Efficient Data Supply for Hardware Accelerators with Prefetching and Access/Execute Decoupling

Tao Chen and G. Edward Suh
Cornell University
Ithaca, NY 14850, USA
{tc466, gs272}@cornell.edu

Abstract— **This paper presents an architecture framework to easily design hardware accelerators that can effectively tolerate long and variable memory latency using prefetching and access/execute decoupling. Hardware accelerators are becoming increasingly popular in modern computing systems as a promising approach to achieve higher performance and energy efficiency when technology scaling is slowing down. However, today's high-performance accelerators require significant manual efforts to design, in large part due to the need to carefully orchestrate data transfers between external memory and an accelerator. Instead, the proposed framework utilizes automated program analysis along with High-Level Synthesis (HLS) tools to enable prefetching and access/execute decoupling with minimal manual efforts. The framework adds tags to accelerator memory accesses so that hardware prefetching can effectively preload data for accesses with regular patterns. To handle irregular memory accesses, the framework generates an accelerator with decoupled access/execute architecture using program slicing. Experimental results show that the proposed optimizations can significantly improve performance of HLS-generated accelerators (average speedup of 2.28x across eight accelerators) and often reduce energy consumption (average of 15%).**

I. Introduction

As the technology scaling slows down, future computing systems will need to increasingly rely on hardware accelerators to improve performance and energy efficiency. In fact, today's mobile SoCs already rely on a number of hardware accelerators to perform compute-intensive tasks such as audio and video processing, image signal processing, always-on speech recognition, etc. We expect that this trend will continue and future systems will contain more specialized accelerators.

This paper proposes a framework to automatically optimize hardware accelerators and enable them to effectively hide long, variable memory latencies of an SoC memory hierarchy by preloading data in parallel to computations. The effective data preloading is achieved through hardware prefetching and design transformations to decouple memory accesses and computations. This framework is generally applicable to stand-alone accelerator designs that are attached to the memory bus or the last-level cache and have their own memory access logic. This accelerator design style is widely adopted both in industry and the research community [1]–[6]. Applying the framework requires minimal manual efforts when used with high-level synthesis. While the principle of access/execute decoupling has been explored in various contexts, to the best of our knowledge, this paper represents the first to systematically apply this principle to the design of stand-alone accelerators and demonstrate how to enable decoupling automatically and efficiently.

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

While the high-level approach can be applied to any accelerator in general, the framework in this paper is designed to target accelerators that are generated using a High-Level Synthesis (HLS) tool. Design complexity represents a major challenge in deploying accelerator-rich architecture in practice, and manually designing an accelerator in RTL for each application is likely to be too expensive, especially given stringent time-to-market requirements and the rapidly evolving nature of emerging applications. High-Level Synthesis (HLS) is one of the most promising approaches to address the design complexity problem as it enables automatically generating hardware accelerators from high-level descriptions, such as C code. HLS is quickly gaining momentum, being used in both industry designs [7] as well as hardware accelerator research [8], [9].

Unfortunately, even with HLS, data supply from memory often needs to be carefully coordinated with manual optimizations in order to achieve high performance in hardware accelerators. For example, today's HLS tools assume a fixed latency of all memory accesses, and rely on accelerator designers to write explicit logic to manage the communication between DRAM and on-chip scratchpad memory. This approach requires serious manual design efforts, and the resulting management logic is accelerator-specific and not reusable for other designs. Alternatively, designers can use caches to ease communication management given locality in memory accesses [10]. However, we found that caches are not sufficient to provide high performance without carefully orchestrated data supply. Unlike modern processors with expensive latency-hiding mechanisms such as dynamic scheduling, typical accelerators rely on a static pipeline schedule and a cache miss stalls the entire pipeline.

This paper aims to enable efficient data supply for HLS-based accelerators without manual efforts necessary today. To achieve this goal, we remove inefficiencies in today's cache-based accelerators in two ways. First, we use a prefetch engine to remove cache misses for easy-to-predict memory accesses. The prefetch engine is general and common across accelerators. For example, we use a stride prefetcher in our experiments. Second, to handle complex memory access patterns,

we propose to decouple memory access logic of an accelerator from the main computation pipeline. For many accelerators, memory addresses of data that need to be accessed are often independent of main computations and can be computed ahead to fetch data in parallel to the main computation. In fact, data supply in manually optimized accelerators rely on such decoupling and preloading. In this paper, we show that this decoupling can be done automatically using program slicing on a high-level accelerator design.

While prefetching and access/execute decoupling have been studied for processing cores, we found that applying them to accelerators introduce new challenges. For prefetching, unlike processing cores, accelerators do not provide PCs that can be used to easily distinguish different sources of memory accesses. In order to apply traditional prefetch algorithms, we augment our accelerator generation process to automatically add additional tags.

We also found that simply decoupling memory accesses from main computations alone do not significantly improve performance of accelerators unless independent accesses can be overlapped. The decoupled access/execute (DAE) architecture on processing cores rely on expensive out-of-order or dataflow execution to perform multiple accesses in parallel. For hardware accelerators with static pipelines, we show that simple decoupling of memory accesses through dedicated forwarding logic is sufficient to achieve good performance with minimal overhead in most cases.

In order to evaluate the effectiveness of the proposed framework, we applied prefetching and access/execute decoupling to eight HLS-based accelerators. The experimental results show that the proposed framework can be applied to accelerators with minimal manual efforts and significantly improve the performance compared to the baseline accelerator. The DAE architecture alone improved performance by 1.89x on average while the average speedup increased to 2.28x when prefetching was added. The optimizations also reduce energy consumption for many accelerators, by 15% on average.

The main contributions of this paper are:

1) A hardware architecture that systematically applies the principle of access/execute decoupling to designing hardware accelerators that achieve efficient data supply.
2) An automated approach to tag accelerator memory accesses to enable effective hardware prefetching for accelerators.
3) An automated framework which combines high-level synthesis, program slicing, and an architectural template written in a hardware generation language. The framework enables generating fully synthesizable, customizable, access/execute decoupled, and prefetching-enabled accelerators with minimal manual effort.
4) Detailed evaluations of the performance and energy impact of prefetching and access/execute decoupling on hardware accelerators using a commercial ASIC flow.

The rest of the paper is organized as follows. Section II provides an overview of the accelerator data supply problem and briefly discusses the proposed solution. Section III describes prefetching as an approach to improve accelerator data supply and a technique to enable efficient prefetching for hardware accelerators. Section IV describes the architecture of access/execute decoupled accelerators as well as a framework to automatically generate them from a high-level description. Section V discusses our evaluation methodology, experimental setup, and evaluation results. Section VI discusses related work, and Section VII concludes the paper.

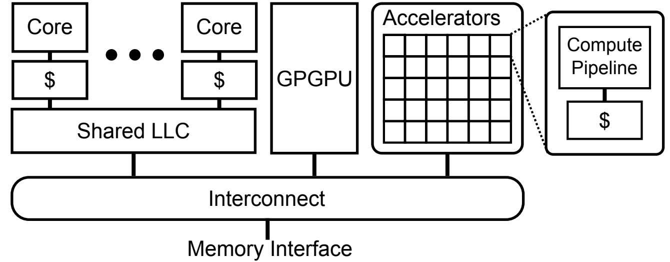

Fig. 1. System architecture.

II. Overview

A. System Architecture

Figure 1 shows the high-level system architecture that we assume in this paper. The system is a heterogeneous SoC that consists of general-purpose processing engines such as processor cores and GPGPUs as well as a large number of accelerators. We consider stand-alone accelerators that are loosely-coupled to the cores and have their own memory interfaces to access main memory. A processing core configures and initiates an accelerator, then the accelerator performs its computation without intervention from the core. Each accelerator has its own compute pipeline and accesses memory through an on-chip cache.

B. High-Level Synthesis

In this work, we target to provide efficient data supply for accelerators that are generated using a High-Level Synthesis (HLS) flow. Figure 2 shows a typical HLS flow that automatically transforms a functional description of the accelerator written in a high-level language such as C or C++ into an optimized register-transfer level (RTL) description. To achieve this, HLS tools first transform source code into control data flow graphs (CDFG), and then perform allocation, scheduling, and binding to generate the final RTL. HLS tools usually pipeline the computation in order to achieve high performance. The pipeline is generated using a static schedule, where each operation is placed in a fixed slot determined at compile time. This approach works well if all functional units and memory operations have a short fixed latency. For operations with an uncertain latency, the HLS tool has to use a best guess for scheduling. For example, cache accesses are usually assumed to be a hit in order to generate a compact pipeline schedule. Then, the pipeline is stalled at run-time if an access turns out to be a cache miss.

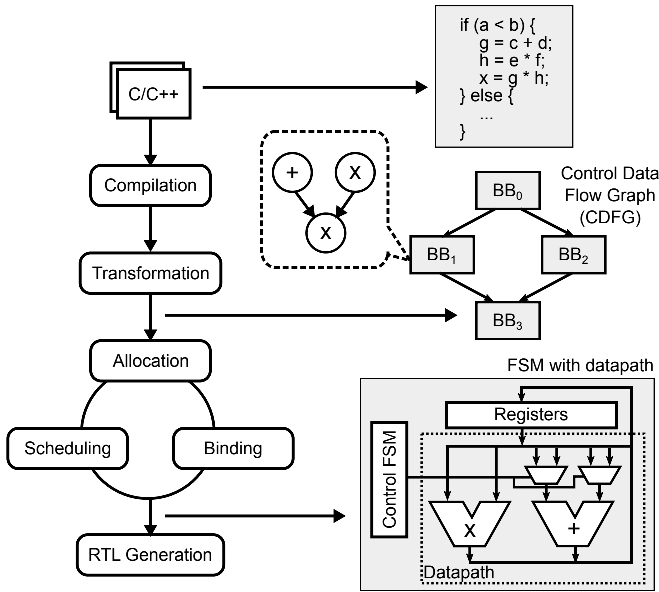

Fig. 2. High-level synthesis flow.

```
for (j = begin; j < end; j++) {
    #pragma HLS pipeline
    Si = val[j] * vec[cols[j]];
    sum = sum + Si;
}
```

Fig. 3. The inner loop of sparse matrix vector multiplication.

C. Impact of Memory Accesses on Accelerator Performance

We use an example to illustrate how a long memory access latency on a cache miss can impact accelerator performance. The code in Figure 3 shows the inner loop of a sparse matrix vector multiplication (spmv) accelerator. Note that the access to the `vec` array is an indirect memory access that has an irregular access pattern, and is likely to miss in the cache.

An example pipeline schedule for the corresponding accelerator is shown in Figure 4. The pipeline has an initiation interval (II) of one, that is, a new iteration can begin execution every clock cycle in the ideal case, as illustrated in Figure 4(a). The three load operations in each iteration are to `val`, `cols`, and `vec`, respectively. Figure 4(b) shows an actual pipeline operation when accessing `vec` in the first iteration incurs a cache miss. Since the schedule is static, the entire pipeline has to stall until the miss is resolved, even though the memory accesses of later iterations might have been hits. The stall due to a long memory access latency can have a large impact on the accelerator's performance. For example, in Figure 4(b), although only one out of four iterations has a cache miss, the effective initiation interval for the four iterations is increased from one to two, essentially lowering the throughput by half. The impact can be even larger for accelerators with deeper pipelines where one cache miss can potentially stall many more operations than what is shown in the example.

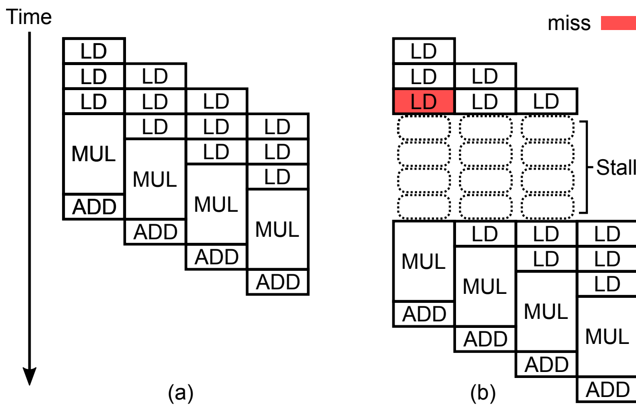

Fig. 4. Example schedule of an HLS spmv accelerator with (a) ideal memory (b) cache that has a miss when accessing `vec`.

Our experimental results on a set of HLS-based hardware accelerators suggest that the performance loss due to long memory accesses is significant. There exists a large performance gap between accelerators with ideal memory (1-cycle) and a realistic cache-based memory hierarchy. This work aims to bridge this gap by developing techniques to automatically preload data for accelerators. An ideal preloading scheme would effectively eliminate cache misses, and allow the pipeline to run at the full throughput possible with the ideal memory.

D. Data Preloading Framework

There are a few challenges in developing a data preloading scheme to enable efficient data supply for accelerators. First, the scheme needs to accurately predict future data needs of an accelerator so that data can be preloaded. Second, the prediction needs to be early enough to hide memory latency. Third, the prediction and memory accesses need to be decoupled from computation so that accesses and computation can happen in parallel. Fourth, all the above need to be performed automatically with minimal manual efforts.

In this work, we use two data preloading techniques to hide long memory accesses: (1) prefetching and (2) access/execute decoupling. These two techniques have complementary characteristics, and can both be applied with minimal manual efforts.

Hardware prefetchers predict likely memory addresses to be accessed in the future by observing a sequence of memory accesses at run-time. For example, a stride prefetcher is widely used to detect and preload streaming memory accesses with a fixed stride. In our example, simple strided accesses such as `val[j]` and `cols[j]` can easily be detected and preloaded by a hardware prefetch engine. Moreover, the prefetch engine is inherently decoupled from accelerators and can perform multiple prefetching operations in parallel. On the other hand, on-line prefetching often cannot accurately predict complex memory access patterns such as the indirect accesses (`vec[cols[j]]`) in our example.

For difficult-to-predict memory accesses, we use decoupled access/execute (DAE) architecture. In this approach, we observe that program slicing techniques can be used to automat-

TABLE I
COMPARISON OF PREFETCHING AND DAE.

	Binding	Accuracy	Timeliness
Prefetch	No	Good when regular	Good
DAE	Yes	Good	Depends

ically separate parts that are necessary to compute addresses for memory accesses (*access part*) from the rest that performs main computations (*execute part*). Then, the access part can run ahead of the execute part to preload data. In a sense, the DAE approach provides a perfectly accurate predictor for future memory accesses. However, decoupling and providing early predictions can be more difficult in the DAE architecture compared to prefetching. In DAE, address generations must be exact (binding) unlike prefetching whose predictions may be incorrect (non-binding). Also, in certain cases, it may be difficult to decouple the access and execute parts due to dependencies. Table I summarizes the characteristics of prefetching and DAE in terms exactness in address generation, accuracy, and timeliness.

Our experiments show that prefetching and DAE can complement each other. DAE enables accurate preloading of memory addresses when possible. Prefetching provides speculative preloading for simple access patterns when DAE cannot generate exact addresses early enough.

III. PREFETCHING

As we mentioned in the previous section, hardware prefetchers observe the memory address stream and predict the addresses that are likely to be referenced in the future. In most cases, just looking at a global address stream is not enough to make good predictions, as the global stream is usually a mixture of multiple data streams with different strides as well as irregular accesses, making it difficult to learn the access pattern and make predictions. Thus, most hardware prefetchers perform stream localization to separate a global address stream into multiple local address streams that can be learned and predicted effectively, and to exclude irregular accesses with poor predictability. Since most hardware prefetchers are designed for general-purpose processing cores, they often use the PC of load and store instructions as a hint for stream localization [11], [12], with the intuition that different streams come from different instructions in the program. In addition, the PC is also used for other purposes such as spatial correlation prediction [13] to improve the accuracy of prefetching. Hardware accelerators, on the other hand, usually do not have a PC. Thus, traditional hardware prefetchers that rely on a PC would not be effective when used naively with hardware accelerators.

We observe that for hardware prefetchers, the fundamental role of a PC is to indicate which memory instruction in a program a memory access comes from. If we replace the PC with a unique identifier for each memory instruction, the prefetcher would work equally well as the identifier provides the same amount of information for stream localization. Thus,

Original Memory Request Message Format

type	len	addr	data

Modified Memory Request Message Format

tag	type	len	addr	data

Fig. 5. Modified memory request message format.

we propose to tag each memory access operation in a hardware accelerator with a unique identifier that is sent to a prefetch engine in place of the PC for each memory access. In our implementation, we modified the memory request message format of the accelerators to include a tag field, as shown in Figure 5. To generate the tags, we modified the HLS flow to add a pass that operates on the CDFG generated by the compiler frontend. The pass traverses the CDFG, tagging each memory operation with a unique identifier that emulates a PC. The pseudo-code of the pass is shown in Algorithm 1. Using the tag, features such as PC-based stream localization would work correctly, and the hardware prefetcher is able to effectively prefetch memory addresses of hardware accelerators.

Algorithm 1 Generate tags for memory accesses

1: **procedure** GENERATETAGS
2: $t \leftarrow 0$
3: **for all** basic blocks in the CDFG **do**
4: **for all** operations in the basic block **do**
5: **if** $op.type = load$ or $op.type = store$ **then**
6: $op.tag \leftarrow t$
7: $t \leftarrow t + 4$
8: **end if**
9: **end for**
10: **end for**
11: **end procedure**

IV. DECOUPLED ACCESS/EXECUTE

While hardware prefetchers are effective in prefetching regular memory accesses, they work less well for complex access patterns or short streams that do not trigger hardware prefetching. The fundamental limit of hardware prefetchers is the lack of semantic information about the computation. Previous studies have proposed various techniques to employ semantic information to enable more accurate prefetching for software programs. For example, software prefetching [14] allows programmers or compilers to embed prefetch instructions into the code, which provide hints to the hardware about the addresses to be accessed in the future. Helper thread [15] and runahead execution [16] pre-execute a part of the program or a specially crafted program slice to bring data into the cache. All these techniques rely on the assumption that memory addresses can often be computed well ahead of when the data are needed. Decoupled access/execute (DAE) [17] materializes

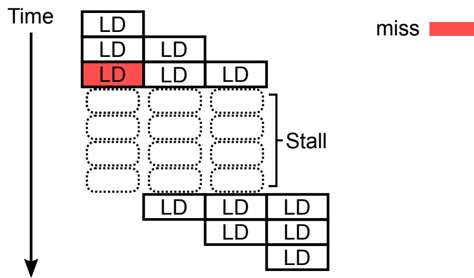

Fig. 6. Example schedule of the access part of a decoupled `spmv` accelerator when there is a cache miss.

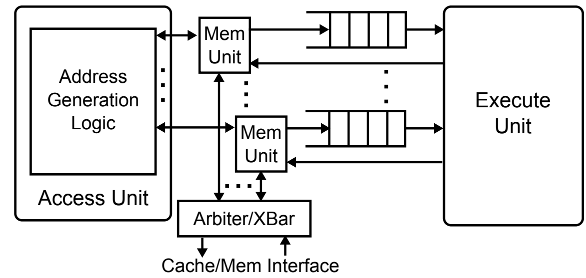

Fig. 7. Architecture of access/execute decoupled accelerators.

this assumption to a full extent by allowing the memory access part, where memory addresses are computed and data accesses are performed, to run ahead of the execute part, where data are consumed. In a typical access/execute decoupled architecture, the access part manages all communications with the memory and supplies data to the execute part; the execute part does not have a memory interface.

A key requirement for achieving performance improvements with DAE is that the access part in the decoupled architecture must run faster than the non-decoupled architecture, otherwise the performance is limited by the access part. However, in highly pipelined accelerators, this is unlikely true. Figure 6 shows an example schedule of the access part of a decoupled `spmv` accelerator where the same miss occurs as in the non-decoupled version shown in Figure 4. The miss has the same performance impact on the access part as in the non-decoupled version. Thus, simply dividing the accelerator pipeline into access and execute parts is unlikely to improve performance significantly when the access part has the same rigid pipeline that cannot tolerate memory latencies. Allowing the access pipeline to tolerate cache misses is a key challenge in designing the DAE accelerator architecture.

Figure 7 shows the architecture of the proposed access/execute decoupled accelerator, consisting of the Access Unit, Execute Unit, Memory Units, and decoupling queues. A visible difference from classic access/execute decoupled architectures is the added memory units, which is a proxy through which memory accesses are performed. Later we will show that this is necessary for tolerating the memory latency. The access unit generates memory addresses and request types, and then sends them to the memory unit to be forwarded to memory. For load operations, once responses come back, the memory unit enqueues the data into the Load Queue (LQ) to be read by the execute unit. For store operations, the memory unit combines the address from the access unit and data from the execute unit, and then sends the request to memory. An access/execute decoupled accelerator can have multiple memory units, which share the cache interface.

A. Access Unit

In a simple DAE accelerator implementation, the access unit is responsible for address generation, handling memory requests/responses, and forwarding data to the execute unit, all in a single static schedule generated by the HLS tool. Among these tasks, address generation and sending out memory requests usually have a fixed latency and thus would work well under the static schedule. Handling memory responses and forwarding data, however, have variable latencies depending on when the response comes back from memory. This has two implications. First, they cannot be executed efficiently under the static schedule generated by HLS. Second, they may stall address generation and sending out requests for other independent accesses. To address this problem, we propose to decouple memory response handling and data forwarding from address generation and sending out requests. Specifically, we delegate these tasks to the memory unit, which handles them independently, decoupled from the access unit.

The result of a load operation can either be used by the execute unit for data computation or by the access unit for address computation. In the first case, the access unit is not involved in handling the load result. This type of load operations are called terminal loads [18]. In the second case, however, the access unit would need to wait for the load result and thus its pipeline could be stalled if the load is a miss. One way to enable the access unit to continue to perform independent operations is to employ an out-of-order core as the access unit, or use dataflow execution for memory accesses [19]. Though these approaches can achieve higher performance, we choose not to employ them because we observe that the load dependency chains in many accelerators are short. In fact, a large portion of load operations are terminal loads. This is because many accelerators mostly perform parallel operations, instead of serial operations through memory such as pointer chasing. Decoupling just terminal loads, i.e. the last node of a load dependency chain, provides most of the benefits with a low cost. Hence, our architecture would work reasonably well for short memory dependency chains, and we trade off the ability to handle long chains for low hardware complexity.

B. Memory Units

Figure 8 shows the hardware structure of the memory unit. It mainly consists of load queue, store queue, forward data queue, dependency checking logic, and memory request/response routing logic.

The Store Address Queue (SAQ) contains store addresses that are not yet sent to memory, either because the store data have not been computed yet, or because it is waiting for access to the memory interface. The Store Data Queue (SDQ) buffers

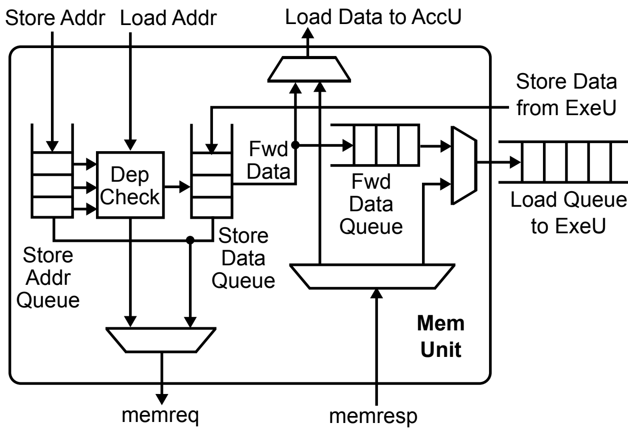

Fig. 8. Hardware structure of the memory unit.

store data from the execute unit. The head entries of SAQ and SDQ are paired to form a store request to be sent to memory.

Each load request from the access unit contains a `dest` field indicating whether the result is used by the access unit or the execute unit. The field is kept in the corresponding response for the request. When the memory unit receives a load request, it checks whether there is an entry in the SAQ matching the store address. If there is a match, the load waits until the corresponding entry in the SDQ is valid, and data is forwarded from the SDQ to either the Forward Data Queue (FwdQ) or the access unit, depending on whether the load operation is a terminal load or not. If there is no match, the load request is sent to the memory. The load and store requests share the memory port. Load requests are given priority over stores to reduce load latency.

The Load Queue (LQ) contains data to be forwarded to the execute unit. When a load response returns from memory, its `dest` field is inspected to route the response data to the LQ, the access unit, or both. Because the execute unit consumes data from memory in a program order, the LQ entries are reserved and maintained in the request order. For example, responses from the memory and the Forward Data Queue are placed in the LQ in the program order. The memory unit supports multiple in-flight requests. If the cache returns responses out-of-order, the LQ is used to reorder and return them in order.

C. Execute Unit

The execute unit is generated using HLS from the execute slice, and mainly consists of the data computation pipeline.

D. Deadlock Avoidance

There exist two possible deadlock situations in the proposed access/execute decoupled architecture. Here we describe them and discuss how to prevent them.

Pipelining-Induced Deadlocks: A deadlock may occur when accelerator pipeline interacts with a store queue of insufficient size. Suppose the execute unit pipeline has latency L and initiation interval II, it needs to consume $N = \lceil L/II \rceil$ inputs before producing the first output. If the store queue size is less than N, it may fill up and block the access pipeline. Because the execute pipeline depends on the access pipeline for data supply, it also blocks and the accelerator deadlocks. The deadlock occurs because pipelines generated by most HLS tools do not support flushing by default. That is, a blocking operation stalls the entire pipeline, not just subsequent iterations. This restriction enables HLS tools to generate simple pipelines without complex control logic and buffering, but causes deadlocks in this situation.

Pipeline synthesis techniques that supports flushing [20] can be used to avoid this deadlock, but may increase area and are not yet available in most HLS tools. Another approach is to ensure that size of the SAQ is larger than $N = \lceil L/II \rceil$, so that it would not become full before the execute pipeline produces the first output. Often the SAQ size required for performance reasons is already greater than N, then no additional changes are needed in this case.

Deadlock Due to Full Load/Store Queues: A deadlock can occur when the queues are full and form a circular dependency. For example, a load response returns from memory when the load queue and store queue are both full, and the memory system cannot accept another request because it has reached the maximum number of in-flight requests. In this situation, the load queue cannot be drained because the execute unit is stalled trying to write to the full store queue, which is waiting for the memory system, which in turn is waiting for the load queue. This creates a circular dependency, causing a deadlock. This deadlock can be avoided by ensuring that not all queues can become full at the same time. Specifically, we track the number of in-flight load operations and free entries in the load queue, and delay issuing a load if the response would cause the load queue to become full.

E. Customization of Memory Units

The memory unit design described in Section IV-B can be customized to fit the needs of a particular accelerator, providing just enough resources and features but not more. The sizes of various queue structures can be adjusted to fit the accelerator's memory characteristics. For example, if the accelerator rarely performs stores, sizing down the store queue would help save area and energy. If a certain feature is unused by an accelerator or does not help too much, it can be removed. For example, if an memory port is read-only, the memory unit can be made much simpler by removing any store-related features such as store queue and dependency checking logic. As another example, store to load forwarding can be removed if the accelerator does not need it.

F. Automated DAE Accelerator Generation

Figure 9 shows the high-level flow for automatically generating accelerators with access/execute decoupling. Starting from a single source code written in a high-level language, program slicing [21] is used to generate access and execute slices. To generate the access slice, the algorithm backtracks from loads and stores in the Control Data Flow Graph (CDFG)

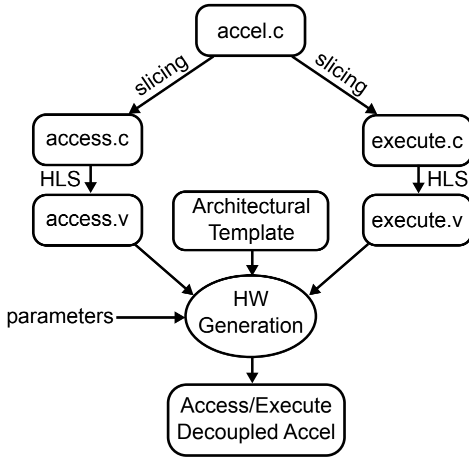

Fig. 9. High-level flow of decoupled access/execute accelerator generation.

of an accelerator and keeps all necessary operations for computing memory addresses, while removing others. To generate the execute slice, the algorithm backtracks from stores and finds all operations needed to compute store values. The slicing process also performs transformations to enable decoupling. In the access slice, stores are transformed into `store_addr` operations, which only have address but not data. Terminal loads are identified as loads that are not in the back slice of address calculations, and are transformed into `load_fwd` operations. In the execute slice, all loads and stores are replaced with queue reads and writes.

We then synthesize the resulting access and execute slices into Verilog RTL using HLS. We implement the memory unit and queue structures as an architectural template written in a hardware generation language PyMTL [22], which allows us to easily generate these modules with configurable parameters to enable full customization as described in Section IV-E. All these components follow the same interface protocol so that they can be easily composed.

The final step of the flow involves setting the parameters and launching the hardware generation process to output the RTL of the access/execute decoupled accelerator.

V. Evaluation

In this section, we present the evaluation results for the proposed data supply framework for accelerators. We first discuss our evaluation methodology and experimental setup. Then, we show the performance, area, and energy results.

A. Methodology

We use a vertically integrated evaluation methodology that combines cycle-level, register-transfer-level, and gate-level modeling.

Cycle-level modeling is used to model the performance of the system components including caches, interconnect, memory controller, and main memory. We use gem5 [23] for this purpose.

TABLE II
SUMMARY OF BENCHMARKS.

Benchmark	Description
bbgemm	Blocked matrix multiplication
bfsbulk	Breadth-first search
gemm	Dense matrix multiplication
mdknn	Molecular dynamics (K-nearest neighbor)
nw	Needleman-Wunsch algorithm
spmvcrs	Sparse matrix vector multiplication
stencil2d	2D stencil computation
viterbi	Viterbi algorithm

Register-transfer-level modeling is used to accurately model the performance of hardware accelerators. Vivado HLS 2015.2 is used to synthesize a C-based description of the accelerators into Verilog. For DAE accelerators, RTL of the memory unit and queue structures are generated from the architectural template. Verilator [24] is used for RTL simulation. We integrated Verilator with gem5 for co-simulation of accelerators and system components.

High-level synthesis involves many parameters and is known to have a large design space [8]. In our experiments, we target high performance instead of low area when setting the parameters. More details can be found in Section V-C.

Gate-level modeling is used to build accurate area and energy models for the accelerators. We synthesized, placed and routed each accelerator using Synopsys Design Compiler and IC Compiler with the TSMC 65nm standard cell library to obtain area numbers. Design Compiler automatically inserts clock gating logic for all designs. Power and energy analysis were performed using Synopsys PrimeTime PX with the switch activity factors obtained from simulations of the place and routed netlist.

B. Experimental Setup

We use a set of eight benchmark accelerators adapted from MachSuite [9] in our experiments. Table II summarizes the accelerators. For each accelerator, we implement a DAE version as well as a baseline version for comparison. Each accelerator has a private L1 cache connected to the DRAM controller through the system bus. For prefetching, we use a stride hardware prefetcher. Table III shows the detailed experiment parameters. We compare the following schemes:

1) **Baseline** is the original accelerator without prefetching or DAE.
2) **Stride** has the stride prefetcher enabled but not DAE. The memory accesses are tagged to facilitate prefetching.
3) **DAE** is the access/execute decoupled implementation, but without the stride prefetcher.
4) **DAE+stride** adds stride prefetching to DAE.

C. Baseline Validation

HLS-based accelerators have a large design space. Depending on the parameters used, the same accelerator can be synthesized to have different area, performance, and power. We use the same set of parameters when synthesizing the baseline

TABLE III
EXPERIMENT PARAMETERS.

Frequency	500MHz
DAE MemUnit	16-entry LQ, 8-entry SQ
Cache	16KB / 2-way / 32B line size / 1 cycle latency / 4 MSHRs
Prefecher	Stride prefetcher, degree=8
DRAM	Single-channel 32-bit LPDDR3-1600, 6.4GB/s BW

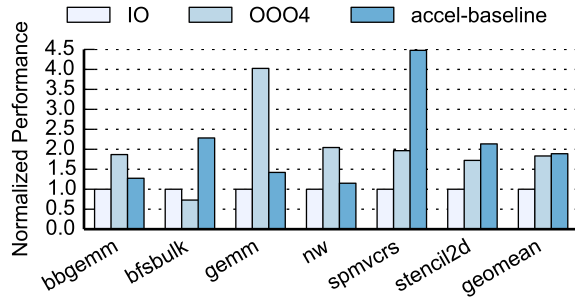

Fig. 10. Comparison of baseline accelerators performance with processors.

and DAE versions to exclude the possibility that the improvement comes from different synthesis parameters. To ensure that the improvement is not from poorly optimized baseline, we apply most HLS optimizations including pipelining and unrolling so that baseline accelerators have best performance within the system-level constraints (such as the number of memory ports or memory bandwidth).

To validate the performance of the baseline, we simulated the performance of functionally equivalent software implementations of these accelerators. Figure 10 shows the performance comparison between in-order, 4-wide out-of-order processors, and the baseline accelerators. Note that `mdknn` and `viterbi` are not included because we use custom-precision fixed-point arithmetic in their implementations, which would be inefficient to emulate in software on processors. On average, the performance of the baseline accelerators is around 2x of an in-order processor, and is comparable to an out-of-order processor. These numbers are roughly in line with previous studies on accelerators [25], [26]. The performance of the baseline accelerators is mainly limited by the memory bottleneck. We will show that with the proposed techniques to enable efficient data supply, the accelerators could achieve much higher performance.

D. Performance Results

Figure 11 compares the performance of the proposed optimization schemes normalized to the baseline accelerator. Overall, the stride prefetcher with memory access tags improves the performance by 45% on average over the baseline. DAE alone achieves an average speedup of 1.89x, while DAE combined with stride prefetching achieves a 2.28x speedup.

Comparing stride prefetching and DAE, DAE usually achieves higher performance due to decoupling and having more precise knowledge about the addresses to be accessed next. One such case is when the access pattern is irregular, but the addresses can be computed early. For example, `mdknn` computes the force between a molecule and its N nearest neighbors. The access pattern is highly irregular because the addresses of the N neighbors in memory usually do not have a pattern. However, the addresses can be computed early because the indices of these N neighbors are known. Hence, the access unit can send out load requests early to hide memory latency. In contrast, the stride prefetcher is unable to predict the addresses and thus unable to prefetch them.

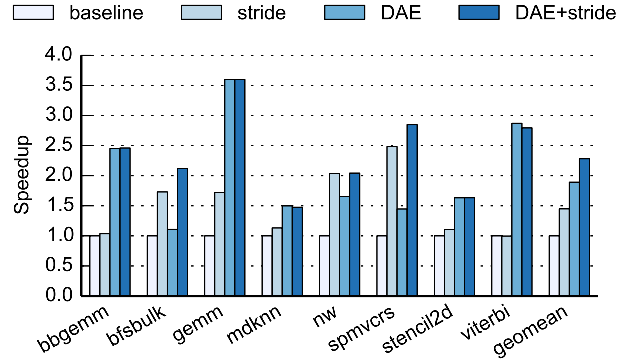

Fig. 11. Performance of the proposed schemes normalized to the baseline accelerator.

DAE also has advantages when the accesses consist of regular but short streams. One example is `viterbi`. The stride prefetcher needs warm-up, thus is too late in sending out prefetch requests. It also prefetches beyond the end of the stream before realizing the stream has ended, wasting memory bandwidth and causing cache pollution. In contrast, DAE has precise information about when the stream begins and ends, thus is able to preload data effectively.

There are some cases where prefetching is more effective than DAE. For example, `bfsbulk` performs a graph traversal, which is dominated by memory accesses with dependencies. As a result, the access unit in the decoupled architecture is not able to pre-calculate the addresses. Prefetching, on the other hand, speculatively fetch data from memory without computing the exact addresses, which improves performance in this case because there is regularity in the access pattern even though the addresses cannot be determined early.

It is also clear from the results that prefetching and DAE can often complement each other, providing a higher speedup compared to using only one of them. For example, in the inner loop of `spmvcrs` (code shown in Section II-C), DAE is able to hide the memory latency for accesses to `val` and `vec`, but not `cols`. Because `cols` is used by the access unit to calculate the address to `vec`, a cache miss for `cols` stalls the access unit. However, the prefetcher can easily detect the strided accesses to `cols` and prefetch it into the cache. As a result, we observe the combined scheme with both DAE and stride prefetching achieves the speedup of 2.85x, which is higher than the speedups, 1.45x and 2.48x respectively, when DAE and prefetching are applied separately.

TABLE IV
AREA AND POWER OF THE BASELINE AND DAE ACCELERATORS. THE AREA UNIT IS μm^2. THE POWER UNIT IS mW. THE ABS COLUMN SHOWS ABSOLUTE NUMBERS, AND THE NORM COLUMN SHOWS RESULTS NORMALIZED TO THE BASELINE.

Bench-mark	Base Area	Base Power	DAE Area		DAE Power	
			Abs	Norm	Abs	Norm
bbgemm	25,191	4.15	52,943	2.10x	8.11	1.96x
bfsbulk	11,507	1.22	14,437	1.25x	1.41	1.16x
gemm	22,127	1.87	47,305	2.14x	3.39	1.81x
mdknn	170,312	32.58	194,034	1.14x	48.40	1.49x
nw	49,094	4.54	89,396	1.82x	8.81	1.94x
spmvcrs	18,686	2.54	42,736	2.29x	4.04	1.59x
stencil2d	27,579	3.88	49,567	1.80x	7.69	1.98x
viterbi	42,963	4.78	80,982	1.88x	11.30	2.36x

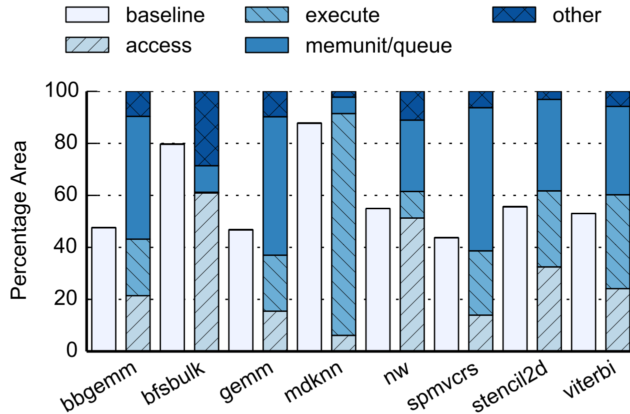

Fig. 12. Area breakdown of DAE accelerators. The baseline area is shown for comparison.

We note that adding prefetching to DAE does not always yield higher performance. For example, we see a slight degradation in performance for `viterbi` when prefetching is enabled. This is because DAE alone can already hide most of the memory latency, while prefetching, due to its imprecise nature, can pollute the cache and contend for memory bandwidth.

E. Area, Power, and Energy Results

Table IV shows the area and power numbers for the baseline and DAE accelerators. The area of DAE accelerators is larger than the baseline by 14% (`mdknn`) to 129% (`spmvcrs`). We note that our area and power analysis only includes the accelerator itself but not the cache, which includes a prefetch unit. The relative overhead will be much lower when the cache, which exists in both the baseline and our architecture, is included.

The area increase comes from several factors: First, the DAE architecture adds additional queues and memory units to accelerators. The area for the queues and memory units are similar across accelerators given that we used the same queue size for all benchmarks. As a result, this overhead represents a large relative overhead for small accelerators such as `spmvcrs`. For larger accelerators such as `mdknn`, the area overhead for queues and memory units only represents a small percentage. Later, we show that this overhead can be reduced significantly by customizing the size of queues for each accelerator.

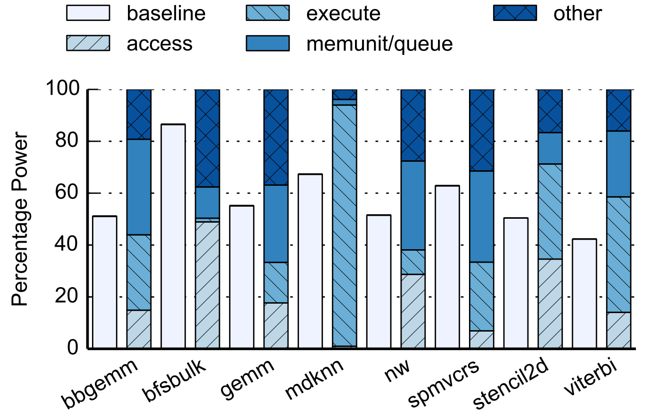

Fig. 13. Power breakdown of DAE accelerators. The baseline power is shown for comparison.

Second, area overhead can come from the reduced resource sharing between the access part and the execute part in the DAE architecture. During the synthesis process, the HLS tool tries to share resources between various parts of the accelerator to reduce area. In the baseline accelerators, such optimizations can be performed across the entire accelerator. For example, a multiplier may be shared between memory address computation logic and value computation logic. In the DAE architecture, such sharing is not possible between the access and execute units because they need to be decoupled and synthesized separately. We note that while reduced resource sharing increases area, it also allows more operations to be performed in parallel and improve performance. The impact of reduced resource sharing is lower for larger accelerators where there are abundant opportunities for resource sharing within the access unit or execute unit.

Figure 12 shows the breakdown of area of the DAE accelerators compared to the baseline. The area is broken down into access unit, execute unit, memory units (including queues), and other components such as configuration registers, miscellaneous control logic, buffers inserted during place and route, etc. The results indicate that the main area overhead comes from the memory units and queues. The combined area of the access unit, the execute unit and other components, which have corresponding logic in the baseline accelerator, is only 13% higher than the area of the baseline on average, indicating the impact of reduced resource sharing is low.

Figure 13 shows the breakdown of power consumption. The percentage of power consumed by memory units and queues ranges from a few percent to around 35%. For `mdknn`, which is relatively large, the power consumption of memory units and queues is only 2.2% of the total power consumption. In addition to the added operations for memory units and queues, the DAE architecture also has higher power consumption compared to the baseline because it has higher activity factors.

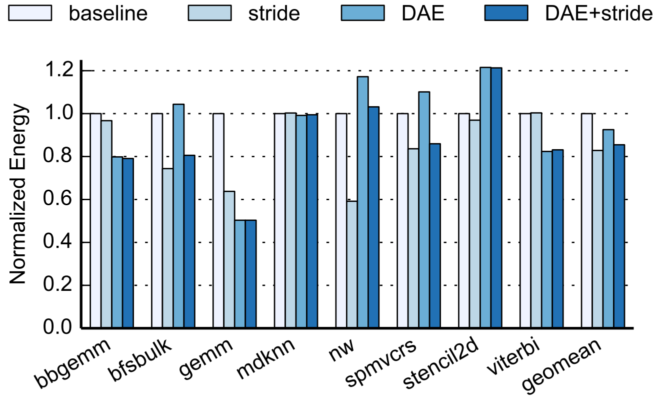

Fig. 14. Energy comparison.

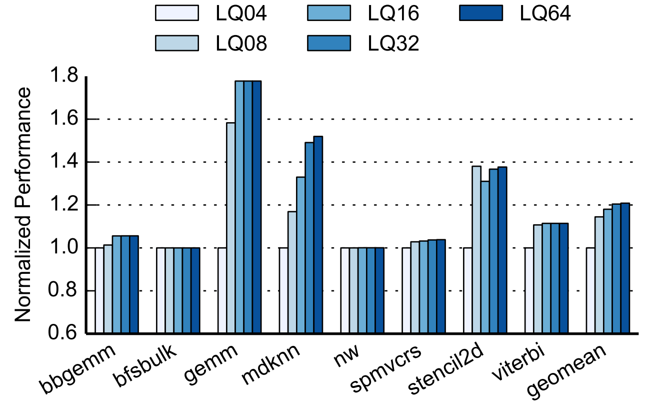

Fig. 16. Performance comparison when varying load queue (LQ) sizes.

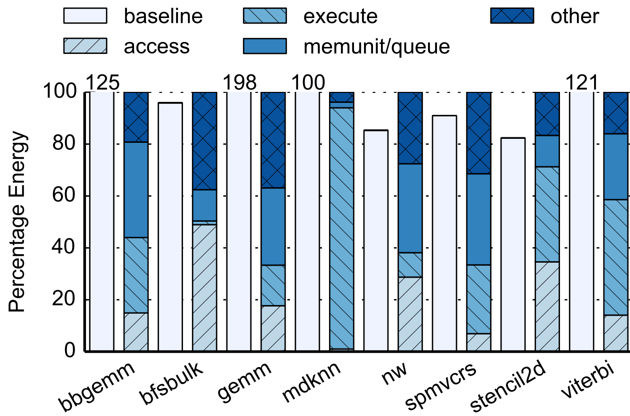

Fig. 15. Energy breakdown of DAE accelerators. The baseline energy is shown for comparison. If baseline energy is higher than DAE, it is annotated with a number on top of the bar.

DAE reduces pipeline stalls waiting for memory accesses and allows accelerators to perform more computations per unit time.

Figure 14 shows the energy consumed by the baseline and DAE accelerators to complete the computation. On average, the stride prefetcher is able to reduce energy for both the baseline and DAE accelerators, by 17.1% and 7.6% respectively. This is because the stride prefetcher is able to reduce memory stalls, leading to shorter execution time. As a result, the accelerators spend less time burning energy without doing useful work.

Compared to the baseline, DAE accelerators often use less or a comparable amount of energy even though they have significantly higher power, which indicates that the higher power mostly comes from doing more useful work per unit time because of reduced pipeline stalls. In cases where DAE accelerators use less energy, it is likely to be because the energy spent while stalling is significant in the baseline. While our design flow automatically inserts clock gating, it does not completely remove static power consumption. Most of these stalls are removed in the DAE accelerators, resulting in lower energy.

Figure 15 shows the breakdown of the energy consumption. The energy numbers are for the baseline and the DAE architecture without prefetching. The percentage of energy consumed by the memory units and queues ranges from 2.2% to 36.8%. Again, the relative overhead is lower for large accelerators compared to small accelerators.

F. Design Space Exploration: Queue Size

The size of queue structures in the decoupled accelerator can impact the performance, area, and energy consumption of the accelerator. Larger queues can provide more decoupling, thus potentially better performance, but may also have larger area and consume more energy. Figure 16 shows the normalized performance of the accelerators when varying load queue sizes from 4 entries to 64 entries. On average, larger load queues yield higher performance, but the improvement diminishes as the queue size increases. This indicates that as long as the access unit runs sufficiently ahead of the execute unit, it can provide the decoupling needed to hide the latency of occasional cache misses.

The results also indicate that some accelerators are less sensitive to queue sizes than others. Thus, accelerator-specific optimization of the queue sizes can be used to reduce the area overhead of decoupled accelerators, with minimal degradation in performance.

Table V shows the impact of customizing queue sizes on the area and performance of DAE accelerators. For each accelerator, we choose a queue size that has lower area but not significantly lower performance, and normalize the area and performance to the configuration with constant 16-entry load queues and 8-entry store queues across all accelerators. On average, the customization reduces queue area and total area by 37% and 15% respectively, while lowering performance by only 2.3%.

TABLE V
IMPACT OF THE QUEUE SIZE CUSTOMIZATION ON AREA AND PERFORMANCE. NUMBERS ARE NORMALIZED TO LQ16/SQ08.

Bench-mark	Custom Size	Total Area		Queue Area		Norm Perf
		Abs	Norm	Abs	Norm	
bbgemm	LQ04/SQ08	40,442	0.76	12,504	0.50	0.94
bfsbulk	LQ02/SQ08	14,437	1.00	1,473	1.00	1.00
gemm	LQ08/SQ02	33,535	0.71	11,423	0.45	0.89
mdknn	LQ16/SQ04	191,183	0.99	9,441	0.77	1.00
nw	LQ04/SQ08	82,449	0.92	17,594	0.72	0.97
spmvcrs	LQ08/SQ02	29,788	0.70	10,582	0.45	0.97
stencil2d	LQ08/SQ02	39,372	0.79	7,232	0.41	1.05
viterbi	LQ08/SQ08	72,661	0.90	19,183	0.70	0.99

VI. RELATED WORK

A. Data Supply for In-Core Accelerators

Accelerators that are tightly integrated into a processor core often rely on the processor pipeline to perform memory accesses. A number of proposals perform memory accesses in a decoupled fashion, following the Decoupled Access/Execute (DAE) paradigm. DAE [17] was originally proposed for in-order processors as a complexity-effective mechanism to address the memory latency problem by dividing a program's instructions into an access stream and an execute stream that run in a decoupled fashion and communicate through architecturally visible queues. Later work extended DAE to out-of-order processors and found that DAE can use two small instruction windows to achieve the effect of a single large instruction window, but with less complexity [27]. In recent work, DeSC [18] explored DAE for heterogeneous architectures and proposed to use an OoO processor core to supply data to a hardware accelerator. MAD [19] proposed to use dataflow to build a specialized engine that executes memory access phases efficiently, which can also be used to supply data to hardware accelerators. Our work differs from previous work as we target stand-alone accelerators that are not tightly integrated with a processor core or dedicated memory access engine. We employ DAE as a paradigm to design accelerators that effectively tolerate the memory latency and thus remove the burden of hand-crafting dedicated memory management logic from accelerator designers.

B. Memory Architecture for Standalone Accelerators

CoRAM [28] is a memory architecture for FPGA-based accelerators. In CoRAM, designers write *control threads* in a C-like language that manages the communication between DRAM and on-chip scratchpad memories. Our work differs from CoRAM in that we provide a framework to automatically transform accelerators into a decoupled architecture, instead of relying on the designer to write application-specific control threads manually.

LEAP [10] is a compiler framework that transforms accelerators that use arbitrary-size scratchpads to use small caches backed-up by a memory hierarchy. It was later extended to handle prefetching [29] but can only use address-based stream localization since accelerators do not have a PC. Our work can improve LEAP by providing better latency tolerance using access/execute decoupling, and enabling more effective prefetching by tagging memory accesses.

CHIMPS [30], [31] is a memory architecture and compilation framework for FPGA accelerators that uses many small, distributed caches implemented using block RAMs. The cache coherence issue is avoided by statically partitioning the memory address space between caches. Our framework can work with this many-cache architecture by connecting each memory unit to a cache and use the same address partitioning technique. This can potentially lead to better performance utilizing higher memory bandwidth.

C. Memory Optimizations in High-Level Synthesis

Deep pipelining is an HLS technique that allocates extra pipeline stages for memory operations in order to tolerate memory latency. However, in cache-based accelerators, it may lower pipeline throughput as it always targets the worst case even though most memory accesses are cache hits.

Tan et al. proposed to synthesize multithreaded pipelines with HLS to tolerate memory latency [32]. The approach mainly targets loop pipelining and allocates a thread for each iteration of a loop. Threads are switched out on a cache miss and stored in a context buffer, and woken up to continue execution when the memory response comes back. This approach achieves good speedup with low resource overhead, but is only applicable to data-parallel kernels where each loop iteration is independent.

Decoupled pipelining was first proposed as a technique to parallelize single-threaded programs. DSWP [33] is a compiler framework that extracts coarse-grained pipeline parallelism from single-thread code and execute using multiple threads. The framework analyzes the program dependence graph and partitions the graph between threads. The threads communicate using message passing. Later work [34] extended it to HLS where an accelerator is transformed into multiple decoupled pipeline stages that communicates through FIFOs. As a result, the impact of a variable-latency memory access can be limited to one stage. Coarse-Grained Pipelined Accelerators (CGPA) [35] extends decoupled pipelining by using multiple workers for pipeline stages that are parallelizable. In comparison, our work uses DAE as the decoupling mechanism and combines hardware prefetching with decoupling to enable more efficient data supply for accelerators.

VII. CONCLUSION

This paper introduces an automated framework to optimize data supply from main memory for HLS-based hardware accelerators. Pipeline stalls due to a long memory latency represents a significant source of performance degradation for hardware accelerators, which often rely on static pipelines without expensive latency-hiding techniques. Today, data supply for accelerators often need to be manually optimized in order to achieve high performance. Instead, this paper shows that hardware prefetching and automated access/execute decoupling based on program slicing can be used as general

mechanisms to optimize accelerators with minimal manual efforts. Experimental results show that the proposed DAE architecture with prefetching can improve accelerator performance significantly and also reduce energy consumption in many cases.

ACKNOWLEDGMENTS

This work was partially supported by the Office of Naval Research (ONR) grant N0014-15-1-2175. We thank the Batten Research Group at Cornell University for sharing their accelerator research infrastructure.

REFERENCES

[1] "Qualcomm Snapdragon 820 Product Brief," https://www.qualcomm.com/documents/snapdragon-820-processor-product-brief.

[2] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 249–260.

[3] E. S. Chung, J. D. Davis, and J. Lee, "LINQits: Big Data on Little Clients," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 261–272.

[4] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 36–47.

[5] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 269–284.

[6] T. Chen, A. Rucker, and G. E. Suh, "Execution Time Prediction for Energy-Efficient Hardware Accelerators," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015, pp. 457–469.

[7] "VP9 Video Hardware RTLs," http://www.webmproject.org/hardware/vp9/.

[8] Y. S. Shao, B. Reagen, G. Wei, and D. M. Brooks, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 97–108.

[9] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. M. Brooks, "MachSuite: Benchmarks for Accelerator Design and Customized Architectures," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2014.

[10] M. Adler, K. Fleming, and A. Parashar, "LEAP Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic," in *Proceedings of the 19th International Symposium on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 25–28.

[11] J. Baer and T. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 1991, pp. 176–186.

[12] K. J. Nesbit and J. E. Smith, "Data Cache Prefetching Using a Global History Buffer," in *Proceedings of the 10th International Conference on High-Performance Computer Architecture (HPCA)*, 2004, pp. 96–105.

[13] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial Memory Streaming," in *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, 2006, pp. 252–263.

[14] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992, pp. 62–73.

[15] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. Lee, D. M. Lavery, and J. P. Shen, "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads," in *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, 2001, pp. 14–25.

[16] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, 2003, pp. 129–140.

[17] J. E. Smith, "Decoupled Access/Execute Computer Architectures," in *Proceedings of the 9th International Symposium on Computer Architecture (ISCA)*, 1982, pp. 112–119.

[18] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: Decoupled Supply-Compute Communication Management For Heterogeneous Architectures," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015, pp. 191–203.

[19] C. Ho, S. J. Kim, and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 118–130.

[20] S. Dai, M. Tan, K. Hao, and Z. Zhang, "Flushing-Enabled Loop Pipelining for High-Level Synthesis," in *Proceedings of the 51st Design Automation Conference (DAC)*, 2014, pp. 76:1–76:6.

[21] M. Weiser, "Program Slicing," in *Proceedings of the 5th International Conference on Software Engineering*, 1981, pp. 439–449.

[22] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," in *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, 2014, pp. 280–292.

[23] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[24] "Verilator," http://www.veripool.org/wiki/verilator.

[25] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 205–218.

[26] V. Govindaraju, C. Ho, and K. Sankaralingam, "Dynamically Specialized Datapaths for Energy Efficient Computing," in *Proceedings of the 17th International Conference on High-Performance Computer Architecture (HPCA)*, 2011, pp. 503–514.

[27] G. P. Jones and N. P. Topham, "A Comparison of Data Prefetching on an Access Decoupled and Superscalar Machine," in *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, 1997, pp. 65–70.

[28] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An In-Fabric Memory Architecture for FPGA-based Computing," in *Proceedings of the 19th International Symposium on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 97–106.

[29] H. Yang, K. Fleming, M. Adler, and J. S. Emer, "Optimizing Under Abstraction: Using Prefetching to Improve FPGA Performance," in *Proceedings of the 23rd International Conference on Field programmable Logic and Applications (FPL)*, 2013, pp. 1–8.

[30] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. J. Eggers, "CHiMPS: A C-level Compilation Flow for Hybrid CPU-FPGA Architectures," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 173–178.

[31] A. Putnam, S. J. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig, "Performance and Power of Cache-Based Reconfigurable Computing," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 395–405.

[32] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded Pipeline Synthesis for Data-Parallel Kernels," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2014, pp. 718–725.

[33] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic Thread Extraction with Decoupled Software Pipelining," in *Proceedings of the 38th International Symposium on Microarchitecture (MICRO)*, 2005, pp. 105–118.

[34] S. Cheng and J. Wawrzynek, "Architectural Synthesis of Computational Pipelines with Decoupled Memory Access," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 83–90.

[35] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August, "CGPA: Coarse-Grained Pipelined Accelerators," in *Proceedings of the 51st Design Automation Conference (DAC)*, 2014, pp. 78:1–78:6.