My research focuses on architecting rich cross-layer abstractions to bridge the semantic gap between the abstraction levels of the computing stack to open up a rich new space of optimization at each level: hardware architecture, compilers, and systems. Today, key information is lost in translation from the programmer’s intent to programming models, to the compiler, and to the hardware architecture, which only sees a sequence of instructions and memory accesses. Layered abstractions are critical to building complex systems but existing cross-layer interfaces restrict what can be done at each level. Crucially, they limit our ability to harness the fast-growing computational power and intelligence that hardware could offer.

My dissertation aims to rethink the interfaces with which the programming model, system, and hardware architecture interact to bridge this semantic gap. I proposed rich cross-layer abstractions that provide layered interfaces to directly communicate higher-level program semantics and intent to the lower levels of the computing stack. These abstractions enable significantly improved (1) performance and efficiency by enabling the system to adapt to application characteristics; and (2) productivity and portability by enabling application software to easily leverage diverse underlying hardware resources without specific knowledge of system details.

I designed abstractions that are demonstrably effective in enabling a wide range of cross-layer optimizations that improve performance, portability, and productivity. At the same time, they are highly practical, low-overhead, and require only small additions, fully compatible with existing interfaces. I developed end-to-end system designs of such abstractions including integration into the programming model, compiler, OS, and hardware architecture in both CPUs and GPUs, with a full system FPGA working prototype.

1. Motivation and Overview

The existing interfaces between layers of the computing stack, e.g., the instruction-set architecture (ISA) and virtual memory, strip any application down to the basics of what is required to execute code correctly: a sequence of instructions and memory accesses. Higher-level information—even the simple notion of data structures, their access semantics, data types, and properties—are all lost in translation. In other words, there is a large semantic gap between the application and the underlying system/hardware. Today, we work around this gap. In the hardware architecture, we try to predict/infer program behavior in the vast majority of optimizations we do (optimizing caches, memory, coherence, computation, and so on). We also rely on the application software to do much of the heavy lifting in optimizing code to the specifics of each architecture, leading to mounting challenges in portability and productivity as architectures get more diverse and complex. There have been numerous proposals for cross-layer optimizations, but since they require full-stack changes for a single optimization, they are challenging to adopt.

1.1. Key Contribution

In my research, I architected unifying cross-layer abstractions to bridge the semantic gap between the application and underlying system/hardware. These abstractions directly communicate higher-level program information, such as data structure semantics, parallelism, and data access properties, to the lower levels of the stack: compiler, OS, and hardware. This information is conveyed by the programmer using our programming abstractions or automatically inferred using software tools. The abstractions are expressive enough to convey a wide range of program information. At the same time, they are designed to be highly portable and low overhead, requiring only small additions to existing interfaces. This makes them highly practical and easy to adopt.

Providing the compiler, OS, and hardware a global view of program behavior, ahead of time, enables these components to actively optimize resources accordingly. For example, we demonstrated that knowledge of data structures and their access semantics enables the OS to place data intelligently in memory to maximize locality and parallelism [1]. Similarly, knowledge of the locality semantics of GPU programs enables the hardware thread scheduler to co-locate threads that share data at the same core to enhance data locality [2]. These are just two simple examples in a large space of innovations enabled in the OS, compiler, and hardware via access to key information. These rich cross-layer abstractions hence form efficient building blocks to enable hardware-software codesign, including numerous previously-proposed cross-layer optimizations.

I developed full-stack designs for both CPUs and GPUs, including: (1) rich programming abstractions that enable expression of application-level information and programmer intent, completely agnostic to the underlying system and hardware; (2) a cross-layer system that efficiently integrates the OS/runtime system and compiler, enabling these components to flexibly tap into a rich reservoir of application information; and (3) a low-overhead implementation in the hardware architecture.

1.2. Key Benefits

While there is a rich space of research opportunities into what a rich cross-layer abstraction enables, the benefits we demonstrated are detailed below.

(1) Enabling intelligent and application-specific cross-layer optimizations in the system/hardware: In addition to the aforementioned examples, more generally, communicating program information enables more intelligent cross-layer
optimizations to manage caches, memory, coherence, computation, and so on. Examples of this information include semantics of how a program accesses its data structures and properties of the data itself. The system/hardware can now effectively adapt to the application at runtime to improve overall system performance [1, 2]: For example, the Locality Descriptor [2] leverages knowledge of an application’s data access properties to enable coordinated thread scheduling and data placement in NUMA (non-uniform memory access) architectures. A richer cross-layer abstraction also enables customized optimizations: Expressive Memory [1] in CPUs enables adding specialization in the memory hierarchy to accelerate specific applications/code segments. For example, the abstraction enables transparently integrating a customized prefetcher for different types of data structures and their access semantics (e.g., hash table, linked list, tensor). Similarly, with Assist Warps [3], I demonstrated customized hardware data compression in GPUs, specific to the data layout of any data structure.

2. Overview of Dissertation Work [1–6]

Expressive Memory [1]: A rich cross-layer abstraction in CPUs to flexibly enable cross-layer optimization. In this work, we proposed a new cross-layer interface in CPUs, Expressive Memory (XMem), to communicate higher-level program semantics and programmer intent to hardware. We built a full-system implementation of Expressive Memory [1] in a RISC-V core on an FPGA platform. It comprises software libraries to interface with programs and full support in the compiler, OS, and hardware. This prototype is open-source to the community and can be used to flexibly implement and test cross-layer optimizations.

Locality Descriptor [2]: Expressing and leveraging data locality in GPUs with a rich cross-layer abstraction. While modern GPU programming models are designed to explicitly express parallelism, there is no clear way to express semantics of the program itself: i.e., how the thousands of concurrent threads access its data structures. We designed a rich cross-layer abstraction that describes how the hierarchy of threads in the GPU programming model access each data structure and the access semantics/properties of the data structures themselves. We then leverage this abstraction to significantly improve the efficacy and ease with which we can exploit data locality in modern GPUs—both reuse-based locality, to make efficient use of the caches, and NUMA locality, to place data and computation in near proximity in a non-uniform memory access (NUMA) system. Exploiting data locality in GPUs today is a challenging but elusive feat both to the programmer and the architect. Software has no access to key components (such as the thread scheduler) and hardware misses key information such as: which threads

1.3. Impact

Research Enabled. Free availability of application-level information with a practical and general abstraction opens up a rich new space of innovations in microarchitectural, OS, and compiler optimization. It pushes forward the boundary of what can be done at these levels, not only in improving the efficacy of existing optimizations, but also in enabling new optimizations and functionality that cannot otherwise be done (more details are in Section 4).

Open-Source Infrastructure. To enable further research in compilers, architecture, and systems in cross-layer optimizations, we built a full-system implementation of Expressive Memory [1] in a RISC-V core on an FPGA platform. It comprises software libraries to interface with programs and full support in the compiler, OS, and hardware. This prototype is open-source to the community and can be used to flexibly implement and test cross-layer optimizations.

Industrial Impact. I worked closely with Nvidia to leverage such cross-layer abstractions to address key challenges in GPU computing. The idea and mechanisms proposed in my research are actively being considered for future GPU architectures.
share data? We designed a powerful abstraction (named by its use case: the Locality Descriptor) that communicates the locality semantics of any application to the hardware. This enables hardware to transparently coordinate many locality optimizations such as co-scheduling threads that share data at the same core and placing data close to the threads that use it. The programming interface is designed to be seamlessly integrated into modern GPU programming models (e.g., CUDA and OpenCL). The abstraction's semantics are defined such that it can be automatically generated via software tools and is highly portable, making no assumptions about the underlying architecture.

**Zorua [4, 5]: Decoupling the programming model from explicit HW resource management.** In accelerators such as modern day GPUs, the available parallelism as well as the memory resources need to be explicitly managed by the programmer—there exists no powerful abstraction between the architecture and the programming model, and the management of many hardware resources is tied to the programming model itself. This leads to underutilization of resources (and hence, significant loss in performance) when the application is not well tuned for a given GPU. Zorua is a new hardware-software interface that decouples the resource management as specified by the programming model and the actual utilization in the system hardware by effectively virtualizing each of the resources. This virtualization enables the hardware to intelligently manage resources at fine granularities at runtime. This makes the performance of any program far less sensitive to the program implementation making high performance much easier to get and far more portable.

**Assist Warps [3, 6]: A HW-SW abstraction to leverage resource idleness.** In modern throughput-oriented processors—even with highly optimized code—imbalance between the compute/memory resources requirements of an application and the available resources can lead to significant idling of compute units and available memory bandwidth. The current programming models and interface to the architecture provides no simple abstraction to manage the utilization of critical resources. To leverage this undesired wastage to perform useful work, we proposed a new hardware-software abstraction—assist warps—in the GPU programming model and architecture. Assist warps [3, 6] allow light-weight execution of secondary code alongside the primary application to perform optimizations to accelerate the program (such as data compression or prefetching), perform background tasks, system-level tasks, etc.

### 3. Other Research [7–12]

I have also actively contributed to research projects outside the scope of my thesis:

**Systems for Machine Learning.** We developed a geo-distributed machine learning system, Gaia [7], that enables efficient training over globally-distributed data. Gaia employs an intelligent communication mechanism over wide-area networks (WANs) to efficiently utilize the scarce WAN bandwidth, while retaining the accuracy and correctness of the ML algorithm.

**Processing-in-Memory (PIM).** PIM is a promising paradigm to address a critical bottleneck—data movement between memory and computation—by placing some computation close to memory (e.g., the logic layer of stacked memories). We proposed accelerator designs and efficient mechanisms in CPUs [12] and GPUs [10] to address key challenges in the adoption of PIM architectures.

**Memory Systems.** Scaling DRAM latency is a significant challenge today and it continues to be a critical performance bottleneck. We developed a low-cost mechanism, ChargeCache [11], that enables faster access to recently-accessed rows in DRAM. To enable further research in DRAM scaling and reliability, we also developed SoftMC (Software Memory Controller) [9], an FPGA-based testing platform that can control and test memory modules designed for the commonly-used DDR interface.

**Data Compression.** Data movement is a critical bottleneck in modern systems. We investigated techniques to enable efficient and low-overhead data compression in hardware [3, 6, 8] as a highly effective technique to reduce the data transferred and stored in the memory hierarchy.

### 4. Future Directions

In my research thus far, I have architected general solutions and frameworks that have many use cases and enable alternative approaches to solve hard problems. I look for practical and realizable solutions in implementing such frameworks with the goal of fundamentally changing our approach to systems optimization, without significant and costly changes.

Moving forward, diminishing returns from traditional technology and system scaling necessitate radically new approaches to drive improvements in performance and efficiency. This includes mainstream integration of new hardware technologies and computing paradigms, such as processing-in-memory, quantum computing, specialized architectures, and reconfigurable fabrics, and a new landscape of cross-layer codesigns. To enable these disruptive changes, I am excited about rethinking the computing stack to redefine roles played by each layer of the stack and re-architect cross-layer abstractions in designing computer systems. I will continue to pursue general and realizable solutions that largely retain existing code bases and abstractions while still enabling big changes and many use cases.

I am also interested in cross-layer full-stack approaches to address key challenges in computer systems that increasingly require full-system solutions to make significant strides, such as providing guarantees in performance isolation, performance predictability, security, and reliability.

Below I describe research thrusts opened up by my research as well as other broader cross-layer research directions that tackle the aforementioned goals.
4.1. Pushing the boundary of what can be done by the system/hardware

4.1.1. Enabling rapid and fine-grain code adaptation at runtime, driven by hardware. As we increasingly rely on clouds and other virtualized environments, co-running applications, unexpected contention, and lack of visibility into available hardware resources make software optimization and dynamic recompilation limited in effectiveness. The hardware today cannot easily help address this challenge since the existing hardware-software contract requires that hardware rigidly execute the application as defined by software.

Vision: Enable system/hardware to dynamically change computation depending on availability of resources and runtime program behavior. The idea is to have the application only convey higher-level functionality, and then enable a codesigned hardware-software system to dynamically change the implementation as the program executes. The benefit of enabling such capability in hardware is greater efficiency in adapting software and more fine-grain visibility into dynamic hardware state and application behavior. For example, the program describes a potentially sparse computation (e.g., sparse matrix-vector multiply). The hardware then dynamically elides computation when it detects zero inputs. Other examples include changing graph traversal algorithms to maximize data locality at runtime or altering the implementation of a forward pass in each neural network layer, based on resource availability and contention. The challenge is in designing a general hardware-software system that enables many such runtime optimizations and integrates flexibly with frameworks such as Halide, TensorFlow, and Spark.

Approach: Design a clearly defined hardware-software abstraction that expresses what computation can be changed based on what runtime information (resource availability/bottleneck). This abstraction should integrate well into common building-block operations of important applications to obtain generality. Another approach is to determine how to effectively abstract and communicate fine-grained dynamic hardware state, bottlenecks, and contention, to enable software frameworks, databases, and other software systems to dynamically adapt applications accordingly.

4.1.2. Widening the scope of cross-layer and full stack optimizations. My research so far has demonstrated the significant performance, portability, and productivity benefits of cross-layer abstractions that are carefully architected to enable full-system coordination and communication to achieve these goals, from the programming model, compiler and OS, to each hardware component (cache, memory, storage, and so on).

Vision: Enable full-system coordination and communication to achieve other challenges such as security, quality-of-service (QoS), meeting service-level objectives (SLOs), and reliability. The challenge is in designing cross-layer abstractions and interfaces that enable very disparate components (e.g., cache, storage, OS thread scheduler) to communicate and coordinate, both horizontally and vertically in the computing stack, to meet the same goal.

Approach: I will look into enhancing the existing compute stack to enable these holistic designs and cross-layer optimizations. This will involve determining how to express application-level requirements for these goals, how to design low-overhead interfaces to communicate these requirements to the system and each hardware component, and then enhance the system accordingly to achieve the desired goal. I believe insights from my completed work will be applicable to solutions here.

Long Term: I will research clean-slate approaches to building systems with modularized components that are designed to provide full-system guarantees for performance isolation, predictability, reliability, and security. The idea is to compose the overall system from smaller modules where each module or smallest unit is designed to provide some guarantee of (for example) strict performance isolation. Similarly the interfaces between module should preserve the guarantees for the overall system. Initial research questions: What are the semantics that define a module? What are the semantics that define interactions between modules?

4.2. Integrating new technologies, compute paradigms, and specialized systems

Future systems will incorporate a diverse set of technologies and specialized chips, that will rapidly evolve. This poses many new challenges across the computing stack in the integration of new technologies, such as memristors, persistant memory, and optical devices, and new paradigms, such as quantum computing, reconfigurable fabrics, application-specific hardware, and processing-in-memory substrates. Below I describe future directions relating to different aspects of this problem.

4.2.1. Enabling applications to automatically leverage new systems and architectures (a software approach). Automatically generating high-performance code for any new hardware accelerator/specialization today, without fully rewriting applications, is a challenging task. Software libraries have been demonstrated to be very inefficient and are not general. Automatic code generating frameworks and optimizing compilers (e.g., [13–15]) are promising approaches to target a changing set of architectures, without rewriting application code. However, such tool chains use a common intermediate representation (IR) to summarize the application and then use specialized backends that optimize the IR according to the characteristics of the new architecture and produce target code. Integration of new architectures or even new instructions/functionalities in CPUs/GPUs (e.g., to add instructions to leverage a processing-in-memory accelerator) into these systems takes significant time, effort, and cross-layer expertise.

Vision: Develop frameworks to enable automatic generation of high performance code for specialized/new hardware, without rewriting existing applications. This would significant reduce the effort required to evaluate and deploy new hardware innovations including new architectures, substrates, or finer granularity hardware primitives (e.g., a faster in-memory operation).
Tensor Cores within GPUs to speed up tensor operations in machine learning, accelerating data intensive computation using the efficiency of the new architecture.


Vision: Enable seamless integration of specialization and reconfiguration into general-purpose architectures, transparently to the software stack. This would enable specialized designs to continuously evolve without creating new primitives/instructions each time (and hence no recompilation/rewriting is required). This addresses the critical portability and compatibility challenges associated with these approaches. For example, this would enable seamless addition of specialized/reconfigurable hardware support within general-purpose cores for: sparse and irregular computations (e.g., graph processing), managed languages (e.g., support for object-based programming, garbage collection), critical computations (e.g., numerical loops in machine learning) and frequent operations (e.g., queries in databases), among very many possibilities.

Approach: Design an abstraction that enables easy and flexible description of the performance characteristics, constraints, and the semantics of the interface to hardware, i.e., the functionality implemented by the instruction-set architecture (ISA) or new primitive. Next, develop tools that enable automatic integration into existing compiler IRs to generate backends based on the description. I will also look into how to enhance existing IRs to capture more semantic content if required.

Long Term: Develop hardware-software frameworks that enable flexible evaluation and design space exploration of how to abstract new hardware technologies and substrates in terms of the overall programmability (how many applications can leverage the technology), portability (how easily we can enhance the architecture without changing the interface), and performance (efficiency of the new architecture).

4.2.2. Enabling software-transparent hardware specialization and reconfiguration (a hardware approach). Today, even within general-purpose cores, architects are turning to different forms of hardware customization and reconfigurability for important computation as a means to drive improvements in performance and energy efficiency. Examples include Tensor Cores within GPUs to speed up tensor operations in machine learning, accelerating data intensive computation using processing-in-memory technologies, or reconfigurable fabrics (e.g., CGRAs) to accelerate important computations.

References


