A Simulator and Compiler Framework for Agile Hardware-Software Co-design Evaluation and Exploration

Invited Talk

Tyler Sorensen
UC Santa Cruz

Aninda Manocha
Princeton University

Esin Tureci
Princeton University

Marcelo Orenes-Vera
Princeton University

Juan L. Aragón
Universidad de Murcia

Margaret Martonosi
Princeton University

ABSTRACT

As Moore's Law has slowed and Dennard Scaling has ended, architects are increasingly turning to heterogeneous parallelism and hardware-software co-design. These trends present new challenges for simulation-based performance assessments that are central to early-stage architectural exploration. Simulators must be lightweight to support heterogeneous combinations of general-purpose cores and specialized processing units. They must also support agile exploration of hardware-software co-design, i.e. changes in the programming model, compiler, ISA, and specialized hardware.

To meet these challenges, we describe our compiler and simulator pair: DEC++ and MosaicSim. Together, they provide a lightweight, modular simulator for heterogeneous systems, offering accuracy and agility designed specifically for hardware-software co-design explorations. The simulator and corresponding compiler were developed as part of the DECADES project, a multi-team effort to design and tape out a new heterogeneous architecture. We will present two case-studies in important data-science applications where DEC++ and MosaicSim enable straightforward design space explorations for emerging full-stack systems.

CCS CONCEPTS

• **Computing methodologies → Modeling and simulation**; • **Software and its engineering → Compilers**.

KEYWORDS

performance modeling, heterogeneous systems, hardware-software co-design. LLVM simulation

ACM Reference Format:
Tyler Sorensen, Aninda Manocha, Esin Tureci, Marcelo Orenes-Vera, Juan L. Aragón, and Margaret Martonosi. 2020. A Simulator and Compiler Framework for Agile Hardware-Software Co-design Evaluation and Exploration: Invited Talk. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415751

1 INTRODUCTION

At this point, technologists have all seen the obituaries for Moore's Law and Dennard Scaling. Powerful eulogies, in the form of flattening scaling graphs (e.g. see [26]), have been widely shown at various conferences and events. This death was inevitable as our communities pushed transistor technologies to their fundamental limits: transistors now are at the point where they can no longer get smaller and faster. These trends will be dearly missed, as they helped describe romantic pop science facts, e.g. a 2019 smart phone has roughly $100,000\times$ as much processing power as the computer on the Apollo 11 spacecraft that first landed on the moon [14].

When these transistor scaling trends existed, computer architects and programming language (PL) designers could focus on improving the design (and the programmability) of general-purpose computational units (e.g. CPUs). As transistor technology improved, it created opportunities for CPU performance improvement through these designs. However, with the death of transistor scaling trends, room for improvement by architects and PL researchers now lies in *specialization*, i.e. because transistors are not getting faster, we must think about how they can be used more efficiently. To achieve this, problem domains are distilled into abstract building blocks, for which specialized architectures (and PLs) can be designed. Modern architectures, or Systems On a Chip (SoCs), are mosaics of specialized processing units. A microscopic view shows small, specialized processing units patched together while a macroscopic view gives the illusion of a unified system performing one complex task, e.g. taking a picture on a smart phone. Die photo analysis has shown that over half the area of modern smartphone SoCs comprises processing elements that are neither CPUs nor GPUs [28].

Designing the architecture of a *heterogeneous system* is extremely difficult. Individual processing units, e.g. ASIC accelerators, semi-programmable accelerators, data supply components, and general-purpose processors, now have mature tooling for design. Thus, a heterogeneous system design has a buffet of processing units and data supply techniques to choose from; ideally, it employs a combination such that the system is optimally efficient across its problem domains, while also respecting resource constraints. Designing the programming language for a heterogeneous system is also extremely challenging. Legacy software should run efficiently with minimal (preferably no) modifications. Users should be able to target the system with familiar languages (e.g. Python and C++). Finally, the system should provide enough flexibility (or reconfigurability) to adapt to the latest algorithm, model, or input data.

This architectural design space explosion, coupled with ambitious PL goals, necessitates new simulator and compiler tooling for the research community. Such tooling should support an agile architecture design, such that different heterogeneous components can be easily swapped and evaluated. Likewise, the programming model should be agile, such that new code transformations (both automatic and manual) can be rapidly implemented and evaluated. Most importantly, the simulator and compiler should have a straightforward interface for each other, such that hardware-software co-designs can be developed in an agile way.

Many widely used simulators (e.g. Sniper [8], ZSim [27], gem5 [7]) are designed to simulate homogeneous ISA systems; this design choice has several significant drawbacks when modeling heterogeneous systems: (1) Accelerator integration is not straightforward as only a limited set of ASIC designs are currently supported; (2) PL innovations are implemented in an ad-hoc manner, for example, by invasive modifications in LLVM, and (3) Individual computation components can have completely different ISAs, e.g. [5].

At the same time, the PL community has largely rallied behind the LLVM compiler framework [17]. There is mature support for various frontends, including C/C++ through Clang [1], Python through Numba [16], and many others [33]. A program written in a supported frontend is first compiled to LLVM IR bytecode. The compiler can then apply transformations, including custom optimizations written for a hardware-software co-design. After these transformations, the LLVM backend can produce binaries for a variety of microarchitectures, including X86, ARM, and RISC-V. Thus, this framework enables software to flexibly target heterogeneous hardware and should be leveraged by a lightweight, modular simulation infrastructure.

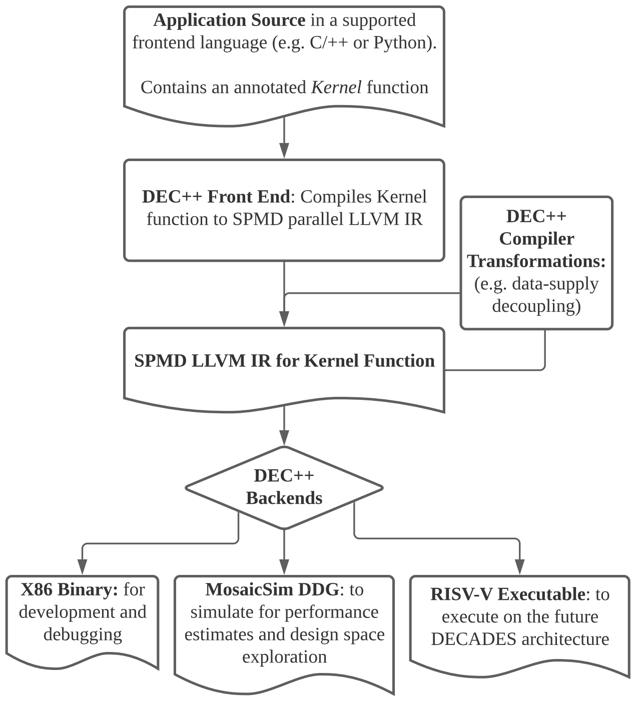
Application Source in a supported frontend language (e.g. C/++ or Python).
Contains an annotated *Kernel* function
DEC++ Front End: Compiles Kernel function to SPMD parallel LLVM IR
DEC++ Compiler Transformations: (e.g. data-supply decoupling)
SPMD LLVM IR for Kernel Function
DEC++ Backends
X86 Binary: for development and debugging
MosaicSim DDG: to simulate for performance estimates and design space exploration
RISC-V Executable: to execute on the future DECADES architecture

Figure 1: The flow of an application through the DEC++ compiler. Because DEC++ is based on LLVM IR, there are several backends that are straightforward to target.

1.1 DEC++ and MosaicSim: a New Compiler and Simulator Infrastructure

This paper describes DEC++ and MosaicSim: a compiler and simulator infrastructure to support agile development of heterogeneous architectures and corresponding programming models. Together, they facilitate early-stage evaluations of hardware-software co-designs by providing interchangeable heterogeneous architecture component models and a robust compiler framework. Our compiler, DEC++, leverages the mature LLVM infrastructure. Our innovations are largely LLVM IR compiler passes that can target code from LLVM's wide variety of frontends, including C/C++ and Python.

Unlike prior simulators, MosaicSim is not tied to a concrete ISA. Instead, it directly simulates LLVM IR. While this can lead to direct instruction-to-instruction mismatches, we have shown that our simulator can accurately characterize critical performance bottlenecks, and capture performance trends, e.g. related to parallel scalability. Simulating LLVM IR provides a straightforward interface to the DEC++ compiler infrastructure, which allows the simulator to target a variety of frontends and support the addition of new instructions (in the form of LLVM functions).

Additionally, our infrastructure is not simply for performance evaluations. The mature backends provided by LLVM allow programs compiled by DEC++ to be further compiled down to an ISA backend and executed as a binary on various architectures. For example, DEC++ can invoke the LLVM x86 backend after compilation to run the application on modern workstations.

Looking forward, DEC++ and MosaicSim were developed for designing a heterogeneous architecture as part of the DECADES project. This design uses the OpenPiton framework with RISC-V Ariane cores [4] to drive accelerator tiles. Because LLVM can target RISC-V, our framework allows a one-stop shop to compile for: (1) x86 emulation; (2) MosaicSim architectural simulation, and (3) execution on a RISC-V based open-source tiled architecture.

This paper is not intended to describe the novel scientific contributions of the DECADES project in either architecture, nor PL. Instead, it gives an overview of our design philosophies, experiences in developing the tools, and use-cases where our tools have provided valuable insights. Related to this work, we have published a technical overview of MosaicSim [21], and given a technical talk on accelerating graph applications using DEC++ and MosaicSim [19].

The paper is organized as follows: Section 2 describes the DEC++ compiler, including provided APIs and automated compiler passes; Section 3 discusses the MosaicSim simulator, including how cores, accelerators, and data supply components are modeled; Section 4 presents two case studies where our tooling has allowed straightforward evaluations of the performance of complex heterogeneous systems; and finally, Sections 5 and 6 discuss related work and conclude, respectively.

2 THE DEC++ COMPILER

Our compiler framework is named DEC++, a name that combines its robust C++ frontend with its development as part of the DECADES project. DEC++ does not implement its own parser or IR; it is a collection of LLVM compiler passes and linker invocations. Because we leverage the mature LLVM framework for its foundational compiler technology, we can focus on novel innovations in programming languages for hardware-software co-designs.

We first describe the programming model supported by DEC++. It varies slightly depending on the frontend, e.g. C++ or Python, but is implicitly parallel, employing a single program, multiple data (SPMD) model. An extensible API targets accelerators and data supply components. We then detail several LLVM transformations that apply data supply optimizations automatically. The section concludes with a discussion on our design approach.

2.1 Programming Model

A DEC++ program consists of two parts: a *kernel* function of interest, and a *wrapper* that calls the kernel. The wrapper is executed sequentially and can perform any operation allowed by its host language and architecture. This includes input-output, data preparation, etc. The kernel function is implicitly SPMD (single-program, multiple-data) parallel, i.e. the same kernel function is executed by many different threads.

A kernel function may use two built-in variables: `num_threads`, which specifies the total number of threads executing the kernel, and `thread_id`, which provides a unique thread id for each thread. The thread id starts at 0 and is consecutive up to `num_threads - 1`. Given these two values, kernels can partition the work amongst the threads. It is best practice to write kernels that are agnostic to number of executing threads. This allows the simulator to more easily explore the design space, as the number of threads and cores in a heterogeneous system can vary. The number of threads is specified at compile-time through a command line flag.

Because DEC++ is built on LLVM, the programming constraints correspond to the constraints of the frontend language. For example, Clang robustly supports C++20 and DEC++ will compile kernels written in arbitrary C++20. The LLVM IR that the kernel is compiled into is largely supported by MosaicSim. This includes nearly all primitive instructions, e.g. memory accesses, arithmetic operations, etc. As we describe later, function calls are either inlined or treated as ISA-extensions to control new architecture components in hardware-software co-designs. We find this support is sufficient to accurately simulate many kernels with very little modifications.

However, the MosaicSim simulator does *not* support all C++20 features and users that expect accurate simulation results need to further constrain their kernels. Specifically, these features are not supported in kernels: file input/output, dynamic memory management (allocation, copy, or free), system calls, and recursion. These are not fundamental constraints and we aim to continue adding support for these operations in MosaicSim as the tools mature and gather more use-cases.

2.2 Frontends: C++ and Python

Here we describe the two frontends we have developed for DEC++: a C/C++ frontend using Clang [1] and a Python frontend using Numba [16]. DEC++ requires parallelism to be provided by the frontend. That is, the compiler passes expect that the kernel is launched with `N` threads and provided with the two built-in variables: `num_threads` and `thread_id`.

C/++ Frontend. As one might expect, DEC++, being based on LLVM, most robustly supports the C/C++ frontend through Clang. The user specifies the kernel function through a special function name: `_kernel_`. This function can take any number (and type) of arguments. It should be called exactly once by the wrapper, and the wrapper can consist of arbitrary C/C++ code. For example, the wrapper can read in a data file and prepare the data in an efficient object. The wrapper can then call the kernel function, passing the objects and other data that the kernel needs.

The last two arguments of the kernel function are reserved for two integer values: `num_threads` and `thread_id`. The program should pass special macro values when calling the kernel. These will be used in the compilation pass to provide SPMD parallelism.

The first stage of the C/C++ DEC++ frontend invokes a Clang Visitor, i.e. a source-to-source transformation. This transformation is responsible for adding parallelism. First, it searches for the `_kernel_` invocation. It then wraps the invocation in a `for` loop with a bound of `N`. The loop is annotated with an OpenMP parallel pragma that launches all iterations in parallel. The final two arguments of the kernel are replaced with the loop iteration (the thread id), and the total number of threads (the loop bound).

After the transformation, Clang is called again to emit LLVM for the program. This uses Clang's OpenMP implementation and creates parallel LLVM IR that the DEC++ LLVM transformations can operate on and further optimize.

Python Frontend. In order to provide data scientists with a more familiar language, DEC++ provides a prototype Python interface. This interface works through Numba: a Python toolkit that allows functions to be annotated with types and then compiled to LLVM IR. The LLVM IR can then be compiled to a native executable using LLVM's backends. Using Numba, we have demonstrated kernel execution speedups ranging from $2 - 250\times$. These improvements attribute to removing the interpreter overhead from the compiled code. For example, extreme speedups can arise from representing irregular graph data-structures as efficient CSR Numpy arrays when compiled with Numba instead of inefficient Python dictionaries when interpreted.

Due to Python's flexibility with passing and returning functions, this frontend does not have the same constraints as the C/++ frontend in terms of calling the kernel function with macro arguments. Users can write a type-annotated Numba kernel function, with the two reserved final integer arguments: `num_threads` and `thread_id`. However, to compile the kernel, the user calls DEC++ through a higher-order function, `DECpp`, which takes as arguments 1) the kernel function and 2) all of its arguments.

The `DECpp` function first wraps the kernel function in a Numba parallel loop[1]. This special Numba construct launches each loop iteration in parallel (implemented using OpenMP). The SPMD variables can then be instantiated in a similar way to the C/++ frontend. This

[1]http://numba.pydata.org/numba-doc/latest/user/parallel.html#explicit-parallel-loops

creates a new parallel kernel function that can then be compiled to LLVM IR using Numba. Next, the DEC++ LLVM transformations are applied to the LLVM IR. The `DECpp` Python function returns an object that can be executed (using Numba's binary backend) or passed to MosaicSim to simulate the kernel.

2.3 Data supply and Accelerator APIs

The DEC++ programming model contains several APIs that can target components of a heterogeneous system. The compiler understands the API and passes the correct hooks to the MosaicSim simulator for performance evaluations. The API also includes software implementations that can efficiently be emulated on workstation machines (e.g. with x86 processors). Thus, applications can be rapidly developed and debugged with efficient native execution. Simulation can be performed later when there is sufficient confidence in application correctness. DEC++ currently supports two APIs: core-to-core communication queues and accelerators. New APIs can straightforwardly be added on the compiler side by providing a software implementation and registering the API calls with the compiler so that it knows to pass them down to the simulator.

Accelerator API. ASIC accelerators are a key component of a heterogeneous system and there is a rich set of processing units that a system could incorporate. Common examples are matrix multiplication and fast Fourier transform. As part of the DECADES project, we are working closely with the ESP team at Columbia [20] to actively add APIs for the accelerators they support. These accelerators are simulated using RTL-derived performance models (provided by the ESP team) to be used by MosaicSim. The team has recently reported supporting a sufficient set of accelerators to perform state-of-the-art embedded ML inference tasks, which we aim to support in the near future [11].

Core-to-Core Communication API. A key research goal for the DECADES project is to explore efficient data supply and communication designs. Recently, there has been a resurgence of classic Decoupled Access Execute (DAE) [31] designs, such as DeSC [12], and DSWP [23]. These approaches require core-to-core communication through specialized hardware queues. The DeSC scheme additionally requires a specialized hardware unit to perform asynchronous memory loads.

To support such research, we have developed a core-to-core message passing API that programmers can use for fine-grained communication between cores. This API consists of `send`, `recv`, and `async_load` (for latency tolerance) commands for various bitwidths. To support software emulation, we have provided a software implementation in C++ which uses non-blocking circular buffers as communication queues between threads.

2.4 Compiler Passes

While a user is free to use the DEC++ APIs to write applications, DEC++ can also use LLVM compiler passes to automate several data supply transformations that have been previously published. To use these transformations, the user does not need to use the communication API in their program. Instead, the compiler can transform the code to use the communication API in a data supply scheme. We discuss three such transformations now.

Decoupled Access Execute (DAE). The classic DAE [31] scheme slices a sequential program into a parallel pair of instruction streams: the Access and the Execute. The Access is responsible for all the memory accesses in the program; it only performs the computation required for the sequential stream of memory loads and stores in the program. On the other hand, the Execute slice does not touch memory and performs value computation. The Access loads values and communicates them to the Execute, while the Execute performs value computation and sends values back to the Access to store to main memory. This scheme can provide latency tolerance by overlapping memory accesses with computation.

DEC++ provides a DAE scheme straightforwardly as an LLVM pass. First the kernel function is copied to Access and Execute kernels. Next, the Access is transformed to follow every load with a communication to the Execute. Stores are transformed to remove the value and replace it with a communication from the Execute. The Execute is transformed to remove all loads with a `recv` from the Access. Stores on Execute are replaced with a `send` to the Access.

After the memory access transformations are applied, we leverage the LLVM framework to clean up all unnecessary instructions by doing a dead-code elimination. This removes value computation from the Access because the resulting value is never used (it is replaced with a communication from the Execute).

Decoupled Supply-Compute (DeSC). A more recent implementation of DAE, called DeSC [12], exploits the insight that many loaded values are not actually needed on the Access slice. Thus, significant performance gains can be achieved if the hardware is able to asynchronously load a value and send the resulting data to the Execute from the Access. This allows the Access to not have to stall waiting for the memory hierarchy to return the data that is loaded. Such loads are called *terminal loads*.

We have provided a DeSC transformation as part of DEC++. The implementation builds directly on top of our DAE implementation. It requires only one additional pass on the Access slice to determine if loaded values are ever used on the Access slice apart from communicating them to the Execute. If not, they can be transformed into terminal loads, using the `async_load` API call.

2.5 DEC++ Design Choice Discussion

Here we briefly discuss design decisions that we found especially useful, and those we found to cause difficulty. Overall, we are optimistic about our design, and given the active, well-supported LLVM community, we believe our difficulties will reduce over time.

Design Choice Positives. The most obvious positive of our design choice is being based on LLVM. The community is well-supported and active. Being built on LLVM allowed us to instantly plug into a variety of frontends. Our most widely used frontend is C/C++ using Clang. To enable a more familiar language for data-scientists, we were able to provide a Python prototype frontend through Numba. According to the LLVM Wikipedia page, frontends exist for many more languages, spanning many domains and programming idioms. This list includes Rust, Fortran, and Haskell [33]. While a DEC++ frontend would not be supported out-of-the-box, e.g. because of our programming model constraints described in Section 2.1, it would be significantly easier than starting from scratch.

Because our compiler passes are all built in a way that does not require any LLVM source code changes, DEC++ can be built alongside existing installations of LLVM. We do not have to host, and maintain, a large custom LLVM repository, which would require manually merging updates to stay up-to-date.

Finally, because of LLVM's well-supported backends, DEC++ can not only target the MosaicSim simulator (described next), but also compile to binaries for a variety of targets. For example, compiling to x86 can create binary applications that natively execute on modern workstations. This allows for rapid development and debugging (although new hardware features are emulated in software). Finally, the DECADES architecture plans to have compute cores that execute the RISC-V ISA. Because modern versions of LLVM support RISC-V, DEC++ can be used as not only a compiler for development and design exploration, but also for producing binaries for the new DECADES architecture.

Design Choice Difficulties. By far the biggest difficulty in the DEC++ design is that LLVM (and the associated tools) have a large learning curve. Developers (or students) who have not been exposed to LLVM may have significant difficulties understanding how passes are written, the nuances of LLVM IR instructions, etc. It is not just the LLVM passes that are complicated; we have found that different frontends, e.g. Numba for Python, are also complex and evolve rapidly. We hope that our suite of passes and frontends can help newcomers get started, and the lively online LLVM community can provide additional support as needed. We believe the learning curve of LLVM tools is countered by their robustness.

The second difficulty is keeping the tools up-to-date. LLVM releases major new versions roughly twice a year. Each new version release typically slightly changes the compiler pass API. Because of this, it is difficult for DEC++ to support multiple versions of LLVM and each update requires the passes to be updated with the new API. To account for this, we have not updated DEC++ with every LLVM release; our trend has been to support every other one. We have updated DEC++ twice now and each update takes around one day of engineering effort. We hope to provide more frequent updates (and possibly support for multiple versions of LLVM) in the future as the tools gain more interest.

3 THE MOSAICSIM SIMULATOR

MosaicSim is a cycle-driven simulator for agile performance evaluation of heterogeneous systems. It works with DEC++ in that it does not simulate a concrete ISA. Instead, it simulates LLVM IR. This design choice allows close coupling with the compiler for hardware-software co-designs, and the ability to simulate designs before committing to low-level details, such as concrete ISAs. A more detailed manuscript was published at ISPASS 2020 [21].

3.1 Design Overview

MosaicSim simulates a heterogeneous system that is made up of tiles, which can either be cores or accelerators. Communication between tiles can occur through a traditional memory hierarchy or through inter-tile communication queues. The simulator takes cycle-level steps; at each step, each tile is activated for exactly one cycle. At this stage, users can specify the depth (one, two, or three cache levels) and parameters (latency, bandwidth) of the memory hierarchy. Users can also specify how many tiles and if they are compute cores (e.g. CPUs) or ASIC accelerators.

To simulate an application, MosaicSim requires an execution trace, which details memory accesses, control-flow decisions, and accelerator arguments. This is achieved through an LLVM pass that annotates instructions of interest in the kernel with logging calls. The transformed LLVM IR can then be compiled to a native binary, e.g. for x86, and run to collect the required trace.

3.2 Core Modeling

To simulate a compute core, MosaicSim first collects a static Data-Dependency Graph (DDG) of the kernel. Because MosaicSim's parallel model is SPMD, the DDG for the single kernel file is replicated and sent to all tiles instantiated as a compute core. From the entry point of the DDG, MosaicSim sees a frontier of active nodes, i.e. instructions that have no unmet dependencies. Each cycle, a set of these frontier nodes are executed and retired, which allows a new frontier to be computed. Nodes that influence control flow query the execution trace to determine their next step. Nodes that are memory accesses query the execution trace to determine the address. A request is then sent to the memory hierarchy model, which returns the latency (number of cycles) incurred by the access.

Depending on the type of instruction, frontier nodes can be placed in different data structures to model different architectural features. Memory access nodes that have been issued, but not committed, can be placed in a Load Store Queue (LSQ) object. The simulated RoB is sliding window across frontier nodes that can issue younger nodes even if older ones have not fully committed. Together, these features provide sufficient capability to model complex out-of-order cores, as well as simple in-order cores.

3.3 ASIC Accelerator Modeling

To model ASIC accelerator designs, the programmer must use the ASIC accelerator API given in the programming model (see Section 2.1). The API calls are registered with the DEC++ compiler, which maintains the API function calls in the kernel. These function calls become nodes in the DDG, which the simulator can call into accelerator timing models.

Our current accelerator performance models come from the ESP design team [20]. They have provided closed-form performance models for a variety of accelerators, including matrix multiplication and matrix element-wise operations which are commonly used in machine-learning applications (e.g. ReLU). The concrete arguments for each accelerator call are recorded similar to the memory trace. Most importantly, these arguments include data dimension sizes, which are the crucial parameters in accelerator performance models. An accelerator performance model can simply take these argument traces and return the number of cycles required for the accelerator invocation. More details on this modeling (including accuracy measurements) can be found in the conference publication [21].

3.4 Inter-Core Communication Modeling

To explore data supply designs, the simulator understands the programming model's data-communication API (see Section 2.1). The compiler maintains kernel calls to this API and then the simulator checks DDG nodes to see if they correspond to an API call. MosaicSim models a producer/consumer queue between every pair of cores. These queues are blocking; a producer that tries to enqueue to a full queue must wait until the consumer dequeues a value. Similarly, a consumer that dequeues from an empty queue must wait until a producer enqueues a value. To support asynchronous memory accesses, the simulator provides an extra memory request unit that can asynchronously issue a memory access and buffer the value in a queue at the point where it was requested. This can simulate the terminal load buffer hardware features of DeSC [12].

The designer can explore various resource constraints by placing limits on the queue sizes and changing the read/write latencies. Prior designs, e.g. DeSC, have modeled queue latencies to be extremely short, e.g. similar to an L1 cache. Other designs may not be able to place queues so close to the core, and latency may have to be increased. For example, a tiled architecture, e.g. DECADES, may have dedicated storage tiles as heterogeneous components, which leads to significantly increased access latency, as requests must go through the NoC.

3.5 MosaicSim Design Choice Discussion

Much like Section 2.5, here we briefly describe MosaicSim design choices that we found useful as well as ones that we found difficult. Similar to DEC++, we believe the positives outweigh the negatives and that a simulator for LLVM-IR is a powerful tool for agile heterogeneous system designs.

Design Choice Positives. Being based on LLVM, MosaicSim could evaluate architectural designs early for the DECADES project, even before a concrete ISA was implemented. Moreover, recent research suggests the rise of heterogeneous systems with cores that mix ISAs, a simulation challenge that MosaicSim completely avoids. Without needing to focus on cycle-level accuracies, we were able to focus on high-level application characterizations. That is, we showed that MosaicSim can accurately classify applications as memory-latency bound, bandwidth-bound, or compute-bound. We could then focus on developing hardware-software co-designs to alleviate these bottlenecks, e.g. [19] explores accelerating data supply for graph applications. Our experience is that the LLVM IR level is a sweet spot for architecture and PL researchers, as they can accurately characterize applications and explore designs without committing to low-level details, such as ISAs.

Our overall design is efficient, with a simulation speed of 0.5 MIPS (millions of instructions per second). This is comparable to the Sniper simulator [8] (also reported at 0.5 MIPS). MosaicSim is faster than gem5 [7] (0.05 MIPS), but is not as detailed, e.g. for coherence models. Future work could add optimizations for instruction-driven simulation and parallel simulation execution, which could significantly increase simulation speed, e.g. similar to ZSim [27].

Design Choice Negatives. The tradeoff of simulating LLVM IR is cycle-level inaccuracies in simulation results. Our conference publication reports up to a 3× difference when simulating an Intel X86 Xeon processor compared to detailed native profiling. Diving deeper, these inaccuracies attribute to ISA mismatches. For example, LLVM requires two instructions for an offset load, `getelementpointer` and *load*, while x86 requires just one: `MOV`. These mismatches could be tuned for a given ISA, e.g. by ignoring certain LLVM IR instructions, but this defeats the purpose of agile and early-stage development and exploration of heterogeneous architectures.

Microarch. Parameter	**Out-of-Order**	**In-Order**
Issue Width	4	1
Instruction Window/RoB/LSQ	128/128/128	1/1/1
Frequency/Tech	2GHz/22nm	2GHz/22nm
Core Count	1	8
Core Area mm^2	8.44	1.01

Table 1: Microarchitecture parameters of the simulated OoO and InO cores. We make an equal-area comparison based on the aggregated area of all cores.

4 CASE-STUDIES

Here we detail two case-studies of how our tooling has enabled us to perform early-stage evaluations of heterogeneous system designs for different application domains. These use cases are drawn from our previous publication with permission from the authors. We are able to provide a more holistic context in this manuscript, as we can tie our motivation to the overarching DECADES project.

As mentioned previously, the DECADES core architecture is being developed and implemented by the research group who designed OpenPiton [6]. They have a robust infrastructure to support a manycore system based on Ariane RISC-V cores [34]. When evaluating performance, one of the most interesting evaluations is to compare the small, energy efficient, in-order (InO) RISC-V cores to state-of-the-art out-of-order (OoO) cores, e.g. Intel Xeon.

Our investigations have so far focused on hardware-software co-designs that enable a heterogeneous system based on these slim in-order cores to compete against beefy OoO for select data-science applications. Because modern OoO cores can have wide issue-widths (allowing multiple instructions to execute at a time) and complex hardware to tolerate latencies, e.g. LSQs and RoBs, we perform an *equal area* performance comparison between OoO cores and IO cores. We have determined, with input from the OpenPiton team and the McPAT tool [18], that one OoO core is roughly the same area as 8 IO cores. We give the details in Table 1.

We present two examples: one where decoupled access/execute latency tolerance schemes can yield significant performance improvements for graph analytics running on in-order cores and one where an ASIC accelerator tile can help latency tolerance for applications that have both dense and sparse linear algebra routines.

4.1 Latency Tolerance in Graph Analytics

Graph analytic kernels are becoming increasingly important for data science. Graphs are used to represent social networks, consumer relations to products, communication networks, etc. These graphs are rapidly growing in size. For example, the number of active Instagram users has nearly doubled every year since 2012 [9]. However, the ability to process such massive social network graphs has not kept up with dataset trends. For example, the Graph500 reports that the top single-node performer (machine and software) on the Breadth First Search (BFS) algorithm has remained the same since 2016 [2]. Because of this disparity, we investigated graph

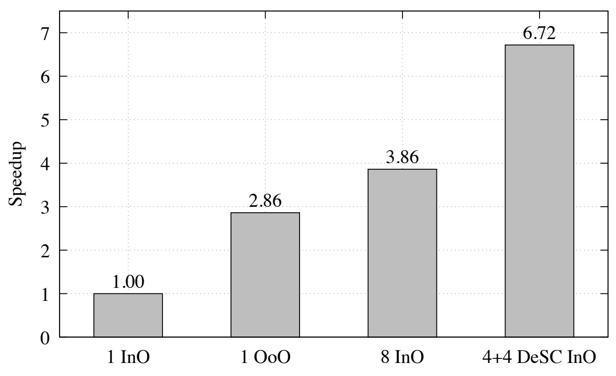

Figure 2: Speedups for the graph projections application. 1 InO core as a baseline; 1 OoO core is representative of state-of-the-art modern cores; 8 InO is a homogeneous parallel system that is area-equivalent to the OoO core; and 4+4 DeSC InO configuration shows an decoupled data-supply approach that is area equivalent to the OoO core. Our results show that in-order systems can be more performant that OoO cores if enough of them are used, and especially if heterogeneous data-supply approaches are implemented.

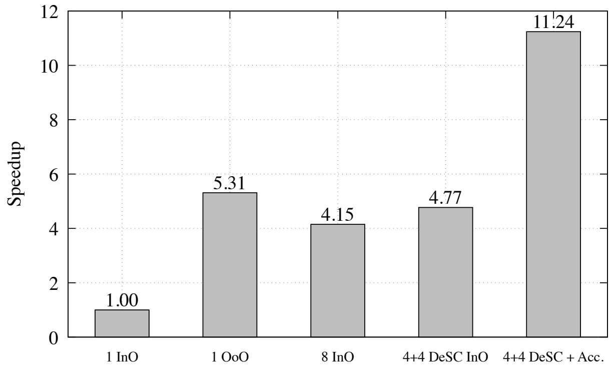

Figure 3: Speedups for the Sinkhorn Distances application. The configurations are the same as Figure 2, with the addition of an accelerator tile for the matrix multiplication half of the application. This application shows that data-supply optimizations and accelerators can cooperate in a heterogeneous system to outperform modern OoO systems.

applications as one of our first hardware-software co-designs using DEC++ and MosaicSim.

Graph Projection. As a starting point, we examined the graph projections algorithm. This application takes a bipartite graph with nodes in one partition, X, connected to nodes in a disjoint partition, Y. The application relates nodes in Y based on common neighbors in X. This application is used, e.g. in recommendation systems, where X might be a set of consumers and Y might be a set of products. This application allows products to be related to one another based on relations through different consumers.

We developed the application in C++ and through DEC++, debugged using native compilation to an X86 binary that we could execute and examine on a workstation. After confidence that the application was correct, we utilized the DEC++ compilation mode to target MosaicSim, where we could characterize the application performance on a single in-order processor. We found that the main bottleneck was the memory latency from irregular memory accesses that occur when nodes access neighbors, which can be scattered seemingly randomly through the graph.

To explore several design choices, we compiled and ran the application for several systems: 1 InO core as a baseline, 1 OoO as representative of a state-of-the-art modern processor, 8 homogeneous parallel InO cores to compare against an OoO core in equal area, and 4 pairs of Access/Execute InO cores using DeSC [12] terminal load optimizations. Figure 2 presents these results. While an OoO core significantly outperforms a single IO core, in an equal area comparison, homogeneous InO parallelism slightly outperforms an OoO core and a DeSC-style decoupling approach is significantly better. This shows that non-invasive hardware-software co-designs for InO architectures are an attractive target for accelerating certain graph applications. Our current work is extending this decoupled data supply for more involved graph applications, e.g. breadth-first search, for which DeSC is not efficient due to complex data-dependencies between the Access and Execute.

4.2 Sparse and Dense Linear Algebra Routines

Some applications, when viewed holistically, have more complex characterizations than compute-bound vs. memory-bound. As an example, we examined Sinkhorn Distances [10], an algorithm for solving the optimal transportation problem that is used in computer vision [25] and NLP [15]. When characterizing this application, we found that there were two bottlenecks: one in a classic dense matrix multiplication and one in a linear algebra routine that sparsely samples a dense matrix.

We found that the sparse linear algebra routine could be accelerated the same way as the graph application, i.e. with DeSC-style latency tolerance. However, this is not useful for the dense matrix multiplication. Luckily, matrix multiplication is a prime kernel for an accelerator tile. We used the accelerator API in DEC++ and then configured MosaicSim to use a matrix multiplication accelerator tile. This yielded 4 systems to evaluate: 1 InO as a baseline, 1 OoO core to represent modern state-of-the-art, 8 InOs for homogeneous parallelism, 4 InO DeSC decoupled pairs, and 4 InO DeSC decoupled pairs with a matrix multiplication accelerator. Figure 3 presents these comparisons. With more heterogeneity and software-hardware co-designs, performance improves. Therefore, coupling the modular design of MosaicSim (different cores and tiles can easily be modeled) with the DEC++ compiler (automatic decoupling techniques) allows for evaluations of such complex heterogeneous systems.

5 RELATED WORK

As mentioned throughout, there is a long history of academic simulators in the research community, e.g. gem5 [7], Sniper [8], Graphite [22], ZSim [27]. Each of these simulators fills a certain role and their track record has shown them to be valuable assets to the community. However, we believe DEC++ and MosaicSim are uniquely positioned in the post-Moore's Law era of architecture and PL research. Prior simulators require commitment to low-level details, e.g. a single ISA, and only support homogeneous parallelism. Furthermore, they are not coupled with a compiler, which requires ad hoc development on both the simulator and programming model. This is difficult to both develop and maintain.

Heterogeneous simulators have been built on top of classic homogeneous simulators. Notably, a collaboration between gem5 and the Aladdin accelerator toolkit [29] has produced gem5-Aladdin [30]. This work has been used to evaluate several data supply schemes between compute cores and accelerators. MosaicSim distinguishes itself with a more abstract accelerator interface and simulations at a higher-level, which allows for an order of magnitude more efficient simulation. Another gem5-based work uses LLVM dependency graphs to model accelerators for a simulator environment [24]. MosaicSim uses similar ideas, e.g. simulating dependency graphs, but has extended the functionality to model important components of out-of-order cores, such as LSQs and RoBs.

While there exist many academic simulators, LLVM has become the de facto standard for PL research, especially related to hardware-software co-design. Two recent decoupled data supply works were built upon LLVM: DeSC [12] and DSWP [23]. Other works leverage the framework to provide load hoisting [32] and prefetching via software hooks automatically added by LLVM passes [3]. Binary instrumentation, e.g. PIN, has been used as an alternative to LLVM, but again commits the user to a specific ISA.

6 FUTURE WORK AND CONCLUSION

DEC++ and MosaicSim are at the beginning of a long and fruitful life. Over the last two years, we have laid the groundwork for a compiler/simulator infrastructure built with agile development and hardware-software co-design as first class principles. Built upon active and well-supported LLVM tools, our infrastructure allows: (1) our framework to be extended with new frontends; (2) programming model innovations to be rapidly developed; and (3) compilation support for a variety of ISAs, which can natively execute on the buffet of modern architectures, including RISC-V. Our MosaicSim simulator accurately characterizes applications, allowing subsequent development of hardware-software co-designs for performance acceleration. DEC++ and MosaicSim are design evaluation tools built for the new golden age of computer architecture [13], and we hope to cultivate a vibrant community around these tools and ideas.

ACKNOWLEDGEMENTS

The authors thank the DECADES team, especially the SLD group led by Luca Carloni at Columbia and the Princeton Parallel Group led by David Wentzlaff at Princeton University. In particular, we thank Davide Giri for providing ESP accelerator performance models. We thank Opeoluwa Matthews, who was the lead designer and author of MosaicSim (and has since moved to an industry job at Apple). We thank Tae Jun Ham as a past contributor to MosaicSim and as a consultant on some of our data supply features (e.g. DeSC). This work was supported in part by the DARPA SDH Program under agreement No. FA8650-18-2-7862. This research was funded in part by the U.S. Government. Prof. Aragón has been partially supported by the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU). The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] [n.d.]. Clang: a C language family frontend for LLVM. http://clang.llvm.org/.
[2] 2019. Graph500: Top from June 2019 BFS. https://graph500.org/ Retrieved Aug. 2019.
[3] S. Ainsworth and T. M. Jones. 2017. Software prefetching for indirect memory accesses. In *The International Symposium on Code Generation and Optimization (CGO)*. 305–317.
[4] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. 2019. OpenPiton+ Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores. In *Third Workshop on Computer Architecture Research with RISC-V, CARRV*, Vol. 19.
[5] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. 2020. BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*. ACM.
[6] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. In *ASPLOS*. ACM, 217–232.
[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.
[8] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *SC*. ACM, Article 52.
[9] Josh Constine. 2018. Instagram hits 1 billion monthly users, up from 800M in September. https://techcrunch.com/2018/06/20/instagram-1-billion-users/ Retrieved Aug. 2019.
[10] Marco Cuturi. 2013. Sinkhorn Distances: Lightspeed computation of optimal transport. *Advances in Neural Info. Proc. Sys.* 26 (2013).
[11] Davide Giri, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca P. Carloni. 2020. ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*. IEEE.
[12] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled Supply-compute Communication Management for Heterogeneous Architectures. In *MICRO*. ACM.
[13] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (2019), 48–60.
[14] Graham Kendall. 2019. Apollo 11 anniversary: Could an iPhone fly me to the moon? *The Independent* (Jul 2019). https://www.independent.co.uk/news/science/apollo-11-moon-landing-mobile-phones-smartphone-iphone-a8988351.html
[15] Matt J. Kusner, Yu Sun 0020, Nicholas I. Kolkin, and Kilian Q. Weinberger. 2015. From Word Embeddings To Document Distances. In *ICML (JMLR Workshop and Conference Proceedings, Vol. 37)*. 957–966.
[16] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM.
[17] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Press.
[18] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*. IEEE/ACM.
[19] Aninda Manocha. 2019. Hardware-Software Co-Design for Efficient Graph Application Computations on Emerging Architectures. Technical talk at FOSDEM'19 https://fosdem.org/2020/schedule/event/graph_hardware_software_co_design/.
[20] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC Development with Open ESP. In *International Conference on Computer-Aided Design (ICCAD), Special Session*. ACM.
[21] Opeoluwa Matthews, Aninda Manocha, Davide Giri, Marcelo Orenes Vera, Esin Tureci, Tyler Sorensen, Tae Jun Ham, Juan L. Aragón, Luca Carloni, and Margaret Martonosi. 2020. MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems. In *Proceedings of the 2020 IEEE International Symposium on*

Performance Analysis of Systems and Software (ISPASS). 13.

[22] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *HPCA*. IEEE Press.

[23] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. http://dx.doi.org/10.1109/PACT.2004.14

[24] Samuel Rogers, Joshua Slycord, Ronak Raheja, and Hamed Tabkhi. 2019. Scalable LLVM-Based Accelerator Modeling in gem5. In *IEEE. Computer Architecture Letters*, Vol. 18.

[25] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. 1997. The Earth Mover's Distance, Multi-Dimensional Scaling, and Color-Based Image Retrieval. In *Image Understanding Workshop*. 661–668.

[26] Karl Rupp. 2015. 40 Years of Mircroprocessor Trend Data. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data.

[27] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *ISCA*. ACM.

[28] Yakun Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2015. The Aladdin Approach to Accelerator Design and Modeling. *IEEE Micro* 35 (06 2015), 1–1. https://doi.org/10.1109/MM.2015.50

[29] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *ISCA*. ACM.

[30] Yakun Sophia Shao, Sam L. Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. 2016. Co-designing accelerators and SoC interfaces using gem5-Aladdin. *IEEE Micro* (2016), 1–12. https://doi.org/10.1109/MICRO.2016.7783751

[31] James E Smith. 1982. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 10. IEEE Press.

[32] Kim-Anh Tran, Trevor E Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2017. Clairvoyance: look-ahead compile-time scheduling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 171–184.

[33] Wikipedia. 2020. LLVM:Front ends — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=LLVM#Front_ends. [Online; accessed 12-August-2020].

[34] Florian Zaruba and Luca Benini. 2018. Ariane: An Open-Source 64-bit RISC-V Application Class Processor and latest Improvements. Technical talk at the RISC-V Workshop https://www.youtube.com/watch?v=8HpvRNh0ux4.