Department: Internet of Things, People, and Processes
Editor: Schahram Dustdar, dustdar@dsg.tuwien.ac.at

Rethinking Divide and Conquer—Towards Holistic Interfaces of the Computing Stack

Schahram Dustdar
TU Wien

Nandita Vijaykumar
University of Toronto

Onur Mutlu
ETH Zürich

***Abstract*—We argue that the abstractions between the layers of the computing stack and the components of computing systems, especially the HW/SW interface, have to be rethought to cope with the ever-growing complexity of problem domains and their manifestations in the underlying computing systems. The divide and conquer approach to hardware/software with a minimal interface is unable to cope with complexity. Rethinking the abstractions and interfaces between the application, system, and architecture can lead to significant benefits in improving performance, efficiency, resilience, security, and programmability, at the same time.**

MOTIVATION

The "divide and conquer" paradigm has been used since the beginnings of computing and its engineering practices. This approach is taught in courses in all Computer Science curricula at universities and has remained unchallenged and unquestioned ever since. The definition of "divide and conquer" according to the Merriam Webster dictionary is: "to make a group of people disagree and fight with one another so that they will not join together against one: 'His military strategy is to *divide and conquer*.'" As such,

Digital Object Identifier 10.1109/MIC.2020.3026245
Date of current version 20 November 2020.

this principle is visibly used in our daily design and engineering practices and is a fundamental property of virtually all organizational and political systems today.

The main issue and often unexpected and counter-intuitive result of the "divide and conquer" methodology is that it utterly fails to create the ever increasing *complex* ultra-large-scale systems. Furthermore, this methodology is challenged in helping to design and engineer systems that provide coherent functionality utilizing often conflicting properties, over time. New technologies including Internet of Things (IoT) sensors and edge devices are now being integrated into mainstream software systems and systems engineers aim at "integrating" them into existing systems following the "divide and conquer" approach. Naturally, this approach causes many issues (similar to an attempt to integrate a new organ into the Autonomic Nervous System (ANS) of the human body, and then manage all communications to provide a neatly integrated systems thereafter).

The "integration paradigm"—as a direct result of the "divide and conquer" approach to engineering—led ultimately to a number of "patches" in systems engineering over the years. In the domain of software, several concepts were introduced to cope with the problems introduced by the consequences of failed true integration: Cookies (1994), Java (1995), JavaScript (1995), Web services (1998), and Semantic Web (1999), just to name a few. In the hardware domain, the ability to integrate is restricted to devices obeying well-defined, yet very inexpressive and rigid, interfaces such as the processor-memory, processor-storage, and various bus/channel interfaces to interconnect components and peripherals.

The "divide and conquer" method has certainly served us well over the past decades. Many industries, including the Hardware, Networking, Storage, and Software industries have thrived based on its principles. The "divide and conquer" principle has been applied on two dimensions: horizontal and vertical. Vertically, across layers of the computing stack, and horizontally, across components in a layer, interfaces are created and obeyed so that one layer (e.g., software) does not need to know much about another layer (e.g., hardware) and so that one component (e.g., the memory chip) does not need to know much about another component, thus each layer/component can independently be optimized and developed. Today, we are reaching a level of complexity in our systems (hardware and software), which makes it very hard for engineers to design, model, program, maintain, and test with the divide and conquer approach. Both, the benefits and the downsides of the "divide and conquer" method are multiple and we will discuss them in the following paragraphs in some detail.

CHARACTERISTICS OF THE DIVIDE AND CONQUER PARADIGM

In virtually all areas of engineering, the divide and conquer method is used in the large as well as in the small as a method to create specific algorithms and designs. It is a powerful tool for solving conceptually difficult problems: fundamentally, it is a method of breaking the problem into subproblems, of finally combining subproblems to solve the original, usually bigger, problem.

Advantages of the "divide and conquer" method include: enabling *distribution of work*, by drawing administrative, organizational, as well as management and technical separations of concerns. It allows for lowering the *communications and coordination* costs during development and processing; it enables *parallelization* of a task across components, assuming the task can be partitioned into components with minimal communication. iIt eases the *reasoning* about the overall system, including its costs. It increases the *efficiency and productivity* of the systems' creation; it allowed for *complex architectures* to be created and maintained.

Disadvantages of the "divide and conquer" method include: if the work cannot be partitioned to separate components with little communication, *data movement* inside the overall system (Computer Architecture as well as Software) becomes extensive and causes large bottlenecks; there is no method to provide a built-in *end-to-end (holistic) management* of the overall system for various different metrics because the components are by design not holistically designed and

coordinated; it has massive limitations and issues in maintaining all *Quality of Service (QoS)-type of guarantees*; it renders the overall system *hard to predict* and hard to reason about.

An example bottleneck created by divide-and-conquer: The Data Movement Wall. One key disadvantage of divide and conquer leading to extreme energy inefficiency (and large performance loss) in modern systems is the fact that it causes large amounts of data movement between different components that are designed separately from each other. Take, for example, the simple separation of the processor and memory from each other, a design choice that has been prevalent for at least the past 60 years. The processor is considered the master and the memory is treated as a device whose sole function is to store data and serve the commands of the processor. As a result, the CPU can only perform any operation on the data once the data is in brought into the CPU cache from main memory. The process of moving data from main memory to the CPU incurs a long latency, and consumes a significant amount of energy. A single memory access consumes 2-3 orders of magnitude more energy than a complex addition operation performed in the processor.[15,18] These costs are often exacerbated by the fact that much of the data brought into the caches is not reused by the CPU,[16,18] providing little benefit in return for high latency and energy cost. Our results show that 62.7% of the entire system energy in a modern SoC running key workloads such as web browsing, machine learning inference (Google Tensorflow), and video capture and decoding, is spent solely on data movement![15] In other works, we have demonstrated that by holistically redesigning the processor and memory such that processing can be done very close to or inside the memory, five key graph processing workloads can be accelerated by 13.8X[16] and their energy reduced by 8X[16] and key query latencies in databases can be reduced by as much as 12X[17], on average, due to the elimination of large amounts of data movement. Thus, there could be great benefits in replacing the divide and conquer methodology for designing processor vs. memory with a more holistic methodology of designing processor and memory.

We argue that we are at a point in computing where the complexity of systems we create and the energy cost of data movement as well as the need for energy efficiency is rendering the "divide and conquer" approach to designing systems extremely inefficient and incapable in providing key end-to-end requirements (such as predictability, security, latency, QoS; see sidebar "Higher System Complexity with Humans in the Loop"). With this in mind, in this paper, we ask the key questions: How can we create coherent, scalable, efficient systems that consists of billions of heterogeneous components? Can the "divide and conquer" method be extended and engineered in a way so we can still keep it but make better use of it? Can we enable much more effective systems that provide larger end-to-end benefits? We identify the current **interfaces between layers and components** as a culprit that prevents holistic and efficient heterogeneous designs. These interfaces are very narrow, rigid, and inexpressive. In this paper, we examine the "interface expressiveness" problem between computing layers and components and propose "holistic interfaces" as a solution direction.

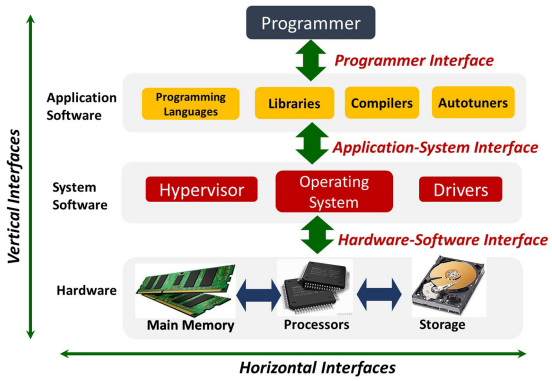

Figure 1. Abstract examples of the interfaces in the computing stack: vertical and horizontal.

INTERFACE EXPRESSIVENESS: A KEY ISSUE FOR HOLISTIC DESIGN

A key characteristic of the "divide and conquer" approach lies in the interfaces it creates across different layers (vertical) and across different components (horizontal), as shown in Figure 1. These interfaces enable communication and coordination across layers/components so that different components can communicate and synchronize with each other. However, due to

Higher System Complexity with Humans in the Loop

Today, we witness the evolution of a new breed of systems, which are composed of three building blocks: people, software services, and things. In the past, people were considered outside the system, interacting with the system, but not a functional building block of the system itself. There was hardly the notion of designing the interfaces of people-based systems and no attention was given to how exactly one can build mechanisms for machines (software/hardware ensembles) and humans to collaborate in an automated way. Computer Science in general and Software Engineering in particular, predominantly focused on the design and engineering of Software (services). More recently, since the dawn of cloud computing, the business side of information systems became more relevant as a design issue, i.e., how many resources and when they are used and how much they cost are designed as firs- class entities in information systems. Things, in the sense of the Internet of Things (IoT), were not even considered for main building blocks of software systems and were considered mainly the domain of electrical engineering projects and certainly not to be tightly integrated with software systems. Of course, that changed dramatically with the emergence of the Internet of Things. IoT sensors of all sorts became of concern for software-intensive projects, e.g., in buildings, factories, and even in so called Smart Cities. Other examples include Smart Homes, Smart Governments, eHealth networks, and energy networks, just to name a few.

If we observe closely, we can say that the field of computer science may be, to a first order, mimicking the human body and brain with all of its developments. This is true for computer architecture in general but also for software. The way we may have assumed how the brain operated (perhaps 70+ years ago) still serves as a blueprint for how the CPU works and is architected today as well as how software itself is designed. The IoT sensors are the newest development along those lines, mimicking the human sensors for their ability to sense their environment and provide that data to a computing device, today, typically operating on the Cloud (i.e., Cloud Data center). Therefore, we can say that the IT industry is developing is a copy of our own "self," for example, mimicking the Autonomic Nervous System (ANS). However, there are also significant differences from the ANS. It is worthwhile noting that all the organs are ontologically distinct but operationally harmonized and balanced in the ANS. This situation is worthwhile to replicate in the IoT/Cloud/Edge Computing ecosystems.

The ANS can be considered a perfect template for what is currently being built in the IoT/Edge/Cloud Computing ecosystem: The ANS consists of a network of networks, connecting organs and nerves via the spine. Sensors are all over the ANS and provide data exchange functions inside those networks as well as to the brain. The analogy to the information systems with IoT sensors (allowing the perceptions for the information systems), edge devices, and cloud computing data centers is truly amazing, but clearly intended. What is important to note in this context is that the fundamental properties of the ANS and its resilience is actually not visible. We can say it is "built-in", an inherent characteristic of the system itself. Another important property is that the ANS - or in fact any system such as this - could not be built by merely integrating subsystems, which were *previously engineered*. This is, however, exactly what is being done in our current state-of-the art in distributed systems and computing in general. We first design and engineer "silos" (often referred to as vertical solutions) and then their integration is being addressed with simple and minimal interfaces. In the past few decades this approach worked to an extent. However, it is becoming more than evident that the level of complexity we have reached today in our distributed systems does not allow to design, engineer, monitor, test, and execute and maintain them efficiently anymore. To use an analogy from the human body, it would be as if we had designed the kidney and then we move on to integrate it with the liver and then with the heart and so on, without providing good interfaces and harmony between these different components. It should be obvious that this approach cannot work successfully in complex ultra-large-scale systems.

the design mentality of the modern divide and conquer approach, these interfaces are quite inexpressive and minimal. As a result, such interfaces greatly limit the building of a holistic system from different components and lead to a profound inability to construct systems that can provide holistic performance, reliability, security, QoS, energy, predictability guarantees, and a profound difficulty in managing the underlying heterogeneous components (in both software and hardware).

Examples of inexpressive and limited interfaces abound in the computing systems we design today, across layers (vertical) and across components (horizontal), including the following (which are loosely depicted in Figure 1):

1. The **hardware/software interface** (i.e., the instruction set architecture), which is mostly limited to the expression of instructions and data locations in terms of memory and register addresses.[8]
2. The **processor/memory interface**, which is mostly limited to primitive read/write/refresh commands, addresses, and simple data communication protocols.[18]
3. The **processor/storage interface**, which is mostly limited to read/write-based communication protocols.
4. The **system software/microarchitecture interface**, which is mostly limited to very specialized instructions and memory addresses.
5. The **programmer/language interface**, which is mostly limited to low-level constructs defined in programming languages (e.g., data structures, control flow constructs), but unable to convey programmer's intent, goals, and problem constraints.

Take, for example, a key cross-layer interface, the **hardware/software interface** (see Figure 2), which creates a boundary between two major layers of computing, the hardware and the software, enabling each to develop and be optimized in a "divide and conquer" manner. If this interface does not enable enough quality communication and creates a bottleneck (because it is not expressive enough, as shown in Figure 2), the hardware would not have enough information about what the program is doing. The hardware could try to re-construct the high-level information that is available in software but gets lost through the interface. This, however, causes great complexity, cost, and energy inefficiency while providing questionable benefit in many workloads.[8–10] The extremely valuable information available at the software level is wasted as it is not communicated to the hardware. We will get back to the HW/SW interface again, since it forms a huge bottleneck that fundamentally affects all our computing systems and their properties today.

Take, as a key example of a component-level interface, the **processor/memory interface** (or the **memory-controller/memory** interface), which creates a boundary between two major components of computing and enabling each component to be developed and optimized in a "divide and

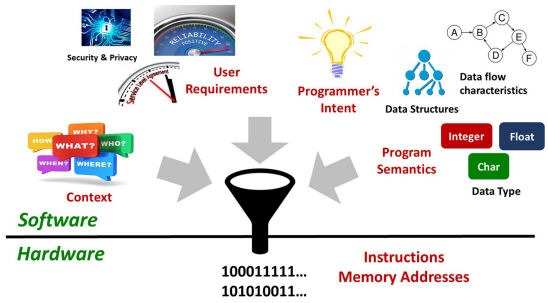

Figure 2. Software/Hardware Interface as a bottleneck.

conquer" manner. In modern systems, this interface is minimal: the processor can simply issue basic commands (read, write, refresh, wait, etc.) and addresses to memory and memory simply responds with data. This results in a dumb memory device, which is incapable of doing anything other than storing data. As a result, a significant amount of data movement ensues because memory can do nothing else to the data but send it to (or receive it from) the processor. This data movement causes great energy inefficiency and performance loss, as we discussed above (under the Data Movement Wall). Similarly, because the memory cannot communicate its physical characteristics and structure to the processor/memory-controller, the processor treats memory essentially as a black box and applies worst-case-driven policies to it.[11–13] For example, every memory cell is refreshed every 64 ms in modern DRAM, even though an overwhelming majority of cells do *not* require to be refreshed for 256 ms and even longer.[11] In general, most information regarding energy, performance, resilience, and security (see the RowHammer problem),[12,19] QoS, predictability (and likely any other key optimization metric) gets lost due to the extremely limited interface between the processor and memory. Alternatively, the system becomes too complex to recover the lost information (via high-cost and high-complexity techniques) that could otherwise be simply communicated via a more expressive interface.[18]

HARDWARE/SOFTWARE INTERFACE: A CRITICAL CROSS-LAYER INTERFACE TO RETHINK

Traditionally, the key interfaces between the software stack and the architecture (the ISA and

virtual memory) have been primarily designed to convey program functionality, ensuring the program is executed as required by software. An application is converted into ISA instructions and a series of accesses to virtual memory for execution in hardware. The application is, hence, stripped down to the basics of what is necessary to execute the program correctly, and the higher level semantics of the program are lost (see the funnel in Figure 2). For example, even the simple higher level notion of different data structures in a program is *not* available to the OS or hardware architecture, which deal only with virtual/physical pages and addresses. While the higher level semantics may be irrelevant for correct execution, these semantics could prove *very useful* to the system for performance, energy, resilience, QoS, privacy, and security optimizations.

Despite the rapid evolution and advancements at *all* levels of the computing stack, from application to hardware, the key abstractions in the computing stack and the role they play *have largely stayed the same.* This leads to an ever-growing *disconnect* between the levels of the stack when it comes to conveying higher level program semantics from the application to the wide range of system-level and architectural components that aim to improve performance, efficiency, resilience, and security, to name a few. The cross-layer abstractions are *narrow*, in terms of the information conveyed and *rigid*, in terms of the roles played by each of the levels. This disconnect has two important implications that makes achieving many key metrics (e.g., programmability, portability, resource efficiency, resilience, performance, energy efficiency, and security) significantly challenging:

Implication 1: ***System/architecture is unaware of higher level program semantics.*** The program is stripped down to the basics of what is solely required to execute the program correctly (to the ISA and memory addresses) and higher level program information is lost (see Figure 2). Optimizations at the system/hardware level could be far more effective when driven by direct knowledge of application behavior and program semantics, thus enabling the system/architecture to efficiently *adapt* to the application characteristics.

Implication 2: ***Application/system manages low-level hardware resources with limited visibility and access.*** The application and system software need to be aware of low-level system resources, and manage them appropriately to tune for the optimization metrics. This causes challenges in programmability, portability, resource efficiency, and other metrics to be satisfied. The software may *not* always have visibility into available resources such as available cache space (e.g., in virtualized environments) and even if it does, software has little access to many hardware features that are critical when optimizing for performance (e.g., caching policies, memory mapping).

The implications of this disconnect are only growing: Trends in ***domain-specific specialization*** has led to the development of domain-specific languages, compilers, and frameworks (e.g., Halide, Pochoir, TensorFlow, CNTK; see sidebar "Prior Approaches to Cross-Layer Communication"); and hardware specialization in the form of accelerators (e.g., GPUs, FPGAs), specialized memories (e.g., 3-D XPoint, HBM) and ASICs (e.g., machine learning accelerators). This leads to ever more diversity and complexity in software and hardware resources, making them harder than ever to effectively employ and harness. Similarly, ***virtualization*** of system resources has become pervasive, where multiple applications are consolidated on the same platform to enable better efficiency, and applications are flexibly deployed across a range of platforms depending on cost constraints and performance requirements (Cloud computing). This further reduces the visibility the system/architecture has over the applications that are running on them and makes static software optimizations far less effective since the available system resources may be unknown and constantly changing. The rise of ***data-centric computing*** over increasing volumes of data places ever more stress on memory and compute resources and has driven rapid development and hence, increased complexity, at different levels of the stack to improve resource efficiency. Increased complexity at each level makes it more challenging for the system/architecture to infer application characteristics and vice versa.

Prior Approaches to Cross-Layer Communication

Hardware-Software Cooperative Approaches

The challenges of predicting program behavior and hence the benefits of knowledge from software in memory system optimization are well known.[1–14] There have been numerous hardware-software cooperative techniques proposed in the form of fine-grain hints implemented as new ISA instructions (to aid cache replacement, prefetching, etc.),[1–7] program annotations or directives to convey program semantics and programmer intent,[8–10] or hardware-software co-designs for specific optimizations.[11–14] These approaches, however, have two significant shortcomings. First, they are designed for a specific optimization and are limited in their implementation to address only challenges specific to that optimization. As a result, they require changes across the stack for a single optimization (e.g., cache replacement, prefetching, or data placement). Second, they are often very specific directives to instruct a particular component to behave in a certain manner (e.g., instructions to prefetch specific data or prioritize certain cache lines). These specific directives create portability and programmability concerns because these optimizations may not apply across different architectures and they require significant effort to understand the hardware architecture to ensure the directives are useful.

Domain-Specific Languages (DSLs) and Expressive Programming Models/Runtime Systems

Domain-specific languages (DSLs) and domain-specific frameworks are specialized for a specific domain of applications (e.g., image processing,[15] graph processing,[16,17] machine learning[18,19]) and improve expressiveness by providing higher-level constructs specialized for that domain. This makes increasing programmer productivity and enables the underlying software stack to produce high performance code implementations by leveraging knowledge of the application characteristics. These languages are however specialized for a single application domain and are hence restricted in usage to what can be expressed in that language. While these languages provide higher-level constructs tailored for a specific domain (e.g., images), they are not powerful enough to express context, reliability/security requirements, SLAs, programmer intent, etc.

Numerous software-only approaches tackle the disconnect between an application, the OS, and the underlying resources via programming models and runtime systems that allow explicit expression of data locality and independence[21–25] in the programming model. This explicit expression enables the programmer and/or runtime system to make effective memory placement decisions in a NUMA system or produce code that is optimized to effectively leverage the cache hierarchy. These approaches are entirely software-based and are hence limited to using the existing interfaces to the architectural resources. Furthermore, these systems are specific to an application type (e.g., operations on tiles, arrays), and hence can only benefit applications that fit a certain type.

REFERENCES

1. X. Gu, T. Bai, Y. Gao, C. Zhang, R. Archambault, and C. Ding, "P-OPT: Program-directed optimal cache management," in *Int. Workshop Languages Compilers Parallel Comput.*, Springer, Berlin, Heidelberg, pp. 217–231, Jul. 2008.
2. J. Brock, X. Gu, B. Bao, and C. Ding, "Pacman: Program-assisted cache management," *ACM SIGPLAN Notices*, vol. 48, no. 11, pp. 39–50, 2013.
3. Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, Sept. 2002, pp. 199–208.
4. K. Beyls, and E. H. D'Hollander, "Generating cache hints for improved program efficiency," *J. Syst. Archit.*, vol. 51, no. 4, pp. 223–250, 2005.
5. J. B. Sartor, S. Venkiteswaran, K.S. McKinley, and Z. Wang, "Cooperative caching with keep-me and evict-me," in *9th Annu. Workshop Interaction Between Compilers Comput. Archit.*, pp. 46–57, Feb. 2005.
6. H. Yang, R. Govindarajan, G.R. Gao, and Z. Hu, "Compiler-assisted cache replacement: Problem formulation and performance evaluation," in *Int. Workshop Languages Compilers Parallel Comput.*, Springer, Berlin, Heidelberg, pp. 77–92, Oct. 2003.
7. "Memory management optimizations on the Intel Xeon Phi coprocessor," Intel Compiler Lab, 2015. [Online]. Available: https:// software.intel.com/ sites/default/les/ managed/b4/24/mem_management_dgemm.pdf
8. N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for GPUs with in heterogeneous memory systems," in *Proc 20th Int. Conf. Archit. Support Programming Languages Operating Syst.*, 2015, pp. 607–618.

9. A. Mukkara, N. Beckmann, and D. Sanchez, "Whirlpool: Improving dynamic cache management with static data classification," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 2, pp. 113–127, 2016.
10. S. R. Dulloor *et al.*, "Data tiering in heterogeneous memory systems," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, pp. 1–16.
11. T. F. Chen, "An effective programmable prefetch engine for on-chip caches," in *Proc. 28th Annu. Int. Symp. Microarchit.*, 1995, pp. 237–242.
12. S. P. Vander Wiel, and D. J. Lilja, A compiler-assisted data prefetch controller. in *Proc. IEEE Int. Conf. Comput. Des.: VLSI Comput. Processors*, Cat. No. 99CB37040, 1999, pp. 372–377.
13. T. C. Chiueh, "Sunder: A programmable hardware prefetch architecture for numerical loops," in *Supercomputing'94: Proc. ACM/IEEE Conf. Supercomputing*,1994, pp. 488–497.
14. Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: a cooperative hardware/software approach," in *Proc. 30th Annu. Int. Symp. Comput. Archit.*, 2003, pp. 388–398.
15. J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. 34th ACM SIGPLAN Conf. Programming Language Des. Implementation*, pp. 519–530, 2013. [Online]. Available: https://doi.org/10.1145/2491956.24621762013
16. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," 2012, *arXiv:1204.6078*.
17. K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. Amber Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proc. 32nd ACM SIGPLAN Conf. Programming Language Des. Implementation*, 2011, pp. 12–25. [Online]. Available: https://doi.org/10.1145/1993498.1993501
18. F. Seide, and A. Agarwal, "CNTK: Microsoft's open-source deep-learning toolkit," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, pp. 2135–2135, 2016.
19. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and M. Kudlur, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symp. Operating Syst. Des. Implementation*, pp. 265–283, 2016.
20. P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *Proc. 20th Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, 2005, pp. 519–538. [Online]. Available: https://doi.org/10.1145/1094811.1094852
21. B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.
22. K. Fatahalian *et al.*, "Sequoia: Programming the memory hierarchy," in *Proc. ACM/IEEE Conf. Supercomputing*, 2006, pp. 83–es.
23. G. Bikshandi *et al.*, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proc. 11th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2006, pp. 48–57.
24. M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 1–11.
25. J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua, "Programming with tiles," in *Proc. 13th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2008, pp. 111–122.

TOWARDS HOLISTIC INTERFACES

We believe there are some fundamental properties of programs and components that do not get conveyed across the key interfaces we have today, including:

- Data structure information and semantics (locality, access pattern, etc.).[8,9]
- Data flow characteristics of programs. Which component needs which data (and potentially when).
- Programmer intent in a given piece of code[6] and data.
- SLA requirements, QoS, and contracts driven by the user.

- Elasticity requirements of the given components.[1-5]
- User goals and context.[7]
- Key internal characteristics of a given component so that another component or layer can manage the component much more efficiently, e.g., internal characteristics of a memory chip exposed to the memory controller.[11–13]
- Reliability requirements of different data[10] and code portions.
- Security requirements of different data and code portions.
- Privacy requirements of data and storage portions.

We elaborate on initial examples of how the communication of some of the above properties across various interfaces could be useful in achieving various goals.

Example 1: Data Structure Information

Today's interfaces communicate little about higher level program semantics from software to underlying layers. Data structure level information is one example. Our recent work, called Expressive Memory (XMem),[8] shows that having an interface that communicates data structure access semantics and locality characteristics from the software to the hardware can enable significant performance benefits by adapting the management policies in the memory system to the characteristics and requirements of different data structures and program portions. XMem provides (i) a flexible and extensible abstraction, called an Atom, enabling the application to express key program semantics in terms of how the program accesses data and the attributes of the data itself, and (ii) new cross-layer interfaces to make the expressed higher level information available to the underlying OS and architecture. By providing key information that is otherwise unavailable, XMem exposes a new, rich view of the program data to the OS and the different architectural components that optimize memory system performance (e.g., caches, memory controllers). XMem enables architectural/system-level techniques to leverage key program semantics that are challenging to predict or infer. It also improves the efficacy and portability of software optimizations by alleviating the need to tune code for specific hardware resources (e.g., cache space). While XMem is designed to enhance and enable a wide range of memory optimizations, our recent work[8] demonstrates the benefits of XMem via two concrete use cases: (i) improving the performance portability of software-based cache optimization by expressing the semantics of data locality in the optimization and (ii) improving the performance of OS-based page placement in DRAM by leveraging the semantics of data structures and their access properties.

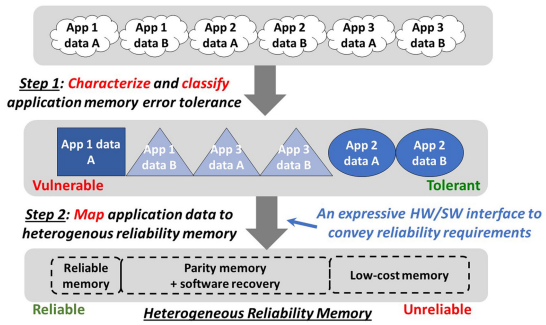

Figure 3. Expressing heterogeneous reliability requirements across the stack.

While extensive future work is needed, we believe XMem is an initial yet important step in breaking the inexpressiveness and rigidity in the cross-layer interfaces between program semantics and system software and hardware. Many use cases of XMem exist, with significant benefits in performance, programmability, and resource efficiency, as demonstrated.[8] We hope XMem encourages future work to explore rearchitecting the traditional interfaces to enable many other benefits that are not possible today.

Example 2: Reliability Requirements of Data

One key information that could be extremely helpful in improving system cost, resilience, complexity, scalability, and security is the *reliability/resiliency requirements* of different pieces of data that programs/system deal with. If the system knows the reliability requirements of data, it could treat *different data* in *different ways:* for example, reliability-critical data can be allocated in extremely reliable (yet high cost and low-capacity memory) whereas data that is not as critical, i.e., data that can tolerate errors, can be allocated in much lower cost, larger capacity, less reliable memory (as shown in Figure 3). Today, such data-level reliability requirements *cannot* be communicated via the inexpressive

and rigid interfaces we have between layers. Yet, our experience with a variety of data-intensive workloads (including Microsoft's production Web Search workload) shows that communicating the reliability/resiliency/error-tolerance requirements of different pieces of data from the software to the operating system and hardware, and managing heterogeneous-reliability memory systems using this information, can lead to great improvements in data center scalability, efficiency, and cost.[10] We believe many such opportunities exist in both software and hardware optimization if the interfaces are enriched with expressive and flexible information about the reliability/resiliency requirements and error tolerance of different data items.

Example 3: Programmer Intent

Another key information that is currently not conveyed across the software/hardware interface is the intent of the programmer in a given piece of code. In our earlier work, the Origins Model,[7] we demonstrate an "intent-based programming model" for context-aware applications in large-scale pervasive systems. In the Origins Model, an origin is an abstraction of any source of context information. Origins are universal, discoverable, composable, migratable, and replicable components that are associated with type and meta-information. They create an adequate foundation for the development of context-aware applications. Based on them, four processing operations are defined in the Origins Model: filter, infer, aggregate, and compose. As such, these operations provide a powerful mechanism to express a rich set of processing schemes in context-aware applications. Based on the Origins Model, we present the Origins Toolkit—a proof-of-concept implementation developed using the Scala programming language and the Akka toolkit to provide a distributed, scalable, and fault-tolerant solution. Similarly, we propose to use such a mechanism to be utilized for mapping the software requirements and intents of the programmer to hardware architectures, including information related to context awareness.

Example 4: Elasticity Requirements

Elasticity is the property of returning to an initial state or form following deformation. Today we mimic this property in computer systems to create elasticity primitives that enable engineers to attach elasticity properties at multiple abstraction levels to software services, ranging from actual programming code and cloud services to hardware elements such as FPGAs, core-to-core interfaces, etc. We argue[1–5] that elasticity should be engineered in a three-dimensional model, trading off between resources, quality, and costs. In this model, the engineer determines the principal rules of how many and which (1) resources (software services, people) using which (2) quality of input and output (data, performance, etc.) under which (3) costs should be possible. Therefore, it is possible to express many types of elasticity properties as a function that should be performed only if it has certain cost properties and has certain data quality inputs and certain data quality outputs. We developed a whole suite of tools and methods,[1-5] enabling the designer of the system to specify the elasticity requirements by mapping them to constraints and selecting a strategy for how to satisfy and enforce those constraints. We believe such an approach can be extended through the hardware/software interface and conveyed elasticity requirements/properties can enable extremely efficient and flexible management of underlying hardware systems and components.

MANAGING GROWING (SOFTWARE AND HARDWARE) COMPLEXITY

As complexity and scale of hardware and software grow and components become increasingly heterogeneous (at all layers and components), overall system design, and management complexity increases. We believe that the management of such complexity and the underlying resources requires much more expressive and fluid interfaces that can convey information across layers and components. For example, heterogeneous and continuously changing compute resources (e.g., CPUs, GPUs, TPUs, FPGAs, specialized ASIC accelerators, in-memory computation engines, quantum accelerators) and memory resources (multiple types of DRAM, NVM, Flash memory, many levels of caches at every component, etc.) are very difficult to manage and exploit if interfaces are inexpressive and rigid.

We, therefore, posit that we need expressive interfaces for heterogeneity management. In fact, we believe the hardware components will be more fluid in the future and software will effectively

allocate and connect the hardware components needed for its most efficient execution and user-level and system-level goals. We argue that future many-core systems should be designed to support the management of many different metrics (e.g., power, performance, reliability, security, energy efficiency, cost) automatically across competing or cooperating applications. The users and the system can specify service-level agreements (SLAs) for tasks and the runtime system should automatically exploit the underlying asymmetry to satisfy SLAs with minimal power/energy consumption. It is our vision that future runtime systems (with the support of hardware) will automatically decide where to run different tasks (or program portions) and how to exploit/morph asymmetric/configurable hardware to maximize power efficiency while satisfying application requirements.

To enable this, there needs to be much research done on questions such as 1) How should asymmetric components be designed to achieve maximum efficiency? 2) What monitoring should be performed in hardware/software for task characterization, matching to components, and mapping? 3) How should the system software be designed to automatically manage resources based on dynamic demand? 4) What are the interfaces necessary across layers of the stack and across software or hardware components that would enable efficient management of heterogeneity? Given the complexity of these tasks and the need for automatic discovery of task demands, we believe statistical/machine learning techniques should play an important role in future systems. Automatic resource management with hardware/software cooperation will also enable a system that continuously optimizes itself by adapting to dynamic changes in operational environment and workloads.

ROLE OF LEARNING AND CONTINUOUS SELF-OPTIMIZATION AND ADAPTATION

We believe learning, training, and inference will play a key role in managing fluid hardware and software resources. Based on learning, software and hardware can optimize the overall system architecture (hardware and software) for the "problem" that is being solved by the system at any given time, both at the macro-level (user intent, SLAs,

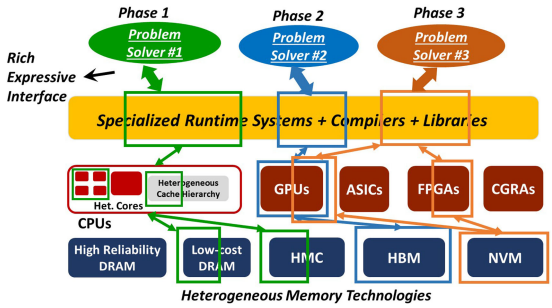

Figure 4. Dynamic "problem solvers" for Phases 1, 2, 3 of a given "problem".

programmer intent, etc.) and the micro-level (efficiency of each hardware components). As a very specialized example from a single, yet extremely important, hardware component, our research shows that a memory controller that is designed using reinforcement learning principles can effectively adapt its control policies to changing workload behavior and system conditions.[20] Such a controller has a much higher performance than a state-of-the-art controller designed based on heuristics chosen by the hardware designer. Many other such opportunities exist to "morph" the hardware resources to continuously changing workload and system characteristics via learning and self optimization.

SPECIALIZED "PROBLEM SOLVERS" FOR HOLISTIC COMPUTING

We argue that the notion of such "problem solvers" (i.e., connected software/hardware components) that are continuously and dynamically generated to match the problem, environment, resources at any given time, is very powerful. This notion can compose different components as well as utilize different computing paradigms with the goal of providing the best-fit system (software+hardware) that satisfies the high-level goals and requirements conveyed by a rich, expressive interface. The interfaces between layers/components and learning mechanisms to compose hardware/software components into specialized "problem solvers," both of which we extensively discussed before, are critical challenges (and opportunities). Such specialization can, perhaps ironically, lead to a more holistic way of designing computing systems and solving problems. Figure 4 depicts an example system cycling through different phases of "problem

solvers" and morphing both the hardware and software components to match the system/user goals/intent at any given phase.

CONCLUSION

We argue that we are at a point in computing where the increasing complexity of systems we create and the energy cost of data movement as well as the need for energy efficiency are concurrently rendering the "divide and conquer" approach to designing systems extremely inefficient and incapable in providing key end-to-end requirements (e.g., energy, performance, predictability, security, latency, and QoS). We provided evidence that the existing interfaces between layers of the computing stack and between different computing components are very limiting due to their inexpressiveness and rigidity. We argue for a new design approach that enables "holistic interfaces" across computing layers and components. Through examples, we show that even simple incarnations of rethinking the abstractions and interfaces between the application, system, architecture, and hardware components can provide significant benefits in improving performance, efficiency, resilience, security, and programmability. We hope that future work builds on this new "holistic interfaces" approach to enable novel ultra-scalable heterogeneous systems with strong end-to-end guarantees and high efficiency.

REFERENCES

1. S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, "Programming directives for elastic computing," *IEEE Internet Comput.*, vol. 16, no. 6, pp. 72–77, Nov./Dec. 2012.
2. D. Moldovan, G. Copil, and S. Dustdar, "Elastic systems: Towards cyber-physical ecosystems of people, processes, and things," *Comput. Standards Interfaces*, vol. 57, pp. 76–82, 2018.
3. G. Copil, D. Moldovan, H. Linh, and S. Dustdar, "Continuous elasticity: Design and operation of elastic systems," *It - Inf. Technol.*, vol. 58, no. 6, pp. 329–348, 2016.
4. G. Copil, D. Moldovan, H. Linh, and S. Dustdar, "rSYBL: A framework for specifying and controlling cloud services elasticity," *Trans. Internet Technol.*, vol. 16, no. 3, pp. 18:1–18:20, 2016.
5. A. Gambi, W. Hummer, H. Linh Truong, and S. Dustdar, "Testing elastic computing systems," *IEEE Internet Comput.*, vol. 17, no. 6, pp. 76–82, Nov./Dec. 2013.
6. S. Nastic, S. Sehic, M. Vögler, H.-L. Truong, and S. Dustdar, "PatRICIA – A novel programming model for IoT applications on cloud platforms," in *Proc. 6th IEEE Int. Conf. Service Oriented Comput. Appl .*, Dec. 16–18, 2013, pp. 53–60.
7. S. Sehic, S. Nastic, M. Vögler, F. Li, and S. Dustdar, "Entity-adaptation: A programming model for development of context-aware applications," in *Proc. 29th Annu. ACM Symp. Appl. Comput.*, Mar. 24–28, 2014, pp. 436–443.
8. N. Vijaykumar *et al.*, "A case for richer cross-layer abstractions: bridging the semantic gap with expressive memory," in *Proc. 45th Int. Symp. Comput. Architecture*, Los Angeles, CA, USA, Jun. 2018.
9. 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 *Proc. 45th Int. Symp. Comput. Archit.*, Jun. 2018, pp. 829–842.
10. Y. Luo *et al.*, "Characterizing application memory error vulnerability to optimize data center cost via heterogeneous-reliability memory," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw .*, Jun. 2014, pp. 467–478.
11. J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-aware intelligent DRAM refresh," in *Proc. 39th Int. Symp. Comput. Archit.*, Jun. 2012, pp. 1–12.
12. Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Proc. 41st Int. Symp. Comput. Archit.*, Jun. 2014, pp. 361–372.
13. D. Lee *et al.*, "Adaptive-latency DRAM: Optimizing DRAM timing for the common-case," in *Proc. 21st Int. Symp. High-Performance Comput. Archit.*, Feb. 2015, pp. 489–501.
14. R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," in *Proc. ISCA*, 2010, pp. 37–47.
15. A. Boroumand *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Operat. Syst.*, Mar. 2018, pp. 316–331.
16. J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. 42nd Int. Symp. Comput. Archit.*, Jun. 2015, pp. 105–117.

17. V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proc. 50th Int. Symp. Microarchit.*, Oct. 2017, pp. 273–287.
18. O. Mutlu, "Memory scaling: A systems architecture perspective," in *Proc. 5th Int. Memory Workshop*, May 2013, pp. 21–25.
19. O. Mutlu, "The RowHammer problem and other issues we may face as memory becomes denser," in *Proc. Des., Automat. Test Eur. Conf.*, Mar. 2017, pp. 1116–1121.
20. E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self optimizing memory controllers: a reinforcement learning approach," in *Proc. 35th Int. Symp. Comput. Archit.*, Jun. 2008, pp. 39–50.

Schahram Dustdar is a Professor of Computer Science heading the Distributed Systems Group, the TU Wien, Vienna, Austria. From 2004–2010, he was Honorary Professor with the Department of Computing Science, the University of Groningen, Groningen, The Netherlands. From 1999–2007, he worked as the cofounder and Chief Scientist with Caramba Labs Software AG, Vienna, Austria. From December 2016 until January 2017, he was a Visiting Professor with the University of Sevilla, Sevilla, Spain, and in 2017, he was a Visiting Professor with UC Berkeley, Berkeley, CA, USA. In summer 2018, he is an MHI Distinguished Visitor, University of Southern California, Center for Cyber-Physical Systems and the Internet of Things USC Viterbi School of Engineering. He is Co-Editor-in-Chief of the new *ACM Transactions on the Internet of Things* as well as Editor-in-Chief of Computing (Springer). He is an Associate Editor of IEEE Transactions on Services Computing, IEEE Transactions on Cloud Computing, *ACM Transactions on the Web*, and *ACM Transactions on Internet Technology*, as well as on the Editorial Board of IEEE Internet Computing and IEEE Computer. He is an IEEE Fellow (2016) for his contributions on Elastic Computing, a recipient of the IEEE TCSVC Outstanding Leadership Award, for Outstanding Leadership in Services Computing (2018), the ACM Distinguished Scientist award (2009), the IBM Faculty Award (2012), an elected member of the Academia Europaea: The Academy of Europe, where he is Chairman of the Informatics Section. For more information see his Webpage at: http://dsg.tuwien.ac.at/Saff/sd. He is the corresponding author of this article. Contact him at dustdar@dsg.tuwien.ac.at.

Onur Mutlu is a Professor of Computer Science with ETH Zurich. He is also a Faculty Member with Carnegie Mellon University, Pittsburgh, PA, USA, where he previously held the Strecker Early Career Professorship. His current broader research interests include computer architecture, systems, security, and bioinformatics. A variety of techniques he, along with his group and collaborators, has invented over the years have influenced industry and have been employed in commercial microprocessors and memory/storage systems. His industrial experience spans starting the Computer Architecture Group, Microsoft Research (2006–2009), and various product and research positions with Intel Corporation, Advanced Micro Devices, VMware, and Google. He received the Ph.D. and M.S. degrees in ECE from the University of Texas at Austin and the B.S. degree in computer engineering and psychology from the University of Michigan, Ann Arbor, MI, USA. His industrial experience spans starting the Computer Architecture Group at Microsoft Research (2006-2009), and various product and research positions at Intel Corporation, Advanced Micro Devices, VMware, and Google. He received the IEEE Computer Society Edward J. McCluskey Technical Achievement Award, ACM SIGARCH Maurice Wilkes Award, the inaugural IEEE Computer Society Young Computer Architect Award, the inaugural Intel Early Career Faculty Award, CMU Ladd Research Award, faculty partnership awards from various companies, a healthy number of best paper or "Top Pick" paper recognitions at various computer systems, architecture, and hardware security venues. He is an ACM Fellow "for contributions to computer architecture research, especially in memory systems", IEEE Fellow for "contributions to computer architecture research and practice", and an elected member of the Academy of Europe (Academia Europaea). His computer architecture course lectures and materials are freely available on YouTube, and his research group makes software artifacts freely available online. For more information, please see his webpage at http://people.inf.ethz.ch/omutlu/. Contact him at omutlu@gmail.com.

Nandita Vijaykumar is an Assistant Professor with the Computer Science Department, the University of Toronto, Toronto, ON, Canada, and the Department of Computer and Mathematical Sciences, the University of Toronto Scarborough, Scarborough, ON, Canada. She received the Ph.D. degree in the electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, USA. Her research interests inlude the general area of computer systems and architecture with a focus on the interaction between programming models, systems, and architectures to improve programmability and portability in achieving high performance. Her industrial experience includes a full-time position at advanced micro devices and internships with Advanced Micro Devices, Microsoft Research, Nvidia Research, and Intel Labs. She received the Bachelor's degree in electrical and electronics engineering from PES Institute of Technology, Bangalore. India. She was the recipient of the Benjamin Garver Lamme/Westinghouse Fellowship. Contact her at nandita@cs.toronto.edu.