Compiler Support for Speculation in Decoupled Access/Execute Architectures

Robert Szafarczyk
University of Glasgow
Glasgow, United Kingdom
robert.szafarczyk@glasgow.ac.uk

Syed Waqar Nabi
University of Glasgow
Glasgow, United Kingdom
syed.nabi@glasgow.ac.uk

Wim Vanderbauwhede
University of Glasgow
Glasgow, United Kingdom
wim.vanderbauwhede@glasgow.ac.uk

Abstract

Irregular codes are bottlenecked by memory and communication latency. Decoupled access/execute (DAE) is a common technique to tackle this problem. It relies on the compiler to separate memory address generation from the rest of the program, however, 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.

In this paper, we present compiler support for speculation in DAE architectures that preserves decoupling in the face of control dependencies. We speculate memory requests in the access slice and poison mis-speculations in the execute slice without the need for replays or synchronization. Our transformation works on arbitrary, reducible control flow and is proven to preserve sequential consistency. We show that our approach applies to a wide range of architectural work on CPU/GPU prefetchers, CGRAs, and accelerators, enabling DAE on a wider range of codes than before.

CCS Concepts: • **Hardware** → **Emerging languages and compilers**; • **Software and its engineering** → *Compilers.*

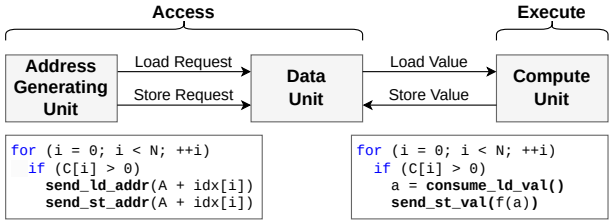
Keywords: decoupled access/execute; compiler speculation

ACM Reference Format:
Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2025. Compiler Support for Speculation in Decoupled Access/Execute Architectures. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC '25), March 1–2, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3708493.3712695

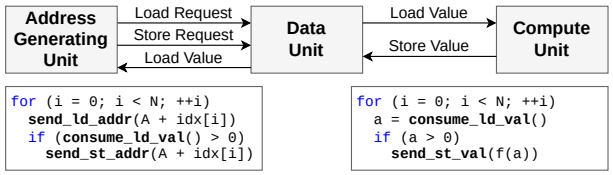
1 Introduction

Irregular codes are characterized by data-dependent memory accesses and control flow, for example:

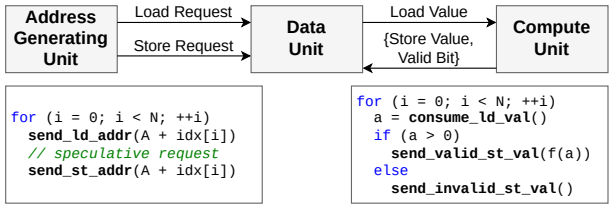
```
for (int i = 0; i < N; ++i)
    if (C[i] > 0)
        A[idx[i]] = f(A[idx[i]]);
```



(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.

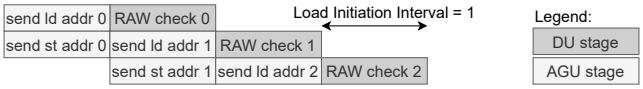
Figure 1. A decoupled access/execute architecture template.

This code has unpredictable control flow that causes frequent branch mis-predictions on CPUs and thread divergence on GPUs. Because of these limitations, and challenges with Moore's Law and Dennard performance scaling, computer architects are interested in adding CPU/GPU structures to accelerate such code patterns, or even to use accelerators specialized for a given algorithm [26].

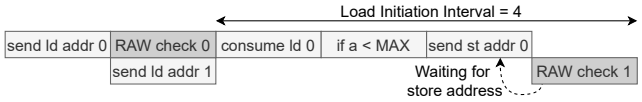
Many of the proposed architectures follow the decades-old idea of a *Decoupled Access/Execute* (DAE) architecture shown in Figure 1. In DAE, memory accesses are *decoupled* from computation to avoid stalls resulting from unpredictable loads [51]. The address generation unit (AGU) sends load and store 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 one-directional, allowing the

CC '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1407-8/25/03
https://doi.org/10.1145/3708493.3712695



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



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

Figure 2. Comparison of a decoupled and non-decoupled address generation. Non-decoupled address generation results in a later arrival of the store address, which stalls the RAW check for the next load, lowering load throughput.

address streams from the AGU to run ahead w.r.t the CU. Figure 1a shows an example of such a DAE architecture implementing the earlier code snippet.

DAE is a general technique applicable to many computational models: it is used in specialized FPGA accelerators generated from High-Level Synthesis (HLS) [11–13, 15, 16, 21, 54, 55]; in Coarse Grain Reconfigurable Architectures (CGRAs) [20, 27, 37, 39, 43, 45, 46, 61]; and in CPU/GPU prefetchers [3, 5, 17, 18, 24, 38, 47, 59]. For example, NVIDIA introduced hardware-accelerated asynchronous memory copies [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 a Tensor Memory Unit.

The common denominator of all these works 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 [9, 57]. If any of the address-generating instructions for array `A` depend on a value loaded from `A`, then there is a *loss-of-decoupling* (LoD) [25]. Access patterns such as `A[f(A[i])]` are rare, but control dependencies that involve loads from `A` are common. For example, consider replacing `C[i]` with `A[i]` in our running example:

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

Here, there is a LoD, because the `A` store is control-dependent on a branch that loads from `A`. Whereas before the load from `C` could be prefetched, now the AGU/DU communication is synchronized, because the AGU waits for `A` values from the DU before deciding if a store address should be generated, as shown in Figure 1b. In turn, the load waits for the store address to ensure that there is no aliasing—the store address is needed for memory disambiguation. As a result, the AGU cannot run ahead of the CU anymore, resulting in decreased pipeline parallelism, which Figure 2 illustrates.

One 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 (speculation), and later *poison* the store in the CU on mis-speculation (store invalidation). 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, as we demonstrate in the next section. The *key challenge* here is to guarantee that 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 (§4).
- We describe compiler support for speculative memory in DAE architectures, solving the LoD problem due to control dependencies. We propose two algorithms: one for speculating memory requests in the AGU, and one 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 show that our work enables DAE on a wider class of codes than before, with applications in CPU/GPU prefetchers, CGRAs, and FPGA accelerators.
- We evaluate our DAE speculation approach on accelerators generated from 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. We show that our approach has no mis-speculation penalty and minimal code size impact (average accelerator area increase $< 5\%$) (§8).

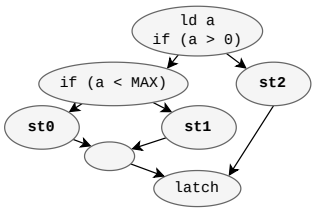
2 Motivating Example

In this brief section, we show why an obvious approach to speculation in DAE architectures is incorrect.

The FIFO-based nature of DAE 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.

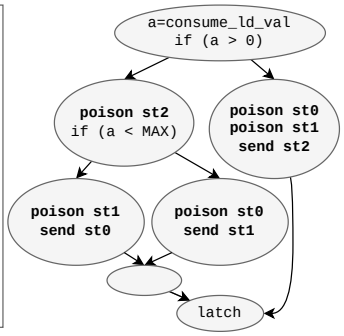
Consider the more complex code in Figure 3a with three stores s_0, s_1, and s_2. Speculating all store requests in the AGU might result in the store request order (s_2, s_0, s_1). In the CU, we need to guarantee the same order of corresponding store values (poisoned or not) on every possible control-flow path through the loop. Unfortunately, the obvious approach that worked for the trivial example in Figure 1c does not work

```
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.

```
// Assume AGU st request order:
//      s2, s0, s1
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) 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 3. Poisoning speculated stores immediately when they become unreachable results in an ordering mismatch between AGU store requests from and CU store values.

here. If we poison values at points where the corresponding speculation becomes unreachable, as illustrated in Figure 3b, then 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 [24], like the one in Figure 1c. Generalized compiler support for store speculation that guarantees the correct order of poisoning is the *key challenge* that we solve in this paper.

3 Background

In this section, we describe the architectural support needed to enable speculative DAE and some compiler preliminaries.

3.1 Architectural Support

Our speculation technique requires architectural support for predicated stores and FIFOs. Store values are tagged 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 CU and does not result in out-of-bounds stores, because *mis-speculated stores are never committed*. Speculative loads can be supported by simply discarding the value of a mis-speculated load.

Predicated stores are easy to support in hardware since the underlying memory protocol usually already uses a *valid* signal. For example, the commonly used AXI4 interface [4] has a strobe signal to indicate which bytes are valid. Architectural FIFOs (queues) are also commonly added in works on CPU/GPU microarchitecture or can be relatively cheaply implemented in software. For example, works on DAE CPU/GPU prefetchers add architectural FIFOs and extend the ISA with instructions for producing load/store addresses and consuming/producing store values [3, 5, 17, 18, 24, 38, 47].

The prefetcher from [24] enables predicated stores with a *store_inv* instruction, but the authors support only simple triangle or diamond control flow patterns, calling for future work on general speculation support. We discuss concrete examples of architectures that can benefit from our work in §7. We evaluate our work on accelerators generated from HLS, where we have complete control of the memory interfaces.

3.2 Compiler Preliminaries

We use an SSA-based compiler representation and associated analyses [49]. In particular, we use the control-flow graph and dominator tree to calculate control dependencies [40], and we use the SSA def-use chain for data dependencies.

We use a canonical loop representation: loops have a single header block and a single loop backedge going from the loop latch to the loop header. Our transformation assumes reducible control flow—CFG edges can be partitioned into two disjoint sets, forward and back edges, such that the forward edges form a directed acyclic graph (DAG). Irreducible control flow can be made reducible with node splitting [7, 44].

For completeness, we briefly describe how our compiler implements a DAE architecture:

1. **AGU:** For each memory operation to be decoupled, we change it to a `send_ld_addr` or a `send_st_addr` function that sends the memory address to the **DU**.
2. **CU:** Dually, in the CU we change each decoupled memory operation to a `consume_val` or `produce_val` function that receives or sends values to or 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.

The `send_ld_addr`, `send_st_addr`, `consume_val`, and `produce_val` functions are implementation dependent. For example, if we target CPU/GPU prefetchers, such as [17, 24], then these would translate to instructions. For accelerators, they would be translated to FIFO writes/reads.

4 Loss-of-Decoupling Analysis

LoD events arise when the address generation for a given memory access depends on a load that cannot be trivially prefetched, causing the AGU, DU, and CU communication to be synchronized. By *non-trivially prefetched* we mean loads that have a RAW hazards, i.e., the DU needs to receive all previous store addresses in program order to perform 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 4.1 (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 4.2 (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 can be expanded or narrowed. For example, if the AGU is implemented in hardware with limited control flow support, then A could include all branch instructions. 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 the `A[f(A[i])]` access. Our speculation approach does not recover decoupling for such cases, but fortunately such accesses are rare. An example of a more common LoD data dependency 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, since performance oriented codes typically do not use dynamically growing structures, instead opting for implementations with bounded space requirements that can be allocated statically [62].

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

Algorithm 1 Control-flow hoisting of AGU requests

```
Input: srcBlocks list of blocks that are the source of a LoD control dependency (defined in §4)
Output: SpecReqMap { basic block: list of hoisted requests to this block }

for srcBB ∈ srcBlocks do
        ▷ traverse the DAG from srcBB to the loop latch
    for fromBB ∈ 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 transformations that enable speculation in the AGU and poison mis-speculations in the CU.

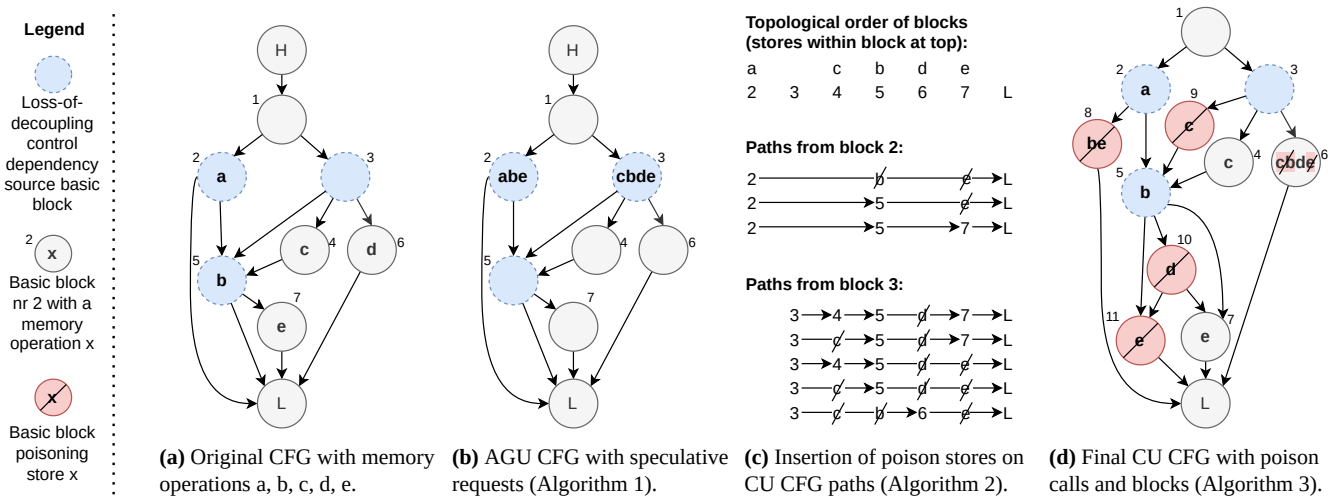
5.1 Speculating Memory Requests

Algorithm 1 describes our approach to introducing speculation in the AGU. Given a 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 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 in a 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 4a. There are three LoD control dependency source blocks (2, 3, 5) and five blocks with memory requests (blocks 2, 4, 5, 6, 7 with requests a, c, b, d, e, respectively). Assume that each block holds a single memory request—multiple memory requests within the same block are treated in the same way by our algorithms. Figure 4c 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 4b. 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.



(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 4. An example of introducing speculative memory requests in the AGU (§5.1); and poisoned stores in the CU (§5.2). Block 6 in subfigure (d) kills stores c, b, then uses the allocation for store d, and then kills store e.

5.1.2 Nested LoD Control Dependencies.

Block 5 in Figure 4b 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 chain head. For example, the Figure 4a CFG has two LoD control dependency chains: 2, 5 and 3, 5—Algorithm 1 considers only blocks 2 and 3.

5.1.3 Why Topological Order in Algorithm 1?

Topological order 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, for example, the requests b and c in Figure 4a. 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 poisoning misspeculations in the CU is to first map a poison call to a CFG edge, and then to map that edge to a poison store call contained in an existing or newly created basic block.

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 (or function exit) in the CU. We call the block where a $r \in specRequests$ becomes true the *trueBB* (for example, the *trueBB* for request b in Figure 4a 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 4c shows which CFG edges are poisoned given the original CFG in Figure 4a and the AGU CFG in Figure 4b. For

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 $edge_{dst}$

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 block 3 in Figure 4c).

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 $edge$ (from $edge_{src}$ block to $edge_{dst}$ block), the $specBB$ block where r was speculated in the AGU, and $trueBB$ where r becomes true 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 $poison(r)$ to 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 $edge_{dst}$.

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.

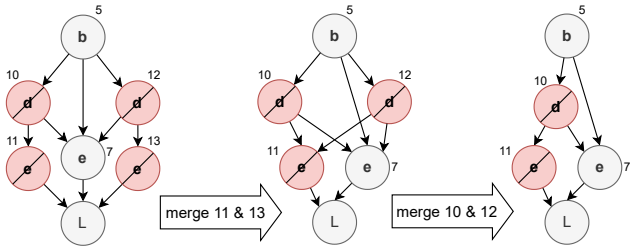


Figure 5. Basic blocks with the same list of poison stores and the same immediate successor can be merged in the CU.

5.2.3 Example of Mapping Poison Edges to Blocks.

Consider how the poisoned edges in Figure 4c are mapped to basic blocks in Figure 4d.

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 5 contains a CFG subregion of our running example from Figure 4. 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.

5.4 Speculative Load Consumption

Speculative loads are relatively easy to support. 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.

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, 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 (§4)—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.

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 Applications

In this section, we highlight three applications for our work: DAE-based prefetchers in CPUs/GPUs, CGRAs, and specialized accelerators generated from HLS. In the next section, we choose HLS as an evaluation vehicle due to its simplicity compared to CPU/GPU prefetchers where the evaluation results can easily be polluted by other architectural factors like cache behavior, branch prediction, etc. However, we emphasize that our speculation support in DAE does not rely on any HLS-specific features and can be applied wherever speculation is combined with the DAE technique.

7.1 CPU/GPU Prefetchers

Most existing works on CPU/GPU prefetchers follow the DAE principle and rely on the compiler to decoupled address generation from compute [3, 5, 17, 18, 24, 38, 47]. All of these works suffer from the control-dependency loss of decoupling (LoD) problem (§4). The work in [24] discusses adding speculation and predicated stores to the CPU microarchitecture to mitigate LoD, but their compiler only supports simple diamond and triangle control flow shapes. In this paper, we have demonstrated generalized compiler support for speculation in DAE, making these works viable for general control flow and thus applicable to a broader set of codes.

Example. The CPU prefetcher proposed in [24] (on which most of the other work is based) separates address generation from compute and extends the ISA with `store_addr`,

`load_produce`, `store_val`, `load_consume`, and `store_inv` instructions that can be directly targeted by our compiler.

7.2 Coarse Grain Reconfigurable Architectures

A CGRA consists of an array of PEs, each with small memories, connected by a network. A CGRA compiler is typically co-designed with the hardware, as the PEs are typically statically scheduled. The job of the compiler is to map the Control/Data Flow Graph (CDFG) to the PEs, and many works follow the DAE technique to tackle the memory wall problem [20, 27, 37, 39, 43, 45, 46, 61]. Our work can help mitigate LoD events when mapping to CGRAs.

Example. The CGRA proposed in [39] is an example of a modern streaming dataflow CGRA. All communication in the CGRA is FIFO-based, and address generation is explicitly decoupled at compile time into AGUs. The compiler generates commands to produce address streams, and to consume or produce values. Control flow is handled with predication and there is a *SD_Clean_Port* command to throw away a value from an output port that can be used to implement predicated stores.

7.3 High-Level Synthesis

In HLS, the CDFG of an algorithm is implemented directly in hardware following a spatial execution model with the freedom to customize the memory system. This makes decoupling easier in HLS compared to the temporal CPU/GPU execution model. HLS-generated accelerators can directly benefit from our work today without any changes, and it is in this domain that we evaluate our implementation in the next section.

Although existing HLS compilers are successful in building non-trivial accelerators for regular code (e.g., [48]), their static scheduling techniques are sub-optimal for irregular codes (for the same reason why traditional VLIW compilers were sub-optimal for irregular codes). Many research works in academia and industry have exploited DAE in HLS to improve the efficiency of HLS-generated accelerators for irregular codes [11–13, 15, 16, 21, 53–55]. By adding compiler speculation support, DAE in HLS can be used on a broader set of codes, which we demonstrate in the next section.

8 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?
- What is the scalability for nested control flow, which increases the number of poison stores and blocks?

We make our work and evaluation publicly available [52].

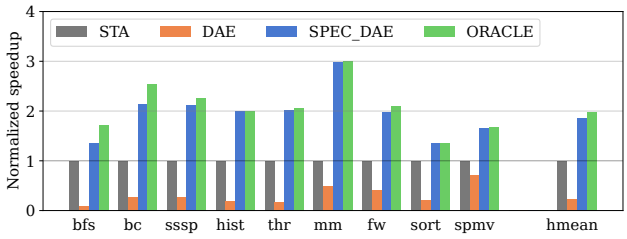


Figure 6. Performance of `DAE`, `SPEC` and `ORACLE` normalized to `STA`. `SPEC` achieves an average 1.9× (up to 3×) speedup.

8.1 Methodology

We generate algorithm-specific accelerators using HLS targeting an Intel Arria 10 FPGA. The C codes are taken directly from benchmark suites without adding any HLS-specific annotations (excluding dynamic structures, like queues, that were replaced with HLS-specific libraries).

We use the LLVM-based Intel SYCL HLS compiler [29] and apply our standard DAE transformation (§3.2) and our proposed speculation transformation (§5) 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, 24, 31, 53].

We report cycle counts from ModelSim simulations. We do not report circuit frequency since our approach does not affect the critical path (see [52] for such details). Area usage is obtained after place and route using Quartus 19.2.

8.1.1 Baselines. For each benchmark, we synthesize the following architectures which represent current state-of-the-art approaches to HLS :

- `STA`: the default, industry-grade approach using static scheduling [29]. Loads that cannot be disambiguated at compile time execute in order.
- `DAE`: a DAE architecture without speculation. OoO loads are enabled by an LSQ. This is the state-of-the-art approach to irregular codes in academia [53], but it 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.

8.1.2 Benchmarks. DAE architectures optimize the latency between memory and compute and are most beneficial for memory-bound codes [24], especially codes with an irregular memory access pattern that prevents static prefetching [18]. We evaluate nine such benchmarks from the graph and data analytics domain, using the GAP graph benchmark

Table 1. Absolute performance and area usage of STA [29], DAE [53], SPEC, and ORACLE accelerators. (*bc uses two LSQs).

Kernel	Poison		Mis-spec.	Cycles				Area (ALMs [28])			
	Blocks	Calls	Rate	STA	DAE	SPEC	ORACLE	STA	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
			Harmonic Mean:	1	3.2	0.51	0.48	1	1.16	1.42	1.36

suite [8] and an HLS benchmark suite [14] 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 (20×20 matrix).

For the graph codes (bfs, bc, sssp) we use a real-world graph email-Eu-core with 1005 nodes and 25,571 edges.

8.2 Performance

Figure 6 reports normalized speedups of each technique over STA. Our SPEC approach gives on average a 1.9× (and up to 3×) speedup over STA. This is within 5% of the ORACLE performance. In contrast, DAE without speculation sees a dramatic performance degradation over STA, because the AGU, DU, CU communication is sequentialized.

Mis-speculation Cost. The SPEC and ORACLE performance gap 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 to guarantee perfect pipelining [34]. 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. SPEC cycle counts as 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

8.3 Code Size

Our speculation approach can increase the number of blocks in the CU, especially for codes with deeply nested control flow. In HLS, an increased number of blocks can result in a higher area usage due to larger scheduler complexity [50].

Table 1 shows the absolute area usage of all accelerators. 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, and sometimes it is possible to reduce the number of blocks using our merging technique (e.g., two poison blocks in mm merged into one).

Impact of Nested Control Flow on Area Usage. 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:

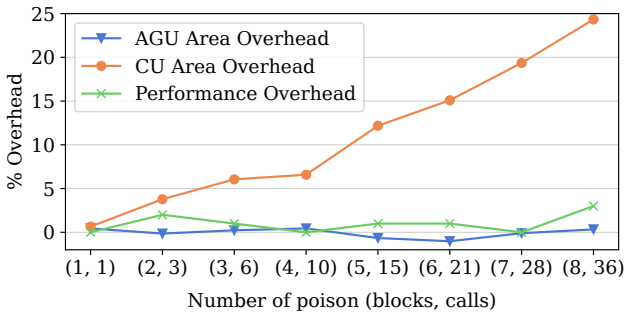


Figure 7. Change in area and performance overhead of SPEC over ORACLE as the number of poison blocks and calls grows.

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 7 shows how the area and performance overhead of SPEC over ORACLE changes as more poison blocks are needed. 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 CU 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.

9 Related Work

Program slicing is used beyond DAE architectures. Decoupled Software Pipelining (DSWP) [41] is a popular transformation that decouples strongly connected components in the program dependence graph into separate pipeline stages mapped over multiple PEs communicating via FIFOs. The PEs can be CPU threads, or pipeline stages in an accelerator generated by HLS [33]. Control dependent pipeline stages in DSWP can also be executed speculatively, although stages with memory operations require versioned memory [58].

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, 42, 50]. Most recently, predicated execution in the form of gated SSA was proposed for HLS with speculation support [23]. The speculation support in this and other works requires costly recovery on mis-speculation [6, 22, 30, 35, 56, 60]. Efficiently squashing speculative computation on the wrong paths in a spatial dataflow architecture is hard, because the architectural state is distributed [10]. Our speculative DAE sidesteps this issue, not requiring 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.

Control-flow handling in GPUs is usually implemented via *predication*. The algorithms used to calculate predicate masks and re-convergence points resemble our work [32]. The SIMT stack approach in GPUs pushes predicate masks onto a stack when entering a control-flow nesting level, and pops 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 inserting of poison calls in Algorithm 1 can be seen as a pass through the CFG with only pop operations where the placement of the pops follows a certain policy just like modern SIMT compilers follow different policies to prevent SIMT deadlock and livelock, or to improve performance [19], instead of popping at the immediate post-dominator.

10 Conclusion

We have presented general compiler support for speculative memory operations in DAE architectures that tackles the LoD problem resulting from control dependencies. We have proposed CFG transformations implementing speculation in the address generation slice, and poisoning of mis-speculations in the compute slice, with a proof of correctness.

We have presented three applications where our work improves support for the efficient execution of irregular codes: DAE-based CPU/GPU prefetchers that require compiler support, CGRA architectures, and HLS-generated specialized accelerators. We have evaluated our work on HLS-generated accelerators, showing an average 1.9× (up to 3×) speedup over non-DAE accelerators on a set of irregular benchmarks where DAE is not possible without our speculation. Our approach has no mis-speculation cost and a small code size footprint, scaling well to deeply nested control flow.

Future work could investigate vector-parallelism support by filling a vector of speculative requests in the AGU and producing a store mask in the CU, similar to the recent work on decoupled vector runahead prefetching in CPUs [36].

Data-Availability Statement

We make our work and evaluation publicly available [52].

Acknowledgments

We thank Intel for access to FPGAs through their DevCloud. This work was supported by a UK EPSRC PhD scholarship.

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. doi: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. doi: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] ARM. 2024. AXI protocol overview. https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview

[5] 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. doi:10.1145/2366231.2337169

[6] 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. doi:10.1145/279361.279391

[7] Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer. 2015. Perfect Reconstructability of Control Flow from Demand Dependence Graphs. *ACM Trans. Archit. Code Optim.* 11, 4, Article 66 (jan 2015), 25 pages. doi:10.1145/2693261

[8] 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

[9] 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. doi:10.1145/165939.165952

[10] 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)*. doi:10.1109/ISPASS.2005.1430572

[11] 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. doi:10.1109/MICRO.2016.7783749

[12] 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. doi:10.1109/MICRO.2016.7783749

[13] 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. doi:10.1145/3431920.3439290

[14] Jianyi Cheng. 2019. *HLS_Benchmarks*. doi:10.5281/zenodo.3561115

[15] 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. doi:10.1109/FPT.2014.7082758

[16] 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/

[17] Neal C. Crago, Sana Damani, Karthikeyan Sankaralingam, and Stephen W. Keckler. 2024. WASP: Exploiting GPU Pipeline Parallelism with Hardware-Accelerated Automatic Warp Specialization. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1–16. doi:10.1109/HPCA57654.2024.00086

[18] 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. doi:10.1145/2024723.2000079

[19] Ahmed ElTantawy and Tor M. Aamodt. 2016. MIMD synchronization on SIMT architectures. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. doi:10.1109/MICRO.2016.7783714

[20] Zhihua Fan, Wenming Li, Shengzhong Tang, Xuejun An, Xiaochun Ye, and Dongrui Fan. 2023. Improving utilization of dataflow architectures through software and hardware co-design. In *European Conference on Parallel Processing*. Springer, 245–259. doi:10.1007/978-3-031-39698-4_17

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

[22] 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. doi:10.1007/978-3-540-78610-8_19

[23] 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. doi:10.1145/3640537.3641581

[24] 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. doi:10.1145/2830772.2830800

[25] 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. doi:10.1145/3075620

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

[27] Tu Hong, Ning Guan, Chen Yin, Qin Wang, Jianfei Jiang, Jing Jin, Guanghui He, and Naifeng Jing. 2020. Decoupling the multi-rate dataflow execution in coarse-grained reconfigurable array. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. doi:10.1109/ISCAS45731.2020.9181275

[28] 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. doi:10.1007/978-3-540-30117-2_16

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

[30] 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. doi:10.1145/3289602.3293914

[31] 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). doi:10.1109/TCAD.2021.3105574

[32] 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. doi:10.1145/800031.808581

[33] 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. doi:10.1145/2593069.2593105

[34] 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. doi:10.1109/ICFPT56656.2022.9974425

[35] 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. doi:10.1145/143371.143529

[36] 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. doi:10.1145/3613424.3614255

[37] 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. doi:10.1145/3466752.3480048

[38] 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. doi:10.1109/HPCA56546.2023.10071026

[39] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 416–429. doi:10.1145/3079856.3080255

[40] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. 1990. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI '90*. doi:10.1145/93548.93578

[41] 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. doi:10.1109/MICRO.2005.13

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

[43] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. 2019. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 137–151. doi:10.1145/3297858.3304025

[44] W. W. Peterson, T. Kasami, and N. Tokura. 1973. On the capabilities of while, repeat, and exit statements. *Commun. ACM* 16, 8 (Aug. 1973), 503–512. doi:10.1145/355609.362337

[45] Raghu Prabhakar and Sumti Jairath. 2021. SambaNova SN10 RDU:Accelerating Software 2.0 with Dataflow. In *2021 IEEE Hot Chips*. 1–37. doi:10.1109/HCS52781.2021.9567250

[46] 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. doi:10.1145/3079856.3080256

[47] Shantian Qin, Wenming Li, Zhihua Fan, Zhen Wang, Tianyu Liu, Haibin Wu, Kunming Zhang, Xuejun An, Xiaochun Ye, and Dongrui Fan. 2023. ROMA: A Reconfigurable On-chip Memory Architecture for Multi-core Accelerators. In *2023 IEEE International Conference on High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application*. IEEE, 49–57. doi:10.1109/HPCC-DSS-SmartCity-DependSys60770.2023.00017

[48] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, et al. 2021. Warehouse-scale video acceleration: co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 600–615. doi:10.1145/3445814.3446723

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

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

[51] 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. doi:10.1145/1067649.801719

[52] Robert Szafarczyk. 2025. *Compiler Support for Speculation in Decoupled Access/Execute Architectures - code & evaluation*. doi:10.5281/zenodo.14678644

[53] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2023. Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. 1–9. doi:10.1109/FPL60245.2023.00009

[54] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2023. A High-Frequency Load-Store Queue with Speculative Allocations for High-Level Synthesis. In *2023 International Conference on Field Programmable Technology (ICFPT)*. 115–124. doi:10.1109/ICFPT59805.2023.00018

[55] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2025. Dynamic Loop Fusion in High-Level Synthesis. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '25)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3706628.3708871

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

[57] 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. doi:10.1145/224170.224301

[58] 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. doi:10.1109/PACT.2007.4336199

[59] Zhengrong Wang and Tony Nowatzki. 2019. Stream-based Memory Access Specialization for General Purpose Processors. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 736–749. doi:10.1145/3307650.3322229

[60] 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. doi:10.1109/ISCA45697.2020.00032

[61] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. 2019. CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 50 (Oct. 2019), 26 pages. doi:10.1145/3358177

[62] 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. doi:10.1109/FCCM.2016.26

Received 2024-11-12; accepted 2024-12-21