To Cache or not to Cache? Exploring the Design Space of Tunable, HLS-generated Accelerators.

Claudio Barone
Pacific Northwest National Laboratory
Richland, WA, USA
claudio.barone@pnnl.gov

Rishika Kushwah
Pacific Northwest National Laboratory
Richland, WA, USA
rishika.kushwah@pnnl.gov

Ankur Limaye
Pacific Northwest National Laboratory
Richland, WA, USA
ankur.limaye@pnnl.gov

Vito Giovanni Castellana
Pacific Northwest National Laboratory
Richland, WA, USA
vitogiovanni.castellana@pnnl.gov

Giovanni Gozzi
Politecnico di Milano
Milano, Italy
giovanni.gozzi@polimi.it

Michele Fiorito
Politecnico di Milano
Milano, Italy
michele.fiorito@polimit.it

Fabrizio Ferrandi
Politecnico di Milano
Milano, Italy
fabrizio.ferrandi@polimi.it

Antonino Tumeo
Pacific Northwest National Laboratory
Richland, WA, USA
antonino.tumeo@pnnl.gov

ABSTRACT

In recent years, hardware specialization has become the dominant strategy for accelerating the performance of emerging applications. However, designing and integrating custom hardware is a complex, time-consuming, and costly process. High-Level Synthesis (HLS) tools mitigate these issues by designing, prototyping, and deploying hardware accelerators on reconfigurable devices (e.g., FPGAs) attached to the computing systems. The conventional HLS tools aim at exploiting instruction level parallelism to achieve superior performance and generate accelerators with fine-grained memory accesses. The limited on-device memory on FPGA necessitates frequent direct access to external memories to satisfy the data requirements of these applications. Thus, improving memory bandwidth utilization is challenging for the HLS flows, and requires rethinking of the accelerators' memory subsystem. In this paper, we present a synthesis methodology that seamlessly introduces custom caches into the HLS-generated designs. The key feature of our approach is the ability to fine-tune cache parameters specific to the accelerator. This is crucial since different workloads exhibit varying spatial and temporal locality, leading to different performance trade-offs. Our methodology also allows for efficient design space exploration without requiring user modifications to the input specification. We demonstrate the practical application of our methodology by studying the effects of various cache configurations on selected kernels' performance. Based on our study, we also present a heuristic that guides users to fine-tune cache parameters. Our study shows that fine-tuning caches can improve performance by up to 3.5× with 13% resource overhead, compared to the conventional non-cached designs.

1 INTRODUCTION

In recent years, machine learning (ML), artificial intelligence (AI), and big data analytics have become the primary drivers for designing custom accelerators [8]. Applications from these domains combine unique requirements for memory and computation: they process exponentially growing amounts of data and employ complex computational patterns (e.g., dense and sparse matrix multiplications, graph traversals, etc.) [10]. Hence, designing domain-specific accelerators for these applications requires optimizing data/memory access as well as computation.

High-level synthesis (HLS) is an automated design process that converts specifications described in high-level programming languages (e.g., C, C++, Python, etc.) into implementations in hardware description languages (HDLs). The conventional HLS tools, both commercial (e.g., AMD Xilinx Vitis, Mentor Catapult C) and open-source (e.g., Bambu), optimize for the computational complexity of the input specifications and generate accelerators operating on the finite state machine with datapath (FSMD) model. Thus, for the HLS-generated accelerators, each load or store memory operation directly corresponds to a memory transaction, resulting in fine-grained memory accesses.

Fine-grained memory accesses coupled with limited on-chip memory results in frequent access to external memories. When the HLS-generated accelerators are integrated into a system-on-chip (SoC), accessing external memory, such as dynamic random access memory (DRAM), has a high latency cost. The accelerator's computation is also stalled until the data movement between the on-chip and external memories is complete, resulting in performance degradation. Moreover, various computational units often access external memories through a shared bus, potentially congesting the interconnect. An expert hardware designer would be required to architect efficient solutions to mitigate the effects of external memory latency. However, as the specifications grow more complex, it is necessary to revamp the HLS methodologies to shorten the accelerator design cycle time. *Next-generation HLS tools need to generate accelerators with optimized memory subsystems.*

Conventional commercial HLS tools provide limited ways to improve external memory access costs, such as leveraging burst memory transactions and implementing on-chip burst buffers. However, these solutions require expert HLS tool users to take advantage of them by restructuring code or instantiating specific storage structures. Moreover, these solutions are useful only for a few specific memory access patterns.

In this paper, we propose an HLS methodology that can generate accelerators with integrated caches and burst memory transaction capabilities. Using caches to reduce memory access latency by exploiting spatial and temporal locality is a well-studied technique for general-purpose processors. Our HLS methodology requires only adding specific `pragma` directives in the input specification header without the need to modify the behavioral description. The HLS process generates highly specialized accelerators that only implement the behavior specified in the high-level programming language. Hence, it is critical to provide the ability to quickly explore cache parameters to adapt the cache to the computational and memory access patterns of the accelerator and identify the best trade-offs between performance and resource utilization.

We have integrated this methodology into the open-source HLS tool Bambu [7]. To the best of our knowledge, Bambu is the only HLS tool that implements an approach to generate configurable accelerator caches. We describe how we implemented the methodology in the tool, discuss the supported parameters, and demonstrate the flexibility of our approach by exploring cache configurations for accelerators generated by a variety of algorithms with different computational and memory access patterns.

The paper proceeds as follows. Section 2 provides an overview of the methodology used by HLS tools to translate high-level descriptions into HDL, focusing on how memory operations are implemented. Section 3 describes both the internal architecture of the integrated caches and their instantiation within the custom accelerator. Section 4 describes the evaluation methodology. Section 5 presents our heuristic for exploring the design space and the evaluation of our solution. Section 6 presents our conclusions.

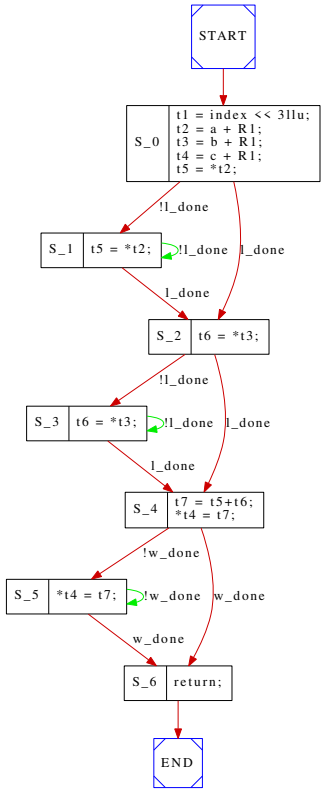

Figure 1: Example State Transition Graph of the specification in Listing 1

2 HIGH LEVEL SYNTHESIS FLOW

High-Level Synthesis (HLS) involves generating the hardware description of a behavioral specification, typically written in common programming languages such as C and C++. The HLS process is organized into a sequence of interrelated steps, akin to a software compilation toolchain. The initial step in the pipeline lowers the input specification to a set of internal representations, capturing various characteristics such as control and data dependencies among operations. This process often leverages software compilers and applies target-agnostic optimizations such as constant propagation and common subexpression elimination.

The subsequent steps encompass the core HLS tasks: scheduling, resource allocation, and resource binding. These tasks can be performed using various algorithms and in different sequences. Scheduling assigns each operation to a specific control step. The scheduled specification is typically represented as a State Transition Graph, which is later translated into a Finite State Machine controller in the generated design. Resource allocation identifies the registers and functional units required for each operation and determines their quantities. Resource binding associates operations with specific instances of resources, which may be shared across multiple operations. Finally, interconnection binding connects the various resources to each other and to memory interfaces.

At this stage, the circuit is generally described using a graph-based internal representation. If so, the final step, netlist generation, translates the graph representation into a hardware description language (HDL) file.

```
1 void foo(uint64_t* a, uint64_t* b,
2          uint64_t* c, uint64_t index) {
3   c[index] = a[index] + b[index];
4 }
```

Listing 1: Simple example specification

Listing 1 provides a simple example specification in C, while Figure 1 illustrates the corresponding State Transition Graph (STG). Load and store operations access external memory with latencies unknown at compile time. Consequently, once these operations are issued, execution stalls until a done signal on the memory interface is asserted. This results in the introduction of one idle state per memory operation (S1, S3, and S5). The depicted STG represents just one possible scheduling. Alternative architectural configurations, such as an accelerator with access to multiple memory ports, might generate different graphs with fewer idle states. However, the presence of idle states can never be entirely eliminated.

3 MEMORY CONTROLLER ARCHITECTURE

While the general methodology is not dependent on a specific protocol for the memory interface, our work focuses on interfaces exchanging data through an AXI [3] bus. This protocol, including both its full version and its subset AXI-Lite, is well-known among designers and widely utilized in commercial products (e.g., Xilinx

devices) to enable communication across system components for both control and data movement The AXI protocol supports burst transfers, which allow for transferring large blocks of contiguous memory in a single transaction. This capability is ideal for use with caches, enabling the reading or writing of entire cache lines at once.

HLS tools, such as Bambu, support automatic data exchange handling via AXI through the use of `pragma` directives. These directives provide the tool with essential information that cannot be inferred, such as the protocol for data access, the number of memory ports available to accelerators, and the specific ports each function parameter should use for data access.

Our solution leverages the existing `pragma` support in Bambu to identify different memory channels available to the accelerator and introduces a new set of `pragmas` used to introduce caches in the design according to the specified configuration.

It is important to note that both sets of `pragma` directives do not require any code restructuring from the user. Unlike the solution proposed by Xilinx tools, our approach allows any functional application, regardless of its memory access patterns, to incorporate caches without modification. Additionally, `pragma` directives only need to be added at the top function level, not at the level of individual memory accesses. This simplifies the synthesis process, as users do not need to search through the code to identify and optimize access patterns, nor do they need to ensure the code meets specific constraints.

In our experiments, we evaluate two distinct memory controller architectures, differing only in the presence or absence of the cache `pragma` in the high-level specification: the Standard AXI memory interface and the AXI Cache memory interface.

3.1 Standard AXI

The Standard AXI architecture is the default memory controller for AXI memory interfaces in Bambu. This controller is straightforward to instantiate, requiring only a single `pragma` directive for each pointer in the top function signature to access data through an AXI bus. As shown in Listing 2, *a*, *b*, and *c* correspond to the parameters of the `mmult` function, and the values assigned to the *bundle* represent the names of the AXI ports mapped to these parameters. Multiple parameters can share the same AXI port by using the same bundle name, or they can be separated by specifying different names. In this example, two separate memory controllers are generated: the first, called `gmem0`, handles memory accesses related to *a*, while the second, called `gmem1`, manages requests for the data of both *b* and *c*.

```
#pragma HLS_interface a m_axi direct bundle=gmem0
#pragma HLS_interface b m_axi direct bundle=gmem1
#pragma HLS_interface c m_axi direct bundle=gmem1

void mmult(int *a, int *b, int *c)
```

Listing 2: Standard AXI pragmas example

The controller operates as a simple finite state machine (FSM), as depicted in Figure 2. It remains in the *IDLE* state until the accelerator requests data access. Upon activation, the controller places the transaction data in the appropriate AXI channels and transitions to either the *W_READ* or *W_WRITE* state. In these states, the controller waits for the AXI handshakes before deasserting the signals on the relevant AXI channels and then waits for the response from the AXI slave device. This necessitates stalling in these states until the entire transaction is complete before returning to the *IDLE* state.

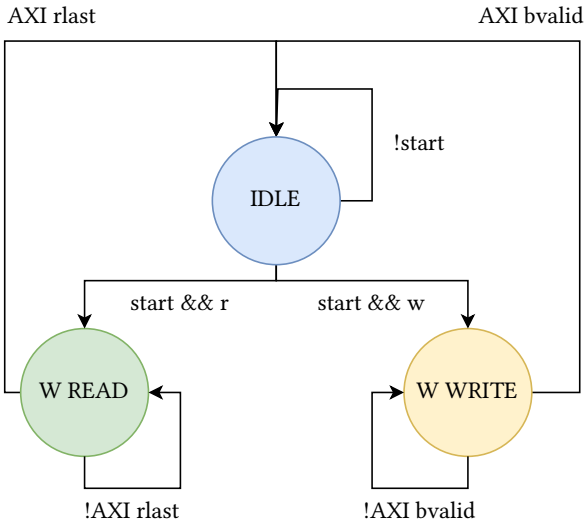

Figure 2: Standard AXI Finite State Machine

While functional, this memory controller has significant drawbacks. It processes requests sequentially, meaning each memory access generates a new AXI transaction and incurs the full latency of external memory access for each data transfer. Additionally, the controller cannot handle multiple requests simultaneously (e.g., concurrent read and write transactions) and can only accept new requests once the previous one has been fully processed by the slave device.

3.2 AXI Caches

Our AXI caches are built upon the Standard AXI controller architecture. When a user decides to instantiate a cache, the HLS tool generates the standard controller module with the same interface. However, its functionality changes significantly. Instead of handling the memory interface directly, it now instantiates and controls a customized cache module, effectively transferring bus control to the cache's internal logic.

To maintain transparency, caches are integrated within the memory controller module rather than as separate entities. This ensures that the rest of the accelerator operates as if memory requests are being served individually, regardless of the cache's presence. However, transparency cannot be preserved at the end of computation due to the nature of cache operations. Caches reduce stalls by minimizing the number of memory transactions. For instance, when writing to external memory, the cache may store data internally and signal readiness for new operations without waiting for the data to be fully transferred. To ensure data consistency, a flush operation is added at the end of execution, compelling caches to complete all pending write transactions before the accelerator signals the end of the computation. Since the Standard AXI controller does not include a dedicated flush port, this operation is triggered by

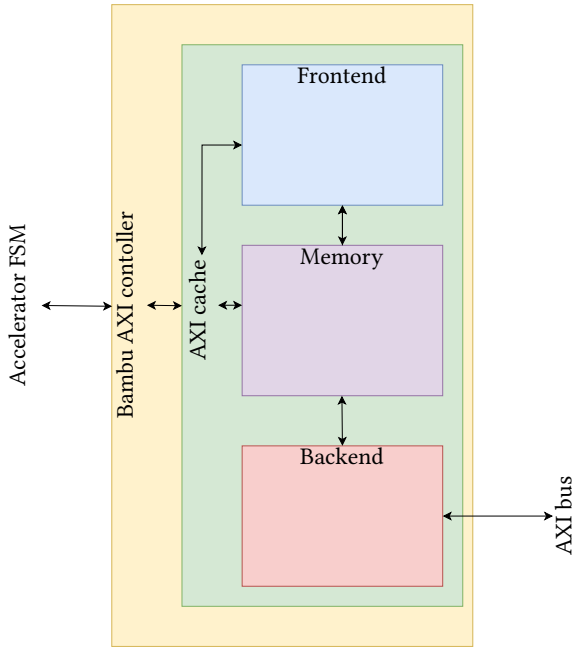

Figure 3: Proposed cache architecture

encoding the end-of-computation as a 0-bit write request to any memory address.

Adding caches offers several advantages over the Standard AXI controller. Caches exploit spatial and temporal locality by storing parts of the dataset, reducing the number of outgoing transactions and total latency when data is reused. Prefetching further reduces average latency by transferring longer sequences, such as cache lines, instead of single elements. Additionally, caches can accept new transactions before completing previous ones and handle multiple active transactions on the AXI channels simultaneously. However, caches also have drawbacks. They require additional logic to manage the internal state and on-chip memory to store data. The need for flushing introduces further complexity. Caches do not guarantee improved performance; applications with poor locality may experience delays due to rarely used prefetched data. Consistency issues may arise as cached data may not be coherent with the external memory state. In this work, we assume the accelerator has exclusive write access to the data it operates on, mitigating some consistency concerns.

3.2.1 AXI Caches architecture. The architecture of the integrated AXI caches are inspired by IObundle (IOb) caches [9]. However, significant enhancements have been made to improve latency hiding capabilities and to incorporate essential functionalities, such as flushing, which are critical for HLS-generated accelerators. Customizability is paramount for these caches. HLS tools can generate accelerators with diverse input/output sizes and memory access patterns. Instantiating a single, non-configurable, cache component would be suboptimal, due to varying size requirements and memory access patterns depending on the generated accelerator. Therefore, Bambu supports caches that are both resizable and adaptable in their internal behavior based on user specifications. The architecture is divided into three main modules: frontend, cache memory, and backend.

- **Frontend**: This module handles requests from the cache controller through a simple valid/ready interface. The valid signal indicates a memory access request from the controller, while the ready signal indicates the completion of the operation by the cache. The frontend also registers input signals necessary for other cache modules and provides output data to the cache controller.
- **Cache memory**: This module is the core of the cache, maintaining the internal state, data, and cache line tags in RAMs with a single-cycle access latency. It uses a register file (one bit for each cache line) to track the validity of cache contents. For write-back caches, a similar register file tracks modified lines that need to be written back to memory upon eviction. The memory module can identify cache hits, which are served in the next clock cycle, or cache misses, which require additional memory operations. For non-direct-mapped caches, this module includes a replacement policy controller that selects the cache line to use for incoming data. The memory module also includes a write buffer to store the address and data of write transactions. This buffer hides write latency by allowing the cache to signal readiness upon transaction insertion, rather than waiting for AXI transaction completion. Our cache design improves upon the original IOb design by using the write buffer with write-back caches. Using the write buffer during memory write transactions prevents stalls when multiple dirty lines are replaced in quick succession. The cache memory module also includes logic for cache flushing, ensuring data synchronization with external memory. For write-through caches, this involves emptying the write buffer. For write-back caches, the dirty bit register file is checked, and dirty lines are moved into the write buffer and written back to the memory. Once all the dirty bits are cleared, the module waits until the write buffer is empty to signal the completion of the flush operation.
- Backend: This module controls the AXI signals and consists of independent controllers for read and write operations, which can operate simultaneously. The read controller is activated on a read miss to fetch the entire missing cache line. The write controller handles every write access for write-through caches and manages dirty line evictions for write-back caches. With respect to the conventional IOb caches, our design supports outstanding write transactions, allowing the write controller to initiate new transactions before receiving responses to previous ones, thus reducing channel latency for frequent writes.

Lastly, our caches do not include a coherency mechanism. Users must ensure exclusive write access to each cache's memory region when annotating their code.

3.2.2 AXI Cache instantiation. As detailed in Section 3.2, AXI caches are integrated into the Standard AXI module. Consequently, cache instantiation requires two distinct sets of `pragma` directives: the

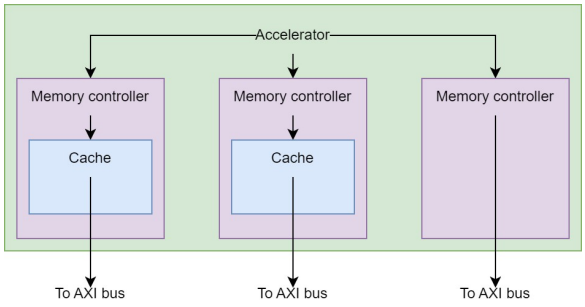

Figure 4: Cache instantiation example

Standard AXI directives and the AXI Cache directives, as illustrated in Listing 3. During HDL generation, the HLS tool checks each memory controller for cache configuration information. If no cache information is found, a Standard AXI controller is generated and connected to the design. Conversely, if cache information is provided, an AXI Cache controller is instantiated, and the cache configuration is customized according to the pragma directives.

```
#pragma HLS_interface a m_axi direct bundle=gmem0
#pragma HLS_interface b m_axi direct bundle=gmem1
#pragma HLS_interface c m_axi direct bundle=gmem2

#pragma HLS_cache bundle=gmem0 way_size=8 line_size=32
#pragma HLS_cache bundle=gmem1 way_size=32 line_size=8
      n_ways=2 bus_size=64 buffer_size=4 rep_policy=lru
      write_policy=wb

void mmult(int *a, int *b, int *c)
```

Listing 3: AXI Caches pragmas example

It is important to note that our caches are exclusively bound to the memory interface. This allows for Design Space Exploration (DSE) by enabling users to selectively include caches only for memory ports that would benefit from them while omitting them where they would be ineffective within the same design. Additionally, caches on different memory channels are fully independent, allowing each memory channel to be customized without affecting others. This modular approach simplifies finding an optimal configuration, as the configuration only needs to be optimal for the memory locations accessed through that specific port.

`pragma` directives for AXI Caches consist of a preamble identifying the directive and a list of parameter assignments. All parameters, except for the *bundle* parameter, are used directly to customize the cache configuration. The directive requires the user to assign values to three mandatory parameters, while the remaining parameters are optional. In the provided example, the first cache `pragma` specifies only the mandatory parameters, whereas the second one includes all possible customization options. Detailed descriptions, possible values, and default values (for optional parameters) of the cache configuration parameters are presented in Table 1. The memory controller architecture inferred by these directives is depicted in Figure 4.

4 BENCHMARKS AND EVALUATION METRICS

To develop a heuristic for guiding designers toward optimal cache configurations, we selected relevant performance metrics and three benchmarks: Fast Fourier Transform (FFT) [1], GramSchmidt [5], and DigitRecognition [2]. Using Bambu, our HLS tools supporting caches, we generated multiple HDL versions for each benchmark: one without caches (serving as a baseline) and a set of 32 predetermined cache configurations. These configurations were derived by permutating five of the seven cache parameters, each with two possible values (small and large). The two unexplored cache parameters — write policy and replacement policy — were left at their default values, as they are expected to have minimal performance impact in a single-level cache architecture. We synthesized these configurations targeting a VCK5000 board which hosts a Versal AI Core VC1902 device. We simulated these configurations using the testbench automatically generated by Bambu HLS, employing realistic memory access latency values, estimated between 85 and 90 ns based on the AXI data bus size, for our target VCK5000 board using the uBench [4] benchmark. Performance metrics were extracted by analyzing Value Change Dump (VCD) files generated during simulations, identifying events such as AXI channel handshakes and memory request start/end times. From these results, we formulated a heuristic to expedite design space exploration and achieve near-optimal solutions Additionally, we synthesized and executed another benchmark, Resnet-18 [6], on the same target board. For Resnet-18, instead of analyzing VCD files, we instrumented the generated code and added memory-mapped registers to collect metrics and validate our heuristic. By iteratively synthesizing, executing, and extracting metrics following our heuristic, we achieved a 3.5× speedup with a 13% resource overhead in three tuning steps.

4.1 Benchmarks

4.1.1 FFT. The Fast Fourier Transform (FFT) algorithm converts a sequence of inputs into their frequency domain representation. It has a high density of load and store operations, with a regular but progressively increasing access step and minimal temporal locality. Initially, caches provide a performance advantage over the baseline due to the regular access pattern. However, as the access step increases, the advantage diminishes because more elements are prefetched on a cache miss and remain unaccessed.

4.1.2 GramSchmidt. The GramSchmidt algorithm decomposes an input matrix into a unitary matrix and an upper triangular matrix. It is a memory-bound algorithm with a balanced mix of load and store operations, exhibiting a regular and stable access pattern with limited temporal locality. These characteristics make it well-suited for cache utilization.

4.1.3 DigitRec. DigitRec implements the k-Nearest Neighbor algorithm for digit recognition. It involves a minimal number of store operations compared to a large number of loads. The load access pattern is sequential with no temporal locality, making cache prefetching particularly beneficial in this scenario.

4.1.4 Resnet-18. Resnet-18 is an 18-layer convolutional neural network, comprising convolutions, batch normalizations, ReLUs, and vector additions. While some layers follow a sequential access pattern, the bulk of the computation occurs in the convolution layers.

Table 1: Cache Parameters

Parameter	Description	Possible values	Default
`bundle`	Name of the AXI port this cache is attached to.	Bundle name must match one of the ports specified with the Standard AXI pragma	None
`n_ways`	The number of positions a specific cache line can be stored in the internal memory. Reduces miss rate at the cost of resource utilization.	Powers of 2	1 (direct-mapped)
`way_size`	The number of cache lines present in each cache way. Reduces the miss rate at the cost of more resources.	Powers of 2	None
`line_size`	The length of each cache line, i.e. how much data is read on a cache miss. Not expressed in bits, but as the number of elements that the line must contain.	Powers of 2	None
`bus_size`	The size, in bits, of the AXI data bus. Increasing the bus size reduces the AXI burst length.	Powers of 2, from 32 to 1024	Width of data-type
`buffer_size`	The maximum number of pending write transactions that can be stored in the write buffer. Reduces the risk of stalling on write requests at the cost of more resources.	Powers of 2	2
`rep_policy`	The replacement policy used for set-associative caches. Changing policy might reduce or increase miss rate.	`lru`: Least Recently Used `tree`: Tree-based LRU approximation	`lru`
`write_policy`	The policy used by to handle write requests.	`wt`: Write-through no-allocate `wb`: Write-back allocate	`wt`

These layers exhibit a balanced mix of load and store operations, with the input data accessed through a sliding window with good temporal locality, and the layer weights accessed sequentially. Both access patterns are expected to perform well with our caches.

4.2 Metrics

4.2.1 AXI handshakes. AXI handshakes estimate the traffic generated on the AXI bus by the accelerator. Ideally, a cache reduces the generated traffic, as cache hits can be served locally without accessing the main memory.

4.2.2 Cache hits and misses. The cache hit-to-miss ratio is crucial in assessing the effectiveness of caches. A high ratio indicates that most requests are served immediately, avoiding the full memory access latency. Conversely, a low ratio means the cache cannot serve many requests in a single clock cycle, leading to frequent stalls. A sufficiently low ratio can even degrade performance compared to the baseline. It is important to note that an access is considered a cache miss if it cannot be served immediately, regardless of the memory's internal state. This includes situations where a write request cannot be processed because the write buffer is full.

4.2.3 Write buffer full and stall clock cycles. These metrics provide insight into the performance of write transactions. Caches must stall write transactions when the write buffer is full. Estimating the time spent in this condition is useful for designers. Both the number of clock cycles lost due to a full write buffer and the number of clock cycles the buffer is full (even if it does not cause a stall) are included. Unlike in our experimental setup, the access latency on real hardware is not fixed. Thus, a full buffer might be emptied quickly enough to avoid a stall in simulation, but not in actual hardware.

4.2.4 Average memory access duration. This metric estimates the average time, in clock cycles, to perform a memory request. Ideally, caches should reduce this value to as close to 1 as possible.

4.2.5 Execution time and clock cycles. These metrics measure the overall effectiveness of the caches.

5 EXPERIMENTAL RESULTS

As outlined in Section 4, we conducted an exhaustive exploration of the design space for the selected benchmarks, evaluating 32 different cache configurations for each. However, many configurations resulted in only marginal variations in performance metrics such as overall latency and cache hit/miss ratios. Therefore, our subsequent discussion focuses on analyzing the pivotal points in the design space.

First, we describe a straightforward approach to infer a heuristic for narrowing the design space. Then, for each benchmark, we discuss the simulation results of the most relevant design points. Finally, we apply this heuristic search to the Resnet benchmark, analyzing performance and synthesis results on FPGA.

5.1 Heuristic

Based on our observations across numerous design points, we found that several cache configurations yield equivalent optimal results. While our approach greatly facilitates design space exploration without requiring modifications to the behavioral specification, it remains desirable to identify a configuration close to the saturation point with the minimal number of explored design points.

The complete synthesis of each design point, from specification to the device can require several hours. When additional configuration aspects (such as different HLS step algorithms) and optimization objectives are considered, the design space can quickly become prohibitive to explore.

To address this, we propose a heuristic to iteratively narrow and explore the design space. The most critical factor appears to be the cache size. Thus, we recommend starting with the smallest cache configuration and evaluating performance improvements over the baseline. If performance degrades, caches may not be suitable. Otherwise, incrementally increase the cache size and observe performance gains. The improvement must be substantial enough to justify the additional resource usage. Repeat this process until the performance gain becomes marginal. At that point, revert to the previous cache size and explore configurations with that size.

If the stalls caused by a full write buffer, or the number of clock cycles with a full buffer, are significant compared to the overall execution time, try a configuration with a larger write buffer to check for improvements. Finally, evaluate whether increasing the size of the AXI data bus could speed up memory transfers or reduce bus traffic.

The exploration steps for each of the benchmarks are detailed in Tables 2, 3, 4, and 5. For brevity, only the steps relevant to our heuristic are included, even for benchmarks where an exhaustive search was performed.

5.2 FFT

Following the heuristic, we describe the iteration process in Table 2. In the first step, adding the smallest cache configuration to FFT yields mixed results. The average number of clock cycles per memory access decreases, and execution time is reduced. However, the number of AXI handshakes generated by the accelerator nearly triples, which could degrade system performance when accessing external memory through a common AXI interconnect.

In the second step, we increase the cache size. This results in significant performance improvements, achieving a 4x speedup over the baseline. Additionally, the number of AXI handshakes is roughly halved compared to the baseline, indicating potential benefits for the entire AXI interconnect.

In the third step, further increasing the cache size does not yield significant improvements. While the average memory access time and clock cycle count decrease, the gains do not justify doubling the cache size. This performance saturation is expected since the dataset size is only 8KB. Therefore, we revert to the previous cache size and explore other parameters.

Notably, the large number of stalls due to a full write buffer prompts us to increase its size in the fourth step. This change eliminates buffer-related stalls and slightly reduces the clock cycle count. Despite the minimal impact on total execution time, we accept this change as it significantly improves write buffer performance without requiring substantial additional logic.

Finally, in the fifth step, we attempt to minimize AXI handshakes by increasing the AXI data bus size. However, the number of handshakes remains similar to the previous iteration. Consequently, we select the configuration from step four.

This configuration closely matches the performance of the best configuration across the entire design space, with only a 1% performance gap and a marginal reduction in AXI traffic, while avoiding the need to double the cache size.

5.3 GramSchmidt

Table 3 outlines the heuristic steps for GramSchmidt. As anticipated, the initial step using the smallest cache configuration results in a dramatic performance improvement. Execution time is more than halved, and the average memory access time is reduced to one-third of the baseline, with a favorable cache hit-to-miss ratio. This improvement comes at the cost of a slightly increased number of AXI handshakes.

Encouraged by this significant performance boost, we increase the cache size in the second step. This yields an additional performance gain, with an almost 1.7× speedup and near-ideal average access time. Additionally, the number of AXI handshakes is substantially reduced, generating less traffic on the interconnect.

In the third step, we further increase the cache size, but no noticeable improvements in the metrics are observed. This is expected, as the entire dataset already fits into the previous cache size. Reverting to the cache size used in step two, we determine that exploring the effect of the write buffer size is unnecessary. The accelerator never stalls due to a full buffer, and the number of cycles with a full buffer is negligible compared to the overall number of clock cycles.

In the fourth and final step, we increase the size of the AXI data bus. However, this change does not yield significant improvements in any of the considered metrics. Consequently, we select the configuration from step two. This configuration compares favorably with the most performant one, offering a similar performance (around 1% slower) and reduction in AXI handshakes, without the need to double the cache size.

5.4 DigitRec

In Table 4, we outline the heuristic steps for DigitRec. As anticipated, introducing a small cache in the first step significantly enhances the performance of this accelerator. All metrics show substantial improvements over the baseline, highlighted by a favorable hit-to-miss ratio that results in a 3.1× speedup and a 2.5× reduction in AXI bus traffic.

Upon increasing the cache size in the second step, there is further improvement in execution time and traffic reduction. However, we find the gains insufficient to justify doubling the cache size. Returning to the previous cache size and noting that the write buffer never reaches full capacity, we decided against exploring different write buffer sizes as it would not yield improvements.

Next, we assess the impact of increasing the size of the AXI data bus. Given the accelerator's heavy reliance on cache prefetching for performance gains, this configuration proves effective, delivering comparable execution time improvements as the cache size increases while significantly reducing AXI handshakes. Consequently, this configuration from step three is preferred over the one from step two.

With no further parameters to adjust without increasing the cache size, our heuristic settles on the step three configuration. Compared to the configuration with the best overall performance,

Table 2: FFT - 1024 × 2 × 4 bytes metrics

Step	Exec. Time (ms)	Cache Size (KB)	Ways	Line size	Bus size	Buffer size	Way size	AXI hs	hit/miss	buf full/stall	mem cc	cc
0	7.2	-	-	-	-	-	-	132802	-	-	18.6	1.43M
1	6.0	4	1	16	64	2	64	324864	26064/29104	30400/9424	16.5	1.20M
2	1.8	16	4	16	64	2	64	68544	53863/1305	171877/21907	1.25	0.37M
3	1.8	32	1	128	64	2	64	68432	53863/1305	172950/22339	1.21	0.36M
4	1.8	16	4	16	64	16	64	68544	55040/128	0/0	1.07	0.36M
5	1.8	16	4	16	256	16	64	67776	55040/128	0/0	1.05	0.35M
best	1.8	32	1	128	256	16	64	67664	55152/16	0/0	1.01	0.35M

Table 3: GramSchmidt - 1024 × 3 × 4 bytes metrics

Step	Exec. Time (ms)	Cache Size (KB)	Ways	Line size	Bus size	Buffer size	Way size	AXI hs	hit/miss	buf full/stall	mem cc	cc
0	14.5	-	-	-	-	-	-	263696	-	-	17.7	2.91M
1	6.4	4	1	16	64	2	64	265902	96754/18446	7936/0	5.5	1.28M
2	3.8	16	4	16	64	2	64	101472	115024/176	7936/0	1.03	0.76M
3	3.8	32	1	128	64	2	64	101448	115176/24	7936/0	1.02	0.76M
4	3.8	16	4	16	256	2	64	100416	115024/176	7936/0	1.03	0.76M
best	3.8	32	1	128	256	16	64	100296	115176/24	0/0	1.01	0.76M

Table 4: DigitRec - ~72000 × 8 bytes metrics

Step	Exec. Time (ms)	Cache Size (KB)	Ways	Line size	Bus size	Buffer size	Way size	AXI hs	hit/miss	buf full/stall	mem cc	cc
0	145	-	-	-	-	-	-	2304024	-	-	18.0	29.1M
1	45.9	8	1	16	64	2	64	907768	1098646/53362	0/0	2.71	9.17M
2	42.7	64	4	16	64	2	64	612058	1116006/36002	0/0	2.16	8.53M
3	42.7	8	1	16	256	2	64	266834	1098646/53362	0/0	2.16	8.54M
best	36.2	2048	4	128	256	2	512	18603	1151445/563	0/0	1.03	7.23M

our selected configuration is approximately 1.3× slower and generates 49× more AXI handshakes. While the reduction in bus traffic is substantial, the selected configuration is 256 times larger than the one recommended by our heuristic. Even considering the performance improvement, the significant increase in size is difficult to justify.

5.5 Resnet-18

Table 5 presents the evaluation of our approach on Resnet-18, deployed on a VCK5000 board with a target frequency of 200MHz. In our first exploration step, introducing relatively small cache results in a notable 2.7× speedup and reduces AXI handshakes without significantly impacting resource utilization.

In the second step, we further increase the cache size but observe quite similar quality of results, with a performance improvement fully compensated by the slightly higher resource requirements. Therefore, we revert to a 4kB cache size and explore other configurations.

Given that the number of cycles with a full write buffer is minimal compared to the overall execution clock cycles, we opt against exploring larger buffer sizes. As a third and final step, we instead opt to increase the size of the AXI data buffer. This adjustment yields improved overall performance and further reduces traffic, making this the optimal configuration according to our heuristic.

Ultimately, we achieve a 3.5× speedup over the cacheless baseline while only requiring approximately 10% more registers and 13% more LUTs.

6 CONCLUSIONS

High-level synthesis (HLS) techniques have proven to be invaluable for accelerating the prototyping of custom accelerators in recent years. However, as modern applications handle increasingly large datasets requiring efficient utilization of external memory bandwidth, there remain significant opportunities for enhancement to achieve optimal performance.

In this paper, we present a methodology for integrating tunable caches into HLS-generated designs. Our approach leverages pragma directives to specify cache configurations without altering the input behavioral specification, thereby streamlining design space exploration. We demonstrate the practicality and effectiveness of our method by implementing it within a state-of-the-art open-source HLS tool. Through exhaustive exploration of multiple cache configurations across selected benchmarks, we propose a heuristic to efficiently navigate the design space for future optimizations.

Applying our heuristic, we optimized the Resnet-18 benchmark on an FPGA device in just a few iterations. Our approach achieved a notable 3.5× speedup compared to a baseline design without caches, with minimal area overhead and a short design cycle, requiring the

Table 5: Resnet-18 -metrics

Step	Exec. Time (s)	Cache Size (KB)	Ways	Line size	Bus size	Buffer size	Way size	AXI hs	hit/miss	buf full/stall	mem cc	cc	Registers (K)	LUTs (K)
0	2543	-	-	-	-	-	-	82B	-	-	21.7	1414B	178	215
1	951	4	1	16	64	2	64	42B	14.9/3.0 B	31/12 M	5.9	190B	178	219
2	925	16	4	16	64	2	64	41B	15.1/2.8 B	31/12 M	5.9	190B	180	223
3	720	4	1	16	256	2	64	35B	16.8/1.1 B	31/12 M	3.3	144B	195	244

exploration of only three points in the design space. This underscores the efficacy of using a parametric cache solution inside an HLS tool to perform design space exploration.

ACKNOWLEDGEMENT

This work is supported by the US DOE Office of Science project "Advanced Memory to Support Artificial Intelligence for Science" at Pacific Northwest National Laboratory.

REFERENCES

[1] 2018. FFT. https://github.com/ferrandi/PandA-bambu/blob/main/examples/fft_example/fft_float.c Online, accessed 06-2024.
[2] 2020. DigitRec. https://github.com/ferrandi/PandA-bambu/blob/main/examples/rosetta/digit-recognition/digitrec_sw.c Online, accessed 06-2024.
[3] 2021. *AMBA AXI and ACE Protocol Specification.* Technical Report. ARM. https://developer.arm.com/documentation/ihi0022
[4] 2021. uBench. https://github.com/SFU-HiAccel/uBench/tree/main Online, accessed 06-2024.
[5] 2023. GramSchmidt. https://github.com/jianyicheng/HLS-benchmarks/blob/master/C-Slow/gramSchmidt/gramSchmidt.cpp Online, accessed 06-2024.
[6] 2023. resnet-18. https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py Online, accessed 06-2024.
[7] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *58th ACM/IEEE Design Automation Conference (DAC'21)*. 1327–1330. https://doi.org/10.1109/DAC18074.2021.9586110
[8] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307
[9] Mário P. Véstias João V. Roque, João D. Lopes and José T. de Sousa. 2021. IObCache: A High-Performance Configurable Open-Source Cache. *Algorithms* (July 2021). https://doi.org/10.3390/a14080218
[10] Antonino Tumeo and John Feo. 2015. Irregular Applications: From Architectures to Algorithms [Guest editors' introduction]. *Computer* 48, 8 (2015), 14–16. https://doi.org/10.1109/MC.2015.233