2017.

[20] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *ISCA*, 2014.

[21] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, P. Dabbelt, J. R. Hauser, A. M. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moretó, A. Ou, D. A. Patterson, B. H. Richards, C. Schmidt, S. M. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," ser. Technical Report No. UCB/EECS-2016-17, 2016.

[22] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, "A Case for Richer Cross-layer Abstractions: Bridging the Semantic Gap with Expressive Memory," in *ISCA*, 2018.

[23] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC*, 2012.

[24] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs," in *ISCA*, 2018.

[25] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "PUMP: A programmable unit for metadata processing," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, 2014.

[26] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language." in *ASPLOS*, ser. ASPLOS XIII, 2008.

[27] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *ASPLOS*, 2002.

[28] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos, "Efficient tagged memory," in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017.

[29] E. A. Feustel, "On the advantages of tagged architecture," *TC*, 1973.

[30] N. Zeldovich *et al.*, "Hardware enforcement of application security policies using tagged memory." in *OSDI*, 2008.

[31] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *ACM SIGPLAN Notices*, vol. 29, no. 11. ACM, 1994, pp. 319–327.

[32] H. M. Levy, *Capability-based computer systems*. Digital Press, 2014.

[33] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for UNIX," in *Proceedings of the 19th USENIX Conference on Security*, 2010.

[34] RISC-V, "RISC-V proxy kernel." [Online]. Available: https://github.com/riscv/riscv-pk

[35] AVNET, "Zynq®-7000 zedboard." [Online]. Available: http://zedboard.org/product/zedboard

[36] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13.

[37] L. Pouchet, "Polybench: The polyhedral benchmark suite." [Online]. Available: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[38] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[39] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18.

[40] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[41] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51, 2018.

[42] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki, "Prefedge: Ssd prefetcher for large-scale graph traversal," in *Proceedings of International Conference on Systems and Storage*, ser. SYSTOR '14.

[43] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS, 2016.

[44] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2018.

[45] N. Talati *et al.*, "Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design," in *HPCA 2021*.

[46] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '98.

[47] H. Al-Sukhni, I. Bratt, and D. A. Connors, "Compiler-directed content-aware prefetching for dynamic data structures," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '03.

[48] Chi-Keung Luk and T. C. Mowry, "Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[49] Zhenlin Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C.

Weems, “Guided region prefetching: a cooperative hardware/software approach,” in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, 2003.

[50] Tzi-cker Chiueh, “Sunder: a programmable hardware prefetch architecture for numerical loops,” in *Supercomputing ’94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*.

[51] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, “Semantic locality and context-based prefetching using reinforcement learning,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15.

[52] M. Annavaram, J. M. Patel, and E. S. Davidson, “Data prefetching by dependence graph precomputation,” in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA ’01.

[53] Z. Wang, K. S. Mckinley, and D. Burger, “Combining cooperative software/hardware prefetching and cache replacement,” in *In IBM Austin CAS Center for Advanced Studies Conference*, 2004.

[54] Z. Wang and T. Nowatzki, “Stream-based memory access specialization for general purpose processors,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19.

[55] K. J. Nesbit and J. E. Smith, “Data cache prefetching using a global history buffer,” in *10th International Symposium on High Performance Computer Architecture (HPCA’04)*, 2004.

[56] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Spatial Memory Streaming,” in *ISCA*, 2006.

[57] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, “Efficiently prefetching complex address patterns,” in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[58] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, “Dspatch: Dual spatial pattern prefetcher,” in *Proceedings of the 52Ndsd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’19.

[59] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Bingo spatial data prefetcher,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[60] J. W. C. Fu, J. H. Patel, and B. L. Janssens, “Stride directed prefetching in scalar processors,” in *MICRO*, 1992.

[61] W. Xu, D. C. DuVarney, and R. Sekar, “An efficient and backwards-compatible transformation to ensure memory safety of c programs,” in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, ser. SIGSOFT ’04/FSE-12, 2004.

[62] S. H. Yong and S. Horwitz, “Protecting c programs from attacks via invalid pointer dereferences,” in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-11, 2003.

[63] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, “Ccured: Type-safe retrofitting of legacy software,” *ACM Trans. Program. Lang. Syst.*, 2005.

[64] T. M. Austin, S. E. Breach, and G. S. Sohi, “Efficient detection of all pointer and array access errors,” in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI ’94, 1994.

[65] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, “Dependent types for low-level programming,” in *Proceedings of the 16th European Symposium on Programming*, ser. ESOP’07, 2007.

[66] D. Dhurjati and V. Adve, “Backwards-compatible array bounds checking for c with very low overhead,” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06, 2006.

[67] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, “Cyclone: A safe dialect of c,” in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’02, 2002.

[68] H. Patil and C. N. Fischer, “Efficient run-time monitoring using shadow processing,” in *AADEBUG*, 1995.

[69] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Softbound: Highly compatible and complete spatial memory safety for c,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’09, 2009.

[70] R. Hastings and B. Joyce, “Purify: Fast detection of memory leaks and access errors,” in *In Proc. of the Winter 1992 USENIX Conference*, 1991.

[71] D. Dhurjati and V. Adve, “Backwards-compatible array bounds checking for c with very low overhead,” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06, 2006.

[72] O. Ruwase and M. S. Lam, “A practical dynamic buffer overflow detector,” in *NDSS*, 2004.

[73] D. Midi, M. Payer, and E. Bertino, “Memory safety for embedded devices with nescheck,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’17, 2017.

[74] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, “Intel MPX explained: A cross-layer analysis of the Intel MPX system stack,” in *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’18, 2018.

[75] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, “Checked c: Making c safe by extension,” in *2018 IEEE Cybersecurity Development (SecDev)*, 2018.

[76] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP ’13, 2013.

[77] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon, “Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, ser. CCS ’13, 2013.

[78] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, “Watchdoglite: Hardware-accelerated compiler-based pointer checking,” in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’14, 2014.

[79] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, “Watchdog: Hardware for safe and secure manual memory management and full memory safety,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA ’12, 2012.

[80] D. Y. Deng and G. E. Suh, “High-performance parallel accelerator for flexible and efficient run-time monitoring,” in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012.

[81] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, “Memtracker: Efficient and programmable support for memory access monitoring and debugging,” in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[82] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, “Shakti-T: A RISC-V processor with light weight security extensions,” in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, ser. HASP ’17, 2017.

[83] A. Rogers, M. Carlisle, J. Laboratories, and L. Hendren, “Supporting dynamic data structures on distributed-memory machines,” *ACM Transactions on Programming Languages and Systems*, vol. 17, 2000.

[84] M. S. Simpson and R. K. Barua, “Memsafe: Ensuring the spatial and temporal memory safety of c at runtime,” in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010.

[85] A. van de Ven, “New security enhancementsin red hat enterprise linuxv.3, update 3.” [Online]. Available: https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf

[86] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM’98, 1998.

[87] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS ’10, 2010.

[88] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code-reuse attack.” 01 2011, pp. 30–40.

[89] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, “On the expressiveness of return-into-libc attacks,” in *Proceedings of the 14th International Conference on Recent Advances in Intrusi-flon Detection*, ser. RAID’11, 2011.

[90] A. Baratloo, N. Singh, and T. Tsai, “Transparent run-time defense against stack smashing attacks,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’00, 2000.

[91] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *Proceedings of the 12th ACM Conference on Computer*

and Communications Security, ser. CCS '05, 2005.

[92] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.

[93] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *NDSS*, 2015.

[94] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[95] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15, 2015.

[96] J. Zhang, R. Hou, J. Fan, K. Liu, L. Zhang, and S. A. McKee, "Raguard: A hardware based mechanism for backward-edge control-flow integrity," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17, 2017.

[97] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "Pac it up: Towards pointer integrity using arm pointer authentication," in *USENIX Security Symposium*, 2018.

[98] N. Roessler and A. Dehon, "Protecting the stack with metadata policies and tagged hardware," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[99] Intel, "Control-flow Enforcement Technology Specification," 2019. [Online]. Available: www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf

[100] C. MITRE, "Cwe-123: Write-what-where condition," September 2019. [Online]. Available: https://cwe.mitre.org/data/definitions/123.html

[101] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, ser. EUROSEC '09, 2009.

[102] J. Chang and G. S. Sohi, *Cooperative caching for chip multiprocessors*. ACM, 2006, vol. 34, no. 2.

[103] B. Li, L.-S. Peh, L. Zhao, and R. Iyer, "Dynamic qos management for chip multiprocessors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 3, p. 17, 2012.

[104] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and J. Torrellas, "iwatcher: efficient architectural support for software debugging," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, 2004.

[105] B. Lucia, L. Ceze, and K. Strauss, "Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations," *SIGARCH Comput. Archit. News*.

[106] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting atomicity violations via access interleaving invariants," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, 2006.

[107] A. Muzahid, N. Otsuki, and J. Torrellas, "Atomtracker: A comprehensive approach to atomic region inference and violation detection," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[108] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-aid: Detecting and surviving atomicity violations," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 277–288, 2008.

[109] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm, "Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, 2010.

[110] P. Zhou, R. Teodorescu, and Y. Zhou, "Hard: Hardware-assisted lockset-based race detection," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[111] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '96, 1996.

[112] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "undo" approach to safe speculation," in *MICRO 2019*.

[113] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *MICRO 2018*.

[114] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure Enclaves in a Speculative Out-of-Order Processor," in *MICRO 2019*.

[115] J. A. Joao, O. Mutlu, and Y. N. Patt, "Flexible reference-counting-based hardware acceleration for garbage collection," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 418–428.

[116] M. Maas, K. Asanović, and J. Kubiatowicz, "A hardware accelerator for tracing garbage collection," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 138–151.

[117] M. Maas, K. Asanovic, and J. Kubiatowicz, "Grail quest: A new proposal for hardware-assisted garbage collection," in *Workshop on Architectures and Systems for Big Data*, 2016.

[118] Synopsys, "Synopsys design compiler." [Online]. Available: https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html

[119] A. L. Shimpi, "Dual core/gt2 ivy bridge die measured: 121mm2." [Online]. Available: https://www.anandtech.com/show/5875/dual-coregt2-ivy-bridge-die-measured-121mm2