MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems

Opeoluwa Matthews* Aninda Manocha* Davide Giri† Marcelo Orenes-Vera* Esin Tureci*
Tyler Sorensen*‡ Tae Jun Ham§ Juan L. Aragón¶ Luca P. Carloni† Margaret Martonosi*

*Princeton University †Columbia University ‡UC Santa Cruz §Seoul National University ¶University of Murcia

***Abstract*—As Moore's Law has slowed and Dennard Scaling has ended, architects are increasingly turning to heterogeneous parallelism and domain-specific hardware-software co-designs. These trends present new challenges for simulation-based performance assessments that are central to early-stage architectural exploration. Simulators must be lightweight to support rich 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 introduce MosaicSim, a lightweight, modular simulator for heterogeneous systems, offering accuracy and agility designed specifically for hardware-software co-design explorations. By integrating the LLVM toolchain, MosaicSim enables efficient modeling of instruction dependencies and flexible additions across the stack. Its modularity also allows the composition and integration of different hardware components. We first demonstrate that MosaicSim captures architectural bottlenecks in applications, and accurately models both scaling trends in a multicore setting and accelerator behavior. We then present two case-studies where MosaicSim enables straightforward design space explorations for emerging systems, i.e. data science application acceleration and heterogeneous parallel architectures.

***Index Terms*—heterogeneity, hardware-software co-design, performance modeling, multi-core architectures, accelerators**

I. Introduction

The last decade has seen a trend of increasing parallelism as a response to the ending of Moore's Law and Dennard scaling. Figure 1 presents microprocessor trends over the past few decades. As computing frequency (red triangles) has plateaued, the number of logical cores (blue squares) has increased. The stagnation in raw computing frequency has also triggered the usage of specialized systems, including heterogeneous architectures and hardware-software co-design, to meet the demands of today's aggressive performance and power goals. Designers of modern systems are therefore employing combinations of distinct computation elements, including small, low-power cores and high-performing hardware accelerators [1–3].

In their Turing award lecture, Hennessy and Patterson describe a New Golden Age for Computer Architecture, where future performance improvement opportunities encourage vertically-integrated system designs [4]. Such systems require innovation in programming models, compilers, specialized hardware, and ISAs. Hence, the system design space has seen a Cambrian explosion in diversity at all levels of the stack, necessitating flexible tools for design space exploration.

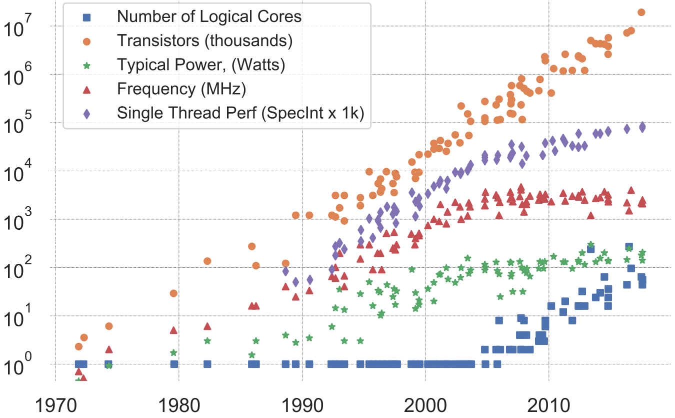
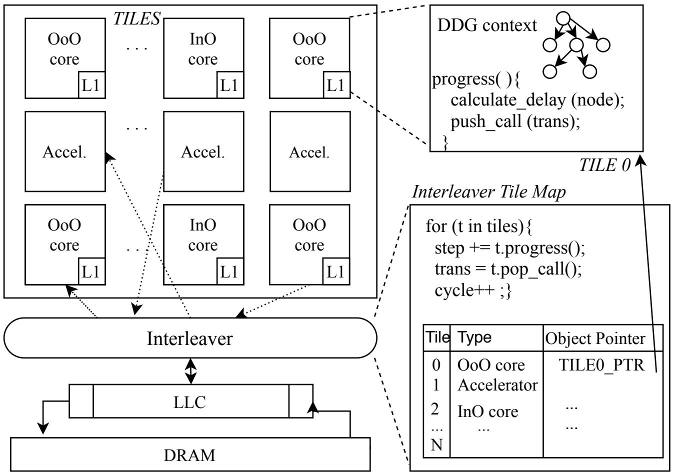


Fig. 1. 42 years of trend data for microprocessor characteristics, graph recreated using data from [7]

Well-known simulators, e.g. gem5 [5], offer detailed simulation infrastructures for conventional microarchitectures, but make it difficult for a designer to explore changes across other layers of the stack, which have increasing influence in the performance of systems today. Other approaches resort to high-level simulation (e.g. 1-IPC models or interval simulation [6]) that do not accurately capture critical memory bottlenecks of many modern data-intensive applications.

We present MosaicSim, a simulation approach that allows for the exploration of optimizations across the hardware-software stack, while providing accurate modeling of performance bottlenecks and application characteristics. To achieve these goals, we leverage the LLVM framework [8], which allows us to utilize a mature compiler infrastructure to capture instruction dependencies and collect memory traces. MosaicSim executes LLVM IR, which enables ISA-agnostic simulation and supports flexible programming models through compiler passes and specialized instructions. MosaicSim then simulates the LLVM IR instructions on modular tile models, which enables straightforward design space exploration of heterogeneous systems. Furthermore, tile modules support a flexible communication model, which allows data-supply hardware-software co-designs to be evaluated.



```
progress( ){
  calculate_delay (node);
  push_call (trans);
}
```

```
for (t in tiles){
  step += t.progress();
  trans = t.pop_call();
  cycle++ ;}
```

Tile	Type	Object Pointer
0	OoO core	TILE0_PTR
1	Accelerator	
2	InO core	...
...	...	...
N		

Fig. 2. MosaicSim integrates different modules (e.g. CPUs and accelerators) via the Interleaver, which combines module behaviors into system-wide performance estimates.

To summarize, MosaicSim is a lightweight, modular simulator for heterogeneous and hardware-software co-design systems. Its main contributions are:

- Enabling flexible programming models, compiler techniques and ISAs through integration with LLVM.
- Abstract tile models, which capture key performance microarchitectural details for a variety of core models and accelerators.
- A holistic simulation approach that allows for the composition of different hardware tiles and communication structures, allowing for the simulation of complex heterogeneous systems.

We evaluate MosaicSim and demonstrate that it:

- Accurately captures trends on existing parallel architectures and accelerators (Section VI).
- Is able to simulate complex heterogeneous systems, illustrated through three case studies (Section VII).

II. MosaicSim Overview

This section describes an overview of the MosaicSim simulation methodology. At a high level, MosaicSim provides tile-based models of different hardware units, including cores, accelerators, and caches, with an *Interleaver* that composes their behaviors to provide total system estimates.

Tiled System Model: Figure 2 displays the tiled system model in MosaicSim. The overall design represents an SoC comprised of CPU and accelerator tiles. Each tile in the SoC has a model of its events that contribute to the performance and power of the entire system, and the Interleaver coordinates the interactions of events from different tiles. MosaicSim simulates a simple homogeneous chip multiprocessor by instantiating several CPU tiles and allowing the Interleaver to coordinate their interactions, e.g. coordinating shared memory hierarchy behavior. Additionally, MosaicSim can simulate more heterogeneous processors by providing (and hence, interleaving) more diverse models. As a design process evolves, accelerators or other specialized hardware can be incorporated as Section IV describes. The direct linkage with an LLVM-based compiler allows straightforward decomposition into models for different units.

As an early-stage tool that targets design space exploration for hardware-software co-design, MosaicSim focuses on kernel simulation. This allows for modeling compute or memory bottlenecks in order to provide hardware designers with the necessary insight to make design decisions (e.g. employing accelerators) accordingly. MosaicSim is not restricted to kernel modeling and can simulate arbitrary codes as long as LLVM-IR can be obtained. However, full application simulation requires performance models that are often only available in later design stages, e.g. filesystem I/O and system calls.

Timing Integration: Distinct tiles may use different notions of execution timing and are modeled to operate concurrently. The Interleaver queries tiles to advance them through the next time unit of execution. Tiles may run at different clock speeds, so the Interleaver queries and coordinates their events accordingly. To communicate, tiles create inter-tile events and enqueue them for the Interleaver to manage. The Interleaver is then responsible for sending a transaction to its destination tile at the right time; it does so by explicitly invoking a destination tile to receive and process message events.

Compiler and Software-Hardware Interface: MosaicSim uses LLVM IR as its ISA, so it is closely integrated with a mature and open-source compiler framework. Wide support for LLVM frontends allows the compiler to take inputs from a variety of languages. While MosaicSim's most developed front-end is C/C++ through Clang [9], we also have prototype support for Python (via Numba [10]) and performance modeling for TensorFlow Keras [11]. The compiler allows further programmer directives to guide hardware components to simulate. For example, the programmer can utilize an accelerator API with common functions (e.g. matrix multiplication) to invoke an accelerator model for specific compute tasks, thus allowing the exploration of design performance trade-offs. New instructions, programming paradigms, and pragmas can be straightforwardly added as functions calls identified through LLVM passes. Relevant parameters can then be relayed to the simulator through traces. The compiler generates dependency graphs of LLVM IR that the simulator can map onto distinct tiles or analyze for lightweight performance estimation.

A. Lightweight Tile Models using Dependence Graphs

Tile models begin as abstract models based on data dependence graphs derived from LLVM IR. Namely, from a full software application written in a compatible language, the compiler can identify kernels for which to perform dependence analysis to create a graph-based model. Section III describes how such models can account for different hardware characteristics to reflect issue width, in- or out-of-order execution, and other processor attributes.

Execution Modeling: In graph-based tile models, a *node* corresponds to a static operation (instruction) and keeps track of its dependents and parents[1]. Dependence analysis is performed to identify basic blocks, which are single-entry, single-exit collections of static instructions in LLVM [12]. Each basic block can have many dynamic instances (e.g. when a basic block is executed repeatedly in a `for` loop), so we call such instances *Dynamic Basic Blocks* (DBBs). In a `for` loop, the basic block remains the same each iteration, but the iteration variable maps to different values, creating different DBBs. *Terminator nodes*, or exit points (e.g. jump instructions), have edges to DBBs that could be executed next.

In order to perform dependence analysis, MosaicSim relies on (1) a Static *Data Dependency Graph* (DDG) Generator and (2) a *Dynamic Trace Generator* (DTG). Both tools operate on compiled source code with the kernel annotated. The static DDG Generator uses a series of LLVM passes to capture static inter-instruction dependencies and provide a graphical representation of the source code. This representation can involve many DBBs. Figure 3 illustrates MosaicSim's execution modeling for a core to run a non-speculative example. Nodes in DBBs correspond to instructions, while edges capture data and control flows within and across DBBs.

Since memory dependencies and control flow paths cannot be completely determined statically, the DTG uses an LLVM pass to create an instrumented x86 executable that, when run, writes two trace files: (1) a control flow trace that records the dynamic control flow decisions; and (2) a memory trace that records the addresses for each memory access. Following the native run on the host machine, MosaicSim uses these trace files in the core model, allowing cycle-driven simulation of different execution possibilities (e.g. in-order vs. out-of-order).

Data Dependencies: The DTG outputs information on all addresses accessed, but address aliasing can occur until the program actually resolves the addresses. Thus, MosaicSim implements a *Memory Address Orderer* (MAO), to ensure that true memory dependencies (i.e. Read-After-Write dependencies) are respected. The MAO is populated with memory operations in program order, and can be instantiated with various parameters, e.g. to model a traditional Load-Store Queue (LSQ) in core models (see Section III).

Before a store instruction executes, it checks the MAO to ensure that there exists no incomplete older memory access with a matching or unresolved address. A load only needs to ensure that there exists no incomplete older store with a matching or unresolved address. If these conditions are not met, the memory operation and its dependent instructions stall.

Control Flow Dependencies: MosaicSim serially launches DBBs based on the control flow path trace and the amount of resources devoted to the core model (see Section III-A). Since multiple tiles each run multiple DBBs, the Interleaver coordinates event timing and communication among tiles (detailed in Section II-C) and with the memory hierarchy.

The DTG provides a list of basic block IDs in execution order. For each ID in the list, MosaicSim launches a new DBB based on the corresponding static basic block. A DBB becomes *live*, or is launched, only after the terminator node that branches to it has completed. Instructions cannot execute unless the DBB they belong to has been launched. Note, however, that despite the serial launching of DBBs, MosaicSim can have multiple live DBBs at a given time because a terminator node is not necessarily the last instruction to be completed. For example, terminator node ⑯ in Figure 3 can be reached in just 5 cycles, but it may take longer to reach node ⑫. Thus, new DBBs for a particular basic block are launched when the terminator node has been reached regardless of whether the current DBB has finished. This leads to a variable number of in-flight DBBs per static basic block.

In summary, MosaicSim enforces the following rules to respect data and control flow dependencies:

1) An instruction cannot be issued unless its DBB has been created *and* all of its parent nodes have completed.
2) When an instruction completes, MosaicSim attempts to issue its dependents, while also decreasing the dependents' count of uncompleted parents. Dependents with no additional uncompleted parents can be issued (subject to hardware resource constraints, as discussed in Section III-A).
3) When a terminator node completes and if resource limits have not been reached, the Interleaver launches the next DBB based on the control flow path trace from the DTG.

B. Task to Tile Mappings

A tile executes a *kernel*, which is given as a specially named LLVM function. Different kernels can be mapped to different tiles if distinct DDGs and traces are generated.

Currently, MosaicSim provides a single program, multiple data (SPMD) approach. That is, the user writes one kernel function K and queries a unique tile ID and number of tiles from the execution environment. This provides a familiar and general parallel programming model, similar to MPI and CUDA. The user specifies the number of tiles T at compile time and the DDG generates T graphs of K. The compiler then creates a native binary that executes K with T threads using OpenMP, generating the necessary traces.

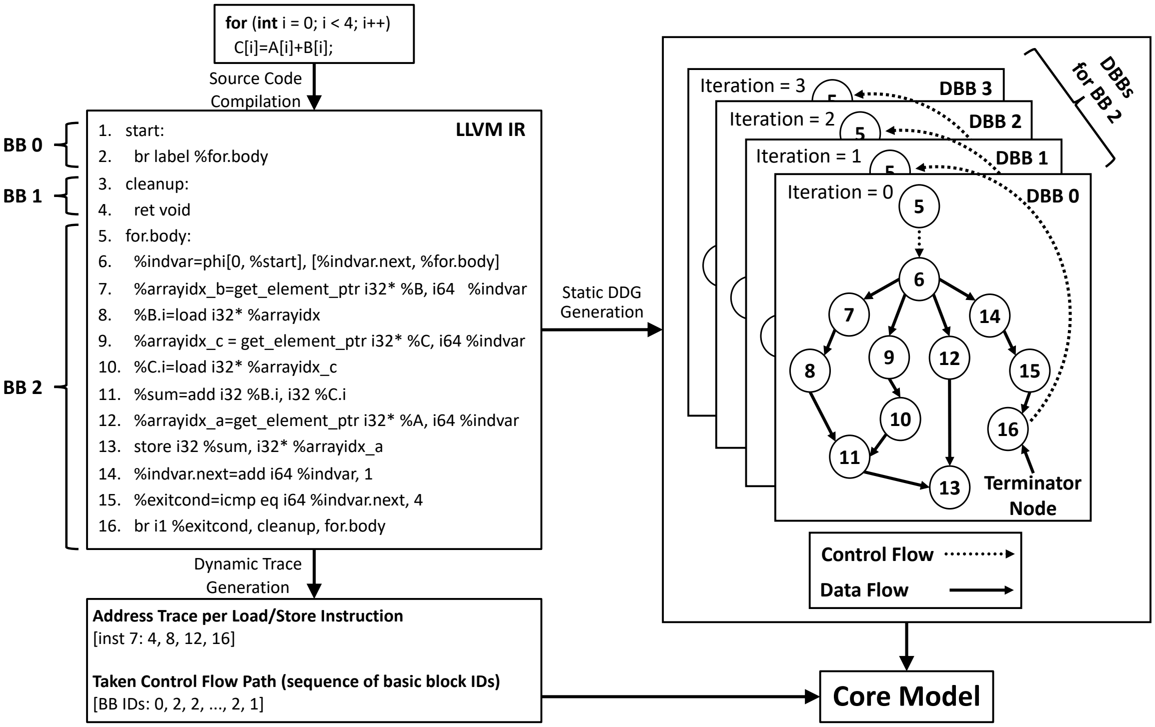
Accelerator tiles (further detailed in Section IV) can be invoked via an API of common accelerated functions, e.g. `SGEMM`. The DDG captures the accelerator call and the DTG records the relevant parameters, e.g. matrix dimensions. During simulation, the accelerator node in the DDG is matched with the trace parameters and the accelerator model is invoked. Sections VII highlights examples of accelerator use.

C. Inter-Tile Communication

Tiles operate alongside each other, each being called upon by the Interleaver (Figure 2) to take a single-cycle step. Tiles can communicate through a traditional shared memory hierarchy, in which memory instructions (i.e. loads and stores) are dispatched to a memory model (discussed in Section V).

Two tiles can additionally communicate with each other through generic *messages*, which can be stored in internal tile buffers. This is realized through a simple message passing

[1] We use *node* and *instructions* interchangeably.



```
for (int i = 0; i < 4; i++)
  C[i]=A[i]+B[i];
```

```
1.  start:
2.    br label %for.body
3.  cleanup:
4.    ret void
5.  for.body:
6.    %indvar=phi[0, %start], [%indvar.next, %for.body]
7.    %arrayidx_b=get_element_ptr i32* %B, i64  %indvar
8.    %B.i=load i32* %arrayidx
9.    %arrayidx_c = get_element_ptr i32* %C, i64 %indvar
10.   %C.i=load i32* %arrayidx_c
11.   %sum=add i32 %B.i, i32 %C.i
12.   %arrayidx_a=get_element_ptr i32* %A, i64 %indvar
13.   store i32 %sum, i32* %arrayidx_a
14.   %indvar.next=add i64 %indvar, 1
15.   %exitcond=icmp eq i64 %indvar.next, 4
16.   br i1 %exitcond, cleanup, for.body
```

Address Trace per Load/Store Instruction
[inst 7: 4, 8, 12, 16]

Taken Control Flow Path (sequence of basic block IDs)
[BB IDs: 0, 2, 2, ..., 2, 1]

Fig. 3. MosaicSim's Execution Modeling Flow

API (i.e. `send`, `recv`). The Interleaver buffers all `send` instructions issued. When the receiving tile issues a `recv` instruction, the Interleaver matches it with the buffered message. This model is simple, generic, and can be used to build more complicated and specialized inter-tile communication models. Section VII-A discusses how these features can be used to implement a Decoupled Access/Execute system [13].

III. Fast Abstract Tile Models in MosaicSim

As previously described, MosaicSim can simulate various tile models that estimate the performance and power costs of a region of LLVM IR. Analysis of the LLVM IR dependence graphs can be shaped to accurately reflect the resource constraints of different tile design choices. This section describes the modeling of different execution scenarios that correspond to microarchitectural resource limits for different tile models.

A. Microarchitectural Resource Limits

In order to be instantiated, a core tile model requires several microarchitectural resource parameters, such as issue width, RoB size, LSQ size, and the number of functional units. Based on these limits, MosaicSim manages resources to accurately model in-order, out-of-order, and accelerator tiles.

Issue Width: MosaicSim models a superscalar issue width W by maintaining a count of issued instructions and ensuring that no more than W instructions can issue each cycle.

ROB: To model an ROB, MosaicSim creates IDs for all instructions that are assigned at DBB creation time. MosaicSim maintains a sliding instruction window (starts with ID 0 and spans the instruction window size) that only allows instructions with IDs within the window to issue. When the oldest issued instruction completes, MosaicSim slides the instruction window forward to issue a younger instruction.

LSQ Size: To model the LSQ, MosaicSim uses the MAO (described in Section II-A) to track loads and stores and ensure that instructions cannot issue if the MAO is full. Memory operations free up space on the MAO upon completion.

Live DBB Limits: MosaicSim provides the option of limiting the number of live DBBs that can run concurrently for each basic block. This limit mimics restricting how many replicated circuits for a loop body appear in a hardware accelerator. Entire DBBs (and their instructions) cannot be launched if the live DBB limit for their basic block has been reached.

Functional Unit Limits: MosaicSim can limit the number of available functional units for each instruction type. It maintains a count of all issued, incomplete instructions and the functional units they utilize. There must be an available functional unit in order to issue an instruction. When instructions complete, they free up the functional units they occupied.

B. Instruction Costs

Individual instructions in MosaicSim have both a latency cost (cycles) and energy cost (Joules). These costs can be predetermined (computation instructions) or dynamic (memory operations). After an instruction with a fixed cost is issued, MosaicSim ensures that it does not complete until its global cycle count has progressed through the latency of the instruction. The fixed energy cost of the instruction is then added to a running total. For instructions with a dynamic cost (e.g. a memory instruction), cost values are determined by querying the memory hierarchy and are subject to factors such as memory contention and cache misses (detailed in Section V).

C. Speculation

MosaicSim is designed to flexibly explore several opportunities for speculation. MosaicSim models control-flow speculation by adding a misprediction latency whenever a modeled branch predictor contradicts the pre-determined control flow path provided by the DTG[2]. By default, MosaicSim must wait until it encounters the terminator node of a basic block before launching a new DBB. However, with speculation, the next DBB can be launched immediately, which makes instructions in the newly launched DBB eligible to be issued. Instructions in a mispredicted path are never executed, as is similar with other instrumentation-based or direct-execution simulators (e.g. Sniper [14] or ZSim [15]).

MosaicSim also leverages information from the DTG to provide an option for perfect *memory address alias speculation*. Since the trace holds information on *all* addresses for all instructions before starting the simulation, MosaicSim knows if any pair of accesses have aliasing addresses ahead of time. Hence, it can "perfectly anticipate" aliasing occurrences and potentially issue memory instructions in the presence of unissued, older instructions with unresolved memory addresses.

IV. Accelerator Simulation

As shown in Figure 2, MosaicSim supports the simulation of heterogeneous SoCs comprised of CPUs and accelerators. MosaicSim offers two styles of accelerator simulation for design progressions from high-level to detailed.

Pre-RTL Accelerator Modeling: Early in the design process, pre-RTL accelerator modeling can help determine which accelerators are useful without their RTL designs. For this purpose, MosaicSim can model accelerators using the same graph-based approach as previously described for CPUs, but with different hardware resource constraints. Fixed-function accelerator models provision hardware resources based on application-specific factors (e.g. loop unroll length and parallelism opportunities). MosaicSim provides knobs to specify the number of active DBBs per basic block (i.e. hardware-supported loop unrolling), number of functional units, etc. In addition, one can use MosaicSim to explore the relaxation of hardware constraints, such as RoB size and instruction window. Rather than targeting a specific hardware implementation, these features enable a high-level exploration of the extent to which an application can benefit from hardware acceleration.

RTL Accelerator Modeling: Later in the design process, MosaicSim allows a high-level accelerator model to be replaced by a more detailed one based on an actual RTL implementation of the accelerator. This is essentially a substitution of one (or several) of the tile models depicted in Figure 2.

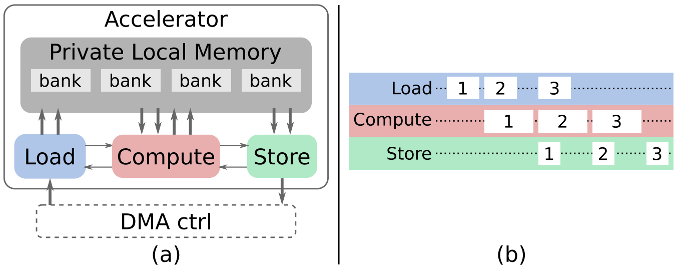


Fig. 4. (a) The accelerator's modules operate in a pipeline with a multi-port, multi-bank private local memory. (b) Computation and communication overlap during the accelerator execution.

A. Accelerator Invocation

When MosaicSim invokes an accelerator, the Interleaver queries the accelerator tile for latency and resource usage information. For graph-based accelerator modeling, this invocation is similar to that of CPU models.

For detailed RTL accelerator modeling, MosaicSim provides an interface tailored to such evaluations. The Interleaver runs a C++ performance model of the accelerator tile, which takes as input two sets of parameters: (1) a standard set of generic system parameters, e.g. technology node, maximum memory bandwidth, number of accelerator instances to be invoked in the system; and (2) a set of accelerator configuration parameters, e.g. number of inputs, input and output sizes. When queried, the accelerator tile model returns to the Interleaver a set of performance estimates, e.g. clock cycles, bytes of memory accessed, and average power consumption. This accelerator data is then included in the final performance results reported by MosaicSim. Individual accelerator tiles can be implemented in various ways as long as they abide by the Interleaver's interface. We next describe our primary methodology for generating the accelerator tile models.

B. RTL Accelerator Model Design

The RTL accelerator modeling approach focuses on accelerators with predictable memory access patterns, i.e. independent accesses. However, the overall MosaicSim approach is more general and can support tile models with any access patterns.

In this modeling approach, accelerators are designed by leveraging the accelerator design flow of the open-source ESP project [16–18], which eases the design effort by using templates to automatically generate most of the accelerator source code. Accelerators are first designed in SystemC and then sent through Cadence's Stratus High-Level Synthesis (HLS) tool to produce an RTL implementation. This methodology is applicable to other languages and tools, for instance ESP provides accelerator design flows also in C/C++, Keras TensorFlow, Pytorch and more.

The accelerators generated through this approach have a pipelined datapath crafted to mask the communication time as much as possible. Fig. 4 presents an accelerator with three concurrent modules: a load process to load data from memory, one or more computation processes, and a store process to send

[2] MosaicSim currently supports static branch prediction in addition to perfect branch prediction. This is useful for early-stage modeling, e.g. obtaining upper bounds. However, future work will support more realistic dynamic branch predictors.

data to memory. The modules communicate through a private local memory, which is a circular or double buffer to enable pipelining of computation and DMA transfers.

Communication Model: In accelerator tile model development, validation of the communication and computation performance characteristics with respect to the expected hardware is key. Both the SystemC and the HLS-generated RTL implementation of an accelerator can be verified in simulation with a SystemC testbench. We augmented the testbench infrastructure of the ESP accelerators with a memory model that accounts for access latency, bandwidth, interconnect bit-width, and average NoC hops between accelerator and memory interfaces (for NoC-based SoCs). These parameters can be tuned to match a target SoC. With this kind of communication model, a designer can focus on the accelerator design without losing sight of its interaction with the rest of the system.

Accelerators can interact with the memory hierarchy in many ways [19, 20]. This work models accelerators as non-coherent; they communicate directly with main memory, bypassing the memory hierarchy. This is common for loosely-coupled accelerators that execute coarse-grained tasks.

Performance Model: MosaicSim has a generic performance model for loosely-coupled, reconfigurable, fixed-function accelerators. The model abstracts an accelerator as a set of concurrent modules, where each module executes one or more loops multiple times. The model can also invoke accelerators in parallel and, given a maximum memory bandwidth, scale execution time and average power consumption accordingly.

The performance model of a specific accelerator employs the generic model by providing the following four arguments: (1) the number of processes; (2) the number of loops per process, which describes the accelerator structure; (3) the total latency of all internal loops; and (4) the number of iterations of each loop, which is function of the configuration parameters of the accelerator invocation.

The designer needs to provide the average power consumption of the accelerator, which can be measured by logic synthesis tools based on the switching activity recorded during RTL simulation. Finally, the accelerator designer should provide an expression to calculate the number of bytes transferred to/from memory as a function of the accelerator configuration.

Accelerator Instrumentation: The generic performance model requires the latency of one iteration of the core loops in each module as input. These are the internal loops, whose latencies do not depend on the accelerator configuration (e.g. input size). To aid the collection of these latencies, we augmented the *ESP accelerator templates* with instrumentation features, so that the accelerator designer can instrument the accelerator to collect the required cycle-accurate latencies.

The instrumentation adds an array of signals to each module and an additional concurrent process, the *collector*, to collect and process all instrumentation signals. Each signal is toggled at every iteration of the corresponding loop. The *collector* measures the toggling latency and communicates the results to the testbench, which ultimately dumps them to a file.

Design Space Exploration: HLS allows for seamless generation and evaluation of multiple RTL implementations of an accelerator given a single high-level SystemC specification. The SoC designer can then choose which specific design point(s) to instantiate as well as how many copies of the same accelerator should be present. The very fast system simulation of accelerators with MosaicSim can greatly help this design-time decision process.

V. Memory Hierarchy

MosaicSim simulates a memory hierarchy that includes caches: both private and shared, and support for two different DRAM models: an in-house model named SimpleDRAM, and the widely-used DRAMSim2 [21].

A. Cache Model

MosaicSim's cache model can be utilized as a per-core private cache or shared cache. Both are independently configurable for size, cache line size, associativity, and access latency. MosaicSim is a timing simulator and therefore need not hold actual data in the caches; the address tags suffice.

The cache hierarchy is conventionally write back, write allocate, and fully-inclusive. Each core tile model maintains a cache queue ordered with respect to the cache hierarchy. Memory requests are initially sent to the L1 cache at the front of the queue and forwarded to the next cache when necessary (e.g. cache misses or writeback of dirty data). At the end of the queue, the LLC forwards requests to the DRAM model (described in Section V-B).

The cache model includes a prefetcher that detects streaming patterns of memory accesses. It simply tracks memory requests to see if there exists a chain of accesses that are k words apart. If so, a number of additional requests are generated by the cache for subsequent cachelines in anticipation of future memory instructions accessing those cachelines. The number of cachelines prefetched and the address distance from the instruction triggering the prefetches can be configured. MosaicSim's memory hierarchy model provides a flexible and straightforward interface to implement more complex or specialized prefetchers as well.

To coalesce memory requests, caches can utilize an MSHR whose size can be configured. When a cache receives a request, it checks the MSHR to see if there exists a pending request to the same cacheline. If so, it saves the request on the MSHR. When the pending request is served, the MSHR notifies all requests waiting on that cacheline.

Precise modeling of NoCs, consistency, and coherence are currently not implemented in MosaicSim, as this level of detail is not required by our early-stage modeling. However, future work aims to provide their support. With MosaicSim's modular design, ports can be added to the abstract tile model to create a message module in order to model NoCs and the necessary communication for coherence and consistency. A directory protocol can easily be implemented by treating the Interleaver as the directory and allowing it to communicate with the caches.

TABLE I
EVALUATION SYSTEM DETAILS INTEL XEON E5-2667 V3

Sockets, Cores	2 sockets, 8 cores each
Node Technology and Frequency	22nm, 3200 MHz
L1-I and L1-D	32KB private / 8-way
L2	2MB private / 8-way
LLC	20MB shared / 20-way
DRAM	128GB DDR4 @ 68GB/s

B. DRAM Model

MosaicSim supports two DRAM models: an in-house model called SimpleDRAM and the widely-used DRAMSim2 [21]. SimpleDRAM ensures that all DRAM requests abide by a minimum latency and maximum bandwidth. Every DRAM request is inserted into a priority queue ordered by minimum request completion time (current cycles plus minimum latency). SimpleDRAM enforces the maximum bandwidth limit in epochs. Every cycle, it attempts to return as many requests as possible that have served the minimum latency. Once the number of requests returned in that epoch has exhausted the maximum bandwidth, SimpleDRAM cannot return requests until the next epoch, but it can continue receiving new requests. SimpleDRAM thus models memory bandwidth contention and throttling due to bandwidth limits.

SimpleDRAM is the default model, but MosaicSim can be configured to use DRAMsim2 for cycle-accurate DRAM modeling, albeit this model executes slower has a larger memory footprint during simulation than SimpleDRAM.

VI. EVALUATION

We make use of our hardware-software toolchain to evaluate MosaicSim on a variety of benchmarks. The simulator relies on the compiler to generate the DDG and the DTG to instrument the code and generate memory and control flow path traces. MosaicSim utilizes the front-end tools in the stack to quickly and accurately simulate heterogeneous and hardware-software co-design systems.

A. Accuracy

In order to measure MosaicSim's ability to accurately capture and characterize application trends, we perform two evaluations. First, we utilize the Parboil benchmark suite [22] to evaluate core and memory hierarchy models. We validate MosaicSim by running benchmarks on the Intel Xeon E5-2667 v3 processor (features detailed in Table I) and collecting measurements of our real machine using Intel VTune Amplifier [23]. By using VTune's function-level filtering to isolate profiling information for the kernel, we obtain cycle and instruction counts to compare against MosaicSim performance estimates. Second, we evaluate our accelerator tile models against both RTL simulation as well as FPGA execution.

Application Characterization: Figure 5 displays the accuracy of MosaicSim's runtime estimates compared to the measured performance of real hardware. MosaicSim achieves a geomean accuracy (simulated cycles/real cycles) of 1.099×. Accuracy discrepancies arise from MosaicSim being ISA-agnostic; it cannot perfectly capture cases where LLVM IR instructions do not have a direct, 1-to-1 mapping to actual ISA instructions. For example, LLVM IR requires two instructions for loading from an address offset: `load` and `getelementptr`, while the x86 ISA can perform this with one instruction: `MOV`. Additionally, a direct comparison of LLVM IR simulation against a native ISA must take into account compiler optimizations applied when producing the binary from the IR (e.g. we have found that using `-O3` and loop unrolling produces a more accurate comparison to an x86 instruction counts). Thus, we expect precise IPC and timing models to be noisy when compared to the execution of a concrete ISA. Figure 5 demonstrates this behavior on the Parboil suite: although the geomean accuracy is high, individual benchmark measurements can be inaccurate. We have found that fine-grained tuning of LLVM IR simulation for concrete ISAs, e.g. simulating pairs of `load` and `getelementptr` as one instruction for x86, can increase accuracy. However, MosaicSim aims to be ISA-agnostic and therefore focuses more on characterization rather than on raw cycle accuracies.

Due to the extra abstraction layer of LLVM IR, it is difficult to perform raw IPC comparisons without tuning to a specific ISA. However, we can use the IPCs that MosaicSim reports to characterize kernels as memory or compute-bound. A lower IPC indicates that a kernel is memory-bound while a higher IPC indicates being compute-bound. These results, e.g. `BFS` being memory-bound and `SGEMM` being compute-bound, match previous characterizations of these common benchmarks [22, 24].

Scaling Trends: In order to evaluate MosaicSim's ability to capture scaling trends, we measure both simulated and real hardware performance for {1, 2, 4, 8} thread(s). We then normalize all performance numbers to those with a single thread and evaluate how benchmark speedups scale with an increasing number of threads.

Figures 7 - 9 highlight the comparison of scaling trends for three well-studied benchmarks with different performance bottlenecks: `BFS` (latency-bound), `SGEMM` (compute-bound), and `SPMV` (bandwidth-bound), respectively. MosaicSim nearly perfectly captures the linear scaling trend of `SGEMM` as the kernel is compute-bound and exposes data-level parallelism. `SPMV` is bandwidth bound, i.e. memory accesses are occasionally throttled, and we accurately capture the resulting sublinear scaling trend here. MosaicSim is not as accurate on the latency-bound `BFS` kernel due to the use of atomic read-modify-write instructions that are difficult to accurately model in the memory system (Section V); future work aims to more accurately model these instructions.

Being ISA-agnostic, MosaicSim demonstrates usefulness as an early-stage tool goal with its ability to capture performance bottlenecks and characterizations. Scaling and IPC characterizations are accurate and in line with prior work. If a designer later requires runtime accuracy for a given ISA, it is possible to add fine-grained tunings for LLVM IR simulation to help account for ISA discrepancies.

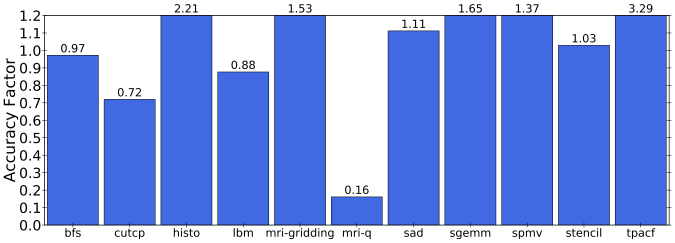


Fig. 5. Despite inaccuracies in individual benchmarks due to ISA differences, MosaicSim achieves a geomean runtime accuracy of 1.099× against an x86 machine.

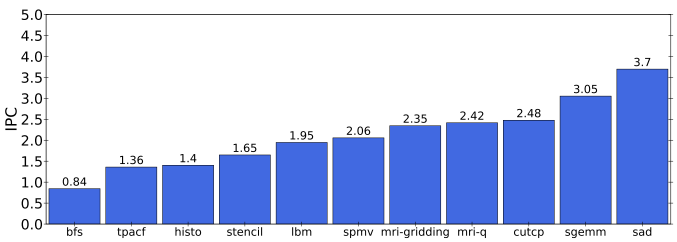


Fig. 6. MosaicSim accurately characterizes applications with IPC measurements (lower implies more memory-bound while higher implies compute bound).

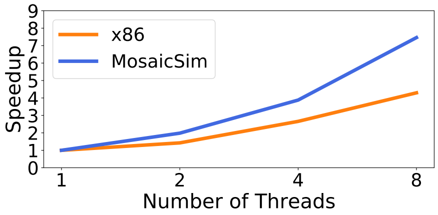


Fig. 7. BFS Scaling Trends

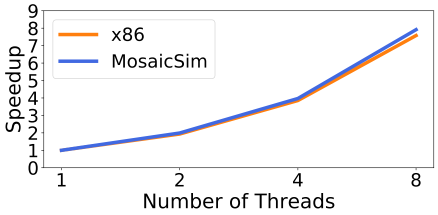


Fig. 8. SGEMM Scaling Trends

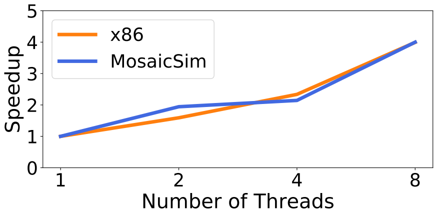


Fig. 9. SPMV Scaling Trends

Accelerator Simulation: With the design flow described in Section IV we created three fixed-function accelerators for matrix multiplication, saturating histogram, and element-wise arithmetic. These accelerators support any input size and any number of inputs per invocation. Using the ESP platform [19, 25], we deployed the accelerators on a Xilinx Ultrascale+ FPGA as part of a many-accelerator SoC capable of running Linux. Therefore, we were able to validate the accelerators both with RTL simulation and on FPGA. With HLS we generated multiple design points for each SystemC specification of the accelerators. Figures 10a-c shows the execution time and area of four design points (with varying PLM size) and four workload sizes. Each design point is a distinct RTL implementation of the accelerator, whose performance model can be invoked by MosaicSim.

Fig. 10d shows the execution time accuracy of the models against RTL simulation of the accelerator and against full system FPGA emulation. The accuracy of each accelerator is the average of its accuracies for all the data points and workload sizes in Figures 10a-c. The average accuracy with respect to RTL simulation is between 97% and 100%, proving that our back-annotated generic performance model captures precisely the behavior of the accelerators. Furthermore, the models exhibit high accuracy ($> 89\%$) when compared to a full SoC running on FPGA, validating the communication model that we added to the *ESP accelerator templates*.

Recent literature shows that for medium to large workloads the overhead of the accelerator invocation through a Linux device driver is negligible [19, 20]. We confirmed these results by measuring the overhead of invoking the accelerators, by invoking them on trivial workloads. We found that the overhead is consistently below 1% of the execution time for the design points in Fig. 10.

These RTL-based accelerator performance models do not actually execute the workloads and therefore take nearly no time to execute. They are several orders of magnitude faster than both RTL simulation and MosaicSim's pre-RTL accelerator modeling. In fact, these performance models are even faster than FPGA execution of the workloads they model.

B. *Using MosaicSim*

This section describes practical details of using MosaicSim as an early-stage design tool for hardware-software co-design.

Designer Effort: MosaicSim provides a comprehensive set of both core and system configuration files that include a number of reconfigurable parameters (e.g. ROB size, issue-width, memory hierarchy details, etc.). These are straightforward to modify or extend, providing minimal designer effort.

Simulation Speed: MosaicSim has a competitive simulation speed, achieving a single-threaded speed of up to 0.47 MIPs. This is comparable to that of Sniper [26] (up to 0.45 MIPS) and is one order of magnitude better than gem5 [27] (up to 0.053 MIPS). When the simulated system includes coarse-grained accelerator performance models (see Section IV), the simulation speed is even higher, as many cycles of accelerator contributions are derived from a closed form equation (using parameters obtained from a dynamic trace).

Storage Requirements: As described in Section II-A, MosaicSim requires both a DDG and memory control flow traces in order to run. The sizes of the DDG and control flow traces are typically less than 1 GB, thus we consider them negligible. However, the memory traces can be several GB large depending on the kernel. For example, in using the default datasets in Parboil, BFS takes 1.3 GB, HISTO takes 1.4 GB, and SGEMM takes 99 MB. While these traces can be large, they are necessary for accurate dynamic modeling of application behavior. MosaicSim therefore aims to strike an appropriate balance between space efficiency and accuracy.

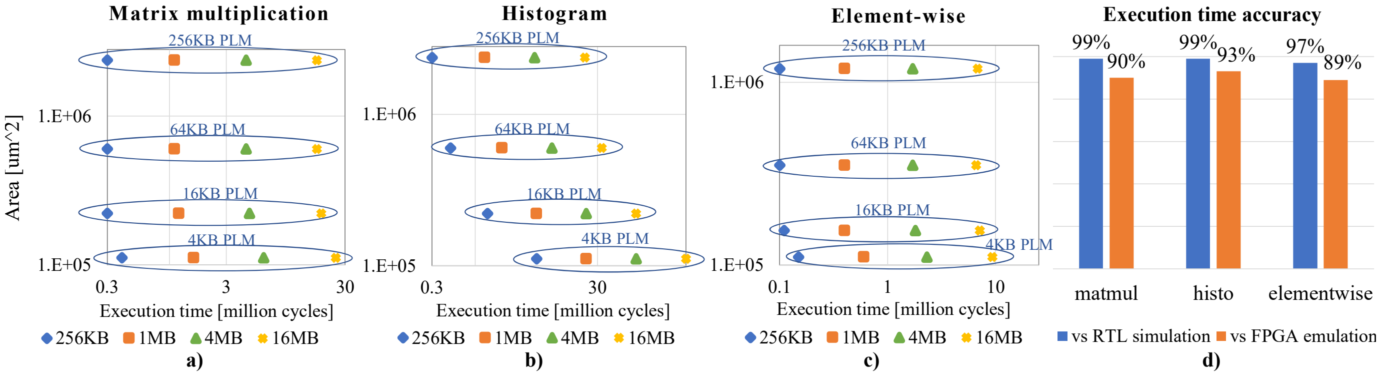


Fig. 10. a,b,c) Design space exploration for multiple workload sizes of 3 reconfigurable fixed-function accelerators. d) Accuracy of RTL-based performance models with respect to RTL simulation and full-system FPGA emulation.

VII. Case Studies

In addition to across-the-board studies of simulation characteristics and accuracy, we provide three case studies that demonstrate the value of MosaicSim to model complex heterogeneous systems and perform hardware-software co-design.

A. DAE for Latency Tolerance

The Decoupled Access/Execute (DAE) paradigm [13] has been widely explored as a technique to tolerate memory latency by dividing a kernel into an *access* slice and an *execute* slice. The access slice performs *all* memory accesses and all computation for an access, i.e. address computations and control flow statements where memory data is involved. Meanwhile, the execute slice performs value computation.

The access slice performs loads and enqueues their data into a communication buffer between the access and execute slices. When the execute slice encounters a load, it simply reads the data from this buffer. Store instructions work conversely. The key idea is that if the access slice can run ahead of the execute slice and produce all of the data required for computation, it can essentially act as a non-speculative "perfect prefetcher". The buffers in DAE are generally proposed in hardware implementations, leading to a *heterogeneous* system, consisting of access and execute cores that execute their respective program slices concurrently.

Due to MosaicSim's support for heterogeneity, we can implement and evaluate DeSC [24, 28, 29], a recently proposed DAE-based system. MosaicSim allows us to evaluate DeSC on different (multi)core models (out-of-order and in-order), and perform area-equivalent design space exploration.

Compiler and Simulator Support: DAE program slicing can be implemented in the LLVM toolchain as a compiler pass. The pass first creates two copies of the kernel, one for access and one for execute. On the access slice, each memory instruction is augmented with a special function to either (1) push to the buffer for loads or, (2) replace a store value with a value from the buffer for stores. The execute slice is transformed similarly.

The DAE simulator support uses MosaicSim's inter-tile message-passing capabilities (Section II-C) to provide direct communication between the access and the execute cores. The load buffer is a `send` from the access slice and a `recv` from the compute slice. The store buffer is implemented conversely. Thus, the Interleaver processes these fine-grained inter-tile messages naturally. Additionally, the default core models were extended to include the structures described in [24], i.e. communication queues, the terminal load buffer, the store address buffer, and the store value buffer.

Results: We evaluate our DAE implementation on the bipartite graph projection kernel, which has a wide set of use cases

TABLE II
Parameters for DAE case-study.

Microarchitecture Parameter	Out-of-Order	In-Order
Issue Width	4	1
Instruction Window/RoB/LSQ	128/128/128	1
Frequency/Tech	2GHz/22nm	2GHz/22nm
Area (mm^2)	8.44	1.01

Memory Parameter	Values
L1	32KB / 22nm node / 8-way / 1-cycle latency
L2	2MB / 22nm node / 8-way / 6-cycle latency
DRAM	DDR3L / 24GB/s BW / 200-cycle latency
Comm. Buffer Sizes	512 entries / 1-cycle latency

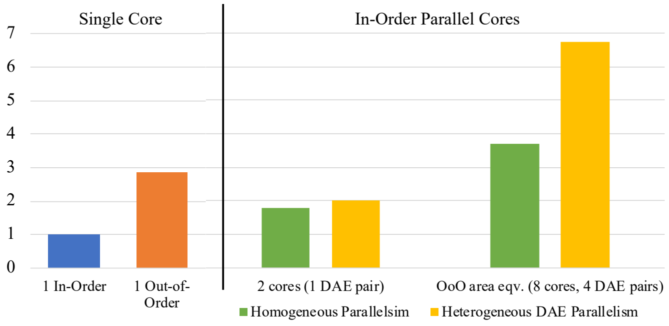


Fig. 11. Speedups of different systems on the graph projections kernel. In an equal-area comparison of 8 In-Order cores to 1 OoO core (right), DAE heterogeneity outperforms OoO by nearly 2×, and is a promising approach for latency tolerance.

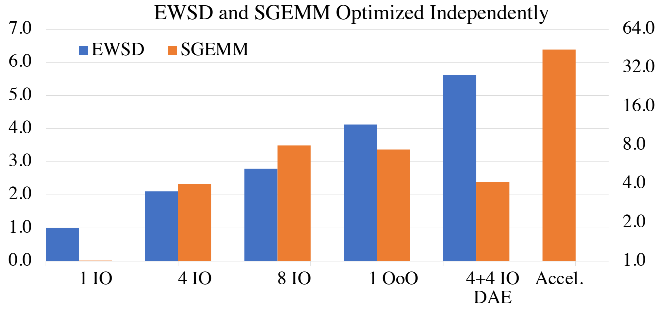


Fig. 12. Speedups of different systems on the EWSD (left axis) and SGEMM (right axis) kernels. EWSD benefits from latency tolerant architectures, such as OoO and DAE systems. SGEMM benefits most from an accelerator.

from recommendation systems [30] to disease association prediction [31]. This application is memory latency bound; each pair of edges in the original bipartite graph updates a projection edge, which creates an irregular memory access.

We consider two base core models: in-order (InO) and out-of-order (OoO) (see Table II, area measurements are from McPAT [32]). We augment the in-order model with DAE components to instantiate a parallel heterogeneous system where half the cores are access and the other half are execute.

Figure 11 highlights the results of this case study. We measure the performances of single-core, and homogeneous and heterogeneous parallel systems, and normalize them to that of a single InO core. As seen on the left, the OoO core, equipped with latency tolerance mechanisms, significantly outperforms the InO core. The right side presents scaling to 2 cores or 1 DAE pair and an OoO area-equivalent scaling to 8 cores or 4 DAE pairs. We see near-linear scaling for homogeneous parallelism (green bars), as a linear number of memory requests are issued in parallel. Finally, we see that heterogeneous parallelism (yellow bars) yields the highest speedups (nearly 2×) via asynchronous issuing of memory requests (proposed by modern DAE systems [24]) and significant memory-level parallelism. Thus, MosaicSim has enabled us to explore a heterogeneous system design as a promising approach for parallel, latency tolerant architectures.

B. Alternating Sparse-Dense Phases

To further highlight MosaicSim's ability to simulate complex heterogeneous systems, we explore the architectural design space for applications which have both dense linear algebra (typically compute-bound) and sparse linear algebra (typically memory-bound). For example, Sinkhorn Distances [33] is an algorithm for solving the optimal transportation problem and is used in computer vision [34] and NLP [35]. The bottleneck of the application is split between a dense matrix multiplication (SGEMM) and an element-wise matrix operation where one operand is sparse and one is dense (EWSD).

Architectures with Multiple Objectives: To study the architectural design space for these types of applications, we start with constructing two microbenchmarks: SGEMM alone and EWSD alone. We simulate the runtime of each microbenchmark under various system configurations (see Tables II) and present the results in Figure 12. We use one InO core as the absolute baseline, as it is the simplest system. Since SGEMM is a compute-bound kernel, we consider the use of a fixed-function accelerator. Specifically, we use the matrix multiplication accelerator introduced in Section VI-A.

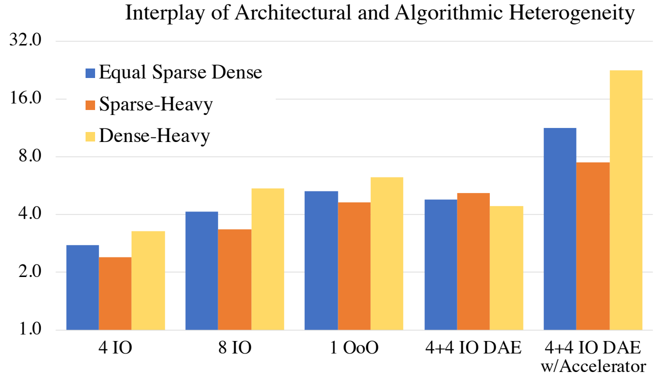


Fig. 13. A heterogeneous system executing a combined kernel of both dense (SGEMM) and sparse (EWSD), with one IO core being the baseline. The most heterogeneous system has the best performance (DAE with an accelerator).

The two microbenchmarks have different architectural performance landscapes. An optimal architecture for a kernel that combines them therefore needs to resolve conflicting demands. SGEMM sees large improvements from computation resources, as a fixed function accelerator for SGEMM provides nearly 45× speedup. Meanwhile, EWSD is memory bound and benefits from the latency tolerance available in a DAE architecture, which provides nearly a 6× speedup.

Heterogeneous System Simulation: MosaicSim's main strength is simulation support for a complex heterogeneous system. To demonstrate this, we now construct a *combined* benchmark that performs SGEMM and EWSD kernels serially. We then instantiated them with three different dataset sizes, where we varied the percentage of the total number of cycles spent in SGEMM versus EWSD based on their expected number of cycles on one InO core. This yielded a dense-heavy (75% SGEMM, 25% EWSD), a sparse-heavy (25% SGEMM, 75% EWSD), and an equally divided kernel. These combinations model workloads found in real-world applications [33–35].

Figure 13 summarizes speedups of various architectures. Depending on the ratio of execution time for each of the two phases, the optimal architecture for the combined approach is non-trivial and requires the simulation of *both* phases using a variety of tiles that make up a complex heterogeneous system. Our results show that in the absence of a specialized accelerator for the dense operation, the combined kernel would benefit most from 4 DAE pairs if the kernel is sparse-heavy and 1 OoO core if it is dense-heavy. With an accelerator however, 4 DAE pairs are the ideal choice for all cases. MosaicSim allows the exploration of many combinations and configurations through its lightweight plug-and-play interface.

C. Performance Modeling of TensorFlow Programs:

To further demonstrate accelerator performance modeling with MosaicSim, we present an example of simulation support for Keras TensorFlow programs. Keras [11] is TensorFlow's high-level API for designing and training deep learning models. Applications of interest are therefore composed of multiple neural network kernels, e.g. convolution, matrix multiplication, pooling, etc. These kernels are computationally intensive and significantly contribute to the overall execution time of deep learning applications. Therefore, they are often deployed on accelerators. Thus, MosaicSim can generate performance estimates of a Tensorflow application using accelerator performance models.

To demonstrate this, we added a Keras TensorFlow API in the compiler to recognize Keras function names in the source code and map them to LLVM accelerator invocation calls when the application is compiled. These function calls are preserved as special instructions in MosaicSim, where we add accelerator performance models for ESP accelerators [16] according to the design flow described in Section IV-B. These accelerators provide kernel support for convolution, matrix multiplication, activation, pooling, etc. The accelerator invocation calls then appear in the instrumented LLVM that MosaicSim operates on, so once the application is compiled and executed, the accelerator invocations are simulated whenever MosaicSim encounters their function calls. We evaluate MosaicSim's TensorFlow application performance modeling with three deep neural network applications below.

`ConvNet` is a type of convolutional neural network (CNN) application. CNNs are used to extract spatial, temporal and spatiotemporal relationship in data such as images, protein structure, language, and weather. The `ConvNet` algorithm contains an initial convolutional layer followed by a ReLu nonlinear layer that is regularized by batch normalization. This is followed by three residual blocks containing convolutional and residual layers. The final residual block is connected to a pooling layer and the model ends with a fully connected and activation layer that outputs a classification prediction.

`GraphSage` combines graph and neural network algorithms and can be used as a recommendation system [36]. The objective of the algorithm is to sample graph data through a random walk and transform this data into a dense vector format that can be fed into a neural network architecture consisting of fully connected layers and ReLU layers. The algorithm mimics the continuous bag of words (CBOW) algorithm where instead of words, visited nodes are inserted into the input vector.

`RecSys` is a recommendation system modeled using neural networks. Training the algorithm takes as input individuals' preferences out of many available options, where the data is vectorized and fed into the model in batches. The neural network itself contains two sequential fully connected layers with ReLU nonlinear steps which are regularized with batch normalization and dropout methods. These layers are followed by a final fully connected layer which outputs new items that the model recommends.

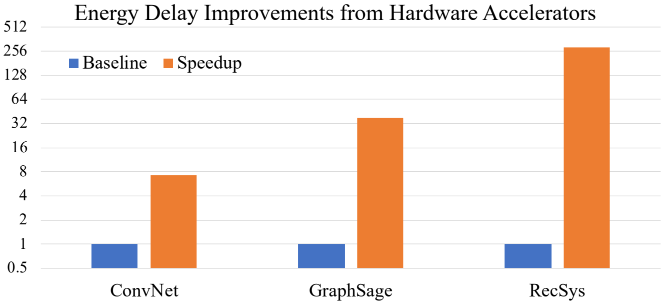


Fig. 14. Energy-delay improvement comparison between out-of-order cores and accelerator-oriented SoCs for three DNN applications (`ConvNet`, `GraphSage`, and `RecSys`).

We simulate and compare the performance of training of these three applications on two systems: an out-of-order server core with no accelerators and an SoC integrating 8 accelerators. We measure performance in energy-delay product, a metric which combines runtime and energy efficiency. Figure 14 highlights the comparisons, showing that `Convnet`, `GraphSage`, and `RecSys` reap 7.22×, 38×, and 282.24× improvements in energy-delay product, respectively. Note that we do not have accelerators for backpropagation of convolutional layers and therefore the modest improvement for ConvNet is due to forward propagation acceleration in the context of the entire training. In addition, GraphSage includes random walk and embedding steps that are not handled by an accelerator. RecSys on the other hand is entirely handled by accelerators and results in its impressive improvement. These results highlight the performance benefits of accelerators for compute-bound kernels in DNN applications. MosaicSim supports detailed accelerator performance modeling suitable for Keras TensorFlow kernels.

VIII. Related Work

Previous work on simulators, i.e. Graphite [37], Sniper [14], ZSim [15] and PriME [38], has focused on increasing (1) accuracy through tuned core models and native execution and (2) simulation scalability by executing simulations on parallel, multicore host machines. ZSim and PriME were designed to make many-core, i.e. thousands of cores, simulation practical. Furthermore, some of these works allow for flexible memory hierarchies. However, all of these prior works only simulate homogeneous core systems.

Sniper extended Graphite by providing a more detailed core model and using interval simulation. However, this results in a trade-off between accuracy and simulator performance. This level of abstraction lies in between 1-IPC models and highly detailed hardware pipelines. MosaicSim also sits at this abstraction level, but makes use of LLVM IR to create a DDG for instruction scheduling in cycle-driven simulation. The use of LLVM IR also allows natural additions of compiler passes and new instructions (e.g. DAE in Section VII-A).

ZSim is a many-core simulator that instruments a binary based on every basic block and memory operation. It leverages the host machine to perform functional simulation and model timing, hindering its ability to simulate different types of cores. MosaicSim also performs code instrumentation and native execution for memory access behavior and control-flow resolution, but its modular, tile-based nature allows it to simulate a variety of tiles in a heterogeneous system.

PriME was designed for many-core system simulation as well, but focuses on microarchitectural exploration, including cache hierarchies, coherence protocols, and NoCs. Though it presents a tile-based architecture like MosaicSim, their tiles require homogeneity, making it an unsuitable simulator for modeling accelerator-oriented many-core systems.

Accelerator Simulation: Other works have focused on simulating accelerator performance. Rogers et al. [39] devised an LLVM-based accelerator model in gem5 that leverages a data dependency graph to simplify the simulation of a many-accelerator system. MosaicSim uses the same front-end, but is not limited to accelerator modeling; it supports a variety of other SoC components, including core models (e.g. in-order and out-of-order). Furthermore, MosaicSim provides tile-to-tile scratchpad communication, e.g. to support data communication schemes like DAE. Because it does not rely on gem5, MosaicSim allows for greater implementation flexibility and higher simulation speed.

Gem5-Aladdin [20] is another gem5-based approach that uses Aladdin [2] for fixed-function accelerator design in the context of an SoC. Gem5-Aladdin measures the impact of DMA overload in an SoC to design accelerators in a holistic manner rather than in isolation. Additionally, the work evaluates simple accelerators where normally the input and output data fit in the local memory of the accelerator. At each invocation these accelerators execute for a few thousands of cycles, which is typically less than the overhead of their invocation from a Linux device driver.

On the other hand, MosaicSim can model accelerators of any complexity, e.g. accelerators for which: (1) communication and computation are decoupled and concurrent, (2) input and output data do not need to fit in the local memory of the accelerator, they can be of arbitrary size. For this reason, we were able to evaluate realistic accelerator workloads in terms of size. If the accelerators are invoked for small tasks, the invocation overhead dominates the execution time and the accelerator hardly achieves any speedup with respect to general purpose cores. Our measurements of accelerator execution time on FPGA included the invocation overhead. Furthermore, MosaicSim considers heterogeneity not only in combining accelerators with a core model, but also in providing flexible core models. Its LLVM-based approach allows natural agile development of programming models, ISA extensions, and novel architectures.

To the best of our knowledge, MosaicSim is the first simulation approach for loosely-coupled heterogeneous systems, offering flexible, early-stage exploration of hardware-software co-design approaches to design new architectures.

IX. Conclusion

This paper presents MosaicSim, a lightweight, modular simulator to flexibly explore the design space of heterogeneous systems via hardware-software co-design. MosaicSim (1) is tightly integrated with the LLVM framework, providing agile programming models, enabling full-stack approaches; (2) provides abstract tile models capturing pragmatic microarchitectural details and specialized tile-to-tile interactions; and (3) provides support for accelerator model integration to create complex heterogeneous systems. MosaicSim is a timely contribution in the New Golden Age of Computer Architecture [4], where flexible hardware-software co-design and heterogeneity are key to performance improvements.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful feedback. 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 supported by the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU) and by Fundación Séneca-Agencia de Ciencia y Tecnología de la Región de Murcia, Programa Jiménez de la Espada (grant 20580/EE/18). 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] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "OpenPiton: An open source manycore research framework," in *ASPLOS*. ACM, 2016, pp. 217–232.

[2] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ISCA*. ACM, 2014.

[3] A. Venkat and D. M. Tullsen, "Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor," in *ISCA*. IEEE Press, 2014.

[4] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[6] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *HPCA*. IEEE, 2010.

[7] K. Rupp, "40 Years of Mircroprocessor Trend Data," https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data, 2015.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*. IEEE Press, 2004.

[9] "Clang: a C language family frontend for LLVM," http://clang.llvm.org/.

[10] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2015.

[11] TensorFlow, "Keras," https://www.tensorflow.org/guide/keras Retrieved May 2019.

[12] The LLVM Foundation, "Basic Block," https://llvm.org/doxygen/group_LLVMCCoreValueBasicBlock.html, 2019.

[13] J. E. Smith, "Decoupled access/execute computer architectures," in *ACM SIGARCH Computer Architecture News*, vol. 10. IEEE Press, 1982.

[14] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC*. ACM, 2011.

[15] D. Sanchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA*. ACM, 2013.

[16] "ESP: The open-source heterogeneous system-on-chip platform," https://esp.cs.columbia.edu/, 2019.

[17] D. Giri, K.-L. Chiu, G. Di Guglielmo, P. Mantovani, and L. P. Carloni, "ESP4ML: Platform-based design of systems-on-chip for embedded machine learning," in *DATE*. IEEE Press, 2020.

[18] L. Piccolboni, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "Broadening the exploration of the accelerator design space in embedded scalable platforms," in *HPEC*. IEEE Press, 2017.

[19] D. Giri,, P. Mantovani, and L. P. Carloni, "Accelerators and coherence: An SoC perspective," *IEEE Micro*, vol. 38, no. 6, pp. 36–45, 2018.

[20] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," *IEEE Micro*, pp. 1–12, 2016.

[21] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[22] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, 2012.

[23] Intel Corporation, "Intel VTune Amplifier 2019," https://software.intel.com/en-us/intel-vtune-amplifier-xe, 2019.

[24] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: Decoupled supply-compute communication management for heterogeneous architectures," in *MICRO*. ACM, 2015.

[25] D. Giri, P. Mantovani, and L. P. Carloni, "NoC-based support of heterogeneous cache-coherence models for accelerators," in *International Symposium on Networks-on-Chip (NOCS)*. IEEE/ACM, 2018, pp. 1–8.

[26] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 3, Aug. 2014.

[27] T. Ta, L. Cheng, and C. Batten, "Simulating multi-core RISC-V systems in gem5," in *Workshop on Computer Architecture Research with RISC-V*, jun 2018.

[28] T. J. Ham, J. L. Aragón, and M. Martonosi, "Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures," *ACM TACO*, vol. 14, no. 2, 2017.

[29] ——, "Efficient data supply for parallel heterogeneous architectures," *ACM TACO*, vol. 16, no. 2, 2019.

[30] T. Zhou, J. Ren, M. Medo, and Y.-C. Zhang, "Bipartite network projection and personal recommendation," *Physical Review*, vol. 76, no. 4, Oct. 2007.

[31] X. Chen, D. Xie, L. Wang, Q. Zhao, Z.-H. You, and H. Liu, "BNPMDA: Bipartite network projection for MiRNA–disease association prediction," *Bioinformatics*, vol. 34, no. 18, pp. 3178–3186, Apr. 2018.

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*. IEEE/ACM, 2009.

[33] M. Cuturi, "Sinkhorn distances: Lightspeed computation of optimal transport," *Advances in Neural Info. Proc. Sys.*, vol. 26, 2013.

[34] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance, multi-dimensional scaling, and color-based image retrieval," in *Image Understanding Workshop*, 1997, pp. 661–668.

[35] M. J. Kusner, Y. S. 0020, N. I. Kolkin, and K. Q. Weinberger, "From word embeddings to document distances," in *ICML*, ser. JMLR Workshop and Conference Proceedings, vol. 37, 2015, pp. 957–966.

[36] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NIPS*. Curran Associates, Inc., 2017, pp. 1025–1035.

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

[38] Y. Fu and D. Wentzlaff, "PriME: A parallel and distributed simulator for thousand-core chips," in *ISPASS*. IEEE Press, March 2014.

[39] R. R. H. T. Samuel Rogers, Joshua Slycord, "Scalable LLVM-based accelerator modeling in gem5," in *IEEE. Computer Architecture Letters*, vol. 18, Jun. 2019.