Hardware-Software Co-Design for an Analog-Digital Accelerator for Machine Learning

Joao Ambrosi*, Aayush Ankit†, Rodrigo Antunes*, Sai Rahul Chalamalasetti*, Soumitra Chatterjee*, Izzat El Hajj‡, Guilherme Fachini*, Paolo Faraboschi*, Martin Foltin*, Sitao Huang‡, Wen-mei Hwu‡, Gustavo Knuppe*, Sunil Vishwanathpur Lakshminarasimha*, Dejan Milojicic*, Mohan Parthasarathy*, Filipe Ribeiro*, Lucas Rosa*, Kaushik Roy†, Plinio Silveira*, John Paul Strachan*

*Hewlett Packard Enterprise, 1500 Page Mill Road, Palo Alto, CA 94304, USA
†Purdue University, School of Electrical and Computer Engineering, West Lafayette, IN 47907, USA
‡University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
Emails: {firstname.lastname}@hpe.com, {aankit,kaushik}@purdue.edu, {elhajj2,shuang91,w-hwu}@illinois.edu

***Abstract*—The increasing deployment of machine learning at the core and at the edge for applications such as video and image recognition has resulted in a number of special purpose accelerators in this domain. However, these accelerators do not have full end-to-end software stacks for application development, resulting in hard-to-develop, proprietary, and suboptimal application programming and executables.**

In this paper, we describe software stack for a memristor-based hybrid (analog-digital) accelerator. The software stack consists of an ONNX converter, an application optimizer, a compiler, a driver, and emulators. The ONNX converter helps leveraging interoperable neural network models developed on frameworks that support ONNX, such as CNTK, Caffe2, Tensorflow, etc. The application optimization layer adapts these interoperable models to the underlying hardware. The compiler generates executable ISA code that the underlying accelerator can run. Finally, the emulator enables software execution without actual hardware which enables hardware design space exploration and testing.

By building a software stack, we have made hybrid memristor-based ML accelerators more accessible to software developers, enabled a generation of better-performing executables, and created an environment that can be leveraged by a multitude of existing neural network models developed using other frameworks to target these accelerators.

I. Introduction

The history of computing has seen analog [1], [2], [3], [4], [5], digital [6], [7], [8], and hybrid computing [9], [10], [11], [12]. Fueled by Moore's law, digital computing in the last four decades has become dominant. Hardware architectures, instruction set architectures (ISA), operating systems, compilers, software tools, and applications have all been developed for digital computing. With the slowing down of Moore's law and the end of Dennard scaling, there is a renewed interest in analog and hybrid analog-digital alternatives for computing architectures. These alternatives require careful hardware-software co-design if they are ever to gain traction in the real world.

Machine Learning (ML) workloads have been the center of attention for many new accelerator architectures due to recent breakthroughs that made them pervasive in many application domains. The architectures that have been proposed have leveraged both digital computing [13], [14], [15], [16], [17], [18] as well as hybrid digital-analog computing using memristive crossbars [19], [20], [21], [22], [23], [24], [25], [26], [27], [28]. The reason memristive crossbars have been particularly attractive is their ability to perform low-energy and low-latency Matrix Vector Multiplication (MVM) operations by leveraging analog circuit behavior [29], [30], [31], [32], [33], [34]. Moreover, their high storage density allows storing large matrices on chip, in contrast with digital alternatives which have lower storage density, thereby incurring off-chip accesses which are detrimental in the absence of data reuse which MVM operations are notorious for. Since many ML workloads perform a large number of MVM operations, hybrid accelerators that use memristor crossbars are a great match.

Of the many hybrid accelerators proposed that use memristor crossbars, some are application specific while others are configurable for a limited set of applications. None of these accelerators are ISA programmable. A key challenge for building an ISA-programmable accelerator is the absence of a software stack to support it. ML applications are developed independently of the underlying hardware or for specific hardware without the notion of interoperability. This is especially true for optimizations at different levels of the stack, including quantization and precision. Moreover, accelerators are considered slave devices without the flexibility to transparently scale solutions up and down, share accelerator devices across applications and users, and offer device-to-device interaction. Finally, in a wide and competitive market with a plethora of ML model libraries, it is required to have a flexible stack that can be optimized at the level of the application (e.g. ONNX model), compiler, device driver, or even the ISA. Without these components, it is not possible to achieve competitive performance across interoperable applications and end-hardware.

To address these challenges, we present a software stack to support hybrid analog-digital accelerators that are ISA-programmable. Our stack supports interoperability with the ONNX standard and includes an optimization layer, a com-

978-1-5386-9170-0/18/$31.00 ©2018 IEEE

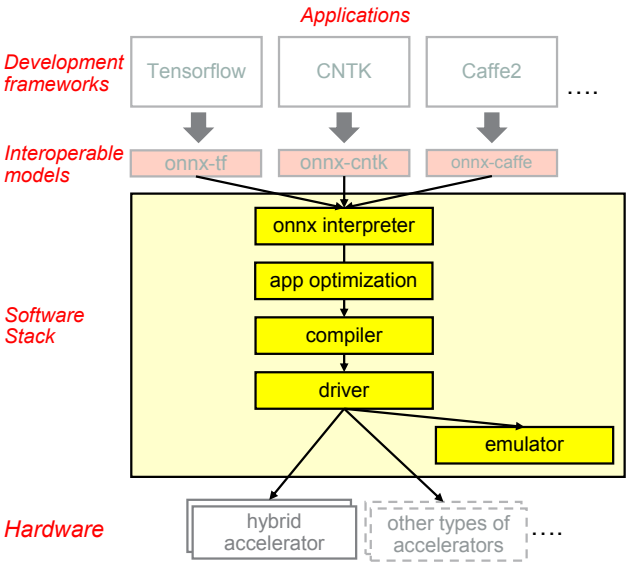

Fig. 1. Hybrid Accelerator Architecture

piler, a device driver, and emulators. An overview of this software stack is illustrated in Fig. 1.

We make the following contributions:

- An ONNX interpreter for translating ONNX models into our native graph representation to enable models developed for a plethora of DL frameworks to use our software stack.
- A set of optimization techniques applied at different levels of the stack to optimize execution for our target class of accelerators.
- A compiler for translating high-level machine learning model representations to an example ISA, mapping the execution of workloads onto cores and tiles.
- An operating system driver that abstracts away the hardware implementation of accelerators and enables the software stack to run several inferences in pipeline and fully customize activation functions to gain performance.
- A set of emulators: (a) *a performance evaluation simulator* that incorporates the basic functionality, timing, and power models of the architecture to enable performance prediction and design space evaluation; (b) *A detailed functional simulator*, that enables hardware development by comparing the state with hardware design tools; and (c) *a plugin into QEMU to enable software development.*

The rest of the paper is organized in the following manner. Section II provides a high-level overview of the target class of hybrid accelerator architectures and an example ISA assumed in this paper. Section III describes the ONNX interpreter. Section IV describes the application optimizations. Section V describes the compiler implementation details. SectionVI describes the driver. Section VII describes the emulator. Section VIII describes and discusses interoperability and accelerator prototypes. Section IX discusses projected performance of our hybrid accelerator. Section X compares our work to related work. Section XI concludes and presents future work.

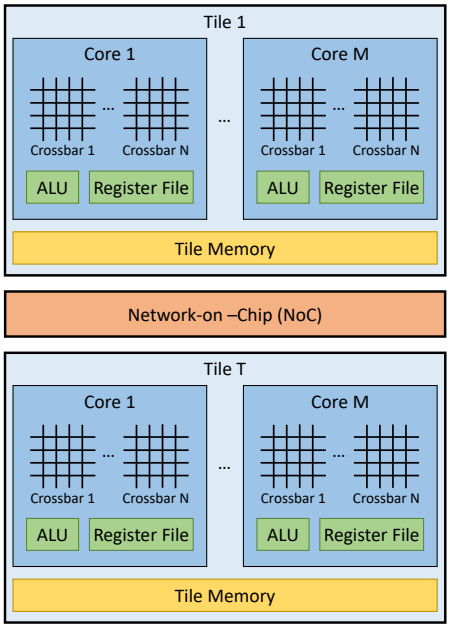

Fig. 2. Architecture Overview

II. Architecture and ISA

This section provides a high-level overview of the abstract hybrid accelerator architecture and example ISA assumed in this paper. Actual accelerator designs may have different implementation details and optmizations. Our objective here is to provide an abstract baseline design to motivate our proposed software stack.

A. Architecture Overview

Fig. 2 shows a high-level diagram of the hierarchical architecture of the hybrid memristor-based accelerator we assume. At the lowest level, N memristor crossbars are grouped into a single core which also contains a register file and an ALU for non-MVM computations. At the next level, M cores are grouped into a single tile with access to a shared tile memory. At the highest level, T tiles are connected via a network-on-chip that enables message passing between tiles within a single node. For large scale applications, multiple nodes can be connected using chip-to-chip interconnect such as CCIX [35], Gen-Z [36], or OpenCAPI [37].

We assume that both cores and tiles in the architecture can execute instructions. An example ISA is described in the following subsections.

B. Core ISA

Fig. 3 summarizes the set of core instructions. The instructions are categorized into compute, data movement, and control flow instructions. These instructions are described in the rest of this section.

Compute Instructions Fig. 3(a) shows the ISA encoding of an MVM instruction which orchestrates the execution of an MVM operation on a crossbar, including digital-to-analog conversion, analog MVM execution, and analog-to-digital conversion. The *mask* operand specifies the crossbars in the core that will be active during the MVM operation. Note

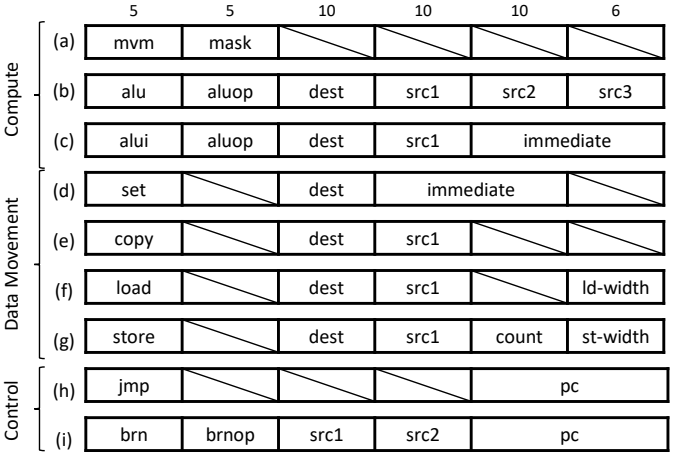

Fig. 3. ISA Encoding of Core Instructions

that the MVM instruction is what uses the analog units in the core (crossbars) while all remaining compute instructions are performed with digital units.

Fig. 3(b) shows the ISA encoding of the *alu* instruction for computing non-MVM vector operations. The *aluop* operand specifies the type of vector operation. The *dest*, *src1*, and *src2* operands are references to the destination and two source registers respectively. The *src3* operand is used to specify the third register for left and right shift operations. Fig. 3(c) shows the ISA encoding of the *alui* instruction which is similar to the *alu* instruction, but takes one immediate operand instead of *src1* and *src2*.

Data Movement Instructions Fig. 3(d) shows the ISA encoding of the *set* instruction that writes an *immediate* to a data memory location. This instruction is used to initialize addresses used by load and store instructions as well as fixed program parameters such as loop bounds. Fig. 3(e) shows the ISA encoding of the *copy* instruction that moves data between the register file and the crossbar input/output registers. Fig. 3(f) and (g) show the ISA encoding of the *load* and *store* instructions respectively for accessing the tile memory. The *count* operand in the *store* instruction specifies the number of times the target memory location will be read before it can be rewritten which is used for synchronization. The *ld-width* and *st-width* operands specify the length of data to be read or written.

Control Flow Instructions Fig. 3(h) and (i) respectively show the ISA encoding of the two supported control flow instructions, unconditional jump (*jmp*) and conditional branch (*brn*). Both instruction take a *pc* with the target instruction address. Additionaly, *brn* takes a *brnop* that specifies the branch condition (equal, not-equal, etc.) and *src1* and *src2* which are operands for the condition evaluation.

C. Tile ISA

Fig. 4(a) and (b) shows the ISA encoding of the *send* and *receive* instructions used to enable communication between tiles via the NoC. The *memaddr* operand specifies the tile memory location where the data to be sent resides or the

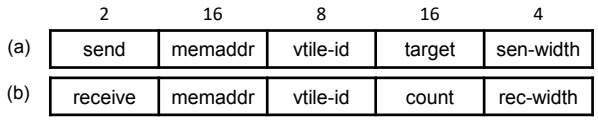

Fig. 4. ISA Encoding of Tile Instructions

data to be received should be written. The *vtile-id* operand specifies the virtual ID of the sender tile with respect to the receiving tile. The *target* operand in a *send* instruction specifies the target tile to which the data is sent. The *count* operand in the *receive* instruction specifies the number of times the target memory location will be read before it can be rewritten, similar to store instructions. Finally, the *sen-width* and *rec-width* operands specify the number of data packets to be sent/received.

III. ONNX Interpreter

The popularity of ML applications, especially neural networks and deep learning, has stimulated the emergence of multiple programming frameworks for supporting these applications. Effectively, these frameworks are Domain Specific Languages (DSLs) for building execution models. Some widely adopted frameworks include:

- Microsoft Cognitive Toolkit (CNTK) [38]: a deep learning framework for Python, C#, and C++ capable of parallelization across multiple GPUs and servers.
- Caffe2 [39]: a deep learning framework initially developed by the Berkeley AI Research group, designed for scale and mobile deployments.
- Tensorflow [40]: Google's framework aimed for flexible deployment across different platforms like GPUs, CPUs, and TPUs [18] on mobile and edge devices.
- Many other frameworks, such as MXNet, PyTorch, Theano, PaddlePaddle, Apache Singa, Apache Mahout, Accord.NET, Brainstorm, etc.

While some of the frameworks target specific hardware platforms, providing better performance, others provide better interoperability across multiple hardware, and yet others provide abstractions for easier model development. However, all of them follow the same computation representation model, a computational graph.

The need to enable model exchange between different frameworks was motivated by the difficulty to optimize performance of frameworks on different hardware platforms. To solve neural network model interoperability, a couple of initiatives where launched to help define an exchangeable format for neural network models [41], [42]. The Open Neural Network Exchange Format (ONNX) [41] has resulted in an initial interest and engagement of the open source community and industry, supporting most of the known frameworks. Therefore, we chose this format to integrate into our software stack solution. ONNX provides a well defined architecture and enough support for a reliable model format to enable interoperabililty with a variety of frameworks for our hardware.

ONNX is defined as an open specification and is organized into two main components: front-end and back-end. The front-end is an interface for framework integration. It defines the

standard types and built-in operators that are supported by the model, and the rules that should be applied for generating a computation graph model from any given framework to the ONNX format. The back-end defines how a framework or runtime platform should read and interpret an ONNX model. Therefore, the software stack solution we propose includes an ONNX back-end which gives us access to models implemented in other frameworks that have ONNX front-ends.

The back-end provides the means through which a framework should interpret and possibly execute a model. Currently, it has been used by multiple platform providers as a means to execute ONNX models on their platform, without the need to translate to a framework representation and then execute. Its interface provides two main methods that should be implemented: *prepare* and *run*, following the general frameworks pattern of executing a neural network model (initialize graph, prepare graph IO, execute graph)

The method *prepare* has as input the model and the graph in the ONNX representation format. Its main objective is to translate the model from ONNX format to the appropriate format of a framework or execution platform, cross-compiling the ONNX code to the desired back-end implementation. Besides the model compilation, in this method many optimizations are applied in order to modify the execution graph to make better usage of the execution platform.

The method *run* provides the interface for loading the model into the platform, input preparation, inference execution, and finally, the collection of the outputs. The IO process provides the means for giving inputs to the graph and reading the outputs, while the execution loads the translated model graph to the desired platform (hardware of software).

In our solution, an initial version of the *prepare* method was developed to interpret ONNX models and generate a native graph representation that can be operated on by our compiler (Section V) to generate ISA code. On the other hand, the initial version of the *run* method was developed to execute the compiled models on our emulator (see Sections VII and VIII-C).

IV. Application Optimization

The software stack presented herein provides multiple layers where different optimization techniques can be applied to improve performance by adapting the models to the underlying hardware. Quantization is required to properly prepare models to execute on the accelerator with little or no loss of accuracy, otherwise, just a simple type casting between default host types (e.g. Float32, Integer32, etc.) to the memristor-based accelerator precision would cause loss of accuracy of the trained weights. Node aggregation and replication aim to make better usage of the dataflow accelerator resources. While the first technique removes unnecessary operations, consequentially, cores and tiles allocation, the second technique makes better usage of the memristor units. This section describes the techniques that were explored and at which level each is applied.

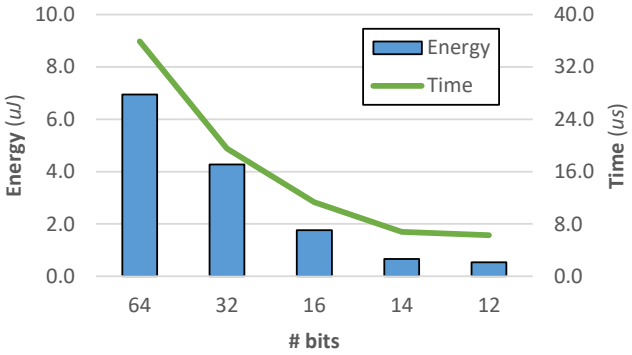

Fig. 5. Quantization Results

A. Quantization

Quantization enables large and complex types (like 32-bit floating points) to be represented using smaller and simpler types (like 16-bit integers) with almost no accuracy loss [43]. It has been widely applied [44], [45], [46], [47] at the edge to optimize models for reduced memory footprint, faster inference and lower energy consumption.

Our software stack enables automatic quantization of pre-trained neural network models based on normalization values provided by the user. For simple models, like MLPs and RNNs, the automated quantization does not significantly impact the accuracy of inference. The tests conducted with a compound GRU-MLP model have shown an accuracy drop of less than 0.1% after applying an 8-bit quantization. For more complex models, like CNNs, it is necessary to perform a more robust calibration in order to fine tune the quantization in order to minimize the accuracy loss. This has not yet been performed and is the subject of future work.

In our stack, quantization is applied at the execution models through specific operations, such as:

- clip(vector, min, max) - Saturate values greater than max and less than the min to max and min respectively.
- normalize(vector) - adjust values distribution

These operations are added to the model whenever it is necessary to calibrate tensors between layers.

Fig. 5 shows the impact of quantization (number of bits for input and model representation) on energy consumption and execution time for a MLP model. A reduction in the number of bits used for model representation (weight data) results in a proportional reduction in the number of memrsitive crossbars used. A reduction in the number of bits used in the input results in a reduction in number of memristive crossbar operations and reduces the cost of ALU and memory access. Consequently, these enable lower energy consumption and faster execution per inference. These results were obtained using our performance simulator (see Section VII).

B. Node Aggregation

To take the most advantage of the memristor-based accelerator, it is desirable to increase the amount of MVM operations performed by a model. Since many neural network models execute the basic perceptron operation (A) below multiple times, we perform the operations aggregation to become (B)

$$\text{(A)}\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + b_1 \\ w_{21}x_1 + w_{22}x_2 + b_2 \end{bmatrix}$$

$$\text{(B)}\begin{bmatrix} w_{11} & w_{12} & b_1 \\ w_{21} & w_{22} & b_2 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + b_1 \\ w_{21}x_1 + w_{22}x_2 + b_2 \end{bmatrix}$$

Fig. 6. Weights and Biases aggregation

with the proper matrix/vectors modifications (see Figure 6).
(A) vector<input> matrix<weights>+ vector<bias>*
(B) vector<input> matrix<weights>*

In the same idea some layers like batch normalization can be aggregated into the convolution, decreasing the number of non-MVM operations in the model.

C. Layers Replication

To improve hardware utilization and to increase performance, layers of the model can be replicated. Depending on the goal and the availability of resources, specific layers can be replicated or the entire model.

One reason to replicate layers is to balance the dataflow architecture's pipeline [24]. In some types of neural networks, the last layers depends on data from the previous layers. For example, computing a convolutional layer requires an amount of data from previous layers that at least matches kernel size. When the pipeline is not balanced, the last layers tend to stay idle several cycles waiting for data to be produced by the earlier layers. This idea can be generalized to replicate any layer that is causing performance bottlenecks in the pipeline while there are unused or dormant resources available.

When the pipeline of the model is balanced and there are still enough resources, the entire model can be replicated, allowing an increase in the number of inferences per second. For example, suppose a model that fits in a single tile and a node that has a total of 5 tiles. The model can be replicated 4 times, occupying all 5 tiles, allowing the inferences to run in batches of 5 inputs, resulting in 5 outputs per inference. Only a minimal increase in latency is expected due to layer synchronization and data distribution in the beginning and in the end of the inference since the models are independent and do not communicate among each other during the execution.

The degree and granularity of replication can be adjusted to meet the performance and power requirements. For example, only a certain amount in the first layers could be replicated to balance the pipeline, and then whole-model replication could be applied to fill the node, thereby increasing overall throughput.

V. Compiler

The compiler generates ISA code from a graph representation of a neural network model that is either constructed by the ONNX back-end or specified by the programmer via our custom API. The compilation flow is shown in Fig. 7. This section describes each of the compiler stages.

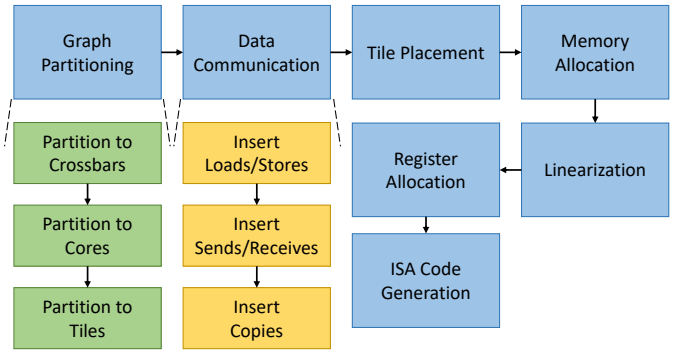

Fig. 7. Compilation Flow

A. Graph Partitioning

In the first stage of compilation, the graph is hierarchically partitioned and the operations in the graph are distributed to different crossbars, cores, and tiles. The hierarchical partitioning uses a bottom-up approach whereby the graph is first partitioned to crossbars, then to cores, then to tiles. The partitioning process starts by assigning all MVM operations that use the same constant matrix to the same *virtual* crossbar. Virtual crossbars, cores, and tiles are used at this stage for separation of concerns; they are mapped to physical crossbars, cores, and tiles at a later stage. Next, each non-MVM operation is assigned affinity to a virtual crossbar by recursively spreading the virtual crossbar affinity from each MVM operation to all source (and destination) operations that feed exclusively into (and out of) that MVM operation. When an operation is reached that has multiple source (or destination) operations each with affinity to different virtual crossbars, heuristics are used to resolve which virtual crossbar to assign those operations to. Once each operation in the graph has been assigned affinity to a virtual crossbar, the first level of partitioning is complete. The graph is now composed of sub-graphs where each sub-graph represents the operations assigned to a virtual crossbar.

The second level of partitioning treats each sub-graph as a node in a graph and aggregates the edges across sub-graphs into a single edge. This new graph is then partitioned, grouping together nodes (i.e., virtual crossbars) that communicate frequently into the same sub-graph (i.e., virtual cores). The partitioning is done by passing the graph to a third-party graph partitioning software such as KaHIP [48] and supplying it with the necessary constraints, namely, the maximum number of nodes per sub-graph (i.e., number of crossbars per core). The second level of partitioning is thus completed.

The third level of partitioning is very similar to the second. Sub-graphs from the second level are treated as nodes, edges are aggregated, the new graph is partitioned whereby each sub-graph represents a set of frequently communicating cores, limited by the maximum number of cores in a tile.

An alternative to this bottom-up hierarchical partitioning approach is a top-down approach where the graph is first partitioned into sub-graphs for each tile, then each sub-graph is partitioned into sub-graphs for each core, and then the same

for crossbars. Exploring this alternative approach is the subject of future work.

B. Data Communication

Once the graph has been partitioned, the compiler inserts data communication operations between producer and consumer operations assigned to different cores and tiles. For all producer-consumer edges going across cores, we insert a store operation on the producer core and a load operation on the consumer core, making sure to avoid redundant operations. If a producer has multiple consumers, only one store operation at the producer core is created and only one load operation per consumer core is created, thereby avoiding redundant load and store operations. After stores and loads are inserted, we identify all store-load edges going across tiles and insert a send operation on the storing tile and a receive operation on the loading tile. If a store has multiple loads, only one send operation and one receive operation is created per loading tile, thereby avoiding redundant send and receive operations. Finally, there is also a need to communicate data across register spaces within a core (crossbar input/output registers and the register file). Whenever there is a mismatch between the register spaces of a producer and consumer operation, intermediate copy instructions are inserted.

C. Tile Placement

Once operations have been assigned to crossbars, cores, and tiles and the data communication has been figured out, the virtual tiles are placed on physical tiles. In this placement, it is important to place tiles that communicate frequently with each other closer together to minimize communication distance. This problem is NP-complete and requires the use of heuristics. The heuristic currently used places the tiles in the order that their matrices are used by the program, which captures the order of layers in the neural network. Therefore, adjacent layers which communicate frequently get placed on adjacent tiles.

Once virtual tiles are mapped to physical tiles, the virtual cores within the virtual tile are mapped to the physical cores in the physical tile. This mapping is trivial because the physical cores within a physical tile are logically equidistant to each other. Finally, the virtual crossbars within a virtual core are mapped to the physical crossbars within a physical core. Again, this mapping is trivial because the physical crossbars within a physical core are logically equidistant.

D. Memory Allocation

Memory allocation is performed by allocating a new tile data memory location for every store and receive operation performed on a tile. The compiler does not currently support reuse of memory locations. Doing so would require a tile-wide analysis of load and store order after linearization (Section V-E) to prevent data hazards. We leave this optimization as future work.

E. Linearization and Register Allocation

The next stage of compilation is linearization. In this stage, the graph is linearized into a sequence of instructions for each tile and core. Linearization ensures that source operations are placed before destination operations in the instruction sequence to guarantee correctness.

Up to this point, virtual registers have been used. Once linear instruction sequences have been generated for each core, it becomes possible to analyze the live ranges of the virtual registers. Doing so enables register allocation for each core with register reuse for non-conflicting live ranges.

F. Code Generation

The final stage is to generate assembly code for each tile and core from the linearized instruction sequences. We have leveraged the standard 64-bit Executable and Linkable Format (ELF) file format specification to develop an ELF format for NN applications which includes the weights, biases, and activation functions that can be accessed by the runtime. The compiler can also optionally generate a plain-text assembly listing, which can be compiled into an ELF executable using the standalone assembler. The loader has been architected to understand this ELF format and works in a tightly coupled fashion with the driver to load instructions and data from the ELF executable to the device, abstracting away device specific features.

G. Error Handling

The DSL and the compiler provide several value adds and failsafes to aid rapid software development, while safeguarding the programmer from common programming pitfalls. For example, the compiler automatically detects input/output tensors without the programmer having to explicitly declare them as such. Further, the compiler diagnoses unused tensors, thus avoiding inadvertent programming errors and conserving precious device memory. The DSL also implements safeguards against out-of-scope tensors used in the model, thus preventing hard to detect runtime issues.

VI. Driver

Most accelerators are implemented as devices on a PCI card connected to a PCIe bus. Therefore, a device driver is required to provide access to these accelerators. The device driver is the layer in the software stack responsible for managing the accelerator. It enables loading tile instructions, core instructions, weights in the crossbars as well as sending input data to the device and returning back the output data to the software stack. This way, it manages the inference execution.

Besides the functionality typical of a device driver, such as providing access to the device, its status, performance data, as well as sharing the device among several running applications, our device driver is designed to keep the maximum usage of the device increasing the throughput for streaming-based application or in-batch executions. The device driver manages a pipeline of running inferences to reach that goal. So, instead of waiting for completing an inference before starting running

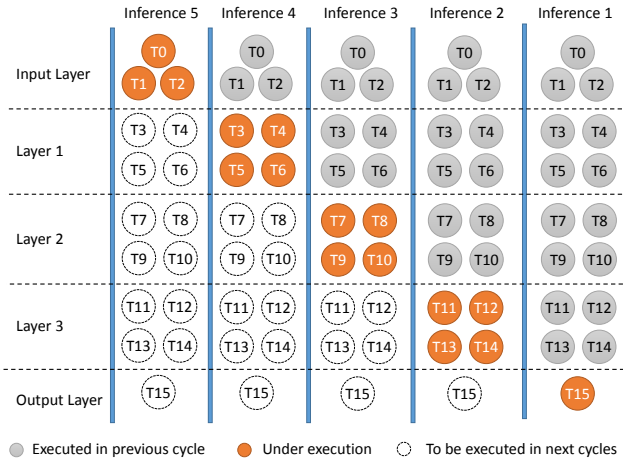

Fig. 8. Inferences running in pipeline

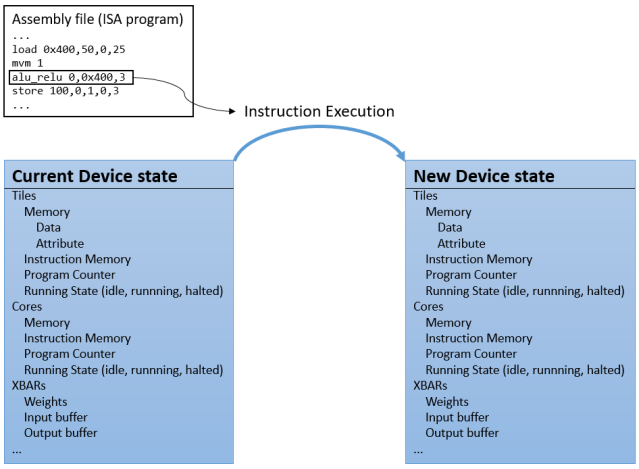

Fig. 9. Single instruction execution

another one, the device driver monitors when the initial tiles get available as soon as the running inference goes deeper in the neural network layers, and it starts running another inference (see Fig. 8). Doing this in the driver simplifies the job of the programmer and compiler because they do not have to worry about implementing/generating streaming logic.

In addition, the device driver constantly monitors the device to ensure the concurrent write of inputs and read of outputs does not result in any output or input data loss. Once the PCIe bus becomes a bottleneck, the device driver has to hold starting the next inference to ensure no overwrite of the data that is running inside the device.

Another important aspect of the device driver is the customization of activation functions. Such activation functions are implemented as look-up tables to save process time and avoid unnecessary complexity. The device driver allows the software stack upper-layer to load fully customized look-up tables. This way different values and different ranges for each activation function can be dynamically adjusted. The look-up tables can be customized per core of any tile given a fully customized mechanism to address the needs for an application.

VII. Emulators and Simulator

We have implemented two types of emulators and one simulator to enable the development and optimization of the software stack. These components are key to verify the performance and behavior of the hardware, architecture, and ISA of memristor-based accelerators. Fig. 1 shows how the emulators fit in the software stack architecture.

A. Performance simulator

To evaluate characteristics of the hardware, such as performance and power utilization, we have developed a hardware simulator. The simulator simulates the execution of ISA instructions, such as those presented in Section II, to collect information about performance and energy and to enable design-space exploration. It was used, for example, to measure the data reported in Fig. 5 and 11.

B. Functional emulator

The functional emulator implements only high level hardware components, such as cores, tiles, and memories with focus on behavior. The emulator executes one instruction at a time following an ISA specification. It abstracts sub-cycles or internal pipeline specific of hardware architecture implementation. A state of the emulated device is stored and each instruction modifies the device to a new state. Examples of elements that compose the device state are: data and metadata of tile memories, register files, crossbar weights, input and output registers of crossbars, tile and core program counter (PC) register, etc. (see Fig. 9.)

The hardware characteristics such as the number of tiles, number of cores per tile, tile memory size, register file size, and crossbar size are configurable parameters. This flexibility can be used for design exploration, experimentation of parameters, tuning compiler heuristics, etc.

The functional emulator was used to collect the data in Fig. 12. It helps evaluate the correctness of models developed by checking whether the results of inferences have expected values.

C. QEMU emulator

A QEMU emulator implements the interface for QEMU hypervisor, emulating the accelerator as a PCI device. The QEMU hypervisor has also been enhanced to take advantage of device characteristics. This includes support for Configuration Space Registers (CSR) in the device and in PCI space. A full-fledged system emulator also helps to anticipate hardware architecture requirements by the software stack evaluating the functionality required by NN models. We used the QEMU emulator to verify the driver functionality and emulate end-to-end integration of the stack.

VIII. Implementation

A. Memristor-based Prototype Implementation

We developed a cycle-level simulator written in Python that implements the abstract architecture and example ISA for the hybrid analog-digital architecture described in Section II. The hybrid architecture executes MVM instructions

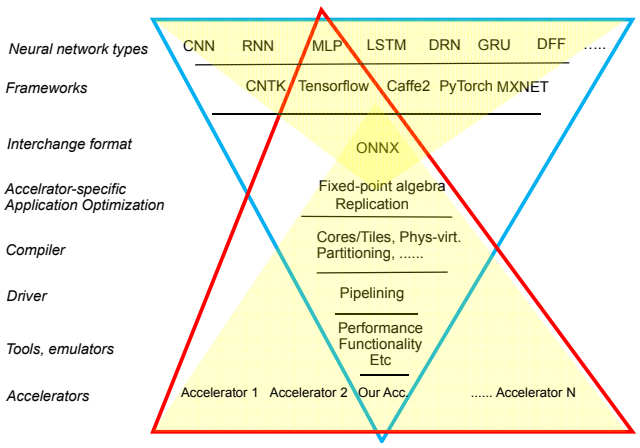

Fig. 10. Interoperability

in analog domain using memristive crossbars. The physical implementation of this hybrid system has been experimentally demonstrated in [34] and fully supports the assumptions in the present work. A memristive crossbar sums up the currents from each cross-point to a column (Kirchhoff's law), where the cross-point's current output is the product of input voltage and programmed conductance (Ohm's law). Additionally, each crossbar is interfaced with Digital-to-Analog converter (to feed digital input from SRAM buffer) and Analog-to-Digital converter (to obtain digital outputs for subsequent computation and communication). The simulator consists of a behavioral model (functionality, power and timing) for MVM instruction using input bit-streaming and weight-slicing as proposed in [24]. ALU instructions are executed on digital CMOS units. The datapath was designed at RTL-level in Verilog and synthesized at IBM 45nm SOI technology to extract the power, area and timing models. Subsequently, the model was integrated with the simulator for evaluation of practical sized workloads.

B. All-digital Prototype Implementation

We have also developed a fully digital implementation of this architecture, and tested it on an FPGA. This can be easily ported to an ASIC fabricated in a conventional CMOS process. The only difference between fully digital and hybrid analog-digital implementations is in the execution of MVM instruction. While in the memristor implementation an output vector element can be calculated in a single clock cycle by adding electrical currents across all memristor row elements connected to a given column, in a digital implementation we step over input rows sequentially. In each step we accumulate in a digital adder the result from the previous step with a product of the present row input and column weight, computed by a digital multiplier. The weights are read from an on-die SRAM memory. Since different matrix columns are independent and use the same row inputs, in both implementations we compute multiple output vector elements in parallel. The micro-architectural details of this implementation are transparent from the core level perspective. The only difference is in the execution time at the same clock frequency, the MVM instruction takes longer to complete in the digital implementation.

TABLE I
SOFTWARE STACK REUSABILITY

Stack	Accelerator-specific	Reusable
Application optimization	layers replication (to balance pipeline)	quantization node aggregation model replication
Compiler	linearization register allocation code generation	graph partitioning data communication tile placement
Driver	accelerator inner cntrl pipelining	OS to accelerator
ISA	inter-core data movement	compute; control; intra-core movement
Architecture	MVM w/ crossbars; ALU inter-core synchronization	instruction exec. pipeline architecture hierarchy

C. Interoperability

ONNX standard enables interoperability between different neural network frameworks on one side and a plethora of accelerators on the other (see two yellow triangles on Fig. 10). Models developed on one framework can be exported and then imported into another one. Similarly, plugins for different accelerators enable development of models only against ONNX and then running them on all accelerators for which there are ONNX plugins.

Our primary motivation for using ONNX was to leverage models developed on multiple frameworks (see blue triangle on Fig. 10). It was less so to leverage the software stack across other accelerators (see red triangle on Fig. 10). We have experience in using our software stack only for the two prototype implementations (memristor-based and all digital). To give some preliminary assessment of what parts of software stack are specific to our accelerator and which are re-usable across other accelerators, we broke down the implementations of stack components in Table I.

D. Discussion

Characterization of the digital implementation performance on an FPGA will be one of future steps. Preliminary projections from FPGA to an ASIC look encouraging. Thanks to the single-step analog computation of output vector elements, the memristor implementation will achieve even higher performance at lower power consumption. This is despite compromises described in [24] to reduce memristor resolution and ADC precision requirements. During FPGA development, we found it important to optimize hardware for fast execution of load, store, copy, and alu instructions, to prevent them from becoming performance bottlenecks. This has been done by commonly used techniques–composing memories from multiple banks, parallelizing data accesses, optimizing datapath widths, and pipelining alu execution. Similarly to other many-core compute architectures, the ISA and RTL implementation need to be simple to avoid excessive logic gate counts and chip area, which would reduce performance per Watt. We found it straight forward to leverage from parametrized RTL building blocks (e.g., an instruction unit, execution control unit, etc.) developed previously for other computing applications, suggesting that the ISA architecture is not unusual or difficult to implement in hardware.

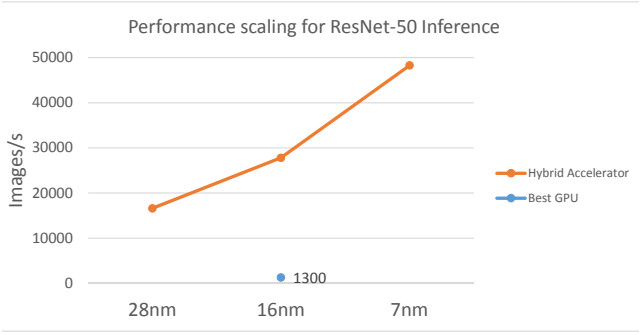

Fig. 11. Projected performance scaling of hybrid accelerator in commercial silicon process at 50W power limit for ResNet-50 Convolutional Neural Network inference, compared with NVidia Tesla P4

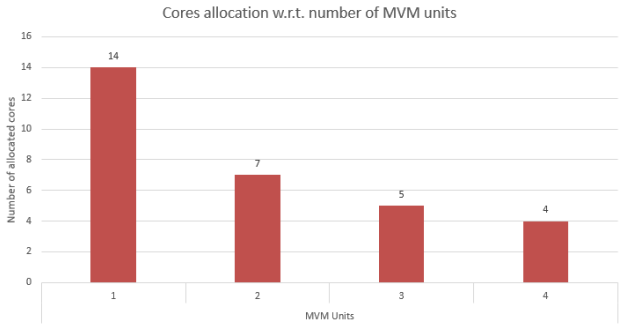

Fig. 12. Effect of number of Matrix Vector Multiplication Units in Cores allocation for a GRU-MLP test model in functional emulator

IX. Results

Fig. 11 shows projected performance of our hybrid accelerator at 50W power limit for ResNet-50 Convolutional Neural Network inference at 28, 16, and 7nm silicon process nodes. For comparison, performance of one of the best GPU inferencing accelerators for the edge currently available on the market (NVidia Tesla P4) is also shown. To give conservative comparison, we show GPU results at batch size 128, which is optimal for it's performance. The GPU performance will be lower at smaller batch sizes because model weights may need to be re-loaded from a cache or an external memory for each batch. Thanks to storing weights in-situ, our accelerator does not require batching of input data. This leads to additional advantage: low inference latency. ImageNet data set is assumed with 224x224 image size. The steeper scaling from 16 to 7nm is due to higher power efficiency at 7nm. Note that at 16nm, hybrid accelerator is projected to be 20x better in performance than GPU, at less than 50% of GPU die size. Although we expect further GPU architecture improvements in the next 1-3 years, we believe that hybrid accelerator can maintain 10x performance advantage at 7nm. Accelerator implementation with hybrid approach will be lower in cost thanks to smaller die size and no need of an external DRAM memory (because all weights are stored on-die).

In general, for neural network applications, we achieved latency between 10 and 10^4 times better than CPUs and between 10 and 10^2 better than GPUs; we achieved bandwidth better than $10^3 - 10^6$ times compared to modern CPUs and more than 10 times better than a modern GPU (when compared at the same power) at a significantly lower cost. All of these projections were conducted using performance simulator.

In Fig. 12 we show results of resource allocation for a compound six layers GRU-MLP model execution in the functional emulator of a hybrid accelerator. With the proposed software stack we are building it is possible to establish the number of Cores and MVMs, and also adjust the size of MVMs to be used by an application, by the compiler and at the functional emulator parametrization. This allows flexibility in resources allocation for partition, distribution and parallelization of models and layers to cores and tiles, as well as for communication and power management of the accelerator subsystems.

The performance advantages reported here could not be achieved without a sophisticated end-to-end software stack.

X. Related Work

There are two efforts towards interoperability of neural network models, ONNX [41] and NNEF [42]. ONNX is a consortium driven by Microsoft, Facebook, and Amazon with the original goal of exchangeable models across development platforms. As recently they also focus on the target hardware platforms. They also explore issues, such model optimization, training of interoperable models, support for closed loops, test and compliance with the standards, etc. These topics are addressed by special working groups. The Khronos Group Inc. released an NNEF specification [49]. NNEF encapsulates network structure and network data. Format is both human readable and parseable.

Many frameworks optimize the execution of ML workloads on traditional systems, including DjiNN [50], an infrastructure for DNN as a service, Caulfield et al. [51], a cloud architecture for executing ML workloads on FPGAs, vDNN [52], a memory manager for virtualizing DNN memory usage for CPU-GPU collaboration during training, PMEM [53], a processor cache optimized for image processing, and Scalpel [54], a system for performing SIMD-aware pruning of DNNs. Our software stack targets special purpose accelerators for ML, particularly those that use memristor crossbars.

Many hybrid accelerators that use memristor crossbars have been designed [19], [20], [21], [22], [23], [24], [25], [26], [27], [28] but only some are configurable for a limited set of applications and none are ISA programmable. NEUTRAMS [55] is a software stack for targeting configurable hybrid accelerators. Our software stack targets ISA-programmable hybrid accelerators. We have evolved our stack by building on our prior work on this topic [56], [57].

There have been many digital-only accelerators designed for machine learning applications [13], [14], [15], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67]. A comprehensive survey and categorization of these works has been done by Sze et al. [68]. Our software stack targets hybrid digital-analog accelerators. Some digital-only accelerators are fully programmable via an ISA [69], [17], [70], [71], [18]. Some can be configured with FPGAs [72], [73], [63], [67], [74], [75], [76], [77]. Our

software stack targets ISA-programmable hybrid accelerators and includes a digital-only implementation on FPGA. Many works propose additional architecture optimizations for digital-only accelerators such as reducing weight/input precision [78], [79], pruning weights to exploit sparsity [16], [80], [81], [82], [70], [83], [84], [85], and others [86], [87], [88], [89], [90], [91]. Our application optimization layer reduces precision of models to make them suitable for memristor-based accelerators.

Several works exploit near-memory computing using DRAM [92], [93], [94] and SRAM [95], [96]. Erudite [97] discusses rebooting the data access hierarchy to leverage near-memory acceleration for cognitive applications. Our software stack targets near-memory accelerators that use memristor crossbars.

Memristors have also been used for building large byte-addressable non-volatile memories for mainstream general-purpose machines. Software support has been proposed for such memory systems tackling issues of consistency in the presence of failure [98], [99], [100], [101] and persistent object representation [102], [103], [104].

Device drivers have always been complex software components to develop due to the delicate timing and performance issues, such as asynchronous behavior, delay dependencies, race conditions, protocol violation due to the delays, throttling of in/out going data, concurrency bugs at hardware/software/firmware levels, and many others [105], [106], [107], [108], [109], [110]. However they can also be enablers in development of both hardware and software, especially using virtualization techniques [111], [112], [113], [114].

There are numerous simulators at different scope, accuracy and for different purposes [115], [116], [117]. In our work, we have come up with a family of three simulators that fit different phases of development (hardware, software, performance): architectural emulator, QEMU simulator, and performance simulator.

XI. Conclusion

In this paper, we present a complete software stack for a programmable accelerator that uses hybrid CMOS-memristive technology for energy-efficient acceleration of machine learning inference workloads. The software stack includes an ONNX back-end for importing models from popular deep learning frameworks, an application optimizer, a compiler for generating an executable binary, a device driver for loading weights and instructions as well as managing streaming workloads, and a functional emulator to assist with performance and energy estimation and design space exploration.

The current stack is primarily targeted at inference accelerators, as we believe this is where energy efficiency of hybrid accelerators pays off first. Inference applications are increasingly being deployed away from datacenters and to the "edge", where small devices are space and energy constrained. However, hybrid accelerators are also being proposed that support training. For this reason, our future work is to extend our software stack to support programming training workloads.

References

[1] Y. Tsividis, "Not your father's analog computer," *IEEE Spectrum*, vol. 55, pp. 38–43, February 2018.

[2] G. D. Mccann, C. H. Wilts, and B. N. Loganthi, "Nonlinear functions in an analog computer," *Electrical Engineering*, vol. 69, pp. 26–26, Jan 1950.

[3] R. M. Gardner, "Analog computer simulates heart response to nerve stimulation," *Electrical Engineering*, vol. 80, pp. 979–982, Dec 1961.

[4] L. B. Wadel and A. W. Wortham, "A survey of electronic analog computer installations," *IRE Transactions on Electronic Computers*, vol. EC-4, pp. 52–55, June 1955.

[5] R. H. Spiess, "Two reasons why a controls laboratory needs an analog computer," in *1986 American Control Conference*, pp. 398–400, June 1986.

[6] T. A. M., "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1936.

[7] J. von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[8] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 4th ed., 2008.

[9] A. Hausner, *Analog and analog/hybrid computer programming*. Prentice-Hall, 1971.

[10] G. Birkel, "Hybrid computers for process control," *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, vol. 79, pp. 726–734, Jan 1961.

[11] J. A. Gibson and T. W. Marks, "Fast hybrid computer implementation of the dynostat algorithm," *IEEE Transactions on Computers*, vol. C-21, pp. 872–880, Aug 1972.

[12] F. Vithayathil, "Hybrid computer simulation of wind-driven ocean currents," in *Engineering in the Ocean Environment, Ocean '74 - IEEE International Conference on*, pp. 308–313, Aug 1974.

[13] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 257–260, May 2010.

[14] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE Computer Society, 2014.

[15] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 367–379, IEEE, 2016.

[16] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 267–278, IEEE Press, 2016.

[17] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 393–405, IEEE Press, 2016.

[18] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 1–12, ACM, 2017.

[19] M. Hu, H. Li, Q. Wu, and G. S. Rose, "Hardware realization of bsb recall function using memristor crossbar arrays," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), pp. 498–503, ACM, 2012.

[20] S. G. Ramasubramanian, R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan, "Spindle: Spintronic deep learning engine for large-scale neuromorphic computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*, pp. 15–20, ACM, 2014.

[21] Y. Kim, Y. Zhang, and P. Li, "A reconfigurable digital neuromorphic processor with memristive synaptic crossbar for cognitive computing," *J. Emerg. Technol. Comput. Syst.*, vol. 11, pp. 38:1–38:25, Apr. 2015.

[22] X. Liu, M. Mao, B. Liu, H. Li, Y. Chen, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu, *et al.*, "Reno: A high-efficient reconfigurable neuromorphic computing accelerator design," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2015.

[23] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 1–13, IEEE, 2016.

[24] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, IEEE Press, 2016.

[25] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 27–39, IEEE Press, 2016.

[26] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 541–552, IEEE, 2017.

[27] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, "Time: A training-in-memory architecture for memristor-based deep neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 26, ACM, 2017.

[28] A. Ankit, A. Sengupta, P. Panda, and K. Roy, "Resparc: A reconfigurable and energy-efficient architecture with memristive crossbars for deep spiking neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 27, ACM, 2017.

[29] F. Alibart, E. Zamanidoost, and D. B. Strukov, "Pattern classification by memristive crossbar circuits using ex situ and in situ training," *Nature communications*, vol. 4, 2013.

[30] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, pp. 61–64, 2015.

[31] G. W. Burr, R. M. Shelby, S. Sidler, C. di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, B. N. Kurdi, and H. Hwang, "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element," *IEEE Transactions on Electron Devices*, vol. 62, pp. 3498–3507, Nov 2015.

[32] S. Yu, Z. Li, P.-Y. Chen, H. Wu, B. Gao, D. Wang, W. Wu, and H. Qian, "Binary neural network with 16 mb rram macro chip for classification and online training," in *Electron Devices Meeting (IEDM), 2016 IEEE International*, pp. 16–2, IEEE, 2016.

[33] P. M. Sheridan, F. Cai, C. Du, W. Ma, Z. Zhang, and W. D. Lu, "Sparse coding with memristor networks.," *Nature nanotechnology*, 2017.

[34] M. Hu, C. Graves, C. Li, Y. Li, N. Ge, E. Montgomery, N. Davila, H. Jiang, R. S. Williams, J. J. Yang, Q. Xia, and J. P. Strachan, "Memristor-based analog computation and neural network classification with a dot product engine," *Advanced Materials*, 2018.

[35] CCIX Consortium, "CCIX interconnect." https://www.ccixconsortium.com, 2017.

[36] Gen-Z Consortium, "Gen-Z interconnect." http://genzconsortium.org/about/, 2016.

[37] B. Wile, "Coherent accelerator processor interface (CAPI) for POWER8 systems," tech. rep., IBM, September 2014.

[38] "CNTK: The Microsoft Cognitive Toolkit." https://cntk.ai/.

[39] "Caffe2: Lightweight, Modular, and Scalable Deep Learning Framework." https://caffe2.ai/.

[40] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: a system for large-scale machine learning.," in *OSDI*, vol. 16, pp. 265–283, 2016.

[41] "ONNX: Open Neural Network Exchange Format." https://onnx.ai.

[42] "NNEF: Neural Network Exchange Format." https://www.khronos.org/nnef.

[43] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *CoRR*, vol. abs/1806.08342, 2018.

[44] M. Z. Alom, A. T. Moody, N. Maruyama, B. C. V. Essen, and T. M. Taha, "Effective quantization approaches for recurrent neural networks," *CoRR*, vol. abs/1802.02615, 2018.

[45] Y. Xu, Y. Wang, A. Zhou, W. Lin, and H. Xiong, "Deep neural network compression with single and multiple level quantization," *CoRR*, vol. abs/1803.03289, 2018.

[46] N. Mellempudi, A. Kundu, D. Das, D. Mudigere, and B. Kaul, "Mixed low-precision deep learning inference using dynamic fixed point," *CoRR*, vol. abs/1701.08978, 2017.

[47] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," *CoRR*, vol. abs/1511.06393, 2015.

[48] P. Sanders and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, vol. 7933 of *LNCS*, pp. 164–175, Springer, 2013.

[49] T. K. N. W. Group, "Neural network exchange format." https://www.khronos.org/registry/NNEF/specs/1.0/nnef-1.0.pdf.

[50] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 27–40, ACM, 2015.

[51] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, *et al.*, "A cloud-scale acceleration architecture," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–13, IEEE, 2016.

[52] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–13, IEEE, 2016.

[53] J. Clemons, C.-C. Cheng, I. Frosio, D. Johnson, and S. W. Keckler, "A patch memory system for image processing and computer vision," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–13, IEEE, 2016.

[54] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 548–560, ACM, 2017.

[55] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–13, IEEE, 2016.

[56] P. Bruel, S. R. Chalamalasetti, C. Dalton, I. El Hajj, A. Goldman, C. Graves, W.-m. Hwu, P. Laplante, D. Milojicic, G. Ndu, *et al.*, "Generalize or die: Operating systems support for memristor-based accelerators," in *Rebooting Computing (ICRC), 2017 IEEE International Conference on*, pp. 1–8, IEEE, 2017.

[57] P. Laplante and D. Milojicic, "Rethinking operating systems for rebooted computing," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, IEEE, 2016.

[58] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pp. 53–60, IEEE, 2009.

[59] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 247–257, ACM, 2010.

[60] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings*

of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pp. 682–687, 2014.

[61] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, (New York, NY, USA), pp. 199–204, ACM, 2015.

[62] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H.-J. Yoo, "4.6 a1. 93tops/w scalable deep learning/inference processor with tetra-parallel mimd architecture for big-data applications," in *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*, pp. 1–3, IEEE, 2015.

[63] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pp. 13–19, IEEE, 2013.

[64] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 92–104, ACM, 2015.

[65] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1737–1746, 2015.

[66] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pp. 269–284, ACM, 2014.

[67] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, (New York, NY, USA), pp. 161–170, ACM, 2015.

[68] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *arXiv preprint arXiv:1703.09039*, 2017.

[69] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 369–381, ACM, 2015.

[70] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[71] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 13–26, ACM, 2017.

[72] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of fpga based neural network accelerator," *arXiv preprint arXiv:1712.08934*, 2017.

[73] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 32–37, IEEE, 2009.

[74] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: automatic generation of fpga-based learning accelerators for the neural network family," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[75] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 14–26, IEEE, 2016.

[76] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[77] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 535–547, ACM, 2017.

[78] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[79] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultra-low power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[80] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 1–13, IEEE, 2016.

[81] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 243–254, IEEE Press, 2016.

[82] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 27–40, ACM, 2017.

[83] J. Chung and T. Shin, "Simplifying deep neural networks for neuromorphic architectures," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[84] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 382–394, ACM, 2017.

[85] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "Circnn: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 395–408, ACM, 2017.

[86] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 124, ACM, 2016.

[87] A. Ren, Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan, "Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 405–418, ACM, 2017.

[88] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[89] Y. Wang, H. Li, and X. Li, "Real-time meets approximate computing: An elastic cnn inference accelerator with adaptive trade-off between qos and qor," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 33, ACM, 2017.

[90] Y. Ji, Y. Zhang, W. Chen, and Y. Xie, "Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), pp. 448–460, ACM, 2018.

[91] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), pp. 461–475, ACM, 2018.

[92] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 380–392, IEEE, 2016.

[93] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on*

Architectural Support for Programming Languages and Operating Systems, pp. 751–764, ACM, 2017.

[94] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 288–301, ACM, 2017.

[95] Z. Wang, R. Schapire, and N. Verma, "Error-adaptive classifier boosting (eacb): Exploiting data-driven training for highly fault-tolerant hardware," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 3884–3888, IEEE, 2014.

[96] J. Zhang, Z. Wang, and N. Verma, "A machine-learning classifier implemented in a standard 6t sram array," in *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*, pp. 1–2, IEEE, 2016.

[97] W. H. Wen-mei, I. El Hajj, S. G. de Gonzalo, C. Pearson, N. S. Kim, D. Chen, J. Xiong, and Z. Sura, "Rebooting the data access hierarchy of computing systems," in *Rebooting Computing (ICRC), 2017 IEEE International Conference on*, pp. 1–4, IEEE, 2017.

[98] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 91–104, ACM, 2011.

[99] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 105–118, ACM, 2011.

[100] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, (New York, NY, USA), pp. 433–452, ACM, 2014.

[101] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 677–694, ACM, 2016.

[102] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan, "Spacejmp: Programming with multiple virtual address spaces," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 353–368, ACM, 2016.

[103] I. El Hajj, T. B. Jablin, D. Milojicic, and W.-m. Hwu, "Savi objects: sharing and virtuality incorporated," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 45, 2017.

[104] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu, "Efficient support of position independence on non-volatile memory," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 191–203, ACM, 2017.

[105] J. Tanguy, J. L. Bchennec, M. Briday, and O. H. Roux, "Reactive embedded device driver synthesis using logical timed models," in *2014 4th International Conference On Simulation And Modeling Methodologies, Technologies And Applications (SIMULTECH)*, pp. 163–169, Aug 2014.

[106] Y. Dong, Y. He, Y. Lu, and H. Ye, "A model driven approach for device driver development," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 122–129, July 2017.

[107] S. Sydow, M. Nabelsee, A. Busse, S. Koch, and H. Parzyjegla, "Performance-aware device driver architecture for signal processing," in *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 67–75, Oct 2016.

[108] V. Vojdani, K. Apinis, V. Rtov, H. Seidl, V. Vene, and R. Vogler, "Static race detection for device drivers: The goblint approach," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 391–402, Sept 2016.

[109] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and precise symbolic analysis of concurrency bugs in device drivers (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 166–177, Nov 2015.

[110] A. Kadav and M. M. Swift, "Understanding modern device drivers," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 87–98, ACM, 2012.

[111] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gtz, "Unmodified device driver reuse and improved system dependability via virtual machines," in *In Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, 2004.

[112] D. Eschweiler and V. Lindenstruth, "Test driven development for device drivers and rapid hardware prototyping," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–9, April 2015.

[113] Z. Hao, W. Endong, W. Yinfeng, Z. Xingjun, C. Baoke, W. Weiguo, and D. Xiaoshe, "Transparent driver-kernel isolation with vmm intervention," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, (New York, NY, USA), pp. 2:1–2:16, ACM, 2013.

[114] Y. Sun and T. c. Chiueh, "Side: Isolated and efficient execution of unmodified device drivers," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, June 2013.

[115] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the simos machine simulator to study complex computer systems," *ACM Trans. Model. Comput. Simul.*, vol. 7, pp. 78–103, Jan. 1997.

[116] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "Cotson: Infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 52–61, Jan. 2009.

[117] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Annual Technical Conference, Freenix Track*, pp. 41–46, USENIX, April 2005.