DISSERTATION - LEVERAGING LLVM IR FOR DESIGN SPACE EXPLORATION AND MODELING OF APPLICATION AND DOMAIN-SPECIFIC HARDWARE

by

Samuel Rogers

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2022

Approved by:

Dr. Hamed Tabkhi

Dr. Arun Ravindran

Dr. Andrew Willis

Dr. Tsing-Hua Her

Abstract

SAMUEL ROGERS. Dissertation - Leveraging LLVM IR for Design Space Exploration and Modeling of Application and Domain-Specific Hardware. (Under the direction of DR. HAMED TABKHI)

The limitations of transistor scaling and the rise of power-constrained compute environments have driven a new renaissance in hardware design through hardware acceleration. Application and domain-specific hardware accelerators offer fantastic efficiencies over traditional, general-purpose processors and are seeing extensive use in applications ranging from embedded platforms to high-performance cloud computing clusters.

Developing new hardware accelerators comes at a great cost in terms of both time and engineering effort. It is greatly outpaced by rapidly evolving algorithmic developments, particularly in emergent domains like AI and deep learning. Hardware developers need new, more agile tools to aid in both simulation and integration of hardware accelerators in modern systems.

To aid in the integration and optimization of application-specific memories for hardware accelerators this work presents two automated solutions based on the Low Level Virtual Machine (LLVM) compiler infrastructure. These solutions explore static data access characteristics of applications to guide hardware designers in optimizing memory hierarchy prior to hardware synthesis. They are also compatible with existing synthesis tool-chains to enable some automated memory optimizations when synthesizing accelerator designs.

This work also introduces a new design space exploration tool for application and domain-specific accelerators in the form of the gem5-SALAM simulator. This simulator provides high fidelity modeling of accelerator power and performance within a full system simulation. It also significantly reduces the amount of time and effort needed to explore the system design space for accelerators versus traditional hardware design flows.

Lastly the work presented as part of gem5-SALAM's revision introduces a new set of simulation abstractions for redefining how simulation models are written for event-driven simulators. These abstractions address the fundamental design challenge of decoupling simulated functionality from timing and concurrency constraints, while maintaining full compatibility and interoperability with

the popular open-source simulation framework gem5.

ACKNOWLEDGEMENTS

I would like to acknowledge and thank my advisor and mentor Dr. Hamed Tabkhi for the support he has given me in my research. Not only has he provided guidance and insight within the fields of computer architecture and academic research, he has also provided valuable guidance and mentoring that has helped me better balance my work and home life.

I would also like to thank the members of the TeCSAR lab for their support in my research. In particular I would like to acknowledge Arnab Purkayastha and Suhas Shiddhibhavi, with whom I developed the OpenCL analysis research. Their expertise and work on the OpenCL synthesis to FPGA component of the research enabled me to focus my efforts on algorithmic analysis.

I would also like to acknowledge and thank Joshua Slycord for his efforts in datapath modeling in the gem5-SALAM project. With his work we have been able to develop a truly amazing tool and publish it in a highly esteemed journal and conference. My thanks also go out to the newest member of the SALAM team, Zephaniah Spencer, with whom I look forward to continue working.

I would also like to thank Dr. Arrvindh Shriraman and his students at Simon Fraiser University for their collaboration and guidance for gem5-SALAM's development. Insights from their hardware research and usage of gem5-SALAM in classroom instruction helped to identify necessary improvements for gem5-SALAM.

Sincere thanks go out to my doctoral committee, Dr. Andrew Willis and Dr. Arun Ranvindran for their help and guidance both in and out of the classroom. I greatly value and appreciate the time you have taken to help me in my PhD. work, and in the guidance for post-graduate life. My thanks also goes out to Dr. Tsing-Hua Her for her participation in my doctoral committee as the external graduate faculty member.

Finally, I would like to acknowledge and thank my loving wife, Jenna Rogers, for her constant support throughout every part of life both in and out of school.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API Application Programming Interface.

CDFG Control and Dataflow Graph

CPU Central Processing Unit.

DMA Direct Memory Access

DNN Deep Neural Network

DSE Design Space Exploration

FPGA Field Programmable Gate Array.

GPU Graphical Processing Unit.

HDL Hardware Description Language

HLS High-level synthesis.

IR Intermediate Representation

ISA Instruction Set Architecture.

LLVM Low Level Virtual Machine.

OpenCL Open Compute Language.

RTL Register Transistor Logic.

SDK Software Development Kit.

SoC System on Chip

SPM Scratchpad Memory

CHAPTER 1: INTRODUCTION

With transistor technology reaching the fundamental limits of Dennard scaling and Amdahl's law, hardware designers are searching for innovative new means for driving device performance and power efficiency. Ironically many recent advances in compute technologies have in fact been reversions from general purpose processors to application-specific designs that can leverage the dark silicon on modern systems on chips (SoCs) with highly optimized and power-efficient datapaths. The fundamental limitation has thus become the costs of developing, testing, and integrating new hardware in a domain where algorithmic advances rapidly eclipse the capabilities of new hardware. To amortize some of the expenses associated with the development of hardware accelerators, we have also seen the development of domain-specific accelerators. These types of devices trade some of the efficiency of fully application-specific accelerators for limited programmability to support a small subset of applications, or domain. The most recognizable example of domain-specific accelerators is the integrated graphics processors found in many modern CPUs.

One domain that has been a particularly significant driver in domain-specific processor design has been deep learning. Deep learning applications are characterized by massively parallel computation along with equally impressive amounts of data movement. Deep learning became popular after general-purpose GPUs provided a platform with the necessary scale to handle the data-intensive training regimens characteristic of neural networks. However, the hardware requirements of modern deep neural networks (DNNs) rapidly outpaced that capacity of general-purpose GPUs. This lead to the rise of domain-specific accelerators for deep learning such as Google's Tensor Processing Unit (TPU) [1] and NVIDIA's tensor cores and open-source NVDLA accelerator. These accelerators leverage large systolic processing arrays to exploit the compute parallelism present in DNNs while supporting flexibility in data movement as data demands shift from input feature limited to network parameter limited. Unfortunately the deep learning application domain has evolved significantly to leverage smaller, sparser networks that swap computation for data-dependent control and hand-

crafted data movement between network blocks. The lower parallelism and increasing runtime control leads to lower occupancy of systolic resources and makes systolic processors like the TPU or NVDLA increasingly inefficient for modern neural networks.

An alternate approach to domain-specific acceleration is to construct a domain-level processor using a series of application-specific processors alongside a flexible interconnect and interfaces. This approach was advocated for in [2], and more recently realized in markets with Xilinx's Versal ACAP [3]. These designs offer the promise of increased flexibility over traditional systolic arrays for emergent deep learning applications, while still offering the efficiency and performance benefits of domain-specific processing.

1.1 Hardware Design Tools and Design Space Exploration

The field of hardware design automation and design space exploration is quite rich with solutions for synthesis, coverage testing, and layout. The process of taking a new design from Register-Transfer Logic (RTL) simulation to chip tapeout has never been easier, with industrial-quality tool chains becoming more and more available to end users. Tool chains like AMD/Xilinx's Vitis Unified Software Platform even bring high performance High Level Synthesis (HLS) tools to developers that are more comfortable with software development than hardware development. Alternatively, expanding support for high-level language constructs in high-level hardware description languages like SystemC, SystemVerilog, and Chisel[4] provide opportunities to develop hardware with a higher degree of abstraction.

Despite robust tool support for post-RTL development, the tool space for pre-RTL design and early design space exploration is much weaker. This space is far more dominated by academic interests that leverage simulation and analytical models for evaluating the potential benefits of new architecture ideas prior to the large time and money investment of extensive RTL development. Software-based simulation solutions offer important opportunities to explore key design considerations around hardware structure and integration long before a firm architecture has been established. Full-system simulations, like those offered by gem5[5], enable designers to evaluate architecture ideas and identify integration challenges ranging from memory interfaces and I/O, to driver support in modern operating systems. Importantly these simulations retain their value even into RTL develop-

ment due to their greater capacity for rapid design space exploration. While simulation solutions are fantastic for modeling classical CPU and GPU-based systems, their support for more fundamental hardware design is severely lacking. Numerous efforts have been made to integrate custom hardware accelerators and other heterogeneous compute elements to simulation, including [6, 7, 8, 9, 10, 11]. Unfortunately these solutions are generally highly tailored to a particular integration challenge and ultimately fail to provide a generalized solution to integrating new custom hardware models into gem5 and other simulators.

A side effect of the lack of support for custom hardware solutions in early design space exploration tools is the increased challenge for exploring memory optimization of such devices. Even the industrial tool chains for RTL development largely lock users into predefined intellectual properties built on top of standards like AXI, AXI-Stream, and AXI-Lite. Meanwhile most pre-RTL solutions focus more on datapath modeling or their own customized solutions for data tiling[9, 8]. As a result there is a lack of tools for both optimizing and exploring unique memory solutions prior to extensive RTL development.

1.2 Contributions

Given the rapidly evolving nature of both hardware and algorithmic design for deep learning applications it is important to have robust tools for hardware-software co-design. Given the heavy data-dependence of DNNs it is just as important to optimize device memory and system integration as it is to optimize computation. There is a need for tools that can both analyze the design constraints of emergent AI applications and rapidly prototype designs in order to explore system level impacts of design optimizations. To this end my work seeks to make three key contributions:

- Automate memory access analysis for memory optimization of hardware accelerators including variable-level memory assignment and allocation[12], and automated hardware prefetching on FPGAs[13].
- Produce an innovative pre-RTL simulation tool for design space exploration of custom hardware accelerators[14, 15].
- Introduce new simulation design abstractions that bring hardware-level insights into event

driven simulator design.

1.3 Report Outline

The remainder of this report is organized as follows. Chapter 2 covers relevant background knowledge and covers other works that relate to the optimization and design-space exploration of hardware accelerators. Chapter 3 and Chapter 4 highlight implementation of the my first contribution through two associated works in hardware accelerator memory optimization. Chapter 5 describes my second contribution by introducing and detailing the design-space exploration tool gem5-SALAM. Chapter 6 takes a deeper dive into the challenges of event-driven simulation, and how gem5-SALAMv2 seeks to fundamentally address these challenges with new design abstractions. Additionally chapter 6 describes the process of redesigning the original gem5-SALAM from the ground up to provide a robust simulation framework for modern compute architectures. Chapter 7 contains concluding remarks from this report.

CHAPTER 2: BACKGROUND AND RELATED WORK

2.1 The Low-Level Virtual Machine (LLVM) Tool Chain

The translation of human readable code in high level programming abstractions to optimized machine code is a complicated and ever-growing task. In the past every new combination of source language and target architecture required a customized compiler solution. This lead to numerous redundancies since the vast majority of code optimizations are completely independent of source language semantics and target architecture structures. Seeing an opportunity to unify and improve code compilation and optimization, researchers developed the Low-Level Virtual Machine (LLVM) infrastructure[16]. LLVM is a modular code optimization infrastructure built around a unique intermediate representation (IR). This IR provides a level of abstraction similar to that found in many machine instruction sets, while maintaining high level control and data flow semantics normally associated with high-level languages. By providing this level of abstraction, LLVM is able to efficiently optimize code in a way that is independent of both source language and target hardware architecture. LLVM also provides a convenient abstraction for algorithmic analysis that makes it popular in non-traditional compilation tasks such as high-level synthesis (HLS) tools. Since its inception LLVM has gained enormous support from both academia and industry, featuring extensively in numerous proprietary tool chains as well as serving as the foundation for new programming languages like Julia.

2.2 Design and Exploration of Hardware Accelerators

One of the biggest challenges for designing and integrating new accelerators into modern SoCs is simulating and exploring design parameters. Often accelerators are initially designed and tested in isolation, which can lead to over-tuning of design parameters based on idealized assumptions about data availability and other system overheads. While synthesis and RTL simulations of accelerators in isolation may be reasonably quick, full-system RTL simulation of an SoC can often take days or longer. As a result, electronic design automation (EDA) companies, like Synopsys, have developed

proprietary tools like Platform Architect that abstract many system-level elements and provide cycle-accurate performance estimations of SoC designs incorporating accelerators through SystemC and transaction-level models (TLM).

As hardware design has shifted from classical RTL design flows to more software developer-friendly approaches the LLVM compiler [16], and its IR, has become a key component of many design flows. Popular HLS tools like Vivado and LegUp [17] internally use a modified clang tool flow for translating hard descriptions written in C to popular RTL targets such as Verilog, VHDL, and SystemC. With growing interests in deep learning acceleration the LeFlow project [18] was developed to integrate TensorFlow's XLA compiler with LegUp and enable the synthesis of deep learning accelerators on FPGAs. In addition to synthesis tools, LLVM has also been employed for pre-RTL design space exploration. The RIP framework [19] leverages LLVM for the identification and modeling of "hot loops" in an application in order to design accelerators for those portions of code. Similarly, Needle [20] and the work described in [21] leverage LLVM for detecting hot portions of code and automatically generating accelerators for DySER-styled [22] architectures. For exploring the design and system-level impacts of loosely-coupled hardware accelerators Lumos+ [23] and LogCa [24] tools can be used. These approaches employ analytical modeling for estimating the power, performance, and area requirements of hardware accelerators in highly heterogeneous accelerator-rich systems.

For researchers that do not have access to proprietary EDA tools, the gem5 system simulation[5] and its derivatives have become a popular solution. gem5 is an open-source, industry-backed system simulator based on a joint C++/Python programming abstraction. This provides cycle-accurate system performance estimation at an abstraction that is more accessible to developers who are looking for alternatives to proprietary and RTL-based development tools. Furthermore, its open-source C++/Python API enables the extension of its capabilities. While the base API of gem5 supports a wide variety of models for system elements such as memory, CPUs, GPUs, and busses, it lacks modeling for application-specific hardware accelerators. To address this shortcoming, researchers have sought ways to integrate application-specific accelerator modeling into gem5.

gem5-Aladdin [7] integrates the Aladdin[6] pre-RTL simulator into gem5. The base Aladdin simulator relies on instrumenting LLVM IR [16], associated with C descriptions of hardware accelerator

functionality, in order to generate a runtime trace of executed LLVM instructions. This runtime trace is then loaded into the simulation engine, where it is further optimized and instrumented with timing information before being passed to an event-driven simulation engine. While Aladdin demonstrates impressive accuracy for a pre-RTL simulation model, its reliance on runtime traces can lead to unreliable results for irregular applications. This is the result of its approach of reverse engineering a datapath based on the parallelism of the dynamic instruction trace. In applications where execution semantics and parallelism vary depending on input data, Aladdin will generate different datapaths for the same kernel source code, as the input data changes. Table 2.1 demonstrates this behavior for the Sparse Matrix-Vector Multiplication (SPMV) built around the Compact Row Storage (CRS) data format. In the source code we added a one bit-shift operation that would activate if the input value fell within an arbitrary range we defined, then included this value in one dataset but not the other to demonstrate this shortcoming. As illustrated in Table 2.1, the number of floating-point adders in the datapaths change between two runs with the same kernel code but different input data sets. Additionally, since the value that triggered the shift condition was not in the data set for the first run of the application, Aladdin did not model the shift operation as part of the datapath.

The system integration of Aladdin into gem5-Aladdin introduces new limitations. Here, adjustments to system parameters such as cache line size and accelerator cache size, which can impact data availability, have the effect of changing Aladdin's datapath and power estimation for even regular applications. This characteristic is demonstrated in Table 2.2. In this scenario, a sweep of the highly regular GEMM application is run over varying cache sizes. As the sizes of the caches are varied, the number of allocated functional units also changes. Since Aladdin is reverse-engineering the datapath, changes in lookup times and cache hits/misses affect the availability of data and subsequently, the simulated datapath. Table 2.2 also shows that switching over to a multi-ported

Table 2.1: Aladdin Datapath vs Data-dependent Execution

Accelerator	Dataset	Functional Units		
		FMUL	FADD	Int Shifter
SPMV-CRS	1	8	4	0
	2	8	8	1

Table 2.2: Aladdin datapath vs. memory design

Accelerator	Memory		Functional Units	
	Type	Size	FMUL	FADD
GEMM N-Cubed (Fully unrolled)	Cache	256B	665	879
		512B	679	903
		1kB	696	928
		2kB	712	948
		4kB	639	843
		8kB	650	864
		16kB	468	624
	SPM		194	258

ScratchPad Memory (SPM) has a significant impact on the datapath that Aladdin simulates. While a hardware developer would certainly want to co-optimize both datapath and memory hierarchy, gem5-Aladdin does not provide developers with the means of decoupling the generated datapath from the impacts of the memory hierarchy. Developers should have the capacity to independently sweep design parameters for both datapath and memory[1].

Another limitation of gem5-Aladdin stems from the way it has integrated into gem5's system infrastructure. The gem5-Aladdin project exposes the Aladdin simulator to gem5's system infrastructure by creating a gem5 object wrapper. However, Aladdin's wrapper is only partially integrated into gem5. The gem5-Aladdin's partial integration inhibits system-level exploration and prototyping of common accelerator-rich scenarios in modern SoCs. The gem5 component of gem5-Aladdin merely serves accounting for memory latency in the performance estimates of individual accelerators. As an example, to transfer data between the accelerator's private SPM and dynamic RAM (DRAM), direct memory access (DMA) operations must be exposed directly to the accelerator source code, so that they can appear in the runtime trace. The consequence is that accelerators and their private SPMs lack the standard communication ports necessary for intercommunication between devices in the system. Another example is that MMRs and other traditional interfaces are not included in the accelerator wrapper in gem5-Aladdin. The host CPU instead communicates with accelerators via a software bypass integrated into gem5's Syscall Emulation (SE) simulation framework. Furthermore,

[1]Both the runtime data tests and the memory hierarchy tests were conducted using the latest build of gem5-Aladdin as of April 2020.

these interfaces cannot merely be added to the Aladdin wrapper without violating design assumptions that are imperative to Aladdin's integration into gem5. Doing so would require a complete redesign of the Aladdin wrapper, custom system elements (i.e., DMAs, SPMs, etc.), device drivers, and user programming experience.

The other gem5 extension, PARADE[8], simulates multi-accelerator architectures based on ARCMP [25]. This simulator offers a full-system simulation with a full Linux kernel in gem5. Accelerators are designed via a scarcely-documented and proprietary custom data-flow language while power and timing information are extracted from HLS. Since PARADE relies on static estimates of power and timing from HLS, it is unable to accurately model accelerators where power and timing can vary based on input data. PARADE also enforces a fairly rigid system architecture, limiting the scope of design space exploration to the model proposed in ARCMP[25]. Rather than being able to communicate directly with accelerators within PARADE's framework, system elements instead send all requests to PARADE's Global Accelerator Manager (GAM) which facilitates all communication within the PARADE framework, as well as across the boundary between gem5 and PARADE. Furthermore, there is no means by which a user can control or modify the internal timing and data flow within PARADE's framework. This includes the order of execution of cascaded accelerators that operate in a producer-consumer fashion, and can result in the out-of-order operation of accelerators. Similar to gem5-Aladdin, any designs that don't precisely match the original design specification of ARCMP[25] and its GAM, require an extensive redesign of PARADE's system elements, scheduler, and even the dataflow language used for describing accelerators.

Another recent tool that also leverages LLVM for parsing and instrumentation is the MosaicSim tool [9], 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. While this can provide rough approximations for the compute performance of an accelerated segment of an LLVM IR compute graph, it lacks many of the fine-grained controls needed to model or profile an application-specific datapath. For instance, users lack the capacity to 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 does also

support integration of 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, as well as a control-flow trace to track the progression of basic blocks. Reliance on execution traces imposes several major restrictions on modeling in MosaicSim. Trace sizes, generation times, and read times introduce significant overheads for simulation. For very large and long-running applications, traces may exceed several GB in size. Additionally, for applications where the load/store and control flow behaviors are dictated by input data, users need to generate new traces for each new set of input data.

For researchers who are more comfortable with SystemC development, gem5 now supports the direct integration of SystemC models [11]. SystemC support in gem5 was achieved by translating transactions within the TLM2.0 IEEE1666 standard to gem5's internal memory transaction model. This offers the most opportunities for design space exploration and simulation, however as an HDL-based option, it will also require a higher degree of design effort than the other options.

2.3 Memory Optimization Challenges for Hardware Accelerators

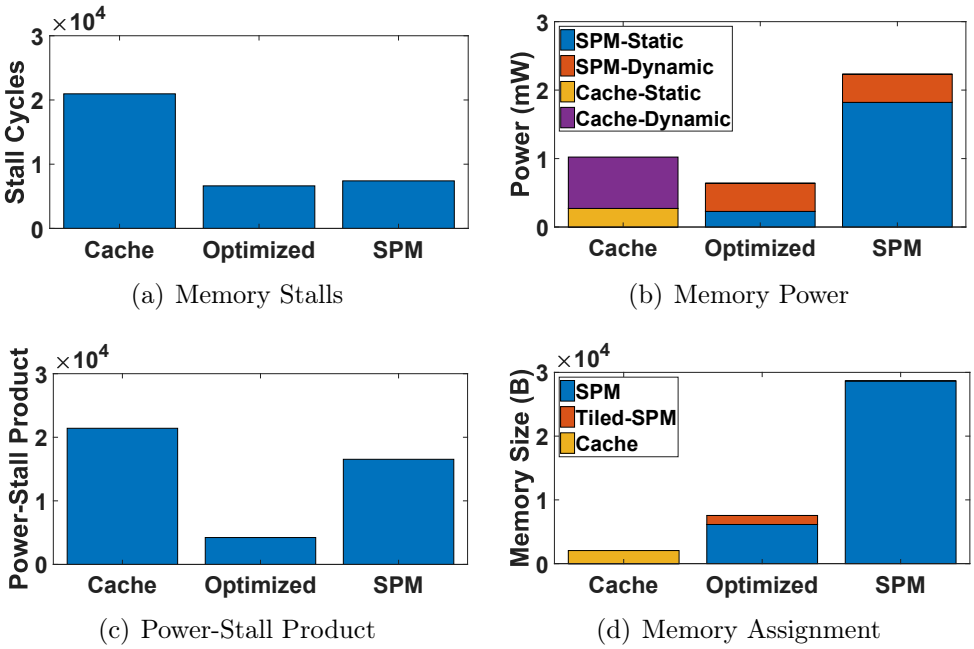
The optimization of memory hierarchies has been studied extensively in general-purpose CPUs. Many innovative solutions have been developed for memory-path optimization (cache vs. scratchpad memory) on embedded CPUs [26, 27, 28, 29] and extensive work gone toward optimizing hybrid memory structures for domain specific CPUs. Such optimizations have shown to lead to significant improvements (20% or more) in both raw performance and overall power consumption within a specific domain [27, 29].

At the same time, while a large amount of effort has been focused on the design and exploration of application-specific datapaths, the exploration of application-specific memory hierarchies has not been explored in depth. For the purpose of simplifying accelerator design, and regularizing data access times in accelerators, hardware architecture generally rely of scratchpad memories and streaming interfaces for data access. The general approach isn't all that different than preloading regularly accessed data in the shared memory of GPUs for improved access times. With this approach large data structures with predictable access patterns can be divided into smaller tiles,

and the costs of off-chip memory access can be hidden by overlapping the execution on one data tile with the movement of subsequent tiles to the scratchpad in a process called *double buffering*. Double buffering provides fantastic efficiencies in hardware design by amortizing off-chip memory access costs, regularizing data access latencies, and limiting the amount of expensive on-chip memory resources that must be allocated to support accessing large data structures[30]. Unfortunately this approach cannot be used on structures with sparse or irregular data access, as seen in sparse matrix operations and my graph traversal algorithms. In these cases the entire structure must be allocated and transferred to the scratchpad memory, regardless of how much of the structure is actually used by the accelerator.

Faced with the limitations of scratchpad and streaming memories for irregular data access in applications, some hardware designers have begun to look into using caching methods employed by modern CPUs and GPUs in their accelerator designs[7]. Caches trade the highly regular access times of scratchpad memories for a smaller memory footprint. By exploiting spatial and temporal locality in data access, caches are able to mitigate off-chip memory accesses while storing a limited subset of the application data. Unfortunately miss penalties for data not present in the cache are high, and the initial access of data in off-chip memory cannot be amortized in the same way that streaming and scratchpad-based memory access can be. This makes caches sub-optimal for accessing highly regular data structures.

In the same way that optimization of an accelerator datapath is highly application-specific, design and optimization of an accelerator memory hierarchy should also consider algorithm-intrinsic characteristics. Fig. 2.1 shows a comparison of different memory configurations for a K-Nearest Neighbors application from the MachSuite [31] benchmark suite over a series of parameters included datapath stalls, power consumption, and allocated memory size. Additionally a new metric, called the power-stall product, is introduced to capture the co-optimization of memory by minimizing power consumption and memory stalls exposed to the datapath. A lower power-stall product is better in this evaluation. The particular K-Nearest Neighbors application explored in Fig. 2.1 consists of three input arrays with runtime data-dependent memory access, one input array with nicely strided data access, and three output arrays with strided data access. Allocating all of these arrays to a scratchpad would require approximately 28KB of on-chip memory. Given the nicely strided



(a) Memory Stalls

(b) Memory Power

(c) Power-Stall Product

(d) Memory Assignment

Figure 2.1: MD-KNN Exploration

data access pattern and large size of four of the seven arrays in the application, a tiled or streaming memory can reduce the required memory allocation by more than three times vs. the monolithic scratchpad memory approach. Execution stalls can also be reduced since the data movement of the streaming and tiled data can be overlapped with computation.

It should be noted that optimizing purely for on-chip memory size would suggest that a cache is the optimal allocation. A sweep of the cache design space for this application found that there was no performance benefit of increasing cache size beyond 2KB. This is because caches are able to exploit data locality to capture the *working set* of an application without needing to store all of an applications data at once. This benefit becomes more pronounced as data size and sparsity increase, making it ideal for applications with runtime-dependent control over large data structures. The downside to caching is the cost of cold start misses the first time data is loaded to cache, as well as the potential need to re-fetch data that is replaced in the cache. Since caches move data at a smaller granularity than the direct memory access (DMA) engines used in scratchpad memory-based approaches, they are less able to amortize the timing and energy costs of mass data transfer from off-chip memory. This results in lower power-stall product efficiency per byte of memory allocated for caches vs. scratchpads. Despite this lower efficiency, the monolithic cache approach achieves

nearly the same level of power-stall product efficiency as a monolithic SPM design, and in some applications the optimal allocation will lean in favor of caching. The challenge then is to identify when data should be cached vs. streamed or bulk transferred to SPMs.

2.4 OpenCL Synthesis and Memory Optimization on FPGAs

Availability of OpenCL synthesis for FPGA development opens up many interesting research areas for developers and users of HLS tools alike. Many works study the efficiency of common massively-parallel workloads on FPGAs devices[32, 33, 34, 35, 36, 37, 38, 39, 40]. This includes the work of Purkayastha et al.[35], which introduced a generic taxonomy to classify and maximize parallelism potential on Intel FPGAs. These approaches primarily focus on application-specific performance optimization techniques [33, 34], or making performance comparisons between FPGAs and GPUs [32, 35].

There is also an interest in using OpenCL pipe semantic for efficient inter-kernel communication across multiple OpenCL kernels [41, 42, 43, 44]. The pipe semantic has been leveraged as part of both the Alterra and Xilinx HLS[45] solutions for synthesizing efficient data movement hardware between design units.However, the above implementation all focus on more traditional producer-consumer buffers between compute kernels that mirrors the well-established AXI-stream semantic.

These design principles are largely explored in the context of multi-threaded, multi-kernel workloads within the reconfigurable computing community[46, 47, 48, 49, 50, 51, 52, 53, 54]. These works generally focus on context switching and partial reconfiguration to support multiple applications on the same hardware. Some work has been done to study multi-threading on singular applications with a higher emphasis on threading performance[55, 56]. Unfortunately multi-threading support is still quite limited in HLS-based design flows that treat OpenCL threads like traditional hardware loops.

The idea of decoupling memory access from compute is well established within the architecture community[57]. Usually works have focused on general-purpose architectures (e.g., CPUs, GPUs, array processors) or application-specific accelerators with a fixed hardware [58, 59, 60]. Works like Domain-Specific Design Space Exploration[39] and Function-Level Processor[40] attempt to extend these concepts to the scope of domain processors. While these works briefly explore the

compute/memory decoupling concept within the scope of broader compute decoupling across the domain, they do not offer any specific solution for memory and compute decoupling.

One possible solution to the decoupling problem is through pre-fetching[61, 62]. Pre-fetching leverages an understanding of data access features within an application in order to load the necessary data prior to its usage. Pre-fetching can either be implemented in hardware such as is the case in [63, 64, 65] or in software as seen in [66, 67, 68]. Some efforts have been made to develop joint hardware/software implementations for pre-fetching[69], particularly with respect to address-based cache pre-fetching and optimization[70, 71].

CHAPTER 3: LOCALITY AWARE MEMORY ASSIGNMENT AND TILING

The first contribution of this research is to aid in and automate the analysis of hardware accelerator memories based on the algorithm-intrinsic characteristics of the accelerated application, presented as Locality Aware Memory Assignment and Tiling in the Design Automation Conference 2018[12].

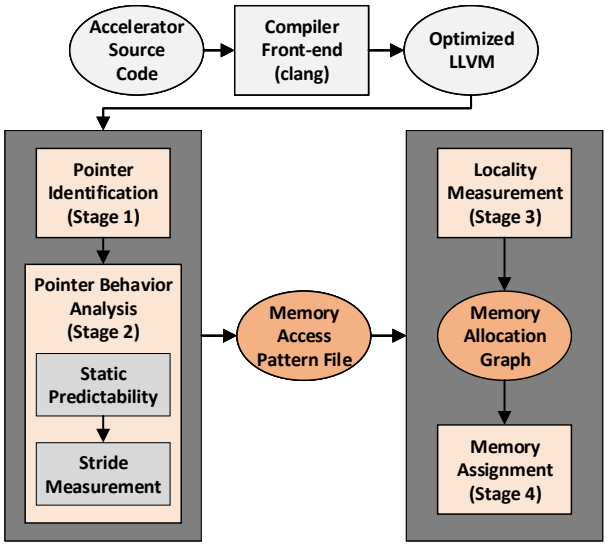


Figure 3.1: Memory Allocation Tool Chain

Rather than sweeping over the entire design space of memory hierarchies for an accelerator I have developed an approach and tool, shown in Fig. 3.1, that examines the algorithm-intrinsic characteristics of memory access in an application, and guides hardware developers in designing a locality-aware memory assignment and tiling design. This process can be broken into two main phases: (1) static memory access analysis and (2) variable-level memory assignment.

3.1 Static Memory Access Analysis

The static memory analysis is performed at the abstraction provided by the LLVM intermediate representation (IR) where memory access strides can be examined on a per-variable basis. As shown in Fig. 3.1, a user starts by compiling the accelerated application to the LLVM IR using the corresponding LLVM frontend compiler (clang for C/C++ applications). This LLVM is then

parsed in two phases: (1) Pointer Identification and (2) Pointer Behavior Analysis.

3.1.1 Pointer Identification

The first pass over the IR focuses on identifying the addressing of memory pointers. Pointers are organized by function call and are classified as local or global with respect to the function in which they are called. This parse works at the function granularity in order to limit analysis of the memory accesses to static behaviors that can be understand at compile-time. The general flow of this parse is shown in Fig. 3.2.

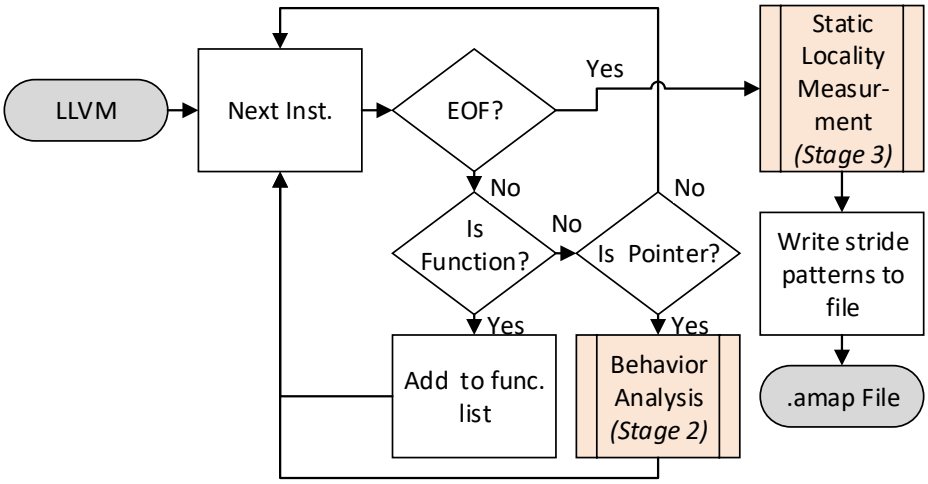


Figure 3.2: Pointer Identification (Stage 1)

Whenever a new memory access is found, corresponding addressing instruction is added to the pointers access list in the tool, and the pointer's access behavior is updated by the Pointer Behavior Analysis stage. Once the entire file has been parsed, access strides are calculated for all pointers found in the file. The tool then dumps the complete memory access pattern information to a file for further analysis in the second part of the approach.

3.1.2 Pointer Behavior Analysis

The purpose of the pointer behavior analysis phase, shown in Fig. 3.3 is to classify the static predictability of a memory access, and determine the stride of accesses without runtime dependencies. Runtime-dependent accesses are characterized by array indices that are dependent on input data in some form. This includes, but is not limited to, cases such as determining an array index from a value stored in another array, or traversing a graph from a user-defined starting point. From

the perspective of the analysis tool, it is simply looking through the critical path leading up to the memory access for any computation that is dependent on a variable that is declared outside of the function's scope. Memory accesses without external dependencies are considered predictable, even if they are addressed by another variable that is itself predictable.

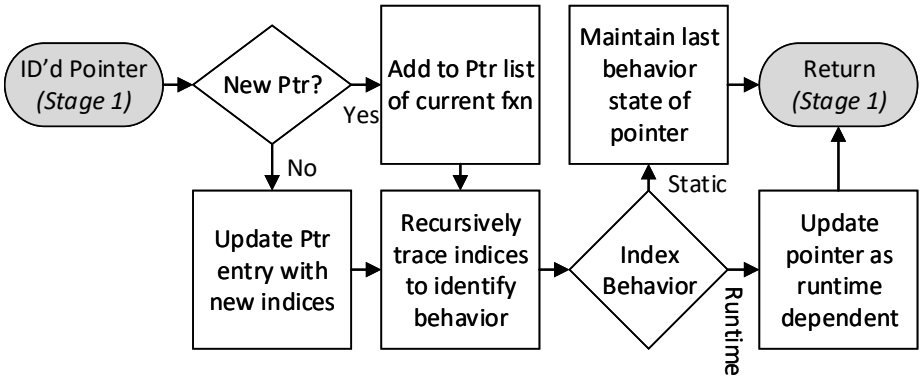


Figure 3.3: Pointer Behavioral Analysis (Stage 2)

A benefit of the LLVM IR is that it maintains application control and dataflow characteristics of high-level programming languages, while providing a low-level abstraction closer to a machine's instruction set architecture (ISA). The explicit data and control flow present in the IR allows for a quick recursive trace of the critical path of a memory access.

When all accesses to particular pointer in a file have been identified, the tool is then able to determine the variable's access stride via the process shown in Fig. 3.4.

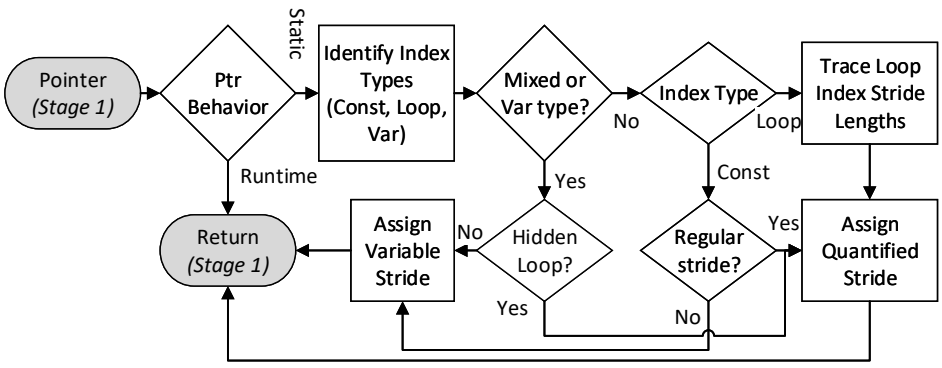


Figure 3.4: Stride Measurement (Stage 3)

Stride measurement is skipped entirely for pointers with runtime-dependent characteristics. For all other pointers, the tool further classifies the pointer indexes as *constant*, *loop*, or *variable*. Constant indexes are characterized by explicit integer offsets or accesses to the base pointer, which

corresponds to the C expression array[0] or *array. Loop indices correspond to incrementing/decrementing counters commonly used in loops. Indices which are the result of some other computation are considered variable. When nested loops are found in a program, there may be instances in which a loop index takes on the form of a variable index. An example of this would be when a multidimensional array is indexed like a 1D array (ex: A[i*rowlength+j]). To capture these cases the classifier performs a trace back on the array index instruction to search for such expressions.

After determining the index type for each access on a pointer, the stride of the accesses can then be computed. The stride of a loop indexed variable can be determined by examining the corresponding increment/decrement operation. For constant indices the tool calculates the stride between successive accesses. If the stride is constant, or within a limited range, the average stride is assigned and the variable is marked as strided. In the case of irregular strides the pointer is marked as variable an extrema are noted. Stride calculations are not performed for pointers with mixed or variable type indices, but extrema are noted if available.

3.1.3 Memory Access Pattern (MAP) File

The Memory Access Pattern (MAP) file generated by the tool provides designers with an overview of memory access in an application, which can be used to guide designers in memory hierarchy design. As a secondary benefit, it provides a quick point of reference to guide the designer through the original LLVM IR file if they want to investigate memory access further by hand.

```
Function: TYPE @FUNC_NAME
Parameters: (PARAMS)
        Variable: %PTR_NAME   STRIDE_LENGTH
        Type: PTR_TYPE
        Accesses:
                %REG_OP -> INDEX_LIST
        Variable: %PTR_NAME2  STRIDE_LENGTH
        Type: PTR_TYPE
        Accesses:
                %REG_OP -> INDEX_LIST
                %REG_OP -> INDEX_LIST
```

Figure 3.5: Memory Access Pattern-file (*MAP-file*)

The structure of the MAP file is shown in Fig. 3.5. In addition to the pointer-specific parameters described above, the MAP file also includes information about calling functions, including parameter lists. This can be used to understand the linkage of variables that might be passed between function

calls, to better understand application-level characteristics. For each pointer that is considered global within a function's scope, its type, corresponding IR instruction, and indices are listed. The stride for the pointer is listed if it could be calculated. Otherwise the classification of the stride is provided.

3.2 Variable-Level Memory Assignment

After the the memory access analysis tool has generated the application's memory access pattern file, the approach proceeds to the assignment and allocation of memory. Variables assigned to scratchpad memories or tiled scratchpad memories are assigned to their own unique memory elements. Due to limitations in evaluation tools, cached variables are all assigned to the same cache. Prior to memory assignment, however, the spatial and temporal locality of each variable must be determined.

3.2.1 Locality Measurement

The locality measurement phase of the approach is responsible for determining the inherent static locality of each variable based on the access patterns found in the MAP file. The spatial locality of a variable is determined by the length of its stride. The formulation for spatial locality formulated by Weinburg et al. [72] is shown in Eq. 3.1. For the purposes of this calculation, the absolute value of the stride is used for any negative-valued strides. In the case that the stride of a variable could not be determined, manual verification of stride is employed.

$$L_{spatial} = \sum_{stride=1}^{\infty} \frac{P(stride)}{stride} \tag{3.1}$$

The temporal locality of a variable is determined by Eq. 3.2, based on the work in Weinburg et al. [72]. Since the Weinburg et al. locality analysis relies on runtime values that are not available in this approach, the distance metric is instead replaced by counting the number of memory accesses to particular variable before a non-loop index is repeated.

$$L_{temporal} = \sum_{i=0}^{log_2(N)} \frac{(dist_{2^{(i+1)}} - dist_{2^i}) * (log_2(N) - i)}{log_2(N)} \tag{3.2}$$

The overall locality of a variable is finally determined by a weighted sum of spatial and temporal locality, as shown in Eq. 3.3. For this formulation spatial locality is given a higher weight than temporal locality due to its greater significance in accelerated applications.

$$L_{total} = \Big(w_s * L_{spatial} + w_t * L_{temporal}\Big) * 100 \tag{3.3}$$

Tiled scratchpad memories are designed to exploit spatial locality, but are limited by temporal locality that may require re-fetching of older data tiles. Caches benefit significantly from both spatial and temporal locality due to the corresponding reduction in cache misses. Traditional scratchpads do not receive any benefit from data locality due to the fact that the entirety of the application data needs to be preloaded into the scratchpad prior to running the accelerator.

3.2.2 Memory Assignment

Final memory assignment and allocation can proceed once the locality of each variable has been calculated. In order to determine the optimal assignment of a variable, both its locality and its overall size (job size) must be taken into account. The final determination of memory assignment is based on the graph presented in Fig. 3.6.

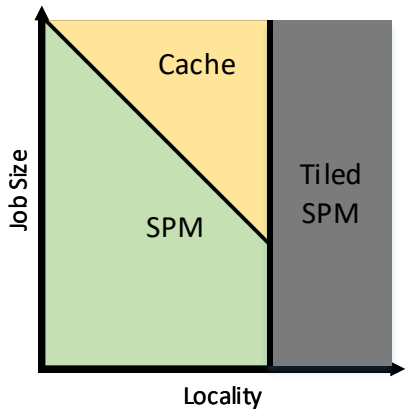


Figure 3.6: Memory Allocation Graph

The design space explorations on common accelerator workloads in [7] showed that there was a preference in some applications for caching over scratchpad allocation when examining power and timing Paretto curves. Applications where variables contained strong spatial locality mapped best to tiled scratchpad memories that employed double-buffering techniques. At the same time, applications that worked on larger datasets with less spatial locality mapped better to cache hierarchies. This is because caches were able to exploit the limited spatial and temporal locality present in the data to operate efficiently with a much smaller memory footprint than traditional SPMs. As the locality of the data improves, so does the cache efficiency. Meanwhile the efficiency of a SPM is static, due to the requirement that it contains the entirety of the data needed to perform the accelerated task or job. A secondary effect of this limitation of SPMs, is that an SPM-based memory hierarchy might require the movement of more data than a cache-based systems for large, but sparesly accesses, data structures. As a result we see the trend shown in Fig. 3.6, in which the decision boundary between cache and SPM designs is determined by the combination of job size and locality. Then, once spatial locality is high enough, a tiled SPM will always be more efficient than caches or traditional SPMs.

3.3 Evaluation

Design space explorations on benchmarks from MachSuite [31] were performed in order to evaluate the quality of the locality-aware memory assignment and tiling approach. The gem5-Aladdin simulator, which supports modeling of both cache and scratchpad memories for hardware accelerators, provided a test bed for evaluating accelerators with both standard memory configurations and optimized hybrid memories.

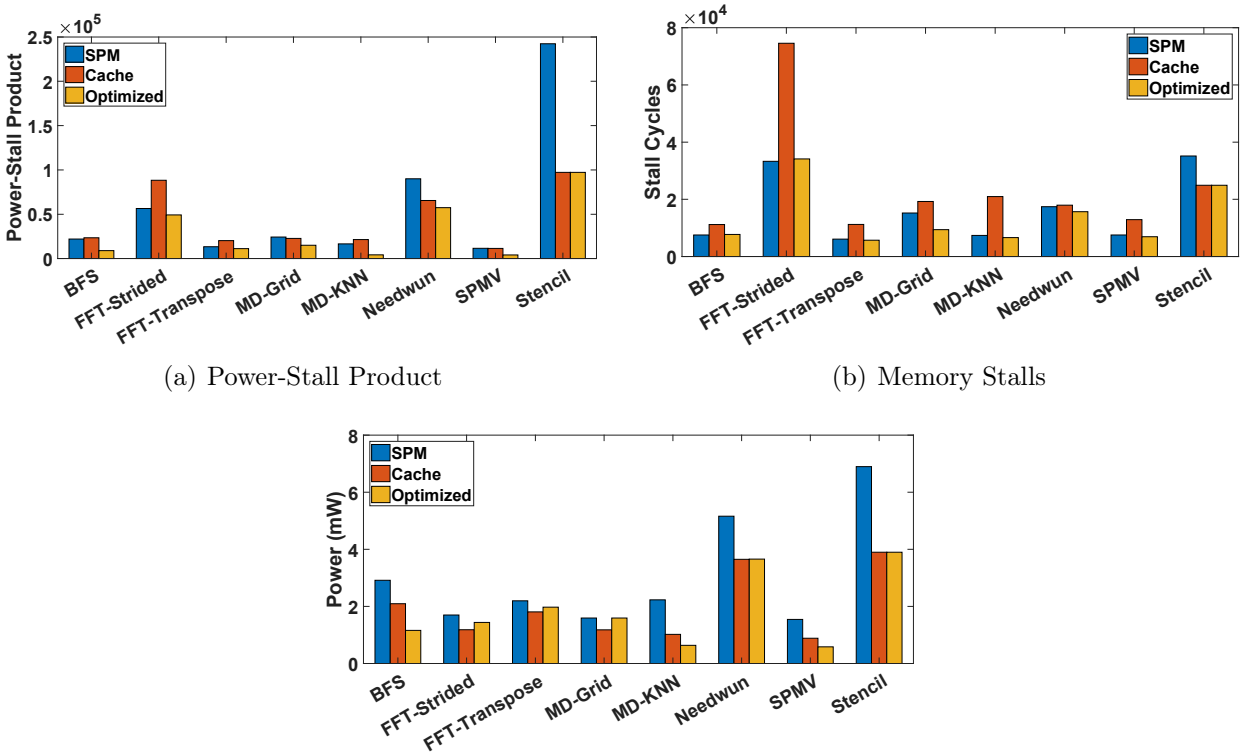
The quality of a memory assignment was assessed by multiplying the total power consumed by an accelerator's memory by the amount of time that the accelerator's datapath was stalled waiting for memory. Stall times and power metrics included the data transfer overheads associated with DMA operations. This joint power-stall product represents the wasted energy of a stalled hardware accelerator which the locality-aware memory assignment approach seeks to minimize.

For explorations of monolithic SPM designs, the SPM was sized to support all data needed to fulfill the accelerator task. Meanwhile cache-based designs were swept across a variety of cache sizes

to find the optimal size for a cache-only approach. For the approach-optimized hybrid designs tiled SPMs were sized to the minimum tile sizes needed and any caches were swept for the optimal cache size. Any remaining data was assigned to a SPM that was large enough to fit the entirety of that data.

3.3.1 Performance and Power Comparison

A examination of the eight benchmarks that were tested from MachSuite found one benchmark in which all could be tiled (MD-Grid), two benchmarks in which the data supported limited tiling (SPMV-CRS and MD-KNN), and five benchmarks without enough locality to support data tiling (BFS, FFT-Strided, FFT-Transpose, Needwun, 3D Stencil). The BFS and Needwun applications featured large amounts of runtime and data-dependent array accesses, while the stride of the FFT applications prevented breaking up data into tiles. Normally a stencil application should support tiling over it's input data structure, however the implementation found in MachSuite destroyed data locality with its indexing scheme.



(a) Power-Stall Product

(b) Memory Stalls

(c) Memory Power

Figure 3.7: Performance and power comparison

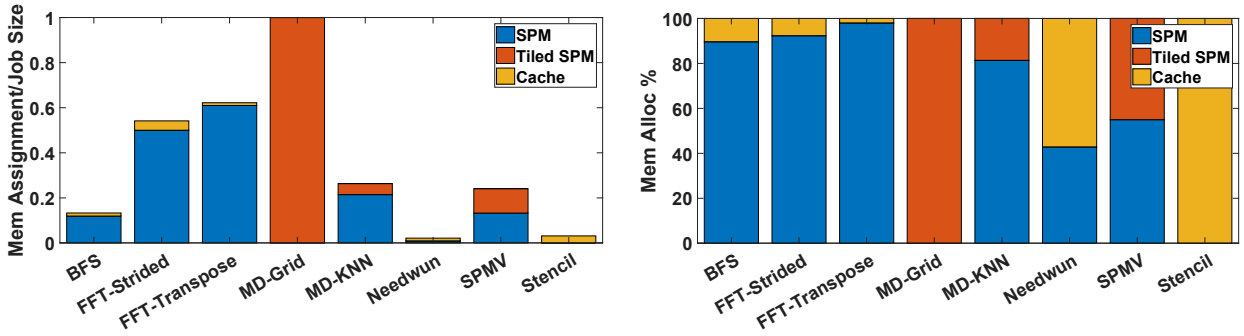
In terms of raw performance Fig. 3.7(b) shows that monolithic scratchpad memories outperformed monolithic cache approaches for all applications. This is not surprising due to the capacity for the applications tested due to a lack of sparsity in data access, and the capacity of DMAs to amortize memory access costs. One exception to this trend was the 3D Stencil application, which showed a strong preference towards caches due to its large input data size can high degree of temporal locality. While SPM approaches had an edge in raw performance, the cache-based approaches had a significant advantage in power efficiency due to the comparatively tiny size requirements for caches to capture the working data set of each application. By considering both delays and power efficiency via the power-stall product Fig. 3.7(a) show that the overall efficiency of cache was worse than monolithic SPM approaches for two applications, nearly the same for two applications, and even an improvement on two applications.

Optimized memory hierarchies recommended by the locality-aware approach demonstrated significant improvements over the scratchpad baseline, with an average improvement of 45% across the eight benchmarks. Some of the most dramatic improvements were seen in MD-Grid, MD-KNN, and SPMV-CRS where the tiling and double-buffering of data was able to dramatically memory power and datapath stalls. The 3D stencil application was also noteworthy in that the locality-aware recommendation was to map all data to cache.

3.3.2 Memory Allocation

The primary factor for the efficiency of the locality-aware memory assignments was the reduction in on-chip memory requirements and the corresponding power reduction. Fig. 3.8(a) shows the total amount of memory allocated to each application with respect to that application's job size. With the exception of MD-Grid, where the data was simply split into two tiles to facilitate buffered load/store, every application saw a significant reduction in on-chip memory footprint. The Needwun and Stencil applications in particular were able to leverage caches to reduce their on-chip memory footprints by over 98%. These two applications were characterized by having the largest input data of all tested benchmarks, and lacked the capacity to leverage data tiling schemes to reduce the footprint of SPM memories. Fig. 3.8(b) shows the logical memory assignment of each memory type vs. the rest. For most benchmarks that leveraged a cache alongside SPMs, the total cache

allocation accounted for only a small portion of the total memory allocation despite handling 50% or more of the data within the application.



(a) Physical memory assignment as a percent of job size

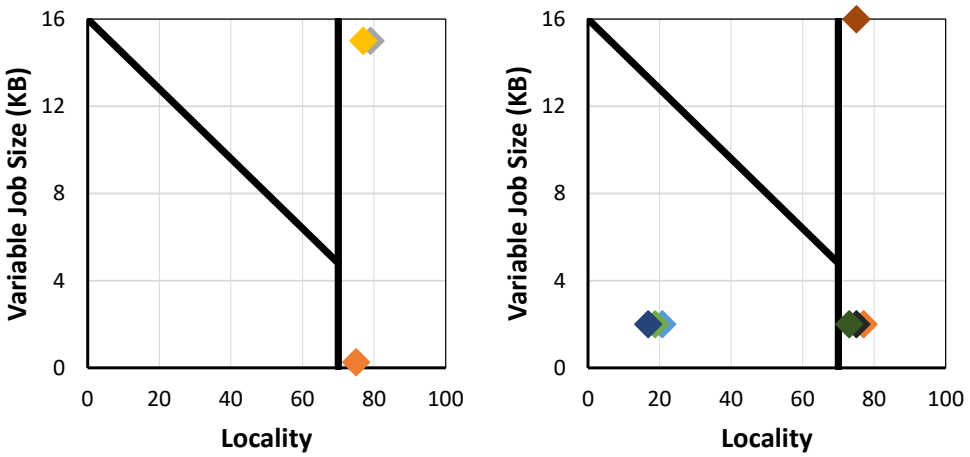
(b) Logical memory assignment per job size

Figure 3.8: Memory type distribution for optimized memory assignment

3.3.3 Algorithm Effects on Optimal Memory Allocation

An interesting outcome of the locality-aware approach is the ability to observe the impacts that algorithmic changes can have on the memory design of the same basic application. The MD-Grid and MD-KNN applications both implement the Molecular Dynamics application in which the net force on a molecule within a system of molecules is calculated based on its distance to nearby molecules. While both applications work on the same input data, and perform the same basic operation, the organization and means of accessing input and output data radically differs. MD-Grid stores position and force data for each molecule in a 4D tensor that requires nested loops to access. In contrast MD-KNN distributes position and force data across 1D arrays and uses a K-Nearest Neighbors method for tracking neighbors for each molecules in a separate array. The memory assignment graphs for each application are shown in Fig. 3.9.

While all variables in MD-Grid support tiling, the densely packed nature of the data and high temporal locality requires large tile sizes to function correctly. This limits the overall efficiency of memory optimization without significant algorithmic reworks. Meanwhile the variables in MD-KNN that support tiling can be tiled efficiently, reducing on-chip memory allocation by 75%. These observations demonstrate the importance on algorithmic optimization for memory optimization and how the locality-aware memory assignment and tiling approach can help guide better hardware-software co-design.



(a) MD-Grid Access Graph

(b) MD-KNN Access Graph

Figure 3.9: Memory Access Graphs for Alternate MD Realizations

CHAPTER 4: LLVM-BASED MEMORY DECOUPLING FOR OPENCL ACCELERATION ON FPGAS

The lessons and techniques learned in the development of the locality-aware memory assignment and tiling approach have also been applied in the optimization of OpenCL-based accelerators on FPGAs. By leveraging analysis of OpenCL kernels at the LLVM IR abstraction, the prior work can be modified to automatically detect FPGA memory optimizations such as memory decoupling. Additionally, the common usage of the LLVM tool chain in existing HLS tools for FPGA synthesis means that this work could potentially be fully integrated into existing HLS tools. This work was published as part of Elsevier Microprocessors & Microsystems, Volume 72, Issue C in Feb 2020 [13].

4.1 The FPGA Memory Wall and Optimization

One of the primary limitations of hardware acceleration on FPGAs is the cost of accessing off-chip memory. Without the advanced caching mechanisms of CPUs, or the cycle-level thread scheduling capabilities of GPUs, FPGAs lack the means to hide or mitigate the costs of off-chip memory access. This in turn limits the effectiveness of deep application-specific datapaths that FPGAs exploit for performance and power efficiency by introducing long delays on every data access. OpenCL HLS tools like those provided by Xilinx[73] and Altera[74] attempt to mitigate the costs of memory access by introducing delay buffers, but these optimizations are limited by memory bandwidth and other devices that share the same off-chip memory.

One solution to mitigate off-chip memory access costs in OpenCL is to move data to *local memory*. In the GPU context, local memory refers to a dedicated scratchpad memory within the GPU architecture that can guarantee fast memory access for the GPU's compute units. In the FPGA context declarations of local memories tell the HLS tools to allocate BRAM blocks within the FPGA to generate small scratchpad memories for the custom datapath. Data marked as local is managed at the OpenCL work group abstraction and is copied between off-chip memory and on-chip scratchpads at the beginning and end of work group execution. This bulk transfer of data amortizes

the costs of the off-chip memory access and leads to noticeable speedups in kernel execution times. This approach has scalability limitations however. Limitations in the availability of on-chip memory limits the maximum size of work groups that rely on data stored in local memory. At the same time, the data movement at the start and end of every work group leads to large delays between work group executions that leave the custom datapaths under-utilized.

Rather than relying on the relatively naïve implementation of local memory provided by OpenCL HLS tools, some groups have leveraged dedicated data-access kernels and manually allocated on-chip memories to guide the HLS tools to implement proper double-buffering and tiling techniques [75, 76]. By moving the data movement between on and off-chip memory from the main compute kernel to dedicated read and write kernels, off-chip memory access can be overlapped with computation. This is because OpenCL HLS tools support the execution of multiple kernels concurrently on FPGAs. Manual data tiling approaches offer significant performance improvements over the local memory implementations provided by OpenCL HLS tools, but face their own set of limitations. As discussed in Ch. 3, runtime-data dependencies and low levels of spatial locality limit the kinds of data that can be tiled. Even when data can be tiled, some applications like the MD-Grid example in Ch. 3 cannot be tiled efficiently without significant algorithmic modifications. In general tiling approaches require a high degree of manual effort and algorithm-specific tuning within the compute kernel, making the barrier to entry far higher than simply leveraging local memory.

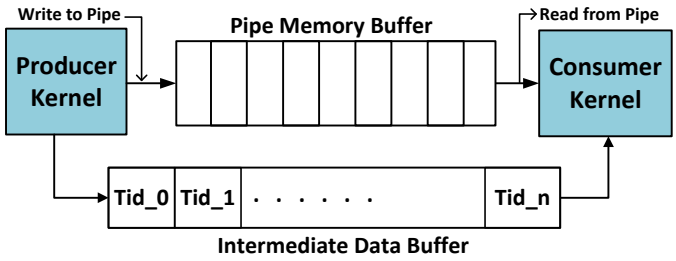


Figure 4.1: OpenCL Pipe semantic

The OpenCL API provides another alternative for memory optimization in the form of pipes, shown in Fig. 4.1. In OpenCL a pipe is a thread-addressable memory buffer that enables the movement of data between different kernels. This enables the generation of memory buffers that can synchronize the execution of producer and consumer kernels. The Altera OpenCL HLS tool chain[74] implements a version of the OpenCL pipe construct that synthesizes on-chip memory buffers called

channels. The thread-level indexing of these channels enables multi-kernel decoupling of memory access from computation at a finer granularity than tiling methods. This is advantageous because channel-based decoupling is more flexible than tiling, and does not require significant algorithmic modification to implement. The downside is that the granularity of data movement in channel-based decoupling schemes is not able to amortize memory access costs to the same degree as bulk transfers used in tiling schemes.

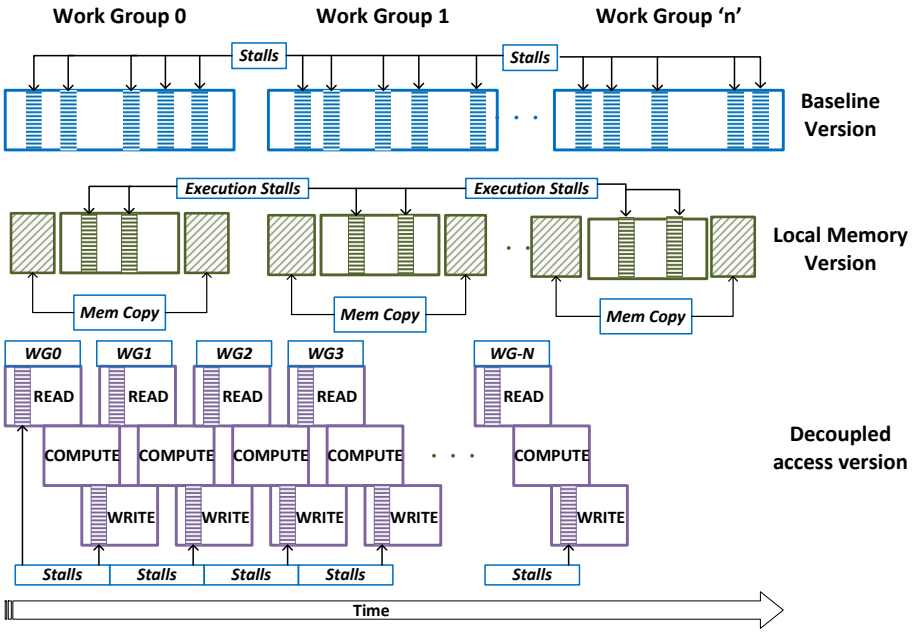


Figure 4.2: Execution pattern comparison

Fig. 4.2 shows a comparison between the baseline OpenCL execution semantic, local memory implementations, and decoupled memory implementations. While the local memory implementation is able to leverage bulk data transfer to remove pipeline stalls and improve performance for a single work group, the benefits are more limited as the number of work groups increases. In comparison, decoupling methods are able to overlap memory operations with computation, hiding stalls in the overall pipeline and improving execution times. Both tiling and channel-based implementations provide these benefits, but for the purposes of automation within a HLS tool chain, the channel-based approach is far simpler to implement.

4.2 Automated LLVM-Based Memory Decoupling

In order to automate the decoupling of memory access within a computation kernel, it is important to identify which data can be decoupled. Fortunately the LLVM abstraction used in Ch. 3 can also be employed in the HLS context since it is already employed by HLS tools to translate the abstraction in high-level programming languages to hardware description languages and RTL. With some modification to the approach used in Ch. 3 it can be re-purposed to identify memory accesses suitable for decoupling from the main computation kernel.

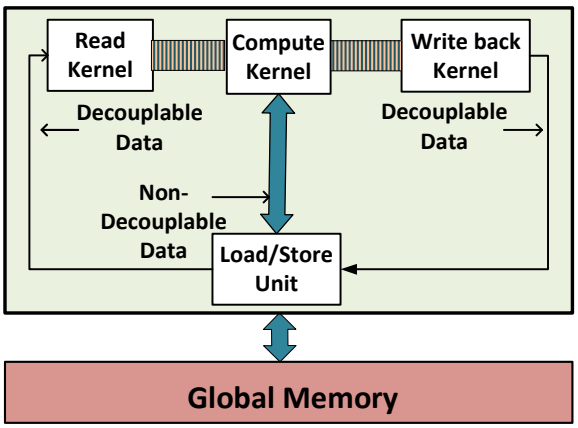


Figure 4.3: Memory decoupling with pipes

In order to safely decouple a variable the access of a variable from computation, it is important to ensure that decoupling the variable doesn't cause a data consistency issue. This restriction between computation kernels is already imposed by the OpenCL programming semantic, but it is an important consideration when working in other contexts. Additionally a variable that is decoupled should not be accessed using a loop index within the OpenCL kernel. This is a hardware limitation since it would require transferring large portions of a data structure through the resource constrained channels. Based on the guidelines produced by Xilinx[73] and Altera[74], such looping structures should be avoided anyway do to their negative impact on pipeline generation, but they may still be necessary in some designs. Data that does not meet either of these conditions can be safely decoupled and is moved to separate read and write kernels as shown in Fig. 4.3.

Algorithm 1 LLVM Analyzer

```
1: function MAIN()
2:     ptrList ← PARSE_POINTERS(INPUT_LLVM_FILE)
3:     cdfg ← PARSE_CDFG(INPUT_LLVM_FILE)
4:     for each ptr ∈ ptrList do
5:         PTR_EVAL(ptr, ptrList, cdfg)
6:     end for
7: end function

8: function PTR_EVAL(PTR, PTR_LIST, CDFG)
9:     for each node ∈ cdfg do
10:        if node.type = GEP and node.ptr = ptr then
11:            ptr.separable ← IS_SEPARABLE(node, CDFG, PTR_LIST)
12:        end if
13:    end for
14: end function

15: function IS_SEPARABLE(NODE, CDFG, PTR_LIST)
16:    bool separable
17:    if NODE.op = THREAD_ID_CALL then
18:        return TRUE
19:    else
20:        for each dep ∈ NODE.deps do
21:            if dep.isINDVAR() then
22:                separable ← FALSE
23:            else if dep ∈ PTR_LIST then
24:                separable ← PTR_EVAL(dep, PTR_LIST, CDFG)
25:            else
26:                separable ← IS_SEPARABLE(dep, CDFG, PTR_LIST)
27:            end if
28:        end for
29:    end if
30:    return separable
31: end function
```

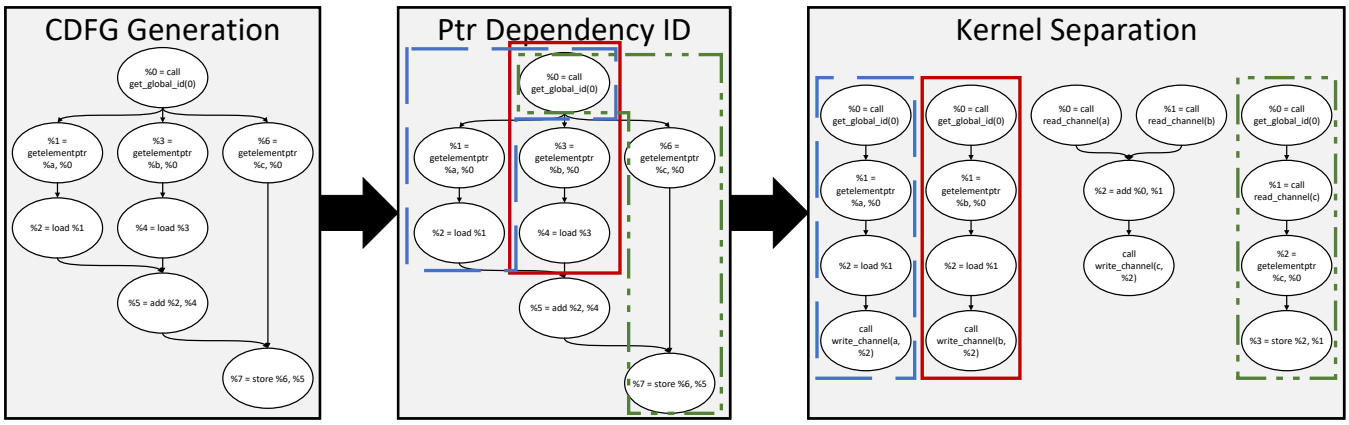


Figure 4.4: Automated Kernel Memory Access Decomposition

4.2.1 Algorithmic Implementation

Algorithm 1 shows the process for identifying decouplable variables in an automated fashion. During synthesis the HLS tool will process the OpenCL source using a modified version of the clang compiler and generate the LLVM IR representation of the application. From here a list of all pointers and the application CDFG can be extracted from the IR (lines 2 and 3). The analyzer will then iterate over the pointer list to identify the memory accesses that can be decoupled from the main computation kernel (lines 4-6). For each memory access the analyzer calls the PTR_EVAL function, which searches for the corresponding address offset calculation, described in LLVM by a *getelementptr* or GEP instruction. This GEP instruction is traced by the IS_SEPARABLE function to identify the critical instruction path for the memory access. In this search the analyzer is looking for induction variables, which LLVM IR's means of tracking and optimizing loop index variables. If an induction variable is found in the critical path, the access is marked as non-separable. If another pointer appears in the critical path for a memory access, then the analyzer recursively traces that access to determine its separability. A variable indexed by another variable remains separable as long as the indexing variable is separable. Since HLS imposes additional function inlining rules, there is no need to track variable separability across function bounds.

Fig. 4.4 shows an example of the entire process for a simple vector addition example. From the initial CDFG the critical paths for each memory access are identified and outlined by a colored box. In this example each memory access is indeed separable and can be separated from the original compute kernel. When separating the accesses to separate kernels, operations from the critical path

of each memory access are copied to the new kernels. The memory accesses in the compute kernel are replaced with channel read/write operations, and any instruction nodes that no longer have users are trimmed from the computation kernel. The new read kernels will write the value read from memory to a channel that is shared with the computation kernel. Likewise the write channels will read from a shared channel and write back to memory. This entire process could in theory be used to continue decomposing the read and write kernels if they also contain a mix of computation and memory access, but for now this optimization is ignored.

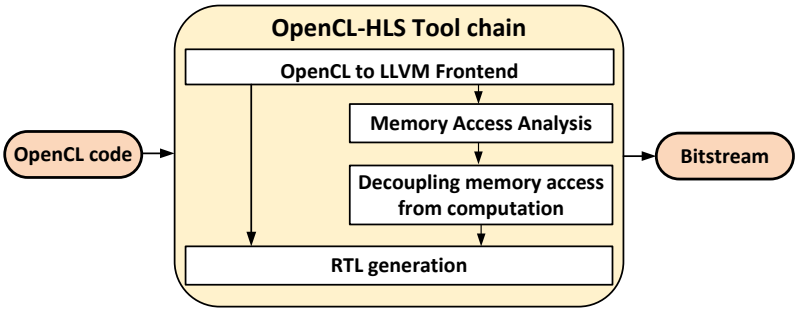


Figure 4.5: Conceptual realization of the decoupled access incorporated into OpenCL HLS

Without access to an open source HLS tool chain for OpenCL it is not possible to integrate the analyzer and CDFG modification directly at this time. Instead for the purposes of evaluation the analyzer was used to generate suggestions that were manually implemented in the accelerator source code. Fig. 4.5 however shows how the LLVM analysis could be integrated into existing HLS tool chains. Since most of the features could be implemented in the LLVM IR optimization passes and libraries, the code changes in the HLS tool itself would be minimal.

4.3 Evaluation

Evaluation of the LLVM analyzer consisted of modifying benchmark source code to guide HLS optimization based on the outputs of the LLVM analyzer. The modified benchmarks were then synthesized using the Altera OpenCL HLS tools and run on an Altera Stratix V FPGA. Timing, memory bandwidth, resource utilization, and power consumption were tracked versus the unmodified benchmarks and local memory-based implementations. The system used for synthesis and evaluation is described in Table 4.1.

Eight benchmark OpenCL applications were chosen from the Rodinia benchmark suite [37] that

Table 4.1: System characteristics used for study.

Host	Intel(R) Core(TM) i7-7700K
Host clock	4.2 GHz
FPGA Family	Stratix V
FPGA Device	5SGXMA7H2FE35C2
CLBs	234,720
Registers	939K
Block Memory bits	52,428,800
DSP Blocks	256

represent an assortment of data intensive applications including, but not limited to, structured grids, graph traversal, and linear algebra. These applications are listed in Table 4.2 along with a breakdown of the proportion of their off-chip memory accesses that could be decoupled from computation using channels.

Table 4.2: Kernel global variable data information per application

Benchmarks	Number of Threads	Decouplable variables		Non-Decouplable variables		Decouplable(%)
		Size per Thread(Bytes)	Total Size(Bytes)	Size per Thread(Bytes)	Total Size(Bytes)	
B+ Tree FindK	65536	12	786432	8	524288	60
B+ Tree RangeK	65536	20	1310720	0	0	100
Gaussian	65536	12	786432	0	0	100
HotSpot	16384	12	196608	0	0	100
BFS	1048576	8	8388608	12	12582912	40
NN	42764	8	342112	0	0	100
Srad Extract	65536	4	262144	0	0	100
LUD Diagonal	4096	4	16384	0	0	100

4.3.1 Performance Evaluation

Baseline values for resource utilization, execution time, off-chip memory bandwidth, datapath stalls, and power consumption were recorded for each benchmark. These values were recorded using Altera runtime profiling tools can be seen in Table 4.3. While the stalls within a datapath can be attributed to a variety of sources, the primary cause of datapath stalls is off-chip memory access. By examining the stall rates in conjunction with the decoupling analysis provided by the LLVM-based analysis, we can identify that applications that might receive the largest benefit from decoupling memory access with channels.

Table 4.3: Baseline profiling information for each application

Benchmarks	Resource utilization(%)			Execution Time(ms)	Bandwidth(MBps)	Stalls(%)	Clock Frequency(MHz)	Power(Watt)
	Logic utilization	Memory blocks	Registers					
B+ Tree FindK	23	24	11	25.79	3266	19.7	223.56	2.7
B+ Tree RangeK	25	24	12	17.39	5247	3.01	218.6	2.2
Gaussian	21	20	9	6.56	3265.5	8.6	228.2	2.1
HotSpot	23	16	4	0.52	1181.1	76.8	233.5	1.86
BFS	22	17	6	2.34	957.4	40.7	288.4	2.3
NN	20	19	8	0.28	3002.7	0.0	245.2	2.17
Srad Extract	20	16	8	1.23	1862	0.0	250	2.15
LUD Diagonal	23	20	10	0.11	140.4	0.23	234.5	2.27

Fig. 4.6 shows the relative speedup over the baseline for the local memory and decoupled memory implementations of each benchmark. The HotSpot application saw the largest improvement and execution time with more than a 4.6x speedup. This performance boost can be attributed to the high capacity for memory decoupling along with its baseline's incredibly high 76.8% datapath stall rate. As the most memory-bound application of the set, it had the most room for improvement of the eight benchmarks tested. It is also worth noting that the performance benefit of using local memory for this application was marginal since the off-chip memory transfers were simply moved outside of the computation kernel, rather than decoupled and overlapped with it. Other high stall applications like B+ Tree FindK and BFS also saw performance improvements, but these benefits were mitigated by the fact that many of their data accesses could not be decoupled from computation. The applications that were computation-bound at the baseline did not receive any noticeable benefit from decoupling. It should be noted that the speedups from using local memory were minimal or nonexistent for many applications due to the fact that the off-chip memory access was simply moved rather than overlapped with computation. In the case of B+ Tree RangeK, there was even a performance decrease due to high decreases of sparsity in the data access. This meant that the local memory optimization actually copied more data from off-chip memory than what was needed by the application.

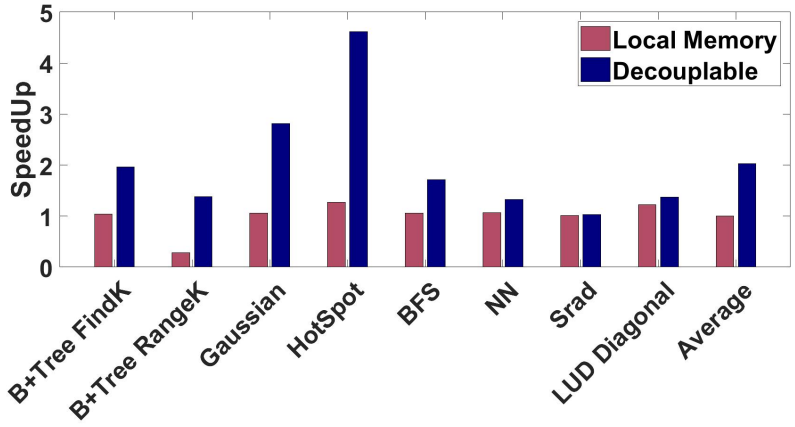


Figure 4.6: Performance improvement over the baseline

Fig. 4.7 and Fig. 4.8 help to paint a clearer picture of where the speedups, or lack thereof, arose. As expected of memory bound applications, both the local memory and memory decoupling optimizations led to significant reductions in the number of datapath stalls. For computation-bound

applications, however, memory decoupling actually added a small increase in datapath stalls within the compute kernel. These stalls can be attributed to waits for the channel buffers to be filled or emptied by the read and write kernels. Since instructions that lead up to the initial memory access were removed from the compute kernel, the modified compute kernel begins with reading the input channel. This input channel takes some small amount of time to fill since the corresponding read kernel is also having to do some small amount of computation. This is why stalls within the compute path only tell half the story, and why it is also important to look at off-chip memory bandwidth.

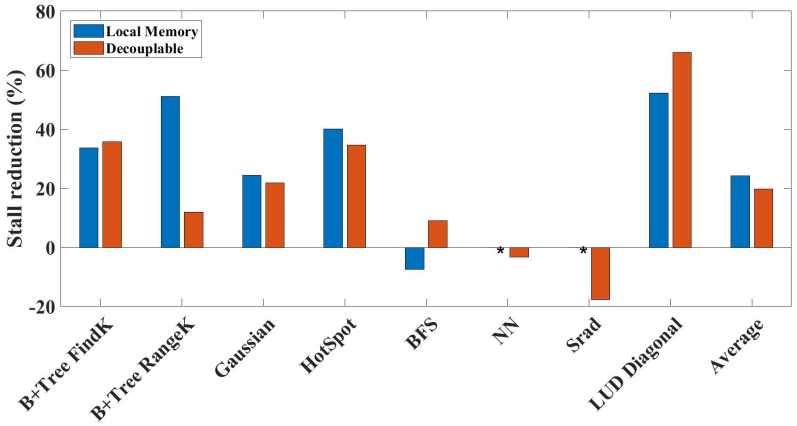


Figure 4.7: Memory stalls reduction over the baseline

The memory bandwidth improvements for shown in Fig. 4.8 explain why the applications with increased stall rates see net improvements in execution times. Every benchmark saw notable improvements in memory bandwidth utilization with memory decoupling.

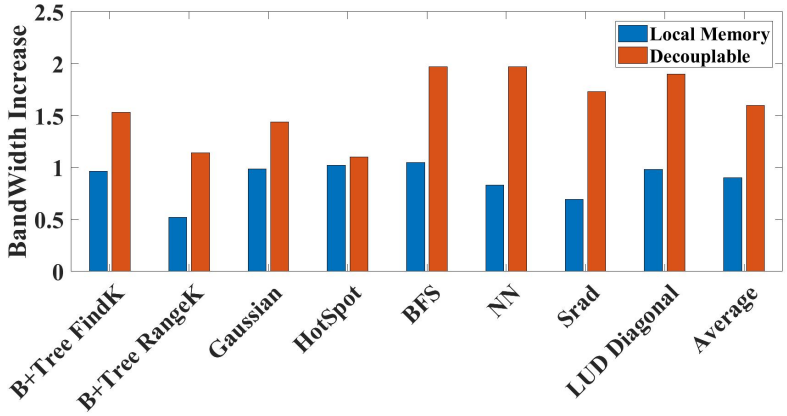


Figure 4.8: Memory bandwidth improvement over the baseline

4.3.2 Resource Utilization Overheads

Memory decoupling using channels does come at a small cost in terms of resource overheads. The addition of the memory channels requires some small amount of memory blocks to construct the buffer. Additionally the thread-addressing of data within the buffers requires some small amount of logic and register blocks on the FPGA. The resource overheads for decoupled memory versions of the benchmarks are shown in Fig. 4.9. Asterisks in this case are used to represent resources for which there was no overhead. In general the resource overhead for each application was quite low, with variation depending on buffer sizes and complexity.

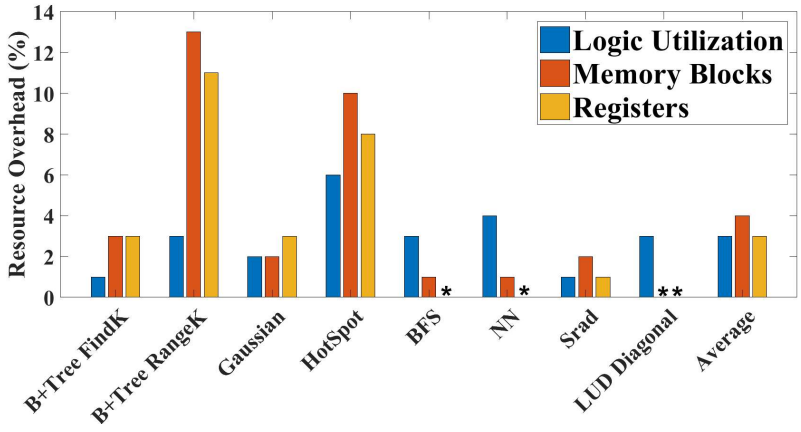
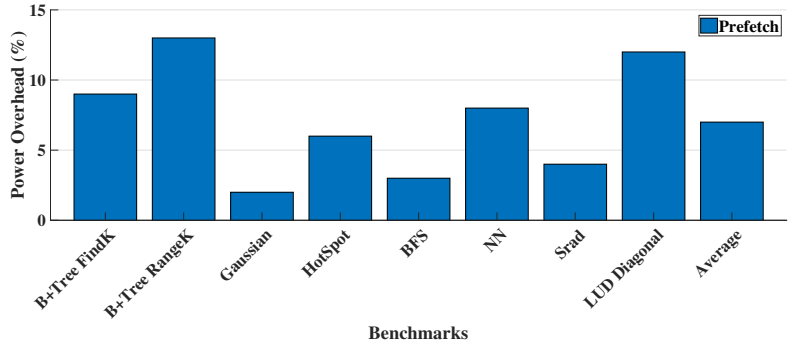


Figure 4.9: Decoupling resource utilization overhead over the baseline

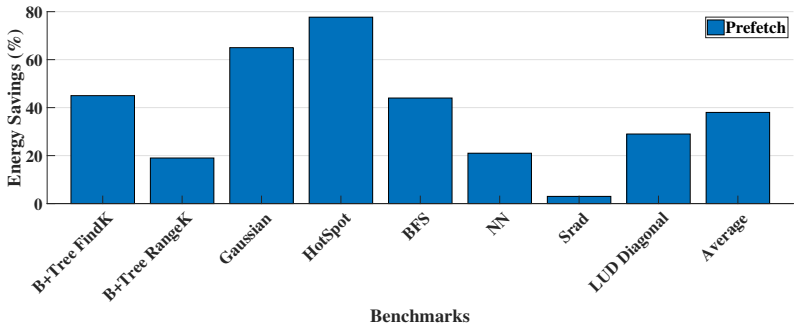
4.3.3 Power Efficiency

Power efficiency, often referred to as performance per Watt, is often an important target in the design of hardware accelerators. As with any other hardware optimization that involves the allocation of additional resources, decoupling memory access with channels does incur a small power penalty shown in Fig. 4.10(a). On average the additional resources resulted in a 7% increase in power consumption over the baseline implementation. The B+ Tree RangeK application saw the largest increase due to the size and complexity of its channel buffers. Despite power increases for all benchmark applications, there is actually a net savings in energy consumption compared to the baseline. These savings can be attributed to the improvements in performance across all benchmarks, with an average energy savings of nearly 40%. The largest savings can be seen once again in the HotSpot application, owing to its enormous improvement in execution times for very little

resource overhead. Likewise the Srad application, which saw very little performance improvement, saw very little energy savings.



(a) Power



(b) Energy

Figure 4.10: Power overhead and energy saving over the baseline

Knowing the power demands for each application it is possible to determine a performance per Watt metrics. Performance can be gauged by averaging the amount of computation performed over the execution time of the benchmark. Using Intel's PIN Pytracer tool [77], the executed instructions for each kernel were extracted. By subtracting memory and control instructions from the PIN statistics it was possible to identify the amount of computation present in the application. By dividing the amount of computation by the execution time and power consumption it is possible to compute a quantifiable value for performance per Watt. Fig. 4.11 shows benchmark performance per Watt values normalized to the decoupled memory implementation.

With few exceptions the local memory optimization generally mirrors the efficiency of the baseline. In all but one case it shows less improvement the decoupling memory with channels. In the case of the LUD application, the channel implementation resulted in one of the largest power overheads of all tested benchmarks, with minimal performance improvement over local memory. This resulted in

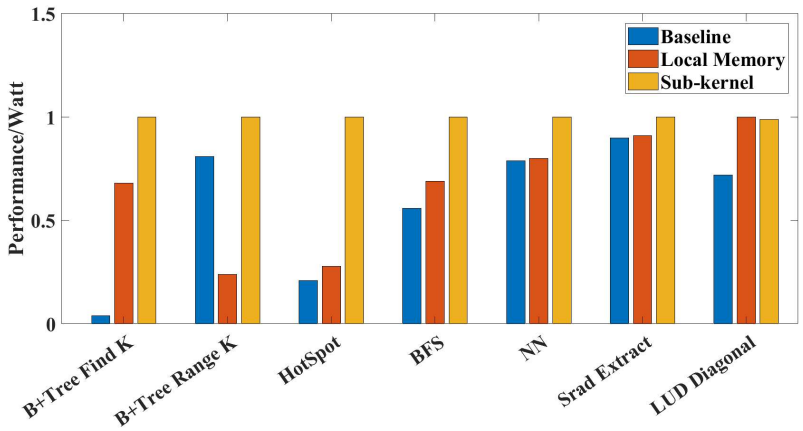


Figure 4.11: Normalized Performance/Watt

a small net efficiency versus local memory. In the case of B+ Tree Range K, the over-access of off chip memory due to the local memory allocation of sparsely accessed data resulted in a significant efficiency loss for local memory versus both the baseline and decoupled implementations.

CHAPTER 5: GEM5-SALAM

The second contribution of this work is the development of a design space exploration tool for application and domain-specific accelerators called gem5-SALAM. gem5-SALAM is an extension to the popular, open-source system architecture simulator gem5. SALAM incorporates cycle-accurate simulation models of hardware accelerators based on LLVM IR into the full system simulations provided by gem5, as shown in Fig. 5.1. gem5-SALAM was first conceptually introduced in IEEE Computer Architecture Letters 2019[14], and expanded upon with a publication in IEEE/ACM MICRO 2020[15].

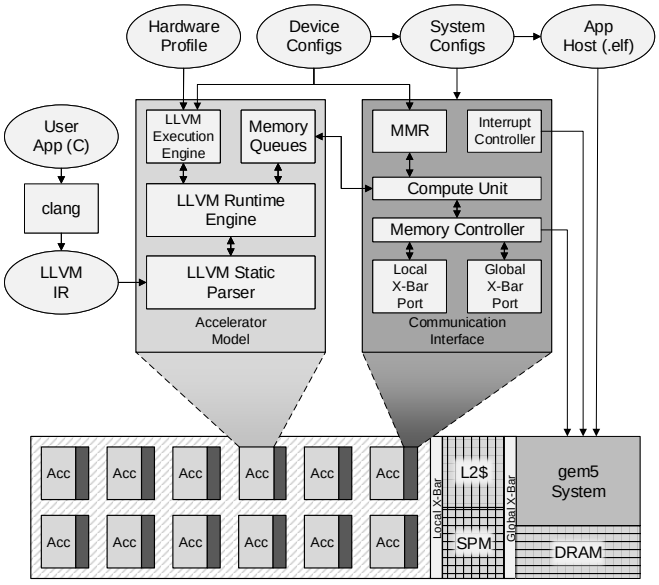


Figure 5.1: gem5-SALAM Full-System Architecture

To design a new accelerator a user starts by describing its functionality in C and compile it to the LLVM IR. SALAM is then able to generate a cycle-accurate model for the accelerator, which can be tuned at the user's discretion via a device hardware profile. By leveraging a custom execution model for hardware accelerators, SALAM is able to avoid many of the scalability concerns of traced-based accelerator simulators like gem5-Aladdin [7], and model large and complex systems comprised of many hardware accelerators. Additionally, system-level components added by the

SALAM infrastructure enable forms of communication and data movement between accelerators that are not currently supported in other existing simulators. To address the limitations found in other simulators, gem5-SALAM offers the following contributions:

1. Accurate modeling of datapath structure, area, and static leakage power based on analysis of algorithm-intrinsic characteristics exposed by LLVM.

2. Cycle-accurate modeling of dynamic power and timing through a dynamic LLVM-based runtime execution engine, through gem5-SALAM's dynamic execute-in-execute LLVM-based runtime engine.

3. Separation of datapath and memory infrastructure to enable independent tuning and design space exploration.

4. Flexible system integration that directly exposes accelerator models to other system elements, within gem5, to enable complex inter-accelerator communication and synchronization, using pre-existing gem5 simulation constructs.

5. General purpose C++/Python API for accelerator modeling that decouples computation from system communication, and enables customization and specialization to match user modeling needs.

5.1 SALAM Infrastructure

Modeling of hardware accelerators in gem5-SALAM can be broken into two main components: 1) static graph elaboration to capture the basic datapath and constraints of the accelerator and 2) a dynamic runtime engine to capture the runtime characteristics of the accelerator. These two components work hand in hand to generate accurate timing, power, and area models for application-specific hardware that are independent of input data.

5.1.1 Static Graph Elaboration

The design of a new hardware accelerator in gem5-SALAM starts with describing the functionality of the accelerator in a programming language like C/C++. Designing accelerators at this level of abstraction is simpler than working with common hardware description languages and provides users

with a test bench for verifying the functionality of their hardware accelerators later. From here the user can generate the LLVM IR for their application using an LLVM compiler like clang, as shown in Fig. 6.5. When generating the IR, a user should consider applying any algorithmic optimizations such as instruction vectorization and loop unrolling. SALAM will elaborate the datapath precisely as it is described in the LLVM IR, meaning that a loop will only be vectorized in the datapath if it was vectorized in the LLVM IR.

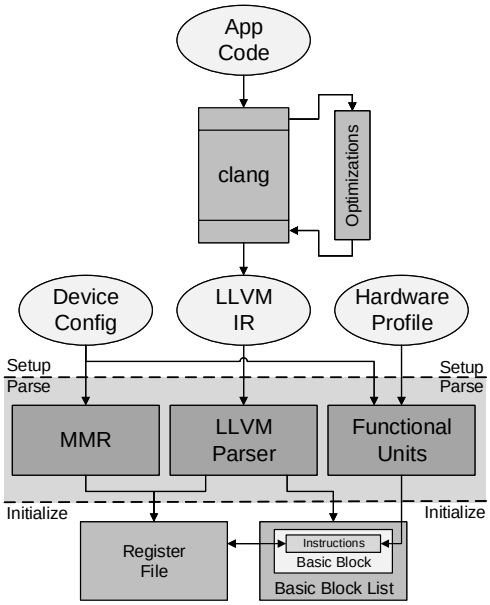


Figure 5.2: Accelerator Model Generation

For very complex applications that have many nested functions, a user should consider whether or not it makes sense to split the application into multiple accelerators. SALAM currently supports one function per accelerator. If the compiler does not automatically inline functions in the IR, this must be done manually by the user. Alternatively, gem5-SALAM supports inter-accelerator communication, enabling users to split complex tasks across multiple accelerators.

Once the LLVM IR has been generated, it is passed to a custom LLVM parser for datapath elaboration. Here the application's CDFG is extracted from the LLVM IR and functional units for the datapath are allocated. This static allocation provides for runtime data-independence in order to accurately model energy and area requirements for the datapath. Users can further tune the datapath structure using device-level configurations. This enables users to enforce functional unit re-use and adjust timing within the datapath based on the hardware profile they wish to use.

5.1.2 Dynamic LLVM Runtime Engine

The runtime model pictured in Fig. 5.3 consists of a series of queues controlled by gem5-SALAMs "runtime scheduler" that tracks and evaluates instruction dependencies, allocates hardware resources, and monitors the statuses of in-flight compute and memory operations.

5.1.2.1 Reservation Queue

Execution begins in the "Reservation Queue", pictured on the right in Fig. 5.3. A dynamic instruction map is generated at the granularity of basic blocks from the statically elaborated CDFG, and the contents of the first basic block of the application are imported. As each instruction (operation) is added to the queue, dynamic dependencies are generated by searching upward in the reservation queue as well as the in-flight compute and memory queues. Additionally, the execution of previous instances of the same instruction and all instructions that read from its destination register are checked to be in-flight or completed. This ensures each instruction can only be launched once all of its dependencies have been met.

Instructions that function as basic block terminators trigger the reservation queue to load the next basic block immediately after evaluation. This enables the simulation of a custom, highly-parallelizable, data-path with support for the pipelining of loop structures. Compute instructions, which have an associated hardware unit mapping, have the additional constraint that their mapped hardware unit is available if resource limitations are defined by users. This enables the user to enforce reuse in portions of the data-path via the "Hardware Profile" provided in the front-end, as shown in Fig. 6.5. When an instruction is ready to execute, it is then transferred to an appropriate operation queue.

5.1.2.2 Compute Queue

Compute instructions are all instructions that can be resolved by the simulator using only values stored in local registers. These instructions are transferred to the "Compute Queue" where their functional units are invoked. For simulation purposes, the computation is done immediately, but the commit of the result can be delayed by a number of operational cycles that is uniquely configurable for each function unit type. The dynamic energy of each active instruction is also calculated at

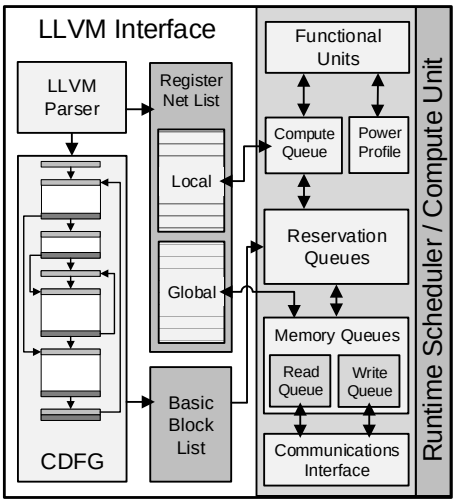


Figure 5.3: LLVM Runtime Engine Simulation Model

this point to estimate the power of the compute data-path. Once an instruction (operation) is ready for commit, the instruction is removed from the queue, the hardware unit is released, and the Reservation Queue is signaled to resolve dependencies on the committed instruction.

5.1.2.3 Memory Queues

Memory instructions will be transferred to the Read/Write queues shown in the bottom-right corner of Fig. 5.3. These queues forward the memory requests to the connected communications interface, described in Sec. 6.2.3, which is responsible for interfacing with gem5's other system elements. The memory queues operate asynchronously from other elements of the runtime engine in order to handle memory requests that complete in between the compute cycles of the runtime engine. When a memory request is ready to commit, the request is removed from the queue and the "Reservation Queue" is signaled to resolve dependencies on the committed request.

5.1.3 Metrics Estimation

All of the elements that make up the simulator perform some form of internal statistics tracking that is fed into the "LLVM Interface" during each phases of operation. The statically elaborated CDFG provides the baseline model for static power and area, while the dynamic runtime generates and records evaluation data each cycle during simulation. Because there are also so many configurable knobs, a brief overview of the related parameters will be in each subsection below, with greater

detail in Sec. 5.1.5.

5.1.3.1 Power and Area

The power estimation model utilizes parameters defined within the hardware profile and the device config as shown in Fig. 5.1 and Fig. 6.5. The hardware profile contains power and area profiles for common fixed and floating-point hardware functional units as well as single bit registers operating with various latency's. The generation of this profile is detailed below in Sec. 5.2.1. The device config allows the user to constrain the amount of each hardware functional unit that is in the system. The static power metrics use the static CDFG to account for all functional units within the system, the simulation runtime, and the hardware profile to determine the leakage power lost in the system due to the functional units. The dynamic power used by the functional units is calculated each cycle for each active functional unit and is the combination of the switching and internal power dissipation as defined in the hardware profile as a function of the accelerator clock speed, which is defined in the device config.

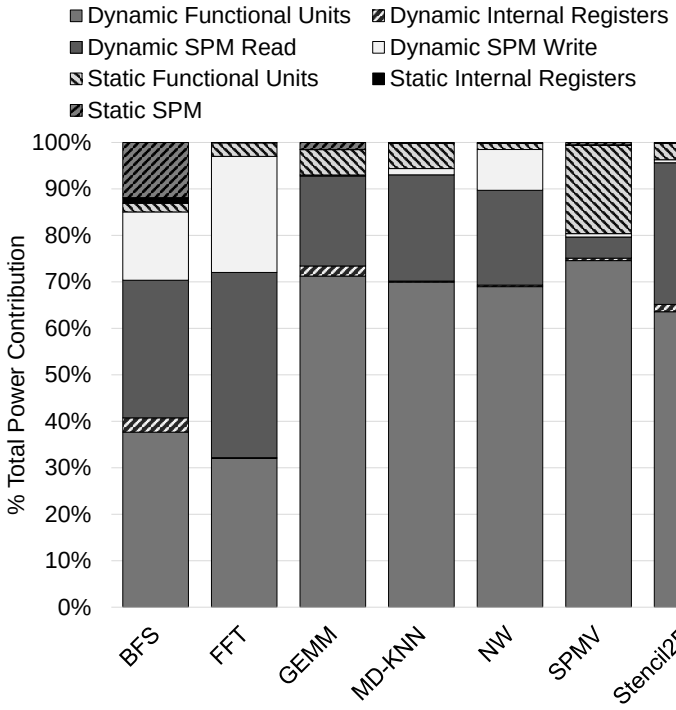


Figure 5.4: Example of total power analysis of multiple benchmarks using private SPM.

Similarly the LLVM IR as used in gem5-SALAM exposes the internal registers and their bit size, while the runtime engine tracks the read and write activity each cycle. This allows gem5-SALAM

to also model the runtime energy consumption of internal data-path logic using the same method as described for the functional units, where the static and dynamic power and area are calculated based around the single-bit register results obtained for the hardware profile.

By utilizing gem5's memory interface, gem5-SALAM also has built-in support for power modeling of the shared memory with respects to user-defined configurations to the gem5 system. To account for situations where the user prefers private memory elements integrated within an individual accelerator, gem5-SALAM takes advantage of McPat's Cacti [78] by automatically passing private memory parameters and usage statistics internally to provide the power and area profile upon runtime completion. Fig. 5.4 shows the type of results generated when performing full power analysis for multiple MachSuite [31] benchmarks ran in parallel with private SPM.

5.1.3.2 Performance and Occupancy

gem5-SALAM also provides a variety of performance metrics to the user post-simulation. Within the device configuration, gem5-SALAM defines the cycle time that 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. 5.2.1. 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 cycle-counts, runtime, and functional unit occupancy of accelerator models.

During the dynamic runtime simulation gem5-SALAM logs which instructions are scheduled or in-flight for each cycle. This additional data point combined with configurable hardware resources allows for a fine grained analysis and exploration tool for exploring occupancy levels within the system. Some examples of this that are explored more in Sec. 6.3 include the ability to view functional unit occupancy as a function of data-availability by sweeping port sizes or optimizing functional unit resources for maximum parallelism.

5.1.4 gem5 Integration and Scalable Full System Simulation

gem5 provides a robust, extensible, and well-tested framework that makes it ideal for evaluating new heterogeneous architectures. Unlike other simulators that built another simulator before integrating into gem5, gem5-SALAM was built from the ground up within the modular APIs offered by gem5. **gem5-SALAM does not require a rebuild of gem5 to add new accelerators.**

Since accelerator models are built on top of native gem5 constructs, they can be integrated anywhere within a gem5 simulation instance that supports a "gem5::TimedObject". This gives designers the freedom to explore both tightly and loosely coupled accelerator designs and even nest accelerators within the datapaths of other system elements. Additionally, gem5-SALAM offers multiple types of DMA devices including block and stream DMAs. gem5-SALAM is the first and only gem5 extension to provide a full suite of extensible simulation models for pre-RTL and pre-HDL design space exploration of application-specific hardware accelerators.

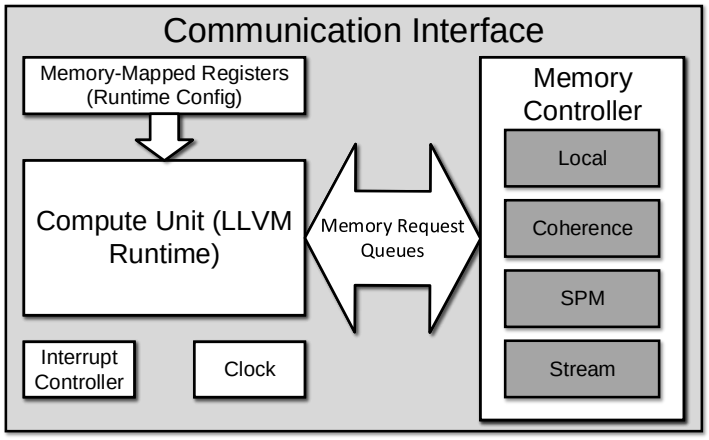


Figure 5.5: Communications Interface

5.1.4.1 Compute Unit and Communications Interface

At the core of our API are two basic models: the Compute Unit and the Communications Interface. A compute unit represents the datapath of a hardware accelerator. An example of this is described in Sec. 6.2.2, however that is the API provided by gem5-SALAM allows for the construction of other simulation models that can hook cleanly into the rest of gem5-SALAM's system infrastructure.

A Communications Interface, shown in Fig. 6.6, provides access to the system interfaces of gem5 for the purposes of memory access, control, and synchronization. It accomplishes this by providing three basic interfaces in its API: Memory-Mapped Registers (MMRs), memory master ports, and interrupt lines. Fig. 6.6 shows the the most basic model of a "Communications Interface", or the "CommInterface". It supports programming via its MMR and access to memory through up to two master memory ports. This enables designers to generate accelerators with parallel access to dif-

ferent memory types in parallel, including SPMs and caches. Furthermore, gem5-SALAM supports more specialized memory access types, such as stream buffers and SPMs with customized partitioning. To demonstrate the extensibility of the gem5-SALAM API, custom interfaces supporting stream inputs and custom-ported memories have been built upon the base "CommInterface" model, that integrate seamlessly with the LLVM Runtime engine, and are employed in the architecture explorations described in Sec. 6.3.

A user-configurable memory controller enables the distribution of parallel memory access across all memory interfaces as shown in Fig. 5.6. For private memory and streaming interfaces, the memory controller also supports the configuration of memory partitioning and bandwidth. Read and write request queues allow tracking of in-flight memory requests, and will automatically notify the Compute Unit when memory requests have been fulfilled. Additionally, the clocks of the Communications Interface and Compute Unit can be configured independently.

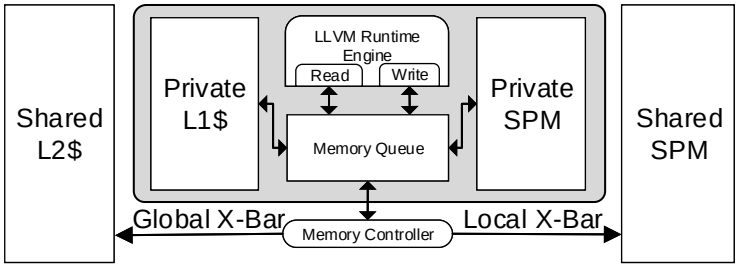


Figure 5.6: Accelerator Memory Model

Empowering users to explore innovative accelerator designs and hierarchies was a major design goal with gem5-SALAM. As a result, all of the Communication Interfaces are interchangeable, without requiring any modification of the corresponding Compute Unit. This stands in contrast to other simulators, like gem5-Aladdin and PARADE, that are unable to decouple the execution models of their accelerators from their control and communications interfaces.

5.1.4.2 Multi-ACC Simulation

gem5-SALAM was built with multi-accelerator designs in mind. The configurable Communications Interfaces enable communication and the sharing of data between hardware accelerators. In order to introduce some degree of device hierarchy, and simplify configuration from the user perspective, gem5-SALAM provides a hierarchical Accelerator Cluster construct. An accelerator

cluster consists of a pool of accelerators coupled with a shared DMA and scratchpad. A local crossbar provides access to shared resources as well as the MMRs of other accelerators in the cluster. This enables accelerators to communicate directly with each other and access shared data. Accelerators within a cluster can still be configured with private scratchpads and other memory interfaces. A global crossbar is also included to grant access to resources outside of the cluster, such as DRAM. If caches are enabled, a last-level cache is added between the global crossbar and system memory interface to enable cache coherency between accelerator clusters and other processing elements.

This setup enables numerous opportunities for design space exploration of accelerator rich systems. For one, users have the ability to track memory statistics such as bandwidth utilization and cache misses on shared system resources. Alternatively, accelerator clusters can be used to construct templates for complex accelerator tasks that can be replicated for parallel execution. Importantly, the capability for accelerators to communicate and self-synchronize in gem5-SALAM reduces host CPU overheads for control and synchronization, This enables the system and simulation to scale better with a larger number of accelerators than other pre-RTL simulators.

5.1.4.3 Control and Synchronization

Control of accelerators within gem5-SALAM is largely enabled via memory-mapped registers. Each of the Communications Interfaces described above comes equipped with configurable status, control, and data registers. This enables low-level device configuration as well as basic synchronization controls. Used in conjunction with the other memory interfaces, this enables direct communication and coordination between and accelerator and host processor, and even between accelerators. Additionally, each Communications Interface also supports the generation of interrupts to the system interrupt controller.

For the synchronization, by default, our accelerator models are capable of generating interrupts through the ARM GIC. Additionally, the MMRs of accelerators are set to respond with their current values when read by the host CPU. The CPU's perspective of the accelerator is the same as it is for any other memory-mapped device. The accelerated portion of the host code is replaced with a device driver that sets the accelerator MMRs and performs and necessary data movement between host and accelerator memories. Drivers for DMAs are included in our project files. Drivers

for accelerators are highly device-specific, but templates are provided to simplify the development process.

gem5-SALAM utilizes gem5's Full System simulation mode, as opposed to Syscall Emulation. To simplify driver development, the simulation is run with a bare-metal kernel. gem5-SALAM also supports simulation with a full Linux kernel (provided by gem5), however drivers will need to be adapted to map virtual memory addresses of device MMRs.

5.1.5 Simulation Setup and Configuration

Setting up a new simulation in gem5-SALAM has been streamlined as much as possible to require minimal effort from the end-user and to be language agnostic with the use of LLVM. The simulation profile needed to run the simulation can be divided into two main categories: (1) single accelerator configuration and (2) accelerators cluster configuration.

5.1.5.1 Single Accelerator Configuration

Each accelerator must first be configured independently before being added to the accelerator cluster. Each accelerator configuration contains the accelerated code segment, which is passed through the Clang compiler to generate the LLVM IR used by the simulator as in Fig. 5.7. The user can customize the underlying structure of the datapath by applying compile-time optimizations like loop-unrolling and vectorization at this point in time. Within the host code itself, the user must define the locations of Memory-Mapped Registers (MMRs) to be used by the accelerator. Similar to the programming abstraction of OpenCl or CUDA, the inputs and outputs are exposed as pointers within the accelerated function declaration. The user must then map these pointers to the MMRs of the accelerator, along with any other configuration variables/flags. This means that the programmer can change where an accelerator reads/writes its data at runtime through a device driver. Overall, minor changes need to be made to the application's host code to map the memory to the device. For example, if the use of DMA transfer is desired by the user then "memcpy" needs to be replaced by "dmacpy" in the application's host code.

Alongside the application's host and device codes, gem5-SALAM requires gem5-python device and system configuration files and the hardware profile. The system configuration file sets the gem5 specific interface parameters including: the number and size of ports, MMR base addresses and sizes,

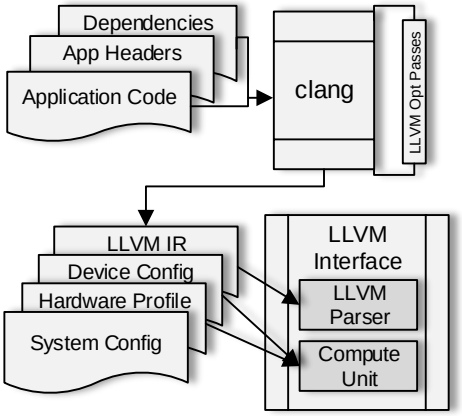


Figure 5.7: Single Accelerator Configuration

and accelerator memory ranges used to route data within the system. The device configuration is used to customize the configuration of the accelerator datapath and tune runtime parameters based on profiling data provided during simulation. This file includes options for customizing memory interfaces, device clocks, and setting datapath constraints. These configurations are passed to gem5-SALAM and the internal communications interface to define the interconnect between accelerator model and other simulation components in gem5. Additionally, within the device configuration there are a few options for configuring the dynamic runtime scheduler. Examples of each type of configuration file are provided to guide users in the design of accelerators. Alternatively users can control and sweep the same design parameters directly in gem5's Python API. This can be useful if using Python for design space sweeps, or integrating with other projects based on gem5.

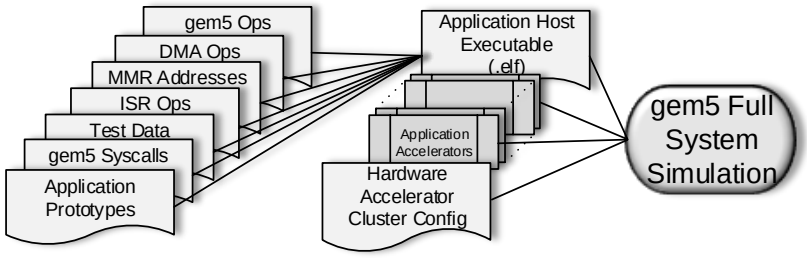


Figure 5.8: Accelerator Cluster Configuration

5.1.5.2 Accelerators Cluster Configuration

One significant feature of gem5-SALAM is empowering users to construct rapid simulation models of accelerator clusters. The accelerator cluster can contain any number of accelerators and the shared resources defined by the user between them. The cluster is generated through gem5-python scripts to initialize each of the individual accelerator and system elements as defined by the gem5-python device and system configurations shown in Fig. 5.8. The hardware accelerator cluster configuration automates the interconnection and initialization of accelerators. To facilitate this, gem5-SALAM provides a library of python classes that represent the hardware components with a C++ wrapper that passed arguments directly into our simulator and allows the user to reconfigure the device and system files without the need to recompile the base code.

5.2 Simulation Results and Validation

The evaluation and validation of gem5-SALAM can be broken down into three categories: (1) Metric validation, (2) single accelerator design space exploration, (3) multiple accelerators design space exploration.

5.2.1 Metrics Validation

Fig. 5.9 presents the validation flow for timing, power and area validation. The timing model of gem5-SALAM was validated on the MachSuite [31] benchmarks against RTL models generated by Vivado HLS. The power and area models for functional units in gem5-SALAM are based on the models used in gem5-Aladdin [7]. These models were also validated on the MachSuite [31] benchmarks against Synopsys Design Compiler, using an open source 40nm standard cell library and the gate switching activity produced by RTL simulation in Vivado. This validated hardware profile is included as the default configuration in gem5-SALAM, although the user can easily modify or extend this profile to explore custom hardware.

Table 5.1: System validation results

Benchmarks	FPGA			Simulation			Error (%)		
	Compute Time (uS)	Bulk Xfer Time (uS)	Total Time (uS)	Compute Time (uS)	Bulk Xfer Time (uS)	Total Time (uS)	Compute Time	Bulk Xfer Time	Total Time
FFT/Strided	879.35	93.58	972.93	867.77	95.58	963.35	1.32	-2.14	0.98
GEMM/ncubed	1343.31	179.01	1522.32	1315.24	181.97	1497.21	2.09	-1.65	1.65
Stencil2D	846.45	268.57	1115.02	854.14	275.98	1130.12	-0.91	-2.76	-1.35
Stencil3D	445.28	444.5	889.78	455.26	446	901.26	-2.24	-0.34	-1.29
MD/KNN	2489.66	118.74	2608.4	2568.45	112.96	2681.41	-3.16	4.87	-2.80
Average	-	-	-	-	-	-	1.94	2.35	1.62

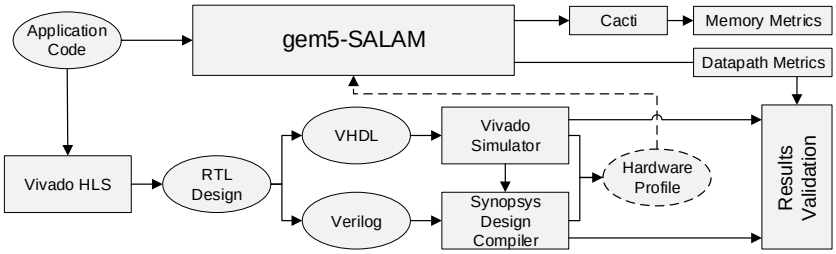


Figure 5.9: Validation Flow

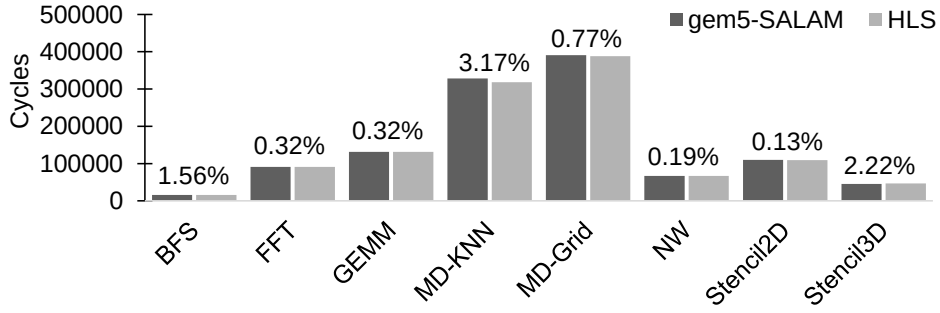


Figure 5.10: Performance Validation

Fig. 5.10 shows the timing performance validation for 8 benchmarks from MachSuite. Overall, the average timing error was approximately 1%. In each case, the input LLVM IR was tuned to reflect the same levels of Instruction Level Parallelism (ILP) as the datapaths generated by HLS. Applications like FFT (0.32% error), GEMM (0.32% error), and Stencil2D (0.13% error) had some of the lowest timing errors due to their highly regular, data-independent control. NW also exhibited a very low timing error of 0.19% due to the mapping of many of its runtime control dependencies to MUXs in both HLS and SALAM. The highest error appears in MD-KNN which relies very heavily on floating-point computation. HLS tools will generally attempt to minimize the number of floating-point functional units employed in a design, and enforce reused of expensive floating point resources. We validated gem5-SALAM by enforcing similar restrictions and reuse in the configuration of the MD-KNN accelerator, however the runtime mechanism for functional units employed by gem5-SALAM only approximates the internal wiring of those reuse circuits. Even so, the power and area estimates detailed below for the same accelerator justify gem5-SALAM's means of modeling functional unit reuse by showing low levels of error in both power and area.

Fig. 5.11 shows the power validation across the same set of benchmarks. Stencil3D was excluded

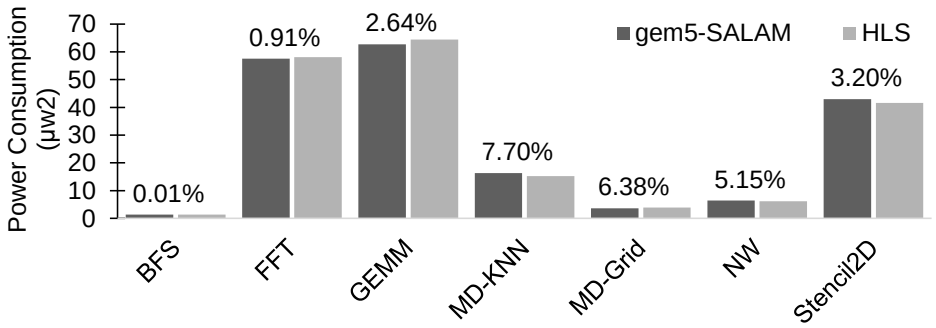


Figure 5.11: Power Validation

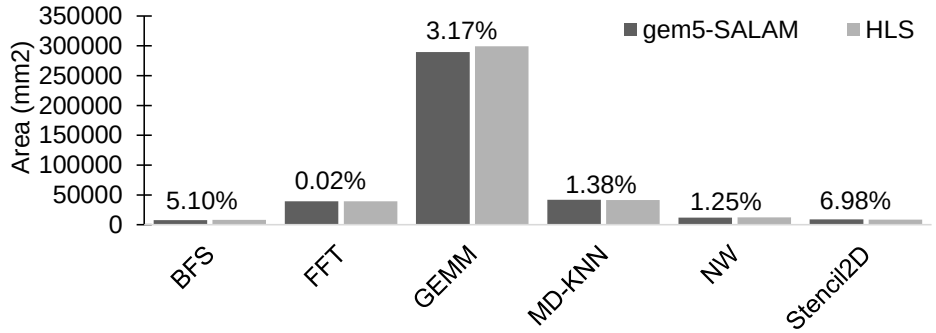


Figure 5.12: Area Validation

from this set due to Design Compiler running out of memory during elaboration. The average error in power estimation is slightly higher, at 3.25%. The MD-KNN, MD-Grid, and NW benchmarks show the highest power error due to heavier reliance on muxes and non-arithmetic operators. Variability in the power consumption of these operators leads to a slight overestimation of power requirements on average. These results are very comparable to those produced by Aladdin.

Fig. 5.12 shows the area validation on the evaluated benchmarks. On average gem5-SALAM is able to estimate chip area with an error of 2.24%. MD-Grid was excluded from this test due to custom IPs within its data-path preventing Design Compiler from providing area estimations.

5.2.2 System Validation on FPGAs

For the purpose of system validation, we synthesized five of the benchmarks and executed them on a Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board, which has the XCZU9EG SoC chip. 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 6.4. 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 a 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.

Table 6.4 displays a similar trend to that seen in the comparison to RTL simulation. Positive error indicates when simulation was faster, while negative errors indicated faster FPGA times. The biggest discrepancies vs. the previous comparison can be seen in GEMM and FFT. These two benchmarks operate on double-precision floating point, whereas most of the other benchmarks operate on integer types. By default, gem5-SALAM 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. On average the absolute compute error across all benchmarks was 1.94%. Likewise, the average absolute error in data transfer times was 2.35%. This is primarily due to a difference in cache invalidation times between the ZCU102 and the simulation.

5.2.3 Design Space Exploration

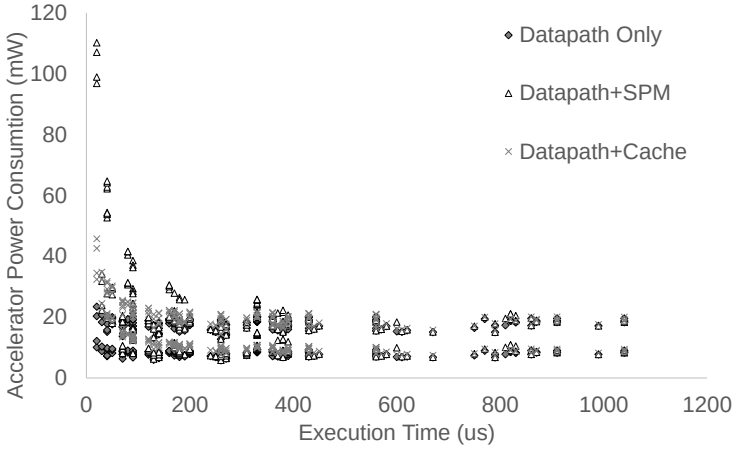


Figure 5.13: GEMM Design Space Pareto Curve

5.2.3.1 Case Study: Generic Matrix Multiply

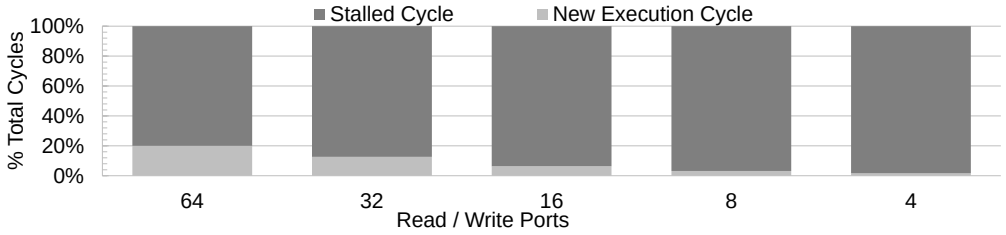
To show the capabilities of gem5-SALAM we have provided an example of the design space exploration that can be achieved by looking at the General Matrix multiply (GEMM) application. A simple bash script was created to sweep the quantity of available functional units defined in the device configuration shown in Fig. 5.7 for a range of memory bandwidth allocations as defined in the accelerator cluster configuration shown in Fig. 5.8 to determine the benefits of memory parallelism with the GEMM application. The output of each of these simulations was exported in CSV format and combined to enable analysis of the power and performance estimations as a Pareto curve in Fig. 5.13. Here, there is an interesting trend of duplicate results with higher power consumption that suggests over-allocation of functional units versus the runtime parallelism that exists in the accelerator. One source of this discrepancy arises from limitations in memory bandwidth.

Fig. 5.14(a) presents how the proportion of stall cycles to running cycles improves as we increase the memory bandwidth. Supporting more than 64 read/write ports in the design provides no additional benefit since this is the maximum width of the data-path. Observing that the amount of stalled cycles is still higher than cycles that scheduled new instructions, we can break the stall sources down even further as shown in Fig. 5.14(b). This graph allows us to see that the design space for GEMM is most heavily influenced by floating-point computations and data transfer into the accelerator. The solid black bands representing cycles of only floating-point computation. In the 32 and 64 port columns, Fig. 5.14(b) indicates that by increasing the bandwidth we are also creating more temporal parallelism in the nested floating-point operations that are decoupled from memory operations.

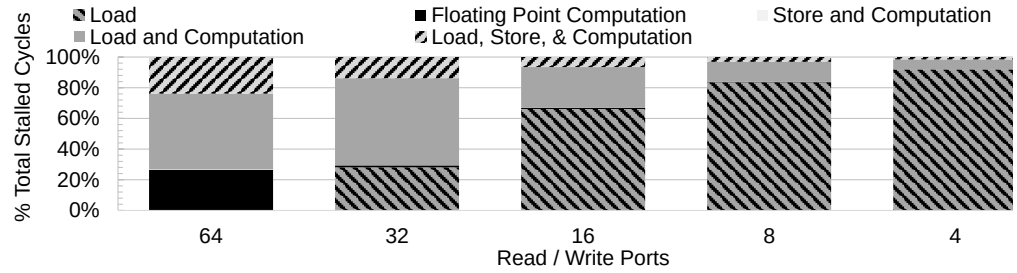
5.2.3.2 Co-Designing using gem5-SALAM

By using gem5-SALAM to examine functional unit occupancy at a cycle granularity, we find low occupancy among floating point adders. Additionally we can determine that 64 total floating points addition units in our accelerator can provide nearly the same throughput as 128 with only an increased performance cost of 4 cycles. Using this now as our basis we will hold the number of floating-point addition units at 64 and re-evaluate our design space domain.

Fig. 5.15 shows a sampling of the additional exploration pathways available to the user with the



(a) Runtime instruction scheduling comparison.

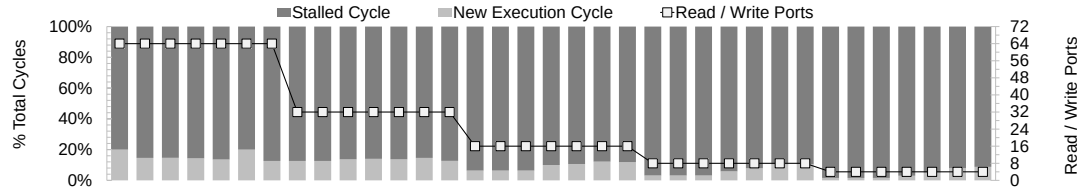


(b) Runtime stall cycle unfinished operation breakdown.

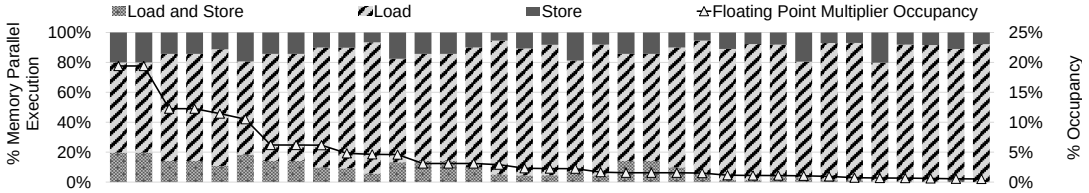
Figure 5.14: GEMM Stalls Breakdown

use of gem5-SALAM to aid in the co-design of accelerator applications. By first using average values across a wide range of sweeps, we have quickly and effectively narrowed the scope of the design space such that we can now explore each remaining path directly. In Fig. 5.15(a), we repeated our initial experiment for each configuration individually and plotted the stalled cycles versus cycles where new operations where executed for the remaining sweeps. We can now explore the parallelism between memory operations and floating-point operations. The details are provided in Fig. 5.15(b) which examines the cycle execution scheduling activities rather than the stalls as in Fig. 5.14(b). Fig. 5.15(b) shows the overlap between load and store operations within the application and overlay the average occupancy of the floating-point multiplication units. These two elements for each column show a clear trend of higher occupancy levels for sweeps with minimal overlap between load and store operations.

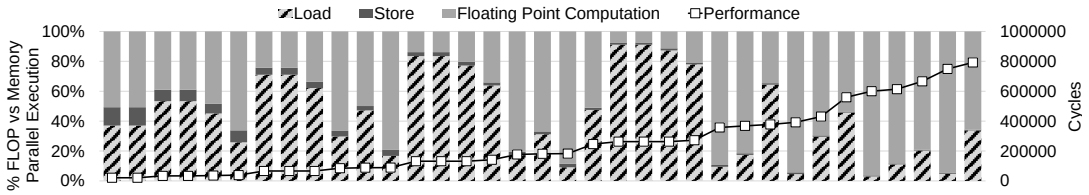
To further explore the analysis, Fig. 5.15(c) incorporates the floating-point computation scheduling activities into the results and overlays the overall performance for each sweep. This allows us a new insight into the causation of the previous results. We can observe that the optimal performance is obtained when the ratio of operations scheduled is nearly the same as the ratio of floating-point operations to memory operations in the GEMM algorithm. Looking at these same metrics from a different perspective, Fig. 5.15(d) evaluates which type of instruction on average is scheduled each



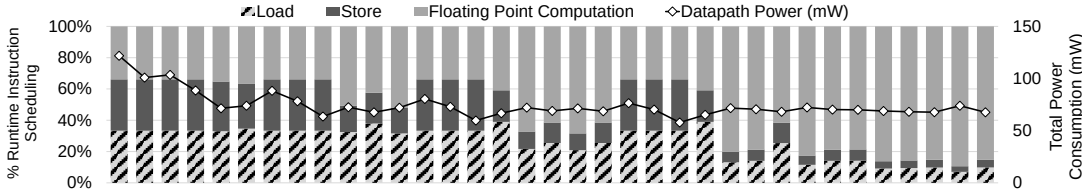
(a) Exploration of Datapath Stalls vs Memory Ports



(b) Memory Parallelism vs Floating Point Function Unit Occupancy



(c) Memory-to-Compute Ratio vs Execution Time



(d) Memory-to-Compute Ratio vs Power Consumption

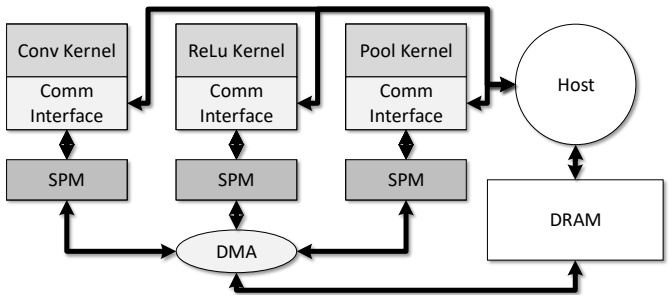
Figure 5.15: GEMM Memory and Compute Design Space Exploration

cycle with an overlay showing the total power consumption.

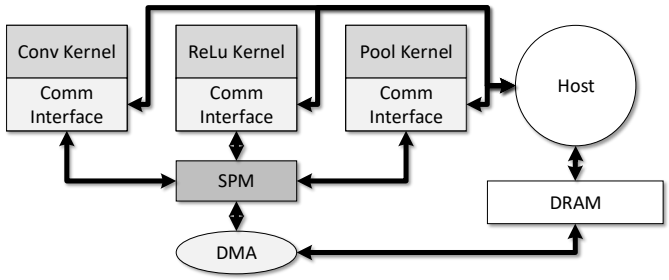
5.2.4 Multi Accelerator Design Space Exploration

One of the key benefits of gem5-SALAM over other existing pre-RTL simulators is its increased support and flexibility for design space exploration of multi-accelerator workloads. Flexibility in system interconnects and hierarchical models like the accelerator cluster enable simulation of complex hardware accelerator interactions not available in gem5-Aladdin or PARADE.

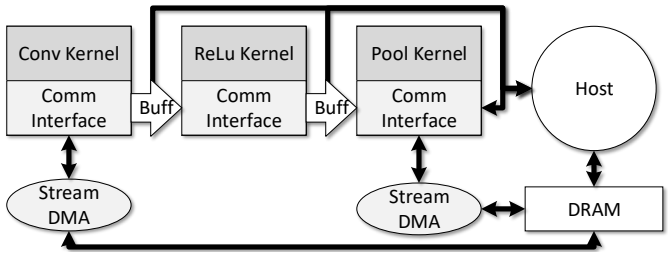
To demonstrate this, we implemented the first layer of a Convolutional Neural Network (CNN) in gem5-SALAM. This consisted of dedicated accelerators for the 2D convolution, max pooling, and rectify linear (ReLU) functions. The cluster of accelerators was evaluated in three different scenarios. In the first scenario, shown in Fig. 5.16(a), each accelerator used its own private memory.



(a) Private Scratchpad Memory



(b) Shared Scratchpad Memory



(c) Direct Communication

Figure 5.16: Producer-Consumer Accelerator Scenarios

Similar to the semantic supported by gem5-Aladdin, DMAs were responsible for data movement between accelerators and the host was responsible for activating and synchronizing the accelerators. For the purposes of timing comparison, this scenario serves as the baseline for comparison. In the second scenario, shown in Fig. 5.16(b), accelerators have a shared scratchpad memory for directly passing data to each other, but no way of knowing when their data is available. In this scenario synchronization with the a central controller is necessary to maintain synchronization across accelerators, similar to the model in PARADE. The removal of data movement between accelerators results in a 25% speedup in the end-to-end execution, but the requirement for external synchronization still limits overall performance. In the third scenario, shown in Fig. 5.16(c), accelerators communicate directly with each other through stream buffers that function in a similar fashion to the AXI-Stream interfaces employed in modern ARM-based SoCs. In this scenario

no centralized controller is needed or used to synchronize the operation of the accelerators. By enabling inter-accelerator pipelining, the end-to-end execution is improved by a factor of 2.08x over the baseline. gem5-SALAM, which simulates all three scenarios, is the only simulator of the three that is capable of modeling this multi-accelerator integration. This is due to the fact that this style of streaming data transfer requires a two-way handshake for synchronizing data movement between devices that may internally operate with different data rates, or otherwise experience fluctuations in their data rates due to runtime variables. The black box accelerator models of gem5-Aladdin and PARADE lack the fundamental interfaces required to facilitate this form of communication. More importantly, the inability to decouple basic control and communication interfaces from the execution models of gem5-Aladdin and PARADE without significant redesigns to their underlying structures means that they will continue fall even further behind as the complexity of SoC designs increase. gem5-SALAM's API was built to be easily extensible and inherently support integration with the work of other developers in the gem5 ecosystem.

CHAPTER 6: EXPANDING SIMULATION DESIGN ABSTRACTIONS WITH GEM5-SALAMv2

The gem5-SALAM project was initially developed to address the shortfalls of other pre-RTL simulators in modeling accelerators with runtime, input-dependent behaviors. The datapath modeling scheme employed by gem5-SALAM was first proposed in "Scalable LLVM-Based Accelerator Modeling in gem5"[14] and was used to provide timing models of simple, single accelerator systems. gem5-SALAMv1 expanded on this simple timing model by adding power and area estimation, as well as validation of full system performance against FPGA implementations of the same benchmarks[15]. While the initial gem5-SALAMv1 offered large amounts of flexibility in system design points, this came at the cost of overwhelming users with large numbers of switches and configuration options as the complexity of systems increased. A major challenge in working with SALAM for large systems arose when debugging errors in-memory mapping of dozens of devices. In the case of the MobileNet example described in Sec. 6.4, users would need to map well over 150 different memory-mapped accelerators, memories, and DMA devices, either by hand or with their own tools. Errors in that memory map would then propagate through over a dozen unique configuration files that would need to be debugged concurrently. Thus, enabling exploration of complex hardware architectures was a significant force driving the development of SALAMv2.

gem5-SALAMv1 [15] established a solid framework for pre-RTL design exploration, but was limited in the scope of the complexity of accelerators that could be configured and modeled due to some shortcomings in the initial design as seen in table 6.1. To broaden the scope of systems that could be modeled we have performed an extensive upgrade on gem5-SALAMv1 that was driven by 3 main design goals. The first of these goals was the improvement of user interfaces to both simplify the design of complex systems while expanding the number of configuration controls and profiling options available to users. The second of these goals was to improve the extensibility and scalability of SALAM's LLVM runtime models through a complete overhaul of the LLVM elaboration and

simulation runtime. Lastly the gem5-SALAMv2 upgrade needed to improve upon the flexibility of system integration present in gem5-SALAMv1 by adding new interfaces and control/synchronization paradigms that were not present in previous version. These improvements and additions allow for modeling far more complex systems, as shown in Fig. 5.1.

Table 6.1: Comparison of some of the shortcomings present gem5-SALAMv1 and how we have improved upon those aspects in gem5-SALAMv2.

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

All features that existed in gem5-SALAMv1 have carried over into 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 greatly expand the capabilities of the system while also increasing performance and decreasing runtime of simulations. The most prominent upgrade from gem5-SALAMv1 is the full integration of LLVM used in our elaboration engine. By embedding LLVM internally, we can now utilize the full LLVM API internally inside SALAMv2. This allows us access to all of the information present in the IR and utilize 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 now have the ability to 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 be able 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, which was bound to a specific application in gem5-SALAMv1 and an independent and parallel system to the simulation engine. gem5-SALAMv1 relied on an integrated scratchpad memory that had to be manually configured for each application, but now gem5-SALAMv2 has the ability to have multiple memory types that can have any arbitrary connectivity as desired by the user.

Lastly gem5-SALAMv2 fundamentally redesigns the design process for integrating new hardware models into gem5 by introducing three new fundamental design abstractions: operations, tasks, and architectures. Through these abstractions designers are able to decouple functionality from timing in gem5's event-driven simulation, while also incorporating hardware design insights into their simulation models.

6.1 Abstracting Timing and Concurrency in Simulation

Before diving into the details of gem5-SALAMv2's core changes, it is important to understand the principles of event-driven simulation and how they guide the design of simulation models in traditional simulators.

6.1.1 Event-driven simulation and gem5's timing model

Simulation within gem5 is built around a priority event queue. Time within the simulation is divided into discrete time units referred to as ticks. Each time the simulation tick is advanced, the top-level scheduler will launch whichever events are scheduled to operate on that tick. In order for a simulation object, or simobject, to operate within the simulation, it must register an event with a timestamp and a callback function that corresponds to some function of the simobject. This creates a great deal of complexity when trying to describe the functionality of a particular piece of hardware. Long-running operations within a piece of hardware need to be discretized into simulation steps of

finite duration. This presents a particularly interesting challenge when a piece of hardware needs to wait for other events to occur in the simulator, since the simobject cannot simply spin and wait without stalling the rest of the simulation. Simple systems, like address-based memories that only need to simulate address lookup and data movement, are generally able to be modeled with few, if any events. More complex systems, like a CPU, can require a large number of events and callbacks spread across numerous simobjects.

In addition to describing the functionality of devices through events, developers must also consider concurrency within the simulated system. All events scheduled for the same simulation tick are considered to be concurrent. Unfortunately the simulation engine itself only maintains a single scheduling thread for processing events. This can lead to some design challenges such as when an event for modeling the end of a delta cycle on a register is scheduled after an event implementing a sensitivity check on that register. In hardware the delta cycle should be complete at the clock edge before the check on the register is made. To resolve these cases, the event queue is actually implemented as a priority queue. Developers can set a particular priority level when scheduling an event, however this adds a new dimension of complexity when trying to map the functional behavior of hardware to event-driven simulation.

gem5 simulations offer varying degrees of fidelity ranging from simple functional simulation, to the more advanced atomic and timing models. The functional simulation model in gem5 ignores timing constraints on memory transactions in the system, allowing for fast simulation at the cost of ignoring memory overheads and related costs in a system. This mode also has the lowest degree of support from common simobjects released as part of gem5. The atomic model of operation functions similarly to the functional model, however it appends additional timing information within the simulation that allow for reasonable timing estimates to be made about a system. This mode is commonly employed for fast-forwarding simulations to a particular timestamp of interest. The timing model of operation heavily leverages the event callback system of gem5 to provide more precise estimates of system overheads such as memory bandwidth. This is the most common mode of operation in gem5 and the only mode supported by gem5's out-of-order CPU models and more advanced memory interfaces. The higher degree of fidelity comes at the cost of significant simulation overheads that drive researchers to only operate in timing mode for time segments of interest. While

the various timing models provide a high degree of flexibility and utility for researchers, they also introduce a significant limitation. During simulation, only one of the three models can be active across the entire system at a time. In order to change models, users must take a snapshot of their simulation state and restart it with the alternate model. This also means that users do not have the option of mixing modes of operation across a single simulation in order to leverage less detailed models for background elements.

6.1.2 Modeling Hardware Timing and Concurrency in gem5-SALAM

gem5-SALAM was developed as an effort to simplify the tasks of introducing new hardware models into gem5 simulation through a highly flexible interface and set of abstractions. The user interface of gem5-SALAM is very similar to HLS design flows in that users describe the functionality of their hardware through a functional description in C/C++. These functional descriptions are then compiled into LLVM IR which is imported into the simulation as a control and dataflow model. While the high-level timing and memory interfaces are provided by gem5, gem5-SALAM provides a wide array of controls to provide a fine degree of datapath level tuning that is more comparable to HLS and RTL design flows than gem5's normal event driven simulation model. gem5-SALAM is built to provide answers to questions that hardware designers, not simulation designers, ask when constructing a system and does so through its unique *operation*, *task*, and *architecture* abstractions.

6.1.2.1 The Operation Abstraction

The base abstraction for simulation in gem5-SALAM is the operation abstraction. Operations have a direct correlation to instructions in LLVM IR in that they consist of operators, with well defined timing and computation cost constraints, that operate on a SSA-styled register format. Operations can vary significantly in complexity, representing anything from a simple bit-shift operator to addressing large N-dimensional tensors. Importantly the simulated costs of these operations can be well defined during static analysis and elaboration of the corresponding LLVM IR. Additionally the execution of the operation should be suitable for interpretation as a single atomic operation. In other words, even if the operation is expected to take thousands of cycles to execute in simulation, the full simulation of the operation can be completed in the same simulation tick in which it was initiated. The one exception to the rules for operations is for memory operators, whose costs are

determined by other elements of the simulated system. Even so, the atomicity of common memory operators makes them suitable for representation as operations.

Outside of the common instructions in the LLVM IR, the operation abstraction is also useful for modeling black-boxed hardware IPs that have well defined input-output characteristics. Often hardware designers do not have access to the specific implementation details of proprietary hardware IPs that they integrate into their designs. At the same time, many of these IPs have well defined dataflow and timing models that aid in their integration into real hardware designs. The operation abstraction of gem5-SALAM enables mapping of such designs to hardware simulation simply by creating a LLVM intrinsic or custom function call for the IP.

The operation abstraction in gem5-SALAM is managed by a custom compute event scheduler and a collection of event execution queues for tracking the progress of in-flight compute and memory operations. The event scheduler leverages dependency information from the statically compiled LLVM control and dataflow graph (CDFG) to assemble a runtime dependency graph among scheduled operations. Operations are scheduled at the granularity of LLVM basic blocks and can be issued to an active compute queue as soon as their runtime dependencies have been met. Operations can be both issued and committed out of order in order to model concurrency with the datapath of the simulated hardware. Additionally users can control the synchronicity of the event scheduler by enabling or disabling a lockstep execution mode. When the lockstep execution mode is enabled, new operations will not be launched for compute until all operations issued in the previous scheduling cycle have committed. Alternatively, with lockstep mode disabled, operations will be launched for compute as soon as their runtime dependencies have been met. This synchronicity constraint enables users to more precisely tune their hardware datapath structure and exists as a switch that can be readily changed without rebuilding the entire simulation. In addition to synchronicity controls in the scheduler, users are also able to manage hardware resource allocation at the operation abstraction. By default gem5-SALAM will allocate one fixed-function hardware unit for each instruction in the LLVM IR CDFG (i.e. one integer adder for each integer add, one FP multiplier for each FP multiply, etc.). Users, however, can override this default allocation by imposing limits on the allocation of specific hardware units in order to enforce reuse within a design. In the case of this more limited hardware allocation, an operation will only be launched for compute when there

is an available hardware resource for it to launch on.

Costing for the operation abstraction is handled at the granularity of the LLVM instruction class. Within a design users have the option of independently tuning the timing, static/dynamic energy, and functional unit area costs of each operation class. This tuning is key for mapping LLVM IR to hardware since the software IR contains many artifacts that do not translate to hardware, such as SSA artifacts like PHI instructions. In these cases those operation classes can be assigned a zero cost such that they do not impact scheduling or estimates of power and area consumption for the hardware. Additionally, since timing costs can be assigned independently for each operation class, the operation abstraction in gem5-SALAM provides users with the capacity to mix timing simulation modes both within and across simulation elements without needing to alter the timing simulation mode of the top-level gem5 simulation. This is key for enabling designers to vary the degree of detail for different parts of simulation and more narrowly focus on the impacts of particular design elements.

Having fine-grained control over the constraints of operations is key to translating LLVM IR to hardware datapath elements. For example as a software intermediate representation LLVM IR does not explicitly handle counters. Instead the counter behavior that controls a loop is derived from a combination of three instructions: *PHI*, *ADD/SUB*, and *ICMP*. The PHI instruction, which manages the current counter state, has no corollary in hardware. As such we can ignore its timing and energy costs by setting those to zero. Additionally we can also force the ADD/SUB and ICMP operators to run in the same cycle by setting one of their timings to zero, enabling gem5-SALAM's event scheduler to merge the operations. This produces the effect of a hardware counter that increments or decrements on every clock tick of the device when activated. This can further be refined by limiting the hardware units allocated in the datapath for one of the components in order to force functional unit reuse within gem5-SALAM's event scheduler. As a result we can model a hardware counter with a load and reset that is re-used during the operation of a piece of hardware.

Another place where the hardware constraint tuning is key is when we want to examine the impacts of small changes in datapath structure. For instance let's say that an accelerated function requires a vectorized add operation of 8 integer values. In the LLVM IR this is expressed as an ADD instruction that operates on <8 x i32> operands and can likewise be handled as a single operation

in gem5-SALAM. As a designer however, I may not want to model a hardware vector add with an issue width of 8. Perhaps I want to also explore the impacts on the system if my hardware's issue width is only 2 or 4-wide. Rather than regenerating the IR, which may impact other parts of the datapath as the compiler remaps vector accesses, I can simply tune the timing and energy constraints of the vector add operation to require more compute cycles with a lower energy cost per cycle. This enables gem5-SALAM's event simulator to implicitly model a vectorized operation with a lower issue width that is invoked multiple times to complete the full-width operation.

6.1.2.2 The Task Abstraction

Building upon the operation abstraction gem5-SALAM also offers the task abstraction. A task in gem5-SALAM consists of a collection of operations with a unified scheduler/controller. In terms of LLVM IR, tasks most closely map to functions or LLVM modules. The task abstraction is most useful for cases where an operation lacks clear atomicity, and requires multiple staged execution phases. Each task in gem5-SALAM maintains its own unique scheduler, and multiple tasks can run concurrently. Combined with the scheduling and timing controls offered at the operation abstraction, tasks offer the capacity to further isolate control flow, timing, and concurrency in designs in order to better reflect their simulated hardware. Additionally tasks have a convenient bridge to the operation abstraction through function calls in the LLVM IR. This enables users to break designs into multiple sub-tasks that are scheduled and invoked as operations within a top-level task. Much like other operations, task invocations can be made subject to the same controls for timing, concurrency and synchronicity.

A prime example of hardware that maps to the task abstraction is a sparse matrix-vector multiplier. In sparse matrix-vector multiplication the amount of computation and time it takes to complete that computation varies depending on the input data. The static timing and resource requirements of the operation abstraction fall short, as even control flow and pattern of execution can vary significantly based on the contents of the matrices and vectors. gem5-SALAM provides a simple interface for mapping this task to hardware simulation. The most simple solution is to express the task as a C/C++ function that can be compiled to LLVM IR and loaded into gem5-SALAM. However users have additional options for modeling and optimization. The sparse matrix-vector

problem consists of two sub-tasks: *vector retrieval* and *vector multiplication*. Provided that the algorithm can be divided cleanly into these sub-tasks, users can independently map each task to an independent hardware unit and run them concurrently with configurable memory channels to pass data between the tasks. This enables users to leverage complex caching and pre-fetching techniques in their hardware designs without writing custom event controllers to coordinate data movement.

6.1.2.3 The Architecture Abstraction

Up to this point the gem5-SALAM abstractions presented have focused almost entirely on the modeling paradigm of the SALAM component. The architecture abstraction is where task-based simulation is wrapped into the broader scope of gem5 simulation via a wrapper gem5 event and memory subsystems. Whereas the operation and task abstractions are primarily focused on the functional aspects of simulation and operation-level timing, the system abstraction provides mechanisms for establishing the base clock for timing simulation as well as a flexible composition of memory interfaces for system I/O tasks. As the entry point for task-based simulation the system interface provides users with the configuration options for all nested tasks and operations within the wrapper. This allows users to assign unique timing, energy, and area estimation models to each interface enabling multiple unique models within the same simulation run.

More importantly the system interface provided at the architecture abstraction serves to integrate the task-based simulation model into gem5's memory-mapped networking model. Each interface provides a memory-mapped programmable I/O interface that enables other elements in the gem5 simulation to communicate with the embedded simulation model through traditional memory channels. From the simulation integration perspective, this means that no custom system calls are needed for other gem5 simulation elements interact with gem5-SALAM's simulation. Additionally gem5-SALAM's system wrapper also provides a scalable memory port system that provides the operation and task abstractions with direct access to gem5's memory mapped communication system. This is key to enabling designers to develop hardware models that directly access other simulation elements via the programming abstraction provided by C/C++ translation to LLVM IR.

The main benefit of the architecture abstraction is that it enables developers to augment the existing gem5 simulation framework with the task-driven simulation model of gem5-SALAM. To

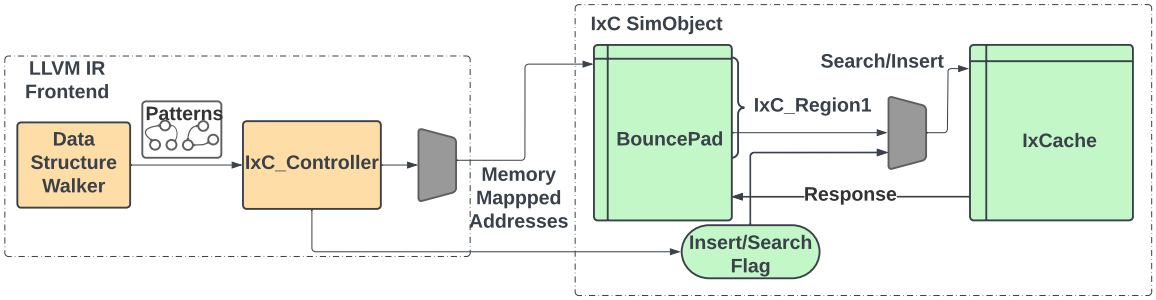


Figure 6.1: Index Cache Implementation

demonstrate this we present an implementation of a domain-specific index-based cache for storing the output of a sparse data structure memory walker based on X-Cache[79]. The implementation of the index cache requires three main components: the cache controller that leverages domain-specific information of the sparse data structures to manage insertions and replacements within the index cache, the domain-specific data structure walker for resolving misses, and a programmable memory space to store the cached data as shown in Fig. 6.1.

The interface for a user of the index cache is split between the cache controller and data structure walker. These components need to be co-designed based on the structure of the index data. Since the index cache works on meta tags rather than on address-based tags, the operation of these features will vary based on both the general structure of the data, and the runtime content of the data structure. Mapping these devices to the traditional gem5 API is cumbersome due to the need for runtime data-dependent control and complex domain-specific reuse characteristics. Attempting to map these devices to the event-driven design of gem5 requires extensive state machine development that is difficult, if not intractable, to generalize for supporting different types of data structures. This in turn leads to high non-recurring engineering costs that include design, build, and testing times for each new walker and controller pairing. In contrast, the task and operation paradigms offered by gem5-SALAM present the perfect test bed for rapidly prototyping and tuning designs based on different data structures. Rather than designing state machines, users can simply describe the data structure walk and look-ups using a set of simple C++ functions and import those into the LLVM runtime engine. From there all operation costs can be tuned to match real hardware performance, without needing to rebuild the entire simulation model. In this way we can tune the

costs of both hits and misses within the index cache to match existing hardware models. Equally important, developing a walker for a new data structure requires nothing more than loading a different LLVM IR file.

While the walker and cache controller require an understand of domain characteristics for data access, the programmable memory only needs to provide basic mechanisms for index-based data storage. This task maps very nicely to the programmable I/O device templates that are already present in gem5. Since we are using the customizable controller for managing insertions, replacements, and reuse within the cache, the only thing the memory component of the cache needs is an interface for handling control signals. To this end we built this memory component as an extension on gem5's PIODevice template, which provides a memory-mapped address space for reading and writing data to the device. We use this memory-mapped space for implementing a request "bounce-pad" and an insert/search control register. By leveraging the architecture abstraction provided by gem5-SALAM, this memory mapped device can be easily integrated into the path of the memory controller. This enables us to decouple the internal structure of the memory for the address-based memory schemes employed by gem5, and construct our meta-tag storage system.

6.2 gem5-SALAMv2 Specification

This section provides an overview of the new automated front-end setup and initialization process, provide insights into the redesigned dynamic LLVM runtime engine and evaluation methodology, detail the importance of the full integration of the LLVM API, and provide details into how these components have been integration into gem5.

6.2.1 System Setup and Initialization

One of the most significant upgrades in gem5-SALAMv2 is the introduction of new automation tools for constructing and exploring accelerator-rich systems. In gem5-SALAMv1, users were responsible for the manual generation of accelerator, cluster, and gem5 configurations. Because of this, developing large-scale systems in gem5-SALAMv1 quickly became impractical to implement and explore. Due to gem5-SALAMv1's development being primarily focused on accurately modeling multi-accelerator systems, there was still a significant amount of development effort required outside the SALAM ecosystem for a functioning system. As shown in Fig. 6.2, this created a situ-

ation where an end-user would be required to learn both gem5's configuration system and manually create and maintain a given system's memory map. While this worked fine for small single-function systems, a system at the scale explored in Sec. 6.4.2 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 in the release of gem5-SALAMv2.

Additionally, in gem5-SALAMv1 the hardware units and the supported variations were hard-coded, which limited the user to the available components without modifying the code. In gem5-SALAMv2 we have totally removed the hard-coded parameters and now support the use of hardware profiles defined by YAML files, as detailed further in Sec. 6.2.1.2. These profiles can now be easily expanded and swapped between applications, and allows the ability to define individual profiles at the accelerator granularity within the same cluster for complex applications. As with the system generator, the hardware generator was designed for ease of use and rapid prototyping.

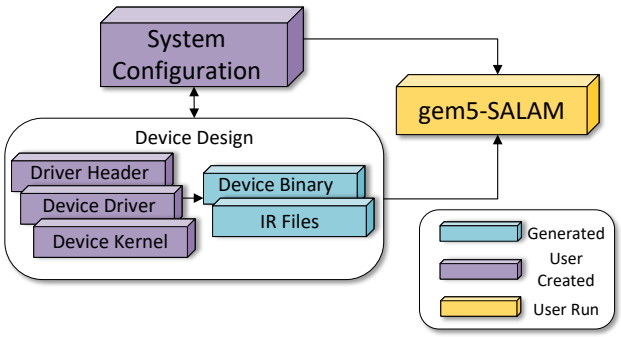


Figure 6.2: Overview of the gem5-SALAMv1 user design methodology. There was significant effort and knowledge of gem5 specific programming semantics required in comparison to gem5-SALAMv2 from the user.

To understand how impactful these new tools are to enabling large-scale architecture exploration, one must first understand the original system design methodology as is shown in Fig. 6.2. In gem5-SALAMv1, an end-user was entirely responsible for maintaining and configuring both the device driver headers and the gem5 system configuration. While these are simple to design if one has an understanding of gem5 and the SALAM's integration into it, these still required manual updates at every design iteration.

While these might just sound like quality of life improvements, there is a more important aspect

that our new design methodology in Fig. 6.3 presents. To allow for larger systems to be explored, we introduce the SALAM Configurator. The SALAM Configurator's primary purpose is to reduce the amount of boilerplate work that an end-user will have to perform on each design iteration. While an end-user might want to slightly modify or create their own system configuration if they are working outside of SALAM paradigms, most designs that could be explored in SALAM will be able to utilize the Configurator. We introduce this improvement because we ran into these difficulties while working with gem5-SALAMv1. For example, we wanted to showcase the MobileNetV2 implementation as shown in Sec. 6.4.2 in gem5-SALAMv1, but we ran into challenges debugging and developing it due to its rapidly increasing complexity. Because of this, we were unable to feasibly finish the implementation in gem5-SALAMv1, but were easily able to iterate and design on this architecture in gem5-SALAMv2.

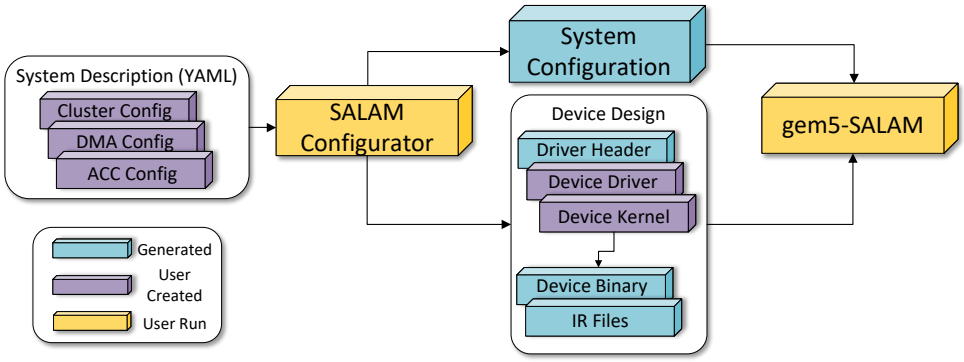


Figure 6.3: Overview of the gem5-SALAMv2 device configuration. By automating the system and header configuration we have removed the need for experience in programming gem5 and dramatically simplified the design process while simultaneously allowing far more complex models.

Designing a new accelerator-rich system in gem5-SALAMv2 starts with the development of a system structure description. This takes the form of a YAML file, visualized in Fig. 6.4, in which the user declares the names and types of devices they want to simulate in the SALAM framework, including accelerator clusters, accelerators, and DMAs. A user can assign memory devices to an accelerator by declaring a variable under the accelerator and providing its size and type. Supported memory types include multi-port scratchpads, register banks, memory streams, and caches. When the user is finished providing the structural outline of their accelerated system, they can invoke the SALAM Configurator, shown to the left in Fig. 6.3. This tool will automatically generate a valid

gem5 system simulation configuration file, a configuration file for SALAM-accelerated components, and a set of headers containing the memory map of the generated 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 both design and debug complex systems.

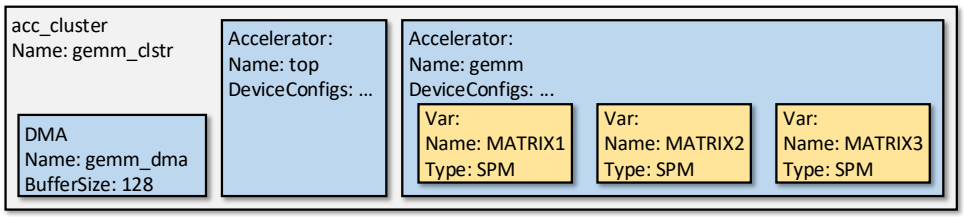


Figure 6.4: Example of the gem5-SALAMv2 device configuration for a accelerator cluster running a general matrix multiplication application.

The most immediately tangible benefit here 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 difficult to diagnose bugs that propagate throughout a design. Depending on how a user allocates their system's memory map, fixing a small misalignment or improperly sized memory-mapped register could require the realignment of dozens or hundreds of other memory-mapped addresses. In gem5-SALAMv1 this meant propagating changes across dozens of configuration files as well as accelerator descriptions and driver headers. In gem5-SALAMv2 this entire process is automated, requiring a user to change only one value in a single file, and saves hours of development and debug 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. gem5-SALAMv2's configuration generator is capable of generating all of the new devices, connections, and updated driver headers without any additional user input.

6.2.1.1 IR Generation and Static Elaboration

In gem5-SALAMv1, the user had to first create functional models of the target hardware accelerators as single in-lined functions before utilizing 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 control flow data graph (CDFG) and allocate the hardware resources needed to execute the application within the runtime simulator. gem5-SALAMv1's 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 come together to form the 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 internally impose caps on the scheduling window size, and were completely unable to handle constructs like functions since the LLVM IR's naming convention for values is not unique across function bounds (the value name %2 can represent multiple different values in multiple different functions). In gem5-SALAMv2 we instead preserve the IR structure as much as possible by leveraging the LLVM APIs directly and recreating many of the core structures in the LLVM IR. Whereas gem5-SALAMv1's parse was a single pass of the IR file with string parsing methods, gem5-SALAMv2 has a 3-stage approach. The first stage is simply leveraging the LLVM libraries to parse the text IR file into the LLVM::Module data structure. A LLVM::Module is composed of LLVM::Value elements representing everything from functions and basic blocks, to instructions and constants. The second stage of the gem5-SALAMv2 parse iterates over the LLVM module and creates a mapping of LLVM::Value elements to SALAM::Value elements. The third stage then initializes all of the SALAM values with their corresponding LLVM values. Internally the structure of SALAM values match their LLVM counterparts. A 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 (which are 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 a simple thing to identify with a basic string parse, complex expressions like "double* inttoptr (i32 268566720 to double*)" are less so. Additionally gem5-SALAMv1's pattern based lookup of dependencies did not provide a good way of differentiat-

ing between instructions, constant expressions, and constants during scheduling. This meant that scheduling lookups included searches for constants that never appeared in the scheduler, leading to unnecessary searches over potentially thousands of scheduling nodes. By mirroring LLVM's IR structure, gem5-SALAMv2 is better able to leverage IR-level insights and an understanding of the static execution graph to enable a more robust scheduling algorithm that is ultimately faster.

In short 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 that is capable to 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.

Overall the improved IR generation and parsing methodology in gem5-SALAMv2 eliminates the previous version's limitation of only simulating a single in-lined function. This improved approach allows gem5-SALAMv2 to now have the capability to model each function as an independent accelerator, which allows for the ability to fine-tune the application with configurable resources at the function level and provide the same power, area, and performance metrics as gem5-SALAM at a higher level of accuracy by providing evaluation metrics at the granularity of individual functions while preserving the benefits from clangs optimization passes, such as loop unrolling/vectorization and the removal of internal memory allocation.

The in-memory representation of the LLVM IR is leveraged for low-level optimizations and application analysis before it is mapped to the corresponding gem5-SALAMv2 static application graph. The corresponding gem5-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 gem5-SALAMv2 hardware profile. It is worth noting that SALAM allocated 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, SALAM 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 SALAM 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 gem5-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 [7] or MosaicSim [9] must generate multiple runtime traces to capture input data-dependent execution behaviors, gem5-SALAM is able to generate 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, SALAM is able to consistently model the same datapath as application inputs change, and even enable users to impose additional constraints on the simulated datapath. In contrast the Aladdin simulator dynamically alters its modelled datapath based on input data, as shown in the gem-SALAMv1 work [15], and MosaicSim bypasses datapath modeling entirely by treating accelerators like out-of-order CPU cores.

6.2.1.2 Hardware Model Generation

In gem5-SALAMv1, the hardware model was statically compiled as part of the SALAM simulation models. This meant that swapping to an alternate technology node required rebuilding the SALAM simulation models. The hardware resource model used in gem5-SALAMv2 is now dynamically generated from YAML configuration files that define a hardware profile as shown in Fig. 6.4. This change enables the user to create and define hardware profiles at the granularity of individual accelerators within the cluster. The hardware model generated from this profile defines how functional units and instructions are handled during the runtime simulation and the power, area, and performance characteristics of each resource. Fig. 6.3 shows how the hardware profile is used to generate the required gem5::SimObject and gem5-SALAMv2 source files when running the hardware configurator. The parameters in the hardware model are also added to the main config-

uration file for gem5-SALAMv2, allowing for a simple user interface that is dynamically updated to reflect the profile in use. This allows the user to redefine any parameter within the hardware model, including creating and linking customized instructions and functional units, and immediately begin a new simulation without the need for recompiling or rebuilding the system.

6.2.1.3 Power and Area Estimation

The hardware profile contains power and area profiles for common fixed and floating-point hardware functional units as well as variable-length registers. The device config limits the amount of each hardware functional unit that is in the system. By default, SALAM will model unique hardware elements for each operation in the parsed LLVM IR. By modifying the device config, the user can further constrain the allocation of hardware elements to explore features like functional unit re-use. Static power estimation uses the static accelerator CDFG to account for all functional units within the system, the simulation runtime, and the hardware profile to determine the leakage power lost in the system due to the functional units. The dynamic power used by the functional units is calculated each cycle for each active functional unit and is the combination of the switching and internal power dissipation as defined in the hardware profile as a function of the accelerator clock speed, which is defined in the device config.

Similarly, the LLVM IR as used in gem5-SALAM exposes the internal registers and their bit size, while the runtime engine tracks the read and write activity each cycle. This allows gem5-SALAM also to model the runtime energy consumption of internal data-path logic using the same method described for the functional units, where the static and dynamic power and area are calculated based on the single-bit register results obtained for the hardware profile.

6.2.1.4 Performance and Occupancy Analysis

gem5-SALAM also provides a variety of performance metrics to the user post-simulation. Within the device configuration, gem5-SALAM defines the cycle time that 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. 6.3. 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 units controller tracks and stores the scheduled amount of functional units each cycle during runtime. This information is passed to the power profiler to update the current dynamic energy usage of the system.

These additional data point combined with configurable hardware resources allows for a fine-grained analysis and exploration tool for exploring occupancy levels within the system. During the post-simulation, this data is used to determine the leakage power and area as well as 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. 6.3.

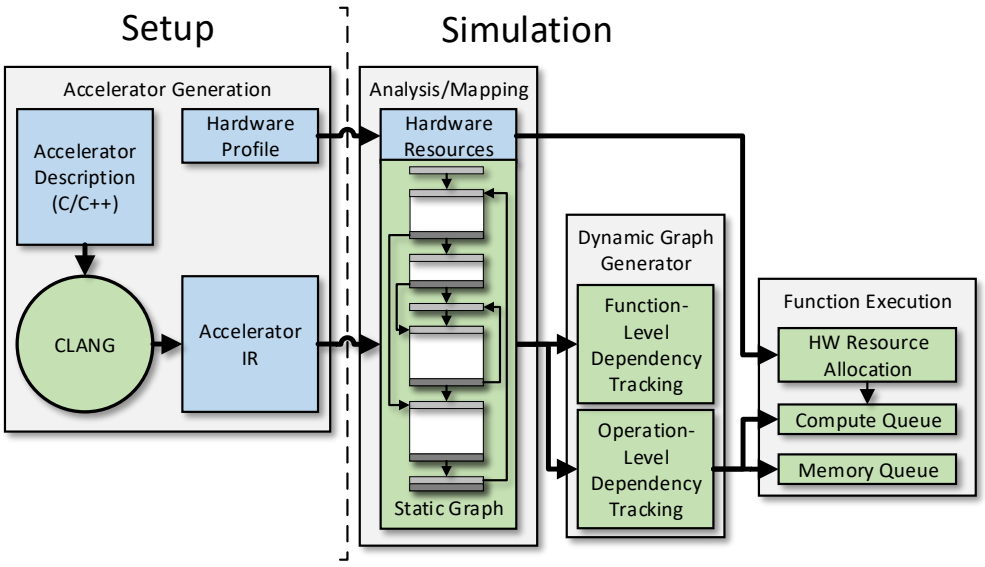


Figure 6.5: Overview of the LLVM runtime model present in gem5-SALAMv2. The automated configuration tools invoked during setup provide the constructs needed for elaboration to generate the static graph. This is then dynamically mapped to the allocated resources to perform a cycle accurate simulation of the application.

6.2.2 LLVM Runtime Engine

In order 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.

6.2.2.1 Event Scheduling and Dependency Tracking

gem5-SALAMv2 internally tracks the execution status of an accelerator pipeline through a set of custom event queues that correspond to events waiting to be executed or "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. In order to optimize search times, only the last instance of a dependency in the queues is tracked. Dependency tracking represents the largest overhead in SALAM's simulation model, with the event queues sometimes tracking dependencies across thousands of events.

One of the design considerations of the gem5-SALAMv2 upgrade was the reduction of scheduling and dependency tracking overheads, leading to a more than 2x performance gain on large applications. When an instruction has all of its runtime dependencies met, 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. In order to keep parallel lanes of the execution synchronized, users can optionally force the datapath to execute in a lockstep fashion. This prevents new execution events from launching until all previous events have been completed, regardless of runtime dependencies. Additionally, users can impose hardware resource limitations and restrictions that model functional units re-use in a datapath. When resource limits are imposed, an execution event will only launch when the necessary hardware resource is available.

6.2.2.2 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 like 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 that have 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 in order to access the gem5 memory system. These events commit 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.

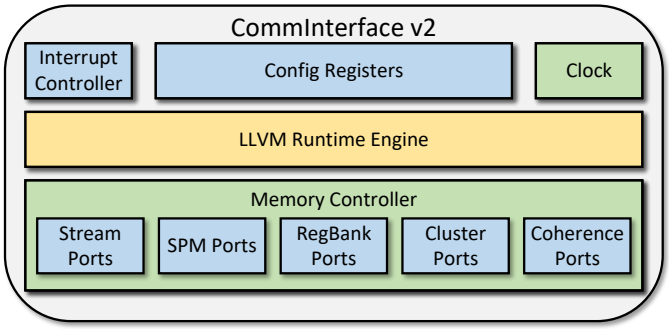


Figure 6.6: Overview of the communications interface in gem5-SALAMv2. This new unified interface handles all communications between gem5-SALAMv2 clusters, accelerators, memory objects, and the gem5 ecosystem.

6.2.2.3 Event Synchronization

Since the parallelism imposed by dependency-based event scheduling can introduce out-of-order execution and commit to the otherwise serialized IR, SALAM introduces a few important synchronization mechanisms. In addition to the lockstep execution mode, SALAM is able to automatically detect loop boundaries and impose barriers at the ends of loop execution. Additionally, SALAM introduces mechanisms for tracking and ensuring the correct order of reads and writes to memory.

6.2.2.4 Function Call Semantics and Advanced Scheduling

One of the most notable changes in SALAMv2's scheduler updates revolves around the handling of function calls. In general function calls introduce many challenges for both hardware modeling and simulation. For elaboration purposes, hardware design and simulation tools need to provide a policy for imposing hardware resource limitations, as well as track connections and dependencies across function bounds. Most other simulators bypass this issue entirely by allocating resources based on a pre-generated execution trace[6], modeling a general-purpose architecture with fixed resources [9], or by forcing users to manually inline functions like gem5-SALAMv1. In gem5-SALAMv2 we sought to directly solve the challenge by introducing a model hierarchy within each hardware accelerator. In gem5-SALAMv2 a function call is handled like a micro-pipelined functional unit within the datapath of the calling function. This enables a few interesting capabilities within gem5-SALAMv2.

First this design enables users to define custom operations within the LLVMRuntime without needing to directly implement 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 involves system-level overheads, like memory access, that cannot be statically determined or modeled correctly by execution traces.

Secondly, SALAMv2's function handling allows for finer grained control over datapath structure and parallelism. gem5-SALAMv2 internally tries to find the highest degree of both 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 is a functionality that does not currently exist outside of RTL simulation, or other handcrafted simulation models.

6.2.3 gem5 Integration and Scalable Full System Simulation

gem5-SALAMv2 expands on the system integration framework first presented in gem5-SALAMv1. Like in version 1 the integration of the LLVM runtime engine into gem5 revolves around the "Communications Interface" or CommInterface. The CommInterface is a gem5 simulation construct that provides system timing as well as 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 hierarchies of accelerators. The first of these changes was the rework 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 respectively. 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. 6.6. 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 implemented solely as a blocking access behavior, accelerators in v2 are now able to query the availability of data in streams prior to 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 are able to poll the connected scratchpads on the availability of data in order 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 a larger number of 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 as 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.

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[9] or execution traces[7], 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 accelerators that operate in a passive state until a change is observed in some other device (such as another accelerator) or they are provided with data on an input stream. To demonstrate this we present the test case shown in Fig. 6.7

In the example shown in Fig. 6.7 we construct a system consisting of three accelerators. Two of the accelerators represent some arbitrary workload that in this instance produce unsorted arrays of arbitrary length. The third accelerator represents a shared resource that sorts of the output of the other two accelerators and returns the sorted arrays. The shared acceleration consists of two functions, Arbitration and Sorting. The Arbitration function polls the status of input streams, shown in green, for tasks from the two input accelerators. When a task is detected from on of the input accelerators, the Arbitration task then launches the Sorting task to read the appropriate

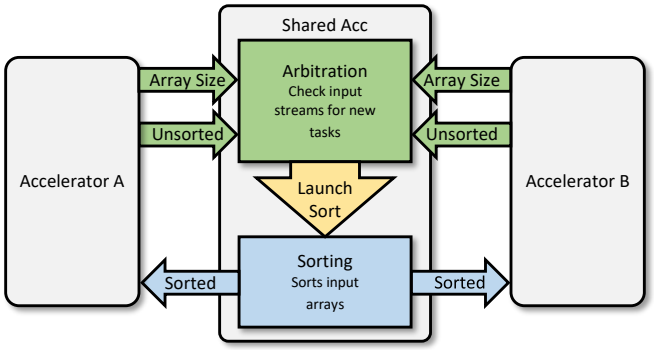


Figure 6.7: Shared accelerator resource scenario detailing the ability of gem5-SALAMv2 to have data driven accelerators.

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 in a similar manner to what is used in shared memory programming models. In gem5-SALAMv2 we can model a system in which both data producers can simply pass their outputs to the shared sorting accelerator via streams and the sorting accelerator launches its sorting task on its own 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.

6.3 Simulation Results and Metrics Validation

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.

6.3.1 Timing, Power, and Area Validation

gem5-SALAMv1 was developed around the LLVM 3.8 compiler tool chain and a far simpler internal elaboration and scheduling system. Given SALAMv2's significant overhauls to elaboration and runtime scheduling, as well as the significant changes in LLVM IR generation and structure that have arisen between LLVM 3.8 and LLVM 9.x, we have re-validated all of the accurate Machsuite benchmarks that gem5-SALAMv1 was validated using the most up to date mode of gem5-SALAMv1. The timing model of gem5-SALAM was validated using all of the accurate MachSuite [31] benchmarks against RTL models generated by Vivado HLS. 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. This validated hardware profile is included as the default configuration in gem5-SALAM, 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 updated LLVM creating slightly different IR.

Table 6.2: gem5-SALAM vs HLS power, area, and performance comparison.

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 6.2 provides absolute timing, power, and area validation metrics for all of 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 elabora-

Table 6.3: gem5-SALAM vs HLS power, area, and performance error percentages.

Bench	Performance Error% (cycles)		Power Error% (μW2)		Area Error% (mm2)	
	SALAMv1	SALAMv2	SALAMv1	SALAMv2	SALAMv1	SALAMv2
BFS	1.56%	1.48%	0.01%	0.00%	5.10%	5.15%
FFT	0.32%	0.11%	0.91%	2.18%	0.02%	0.11%
GEMM	0.32%	0.61%	2.64%	1.46%	3.17%	3.24%
MD-KNN	3.17%	3.16%	7.70%	5.17%	0.00%	3.56%
NW	0.19%	0.37%	5.15%	1.44%	1.25%	0.35%
Stencil2D	0.13%	0.19%	3.20%	4.45%	0.00%	6.37%
Stencil3D	2.22%	1.40%	-	-	-	-

tion in Table 6.3. For each test case we used the same clang optimizations for generating the LLVM 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 gem5-SALAMv1 and HLS.

Table 6.3 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%. Much like in gem5-SALAMv1, power estimations in gem5-SALAMv2 trend toward a slight over-estimation. The MD-KNN benchmark shows the highest power error due to heavier reliance on muxes and non-arithmetic operators. Variability in the power consumption of these operators leads to a slight overestimation of power requirements on average. In terms of area validation, on average gem5-SALAM is able to estimate chip area with an error of 2.24%. While error rates across timing, power, and area are comparable between gem5-SALAMv1 and gem5-SALAMv2, there are some conflating factors that result in discrepancies between their estimates. For one, LLVM IR generation has seen significant changes in terms of 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 did show notable differences in code structure. SALAMv2's internal scheduling is also generally more conservative than SALAMv1 in order to prevent memory consistency errors that could occasionally occur in gem5-SALAMv1. This results in higher timing estimates on average vs gem5-SALAMv1. Whereas gem5-SALAMv1 had a tendency to underestimate timings vs. RTL

simulations, gem5-SALAMv2 generally overestimates by a small margin.

6.3.2 FPGA System Validation

Table 6.4: FPGA vs gem5-SALAM system validation comparison.

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

For the purpose of system validation, we synthesized five of the benchmarks and executed them on a Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board, which has the XCZU9EG SoC chip. 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 6.4. 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.

Table 6.5 displays a similar trend to that seen in the comparison to RTL simulation for each version of gem5-SALAM. Positive error indicates when the simulation was faster, while negative errors indicated faster FPGA times. One notable difference is the timing between gem5-SALAM version is the increase in transfer times. This is due to gem5-SALAMv2 utilizing a top-level accelerator independent from the application to control data movement by default. This does slightly increase the transfer time overhead but provides far more flexibility in system design and is almost negligible

Table 6.5: FPGA vs gem5-SALAM system validation comparison results.

Bench	SALAM	SALAMv2	SALAM	SALAMv2	SALAM	SALAMv2
	Compute Error		Xfer Error		Total Error	
FFT	1.32%	2.16%	-2.14%	4.61%	0.98%	1.51%
GEMM	2.09%	2.14%	-1.65%	4.88%	1.65%	1.31%
MD-KNN	-3.16%	0.10%	4.87%	6.41%	-2.80%	0.38%
Stencil2D	-0.91%	3.56%	-2.76%	4.72%	-1.35%	1.57%
Stencil3D	-2.25%	3.36%	-0.34%	0.49%	-1.29%	1.43%

in effect is has on the total time. The biggest discrepancies in computation error can be attributed to benchmarks that operate on double-precision floating-point, whereas most of the other benchmarks operate on integer types. By default, gem5-SALAM 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.

6.3.3 Simulation Timing Comparison

One of the design goals of gem5-SALAMv2's runtime engine upgrade was to improve simulation performance. This upgrade required numerous modifications to SALAM's in-memory representation of CDFGs as well as the processes for tracking runtime dependencies across multiple execution graphs and scheduling runtime events. Table 6.6 shows a comparison of 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.

Table 6.6: 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

We examined the timing performance of both the datapath parse/setup and simulation time for accelerators in SALAMv1 vs SALAMv2. In gem5- 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. By leveraging the LLVM libraries for IR parsing in SALAMv2, these setup times are far more normalized. In general, the base cost of this approach leads to slower parse times in SALAMv2 for smaller applications. However, gem5-SALAMv2 is far more efficient at parsing larger IR files as well as files with more

complex data structures, as shown with GEMM and Stencil3D in Table 6.6.

Comparing simulation times between gem5-SALAMv1 and gem5-SALAMv2 shows a similar pattern. The base overheads of event scheduling and memory access in gem5-SALAMv2 are slightly higher due to the higher degree of complexity that gem5-SALAMv2 seeks to support. This means that for very small accelerators, gem5-SALAMv2 will generally run more slowly. This is most apparent in the SPMV application, which due to runtime loop dependencies, cannot be statically loop unrolled. The resulting small static CDFG constrains runtime parallelism and therefore also constrains SALAM's event scheduling windows. In contrast, the GEMM and Stencil3D benchmarks contain large amounts of loop unrolling, meaning that there are far more dependencies to track simultaneously in SALAM's event scheduling windows. In SPMV, the largest basic block for dependency tracking contains around 10 operations, whereas GEMM and Stencil3D contain blocks with more than 500 operations. Here the updates to gem5-SALAMv2 enable us to drastically reduce both dependency lookup times and compute event scheduling times. The end result is that while gem5-SALAMv2 may lose milliseconds in the execution of very small accelerators, it can improve simulation times by much larger factors in more complex designs. In Sec. 6.4 we explore systems built around neural network architectures. In the case of the MobileNetv2 design described in Sec. 6.4, gem5-SALAMv2 saw a more than 3x speedup over gem5-SALAMv1, which reduced simulation times for full-network runs by several hours.

6.4 Design Space Exploration

To demonstrate the increased flexibility that has been added for design space exploration, we explore different hardware architectures for two unique CNNs, with the goal being to showcase simulation flexibility rather than novel architectural insights. We initially showcase the ability to rapidly explore different architectural changes with LeNet-5 by exploring how the network behaves across three different architectures in Sec. 6.4.1. We also showcase a more complex MobileNetV2 design in Sec. 6.4.2 to demonstrate the ability to model large networks in gem5-SALAMv2.

6.4.1 Case Study: LeNet-5

Because LeNet-5 is significantly smaller than modern CNNs, we utilize this example as a way to showcase 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 differing hardware designs that we simply introduce as Designs A, B, and C. We introduce them in this manner to further exemplify that these are not novel architectural designs, but simply a vehicle to show the ease with which a designer can explore in SALAMv2. In the following sections, we discuss some of the significant benefits of using gem5-SALAMv2 over other simulators and present and perform analysis on each of the three designs.

6.4.1.1 System Setup and Configuration

We first started 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 section 6.5, with lockstep execution and memory hazard prevention enabled.

Each cluster contains a standard top-level accelerator to control communication between the gem5 system and cluster accelerators. The memory storage techniques used within the accelerator clusters are either scratchpad memories or stream buffers. Both of 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.

6.4.1.2 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 utilize three separate configurations on our three separate topographies to showcase how one can utilize the gem5-SALAMv2 tool-chain to explore new architectural designs.

We furthermore identify six internal dimensions to be used 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, so we created three variants of the LLVM IR used with each topology. The first variant doesn't contain

any loop unrolling, the second contains a fully unrolled kernel vector, and the third fully unrolls the kernel and the channel vectors. We defined these as "No Unroll", "Input Unroll", and "Output Unroll" respectively. We choose these three factorization factors highlighted below to simply 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.

With these three configurations defined, we move on to defining our three system topologies. All designs posses a top-level accelerator, that is further referred to as "Top". The Top utilizes varying levels of granularity in its control over DMA and synchronization events between network layers. We first present *Design A*, 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. 6.8. Our second configuration, *Design B*, connects accelerators together in a streaming-like fashion via scratchpad memories and a dedicated "Data Sync" accelerator, as shown in Fig. 6.9. The final configuration, *Design C*, has each neural net functional unit contain both convolution and pooling layers that have internally managed line buffers, as shown in Fig. 6.10.

6.4.1.3 Simulated Design A

Design A's implementation contains one of the simplest possible designs for a given CNN. At a high level, the design is simply disconnected accelerators that each have their memory accessed from a common DMA. The 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 full output feature

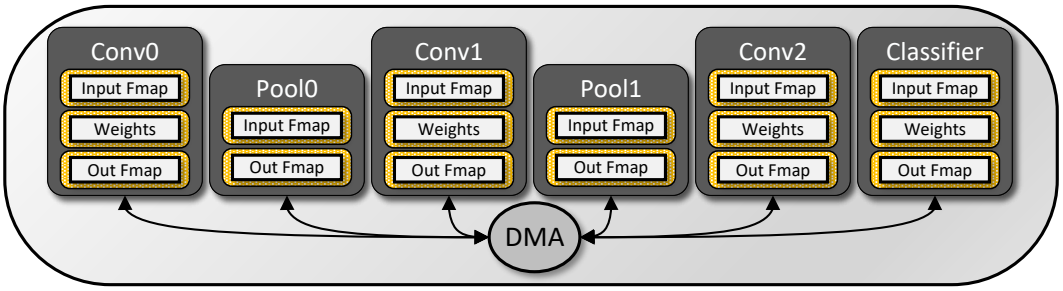


Figure 6.8: Design A - All accelerators in this architecture are shown connecting to a single DMA for all memory transfers.

map of the previous layer. This allows for a very simple synchronization methods, where the Top just runs each layer of the network successively and handles corresponding memory transfers.

6.4.1.4 Simulated Design B

One of the major benefits of design space exploration in gem5-SALAM is the capacity to explore compute parallelism across multiple accelerators via communication through the gem5 memory system. To take advantage of this in Design B, we connect accelerators together in a streaming-like fashion via scratchpad memories and a dedicated "Data Sync" accelerator, as shown in Fig. 6.9. This self-synchronization significantly reduces the control overheads that the Top introduces, and allows for overlapping execution.

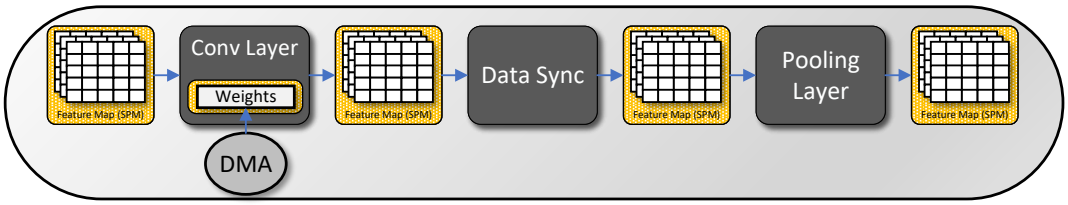


Figure 6.9: Design B Functional Unit - With the increased complexity of Design B, only a single network functional unit is shown that demonstrates how the Convolution, Data Sync, and Pooling layers are connected internally.

Streaming in CNNs is challenging due to significant imbalances in data production vs. data consumption. While the data may only be written once, it will be read many times due to shifting convolutional windows. While the Design A 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.

Desiring a more efficient data management solution than what is 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, albeit with some key differences. Each data sync accelerator is designed with the data access patterns of the the input and output layers in mind; allowing for data to be transferred between the 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 memory devices in the system. This capability enables each accelerator to perform execution based on the availability of data, without a reliance on a Top.

6.4.1.5 Simulated Design C

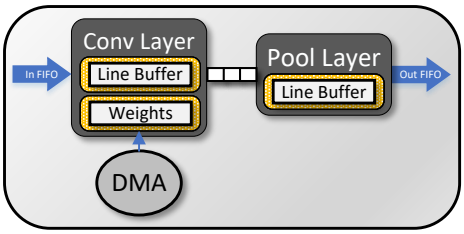


Figure 6.10: Design C Functional Unit - With the integration of data management into the convolution accelerator, there is now only a Convolution and Pooling layer at a functional unit level. The Convolution accelerator stores data from the input FIFO to the Line Buffer SPM to be utilized for the operation, all accelerators are interconnected with streaming FIFOs.

While Design B's architecture allows for increased amounts of spatial parallelism to be used, it is relatively inefficient due to the low reuse of allocated memory. Because gem5-SALAMv2 enables the use of interconnected stream buffers, we introduce Design C. This design uses streaming line buffers embedded in the convolution and pooling layers of the application to improve the scalability and efficiency over Designs A and B. While this architecture cannot support unrolling on the output channel because of the added line buffer, the design has a significantly higher amount of memory re-use and reduced SPM sizes across all network layers. This translates to a reduction of area and energy consumption while maintaining similar performance of the Massive Streaming architecture.

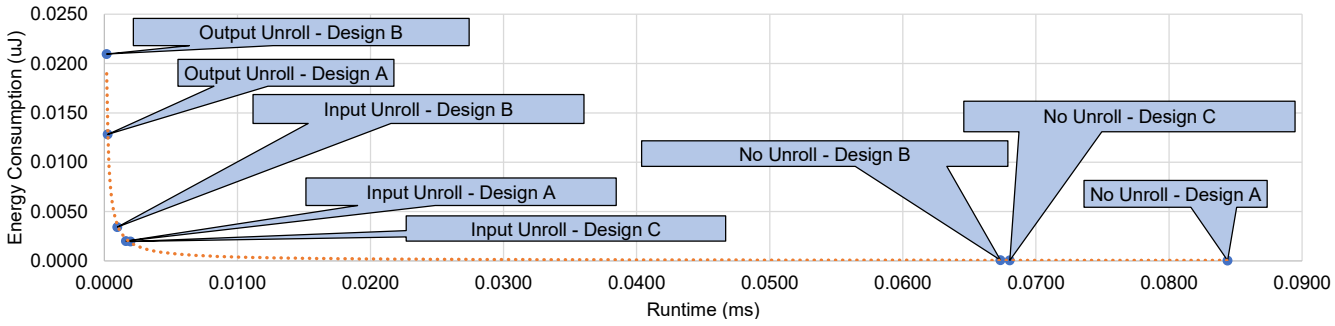


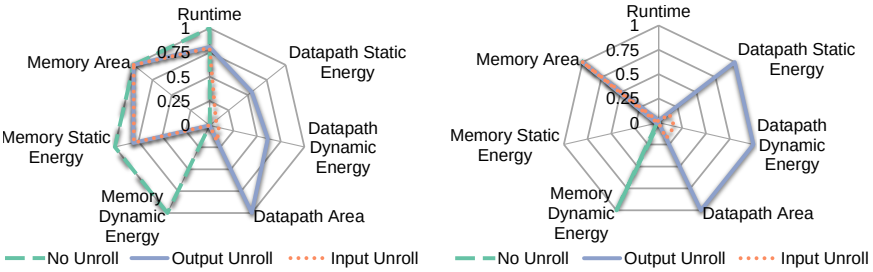
Figure 6.11: Total Data-Path Computational Energy Consumption VS Runtime Plot for LeNet-5 Test Configurations

6.4.1.6 LeNet-5 Results and Analysis

Because of the increased granularity that gem5-SALAMv2 allows for hardware statistics, we are able to generate Fig. 6.11 and Fig. 6.12. To create these figures, we ran each of our three separate architectures across each of our three separate configurations. Fig. 6.11 confirms the trends we expected to emerge in our computational performance from our design methodology. For each architecture, the implementation with no unrolling was energy efficient but is held back in performance due to the high overheads of sequential execution. Unrolling the kernel input increases the performance but also increases the energy consumption, with a diminishing return on performance and a huge increase in energy consumption when also unrolling the output channel. While these are not novel insights, we demonstrate the capability of gem5-SALAMv2 to accurately model expected behaviors.

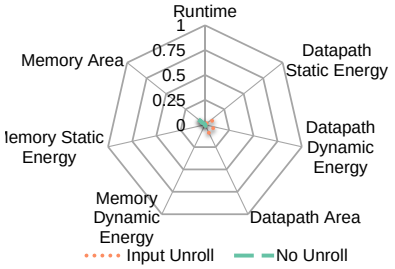
Additionally, we can gather from Fig. 6.11 that the internal streaming buffer used in Design C was the most energy-efficient design for computational performance, but it was only a marginal improvement over Design A. If we broader our scope and now look at all evaluation metrics for each test case in Fig. 6.12, it becomes immediately apparent that the system is defined far more by the energy consumption and area of the memory and not the datapath.

Fig. 6.12 provides an overview of the full system power, area, and performance metrics that can be obtained by gem5-SALAMv2. As expected, we see that the Design A, shown in Fig. 6.12(a) has a large static memory overhead due to inefficient use of the local DMA, and only transferring data upon full execution of each layer. Further analysis shows that the Design B, as shown in Fig. 6.12(b), has a comparable runtime but suffers from high dynamic energy memory usage with no



(a) Design A

(b) Design B



(c) Design C

Figure 6.12: 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.*

unrolling, high data-path energy consumption with kernel and output channel unrolling, and the largest area footprint in all configurations. We also see that Design C shown in Fig. 6.12(c) is the most effective implementation. While it is only marginally faster than Design B's input unrolled config, the area and energy consumption requirements are far lower than any of the other test applications.

The results of our design space exploration on LeNet-5 have provided insights that can help guide future network design decisions 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. 6.4.2.

6.4.2 MobileNetV2 Exploration

With the new design automation features that 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 tweak small architectural parameters with ease, these are still toy examples that do not convey the complexity and scale of systems that gem5-SALAMv2 supports. With this in mind, we also present MobileNetV2, an architecture that

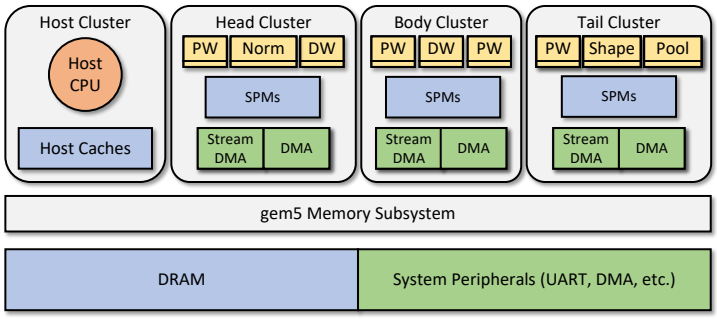
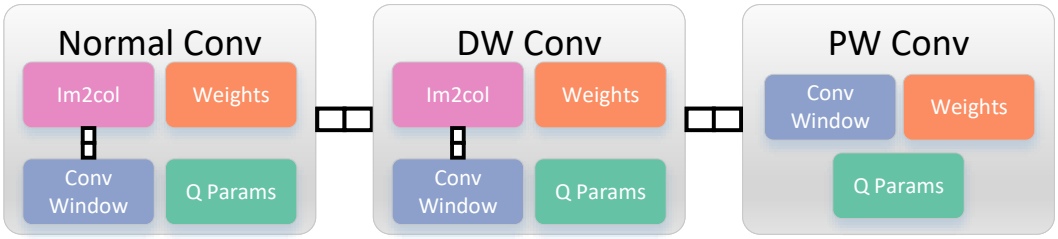


Figure 6.13: MobileNetV2 System Architecture - System architecture details for the MobileNetV2 design. The Head, Body, and Tail clusters are represented here, and show how the design interfaces with the gem5 system.

supports modern CNN features such as residual connections and separable convolutions to showcase how larger architectures can be implemented in gem5-SALAMv2.

With this increased complexity, it becomes unrealistic to fully map each layer of the network to an individual accelerator as we did in Sec 6.4.1. Because we have the ability to create isolated accelerator clusters in gem5-SALAMv2, we break the MobileNetV2 architecture into four core blocks of computation by assigning each block to an individual cluster. Each of these compute clusters contains unique structures of the network, with the head, tail, and classifier being single-use clusters. Because of the dynamic configurability of accelerators in gem5-SALAMv2, we are able to create a single body cluster that is capable of being re-used. The system-wide architecture is shown in 6.13, and the Head, Body, and Tail clusters can be seen in further detail in Fig. 6.14. Notably, the Classification cluster is left out of both figures due to the very simple nature of its design, but is present in the implementation of the network.

Fig. 6.13 showcases the complex design of MobileNetV2 at a system level. We show this to describe the system, but also to 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 hardware for applications such as MobileNetV2 at a significantly faster pace, as one no longer has to create and maintain the gem5 system configuration. Another addition that helps with organization, is the ability to easily partition accelerators into isolated accelerator



(a) Head Cluster

(b) Body Cluster

(c) Tail Cluster

Figure 6.14: MobileNetV2 Cluster Architecture - Individual cluster configurations for the Head, Body, and Tail. The Head (a), shows a detailed overview of the Normal, Depthwise, and Pointwise convolutional functional units (FUs). The Body (b) and Tail (c), use a summarized version of each of these FUs to demonstrate how they are configured at the cluster level.

clusters, which is useful for organizing and maintaining designs.

Because all clusters utilize either Normal, Depth-wise, or Point-wise convolutions, we create functional units (FUs) for each of these essential operations. Due to the producer-consumer disparities in CNNs discussed in Sec. 6.4.1, we create an internal Im2col accelerator for both the Normal and Depth-wise convolution FUs. These process data from an input FIFO stream to prepare the convolution window for the main compute accelerator. 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 its own 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 is configured by the Top. Weights and quantization parameters are transferred to their respective SPMs before accelerators begin computation.

The Head cluster, shown in Fig. 6.14(a), is responsible for the first two layers of the network, a normal convolution, and a unique inverted residual block (IRB). We describe the Body in Fig.

Table 6.7: MobileNetV2 96x96 Sim time and latency

	$\alpha = .35$		$\alpha = .75$		$\alpha = 1$	
Cluster	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

6.14(b), this describes the construction of the Body cluster, which is where the majority of the network's computation takes place. The Body contains an IRB and support for residual connections. As mentioned previously, we utilize SALAMv2's support for configurable accelerators to re-use the body cluster in a sequential fashion to process each subsequent network layer. Furthermore, Fig. 6.14(c), shows the Tail of the network which embeds the features for the classifier and computes an average pool. Finally, the classifier simply 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 6.8, there are significant changes in how much computation and memory must be utilized to perform operations on a frame. This allows us to perform an in-depth analysis of our proposed architecture.

Table 6.8: "MobileNetV2 Network Complexity for 96x96 Input"

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

With the data that we are able to gather in gem5-SALAMv2, we run all three of these configurations and record the results in Table 6.7. We are able to 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 are able to 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 take 94% of the execution of the cluster. Another solution that would be 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.

Table 6.9: Runtime Comparison of MobileNetV2 on SALAMv1 and SALAMv2

Simulated | 96x96 | $\alpha = 0.35$	V1 Sim Time (m)	V2 Sim 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

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 6.9, that there are significant performance benefits that enable a more rapid exploration of large-scale architectures. Notably, we see a trend confirmed that 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 further. However, ultimately gem5-SALAMv1 was not able to run the fully-scaled network ($\alpha = 1$) end-to-end in a single run for a proper comparison.

CHAPTER 7: CONCLUSION

This work started with the simple idea that application and domain-specific processors need application and domain-specific memory requirements and optimizations. Identifying and designing memories around those requirements, however, required both application-specific insights and a platform in which to implement them. The rich data and control flow information present in the LLVM IR presented the perfect starting point for this research. LLVM IR provides the capacity to distill intrinsic algorithmic characteristics from high-level programming languages that is at a low enough level to be translatable to hardware. At the same time it sits high enough above the specific implementations of hardware devices to serve as the baseline for a set of generalized frameworks.

Analysis of LLVM IR provided the framework for the first technical thrust of this work, automated memory analysis of application-specific accelerators to enable memory optimization. Locality Aware Memory Assignment and Tiling[12] based on LLVM IR analysis provides a mechanism for analyzing memory access patterns on a per-variable basis and provides recommendations for the development of an application-specific memory hierarchy. These recommendations led to a 45% improvement to the power-stall product, a metric which sought to jointly minimize the average latency and power consumption associate with memory access, for the evaluated applications from MachSuite. The codebase for this work can be found on GitHub at https://github.com/TeCSAR-UNCC/LocalityAwareMemoryAssignment.

The IR analysis approach presented in Locality Aware Memory Assignment and Tiling was also repurposed to aid in the memory path synthesis of OpenCL applications on FPGAs[13]. High-level synthesis tools for OpenCL were also built upon the LLVM framework, however they currently make no attempts to optimize the memory hierarchy without explicit user guidance. The LLVM analysis presented in this work leveraged an understanding of memory dependencies in OpenCL applications to identify memory accesses within the applications that could be decoupled from the compute portion of the datapath. While this memory decoupling optimization does incur a small

resource overhead (increasing power consumption by around 7%), it led to an average compute speedup of 2x and 40% reduction in energy consumption on the tested workloads. Furthermore, unlike other memory optimization techniques like memory streaming and tiling, the decoupling approach is far less restrictive in the types of memory access that can be optimized and does not require rewriting the compute component of the application to support a particular memory access pattern. The memory decoupling is orthogonal to other memory optimizations and is automated in a way that can be directly integrated into existing HLS tools for OpenCL applications.

The biggest obstacle to further research on memory optimization was a lack of frameworks for evaluating more complex memory designs. In performing memory evaluations for application-specific accelerators using Locality Aware Memory Assignment and Tiling, many significant limitations were found in the gem5-Aladdin framework being used as a test bed. Existing frameworks at the time like gem5-Aladdin and PARADE lacked the support for memory systems extending beyond simple DMA-driven scratchpads and caches. Applications and domains with more irregular data access patterns required more specialized memory interfaces that what was provided. To address that need this work presented the original gem5-SALAM simulator[14, 15]. The original gem5-SALAM extended the widely-used gem5 full system simulator to support the integration of custom, application-specific accelerators. Unlike other existing solutions that leverage static timing estimates and runtime execution traces for managing functionality and timing, gem5-SALAM directly executes the LLVM IR CDFG of an application with timing, power, and area constraints determined at the operation level. gem5-SALAM's dynamic graph execution mechanism is the first of its kind within the scope of event-driven simulation and provides mechanisms for handling runtime data-dependent behaviors of applications that other pre-RTL simulators do not offer. The initial release of gem5-SALAM supported communication between accelerators and other system element via shared scratchpad memories and caches. The original gem5-SALAM release is available on GitHub at https://github.com/TeCSAR-UNCC/gem5-SALAM/releases/tag/v1.0.

In trying to map more complex applications to the original gem5-SALAM, several key issues arose. The simulation of an end-to-end neural network model like MobileNetv2 required memory mapping of hundreds of memory interfaces, advanced inter-device communication mechanisms, and an extended LLVM IR scope that was not supported in gem5-SALAMv1. This prompted the

design of gem5-SALAMv2 and a fundamental rethinking of the user-side interface for simulation. gem5-SALAMv2 formalizes a series of design abstractions for simulation that greatly simplify the design and integration of new simulation models. The operation, task, and architecture abstractions provided by gem5-SALAMv2 provide mechanisms for designers to more directly integrate classical hardware design principles into the design of simulation models. Coupled with new automation for generating memory maps and device configurations, users can develop complex systems of hundreds of custom hardware models in a fraction of the time and with far less need for debugging than other existing design flows. The updates to the system integration of SALAM simulation objects through the revised architecture abstraction provide new communication mechanisms to users, that previously only existing in RTL design flows. These works have been submitted as part of a journal revision in ACM Transactions on Architecture and Code Optimization. Lastly the redesign of the LLVM IR runtime engine boasts support for a broader scope of the LLVM IR than gem-SALAMv1, while also being easier to extend for new custom instructions. At the time of writing, the work on simulation abstractions in under preparation for submission to IEEE Computer Architecture Letters. gem5-SALAMv2 is currently being used both in teaching and cutting edge architecture research like domain-specific cache architectures for sparse data structures. Additionally gem5-SALAMv2 is currently being prepared for integration into the main release of gem5. The most recent release of gem5-SALAMv2 is available on GitHub at https://github.com/TeCSAR-UNCC/gem5-SALAM.

One of the most immediate impacts of gem5-SALAM has been its usage as a teaching tool in the classroom. gem5-SALAM has been integrated as a teaching aid for the computer architecture course at Simon Fraiser University. Using gem5-SALAM, students were able to design their own hardware accelerators for small neural networks and explore the process of integrating them into a modern SoC in simulation. gem5-SALAM is ideal for this kind of work due to its simple interface and open source nature. Feedback from this partnership has driven further improvements in gem5-SALAMv2's development.

In summary, this work makes three key contributions:

Automated Memory Analysis: The Locality Aware Memory Assignment and Tiling tool provides variable-level insights for memory optimization of hardware accelerators that can be applied

in other design and synthesis tools. With some small adjustments, this tool can also be used to automate hardware pre-fetching for OpenCL HLS on FPGAs.

New Simulation Tools: gem5-SALAM provides an innovative and highly customizable solution for pre-RTL simulation built around a unique LLVM execution engine. Its integration into the gem5 ecosystem with its configurable system interface and other new system design elements enables design space exploration for systems that cannot currently be simulated in any other pre-RTL simulator. Additionally the new automation tools added in gem5-SALAMv2 enable rapid prototyping with significantly lower development overhead than other simulation solutions.

New Simulation Abstractions: The new simulation abstractions of *operation*, *task*, and *architecture* built into gem5-SALAMv2 offer a new approach to designing and integrating new hardware designs into event-driven simulation. These abstractions separate the design concerns of timing and functionality to simplify the design task and reduce non-recurring engineering costs when exploring hardware designs.

In order to foster further development, and support integration into both research and teaching, all of these contributions have been made freely available via GitHub at https://github.com/TeCSAR-UNCC.

Bibliography

[1] "Google tpu." `https://cloud.google.com/tpu/docs/tpus`, 2018.

[2] H. Tabkhi, R. Bushey, and G. Schirner, "Function-level processor (flp): A high performance, minimal bandwidth, low power architecture for market-oriented mpsocs," *IEEE Embedded Systems Letters*, 05/2014 2014.

[3] "Versal: The first adaptive compute acceleration platform (acap)," Tech. Rep. WP505, Xilinx, 2018.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, 2012.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] S. Rogers and H. Tabkhi, "Locality aware memory assignment and tiling," in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), pp. 130:1–130:6, ACM, 2018.

[13] A. A. Purkayastha, S. Rogers, S. A. Shiddibhavi, and H. Tabkhi, "Llvm-based automation of memory decoupling for opencl applications on fpgas," *Microprocess. Microsyst.*, vol. 72, feb 2020.

[14] 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.

[15] 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.

[16] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.

[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] T. Nowatzki and K. Sankaralingam, "Analyzing behavior specialized acceleration," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 697–711, ACM, 2016.

[22] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *HPCA*, pp. 503–514, Feb 2011.

[23] 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.

[24] 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.

[25] J. Cong and et al., "Architecture support for accelerator-rich cmps," in *Design Automation Conference (DAC)*, pp. 843–849, 2012.

[26] F. Piovezan, T. E. M. Crocomo, and L. C. V. dos Santos, "Cache sizing for low-energy elliptic curve cryptography," in *29th Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2016.

[27] G. Wang, L. Ju, Z. Jia, and X. Li, "Data allocation for embedded systems with hybrid on-chip scratchpad and caches," in *IEEE International Conference on High Performance Computing and Communications*, pp. 366–373, 2013.

[28] J. Sancho and D. Kerbyson, "Analysis of double buffering on two different multicore architectures: Quad-core opteron and the cell-be," in *International Symposium on Parallel and Distributed Processing*, pp. 1–12, April 2008.

[29] L. Wu and W. Zhang, "Cache-aware spm allocation algorithms for hybrid spm-cache architectures," in *Sixteenth International Symposium on Quality Electronic Design*, pp. 123–129, March 2015.

[30] H. Tabkhi, M. Sabbagh, and G. Schirner, "An efficient architecture solution for low-power real-time background subtraction," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pp. 218–225, 2015.

[31] 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.

[32] D. Chen and D. P. Singh, "Fractal video compression in opencl: An evaluation of cpus, gpus, and fpgas as acceleration platforms," in *18th Asia and South Pacific Design Automation Conference*, 2013.

[33] J. Andrade, G. Falcão, V. Silva, and K. Kasai, "Flexible non-binary ldpc decoding on fpgas," in *IEEE International Conf. on Acoustics, Speech, and Signal Processing - ICASSP*, vol. 1, pp. 1–5, 2014.

[34] Q. Gautier, A. Shearer, J. Matai, D. Richmond, P. Meng, and R. Kastner, "Real-time 3d reconstruction for fpgas: A case study for evaluating the performance, area, and programmability trade-offs of the altera opencl," in *International Conference on Field-Programmable Technology (FPT)*, 2014.

[35] A. A. Purkayastha, S. A. Shiddhibhavi, and H. Tabkhi, "Taxonomy of spatial parallelism on fpgas for massively parallel applications," (Washington DC), IEEE, International System on Chip Conference (SoCC), 2018.

[36] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, 2016.

[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, 2009.

[38] O. Segal, N. Nasiri, M. Margala, and W. Vanderbauwhede, "High level programming of fpgas for HPC and data centric applications," in *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*, 2014.

[39] J. Zhang, H. Tabkhi, and G. Schirner, "Ds-dse: Domain-specific design space exploration for streaming applications," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 165–170, March 2018.

[40] H. Tabkhi, R. Bushey, and G. Schirner, "Function-level processor (flp): Raising efficiency by operating at function granularity for market-oriented mpsoc," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pp. 121–130, June 2014.

[41] A. Momeni, H. Tabkhi, Y. Ukidave, G. Schirner, and D. R. Kaeli, "Exploring the efficiency of the opencl pipe semantic on an FPGA," *SIGARCH Computer Architecture News*, 2015.

[42] S. Lee, J. Kim, and J. S. Vetter, "Openacc to fpga: A framework for directive-based high-performance reconfigurable computing," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 544–554, May 2016.

[43] Y. Ukidave, C. Kalra, D. R. Kaeli, P. Mistry, and D. Schaa, "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus," in *26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, October 22-24, 2014*, 2014.

[44] S. O. Settle, "High-performance dynamic programming on fpgas with opencl," 2013.

[45] "Sdaccel environment optimization guide," in *SDAccel Environment Optimization Guide*, p. 63:64, 2017.

[46] M. Z. Hasan and S. G. Sotirios, "Customized kernel execution on reconfigurable hardware for embedded applications," *Microprocessors and Microsystems*, 2009.

[47] E. Nurvitadhi, J. C. Hoe, S.-L. L. Lu, and T. Kam, "Automatic multithreaded pipeline synthesis from transactional datapath specifications," in *Proceedings of the 47th Design Automation Conference*, DAC '10, ACM, 2010.

[48] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded pipeline synthesis for data-parallel kernels," in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, 2014.

[49] R. J. Halstead, J. Villarreal, and W. Najjar, "Exploring irregular memory accesses on fpgas," in *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '11, ACM, 2011.

[50] R. J. Halstead and W. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, 2013.

[51] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, 2015.

[52] K. Turkington, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Outer loop pipelining for application specific datapaths in fpgas," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2008.

[53] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, "Single-dimension software pipelining for multi-dimensional loops," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004.

[54] M. Z. Hasan and S. G. Ziavras, "Customized kernel execution on reconfigurable hardware for embedded applications," *Microprocessors and Microsystems - Embedded Hardware Design*, 2009.

[55] E. Nurvitadhi, J. C. Hoe, S. Lu, and T. Kam, "Automatic multithreaded pipeline synthesis from transactional datapath specifications," in *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, 2010.

[56] R. J. Halstead and W. A. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2013, Montreal, QC, Canada, September 29 - October 4, 2013*, 2013.

[57] J. E. Smith, "Decoupled access/execute computer architectures," in *25 Years ISCA: Retrospectives and Reprints*, pp. 231–238, ACM, 1998.

[58] T. Chen and G. E. Suh, "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 46:1–46:12, IEEE Press, 2016.

[59] S. Cheng and J. Wawrzynek, "Architectural synthesis of computational pipelines with decoupled memory access," in *FPT*, pp. 83–90, IEEE, 2014.

[60] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A reconfigurable computing approach for efficient and scalable parallel graph exploration," in *23rd IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2012, Delft, The Netherlands, July 9-11, 2012*, 2012.

[61] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," 2000.

[62] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 2015.

[63] B. Panda and S. Balachandran, "Hardware prefetchers for emerging parallel applications," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, 2012.

[64] T. Kim, D. Zhao, and A. V. Veidenbaum, "Multiple stream tracker: A new hardware stride prefetcher," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, 2014.

[65] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, 2004.

[66] D. Bernstein, D. Cohen, and A. Freund, "Compiler techniques for data prefetching on the powerpc," in *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, PACT '95, 1995.

[67] G. Marin, C. McCurdy, and J. S. Vetter, "Diagnosis and optimization of application prefetching performance," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, 2013.

[68] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," *SIGARCH Comput. Archit. News*, 1991.

[69] D. F. Zucker, R. B. Lee, and M. J. Flynn, "Hardware and software cache prefetching techniques for mpeg benchmarks," *IEEE Transactions on Circuits and Systems for Video Technology*, 2000.

[70] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, 1990.

[71] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The performance of runtime data cache prefetching in a dynamic optimization system," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[72] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely, "Quantifying locality in the memory access patterns of hpc applications," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pp. 50–50, Nov 2005.

[73] "Xilinx opencl." `http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`, 2015.

[74] "Altera sdk for opencl." `http://www.altera.com/literature/lit-opencl-sdk.jsp`, 2015.

[75] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for hls," in *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, (New York, NY, USA), pp. 43:1–43:6, ACM, 2017.

[76] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus: (abtract only)," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, (New York, NY, USA), pp. 288–288, ACM, 2018.

[77] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: A binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, 2004.

[78] 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 *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 469–480, ACM, 2009.

[79] A. Sedaghati, M. Hakimi, R. Hojabr, and A. Shriraman, "X-cache: A modular architecture for domain-specific caches," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, (New York, NY, USA), p. 396–409, Association for Computing Machinery, 2022.