Compiler Support for Speculation in Decoupled Access/Execute Architectures

Anonymous Author(s)

Abstract

Decoupled access/execute is a popular technique to increase memory parallelism in accelerators. The technique relies on the compiler to separate memory address generation from the rest of the program, but such a separation is not always possible due to control and data dependencies between the access and execute slices, resulting in a loss of decoupling. The loss-of-decoupling problem has traditionally limited the applicability of decoupled access/execute accelerators.

In this paper, we present compiler support for speculation in decoupled access/execute architectures that preserves decoupling in the face of control dependencies. We propose an algorithm that implements speculative memory requests in the access slice and a dual algorithm that kills mis-speculations in the execute slice without the need for costly recovery or synchronization. Our transformations work on arbitrary, reducible control flow and are proven to preserve sequential consistency. Our speculation approach enables the use of the decoupled access/execute technique on codes from the graph and data analytics domain that previously suffered from loss of decoupling, improving average speedup by 1.9× (up to 3×) with virtually no code size and mis-speculation overhead.

ACM Reference Format:
Anonymous Author(s). 2024. Compiler Support for Speculation in Decoupled Access/Execute Architectures. In . ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

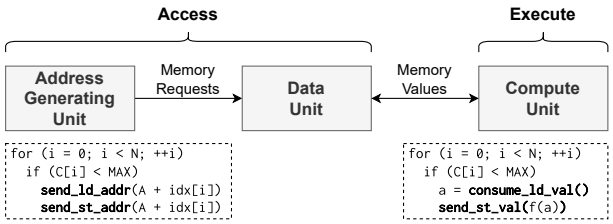
1 Introduction

Irregular codes are common in domains like graph analytic and sparse linear algebra. They are characterized by data-dependent memory accesses and control flow, for example:

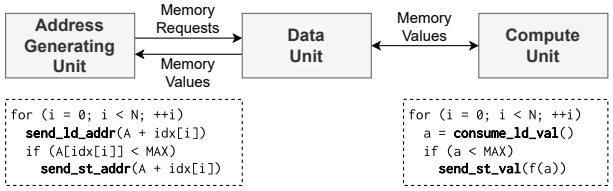
```
for (int i = 0; i < N; ++i)
  if (C[i] < MAX) A[idx[i]] = f(A[idx[i]]);
```


Conference'17, July 2017, Washington, DC, USA

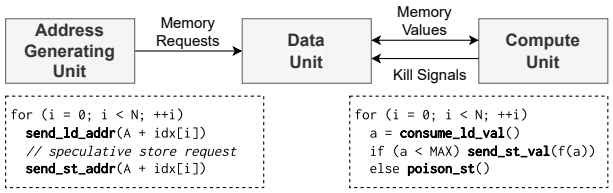
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) An architecture with decoupled address generation, memory access, and compute.



(b) Loss-of-decoupling between address generation and memory access due to a dependency on the memory value.



(c) Our contribution: compiler support for speculation removes loss-of-decoupling due to control dependencies.

Figure 1. A decoupled access/execute accelerator template.

This code has unpredictable control flow that would cause frequent branch mis-predictions on CPUs and thread divergence on GPUs, resulting in low instruction level parallelism. Furthermore, the indirect memory access can make caches ineffective on CPUs and does not fit the coalesced access pattern of GPUs, resulting in low memory level parallelism.

Because of these fundamental architectural limitations, and challenges with Moore's Law and Dennard performance scaling, computer architects are interested in building specialized accelerators for irregular applications [22]. Many of the proposed architectures follow the decades-old idea of a *Decoupled Access/Execute* (DAE) architecture shown in Figure 1. In DAE memory accesses are *decoupled* from computation to avoid stalls in the datapath resulting from unpredictable loads [45]. The address generation unit (AGU) sends load and store requests to the data unit (DU), while the DU sends load values to and receives store values from the compute unit (CU). All communication is FIFO based and ideally the

AGU to DU communication is feed-forward (one-directional), allowing the address streams from the AGU to run ahead w.r.t the CU, which in turn allows the DU to perform timely hazard detection and prefetching. Figure 1a shows such DAE accelerator implementing the earlier code snippet, enabling pipeline parallelism and achieving better performance and energy efficiency than CPUs or GPUs.

In addition to standalone DAE accelerators [9, 10, 12, 13, 16, 33, 41, 42], the DAE principle can also be used in CPUs [14, 20, 34] and GPUs [3, 4]. For example, NVIDIA recently introduced hardware-accelerated asynchronous memory copies between global and shared memory in the Ampere GPU architecture and later extended the idea to the Tensor Memory Unit (TMA) in the Hopper architecture [3]. The CUDA programmer can provide a "copy descriptor" of a tensor to copy and the hardware will run ahead and generate the corresponding addresses in the TMA, leaving the compute threads to do other useful work.

The common denominator of all these DAE architectures is that they rely on either the programmer or the compiler to decouple address-generating instructions from the rest of the program. However, it has long been recognized that such a decoupling is not always possible [7, 49]. If any of the instructions generating an address for an array `A` depend on a value loaded from `A`, then there is a *loss-of-decoupling* (LoD) [21]. Access patterns such as `A[f(A[i])]` are rare, but control dependencies that involve loads from `A` are commonplace. For example, consider replacing `C[i]` with `A[i]` in our running example:

```
for (int i = 0; i < N; ++i)
  if (A[i] < MAX) A[idx[i]] = f(A[idx[i]]);
```

Here, there is a LoD, because the store to `A` is control-dependent on a branch that loads from `A`. Whereas before the load from `C` could be trivially prefetched, now the AGU/DU communication is synchronized, because the AGU needs to receive values from the DU before deciding if a store address should be generated (see Figure 1b). As a result, the AGU cannot run-ahead of the CU anymore LoD is one of the key reasons why DAE accelerators have not traditionally been used to accelerate control-dominated code [8, 46].

A common approach for restoring decoupling in this case is control speculation. As shown in Figure 1c, we can hoist the store request out of the *if*-condition in the AGU and later send a corresponding kill signal in the CU if it turns out that we have mis-speculated. However, it is unclear how the compiler should coordinate the speculation and recovery transformations across two distinct control-flow graphs. While the example from Figure 1c is trivial, the task quickly becomes complicated with more speculated stores and nested control-flow (§4 gives a concrete example). Efficiently squashing speculative computation in a spatial dataflow architecture has always been hard, because the architectural state is distributed [8]. *The key challenge* here is to guarantee that

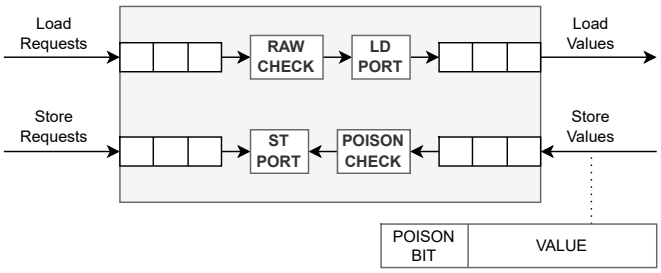


Figure 2. Organization of a typical data unit (DU) in a decoupled access/execute accelerator. Loads can execute out-of-order w.r.t. stores and are checked for read-after write (RAW) hazards. Store values are tagged with a poison bit that is used to drop corresponding mis-speculated store requests.

the order of store requests sent from the AGU, matches the order of store values or kill signals sent from the CU on all control-flow paths.

General compiler support for speculated stores in DAE architectures is an open question that we tackle in this paper, making the following contributions:

- We give a formal description of the fundamental reasons why address generation cannot always be decoupled from the rest of the program (§3).
- We describe compiler support for speculative memory in DAE architectures, effectively solving the loss-of-decoupling problem due to control dependencies. We propose an algorithm for introducing speculative memory requests in the AGU, and an algorithm for poisoning mis-speculations in the CU (§5).
- We prove that our speculation approach preserves the sequential consistency of the original program and does not introduce deadlocks (§6).
- We evaluate our DAE speculation approach on accelerators generated from High-Level Synthesis (HLS) implementing codes from the graph and data analytics domain. We achieve an average 1.9× (up to 3×) speedup over the baseline HLS implementations, which is within 5% of the theoretical peak DAE performance. We show that our approach has no mis-speculation penalty and minimal code increase impact with an average accelerator area increase of < 5% (§7).

2 Background

In this section, we present a model of a typical decoupled access/execute accelerator and we show the impact of loss-of-decoupling on its performance, motivating the use of speculation.

2.1 Accelerator Model

The original Decoupled Access/Execute (DAE) architecture idea proposed to decouple the program into two threads: one for loading and storing values, and one for consuming and

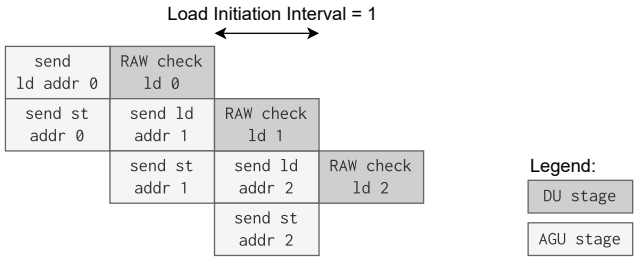
producing them [45]. In later work the access thread was further divided into an address generation unit (AGU) and a data unit (DU), the advantage being that communication between the AGU and execute threads can be avoided [14, 21]. This extension is sometimes referred to as Decoupled Supply/Compute (DeSC) [20, 21, 35], but we refer to it as DAE throughout the paper. A decoupled AGU is part of many academic [4, 9, 12, 14] and industry accelerators [3, 13, 41, 42]. Our Figure 1 DAE model closely follows these works.

The DU in DAE architectures is FIFO based (Figure 2 shows a typical organization). A common memory disambiguation scheme in accelerators is partial ordering – stores execute in order, but loads execute out of order w.r.t stores as long as all previous stores have computed their address [18]. Memory accesses with a regular access pattern can often be disambiguated at compile time [40]. In this paper, we consider accelerators for irregular codes where aliasing cannot be disproved at compile time, requiring read-after-write (RAW) detection in the DU. The RAW check compares the address of the current load with the addresses of all previous stores in program order. On detecting aliasing, the dependent store value is forwarded if it has already been produced or the load is stalled if not. This is a similar operating principle to load-store queues in CPUs and has been used many iregular code accelerators before [1, 20, 27, 47]. Crucially, partial ordering requires a successful decoupling of the address stream.

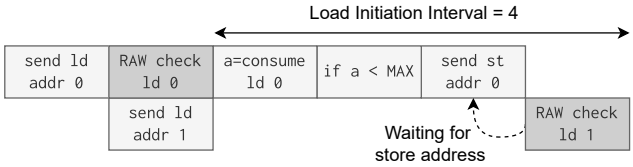
2.2 Motivation for Store Speculation

We motivate the importance of the loss-of-decoupling (LoD) problem by showing its implications on throughput of a DAE accelerator. First, consider the code from Figure 1a that can be decoupled. For the sake of discussion, assume that all operations have a latency of one cycle. Figure 3a shows the AGU pipeline for this code. The AGU is truly decoupled and is able to generate one new load and store address per cycle. This means that, as a given load address arrives at the DU, all previous store addresses in program-order have already arrived, enabling a RAW check without having to wait for further store addresses. Assuming there are no real RAW hazards at runtime, the load initiation interval will be 1.

Now consider the code from Figure 1b that has a LoD. The store address generation is control-dependent on the load value, which means that a store address for iteration i can only be generated 3 cycles after the generation of the load address for iteration i. This in turn results in the load address for iteration $i + 1$ having to wait in the DU until the store address for iteration i has arrived to perform its RAW check. As a result, the load initiation interval increases to 4, even if there are no real RAW hazards at runtime. This 4× throughput decrease is assuming that all operations have a single cycle latency. In practice, the throughput decrease is often higher. It is clear, that using a DAE architecture for codes with LoD would result in terrible performance, motivating the use of speculation to mitigate LoD.



(a) Pipeline of decoupled address generation from Figure 1a.



(b) Pipeline of non-decoupled address generation from Figure 1b.

Figure 3. Comparison of a decoupled and non-decoupled address generation pipeline. Arrows show dependencies for RAW checks. Non-decoupled address generation results in a later arrival of the store address, which stalls the RAW check for the next load, thus lowering load throughput.

2.3 Hardware Speculation Support

We now describe how the DU in a DAE architecture can be extended to support speculative memory operations.

Speculative loads are relatively simple to support, because they are usually side-effect free in accelerators – a mis-speculated load can simply be discarded. A speculative load request could potentially have an out-of-bounds address, but this can be easily handled in the DU (e.g. by omitting out-of-bounds loads and instead returning dummy values).

Speculative stores can be supported by tagging store values with a *poison bit* that, when set, causes the corresponding store request to be dropped in the DU without committing a store. We say that a store request gets killed (or poisoned) if its corresponding store value has the poison bit set. This is a lightweight form of speculation that does not require replays in the compute unit and does not result in out-of-bounds stores, because *mis-speculated stores are never committed.*

Such speculation support resembles the work done on predicated execution [2, 38]. While improvements to branch predictors have made predicated execution less interesting for out-of-order CPUs, the technique is still widely used in cores without branch predictors and in accelerators.

FIFO based DUs mean that the compiler needs to guarantee that the order of store requests (speculative or not) in the AGU matches the order of store values (poisoned or not) in the CU on every possible control-flow path.

2.4 Compiler Preliminaries

We use an SSA-based compiler representation and associated analyses [43]. In particular, we use the control-flow graph

and dominator tree to calculate control dependencies [36], and we use the SSA def-use chain for data dependencies.

Accelerator targeted code is typically loop based, with the performance of innermost loops being critical for overall performance. Here, we discuss implementing speculation within a single loop, but we support arbitrary nested loops (as shown in the evaluation §7). However, we do not execute whole loops speculatively, because, except for very short loops, it would not be beneficial in our context.

We use a canonical loop representation: loops have a single, unique header block and a single loop backedge going from the loop latch block to the loop header block. Our transformation assumes reducible control-flow within loops – CFG edges can be partitioned into two disjoint sets, forward and back edges, such that the forward edges form a directed acyclic graph (DAG). Reducible control flow is a common requirement for accelerator code (most accelerator programming models do not support `goto`).

For completeness, we briefly describe how a compiler targets a DAE architecture.

1. **AGU:** Starting with the original code, for each memory operation to be decoupled, we change it to a `send_address` function that sends the memory address to the **DU**.
2. **CU:** Dually, the CU starts with the original code, but each memory operation to be decoupled is changed to a `consume_value`/`produce_value` function that receive/send values to/from the DU.
3. **Dead code elimination:** We run a standard DCE pass in the CU to remove the now unnecessary address generation code. In the AGU, we delete all side effect instructions that are not part of the address generation def-use chains, and then also run a standard DCE pass. We also use a control-flow simplification pass that removes empty blocks potentially created by DCE.

In our implementation, each array has a private DU, but the AGU is shared by all arrays in the program. The decision to have private/shared DUs and AGUs depends on the DAE implementation and the target codes. For example, we might want to share a DU across all arrays if the target has limited memory ports; or we might want to have private AGUs and DUs for each array if we target hash based algorithms with chains of indirect memory accesses, because the output of one DU might be used in the AGU for another array.

3 Loss-of-Decoupling Analysis

Loss-of-decoupling (LoD) arises when the address generation for a given memory access depends on a load instruction that cannot be trivially prefetched, causing the AGU, Du, and CU communication to be synchronized. The definition of *trivially prefetched* will necessarily depends on the hardware context. In our case it means loads that have have a RAW hazards, i.e. the DU needs to perform dynamic memory disambiguation before executing the load.

Given a set of address-generating instructions G, and a set of memory load instructions A using addresses generated by instructions in G, there is a loss of decoupling if:

Definition 3.1 (LoD Data Dependency). There exists a path in the def-use chain from $a \in A$ to $g \in G$. While encountering a ϕ-node on the def-use chain leading to g, we also trace the def-use paths of the terminator instructions T in the ϕ-node incoming basic blocks to see if any terminator instruction in T depends on any $a \in A$.

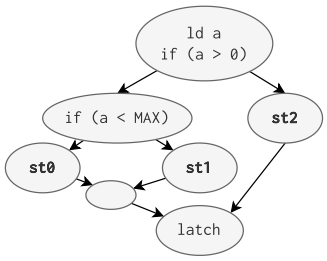
Definition 3.2 (LoD Control Dependency). There exists an instruction $g \in G$ that is control-dependent on a branch instruction b, and there is a path in the def-use chain from $a \in A$ to b. We call the basic block that contains b the *LoD control dependency source*. Note that the LoD control dependency source need not be the immediate control dependency of g, and that g might have multiple LoD control dependencies.

Depending on the hardware context, the definition of the A set could be expanded or narrowed. For example, if the AGU is implemented in hardware with limited control flow support (a common situation in accelerators), then A could also include all branch instructions. This would result in more speculative memory requests, but would remove the need for branches in the AGU. On the other hand, given an address generating instruction, we could limit A to only include loads from the same array for which the given address is generated. This could be useful if we only want to preserve decoupling for that array and do not care about losing decoupling for other arrays. Our speculation technique applies equally well to all these definitions.

An example of a LoD data dependency is an access such as `A[f(A[i])]`. Our speculation technique does not recover decoupling for such cases, but fortunately codes with such an access pattern are rare. An example of a more common data dependency that causes a LoD is the code pattern `if (A[i]) A[i++] = 1`. In this case, the def-use chain leading to the definition of the store address contains a ϕ-node (`i`) whose value depends on loading from A. Such a pattern is sometimes found in algorithms that operate on dynamically growing data structures, e.g. queues or stacks. Our speculation technique does not work on such cases either, but this is not a large limitation in the context of our work, since accelerators typically do not use dynamically growing structures, instead opting for implementations with bounded space requirements that can be allocated statically [53].

An example of LoD due to a control dependency is shown in the motivational example from Figure 1b. This case is much more common than a direct data dependency and is the focus of this paper.

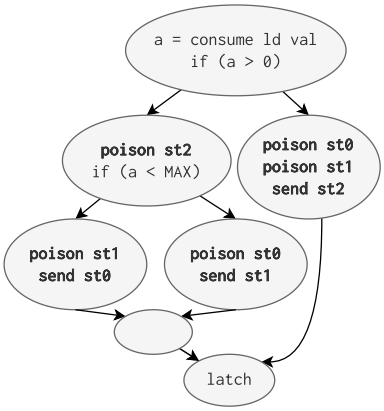
```
for (i = 1; i < N - 1; ++i)
  a = A[i]
  if (a > 0)
    if (a < MAX1)
      A[i + 1] = a + 1 // st0
    else
      A[i - 1] = a + 1 // st1
  else
    A[i] = a + 1       // st2
```



(a) Code and control-flow graph of a loop with three control-dependent stores causing a loss-of-decoupling.

```
// AGU store requets order:
// s0, s1, s2

// Compute unit loop:
for (i = 1; i < N - 1; ++i)
  a = consume_ld_val()
  if (a > 0)
    poison_st2()
    if (a < MAX1)
      poison_st1()
      send_st0(a + 1)
    else
      poison_st0()
      send_st1(a + 1)
  else
    poison_st0()
    poison_st1()
    send_st2(a + 1)
```



(b) Incorrect implementation of speculation. Depending on control flow, the order of store values can be: (s_2, s_1, s_0), (s_2, s_0, s_1), (s_0, s_1, s_2), but only (s_2, s_0, s_1) is correct.

Figure 4. Poisoning speculated stores immediately when they become unreachable results in an ordering mismatch between store requests and values.

4 The Ordering Problem

Before presenting our approach for implementing speculative memory in a DAE architecture, we provide an example of why an obvious approach might be incorrect.

The FIFO-based nature of data units (DUs) and the lack of a program counter in accelerators requires that the order of memory requests (speculative or not) generated in the AGU matches exactly the order of load/store values (poisoned or not) in the CU. The motivating example in Figure 1c contains just one speculative store and one path through the compute CFG where the speculation becomes unreachable, making the problem of ordering trivial in that case.

Consider the more complex code in Figure 4a with three stores s_0, s_1, and s_2. Speculating all store requests in the AGU might result in the store request order (s_0, s_1, s_2). In the CU, we need to guarantee the same order of corresponding store values (poisoned or not) on every possible control-flow path through the loop. Unfortunately, the obvious approach that works for the trivial example in Figure 1c does not work here. If we poison values at points where the corresponding speculations become unreachable, as illustrated in Figure 4b, we end up with three possible orderings of store values depending on the CFG path in the CU, but only one of the orderings is correct. This is why any previous implementations of speculative stores in DAE architectures has only considered trivial triangle or diamond shaped CFGs [20], like the one in 1c. Generalized compiler support for store speculation that guarantees the correct order of poisoning is the *key challenge* that we solve in the next section.

Algorithm 1 Control-flow hoisting of AGU requests

Input: *srcBlocks* list of blocks that are the source of a LoD control dependency (defined in §3)
Output: *SpecReqMap* { basic block: list of hoisted requests to this block }

for $srcBB \in srcBlocks$ **do**
  ▷ *traverse the DAG from srcBB to the loop latch*
  for $fromBB \in reversePostOrder(srcBB)$ **do**
    if $fromBB$ contains memory requests **then**
      hoist $fromBB$ requests to the end of *srcBB*
      add requests to $SpecReqMap[srcBB]$

5 Compiler Support for Speculation

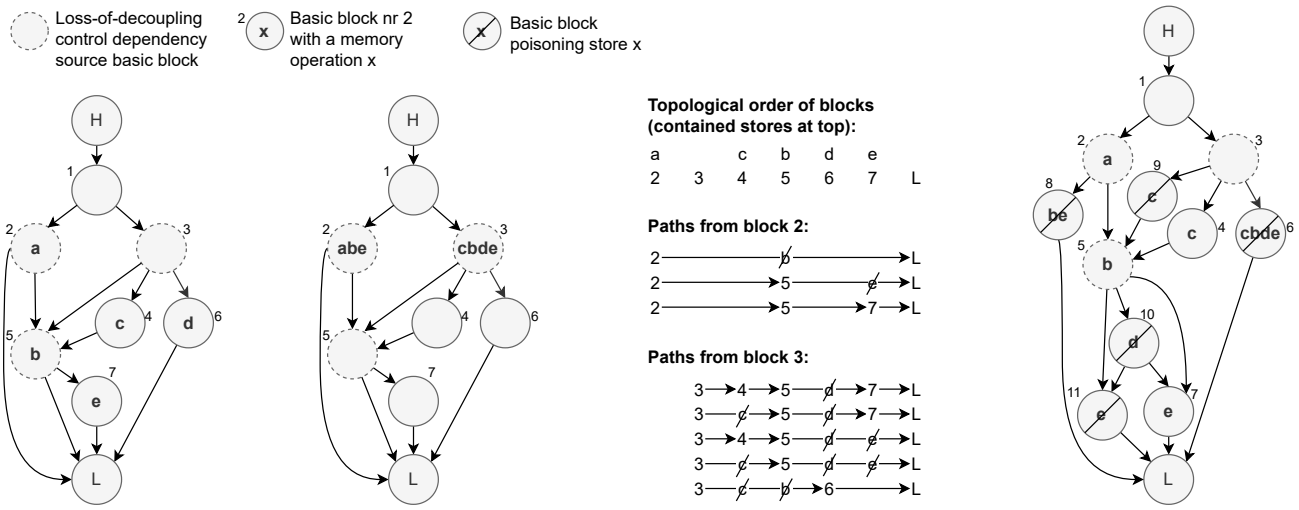
We now describe our dual compiler transformations that enable speculation in the address generation unit (AGU) and send kill signals for mis-speculations in the compute unit (CU).

5.1 Speculating Memory Requests

Algorithm 1 describes our approach to introducing speculation in the AGU. Given a loss-of-decoupling (LoD) control dependency source block *srcBB*, we hoist all memory requests that are control dependent on *srcBB* to the end of *srcBB*. There can be multiple blocks with memory requests that have a LoD control dependency on *srcBB*, which poses the question in which order should they be hoisted to *srcBB*. We use reverse post-order in Algorithm 1. Assuming reducible control flow, the CFG region from *srcBB* to the loop latch is a directed acyclic graph (DAG). The reverse post-order of a DAG is its topological order. Topological ordering gives us the useful property that given two distinct basic blocks A and B in a given loop, if $A \prec B$ in any path through the loop then $A \prec B$ in the topological ordering. Note that there can be multiple topological orderings for a DAG, but it does not matter which one is chosen in our algorithm.

Algorithm 1 traverses the CFG region from *srcBB* to the end of its loop (or to the end of the function if *srcBB* is not any loop). During the traversal, we ignore CFG edges leading to loop headers – we do not enter loops other than the innermost loop containing *srcBB*.

5.1.1 Example of hoisting. Consider the CFG from Figure 5a. There are three LoD control dependency source



(a) Original CFG with memory operations a, b, c, d, e.

(b) AGU CFG with speculative requests (Algorithm 1).

(c) Insertion of poison stores on CU CFG paths (Algorithm 2).

(d) Final CU CFG with poison calls and blocks (Algorithm 3).

Figure 5. An example of introducing speculative memory requests in the AGU (§5.1); and poisoned stores in the CU (§5.2).

blocks (2, 3, 5) and five blocks with memory requests (blocks 2, 4, 5, 6, 7 with memory requests a, c, b, d, e, respectively). Assume that the blocks hold a single memory requests – multiple memory requests within the same block are treated in the same way by our algorithms. Figure 5c shows the topological order of the loop (block 1 is omitted for brevity). Algorithm 1 will hoist b, e to the end of block 2, and c, d, e to the end of block 3 – the result is presented in Figure 5b. Note that the requests b and e were hoisted to both block 2 and 3, because they are reachable from both blocks. Nothing is hoisted to block 1 since it is not a LoD control dependency source.

5.1.2 Nested LoD control dependencies.

Block 5 in Figure 5b does not contain any speculative requests because it itself has a LoD control dependency on block 2 and 3. Algorithm 1 considers only LoD control dependency source blocks that are not themselves the destination of another LoD control dependency. Given a chain of nested LoD control dependencies, we only consider the head of the chain. For example, the CFG in Figure 5a has two chains of LoD control dependencies 2, 5 and 3, 5, but we only consider blocks 2 and 3 in Algorithm 1.

5.1.3 Why topological ordering in Algorithm 1?

Topological order traversal is needed to make it possible to match the order of speculative requests made in the AGU with the order of values that will arrive from the CU on all its possible CFG paths. Consider the example of memory requests b and c in Figure 5a. We first want to hoist c to block 3 before hoisting b, because there exists a CFG path where c comes before b, but not vice versa. If b were hoisted before c, then the speculative requests order would be $b \prec c$ which would be impossible to match with values in the CU on the CFG path 3, 5, 7.

Algorithm 2 Mapping Poison Stores to CFG Edges in CU

```
1:  Input: SpecReqMap { basic block: list of requests hoisted to this block in Algorithm 1 }
2:
3:  for specBB, specRequests ∈ SpecReqMap do
4:      for path ∈ allPathsToLoopLatch(specBB) do
5:          trueBlocks ← ∅                ▷ set keeps insertion order
6:          for r ∈ specRequests do
7:              trueBB ← block where r is true
8:              trueBlocks.insert(trueBB)
9:          for edge ∈ path do
10:             for trueBB ∈ trueBlocks do
11:                 if edge_dst = trueBB then
12:                     trueBlocks.remove(trueBB)
13:                     break                ▷ to the next edge
14:                 if trueBB not reachable from edge_dst then
15:                         ▷ reachability ignores loop backedges
16:                     poison trueBB requests on edge ▷ Alg. 3
17:                     trueBlocks.remove(trueBB)
```

5.2 Poisoning Mis-speculated Stores

Our strategy for killing misspeculations in the CU is to first map poison calls to CFG edges, and then to map poisoned CFG edges to poison store calls contained in basic blocks.

Algorithm 2 describes the first step. Given block *specBB* that contains speculative memory requests *specRequests*, we consider each path in the DAG from the *specBB* to the loop latch in the CU. We call the block where a $r \in$ *specRequests*

Algorithm 3 Poisoning Stores on Edges in CU

Input: store request r; CFG $edge$; block $specBB$ where r was speculated; block $trueBB$ where r is true

$poisonBlockReuse \leftarrow \emptyset$ ▷ *preserve set across calls*
if $edge_{dst}$ is reachable from $trueBB$ **then**
  $poisonBB \leftarrow$ create new block on $edge$ **or**
    get from $poisonBlockReuse$ if exists
  append $poison(r)$ to the end of $poisonBB$
  $poisonBlockReuse.insert(poisonBB)$
else if $specBB$ does not dominate $edge_{dst}$ **then**
  $poisonBB \leftarrow$ create new block on $edge$
  append $poison(r)$ to the end of $poisonBB$
    ▷ *create recursively on specBB → $edge_{src}$ paths*
  create $\phi(1, specBB)$ value in $edge_{src}$
  branch from $edge_{src}$ to $poisonBB$ on $\phi = 1$
else
  append $poison(r)$ to the start of $edge_{dst}$

becomes true the $trueBB$ (for example, the $trueBB$ for request b in Figure 5a is block 5). For each CFG path, we use the $trueBlocks$ list to keep track of which requests were already used or poisoned on the path – the list contains the $trueBB$ for each $r \in specRequests$.

Given an edge in the traversal, the edge is skipped if the next $trueBB \in trueBlocks$ is still reachable from $edge_{dst}$. This guarantees that the order of speculative requests in the AGU matches the order of values in the CU, i.e. a speculative request for a given $trueBB$ block is not poisoned immediately when $trueBB$ becomes unreachable if there is an earlier speculative request that can still be used.

5.2.1 Example of mapping poison stores to CFG edges.

Figure 5c shows which CFG edges are poisoned given the original CFG in Figure 5a and the AGU CFG in Figure 5b. For example, the path $3 \rightarrow 5 \rightarrow L$ will have: $poison(c)$ on the $3 \rightarrow 5$ edge; and $poison(d)$, $poison(e)$ on the $5 \rightarrow L$ edge (4th path from BB 2 in Figure 5c).

5.2.2 Mapping poisoned edges to basic blocks.

Algorithm 3 shows how poisoned CFG edges are mapped to actual poison calls placed in a concrete basic block. Given a poisoned request r on $edge$, there are three cases:

1. There exists a path from $trueBB$ to $edge_{dst}$. In this case, we cannot insert $poison(r)$ in $edge_{dst}$, because we would end up with a CFG path where the store is both true and poisoned. To avoid this, we create a new $poisonBB$ block on $edge$ and append $poison(r)$ it.
2. There exists a path from the loop header to $edge_{dst}$ that does not contain $specBB$. In this case, we cannot insert $poison(r)$ in $edge_{dst}$, because we would end up with a CFG path where r was not speculated in the AGU, but was poisoned in the CU. To avoid this, we create a new block $poisonBB$ on the edge and append $poison(r)$ to it. We also add steering instructions to the path from $specBB$ to $poisonBB$ that will branch from $edge_{src}$ to $poisonBB$ only if $specBB$ was encountered on the current CFG path.
3. Otherwise, $poison(r)$ can safely be prepended to the start of $edge_{dst}$.

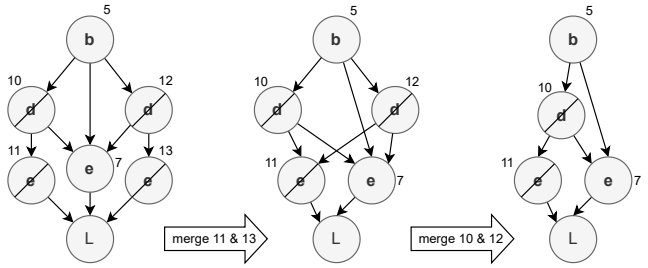


Figure 6. Basic blocks with the same list of poison stores and the same immediate successor can be merged in the CU.

Algorithm 3 is executed only once per $(edge, r)$ tuple – a given request is poisoned at most once on a given edge. Also, poison blocks created in case 1 in Algorithm 3 can be reused to poison other requests.

5.2.3 Example of mapping poison edges to basic blocks.

Consider how the poisoned edges in Figure 5c are mapped to basic blocks in Figure 5d.

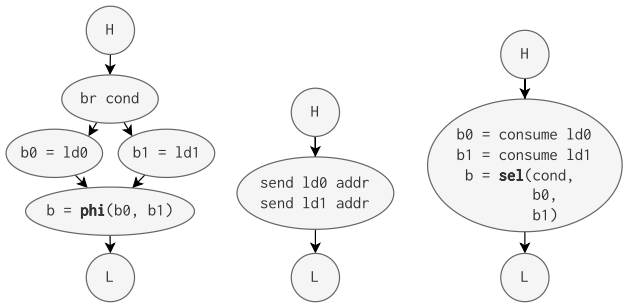
Case 1: Store c is poisoned on the $3 \rightarrow 5$ edge. Since there is a path from the true block of c (block 4) to the edge destination block (block 5), we create a new block on the $3 \rightarrow 5$ edge and append $poison(c)$ to it.

Case 2: Store d is poisoned on both the $5 \rightarrow 7$ and $5 \rightarrow L$ edges. The $specBB$ for d is block 3. Since there exists the path $H \rightarrow 1 \rightarrow 2 \rightarrow 5$ that does not contain block 3, we create a new block on the $5 \rightarrow 7$ edge with the $poison(d)$ call. We add steering instructions to the $3 \rightarrow 5$ and $3 \rightarrow 4 \rightarrow 5$ paths that will cause block 5 to branch to the new poison block on the $5 \rightarrow 7$ edge only if block 5 was reached from a path containing block 3.

Case 3: Store c is also poisoned on the $3 \rightarrow 6$ edge, but here it is safe to prepend $poison(c)$ to the start of block 6.

5.3 Merging poison blocks

Case 1 and 2 of Algorithm 3 might create multiple poison blocks for the same store on different CFG edges. It is possible to merge two poison blocks into one if they contain the same list of poison stores and if they have the same list of immediate successors (when merging, we keep instructions from just one block). We apply such merging iteratively after Algorithms 2 and 3. For example, Figure 6 contains a CFG sub-region of our running example from Figure 5. Algorithm 3 inserted poison blocks 10, 11, 12, 13 to poison stores d and e. Block pairs (11, 13) and (10, 12) can be merged.



(a) Original CFG. **(b)** AGU CFG with hoisted loads. **(c)** CU CFG with a select instruction instead of a phi.

Figure 7. The hoisting of `load consume` calls in the CU requires changing ϕ-nodes to select instructions.

5.4 Speculative Load Consumption

To match the order of `load_consume` calls in the CU with the order of speculative `send_load_addr` calls in the AGU we can hoist the `load_consume` calls to the same block where the corresponding `send_load_addr` were hoisted in the AGU. Then, the CU can either use the load value or discard it. After hoisting, we need to update all ϕ instructions that use the load value, since the basic block containing the loaded value will have changed. Alternatively, we can transform ϕ instructions using the load value into `select` instructions, as shown in Figure 7.

6 Safety and Liveness

We prove that our transformations preserve the sequential consistency of the original program and that they do not introduce deadlock. Deadlock freedom is a corollary of sequential consistency, so we focus only on the latter. We show that on every CFG path the order of speculative store requests in the AGU matches the order of store values in the CU, and that the non-poisoned store value sequence in the CU matches the store sequence of the original code.

In the following discussion, we assume blocks with a single store; the proof trivially extends to blocks with multiple stores since all speculative stores in the same block are treated the same. We also assume that all stores are speculative, since the relative order between non-speculative and speculative stores is guaranteed by definition. Given a a non-speculative store s_1 and a speculative store s_2, our Algorithm 1 will not change the relative program order of s_1 and s_2, i.e. if $s_1 \prec s_2$ in the original program order, then it is not possible to hoist s_2 such that $s_2 \prec s_1$. This follows from the control dependency definition (§3) – s_2 hoisting stops at its LoD control dependency source *srcBB*, which must come after the block containing s_1 in topological order. If *srcBB* would come after s_1 in topological order, then the block containing s_1 would also have a LoD control dependency on *srcBB* and would have been hoisted, which is a contradiction since we assumed that s_1 was non-speculative. A similar argument can be made if $s_2 \prec s_1$ in the original program order.

Lemma 6.1 (Sequential Consistency)**.** Given an ordered list of n speculative store requests $L_a = \{a_0, a_1, ..., a_{n-1}\}$ made in the AGU loop CFG on some fixed iteration k, Algorithms 2 and 3 transform the CU CFG such that every possible path through its loop CFG on iteration k produces an ordered list of n tagged store values $L_v = \{(v_0, p_0), (v_1, p_1), ..., (v_{n-1}, p_{n-1})\}$, such that each $(a_i, v_i, p_i), 0 \leq i < n$ triple corresponds to a $A[a_i] \leftarrow v_i$ store in the original program CFG, and $p_i = 1$ (poison bit) if that store is not executed on the path through the original loop CFG on iteration k.

Proof. We use a proof by induction on the transformed CFG.

Base case: $L_a = \emptyset$ (no speculated requests in the AGU). Algorithm 2 does not change the CU CFG. Thus, the order of store addresses in the AGU and store values in the CU trivially matches, $L_a = L_v = \emptyset$.

Inductive hypothesis: assume Lemma 6.1 holds at basic block B_i in the current CFG path. All store requests $a_j \in L_a$ contained in blocks reached before B_i in the path were matched with the correct store value call $(v_j, p_j) \in L_v$, such that $p_j = 1$ if $A[a_j] \leftarrow v_j$ was not executed on the path in the original loop CFG.

Inductive step: The next store address in the AGU L_a sequence is $a_{j+1} \in L_a$. The next store value in the CU CFG path should be $(v_{j+1}, p_{j+1}) \in L_v$, where $p_{j+1} = 1$ iff the store $A[a_{j+1}] \leftarrow v_{j+1}$ is not reached on the current CFG path in the original program. Algorithm 2 considers the $edge_{src} \rightarrow edge_{dst}$ next. There are three cases:

1. $edge_{dst} = trueBB$, where $trueBB$ is the block containing the store $A[a_{j+1}] \leftarrow v_{j+1}$ in the original program CFG. In this case, Algorithm 2 will not poison this store on this path through the CU CFG, i.e. the next item in the L_v sequence will be the correct $(v_j, 0)$.
2. $edge_{dst} \neq trueBB$ and $trueBB$ is not reachable from $edge_{dst}$, in which case Algorithm 2 will insert a poison store on this edge. Algorithm 3 will map this poison store to a basic block, with the effect that taking the *edge* will result in the poison call being executed and control transferring to $edge_{dst}$. The next item in the L_v sequence will be the correct $(v_j, 1)$.
3. $edge_{dst} \neq trueBB$ and $trueBB$ is reachable from B_l, in which case Algorithm 2 will traverse the path until Case 1 or 2 is matched.

Since Lemma 6.1 holds for the base case, for basic blocks on the path up to B_i, and for some successor block of B_i, it must hold at any block on the path. If it holds at any block on the path, it holds for the whole path. Since a given store request r is poisoned at most once on a given CFG edge and since, by definition of Algorithm 2, any given path will contain at most one edge where r is poisoned, we conclude that Lemma 6.1 holds for all paths. □

7 Evaluation

In this section, we answer the following questions:

- What is the performance benefit of using a DAE architecture (enabled by our speculation approach) to accelerate codes with LoD control-dependencies?
- What is the cost of mis-speculation in our approach?
- What is the impact on code size (accelerator area usage) of our speculation approach?
- How do these metrics scale for nested control-flow, which increases the number of poison store locations?

We make our implementation and evaluation publicly available[1].

7.1 Methodology

We answer the above questions using accelerators generated from High-Level Synthesis (HLS) targeting an Intel Arria 10 FPGA. The C input codes are taken directly from benchmark suites without adding any HLS specific annotations (with the exception of dynamic data structures, e.g. queues, where these were replaced with HLS specific libraries). We use the LLVM based Intel SYCL HLS compiler [24] applying the standard DAE transformation (§2.4) and our proposed speculation transformation (§5) in the middle-end as LLVM passes. The codes use deterministic dual-ported on-chip SRAM capable of 1 read and 1 write per cycle. To enable out-of-order loads, we use a load-store queue (LSQ) designed for HLS (load/store queue sizes of 4/32), which is commonly found on accelerators for irregular codes [1, 20, 27, 47]. We run cycle-accurate simulations of the generated accelerators using ModelSim and report cycle counts. Accelerator area usage is obtained after place and route using Quartus 19.2.

7.1.1 HLS as an evaluation vehicle. Commercial [13, 41] and academic DAE [4, 9, 12, 14] architectures are often domain specific, limiting the types of codes that can be evaluated; employ non-deterministic memory hierarchies with caches, which pollute measurements; and are difficult to reproduce. Our DAE architecture model synthesized from HLS is simple, yet representative of the DAE methodology, and using deterministic memory allows us to be precise in our conclusions of the benchmark results.

7.1.2 Baselines. For each benchmark, we synthesize the following architectures:

- `INORD`: the default approach of current HLS tools (a single modulo-scheduled datapath). Loads that cannot be disambiguated at compile time execute in-order.
- `DAE`: a decoupled access/execute architecture without speculation. OoO loads are enabled by an LSQ. This approach suffers from control-dependency LoDs.
- `SPEC`: the same as `DAE`, but with our speculation technique which mitigates control-dependency LoDs.
- `ORACLE`: the same as `DAE`, but all LoD control dependencies are removed manually from the input code. The `ORACLE` results are wrong, but give a bound on the performance of `SPEC` and show its area overhead.

[1] https://doi.org/10.5281/zenodo.13373865

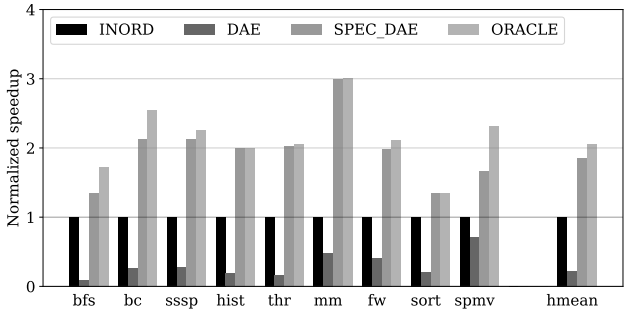


Figure 8. Performance for `DAE`, `SPEC` and `ORACLE` normalized to the `INORD` approach. Our `SPEC` approach achieves an average 1.9× (up to 3×) speedup.

7.1.3 Benchmarks. DAE architectures optimize the latency between memory and compute and are most beneficial for memory bound codes [20], especially codes with an irregular memory access pattern that prevents static prefetching [14]. We evaluate nine such benchmarks from the graph and data analytics domain, using the GAP graph benchmark suite [6] and an HLS benchmark suite [11] of irregular programs. We select only codes that can benefit from our `SPEC` approach, i.e. codes with LoD control-dependencies:

- `bfs`: breadth-first traversal through a graph.
- `bc`: betweenness centrality of a single node in a graph.
- `sssp`: single shortest path from a single node to all other nodes in a graph using Dijkstra's algorithm.
- `hist`: histogram, similar to Figure 1b (size 1000).
- `thr`: zeroes RGB pixels above threshold (size 1000).
- `mm`: maximal matching in a bipartite graph (2000 edges).
- `fw`: Floyd-Warshall distance calculation of all node to node pairs in a dense graph (10 × 10 distance matrix).
- `sort`: using bitonic mergesort (size 64).
- `spvm`: sparse vector matrix multiply that skips zero columns (20×20 matrix).

The small input sizes are due to long simulations times and limited on-chip memory capacity. For the graph codes (`bfs`, `bc`, `sssp`) we use a real-world graph `email-Eu-core` with 1005 nodes and 25,571 edges.

7.2 Performance

Figure 8 reports normalized speedups of each technique over `INORD`. Our `SPEC` approach gives on average a 1.9× (and up to 3×) speedup. This is within 5% of the `ORACLE` performance. Applying the `DAE` approach without speculation to these benchmarks sees a dramatic performance degradation, because the AGU, DU, CU communication is sequentialized.

Table 1. Absolute performance and area usage of INORD, DAE, SPEC and ORACLE. (*bc uses two LSQs).

Kernel	Poison		Mis-spec.	Cycles				Area (ALMs [23])			
	Blocks	Calls	rate	INORD	DAE	SPEC	ORACLE	INORD	DAE	SPEC	ORACLE
bfs	1	1	95%	37,243	398,616	27,561	21,569	7,361	7,525	13,404	13,706
bc	2	2	95%, 82% *	109,061	406,178	51,109	42,942	9,709	10,859	16,582	16,558
sssp	1	1	95%	108,995	391,426	51,227	48,208	10,565	11,668	17,426	17,395
hist	1	1	2%	2,061	11,100	1,033	1,031	2,391	2,807	3,117	3,137
thr	1	3	97%	2,131	13,147	1,052	1,034	5,662	6,144	6,278	6,622
mm	1	2	31%	12,164	25,125	4,069	4,044	5,076	4,986	7,813	7,528
fw	1	1	85%	6,821	16,485	3,433	3,238	3,407	4,210	4,008	4,007
sort	1	2	49%	2,358	11,109	1,748	1,746	2,814	4,361	5,260	5,269
spmv	1	1	32%	13,319	18,693	8,028	7,984	3,895	5,085	4,416	4,336

7.2.1 Mis-speculation cost.

The performance gap between SPEC and ORACLE is highest on the bfs and bc codes, because of its deep pipeline between the load and store that form a RAW hazard. The deep pipeline means that more store allocations need to be held by the LSQ [30] to guarantee perfect pipelining. This, together with a high mis-speculation rate in these benchmarks (Table 1), can cause the LSQ to fill with store addresses that are mis-speculated, potentially stalling later loads that have to wait for future store addresses to arrive. This problem can be solved by increasing the store queue size in the LSQ. The increased number of requests and the need for more buffering is one of the limitations of our approach. Codes with a shallower pipeline that do not need large LSQ sizes have no mis-speculation penalty.

To prove this, we choose three benchmarks where we can instrument the input data so that we can vary the mis-speculation rate. Table 2 shows how the mis-speculation cost changes as the mis-speculation rate increases. As can be seen, there is no correlation between the mis-speculation rate and cost, with the slight variability in clock cycle counts attributable to the subtle difference in the number of true RAW hazards due to the varying data distribution.

Table 2. Change in cycle counts of our SPEC approach as the mis-speculation rate changes.

Kernel	Mis-speculation rate						σ
	0%	20%	40%	60%	80%	100%	
hist	1044	1013	1029	1029	1012	1051	16
thr	1082	1109	1047	1073	1058	1071	21
mm	4107	4096	4074	4063	4106	4081	18

7.3 Code Size

Our speculation approach can increase the number of blocks in the CU, especially for codes with deeply nested control flow. An increased number of blocks can result in a higher area usage due to increased scheduler complexity [44].

Table 1 shows absolute area usage of all accelerator architectures, using the Adaptive Logic Module (ALM) as a unit of area [23]. The area number combines the area usage of all AGUs and CUs implementing the code (we omit LSQs). We observe virtually no area overhead of SPEC over ORACLE on the evaluated benchmarks. This is because most of the codes have at most two control-flow nesting levels where new poison blocks are inserted. Furthermore, sometimes it is possible reduce the number of poison blocks using our merging technique (e.g. two poison blocks in mm were merged into one).

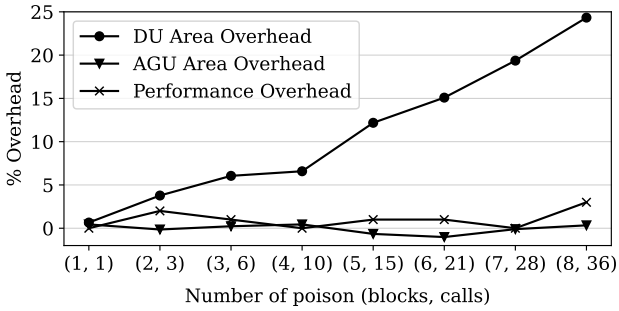


Figure 9. Change in area and performance overhead of SPEC over ORACLE as the number of poison blocks and calls grows.

We omit frequency from Table 1 because all approaches achieve values within 10% of each other, with no clear trend – our approach has no impact on circuit frequency.

7.3.1 Impact of nested control-flow on area.

To give a more meaningful measure of how nested control-flow impacts the area overhead of our SPEC approach, we create a synthetic benchmark template where we can tune the number of poison blocks generated by SPEC:

if $x > 0$ **then**
  $store_1$
  if $x > 1$ **then**
    $store_2$
    if $x > 2$ **then**
      ...

Each nesting level in this template will result in one poison block in the SPEC architecture. With n stores and assuming one store per nesting level, there will be n poison blocks and $\sum_{i=1}^{n} i = \frac{n \times (n+1)}{2}$ poison calls.

Figure 9 shows how the area and performance overhead of SPEC over ORACLE changes as more poison blocks are needed in SPEC. The performance overhead is close to 0% and does not change with more poison blocks. The area overhead of the AGU unit is similarly close to 0%, because SPEC hoists stores out of the *if*-conditions, causing the blocks to be deleted. The area overhead of the DU unit grows by a few percent (< 5%) with each added poison block, but even for the pathological case of eight nested *if*-conditions the overhead is below 25%. In real codes, with more compute and lower control-flow nesting, the area overhead of SPEC should be minimal (within a few percent, as demonstrated in Table 1).

8 Related Work

Program slicing is used beyond DAE architectures. Decoupled Software Pipelining (DSWP) [37] is a popular compiler transformation that decouples strongly connected components in the program dependence graph into separate pipeline stages mapped over multiple processing elements (PEs) communicating via FIFOs. The PEs can be CPU threads, or pipeline stages in an accelerator generated by HLS [29]. Control dependent pipeline stages in DSWP can also be executed speculatively, although stages with memory operations require versioned memory [50]. Other decoupling policies than the original DSWP formulation are also possible [10, 33, 34], as is time-multiplexing multiple pipeline stages over the same PE [39, 51].

Control speculation has its roots in compilers for VLIW machines. Instruction scheduling in HLS is very similar to VLIW scheduling (no hardware support for speculation, static mapping to functional units, etc.), with many algorithms like modulo-scheduling and *if*-conversion originally developed for VLIW directly applicable to HLS [2, 38, 44]. Most recently, predicated execution in the form of gated SSA was proposed for HLS with speculation support [19]. The speculation support in this and other works requires costly recovery on mis-speculation [5, 17, 26, 31, 48, 52]. Efficiently squashing speculative computation on the wrong paths in a spatial dataflow architecture is extremely hard, because the architectural state is distributed [8]. Our speculative DAE sidesteps this issue and does not require any recovery: we speculate early (run ahead) in the AGU, and later handle mis-speculations in the CU by taking an appropriate path in its CFG.

Decoupled Access/Execute (DAE) accelerators were discussed in our background section (§2), but the technique can also be applied to CPUs to improve load prefetching [32, 35] or to enable compiler-controlled Dynamic Voltage Frequency Scaling (DVFS) [25]. Ham *et al.* proposed to integrate explicit DAE support in an OoO core, achieving load latencies close to a perfect cache [20]. Their approach, called Decoupled Supply-Compute Communication (DeSC), relies on compiler support to perform the decoupling. They mention architectural support for poisoning mis-speculated store addresses (similar to our approach §2.1), but fall short of providing general compiler support that handles arbitrary control flow. Our work could be directly used to provide such compiler support in DeSC.

Control-flow handling in GPUs is most commonly implemented via **predication**. The algorithms used to calculate predicate masks and reconvergence points bear a striking resemblance to our work [28]. The SIMT stack approach in GPUs pushes predicate masks onto the stack when entering a control-flow nesting level, and performs a pop when exiting. Our Algorithm 1 implementing speculative requests can be seen as a pass through the CFG with only push operations, where the push is onto individual stacks of control-dependency sources. Dually, our Algorithm 1 inserting poison calls can be seen as a pass through the CFG with only pop operations where the placement of the pops follows a certain policy instead of popping at the immediate post-dominator as in the traditional SIMT formulation. Modern SIMT implementations often employ CFG path analysis to optimize the placement of pops to prevent SIMT deadlock/livelock or to improve performance [15].

9 Conclusion

We have presented general compiler support for speculative memory operations in decoupled access/execute (DAE) accelerators that tackles the loss-of-decoupling problem resulting from control-control dependencies. We proposed CFG transformations implementing speculation in the address generation slice, and to store poisoning in the compute slice of a DAE architecture, with a proof of correctness. Our approach has no mis-speculation cost and has a small code size footprint (< 5% average area usage in the accelerators). We show an average 1.9× (up to 3×) speedup over non-DAE accelerators on a set of benchmarks from the graph and data analytics domain where the DAE approach is not possible without speculation.

References

[1] Mythri Alle, Antoine Morvan, and Steven Derrien. 2013. Runtime dependency analysis for loop pipelining in High-Level Synthesis. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–10. https://doi.org/10.1145/2463209.2488796

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085

[3] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. 2022. NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[4] José-María Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2012. Boosting mobile GPU performance with a decoupled access/execute fragment processor. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, USA, 84–93.

[5] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran, and Wen-mei W. Hwu. 1998. Integrated predicated and speculative execution in the IMPACT EPIC architecture. *SIGARCH Comput. Archit. News* 26, 3 (apr 1998), 227–237. https://doi.org/10.1145/279361.279391

[6] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[7] Peter L. Bird, Alasdair Rawsthorne, and Nigel P. Topham. 1993. The effectiveness of decoupling. In *Proceedings of the 7th International Conference on Supercomputing* (Tokyo, Japan) *(ICS '93)*. Association for Computing Machinery, New York, NY, USA, 47–56. https://doi.org/10.1145/165939.165952

[8] M. Budiu, P.V. Artigas, and S.C. Goldstein. 2005. Dataflow: A Complement to Superscalar. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://doi.org/10.1109/ISPASS.2005.1430572

[9] Tao Chen and G. Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783749

[10] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 69–80. https://doi.org/10.1145/3431920.3439290

[11] Jianyi Cheng. 2019. *JianyiCheng/HLS-benchmarks: HLS_Benchmarks_First_Release*. https://doi.org/10.5281/zenodo.3561115

[12] Shaoyi Cheng and John Wawrzynek. 2014. Architectural synthesis of computational pipelines with decoupled memory access. In *2014 International Conference on Field-Programmable Technology (FPT)*. 83–90. https://doi.org/10.1109/FPT.2014.7082758

[13] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38 (March 2018), 8–20. https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/

[14] Neal Clayton Crago and Sanjay Jeram Patel. 2011. OUTRIDER: efficient memory latency tolerance with decoupled strands. *SIGARCH Comput. Archit. News* 39, 3 (jun 2011), 117–128. https://doi.org/10.1145/2024723.2000079

[15] Ahmed ElTantawy and Tor M. Aamodt. 2016. MIMD synchronization on SIMT architectures. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 11, 14 pages.

[16] Shane T. Fleming and David B. Thomas. 2017. Using Runahead Execution to Hide Memory Latency in High Level Synthesis. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 109–116. https://doi.org/10.1109/FCCM.2017.33

[17] Hagen Gädke and Andreas Koch. 2008. Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens. In *Reconfigurable Computing: Architectures, Tools and Applications*, Roger Woods, Katherine Compton, Christos Bouganis, and Pedro C. Diniz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–195.

[18] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. 2010. Processor Microarchitecture: An Implementation Perspective. In *Transactional Memory* (2nd ed.), Tim Harris, James Larus, and Ravi Rajwar (Eds.). Morgan and Claypool Publishers, Chapter 1.

[19] Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. 2024. A Unified Memory Dependency Framework for Speculative High-Level Synthesis. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction* (Edinburgh, United Kingdom) *(CC 2024)*. Association for Computing Machinery, New York, NY, USA, 13–25. https://doi.org/10.1145/3640537.3641581

[20] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 191–203. https://doi.org/10.1145/2830772.2830800

[21] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2017. Decoupling Data Supply from Computation for Latency-Tolerant Communication in Heterogeneous Architectures. *ACM Trans. Archit. Code Optim.* 14, 2, Article 16 (jun 2017), 27 pages. https://doi.org/10.1145/3075620

[22] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* (2019). https://doi.org/10.1145/3282307

[23] Mike Hutton, Jay Schleicher, David Lewis, Bruce Pedersen, Richard Yuan, Sinan Kaptanoglu, Gregg Baeckler, Boris Ratchev, Ketan Padalia, Mark Bourgeault, Andy Lee, Henry Kim, and Rahul Saini. 2004. Improving FPGA Performance and Area Using an Adaptive Logic Module. In *Field Programmable Logic and Application*, Jürgen Becker, Marco Platzner, and Serge Vernalde (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 135–144.

[24] Intel. [n. d.]. Intel/LLVM. https://github.com/intel/llvm/tree/sycl

[25] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. 2018. Fix the code. Don't tweak the hardware: A new compiler approach to Voltage-Frequency scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 262–272. https://doi.org/10.1145/2544137.2544161

[26] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. 2019. Speculative Dataflow Circuits. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 162–171. https://doi.org/10.1145/3289602.3293914

[27] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2022. From C/C++ Code to High-Performance Dataflow Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022). https://doi.org/10.1109/TCAD.2021.3105574

[28] Adam Levinthal and Thomas Porter. 1984. Chap - a SIMD graphics processor. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. Association for Computing Machinery, New York, NY, USA, 77–82. https://doi.org/10.1145/800031.808581

[29] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. 2014. CGPA: Coarse-Grained Pipelined Accelerators. In *Proceedings of the 51st Annual Design Automation Conference* (San Francisco, CA, USA) *(DAC '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2593069.2593105

[30] Jiantao Liu, Carmine Rizzi, and Lana Josipović. 2022. Load-Store Queue Sizing for Efficient Dataflow Circuits. In *2022 International*

Conference on Field-Programmable Technology (ICFPT). 1–9. https://doi.org/10.1109/ICFPT56656.2022.9974425

[31] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. 1992. Sentinel scheduling for VLIW and superscalar processors. *SIGPLAN Not.* 27, 9 (sep 1992), 238–247. https://doi.org/10.1145/143371.143529

[32] Ajeya Naithani, Jaime Roelandts, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout. 2023. Decoupled Vector Runahead. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 17–31. https://doi.org/10.1145/3613424.3614255

[33] Quan M. Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1064–1077. https://doi.org/10.1145/3466752.3480048

[34] Quan M. Nguyen and Daniel Sanchez. 2023. Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1262–1274. https://doi.org/10.1109/HPCA56546.2023.10071026

[35] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L. Aragón, David Wentzlaff, and Margaret Martonosi. 2022. Tiny but mighty: designing and realizing scalable latency tolerance for manycore SoCs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 817–830. https://doi.org/10.1145/3470496.3527400

[36] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. [n. d.]. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI '90*.

[37] G. Ottoni, R. Rangan, A. Stoler, and D.I. August. 2005. Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. 12 pp.–118. https://doi.org/10.1109/MICRO.2005.13

[38] Joseph CH Park and Mike Schlansker. 1991. *On predicated execution*. Hewlett-Packard Laboratories Palo Alto, California.

[39] Michael Pellauer, Angshuman Parashar, Michael Adler, Bushra Ahsan, Randy Allmon, Neal Crago, Kermin Fleming, Mohit Gambhir, Aamer Jaleel, Tushar Krishna, Daniel Lustig, Stephen Maresh, Vladimir Pavlov, Rachid Rayess, Antonia Zhai, and Joel Emer. 2015. Efficient Control and Communication Paradigms for Coarse-Grained Spatial Architectures. *ACM Trans. Comput. Syst.* (2015). https://doi.org/10.1145/2754930

[40] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/2435264.2435273

[41] Raghu Prabhakar and Sumti Jairath. 2021. SambaNova SN10 RDU:Accelerating Software 2.0 with Dataflow. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–37. https://doi.org/10.1109/HCS52781.2021.9567250

[42] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 389–402. https://doi.org/10.1145/3079856.3080256

[43] Fabrice Rastello. 2016. *SSA-based Compiler Design* (1st ed.). Springer Publishing Company, Incorporated.

[44] B. Ramakrishna Rau. 1994. Iterative modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*. https://doi.org/10.1145/192724.192731

[45] James E. Smith. 1982. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Austin, Texas, USA) *(ISCA '82)*. IEEE Computer Society Press, Washington, DC, USA, 112–119.

[46] Michael Sung, Ronny Krashinsky, and Krste Asanović. 2001. Multithreading decoupled architectures for complexity-effective general purpose computing. *SIGARCH Comput. Archit. News* 29, 5 (dec 2001), 56–61. https://doi.org/10.1145/563647.563658

[47] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2023. Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis. In *2023 33nd International Conference on Field-Programmable Logic and Applications (FPL)*.

[48] Benjamin Thielmann, Jens Huthmann, and Andreas Koch. 2012. Memory Latency Hiding by Load Value Speculation for Reconfigurable Computers. *ACM Trans. Reconfigurable Technol. Syst.* (2012). https://doi.org/10.1145/2362374.2362377

[49] N. Topham, A. Rawsthorne, C. McLean, M. Mewissen, and P. Bird. 1995. Compiling and Optimizing for Decoupled Architectures. In *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. 40–40. https://doi.org/10.1145/224170.224301

[50] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative Decoupled Software Pipelining. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 49–59. https://doi.org/10.1109/PACT.2007.4336199

[51] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. 2021. Aurochs: An Architecture for Dataflow Threads. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 402–415. https://doi.org/10.1109/ISCA52012.2021.00039

[52] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 268–281. https://doi.org/10.1109/ISCA45697.2020.00032

[53] Zeping Xue and David B. Thomas. 2016. SynADT: Dynamic Data Structures in High Level Synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 64–71. https://doi.org/10.1109/FCCM.2016.26