EXPANDING HARDWARE ACCELERATOR SYSTEM DESIGN SPACE EXPLORATION WITH GEM5-SALAMV2

by

Zephaniah Spencer

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2024

Approved by:

_____
Dr. Hamed Tabkhi

_____
Dr. Ronald Sass

_____
Dr. Ke Wang

ABSTRACT

ZEPHANIAH SPENCER. Expanding Hardware Accelerator System Design Space Exploration with gem5-SALAMv2. (Under the direction of DR. HAMED TABKHI)

With the prevalence of hardware accelerators as an integral part of the modern systems on chip (SoCs), the ability to model accelerators quickly and accurately within the system in which it operates is critical. This paper presents gem5-SALAMv2 as a novel system architecture for LLVM-based modeling and simulation of custom hardware accelerators integrated into the gem5 framework. It overcomes the inherent limitations of state-of-the-art trace-based pre-register-transfer level (RTL) simulators by offering a truly "execute-in-execute" LLVM-based model, enabling scalable modeling of dynamically interacting accelerators with full-system simulation support. To create a sustainable expansion compatible with the gem5 framework, gem5-SALAM offers a general-purpose and modular memory hierarchy integrated into the gem5 ecosystem, streamlining designing and modeling accelerators for new and emerging applications. gem5-SALAMv2 expands the framework established in gem5-SALAMv1 with improved elaboration and simulation, system integration, and automation to simplify rapid prototyping and design space exploration. Validation on the MachSuite [1] benchmarks presents a timing estimation error of less than 1% against the Vivado HLS tool. Results also show less than a 4% area and power estimation error against Synopsys Design Compiler. System validation against implementations on an Ultrascale+ ZCU102 shows an average end-to-end timing error of less than 2%. Lastly, we demonstrate the upgraded capabilities of gem5-SALAMv2 by exploring accelerator platforms for two deep neural networks, LeNet5 and MobileNetv2. In these explorations, we demonstrate how gem5-SALAMv2 can simulate such systems and guide architectural optimizations for these types of accelerator-rich architectures.

DEDICATION

To my wife Amanda-Joi and son Finley, thank you for letting me be in a position to do this and future work while bringing joy to my life.

ACKNOWLEDGEMENTS

Thank you to my Advisor, Dr. Hamed Tabkhi, for supporting this project and me throughout my academic career.

Thank you to my committee members, Dr. Ronald Sass and Dr. Ke Wang for their advice and guidance to me throughout my studies.

Thank you to Samuel Rodgers and Joshua Slycord for their mentorship, guidance, and friendship.

Thank you to my parents and siblings Michael, Andrea, Isaac, Della, Emma, and Jordan for their continuing love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI Artificial Intelligence

API Application Programming Interface

CNN Convolutional Neural Network

CPU Central Processing Unit

DDR Double Data Rate

DSE Design Space Exploration

FPGA Field Programmable Gate Array

FPS Frames Per Second

GPU Graphics Processing Unit

HLS High Level Synthesis

IR Intermediate Representation

LLVM Low Level Virtual Machine

MLIR Multi-Level Intermediate Representation

RTL Register-Transfer Level

SALAM System Architecture for LLVM-based Accelerator Modeling

SOTA State of The Art

CHAPTER 1: INTRODUCTION

RTL co-simulation, pre-RTL modeling, and prototyping provide a more cost-effective alternative for system-level design space exploration of accelerator-rich heterogeneous architectures. Leveraging full-system simulators like gem5 [2], these works can offload tracking for most system overheads to a robust and well-supported open-source simulator. The choice between RTL co-simulation models like [3, 4, 5, 6, 7, 8] and pre-RTL models like [9, 10, 11] is essentially a trade-off between simulation fidelity and ease of use. Pre-RTL simulators are generally easier to set up and modify for large-scale design sweeps of both accelerator-level and system-level variables. They accomplish this by abstracting the execution of accelerators with methods like data-flow modeling [12], static timing analysis [11], or by trace-based simulation on instrumented execution traces [9, 10].

These abstractions lead to a minimal loss in fidelity when an accelerator's execution is not dependent on input data, but can lead to significant errors in simulation fidelity when accelerators demonstrate input-driven behaviors [13]. RTL co-simulation models resolve this by running full cycle-accurate RTL simulations on the data input to the accelerator. This allows for modeling dynamic input-based execution at the cost of both ease of use and simulation time. This is because RTL simulation requires both an HDL design and verification tools for the design and manipulation of accelerators. These models also require separate tuning and elaboration whenever an interface is changed, in contrast to pre-RTL models, which can broadly explore sweeps of accelerator design parameters from the same Python interface used by gem5 for its system design sweeps.

One exception to the general trends of pre-RTL simulation vs. RTL co-simulation

is gem5-SALAM [13], which leverages a unique dynamic graph execution engine based on LLVM [14] for modeling hardware accelerators. This enabled SALAM to accurately model the dynamic input-dependent characteristics of accelerators without HDL design and elaboration costs. While its core functionalities allowed for automation of system-level and accelerator-level design sweeps, its somewhat obtuse interface and reliance on LLVM 3.8 could lead to significant limitations in long-term development and maintenance.

To address these concerns, this article presents gem5-SALAMv2. Fig.1.1 presents a generalized full-system architecture model used by gem5-SALAMv2. gem5-SALAMv2 includes a completely redesigned elaboration and execution engine compatible with the latest LLVM releases. This new engine enables new capabilities, including dynamic function inlining, custom and arbitrary precision data types, and significant performance improvements for elaboration and event scheduling. Coupled with improvements to gem5-SALAM's memory interface to gem5, we have enabled more flexible system-level connectivity and the creation of hierarchical memory interfaces. gem5-SALAMv2 also expands on the simplistic compute acceleration modeling of gem5-SALAMv1 by enabling the creation of more complex custom hardware models. This empowers users to integrate advanced memory interfaces and other hardware models that traditionally required handcrafted simulation solutions into gem5's full system simulation through a simple LLVM interface. In addition to these improvements, gem5-SALAMv2 includes new system-level design tools that simplify and automate the development effort required when integrating large numbers of accelerators and supporting devices into a simulated system. In doing so, gem5-SALAMv2 removes most of the burdens of integrating new hardware devices into a full system simulation and replaces them with an intuitive and easily modified front-end user interface.

In summary, the contributions that the gem5-SALAMv2 framework provides are:

- Accurate & performant pre-RTL modeling of algorithms at C/C++ level, represented in LLVM IR
- Integration into gem5 for system overhead modeling
- Modeling of common memories used in accelerator designs, from local scratchpad to shared memory
- Deeply configurable hardware profiles to provide ease of use and deep insight on execution

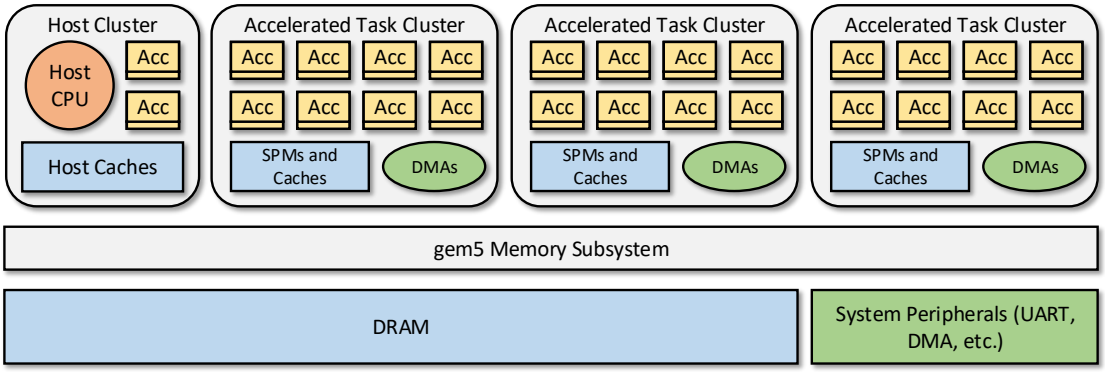


Figure 1.1: *A generalized full-system architecture model used by gem5-SALAMv2. Any arbitrary amount of clusters, accelerators, memory, and hardware devices can be configured and connected as needed for the design space exploration of an application.*

In Sec. 2, we review other related works and discuss some of the shortcomings and challenges present in existing tools that we sought to address. Following that in Sec. 3, we describe the updated structure of gem5-SALAMv2, including under-the-hood improvements and updates to the end-user experience. Sec. 4 details the validation of the updated gem5-SALAMv2 accelerator models and provides examples of how to leverage gem5-SALAMv2 for exploration of application-specific architectures for convolutional neural networks such as LeNet-5 [15] and MobileNetV2 [16] discussed in Sec. 5.

CHAPTER 2: RELATED WORK

Modern design flows have shifted from classical RTL design flows to more software developer-friendly paradigms, moving away from pure HDL designs. The LLVM compiler and IR [14] have quickly become a staple of modern Synthesis and Design Space Exploration (DSE) tools. Vivado and LegUp [17], both HLS tools, utilize modified Clang toolchains to translate their C hardware descriptions into industry-standard RTL targets such as Verilog, VHDL, and SystemC. Due to the explosion of interest in deep learning (DL) applications, several projects have begun working on exploring DL-specific accelerators, such as the LeFlow project [18]. LeFlow was designed to integrate TensorFLow's XLA compiler with LegUp to create an easier workflow for implementing deep learning accelerators on FPGAs. In addition to synthesis tools, LLVM has also been employed for pre-RTL DSE. Both the RIP framework [19] and Needle [20], leverage LLVM for the identification and modeling of "hot" portions of the Control and Dataflow Graph (CDFG) in a given application to generate accelerators for DySER-styled [21] architectures. The Lumos+ [22] and LogCa [23] tools employ analytical modeling on loosely coupled accelerators to estimate power, performance and area requirements in highly heterogeneous accelerator-rich systems.

Another tool that leverages LLVM for parsing and instrumentation is the MosaicSim tool [12], which offers a lightweight simulation of heterogeneous systems comprised of CPUs and accelerators. MosaicSim roughly models hardware accelerators as simple in-order or out-of-order compute cores, providing configuration options like instruction issue width, re-order buffer size, and load-store queue (LSQ) size. Although this can provide rough approximations for the computing performance of an accelerated segment of an LLVM IR computing graph, it lacks many of the fine-

grained controls needed to model or profile an application-specific datapath. For instance, users cannot model hardware constraints such as functional unit re-use or even adjust the timing/power/energy costs of individual elements of the compute datapath. The tool also supports integrating RTL models generated by HLS tools. However, these RTL models are not integrated into the same tiled memory model of the more abstract models. MosaicSim relies on pre-generated execution traces for the modeling of accelerators. In MosaicSim, this includes both a trace of load/stores ops and a control-flow trace to track the progression of basic blocks. Reliance on execution traces imposes several significant restrictions on modeling in MosaicSim. Trace sizes, generation times, and read times introduce significant simulation overhead. For extensive and long-running applications, traces may exceed several gigabytes. Additionally, for applications where input data dictate the load/store and control flow behaviors, users need to generate new traces for each new input data set.

As mentioned in Sec. 1, other existing pre-RTL solutions for exploring the system-level integration of accelerators are gem5 [2] and its derivatives gem5-Aladdin [10] and PARADE [11]. While gem5 offers a high degree of flexibility in system-level design space exploration, it lacks any base models for integrating application-specific hardware accelerators. gem5-Aladdin and PARADE heavily constrain the design space to align with their particular simulation semantics to offer modeling capabilities similar to gem5-SALAM. These same limitations are reflected in the accuracy of the accelerator model generated, in which data availability, compute parallelism, and timing are independent of the input data and system hierarchy.

In the case of gem5-Aladdin, the simulated accelerator datapath could change dramatically simply by changing the input data of some applications, even going so far as introducing entirely new execution paths and functional units if some control paths were entirely data dependent [13]. Changing the Aladdin model's datapath parallelism is possible by adjusting a system parameter like cache size [13]. Also, gem5

now supports directly integrating models in SystemC [3]. Although a more significant design effort is required when using an RTL-based option, this also offers the most opportunities for design space exploration and simulation.

In addition to the RTL and pre-RTL simulators previously discussed, there are also domain-specific simulators such as STONNE [24], SCALE-Sim [25], and others [26, 27, 28, 29] that model specific datapaths or utilize analytical models improve simulation time. gem5-SALAMv2 does not attempt to optimize its framework for a specific set of algorithms and focuses on flexibility and scalability within the gem5 framework. Additionally, gem5-SALAM is not built to simulate generalized datapaths like the systolic array-based NVDLA, Coral Edge, etc. These are domain-specific platforms that rely on domain-optimized programmable compute elements and gem5-SALAM does not model data paths consisting of general ALUs. It instead models highly customized and deeply pipelined application-specific data paths. gem5-SALAM is unique because you can evaluate acceleration potential while accounting for system-level overheads.

Not only can you sweep the parameters of an accelerator, you can also explore various memory and other system-level integrations. These are important considerations that can impact continued RTL development. Other simulators do not offer this, and even RTL-based design flows can only really explore these kinds of design considerations later in their development. As with most engineering questions, there will be a trade-off between simulation time, ease of use, and fidelity across these simulators, but we feel that gem5-SALAMv2 is able to find a meaningful balance between these.

2.1 Improvements over gem5-SALAMv1

The gem5-SALAM project was initially developed to address the shortcomings of other pre-RTL simulators in modeling accelerators with run-time and input-dependent behaviors. The datapath modeling scheme employed by gem5-SALAM was first proposed in "Scalable LLVM-Based Accelerator Modeling in gem5" [30] and was used

Table 2.1: Shortcomings present gem5-SALAMv1 and Solutions/Features Introduced by gem5-SALAMv2 to address these.

gem5-SALAMv1	gem5-SALAMv2
• Locked to LLVM 3.8 • Manual System Configuration • Single Function Accelerators • Fixed Memory Interfaces • Single Integrated SPM • Statically Defined Memory and Hardware Units • Simplified and Flattened IR Structure • Support For Standard Datatypes	• Supports LLVM 9 and newer IR • Automated System Configuration • Enables Multi-Function Accelerators • Configurable Accelerator Memory Interfaces • User-Defined Memory Hierarchies • Dynamically Generated Hardware and Memory Units • LLVM IR Structure Preserved • Support For Custom and Arbitrary Datatypes

to provide timing models of simple, single accelerator systems. gem5-SALAMv1 expanded on this simple timing model by adding power estimation, area estimation, and validation of complete system performance against FPGA implementations of the same benchmarks [13]. While the initial gem5-SALAMv1 offered flexibility in system design points, this came at the cost of overwhelming users with numerous switches and configuration options as the complexity of a given system increased. Working with SALAMv1 became a significant challenge due to the development cost of maintaining memory mapping and gem5 configuration binds across dozens of devices. In the case of the MobileNetv2 example described in Sec. 5, users would need to map well over 150 different memory-mapped accelerators, memories, and DMA devices by hand. Errors in that memory map would then propagate through more

than a dozen unique configuration files that must be debugged concurrently. This significantly hindered the exploration of complex hardware architectures and greatly influenced the design of SALAMv2.

In addition to configuration issues, gem5-SALAMv1 [13] was also limited in accelerator complexity scope by limitations shown in Table 2.1. To broaden the scope of systems that could be modeled, we have performed an extensive upgrade on gem5-SALAMv1 driven by three primary design goals.

1. *Simplify the design of complex systems while expanding the number of configuration controls and profiling options available to users.*

2. *Increase extensibility and scalability of SALAM's LLVM runtime models through a complete overhaul of the LLVM elaboration and simulation runtime.*

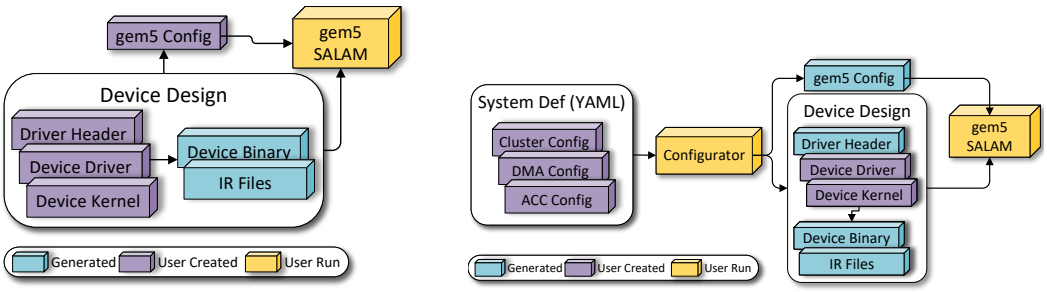
3. *Improve upon the flexibility of the system integration present in gem5-SALAMv1 by adding new interfaces and control/synchronization paradigms that were not present in the previous version.*

These improvements and additions allow for modeling far more complex systems, as shown in Fig. 1.1. All features of gem5-SALAMv1 have been carried over to the new iteration, and improvements have been made to every aspect of the original simulator. This upgrade to the core framework has allowed our team to significantly expand the system's capabilities while increasing performance and decreasing the runtime of large simulations. The most prominent upgrade from gem5-SALAMv1 is the full integration of LLVM into our elaboration engine. By embedding LLVM internally, we can now utilize the entire LLVM API inside SALAMv2. This allows us access to all of the information in the IR and uses the LLVM framework for analyzing data structures within the application to optimize scheduling and our CDFG generator, while also no longer being bound to a specific version of LLVM.

Additionally, the internal use of the LLVM API has extended the scope of our modeling capabilities. In gem5-SALAMv1, the simulation was limited to single application in-lined accelerators, but in gem5-SALAMv2, we can now model multiple applications and support internal function calls. To further provide support from the LLVM API, we have decoupled the instructions and hardware components of the simulator to maintain functionality with agnostic versions of LLVM and provide a means to reconfigure and add new components without having to modify the code.

This functionality is supported by a greatly expanded Communications Interface (CommInterface), which in

SALAMv1 was bound to a specific application and was an independent entity that ran parallel to the simulation engine. In gem5-SALAMv2, we have fully integrated the CommInterface into the gem5 ecosystem and expanded its functionality, as shown in Fig. 3.4. gem5-SALAMv1 also relied on a single integrated scratchpad memory (SPM) that had to be manually configured for each application. With the introduction of the CommInterface, gem5-SALAMv2 can have multiple memory types that can have any arbitrary connectivity as desired by the user. For an example of this interconnectivity, see Fig. 5.2.



(a) SALAMv1 Design FLow

(b) SALAMv2 Design Flow

Figure 2.1: *Showcases the difference in design flows from SALAMv1 to SALAMv2. Here, we can see that a designer would be required to implement their system's "front-end" in gem5 by hand with the need to update this front-end with each design iteration. In the SALAMv2 flow, this process is automated by the SALAM Configurator, removing the need for manually created and updated gem5 configuration files and design headers.*

CHAPTER 3: gem5-SALAMv2

gem5-SALAMv2 is a large collection of improvements, new features, and additional tools added to gem5-SALAM. All of these combine to enable large-scale systems to be explored while having a pragmatic approach to designing these systems. Our description of these is broken out into the following sections:

- 3.1 - *An overview of the tool-assisted system setup and initialization process*
- 3.2 - *A description of how SALAM analyses LLVM IR for simulation setup*
- 3.3 - *Covers how LLVM IR parameterization has been improved upon over v1*
- 3.4 - *Insights into the redesigned run-time engine, evaluation methodology, and the benefits of our LLVM API integration*
- 3.5 - *An in-depth description of how these components have been integrated into gem5*

3.1 System Setup and Initialization

One of the most significant upgrades in gem5-SALAMv2 is the introduction of new automation tools to build and explore accelerator-rich systems. We have provided two sets of tools to aid in the design process. The SALAM Configurator (3.1.1) is designed to automate much of the boilerplate required when writing a SALAM benchmark in gem5. This generates the full system gem5 configuration script and a C memory map that a user can include in their design. We have also built a set of HW Tools (3.1.2) that provide front-ends for configuring elaborated functional units and integrating power and area tracking into gem5-SALAM. This enables the automation

of generating new power profiles and configuring an accelerator's functional unit and instruction parameters through a defined interface.

3.1.1 SALAM Configurator

In gem5-SALAMv1, users followed the design flow illustrated in Fig. 2.1. This design flow put the burden of creating gem5 system configurations and maintaining memory map headers for their benchmarks manually. Consequently, the complexity of implementing and exploring large-scale systems in gem5-SALAMv1 could quickly become unmanageable from a development effort standpoint. This limitation in design came from v1's development being solely focused on exploring how to model multi-accelerator systems while still requiring development outside of SALAM to create a functioning simulation. This resulted in an end-user being required to modify their configuration files and design headers each time devices were added or sizing parameters changed. Although this worked fine for small single-function systems, such as our validation benchmarks, a system at the scale explored in Sec. 5.2 contains over 150 memory-mapped devices and was infeasible to design, develop, and debug in gem5-SALAMv1. To make large-scale design space exploration possible for users, system design tasks have been simplified, unified, and automated with the release of gem5-SALAMv2 through the SALAM Configurator, as shown in Fig. 2.1.

The initial step in creating a new accelerator-rich system within SALAMv2 begins with the development of a system structure description. This takes the form of a YAML file, visualized in Fig. 3.1, in which the user declares the names and types of devices they want to simulate in the SALAM framework, including clusters, accelerators, and DMAs. Users can define four different types of devices in the YAML file: Accelerator Clusters (acc_cluster), Accelerators (Accelerator), DMAs (DMA), and Variables (Var). Each of these provide their own configuration interface, with the one constant being "Name" for setting both the SimObject and memory map name of the device.

The accelerator cluster is meant to be a localized collection of Accelerators, DMAs, and vars that are defined within it. Accelerators define a CommInterface that is able to be connected to any device within the cluster, including the local bus. DMAs allow for the configuration of two DMA types, either NonCoherent or Streaming and provide connectivity and buffer sizing parameters. A user can also assign memory devices (vars) to an accelerator or accelerator cluster by declaring a variable under it and providing its size and type. Supported memory types include multi-port scratchpads, register banks, memory streams, and caches.

When the user finishes providing the structural outline of their accelerated system, they can invoke the SALAM Configurator, shown to the left in Fig. 2.1. This tool will automatically generate a valid gem5 system simulation configuration, a configuration for SALAM-accelerated components, and a set of headers containing the memory map of the developed system for use in creating accelerator IR and host-side drivers. From here, users can update their accelerator code and drivers to use the automatically generated memory devices and run a full-system simulation. This new automation makes it possible to design and debug complex systems.

The most immediately tangible benefit is when a user wants to rapidly prototype and explore different memory hierarchies for accelerators or groups of accelerators. Mistakes in the memory mapping of devices and memory interfaces can introduce numerous difficulties in diagnosing bugs propagating throughout a design. Depending on how a user allocates their system's memory map, fixing a slight misalignment or improperly sized memory-mapped register could require the realignment of dozens or hundreds of other memory-mapped addresses. Previously, this meant propagating changes across dozens of configuration files, accelerator descriptions, and driver headers. In gem5-SALAMv2, this process is automated, requiring the user to change only one value in a single file and saving significant development and debugging time. Additionally, swapping a variable's access between custom multi-ported scratchpads

and a multi-layer cache hierarchy is as simple as changing a few lines in the system structure description as the configurator creates the new devices, connections, and updated driver headers without any additional user input.

3.1.2 Hardware Model Configuration & Generation

Previously, the hardware resource model was a global entity statically compiled as part of the gem5-SALAM binary. This meant that a designer was limited to one technology node for an entire design, requiring modifications to the SALAM source and rebuilding the binary to tweak or swap technology nodes. Now, the hardware resource model used in SALAMv2 is dynamically generated from YAML configuration files that define a hardware profile, as shown in Fig. 3.2. Notably, the configured hardware resource model defines how functional units and instructions are parameterized during a given simulation. These parameters include each resource's power, area, and performance characteristics. Furthermore, where we previously supported a single hardware resource model, a designer can now create customized instructions and define per-accelerator hardware profiles without recompiling or rebuilding the system.

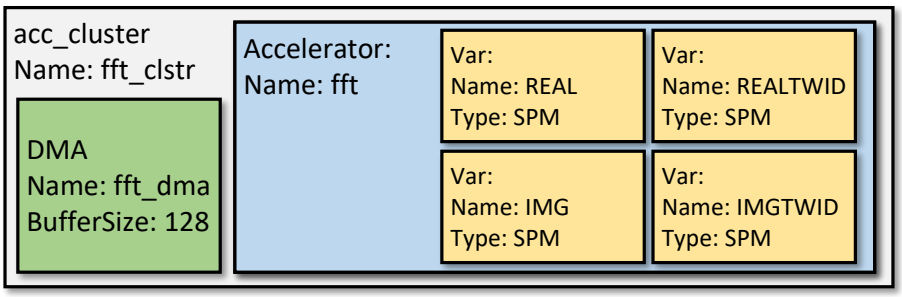


Figure 3.1: *This figure provides a high-level overview of a gem5-SALAMv2 device configuration for an FFT cluster. Devices are described in the hierarchy that a user defines, with Clusters, Accelerators, DMAs and Vars being abstractions for gem5-SALAM SimObjects. In this case, the cluster contains a DMA and an FFT accelerator with four private SPMs.*

3.2 LLVM IR Analysis & Elaboration

In gem5-SALAMv1, the user had to first create functional models of the target hardware accelerators as single inlined functions before using clang to generate the LLVM IR, which was read by the simulator using traditional string parsing techniques. The IR was then statically elaborated internally to generate the CDFG and allocate the hardware resources needed to execute the application within the runtime simulator. gem5-SALAMv1âs handling of LLVM IR was incredibly simplistic and had no context of Value, Constant, Instruction, or any of the dozens of other hierarchical structures that comprise LLVM IR. Instead, gem5-SALAMv1 simply parsed a text file and stored a comparatively flat hierarchy of basic blocks and instructions that were simply linked by the associations of strings. This led to numerous challenges when scheduling large blocks of IR, where dependency look-up was based on searching for specific string patterns. As a result, we needed to impose caps on the scheduling window size internally. We could also not handle LLVM constructs such as llvm::func op because LLVM IR's naming convention for values is not unique across function bounds (e.g., the value name %2 can represent multiple different values in multiple different functions).

In gem5-SALAMv2, we preserve the IR structure as much as possible by leveraging the LLVM APIs directly and recreating many of the core structures within the IR. Previously, the parse was a single pass of the IR file with string parsing methods; now, SALAMv2 has a 3-stage approach. *The first stage* leverages the LLVM libraries to parse the text IR file into the llvm::Module data structure. We re-create the llvm::Module because the IR's top-level structure comprises llvm::Value elements that holistically represent the datapath. *The second stage* of the gem5-SALAMv2 parse iterates over the LLVM module and maps llvm::Value elements to SALAM::Value elements. *The third stage* then initializes all of the SALAM values with their corresponding LLVM values. Structurally, SALAM values match their LLVM counterparts. A

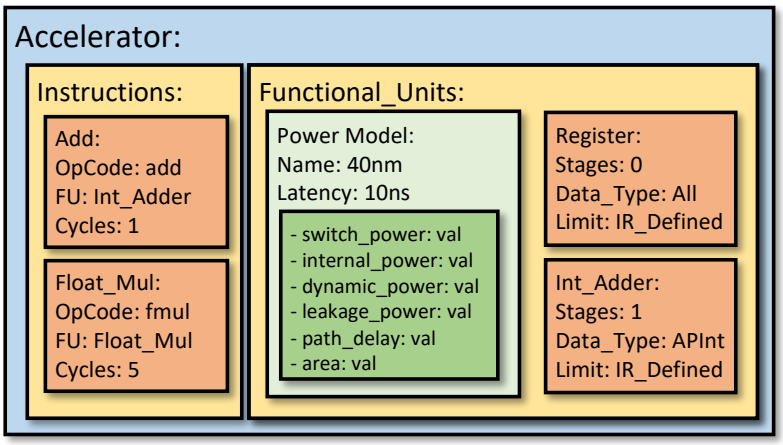


Figure 3.2: *Describes a generic accelerator's hardware configuration within SALAMv2. This shows how one can define individual instruction parameters, functional unit parameters, and power models at an accelerator granularity.*

SALAM::AddInstruction has a similar structure to llvm::AddInstruction, with proper connections to its corresponding basic block, parent function, and other instructions and constants. This more hierarchical structure makes look-ups of dependencies much quicker when adding new instructions to the scheduler. It also enables us to track dependencies across function calls.

An additional scheduling benefit comes from the capacity to recognize constants in the IR. Constants and function arguments (considered constants when a function is launched) do not need to be looked up as dependencies. While a constant integer like *i32 1* may be simple to identify with a basic string parse, complex expressions like *double* inttoptr (i32 268566720 to double*)* are less so.

Additionally, the previous pattern-based lookup of dependencies did not provide a sufficient means of differentiating between instructions, constant expressions, and constants during scheduling. This meant scheduling look-ups included searches for constants that never appeared in the scheduler, leading to unnecessary searches over potentially thousands of scheduling nodes. By mirroring LLVMâs IR structure, gem5-SALAMv2 can better leverage IR-level insights and an understanding of the static execution graph to enable a more robust scheduling algorithm that is ultimately faster.

Overall, the improved IR generation and parsing methodology in SALAMv2 eliminates SALAMv1's limitation of only simulating a single in-lined function. This enhanced approach allows SALAMv2 to model each function as an independent accelerator, allowing for fine-tuning applications with configurable resources at the function level while providing the same power, area, and performance metrics as SALAMv1. Furthermore, these improvements increase SALAM's accuracy by providing evaluation metrics at the granularity of individual functions while preserving the benefits from Clang's optimization passes, such as loop unrolling/vectorization and the removal of internal memory allocation.

The key advantages of gem5-SALAMv2's new elaboration approach versus gem5-SALAMv1 can be summarized as:

1. *Robust parse and elaboration that isn't tied to a particular LLVM IR version.*
2. *Improved dependency tracking within the IR capable of crossing function bounds.*
3. *Expanded data typing support with support for custom data types.*
4. *Improved scheduling performance achieved by leveraging knowledge of the LLVM IR structure.*

3.2.1 Static Elaboration

The in-memory representation of the LLVM IR is leveraged for low-level optimizations and application analysis before it is mapped to the corresponding SALAMv2 static application graph. The corresponding SALAMv2 representation of the application graph closely mirrors the structure of the LLVM IR; however, it is optimized for memory footprint, augmented for dynamic dependency tracking, and is tied into the SALAMv2 hardware profile.

It is worth noting that SALAMv2 allocates hardware within a datapath at the granularity of individual instructions and registers. Whereas MosaicSim generalizes

the execution of the LLVM IR graph to a general-purpose execution unit, SALAMv2 will allocate a unique functional unit for each operation. A 16-bit add instruction in the IR corresponds to a 16-bit adder, while a 32-bit multiplication corresponds to a 32-bit multiplier. While SALAMv2 does need to infer some common structural components like multiplexers or counters from the IR, it does not allocate more general compute elements like ALUs.

The statically elaborated application graph generated by SALAMv2 provides a structural framework for executing the accelerated application independent of runtime characteristics. That framework is then leveraged to identify dependencies between functions and instructions at runtime to generate the dynamic application graph executed by the runtime engine. This joint static-dynamic graph approach enables the modeling of more complex hardware accelerators in which runtime control is governed by input data to the accelerator. Whereas trace-based simulators like Aladdin [10] or MosaicSim [12] must generate multiple runtime traces to capture input data-dependent execution behaviors, *SALAMv2 generates its execution graph at runtime* based on the execution pattern governed by the input data. Furthermore, by leveraging the static application graph to construct the dynamic execution graph, SALAMv2 can consistently model the same datapath as application inputs change, enabling users to impose additional constraints on the simulated datapath. In contrast, the Aladdin simulator dynamically alters its modeled datapath based on input data as shown in the SALAMv1 work [13], while MosaicSim bypasses datapath modeling entirely by treating accelerators like out-of-order CPU cores.

3.3 LLVM IR Parametrization

The changes to how IR is modeled inside of SALAM required changing the simulator's IR parameterization for elaborated designs. Coupled with the new hardware configuration system, this allows gem5-SALAMv2 to utilize parameters from the hardware and device configuration files to model unique hardware elements for

each operation in the parsed LLVM IR. The power estimation model tracks static and dynamic power for functional units and internal data-path logic while accounting for leakage power, switching power, and internal power dissipation. This is further described in Sec. 3.3.1. gem5-SALAMv2 also provides post-simulation performance metrics, allowing users to define specific latencies and configure clock speeds within the accelerator and to set the maximum quantity of each functional unit. This enables fine-grained analysis of system occupancy levels, with data used to determine leakage power, area, and average occupancy for each type of functional unit, allowing for updates to the dynamic energy usage of the system.

3.3.1 Power and Area Estimation

Like in SALAMv1, the power estimation model in SALAMv2 is based on McPat's Cacti [31]. To enable a more flexible configuration of these parameters, SALAMv2 introduces a hardware profile and config shown in Fig. 3.2. The *hardware profile* contains power and area profiles for fixed and floating-point hardware functional units and variable-length registers. While the *hardware config* limits the number of hardware functional units in the system. By default, SALAM will model unique hardware elements for each operation from the parsed LLVM IR. To explore features such as functional unit reuse, a user can further constrain the allocation of hardware elements by modifying the device config.

When estimating the *static power* of a given accelerator, the static CDFG of the elaborated datapath is used. This accounts for all functional units within the system, simulation runtime, and hardware profile. These elements are all used to determine the total leakage power lost in the system due to these elements. The *dynamic power* used by the functional units is calculated for each cycle for each active functional unit and is the combination of switching and internal power dissipation as defined in the hardware profile.

Additionally, the new IR modeling capabilities introduced in gem5-SALAMv2 ex-

poses the internal registers and their bit size to the runtime engine. This allows for tracking of read and write activities on each cycle and enables gem5-SALAMv2 to model the runtime energy usage of internal data-path logic. This is accomplished using the same method described for functional units, where static and dynamic power and area are calculated based on the single-bit register results obtained for the hardware profile.

3.3.2 Performance and Occupancy Analysis

SALAMv2 also provides a variety of performance metrics to the user post-simulation. Within the device configuration, gem5-SALAM defines the cycle time each LLVM IR instruction takes to execute in the compute queues, where the default values were tuned and validated vs HLS performance below in Sec. 4. The user can define the latency of hardware devices and the clock speed within the accelerator. These knobs enable users to accurately model and explore their effects on accelerator models' cycle counts, runtime, and functional unit occupancy.

One knob available to the user is the ability to directly set the maximum quantity of each functional unit or allow the simulation to determine the maximum potential parallelism by dynamically activating portions of the datapath based on the user-defined width of the runtime scheduler. In either case, during the dynamic runtime simulation gem5-SALAM logs, instructions are scheduled or in flight for each cycle while the functional unit controller tracks and stores scheduled functional units each cycle during runtime. This information is passed to the power profiler to update the system's current dynamic energy usage.

These additional data points, combined with configurable hardware resources, allow for a fine-grained analysis and exploration tool for exploring occupancy levels within the system. During post-simulation, this data is used to determine the leakage power and area and the average occupancy for each type of functional unit. The scheduled amount is also available to the user as a printable result and can be used to view

the runtime scheduling activities of the datapath and functional units utilization graphically. Some examples of the flexibility in design and the available analytics from gem5-SALAMv2 are explored more in Sec. 4.

3.4 LLVM Runtime Engine

To capture the dynamic characteristics of the execution of an accelerator, gem5-SALAMv2 dynamically assembles a runtime execution graph that leverages blocks of the static CDFG and a series of queues that track the progress and flow of data through that graph. Execution begins by loading the entry basic block of the top-level function.

3.4.1 Event Scheduling and Dependency Tracking

SALAMv2 internally tracks the execution status of an accelerator pipeline through a set of custom event queues that correspond to reservation events, active compute events, and active read/write events. Event scheduling within the scope of a function occurs at the basic block granularity. When scheduling begins or a branch is evaluated, the next basic block of instructions is loaded from the static CDFG.

As instructions are added to the reservation queue, a query is performed to check for any data dependencies that might already exist in the reservation, compute, or memory queues. If a dependency is found, a connection is created between the runtime instructions. The dependency is cleared once the instruction that creates the dependency is committed. To optimize search times, only the last instance of a dependency in the queues is tracked. Dependency tracking represents the most significant overhead in SALAM's simulation model, with the event queues sometimes tracking dependencies across thousands of events.

Another design consideration during the creation of gem5-SALAMv2 was the reduction of scheduling and dependency tracking overheads. In gem5-SALAMv2, when an instruction satisfies all its runtime dependencies, it will be executed on the next

available scheduling cycle. This scheduling paradigm enables us to represent a parallelized pipeline in which independent instructions can run concurrently. Because of this parallelism, a few synchronization mechanisms are introduced to the scheduler and described in Sec. 3.4.3. When resource limits are imposed, an execution event will only launch when the necessary hardware resource is available. This has led to a performance gain of more than 2x on large applications.

3.4.2 Compute Events

When all dependencies, scheduling, and hardware limitations are cleared and accounted for, a scheduled operation is removed from the reservation event queue and added to its corresponding execution event queue. Control and data flow instructions such as phi, select, and terminators are executed in place without adding any additional queues. For terminators like branches, execution means loading the next basic block of instructions to the event scheduler and dependency tracker. Instructions with a computational component, such as arithmetic or logical operators, are transferred to the computation event queue. Instructions with compute events are polled for completion at the start of each accelerator clock cycle. Upon completion, they signal all dependent instructions to clear the runtime dependency and are removed from the compute queue.

Memory operations are sorted into appropriate read and write queues and passed to the communications interface to access the gem5 memory system. These events are committed as soon as the corresponding gem5 memory events are completed. Like compute events, committing a memory event causes the instruction to signal its dependents before it is removed from the memory queue.

3.4.3 Event Synchronization

With the parallelization of the instruction pipeline in SALAMv2, the scheduler needs to prevent out-of-order execution and commits to an otherwise serialized IR.

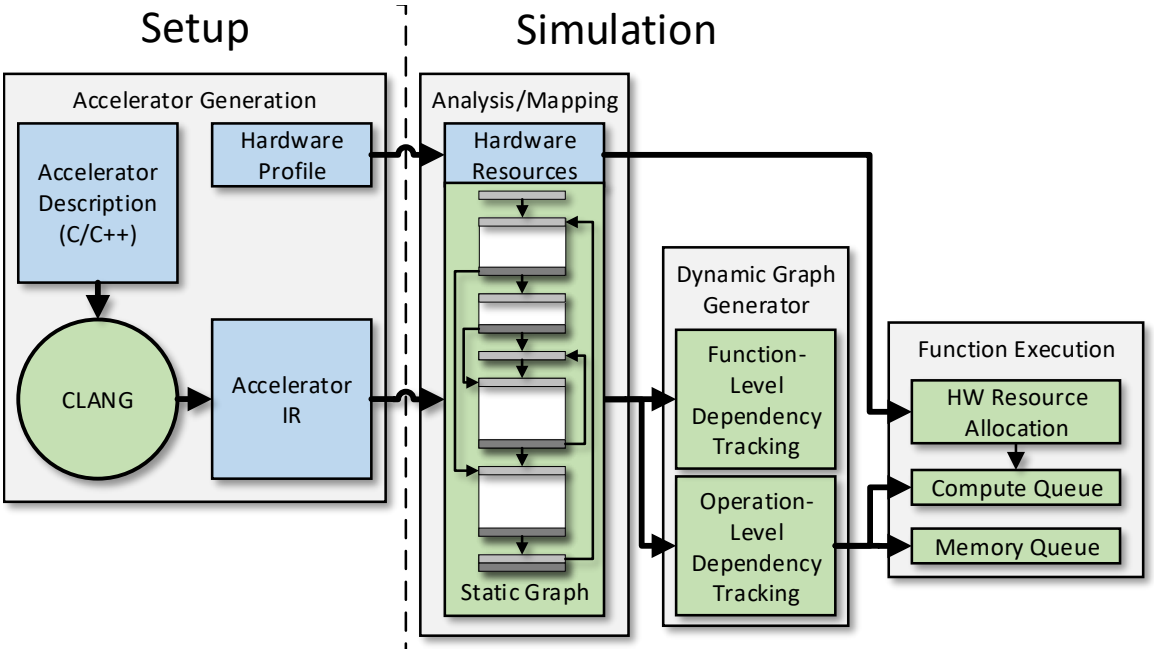


Figure 3.3: *An Overview of the LLVM runtime model present in gem5-SALAMv2. The automated configuration tools invoked during setup provide the constructs needed for elaboration to generate the static graph. This is then dynamically mapped to the allocated resources to perform a cycle-accurate simulation of the application.*

To properly handle this hazard, SALAMv2 introduces a few synchronization mechanisms. First, SALAMv2 automatically detects loop boundaries and imposes barriers at the ends of loop execution. Second, SALAMv2's scheduler provides tracking to ensure the correct order of reads and writes to memory. Third, SALAMv2 provides an optional lock-step execution mode that prevents new execution events from launching until all previous events have been completed, regardless of runtime dependencies. From a configuration standpoint, users have multiple options for configuring the LLVMInterface:

- Scheduler Threshold: A hard limit on the scheduling window size to prevent scheduling window size from exploding during regions of high loop parallelism
- Lockstep Mode: Enables or disables stalling when any op stalls (enabled by default)

- Clock Period: The desired clock speed for the datapath

In addition to these explicit configuration options through the gem5 build config, a user can explore functional unit reuse through the hardware configuration by imposing hardware resource limitations and restrictions.

3.4.4 Function Call Semantics and Advanced Scheduling

One of the most notable changes in SALAMv2's scheduler updates revolves around handling function calls. In general, function calls introduce complex challenges for hardware modeling and simulation. For elaboration purposes, hardware design and simulation tools must provide a policy for imposing hardware resource limitations and tracking connections and dependencies across function bounds.

Other simulators bypass this issue entirely by either allocating resources based on a pre-generated execution trace [9], modeling a general-purpose architecture with fixed resources [12], or forcing users to manually inline functions [13]. To solve this issue in gem5-SALAMv2, we introduce a model hierarchy within each hardware accelerator. Function calls are handled like micro-pipelined functional units within the datapath of the calling function, enabling interesting capabilities within gem5-SALAMv2.

First, this design enables users to define custom operations within the LLVMRuntime without directly implementing handling for new IR instructions. These operations can be designed with fixed or variable timings based on the nature of data access or computation within the function. This is particularly helpful when an operation involving system-level overheads, such as memory access, cannot be statically determined or modeled correctly by execution traces.

Second, SALAMv2's function handling allows finer-grained control over datapath structure and parallelism. gem5-SALAMv2 internally tries to find the highest degree of spatial and temporal parallelism in the scope of a function with its out-of-order execution and commit paradigm. By leveraging functions, users can fine-tune the degrees of parallelism in the simulated datapaths to better represent the concurrency

of hardware models and implement more complex parallelism semantics like barriers. This functionality does not currently exist outside of RTL simulation or other handcrafted simulation models.

3.5 gem5 Integration and Scalable Full System Simulation

gem5-SALAMv2 expands on the system integration framework first presented in gem5-SALAMv1. Like in v1, integrating the LLVM runtime engine into gem5 revolves around the "Communications Interface" or CommInterface. The CommInterface is a gem5 simulation construct that provides system timing and interfaces for configuration, synchronization, and memory access. These include access to the system clock, interrupt control lines, memory-mapped configuration registers, and memory request ports. The basic integrations provided in gem5-SALAMv1 enabled explorations of accelerator integrations ranging from discrete off-chip accelerators to co-processing elements integrated directly into the datapaths of other devices. In gem5-SALAMv2, we expanded on the functionality of the CommInterface to enable more complex accelerator hierarchies. The first of these changes was the rework of the structure of memory request ports. In gem5-SALAMv1, the CommInterface had two types of access ports. These were specified as access ports for local accelerator resources and global system resources. Additionally, the v1 CommInterfaces had an additional internal mechanism for accessing scratchpad memories that bypassed gem5's standard memory system to enable wider multi-port access. In gem5-SALAMv2, this has been reworked to a more flexible interface, as shown in Fig. 3.4. In the updated model, the CommInterface has a more flexible interface to memory that has been grouped into four categories for convenience.

The stream port interface enables the connection of streaming devices with an integrated handshake mechanism comparable to the AXI-stream specification. While this feature was introduced towards the end of gem5-SALAMv1's development, it has been expanded and improved upon in v2. Whereas in v1 streaming access was

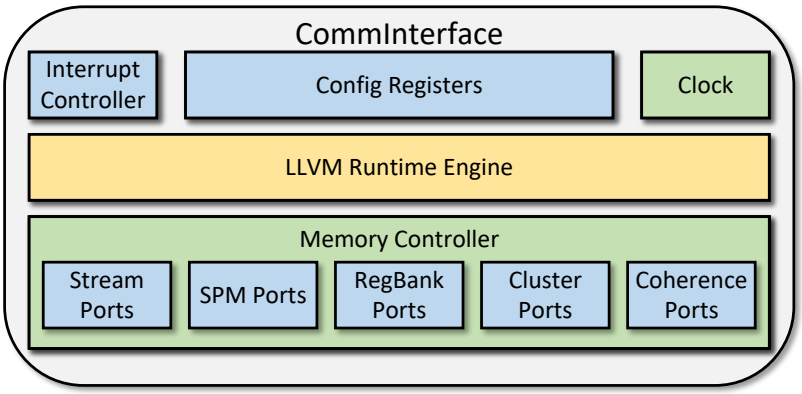


Figure 3.4: *Provides an overview of the communications interface in gem5-SALAMv2. This new unified interface handles all communications between gem5-SALAMv2 clusters, accelerators, memory objects, and the gem5 ecosystem.*

implemented solely as a blocking access behavior, accelerators in v2 are now able to query the availability of data in streams before initiating an access request. This enables the modeling of devices in which runtime control can be altered based on the availability of data, and enable implementation of access arbitration schemes on shared resources.

The SPM port interface enables flexible connections of scratchpad memory devices that include the data status functionality first described in gem5-SALAMv1. Accelerators connected to scratchpad memories via these ports can poll the connected scratchpads on the availability of data to access data elements as soon as they are available. In gem5-SALAMv1 scratchpad memories were directly integrated into the accelerator model, which imposed additional design challenges when constructing shared scratchpad memories across accelerators. In gem5-SALAMv2 these integrated scratchpads have been properly decoupled from the accelerator models, and the CommInterface has been upgraded to support numerous connected scratchpad memories while retaining the wide, multi-ported access present in v1.

The RegBank port interface of SALAMv2's CommInterface enables a new type of memory device in the form of a register bank. Register banks provide users with the capacity to model data storage in registers that aren't explicitly allocated by

SALAM's LLVM parser (usually arrays). Register banks offer the same memory access and delta timing characteristics as registers explicitly elaborated in the datapath and are designed to be private to the accelerator they are connected to.

The cluster port and coherence port interfaces are designed to provide flexible interconnects to other system resources. They are separated to provide priority access to devices through the cluster ports interface with the coherence interface as a fallback. The most common usage of these interfaces is to separate accesses to shared accelerator resources vs system-level resources that may require coherence with the CPU and its caches. This separation is useful when looking to model a system that separates accelerators into tiles with mesh connections between tiles. In this case, the communications within the tile would pass through the cluster interface, while communications across tiles pass through the coherence interface.

A more general description of these SALAM Modules, such as the CommInterface or an SPM, is that they are a gem5 SimObject that utilizes the gem5 memory system and does not implement the Ruby memory system. From a system simulation perspective, the overheads for these SimObjects will be relatively low as they are just responsible for memory transactions and are decoupled from the datapath simulation that the LLVM Interface is responsible for. LLVM Interfaceâs modeling overheads will depend highly on the application because we elaborate the datapath at runtime.

In addition to the interface updates on the CommInterface, we have also expanded functionality in the control and operation of SALAM accelerator models. One of the benefits of the dynamic execution of LLVM graphs employed by gem5-SALAM is that we can model execution patterns that are dependent on the status and availability of other devices in a system. Whereas in other simulators that rely on memory[12] or execution traces[10], in which these events must somehow be baked into the trace, gem5-SALAM can directly handle such variability in execution and control. Coupled with the improved status tracking on SALAMv2's stream modeling, users can design

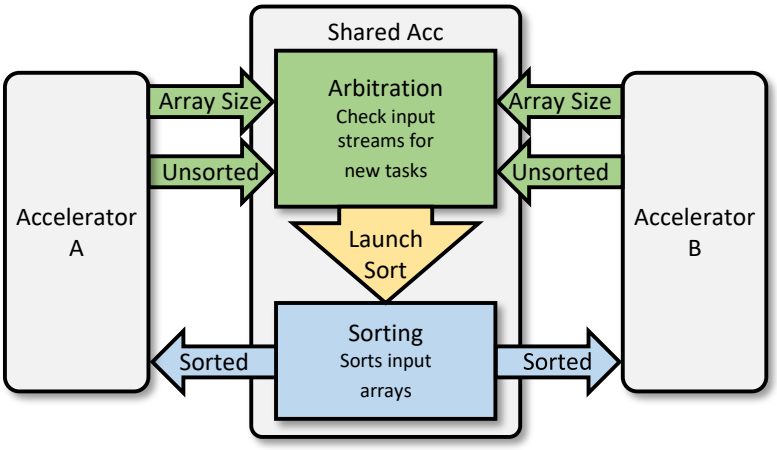


Figure 3.5: *A shared accelerator resource scenario demonstrating gem5-SALAMv2's ability to have data-driven accelerators.*

accelerators that operate in a passive state until a change is observed in some other device (such as another accelerator) or are provided with data on an input stream. To demonstrate this we present the test case shown in Fig. 3.5

In the example shown in Fig. 3.5, we construct a system consisting of three accelerators. Two of the accelerators represent an arbitrary workload that, in this instance, produces unsorted arrays of arbitrary length. The third accelerator represents a shared resource that sorts the output of the other two accelerators and returns the sorted arrays. The shared acceleration consists of two functions: Arbitration and Sorting. The arbitration acc polls the status of input streams (shown in green) for available data from the two input accelerators. When data is detected from one of the input accelerators, the arbitration acc launches the sorting task to read the appropriate unsorted data stream and respond with the sorted data.

In gem5-SALAMv1 and other existing simulators, this system would require the implementation of a scheme by which the two data-producing accelerators could negotiate the usage of the shared resources similarly to what is used in shared memory programming models. In gem5-SALAMv2, we can model a system in which data producers can pass their outputs to the shared sorting accelerator via streams, allow-

ing the sorting accelerators to launch sorting tasks as data becomes available. This enables the modeling of a system in which arbitration can be handled directly in the hardware of a shared resource. While this functionality may appear trivial, the capacity for simulated accelerators to model runtime-dependent control behaviors based on the *availability* of input data is not available in any other pre-RTL simulations, gem5-SALAMv1 included.

These ideas can be further expanded upon to create other, more generalized hardware devices. Through clever combinations of gem5-SALAMv2's LLVM runtime, new system resources, and gem5's standard system resources, gem5-SALAMv2 offers the opportunity to rapidly implement new hardware constructs without the hassle of writing custom simulator models. In addition to the time savings vs. developing and testing new hardware simulation models, gem5-SALAMv2 offers the direct integration of power and area estimation with high degrees of user control and customization. This allows users to explore the modeling of system-level hardware concepts without many of the traditional development overheads imposed by the APIs of full system simulators like gem5.

CHAPTER 4: SIMULATOR VALIDATION & COMPARISON

In the following sections, we present our results and analysis in detail to demonstrate the benefits of gem5-SALAM. To show this, we validate our pre-RTL timing model, performance, power, area, and system timing metrics and give results that show the expanded capabilities of gem5-SALAMv2 in the domain of multiple accelerator design space exploration.

4.1 Timing, Power, and Area Validation

gem5-SALAMv1 was developed around the LLVM 3.8 compiler toolchain and a simpler internal elaboration and scheduling system. Given gem5-SALAMv2's significant overhauls to elaboration and runtime scheduling, as well as the substantial changes in LLVM IR generation and structure that have arisen between LLVM 3.8 and LLVM 9.x, we have re-validated on all of the Machsuite [1] benchmarks used in gem5-SALAMv1.

For the timing model, we validate against all of the accurate MachSuite benchmarks against RTL models generated by Vivado HLS. We choose MachSuite for validating the timing model for two primary reasons: 1.) We want to compare datapath verification against both SALAMv1 and, implicitly, gem5-aladdin. 2.) MachSuite provides us with a robust set of benchmarks to validate different components of the IR modeling, including compute, control, and dataflow aspects across these benchmarks.

The power and area models were validated against Synopsys Design Compiler elaborations, using an open-source 40nm standard cell library and the gate switching activity produced by RTL simulation in Vivado. we have validated with the 40nm library for functional validation and for maintaining continuity of comparisons between

SALAMv1 and other simulation frameworks like gem5-aladdin.

We have only validated designs small enough that we would not expect device-specific overheads for things like routing delays. Of course, we cannot estimate such things because these estimates would require a gate-level understanding of these designs vs. a particular target system and gem5-SALAMv2 is explicitly a pre-RTL simulation. Additionally, gem5-SALAMv2's modeling is more abstract than even SystemC which is, at best, an estimate in terms of timing vs. a real board. For a comparable set of IPs with comparable resources, you would expect comparable timing results across boards.

This validated hardware profile is the default configuration in gem5-SALAMv2, although the user can easily modify or extend this profile to explore custom hardware. The minor differences in the results of the evaluation metrics are due to the changes to LLVM's IR from 3.8 to 9.x.

Table 4.1 provides absolute timing, power, and area validation metrics for all the accurate and synthesizable benchmarks obtained from MachSuite. Using the last version of gem5-SALAMv1, we found the relative error comparisons between the previous iteration, SALAMv2, and HLS elaboration in Figure 4.2. For each test case, we used the same clang optimizations for generating the LLVM IR to ensure the same levels of Instruction Level Parallelism (ILP) as the datapaths generated by HLS. From these results, we can demonstrate the same level of timing accuracy in gem5-SALAMv2 as in gem5-SALAMv1 and HLS.

Figure 4.1 also shows the power area validation across the same set of benchmarks. Stencil3D was excluded from this set due to Design Compiler running out of memory during elaboration. The average error in power estimation is slightly lower than in gem5-SALAMv1, at 2.45% vs 3.27%. Like in gem5-SALAMv1, power estimations in gem5-SALAMv2 trend toward a slight overestimation due to the variability in the power consumption of muxes and non-arithmetic operators. The MD-KNN bench-

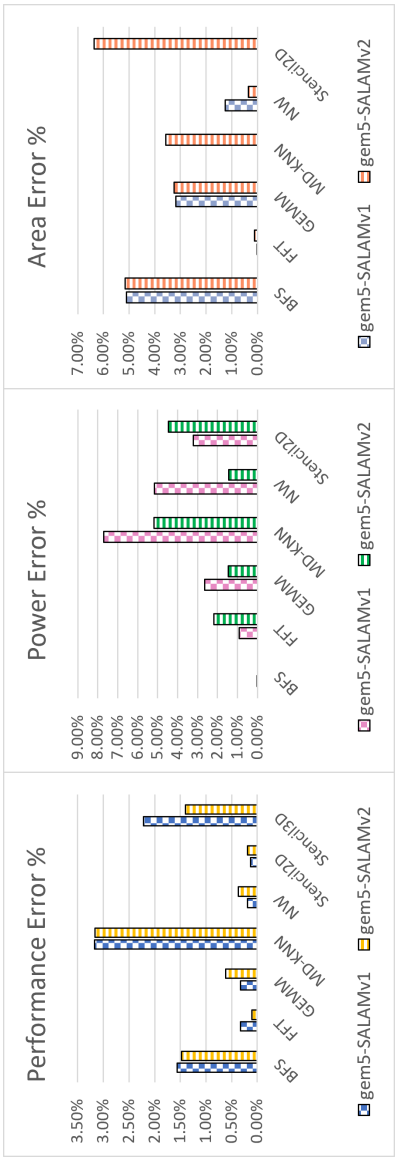


Figure 4.1: gem5-SALAM versus Vivado HLS power, area, and performance error percentages.

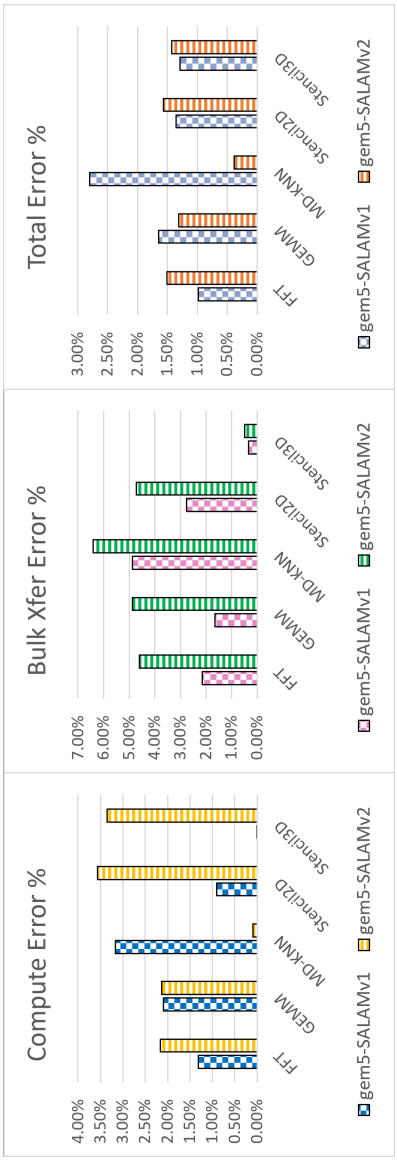


Figure 4.2: gem5-SALAM's relative error when compared to ground-truth FPGA timing

mark showcases this by having the highest power error due to heavier reliance on these operators. For area validation, gem5-SALAMv2 can estimate chip area with an error of 2.24% on average.

While error rates across timing, power, and area are comparable between gem5-SALAMv1 and gem5-SALAMv2, some conflating factors result in discrepancies between their estimates. For one, LLVM IR generation has seen significant changes in structure and optimization between version 3.8 and the LLVM 9.x build used for validation. While both sets of validations employed O1 optimizations and targeted unrolling during IR generation, the resulting IR used for elaboration and simulation shows notable differences in code structure. This, coupled with SALAMv2's more conservative scheduling to address memory errors in gem5-SALAMv1, results in slightly higher timing estimates on average versus gem5-SALAMv1.

4.2 FPGA System Validation

For system validation, we synthesized five benchmarks and executed them on a Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board with an XCZU9EG SoC chip, and the ARM processors clocked at 1.2GHz. We used Vivado HLS 2018.3 to synthesize the benchmarks and Vivado SDSoC 2018.3 to cross-compile the host programs, which invoke the kernel synthesized by Vivado. The targeted benchmarks are summarized in table 4.2. The reported bulk transfer time is the summation of both read/write time from/to shared DDR memory. To match the configuration of the FPGA programmable logic, an accelerator cluster was instantiated within gem5-SALAM consisting of a DMA, an accelerator for the top-level function, and an accelerator for the benchmark kernel. The top accelerator was programmed by the host CPU and used to schedule memory transfers and invoke the benchmark accelerator. The burst width of the cluster DMA was tuned to match the burst width of the data mover.

Figure 4.2 displays a similar trend to the RTL simulation results. Positive error

indicates when the simulation was faster, while negative error indicates faster FPGA times. One notable difference in the timing between v1 and v2 is the increase in transfer times due to gem5-SALAMv2 utilizing a top-level accelerator independent from the application to control data movement. While this slightly increases transfer time overhead, it provides far more flexibility in system design with a negligible effect on total execution time.

The most significant discrepancies in computation error can be attributed to benchmarks operating on double-precision floating-point, with the more accurate benchmarks operating on integer types. By default, gem5-SALAMv2 approximates floating-point operations using 3-stage FP adders and multipliers, which do not precisely match the floating-point DSP IPs employed by SDSoC. Even so, the timing is close enough to maintain a high degree of fidelity with the FPGA implementation, with variances primarily due to a difference in cache invalidation times between the ZCU102 and the simulation.

4.3 Simulation Timing Comparison

SALAMv2's runtime engine upgrade was designed to improve simulation performance over SALAMv1. This upgrade required numerous modifications to SALAM's in-memory representation of CDFGs and the processes for tracking runtime dependencies across multiple execution graphs and scheduling runtime events further described in Sec. 3.4. Table 4.3 compares setup and simulation times for gem5-SALAMv1 and gem5-SALAMv2 when run for 9 Machsuite benchmarks on a system with a Ryzen 3900x and 32GB of RAM.

To compare the two versions, we examined the timing performance of the datapath parse/setup and simulation time for accelerators in SALAMv1 vs SALAMv2. In gem5-SALAMv1, the datapath parse and setup were highly variable based on the complexity of the IR (number of operations, the complexity of data types, etc.). This could lead to an order of magnitude difference in parse times between applications like

GEMM, with its high degrees of unrolling, and a smaller benchmark like BFS. These setup times are far more normalized by leveraging the LLVM libraries for IR parsing in SALAMv2. However, the base cost of SALAMv2's approach leads to slower parse times in SALAMv2 for smaller applications. However, SALAMv2 is far more efficient at parsing larger IR files that are larger and contain more complex data structures, as shown with GEMM and Stencil3D in Table 4.3.

This means that SALAMv2 will run slower for very small accelerators. This is most apparent in the SPMV application, which cannot be statically loop unrolled due to runtime dependencies. The resulting static CDFG constrains runtime parallelism and also constrains SALAM's event scheduling windows. In contrast, the GEMM and Stencil3D benchmarks contain large amounts of loop unrolling, meaning there are far more dependencies to track simultaneously in SALAM's event scheduling windows. SPMV's largest basic block for dependency tracking contains around 10 operations, whereas GEMM and Stencil3D have blocks with more than 500 operations. Here, the updates to gem5-SALAMv2 enable us to drastically reduce dependency lookup times and compute event scheduling times. The result is that while gem5-SALAMv2 may lose milliseconds in the execution of very small accelerators, it can improve simulation times by significant factors in more complex designs.

In Sec. 5, we explore systems built around neural network architectures. In the case of the MobileNetv2 design described in Sec. 5, gem5-SALAMv2 sees a more than 3x speedup over gem5-SALAMv1, which reduces simulation times for full-network runs by several hours.

Table 4.1: gem5-SALAM & Vivado HLS performance, power, and area data.

Bench	Performance (cycles)			Power (μW2)			Area (mm^2)		
	HLS	v1	v2	HLS	v1	v2	HLS	v1	v2
BFS	15834	15587	15600	1.3497	1.3496	1.3497	8114	7700	7696
FFT	91168	90874	91265	58.0836	57.5524	59.3513	39185	39179	39228
GEMM	131098	131524	131900	64.4219	62.7213	65.3655	299086	289597	289400
MD-KNN	317969	328050	328025	15.1745	16.3431	15.9594	41319	41319	42791
NW	66712	66587	66962	6.1093	6.424	6.1975	12195	12042	12152
Stencil2D	109358	109500	109563	41.5812	42.9137	43.4334	8461	8461	9000
Stencil3D	46559	45526	47210	-	-	-	-	-	-

Table 4.2: gem5-SALAM & ground-truth FPGA timing data from the five MachSuite benchmarks validated against.

Bench	FPGA			SALAMv1			SALAMv2		
	Compute	Xfer	Total	Compute	Xfer	Total	Compute	Xfer	Total
FFT	879.3 μs	93.6 μs	972.9 μs	867.8 μs	95.6 μs	963.4 μs	860.4 μs	97.9 μs	958.2 μs
GEMM	1343.3 μs	179.0 μs	1522.3 μs	1315.2 μs	182.0 μs	1497.2 μs	1314.6 μs	187.7 μs	1502.4 μs
MD-KNN	2489.6 μs	118.7 μs	2608.4 μs	2568.5 μs	113.0 μs	2681.4 μs	2487.2 μs	111.1 μs	2598.4 μs
Stencil2D	846.4 μs	268.5 μs	1115.0 μs	854.1 μs	276.0 μs	1130.1 μs	876.6 μs	255.9 μs	1132.5 μs
Stencil3D	445.2 μs	444.5 μs	889.8 μs	455.2 μs	446.0 μs	901.3 μs	460.2 μs	442.3 μs	902.5 μs

Table 4.3: gem5-SALAM simulation timing values and comparison.

Bench	SALAMv1			SALAMv2			SALAMv2 Speedup		
	Setup	Sim.	Total	Setup	Sim.	Total	Setup	Sim.	Total
BFS	0.239 ms	0.409 s	0.409 s	0.507 ms	0.327 s	0.328 s	0.471x	1.249x	1.248x
FFT	0.261 ms	1.120 s	1.120 s	0.740 ms	1.887 s	1.888 s	0.353x	0.594x	0.594x
GEMM	4.76 ms	21.387 s	21.391 s	0.409 ms	10.028 s	10.028 s	11.646x	2.133x	2.133x
MD-KNN	0.264 ms	5.689 s	5.689 s	0.711 ms	4.467 s	4.468 s	0.372x	1.274x	1.274x
NW	1.53 ms	1.548 s	1.550 s	0.456 ms	1.606 s	1.606 s	3.364x	0.964x	0.965x
SPMV	0.172 ms	0.152 s	0.153 s	0.481 ms	0.487 s	0.487 s	0.357x	0.313x	0.313x
Stencil2D	0.378 ms	2.066 s	2.067 s	0.397 ms	2.918 s	2.918 s	0.951x	0.708x	0.708x
Stencil3D	5.09 ms	4.050 s	4.055 s	0.426 ms	2.003 s	2.003 s	11.950x	2.022x	2.024x

CHAPTER 5: DESIGN SPACE EXPLORATION

To demonstrate the increased flexibility added in gem5-SALAMv2 to design space exploration, we explore different hardware architectures for two unique CNNs. We first showcase the flexibility that one has to tweak design knobs by exploring three different LeNet-5 architectures in Sec. 5.1. This shows how a designer could utilize the improved design automation tools and simulation features within gem5-SALAMv2 to rapidly explore architectural changes. We then showcase the ability of gem5-SALAMv2 to simulate and evaluate a large-scale design with a full implementation of the MobileNetV2 CNN [16] in Sec. 5.2.

5.1 Case Study: LeNet-5

Because LeNet-5 is significantly smaller than modern CNNs, we use this example to demonstrate how a design can be iterated on and improved inside of gem5-SALAMv2 in a way that was previously arduous in gem5-SALAMv1. We explore three hardware designs: Naive, Massively Parallel, and Efficient Streaming. These are meant to show how a designer can rapidly explore architectures in SALAMv2 and are not intended to be presentations of novel architectures. In the following sections, we discuss some of the significant benefits of using gem5-SALAMv2 over other simulators and present and analyze each of the three designs.

5.1.1 System Setup and Configuration

We first start the development of our systems by defining constant system and hardware configurations to be used for our design space exploration. All hardware and power profiles used for testing were based on an open source 40nm standard cell library with a 10ns device and system latency. We used the same functional

unit timings and configuration from our system validation in Sec. 4.2, with lockstep execution and memory hazard prevention enabled.

Each cluster contains a standard top-level accelerator to control communication between the gem5 system and the cluster accelerators. The memory storage techniques used within the accelerator clusters are either scratchpad memories or stream buffers. These are simulation objects within the gem5-SALAMv2 simulator that utilize user-defined hardware profiles for parameterization and analysis. Additionally, we used the bare-metal ARM implementation of gem5, with 4GB of 2400MHz DDR4 RAM and the standard DerivO3CPU CPU type included in gem5.

5.1.2 Application Metrics and Testing

For our design space exploration, we sought to profile how our designs would perform across varying spatial (loop unrolling/vectorization) and temporal (dynamic execution) factors. For metrics, we observe the variations in power, area, and latency that each design exhibits with varying configurations. We use three separate configurations on our three separate topographies to showcase how one can use the gem5-SALAMv2 toolchain to explore new architectural designs.

We identify six internal dimensions for exploration: output height/width, kernel height/width, and input/output channel depth. The kernel and channel depth parameters define the internal loop structure, which can be unrolled to increase datapath parallelism. This enables us to create three IR variants to benchmark on each architecture. The first variant does not contain any loop unrolling, the second includes a fully unrolled kernel vector, and the third fully unrolls the kernel and the channel vectors. We define these as "No Unroll," "Input Unroll," and "Output Unroll" to showcase how one can vary this knob in a given design.

- **No Unroll:** Solely utilizes temporal compute parallelism within each accelerator.

- **Input Unroll:** Fully unrolls the kernel height/width and input channel dimensions of each network layer. For example, Conv1 has a 5x5x6 convolution window. We would fully unroll this input window to a factor of 150. We then match the porting of the feature map and weight scratchpads to the unroll factor.

- **Output Unroll:** Fully unrolls kernel height/width and input/output channel dimensions. This means in Conv1 we unroll across a 16x5x5x6 set of loops for a factor of 2400.

5.1.3 Naive Design

With these three configurations defined, we define our three system topologies. All designs possess a top-level accelerator, which is further called "Top." The Top utilizes varying levels of granularity in its control over DMA and synchronization events between network layers. We first present the *Naive* Design, an implementation to be used as a baseline against different architectural features. This system utilizes direct DMA transfers between scratchpads and runs each layer sequentially, as shown in Fig. 5.1. Our second configuration, named *Massively Parallel*, connects accelerators in a streaming-like fashion via scratchpad memories and a dedicated "Data Sync" accelerator, as shown in Fig. 5.2. The final configuration, *Efficient Streaming*, is comprised of functional units that contain convolution and pooling layers with internally managed line buffers, as shown in Fig. 5.3.

The Naive implementation contains a very straightforward architecture for the given CNN. At a high level, the design is disconnected accelerators with memory accessed from a common DMA. An implicit Top controls all memory transfers to and from these accelerators and maintains accelerator synchronization. Because of how memory is managed, there is no overlap between accelerator execution, as each successive accelerator depends on the entire output feature map of the previous layer.

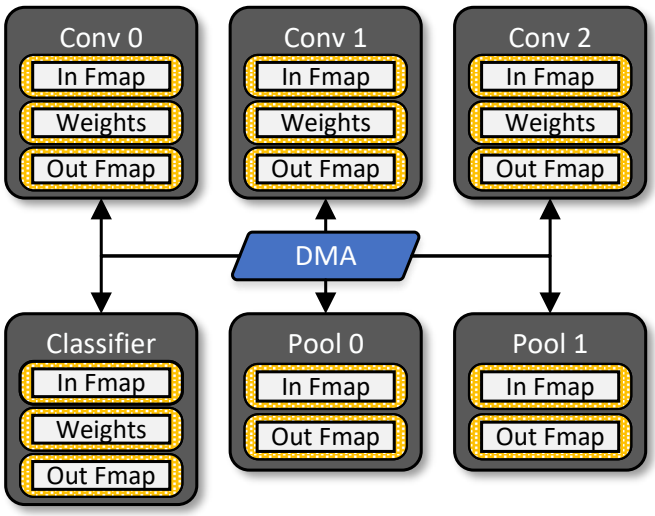


Figure 5.1: Naive Design - *This "Naive" architecture has each accelerator fully controlling its input and output SPMs, with all accelerators connecting to a single DMA for inter-accelerator memory transfers.*

This allows for a straightforward synchronization method where the Top runs each network layer successively and handles corresponding memory transfers.

5.1.4 Massively Parallel Design

One of the significant benefits of design space exploration in gem5-SALAM is the ability to explore compute parallelism across multiple accelerators via communication through the gem5 memory system. To take advantage of this in the Massively Parallel architecture, we connect accelerators in a streaming-like fashion via scratchpad memories and a dedicated "Data Sync" accelerator, as shown in Fig. 5.2. This self-synchronization significantly reduces the control overheads that the Top introduces and allows for overlapping execution.

Streaming in CNNs is challenging due to significant data production and consumption imbalances. Although the data may only be written once, it will be read many times due to shifting convolutional windows. While the Naive design resolves this by allocating separate input and output memories with a DMA to move the data, it lacks an understanding of the data usage and availability in the network that could

be leveraged to improve runtime compute parallelism.

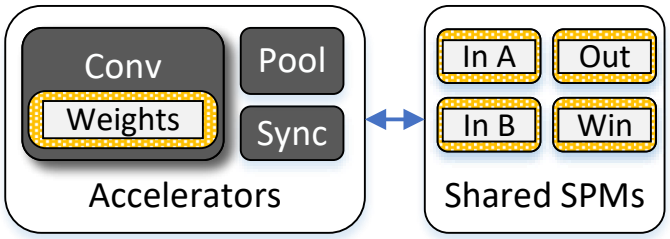


Figure 5.2: Massively Parallel Architecture - *With the increased complexity of the Massively Parallel Design, we present an overview of the devices that comprise the functional unit repeated throughout the design. Notably, the functional unit is comprised of Convolution (Conv), Pooling (Pool), and Data Sync (Sync) accelerators that are directly connected to their relevant SPMs.*

Desiring a more efficient data management solution than possible with a traditional DMA, we leveraged SALAMv2's updated LLVM engine and system interfaces to design a Data Sync accelerator. The data sync accelerator manages the movement of data between the output of one layer and the input of the next in a similar fashion to a DMA with some key differences. Each data sync accelerator is designed with the data access patterns of the input and output layers in mind, allowing data to be transferred between layers as soon as it is ready. This effectively turns the connected input and output scratchpads into a massively parallel stream buffer with the capacity for a single write with multiple reads. We synchronize data access by using the capacity of SALAM accelerators to track the status of memory devices in the system, enabling each accelerator to perform execution based on data availability without reliance on the Top.

5.1.5 Efficient Streaming Design

Building on the features of gem5-SALAMv2, we introduce the Efficient Streaming design depicted in Figure 5.3. This design demonstrates how gem5-SALAMv2's dynamic execution, based on data availability, enables designers to explore complex architectures. In this design, we aim to address the inefficient reuse of allocated

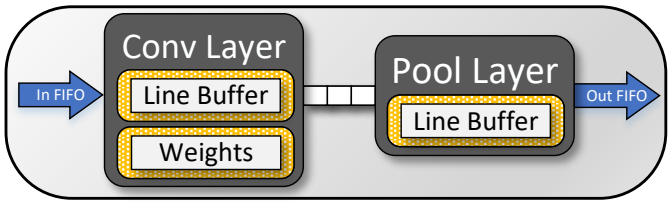


Figure 5.3: Efficient Streaming Functional Unit - *With the integration of data management into the convolution accelerator, there is now only a Convolution and Pooling layer at a functional unit level. The Convolution accelerator stores data from the input FIFO to the Line Buffer SPM to be utilized for the operation; all accelerators are interconnected with streaming FIFOs.*

memory in the Massively Parallel design, while maintaining a higher occupancy level than that of the Naive design. To this end, the Efficient Streaming design substantially increases memory reuse by employing gem5-SALAMv2âs streaming line buffers interconnected between convolution and pooling accelerators. This design further streamlines the overall architecture by relocating data synchronization to the convolution accelerator.

A notable limitation of this architecture is its inability to support unrolling on the output channel due to the inclusion of the line buffer. However, the design exhibits increased levels of memory reuse and diminished SPM sizes across all network layers, resulting in significantly reduced area and energy usage while performing comparably to the Massively Parallel architecture.

5.1.6 LeNet-5 Results and Analysis

With the increased granularity that gem5-SALAMv2 allows for hardware statistics, we can generate Fig. 5.4 and Fig. 5.5. To create these figures, we ran our three separate architectures across our three configurations. Fig. 5.4 confirms the expected trends in computational performance from our design methodology.

For each architecture, implementations with no unrolling are the most energy-efficient but are held back in performance due to the high overheads of sequential execution. Unrolling the kernel input increases the performance and energy usage,

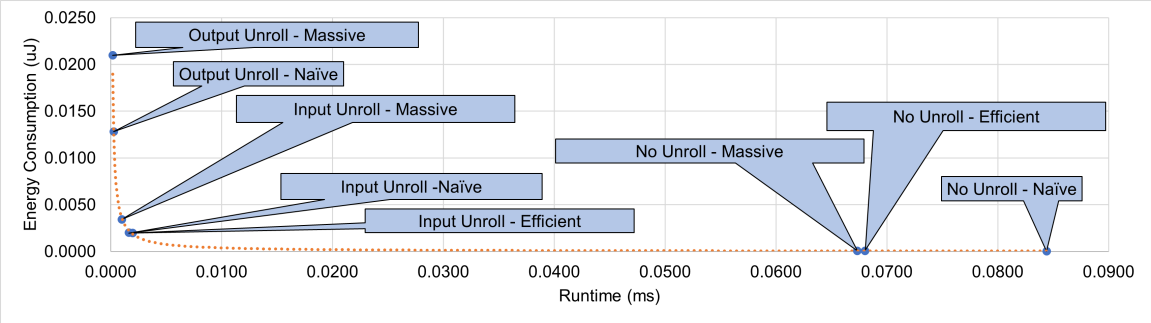
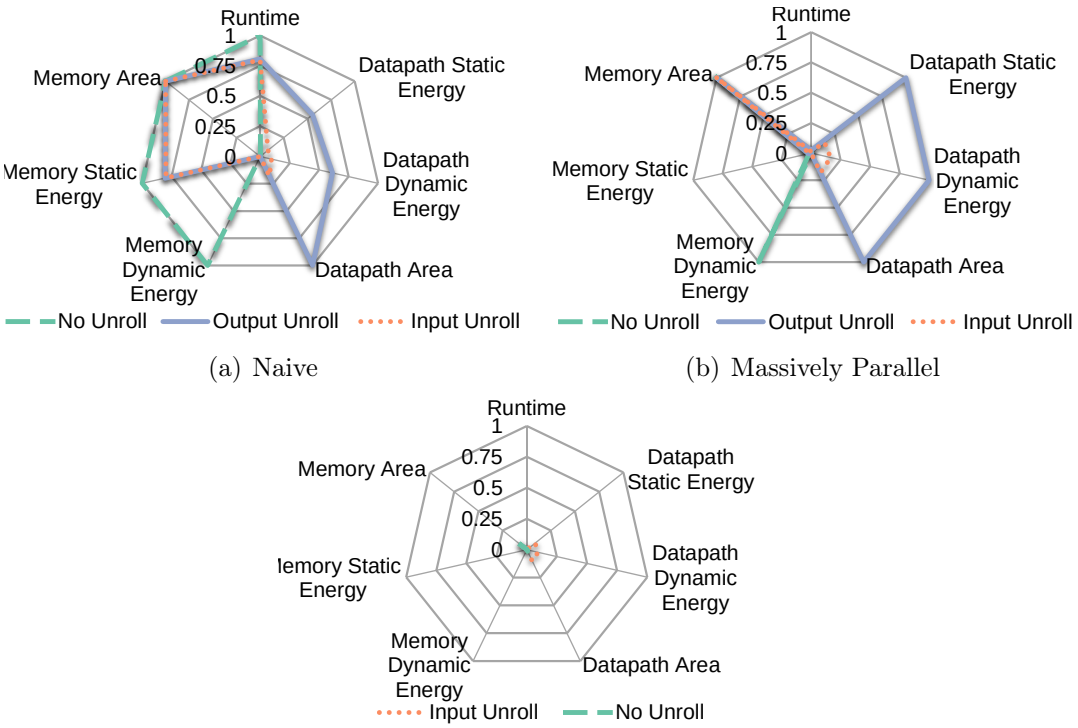


Figure 5.4: Total Data-Path Computational Energy Consumption Vs. Runtime for LeNet-5 Configurations

with diminishing returns on performance and considerable increases in energy usage when unrolling the output channel. While these are not novel insights, these explorations demonstrate the capability of gem5-SALAMv2 to model expected behaviors accurately.

Additionally, looking at Figure 5.4, we see that the internal streaming buffer used in the Efficient Streaming design was the most energy-efficient design for computational performance but performed similarly to both the Naive and Massively Parallel designs. Broadening our scope to utilize metrics that gem5-SALAMv2 can now provide, we consider the results shown in Figure 5.5. These normalized values show that the Efficient Streaming architecture significantly improves energy and area metrics while keeping runtime performance comparable to the previous designs.

Figure 5.5 shows some complex metrics available with gem5-SALAMv2. Our simulations show normalized runtime, area, and static and dynamic energies for memories and datapaths. The Naive design has significant runtime, area, and energy overheads versus the Massively Parallel and Efficient Streaming architectures. This is mainly because of the inefficient use of the local DMA and the lack of accelerator occupancy due to accelerators running serialized. Looking at the Massively Parallel Design, we see that it has significant runtime improvements to the Naive design but has high area and energy requirements compared to the Efficient Streaming design. This is



(a) Naive

(b) Massively Parallel

(c) Efficient Streaming

Figure 5.5: LeNet-5 Power, Area, and Performance Values - *These values were normalized by dividing all results by the max value obtained in any of the three architectures for each category. This technique preserves the ratio of the results between architectures on a scale from 0-to-1.*

due to the design being able to execute all accelerators in parallel. However, because of the large SPMs allocated, it has the most significant area and energy footprint of all three configurations. We also see that the Efficient Streaming design is the most efficient implementation while performing marginally slower than the Massively Parallel design at both degrees of parallelism supported (no and input unroll). This is because the Efficient Streaming design can execute in parallel but must update the line buffer before each convolution window. This results in area and energy usage requirements far lower than the other architectures.

The results of our design space exploration on LeNet-5 have provided insights that can help guide future exploration for much larger architectures, and we have applied these insights to aid in developing the full system configuration used for our

MobileNetV2 design space exploration in Sec. 5.2.

5.2 MobileNetV2 Exploration

With the new design automation features we have introduced in gem5-SALAMv2, we have enabled the ability to explore significantly more complex architectures over gem5-SALAMv1. While the LeNet-5 architectures showcase how a designer can easily tweak small architectural parameters, these are still toy examples that do not convey the complexity and scale of systems that gem5-SALAMv2 supports. With this in mind, we present a MobileNetV2 architecture that supports modern CNN features such as residual connections and separable convolutions to showcase a complex architecture implemented in gem5-SALAMv2.

With this increased complexity, it becomes unrealistic to fully map each network layer to an individual accelerator, as in Sec 5.1. Because we can create isolated accelerator clusters in gem5-SALAMv2, we break the MobileNetV2 architecture into four core computation blocks by assigning each block to an individual cluster. These compute clusters contain unique structures of the network, with the head, tail, and classifier being single-use clusters. Because of the dynamic reconfigurability of accelerators in gem5-SALAMv2, we can create a single-body cluster that can be reused. The system-wide architecture is shown in Figure 5.6, with the Depthwise (DW) and Pointwise (PW) functional units shown in further detail in Figures 5.7 and 5.8. Notably, the Classification cluster is left out of Figure 5.6 due to the straightforward nature of its design, but is present in the implementation of the network.

Figure 5.6 showcases the design of MobileNetV2 at a system level. We show this to describe the system and demonstrate how the SALAM Configurator significantly improves a designer's interaction with gem5 at the system design level. In gem5-SALAMv1, all memory connections between devices were manually defined within the same cluster; however, these connections are now an automated feature of gem5-SALAMv2 and require no manual configuration. This enables iterating on large-scale

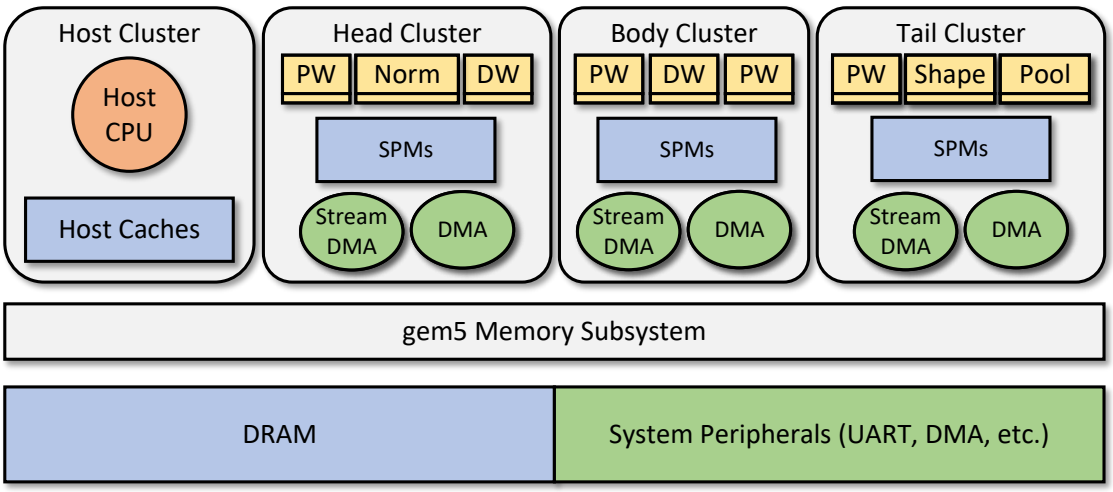


Figure 5.6: MobileNetV2 System Architecture - System architecture details for the MobileNetV2 design. The Head, Body, and Tail clusters are represented here and show how the design interfaces with the gem5 system.

hardware for applications such as MobileNetV2 at a significantly faster pace, as one no longer has to create and maintain the gem5 system configuration and memory map. Another addition that helps with organization is the ability to easily partition accelerators into isolated accelerator clusters, helping organize and maintain designs.

Because all clusters utilize Depthwise (DW), Pointwise (PW), or Normal convolutions, we create functional units (FUs) for each of these essential operations. The DW functional unit shown in Figure 5.7 is designed to resolve the producer-consumer disparities in CNNs discussed in Sec. 5.1. In this FU we create an internal Im2col accelerator for the DW convolution that processes data from an input FIFO stream to prepare the convolution window for the main compute accelerator. The PW functional unit has a different structure due to the access pattern of the PW operation and is shown in Figure 5.8. Here we can see that the PW FU is a single accelerator that is responsible for its own data management and computation. The Normal convolution shown in 5.6 is simply the DW FU but processes 3-channel RGB image. Each FU writes an output FIFO stream that feeds to the next FU in the chain. Because our point-wise accelerator does not need to reorder its input data, it manages

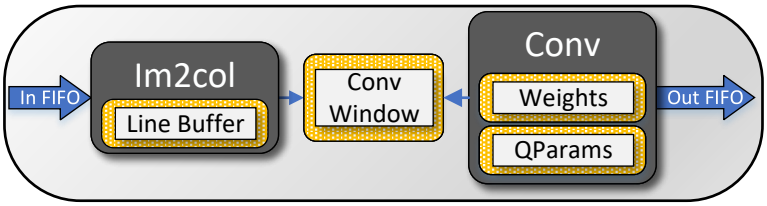


Figure 5.7: Provides an overview of the Depthwise (DW) functional unit used in this MobileNetv2 architecture. Each accelerator performs a discrete function, with memory types used being FIFO buffers or SPMs. The two accelerators are image-to-column transformation (Im2col) and the convolution window computation (Conv).

its convolution window and computation.

Using these functional units as the building blocks for our clusters, we create our four separate clusters and interconnect the functional units with FIFO buffers. Each cluster also contains DMAs for access to the main memory and a top-level accelerator to control memory transfers and accelerator initialization. Network inputs and outputs utilize FIFO buffers accessed by a Stream DMA that the Top configures. Weights and quantization parameters are transferred to their respective SPMs before accelerators begin computation.

The Head cluster is responsible for the first two layers of the network, a normal convolution and a unique inverted residual block (IRB). We present the Body cluster in Figure reffig:mobilenet-body, where most of the network's computation occurs. The Body contains an IRB and support for residual connections. As mentioned, we utilize SALAMv2's support for configurable accelerators to re-use the body cluster sequentially to process each subsequent network layer. The Tail cluster of the network embeds the features for the classifier and computes an average pool. Finally, the classifier runs the final fully connected layer of the network.

gem5-SALAMv2's increased design automation and configuration flexibility allow us to rapidly explore how our architecture performs with varying network complexities. Specifically, we vary one of MobileNetV2's hyper-parameters, α, across three different sizes: .35, .75, and 1. As shown in Table 5.2, there are significant changes

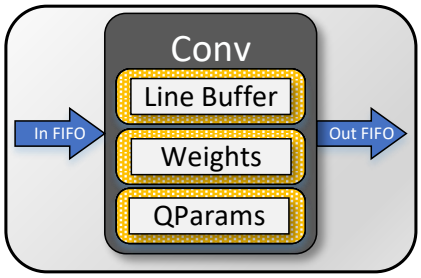


Figure 5.8: Describes the Pointwise (PW) functional unit used in the Head, Body, and Tail clusters. We see a single accelerator responsible for its data management and computation, enabled by gem5-SALAMv2's mechanism for the Conv accelerator to poll for the availability of data on the FIFO buffer.

in how much computation and memory must be utilized to perform operations on a given frame. Using gem5-SALAMv2's metrics allows us to perform an in-depth analysis of our proposed architecture.

Using these metrics, we run all three configurations and record the results in Table 5.1. We can see that our architecture performs as expected across all three complexities, but the system becomes significantly more constrained as computation complexity increases. We see this with the least compute-intensive variant getting 14.08fps and the most intensive variant getting 4.50fps.

With this data, we can analyze what is most constraining the execution of the network. We see that our Body cluster takes up approximately 88% of the end-to-end latency for an α of .75. Because this is the most complex portion of the network, further improvements, such as implementing a form of tiling or increasing parallelism, could be made to increase performance. Additional considerations, such as memory overheads, can also be made. For example, in our Classifier cluster, the initial loading of the network's weights takes 94% of the execution of the cluster. Another solution worth exploring in gem5-SALAMv2 would be to implement weight streaming in later stages of the network by making small changes to the Body. This is because parameter sizes in later layers become significantly larger than their respective feature maps.

Finally, we compare MobileNetV2 run-times in gem5-SALAMv1 and v2. To do

this, we back-ported the MobileNetV2 design and gem5-SALAMv2 Configurator into gem5-SALAMv1, as this exploration is not possible without the new additions. As expected, we see in Table 5.3 that there are significant performance benefits that enable more rapid exploration of large-scale architectures. Notably, we see a trend that confirms short-running segments of the network, such as the Tail, perform worse at -.58x, but the largest segment of the network, the Body, receives the largest speedup at 3.51x. As we scaled the network parameters up, the performance improvement on individual segments also improved. However, gem5-SALAMv1 could not run the fully scaled network ($\alpha = 1$) end-to-end in a single run for a proper comparison.

Table 5.1: MobileNetV2 96x96 Sim time and latency

Cluster	$\alpha = .35$		$\alpha = .75$		$\alpha = 1$	
	Sim Time (m)	Latency (us)	Sim Time (m)	Latency (us)	Sim Time (m)	Latency (us)
head	1.45	8250.55	2.86	11214.57	3.15	14360.49
body	24.25	58004.01	301.55	145388.46	448.05	200535.15
tail	0.04	1119.05	3.65	3529.61	4.32	3826.65
classifier	6.41	3605.61	6.98	3606.31	6.48	3606.73
Total	32.15	70.98ms	315.03	163.74ms	462.00	222.33ms

Table 5.2: MobileNetV2 Network Complexity for a 96x96 Input and a varying α at points .35, .75, and 1.0

MobileNetV2 Complexity	$\alpha = .35$	$\alpha = .75$	$\alpha = 1$
Computation (MACs)	13705412	6.61x	8.85x
Model Size (KB)	203	2.82x	4.32x
Feature Map Traffic (KB)	30702	6.04x	8.05x

Table 5.3: Runtime Comparison of MobileNetV2 on SALAMv1 and SALAMv2 with an input resolution of 96x96 and $\alpha = 0.35$

Cluster	V1 Time (m)	V2 Time (m)	Speedup
head	4.44	1.45	2.07x
Body	109.39	24.25	3.51x
Tail	0.02	0.04	-0.58x
Classifier	25.11	6.41	2.92x
Total	138.96	32.15	3.32x

CHAPTER 6: CONCLUSION

This paper presented gem5-SALAMv2 as a fully integrative LLVM-based simulation platform for scalable simulation of accelerator-rich SoCs. gem5-SALAMv2 extended the work presented in "gem5-SALAM: A System Architecture for LLVM-based Accelerator Modeling" from MICRO 2020 [13] by revamping the gem5-SALAM internals to provide more robust and extensible simulations, as well as introducing automation mechanisms for expanding and simplifying design space exploration. Importantly, while gem5-SALAMv1 sought to primarily focus on supporting application-specific compute simulators, gem5-SALAMv2 has expanded this scope to support broader hardware simulations with its LLVM Interface.

This is accomplished through a ground-up redesign of the SALAM elaboration and execution engine, improvements to the gem5 memory system integration, and expansion of the SALAM toolchain with tools such as the SALAM Configurator. With the introduction of these features, gem5-SALAMv2 offers unique hardware modeling opportunities that are otherwise unavailable in hardware and system-level simulation without extensive hand-crafted development. In addition to these modeling capabilities, gem5-SALAMv2 also provides significant speedups in simulation development, design iteration, and simulation times for large designs.

With these significant changes to SALAM and the generation and structure of LLVM IR, we re-validated the SALAM framework on the Machsuite [1] benchmarks that were also used as a baseline in gem5-SALAMv1. Doing so gave us the unique opportunity to compare the validation and runtime differences between the two versions. We found that on average, gem5-SALAMv2 had an error of 4.06% for cycle count, 2.45% for power, and 3.13% for area when compared to the Vivado HLS

implementation of MachSuite benchmarks. When comparing the execution time of gem5-SALAMv1 and gem5-SALAMv2, we found that for very small simulations, the utilization of the LLVM API introduced significant setup overheads ($\sim$3x), while larger simulations like Stencil3d and GEMM saw performance improvements of more than 11x.

In addition to validating the MachSuite benchmarks in gem5-SALAMv2, we illustrated the increased flexibility of our framework for design space exploration by demonstrating several hardware architectures for the LeNet-5 and MobileNetV2 CNNs. These designs were made possible by the SALAM Configurator, a tool that automates gem5 system configuration files and memory maps for an end user. For LeNet, we presented three architectures with unique characteristics to demonstrate how a user can rapidly iterate on a design in gem5-SALAMv2. For MobileNetV2, we implemented one large architecture with more than 150 memory-mapped devices and changed the hyperparameter α to inspect how the gem5-SALAMv2 framework would perform on a large-scale design. We further compared runtime between SALAMv1 and SALAMv2 by backporting the MobileNetV2 design to SALAMv1 and found a performance improvement of 3.32x.

With the open-endedness of both gem5 and gem5-SALAM, there is significant leeway on topics that could be further researched. As we have demonstrated in Section 5, gem5-SALAMv2 provides a reasonable simulation framework for CNN-based architectures. A natural next step would be simulating compute and memory-intensive architectures such as LLM-like models, e.g. GPT4 and LLAMA2. Additionally, research into how specific accelerators interact within a heterogeneous system could be worthwhile, particularly with the addition of GPU support in gem5. Another aspect that we have not explored is the impact that LLVM vectorization instructions would have on simulation times with these applications. While technically possible with our usage of the LLVM IR, this is not something we currently implement but would most

likely result in a significant speedup for these kinds of applications.

REFERENCES

[1] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization*, (Raleigh, North Carolina), October 2014.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[3] C. Menard, J. Castrillon, M. Jung, and N. Wehn, "System simulation with gem5 and systemc: The keystone for full interoperability," in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 62–69, July 2017.

[4] T. Nikolaos, K. Georgopoulos, and Y. Papaefstathiou, "A novel way to efficiently simulate complex full systems incorporating hardware accelerators," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 658–661, March 2017.

[5] K. Iordanou, O. Palomar, J. Mawer, C. Gorgovan, A. Nisbet, and M. LujÃ¡n, "Simacc: A configurable cycle-accurate simulator for customized accelerators on cpu-fpgas socs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 163–171, April 2019.

[6] C. Pham-Quoc, I. Ashraf, Z. Al-Ars, and K. Bertels, "Heterogeneous hardware accelerators with hybrid interconnect: An automated design approach," in *2015 International Conference on Advanced Computing and Applications (ACOMP)*, pp. 59–66, Nov 2015.

[7] T. Liang, L. Feng, S. Sinha, and W. Zhang, "Paas: A system level simulator for heterogeneous computing architectures," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sep. 2017.

[8] K. Gent and M. S. Hsiao, "Functional test generation at the rtl using swarm intelligence and bounded model checking," in *2013 22nd Asian Test Symposium*, pp. 233–238, Nov 2013.

[9] Y. S. Shao, B. Reagan, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[10] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin," in *The 49th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[11] J. Cong, Z. Fang, M. Gill, and G. Reinman, "Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.

[12] O. Matthews, A. Manocha, D. Giri, M. Orenes-Vera, E. Tureci, T. Sorensen, T. J. Ham, J. L. Aragón, L. P. Carloni, and M. Martonosi, "The mosaicsim simulator (full technical report)," *CoRR*, vol. abs/2004.07415, 2020.

[13] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi, "gem5-salam: A system architecture for llvm-based accelerator modeling," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 471–482, 2020.

[14] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, 2004.

[15] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[16] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2019.

[17] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, pp. 24:1–24:27, Sept. 2013.

[18] D. H. Noronha, B. Salehpour, and S. J. E. Wilton, "Leflow: Enabling flexible FPGA high-level synthesis of tensorflow deep neural networks," *CoRR*, vol. abs/1807.05317, 2018.

[19] W. Zuo, L. Pouchet, A. Ayupov, T. Kim, Chung-Wei Lin, S. Shiraishi, and D. Chen, "Accurate high-level modeling and automated hardware/software co-design for effective soc design space exploration," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2017.

[20] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm, and A. Shriraman, "Needle: Leveraging program analysis to analyze and extract accelerators from whole programs," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 565–576, Feb 2017.

[21] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically Specialized Datapaths for energy efficient computing," in *High Performance Computer Architecture (HPCA)*, pp. 503–514, 2011.

[22] L. Wang and K. Skadron, "Lumos+: Rapid, pre-rtl design space exploration on accelerator-rich heterogeneous architectures with reconfigurable logic," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 328–335, Oct 2016.

[23] M. S. B. Altaf and D. A. Wood, "Logca: A high-level performance model for hardware accelerators," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 375–388, June 2017.

[24] F. Munoz-MartÃnez, J. L. Abellan, M. E. Acacio, and T. Krishna, "Stonne: Enabling cycle-level microarchitectural simulation for dnn inference accelerators," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 201–213, 2021.

[25] A. Samajdar, Y. Zhu, P. N. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic CNN accelerator," *CoRR*, vol. abs/1811.02883, 2018.

[26] L. Zhu, W. Fan, C. Dai, S. Zhou, Y. Xue, Z. Lu, L. Li, and Y. Fu, "A noc-based spatial dnn inference accelerator with memory-friendly dataflow," *IEEE Design Test*, vol. 40, no. 6, pp. 39–50, 2023.

[27] Y. Zhao, C. Li, Y. Wang, P. Xu, Y. Zhang, and Y. Lin, "Dnn-chip predictor: An analytical performance predictor for DNN accelerators with various dataflows and hardware architectures," *CoRR*, vol. abs/2002.11270, 2020.

[28] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 304–315, 2019.

[29] Y. Wu, P. Tsai, A. Parashar, V. Sze, and J. S. Emer, "Sparseloop: An analytical approach to sparse tensor accelerator modeling," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, (Los Alamitos, CA, USA), pp. 1377–1395, IEEE Computer Society, oct 2022.

[30] S. Rogers, J. Slycord, R. Raheja, and H. Tabkhi, "Scalable llvm-based accelerator modeling in gem5," *IEEE Computer Architecture Letters*, pp. 18–21, jan 2019.

[31] 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 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480, 2009.