Dynamic Loop Fusion in High-Level Synthesis

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

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

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

Abstract

Dynamic High-Level Synthesis (HLS) uses additional hardware to perform memory disambiguation at runtime, increasing loop throughput in irregular codes compared to static HLS. However, most irregular codes consist of multiple sibling loops, which currently have to be executed sequentially by all HLS tools. Static HLS performs loop fusion only on regular codes, while dynamic HLS relies on loops with dependencies to run to completion before the next loop starts.

We present dynamic loop fusion for HLS, a compiler/hardware co-design approach that enables multiple loops to run in parallel, even if they contain unpredictable memory dependencies. Our only requirement is that memory addresses are monotonically non-decreasing in inner loops. We present a novel program-order schedule for HLS, inspired by polyhedral compilers, that together with our address monotonicity analysis enables dynamic memory disambiguation that does not require searching of address histories and sequential loop execution. Our evaluation shows an average speedup of 14× over static and 4× over dynamic HLS.

CCS Concepts

• **Computer systems organization → Reconfigurable computing**; • **Software and its engineering → Compilers**.

Keywords

high-level synthesis; loop fusion; decoupled access/execute

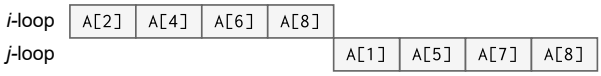
ACM Reference Format:
Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2025. Dynamic Loop Fusion in High-Level Synthesis. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25), February 27–March 1, 2025, Monterey, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3706628.3708871

1 Introduction

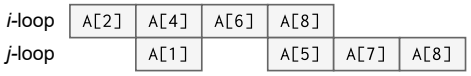
High-Level Synthesis (HLS) increases designer productivity, makes code more maintainable, accelerates verification, and makes design space exploration easier [51]. However, this is usually only true for regular codes where the compiler can discover instruction- and memory-level parallelism statically [11, 53]. Domains like graph analytics and sparse linear algebra contain irregular codes with unpredictable memory dependencies and control flow, which break the traditional static scheduling approach. This prompted research into dynamically scheduled HLS [40] and approaches to combine it with existing industry-grade static HLS compilers [15, 60].

```
for (i = 0; i < N; ++i)
    A[f(i)] = workA(A[f(i)]);
for (j = 0; j < M; ++j)
    B[g(j)] = workB(A[g(j)]);
```

(a) Two loops with non-affine access patterns.



(b) Pipeline achieved by current static and dynamic HLS tools.



(c) Pipeline achieved by our work.

Figure 1: Dynamic Loop Fusion enables fine-grained parallelism *across* loops with memory dependencies.

Dynamic HLS uses load-store queues (LSQs) to perform memory disambiguation at runtime [20, 21, 29, 34, 38, 61]. These works effectively pipeline single loops with arbitrary memory dependencies, but they have to sequentialize multiple loops if they share a memory dependency. For example, they would sequentialize the *i*- and *j*-loops in figure 1a, resulting in the figure 1b pipeline. But, as shown in figure 1c, there might be plenty of parallelism *across* the two loops.

There are two reasons why current dynamic HLS tools have to sequentialize these loops. Firstly, they use a program-order schedule that relies on loops to run to completion before the next loop starts. For example, the LSQ used in Dynamatic HLS sequentializes LSQ requests based on the program order of basic blocks [38]; other approaches carry explicit dependencies through the pipeline, preventing downstream loops from starting without resolving the dependency [25, 61]. Secondly, they rely on the checking of address histories to detect hazards, without making any assumptions about the underlying address distributions. This makes them general, but requires them to wait for all addresses from one loop to be produced before they can start processing the next loop. These are two *key challenges* that we tackle in this paper.

Static loop fusion also fails to fuse the loops in our figure 1a example, because the fused loop may introduce a negative dependency distance [41]—the compiler gives up if it cannot prove that $f(i) = g(j) \implies i < j$. This is assuming that the $f(i)$ and $g(i)$ functions can be analyzed by the compiler in the first place. If that is not the case, e.g., if they involve an array access, then loop fusion is also not applied.

FPGA '25, February 27–March 1, 2025, Monterey, CA, USA

ACM ISBN 979-8-4007-1396-5/25/02
https://doi.org/10.1145/3706628.3708871

Our dynamic loop fusion approach can automatically synthesize a Read After Write (RAW) check that will protect the $A[g(j)]$ read in the figure 1 code, achieving the fine-grained inter-loop parallelism from figure 1c. We decouple each loop into an independently scheduled Processing Element (PE). Memory dependencies across loops are handled in a Data Unit (DU) specialized by our compiler for the program. Our only requirement is that the `f(i)` and `g(j)` functions are monotonically non-decreasing in the innermost loop (outer loops can be non-monotonic). This is a weaker requirement than the affine functions expected by static loop fusion, allowing us to fuse more loops, including codes with data-dependent addresses.

To the best of our knowledge, we are the first to propose dynamic memory disambiguation that can work *across* loops. We make the following contributions:

- A compiler pass to decouple loop nests into PEs. A PE is further decoupled into an Address Generation Unit (AGU) and a Compute Unit (DU) following the decoupled access/execute (DAE) architecture (section 2.1).
- A compiler analysis, based on the chain of recurrences theory [5, 64], that checks if addresses are monotonically non-decreasing in inner loops, and that detects non-monotonic outer loops (section 3).
- A hardware-efficient program-order schedule representation that does not require sequentializing loops. We show how the compiler instruments AGUs with instructions that generate the schedule for each memory operation. We also show how non-monotonic outer loops can be integrated with our schedule (section 4).
- A parameterizable DU performing dynamic memory disambiguation across loops. We show how the compiler can specialize the DU given the dependency graph of the program and the address monotonicity analysis. We discuss how the DU optimizes DRAM bandwidth by using dynamic coalescing and on-chip store-to-load forwarding (section 5).
- An evaluation on irregular applications showing an average speedup of 14× over static HLS and 4× over dynamic HLS. We discuss which codes benefit from dynamic fusion and we study the impact of store-to-load forwarding (section 7).

2 Background

In this paper, we focus on codes using DRAM, as its unpredictable latency and limited bandwidth pose greater challenges than BRAM. There is no fundamental reason why we could not protect BRAM or use a memory hierarchy with BRAM caches, which we briefly discuss in section 8.

In this section, we describe FPGA streaming architectures commonly used with DRAM. We discuss techniques to optimize DRAM bandwidth in irregular codes that inform the design of our DU. And we describe existing loop fusion approaches and their compiler theory, informing the design of our program-order schedule representation.

2.1 Baseline Streaming Architecture

Streaming FPGA architectures are a popular choice for implementing DRAM-based codes [19, 22, 55, 63, 65]. They decouple memory accesses and compute into separate PEs, either automatically

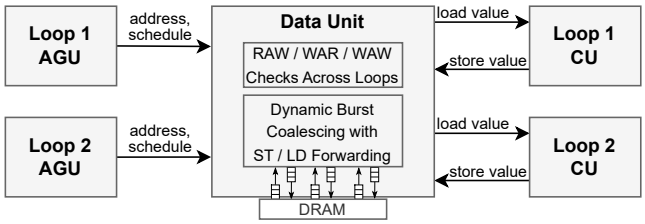


Figure 2: An example DAE streaming FPGA architecture.

[22, 65] or manually [19, 55, 63]. The use of a streaming architecture is predicated on an accurate memory dependency analysis so that memory shared between PEs can be transformed into FIFO communication. If the analysis fails, as it invariably does for irregular codes, then the shared data has to be communicated via DRAM and the execution of PEs has to be sequentialized, thus losing much of the benefits of using a streaming FPGA architecture.

To tackle the problem of irregular memory accesses, we propose to use a compiler-parameterized DU, shown in figure 2, that protects memory shared across loops by performing dynamic memory disambiguation at runtime. The DU interfaces with DRAM, but is also able to directly forward values from producer to consumer PEs if the respective load/store operations exhibit temporal locality, thus saving DRAM bandwidth as in traditional streaming FPGA architectures.

2.1.1 Using DRAM Bandwidth Efficiently. We use Altera's DRAM IP generated by its HLS compiler to implement DRAM load/store units (LSUs). Our DU can have multiple LSUs connected to the DRAM controller using a ring topology, depending on the number of load/store operations in the input program. To use DRAM bandwidth efficiently, the LSUs coalesce multiple loads/stores into one wide request to the memory controller in order to use the full DDR channel width (512-bit in our case). To achieve this for codes with irregular access patterns, the LSUs use additional logic and buffering to perform coalescing dynamically [4, 69]. DRAM requests are buffered until the largest possible burst can be made. If no new requests arrive in N consecutive cycles, then an incomplete burst is made (in our case $N = 16$).

Asynchronous address supply is essential for efficient use of DRAM, because of the high access latency, and to allow the dynamically bursting LSU to look ahead in the address stream. Streaming FPGA architectures achieve this by following the decades-old DAE principle [57], where the address generation is decoupled into its own thread of execution, running ahead of the compute threads that consume and produce values [12, 17, 27, 47]. Note that dynamic loop fusion is not limited to DAE architectures; it can be realized in other model of computations, e.g., with dynamic dataflow [40].

2.1.2 DAE Transformation. A DAE architecture is automatically generated by our compiler. Given a forest of loop trees, loop CUs and AGUs are decoupled into their own PEs following the strategy from figure 3. AGUs feed addresses to the DU; the DU sends load values to and receives store values from CUs. All communication is FIFO based, following a latency-insensitive protocol [26]. To analyze which values should be computed by which decoupled unit

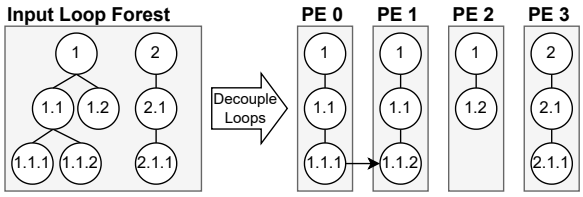


Figure 3: Example decoupling of a loop forest. A leaf loop is decoupled into its own PE, which includes loop control of the outer loops. Parent loop body instructions are included only if they come before the leaf loop in the topological order. FIFOs are used to communicate scalar data dependencies (e.g. from loop 1.1.1 in PE 0 to loop 1.1.2 in PE 1). FIFOs are written in the loop exit block and read in the loop pre-header block. Each loop PE might have an AGU producing memory requests fo the accesses in that loop PE.

and which values should be communicated, we use the def-use chain encoded in the SSA form of the code (each SSA value usage can be traced to its unique definition [52]). We follow a standard approach to automatically generate a DAE architecture [60]:

(1) **AGU:** Each memory operation to be decoupled is changed to a `send_address` FIFO write that sends the memory address to the DU.
(2) **CU:** Dually, in the CU each memory operation to be decoupled is changed to a `consume_value` or `produce_value` FIFO read function that receive or send values to or from the DU.
(3) **Dead code elimination (DCE):** We apply DCE in the CU to remove any unnecessary address generation code. In the AGU, we delete side effect instructions that are not part of the address generation def-use chains, and then also apply DCE followed by control-flow simplification to remove redundant basic blocks.

2.2 Static Loop Fusion

Polyhedral compilers represent memory operations inside loop nests as integer sets [10, 28, 30]:

(1) The *domain* set describes the set of loop iterations in which a statement is executed.
(2) The *schedule* set maps domain elements to a point in time. Given two schedule instances, we can determine which one comes first in program order.
(3) The *access* set maps domain elements to a point in space, representing the accessed memory location.

For example, the domain (D), schedule (S), and access (A) functions of the i-loop store st_A and j-loop load ld_A in figure 1a are:

$$D_{st_A} = \{st_A[i] : 0 \le i < N\}, \quad D_{ld_A} = \{ld_A[j] : 0 \le i < M\}$$
$$S_{st_A} = \{st_A[i] \to [0, i]\}, \quad S_{ld_A} = \{ld_A[j] \to [1, j]\}$$
$$A_{st_A} = \{st_A[i] \to f(i)\}, \quad A_{ld_A} = \{ld_A[j] \to g(j)\}$$

The set intersection of two access relations can be used to find dependencies between the two corresponding operations.

Static loop fusion for the code in figure 1a can be expressed as a transformation on the schedule of the load: $\mathcal{T}_{Fusion} = \{[1, j] \to [0, i]\}$ (together with transformations to account for $N \neq M$). $\mathcal{T}_{Fusion}$ might introduce a new dependency between the store and load. The transformation is only legal if the dependency distance of the new dependency is non-negative: this implication has to hold $A_{st_A}[k] = A_{ld_A}[l] \implies k < l$, where k, l are some iterations in the fused loop. In other words, if in the original program a given store writes to an address that a given load later uses, then in the fused loop the store must execute in an earlier iteration than the load.

The legality of loop fusion can be reduced to checking the legality of pairwise loop permutation [50]—the permutation should not break dependencies. However, if the address expressions do not form affine functions, then the legality check does not have enough information about dependency distances to be useful. One can over-approximate non-affine functions as affine [6], but this does not help in all cases, e.g., over-approximation can introduce spurious dependencies on codes with data-dependent addresses. Our dynamic loop fusion is more lenient, requiring only monotonically non-decreasing addresses. However, we stress that our aim is not to replace the polyhedral approach to static loop fusion. Clearly, static loop fusion is preferable whenever possible, especially since it can be combined with other transformations in one framework [50]. Rather, we aim to enable fusion in cases where static approaches are fundamentally infeasible.

3 Address Monotonicity

We now describe the concept of address monotonicity in more detail and contrast it with affine addresses.

3.1 Motivation for Monotonicity

Assume that we have a memory dependency across loops. If we can prove at compile time that the address of the dependency source is monotonically non-decreasing, then at runtime the loop with the dependency destination only has to check if the address it accesses is lower than the most recently accessed address in the source loop—the dependency destination does not need to see the full history of memory accesses made in the other loop. This paves the way for our efficient hardware dynamic memory disambiguation across loops described in section 5. We now describe how addresses can be proven to be monotonically non-decreasing.

3.2 Monotonic Chain of Recurrences

Compilers can represent expressions inside loops as a *Chain of Recurrences* (CR) [5, 49, 64]:

$$\{base, \odot, step\},$$

where $base$ and $step$ can themselves be a CR, and $\odot \in \{+, \times, \div\}$. To reason about memory addresses, we typically use the constraints: $base, step \in \mathbb{N}$ if they are not a CR; and $\odot = \{+, \times\}$. Both LLVM and GCC provide a CR analysis called Scalar Evolution (SCEV) [7, 48].

A CR is *affine* iff it is an add recurrence and iff its step is a constant expression not containing any CRs [30]. A CR is *monotonically non-decreasing* iff its step is non-negative [71]. For brevity, we use the term *monotonic* to mean monotonically non-decreasing in the rest of the paper.

Monotonic CRs are more general than affine CRs and handle control flow better [71]. For example, the CR of a row-major $N \times N$

matrix traversal is affine and monotonic: $\{\{0, +, N\}, +, 1\}$. But the CR for an FFT traversal is not affine anymore, only monotonic: $\{\{0, +, 1\}, +, \{2, \times, 2\}\}$.

An address expression is monotonic w.r.t. a given loop depth iff the loop CR expression consists of only monotonic CRs. Monotonically non-increasing addresses (i.e., using $step \in \mathbb{Z}$ and adding $\div$ to $\odot$) can also be supported by just flipping signs in the hazard detection logic, but we do not discuss this further in this paper.

3.3 Monotonicity in Sparse Array Formats

Data-dependent accesses cannot be analyzed using the CR formalism, yet their underlying access pattern is often monotonic. For example, sparse matrix formats, like CSR, produce address sequences that retain the partial order of the original row-major matrix traversal. Other data-dependent accesses that are not monotonic by definition can be made monotonic with pre-sorting. To support dynamic loop fusion on these codes, we allow the user to annotate memory operations asserting that the address is monotonic in a given loop.

3.4 Non-Monotonic Outer Loops

We require a monotonic CR for the innermost loop of the memory dependency source; the outer loop CRs can be non-monotonic. Consider this producer-consumer example:

```
for (i=0; i<ITERS; ++i)
    for (j=0; j<N; ++j)
        store A[j];
for (k=0; k<M; ++k)
    load A[k];
```

The store innermost j-loop is monotonic, but the outer i-loop is not—advancing the i-loop causes the store address to reset. We encode this information in our schedule (section 4), so that in this case our DU will know that it has to wait for the last i-loop iteration to be sure that a given $A[j]$ store address in the j-loop will not be repeated.

3.4.1 Detecting Non-Monotonicity. Given an address expression $f(i_1, i_2, ..., i_n)$ nested within n loops (where n is the innermost loop depth), a $k, 1 \leq k < n$ loop depth is *non-monotonic* if there exists a $j > k$ loop depth such that $CR_k.step < (CR_j.step \times tripCount_j)$, where $CR_k.step$ is the step component for loop k, and $tripCount_j$ is the number of times loop j executes. In other words, a given outer loop k is non-monotonic if there exists a deeper nested loop whose entire execution contributes a larger value to the address value than one k-loop iteration. A CR_k for loop k might not exist, in which case that loop depth is trivially marked as non-monotonic.

For example, the outer loop in a row-major $N \times M, N > 1, M > 1$ matrix traversal is monotonic, because its step is M, which is not lower than $CR.step \times tripCount = M$ of the inner loop. On the other hand, the outer loop in a column-major traversal is non-monotonic, because its step value is 1, which is lower than $CR.step \times tripCount = M \times M$ of the inner loop.

The above expressions are usually symbolic. We substitute symbols with their maximum values (after a value range analysis). This makes our monotonicity checks conservative—we might get false positives, but never false negatives. The checks could be performed at runtime instead, which would make the result precise. However, false positives did not occur in our evaluation, so we leave this for future work.

4 Program-Order Schedule for Hardware

Our schedule representation allows multiple loops to run in parallel, as opposed to being sequentialized as in existing dynamic memory disambiguation approaches for HLS [21, 29, 34, 38, 61]. Section 2.2 discussed the schedule representation used in polyhedral compilers. We use a similar representation at runtime, but with the following optimizations for hardware:

(1) Each loop depth is represented by one element in the schedule tuple, instead of a multi-dimensional point.
(2) Each schedule element is incremented by 1 for each invocation of the loop body corresponding to that element—no dependencies between schedule elements are introduced across loops. Repeated invocations of inner loops do not cause the corresponding schedule elements to wrap around.
(3) Schedule comparisons between two operations involve just one comparison between the schedule elements corresponding to the innermost shared loop depth of the operations, as opposed to comparing whole tuples as is the case in the polyhedral schedules.

Consider these two nested loops for example:

```
for (i=0; i<N; ++i)
    for (j=0; j<2; ++j)
        ld_0; st;
    for (k=0; k<4; ++k)
        ld_1;
```

Our DAE pass will decouple this code into two loop PEs:

```
for (i=0; i<N; ++i)
    for (j=0; j<2; ++j)
        ld_0; st;
```

```
for (i=0; i<N; ++i)
    for (k=0; k<4; ++k)
        ld_1;
```

Assume $i = 1, j = 0$ for the left PE; and $i = 0, k = 3$ for the right PE. The st schedule will be $\{2, 3\}$; the ld_1 schedule will be $\{1, 4\}$. To check if a st schedule instance comes before a ld_1 schedule instance in program order, written as $schedule_{st} \prec schedule_{ld_1}$, we compare the schedule elements corresponding to the i-loop. Similarly, to check $schedule_{st} \prec schedule_{ld_0}$, we compare the j-loop schedule elements.

The below table shows the difference in evolution of our and the polyhedral schedule representation for the st operation:

iters:	i=0, j=0	i=0, j=1	i=1, j=0	i=1, j=1
poly:	{0, 0, 0, 1}	{0, 0, 1, 1}	{1, 0, 0, 1}	{1, 0, 1, 1}
ours:	{1, 1}	{1, 2}	{2, 3}	{2, 4}

The additional dimensions in the polyhedral schedule are used to represent program order within loops. How can we avoid the additional dimensions in our schedule and still recover program order within loops? For example, we want to know that $schedule_{ld_0} \prec schedule_{st}$ even when both schedules will be equal to $\{2, 3\}$. Our insight is to *configure the schedule comparator* based on the topological order of memory operations in the program. In a $schedule_{ld_0}[1] \odot$

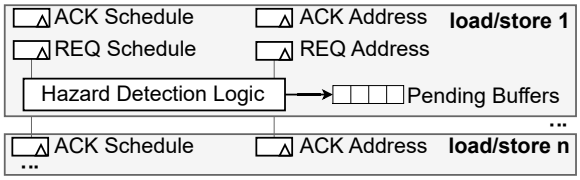


Figure 4: Data Unit (DU) consisting of of n Load Store Units.

$schedule_{st}[1]$ comparison, where the index 1 refers to the i-loop, we will configure $\odot = \leq$. Dually, to check $schedule_{st} \prec schedule_{ld_0}$, we would synthesize: $schedule_{st}[1] < schedule_{ld_0}[1]$.

In summary, our compiler pass statically configures schedule comparators used in the DU for each dependency pair, so that we can recover total ordering without additional schedule dimensions and without the need to compare entire schedule tuples.

4.1 Integration of Non-Monotonic Outer Loops

For each non-monotonic outer loop k, we add a *lastIter* bit to the schedule that will be set in the AGU if the corresponding request was generated on the last k-loop iteration. Our DU uses *lastIter* bits as hints to expedite disambiguation—they are not essential for correctness. Non-monotonic loops for which *lastIter* bits cannot be generated are still supported.

4.2 Schedule Generation in AGUs

Our compiler adds schedule-generating instructions for each AGU memory request as follows:

(1) At the start of the AGU, an n-tuple *schedule* is initialized to 0, where n is the request loop depth.
(2) At each loop depth $1 \leq i \leq n$, a $schedule[i]$ increment instruction is inserted to the beginning of the first non-exiting basic block of the i-loop body.
(3) For each non-monotonic loop k, we add a $lastIter[k]$ comparison instruction that evaluates to true if this is the last k-loop iteration. This involves calculating loop predicates one iteration in advance. The *lastIter* bit is just a hint and is set to false if the loop predicate cannot be calculated one iteration in advance.
(4) At the end of the AGU, each *schedule* element is set to a sentinel value that signals to the DU that there will be no more requests from this AGU.

Schedules are implemented in 32-bit registers and are shared between all memory operations in the same AGU. Future work could use range analysis to decrease schedule bit sizes.

5 Data Unit with Hazard Detection

Each program base pointer that has unpredictable dependencies, or that has dependencies across loops that cannot be fused statically, is assigned its own DU to perform dynamic disambiguation. Figure 4 shows a high-level DU organization. In our implementation, each program load and store gets its own port; future work could study port sharing.

Each load and store keeps track of the address and schedule corresponding to the most recent ACK received from, and the next request to be sent to, the memory controller. It also has buffers to hold addresses, schedules, and values (in case of stores) for pending requests (not yet ACKed requests) .

The hazard detection logic compares the address and schedule of its next request with the address and schedule of the most recent ACK of its dependency sources. The next request will only be sent to the memory controller and moved to the pending buffer if the check succeeds. The check and enqueueing logic is spread across multiple pipeline stages—there is no negative load latency impact, because, thanks to the DAE architecture, load addresses run ahead of load consumers giving us ample cycle budget. The pending buffers are implemented in registers to enable associative searching needed for store-to-load forwarding (section 5.5)—their size depends on the DRAM burst size.

In the rest of this section, we describe how the monotonicity property and our schedule representation are used to enable dynamic memory disambiguation across loops.

5.1 Hazard Detection Problem Statement

We are trying to check if a memory operation a has a data hazard with memory operation b. Assume a is nested in n loops, b is nested in m loops, and they both share a loop at depth $k, k \leq n, k \leq m$. Informally, given a $req.schedule_a$ and $req.address_a$ corresponding to the next a request, and $ack.schedule_b$ and $ack.address_b$ corresponding to the most recent ACK for operation b, our hazard detection logic deems the next a request safe if either of the two conditions holds:

(1) The next a request comes before the most recent b ACK in program order.
(2) The next a request comes after the most recent b ACK in program order, but $req.address_a$ will no be accessed by operation b in the the $(ack.sdchedule_b, req.schedule_a)$ range.

We now describe each of these points in more detail, before composing the equations implementing these two checks into a general Hazard Safety Check. In the following discussion, we use the term "$(schedule_a, schedule_b)$ time range" to mean the sequence of memory requests b' such that $schedule_a[k] < schedule_{b'}[k] < schedule_b[k]$, where k is the innermost common loop depth of operation a and b. We use open parenthesis and box brackets to represent open and closed intervals, respectively.

5.2 Comparing Schedules

If operations a and b do not share any loops ($k = 0$), then the relative schedule program order will always match their topological program order and we do not need to synthesize any comparisons. Otherwise, if the shared loop depth $k > 0$, we synthesize the following comparison to check if the next a request comes before the most recent b ACK:

(Program Order Safety Check)

$$req.schedule_a[k] \odot ack.schedule_b[k] \;\|\; \left(req.schedule_a[k] \odot req.schedule_b[k] \;\&\; noPendingAck_b\right)$$

Where $\odot = \leq$ if $a \prec b$ in topological program order, else $\odot = <$. The *noPendingAck* term is a single bit that is set if b is not waiting for any ACKs. The second equation line makes sure that the a request is

deemed safe if there are no further b requests in the $[ack.schedule_b, req.schedule_a)$ time range.

Since we only use the schedule element corresponding to the innermost shared loop of the two memory operations, we do not need to synthesize the rest of the schedule.

5.3 Checking Address Reset in Schedule Range

If the above check fails, then for request a to be safe we check that operation b will not access $req.address_a$ in the $(ack.schedule_b, req.schedule_a)$ time range. If all operation b loop depths are monotonic, this is a simple $req.address_a < ack.address_b$ check. If some b loops are non-monotonic, we need to guarantee that $ack.address_b$ will not be reset in the considered schedule range:

(No Address Reset Check)

$$lastIterCheck \ \& \ req.schedule_a[l] = ack.schedule_b[l] + \delta$$

Here, $\delta = 1$ if $a \prec b$, else $\delta = 0$; l is the deepest non-monotonic loop depth in the b operation loop nest such that $l \leq k$; and the $lastIterCheck$ term is an AND-reduction of the b $lastIter$ bits:

$$ack.lastIter_b = (bit_1, ..., bit_k, \underbrace{bit_{k+1}, ..., bit_{m-1}}_{\text{AND-reduction}}, bit_m),$$

where $bit_j, 1 \leq j \leq m$ is set to 1 at compile time if the j loop is monotonic and thus optimized away from the reduction; otherwise bit_j will be set dynamically on the last iteration of the j loop according to the procedure from section 4.1.

The first term in the No Address Reset Check guarantees that all non-monotonic child loops of k are on their last iteration, and thus will not reset the b address. The second term guarantees that the b address will not reset as a result of advancing in some parent loop of k. Only bits corresponding to non-monotonic loop depths are considered in the AND-reduction. Similarly, if all $[1, k]$ loops are monotonic, then the second term is omitted.

5.3.1 Example. Consider the following code:

```
for (; a < A; ++a)       // depth 1: non-monotonic
  for (; b < B; ++b)     // depth 2: monotonic
    for (; c < C; ++c)   // depth 3: non-monotonic
      for (; e < E; ++e) // depth 4: monotonic
        mem_op_b;
    for (; d < D; ++d)   // depth 3
      mem_op_a;
```

Here, the b address is non-monotonic at loop depth 1 and 3. The innermost common loop depth of operations a and b is $k = 2$. The innermost non-monotonic b loop depth that is lower than k is $l = 1$. Thus, the No Address Reset Check checks $req.schedule_a[1] = ack.schedule_b[1]$ to guarantee that b will not have any more l-loop iterations until reaching the $req.schedule_a$ point. And it will check if $ack.lastIter_b[3]$ is set to guarantee that the b address will not reset by advancing in the non-monotonic $3 > k$ loop.

5.4 Hazard Safety Check

With the ability to compare program order schedules and guaranteeing that addresses do not reset in a given schedule range, we can now construct a general data hazard check. The next a request is safe to execute w.r.t the most recent b ACK if:

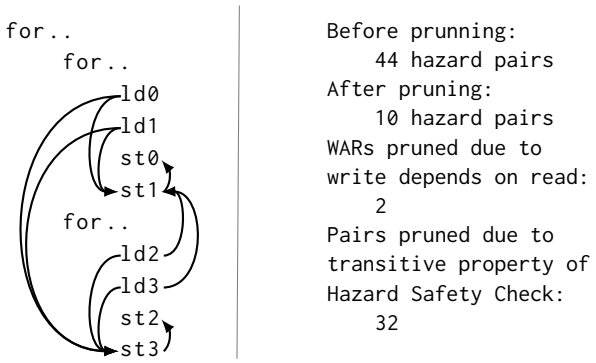


Figure 5: Result of pruning hazard pairs in the later evaluated FFT code. Each memory operation checks for safety against at most one operation per loop depth (e.g., ld_0 checks against st_3 in its first loop depth, and against st_1 in the second).

(Hazard Safety Check)

$$ProgramOrderSafetyCheck \ \| \ (req.address_a < ack.address_b \ \& \ NoAddressResetCheck)$$

5.4.1 Complexity. The Hazard Safety Check simplifies to just one $req.address_a < ack.address_b$ comparison if a and b do not share loops. If b has non-monotonic loops, then the No Address Reset Check adds at most one AND reduction and one equality check. The number of comparisons grows to three if there is a shared loop thanks to the Program Order Safety Check. In general, given a program with n operations, if we check every possible dependency pair, then the number of comparisons is $O(n^2)$—reducing complexity becomes important as the number of loads and stores grows. Loads do not have to check for hazards against other loads. Also, WAR checks where the written value depends on the read value can be omitted, as previous work has already pointed out [39].

However, by exploiting the transitive property of our Hazard Safety Check we can prune many more hazard pairs. Assume that we have three memory operations with the following topological program order $c \prec b \prec a$. The safety check of a against c can be omitted, since a already checks against b, and b checks against c. Operation c still has to be checked against a if there is a CFG path via a loop backedge from a to c. With pruning, the worst case number of comparisons reduces to $O(nd)$, where d is the maximum loop depth. For example, in the an FFT code which we later evaluate, the above pruning procedure decreased the number of hazard safety checks from 44 to 10 (32 checks were pruned due to our transitivity property, 2 due to a store to load dependency). Figure 5 shows the result of such pruning.

5.5 Store-to-Load Forwarding

We support store-to-load forwarding by allowing loads to directly access values from a dependent store's pending buffer. We specialize the Hazard Safety Check for RAW dependencies: instead of using the address and schedule of to the most recent store ACK, we use the address and schedule of the next store request. In addition, we

perform an associative search of the pending store buffer, using the load address as a key. If the modified RAW check succeeds, then the dependent value will either already have been committed and ACKed, or it is in the store pending buffer and our associative search will find it. Hits from the buffer search can be used by the load directly, without issuing a DRAM request. If there are multiple values with the same address in the pending buffer, the youngest is chosen (this is cheap to implement in FIFO buffers).

The case where two stores that can both forward a value with the same address to the same load is impossible. Assume the following program order of operations that all use the same address: $store_0 \prec store_1 \prec load$. The $store_1$ will not be able to move its value to its pending buffer until after the $store_0$ value has been ACKed—its WAW hazard detection will stall it. Conversely, the *load* will not use the $store_0$ value, because it will stall on the RAW check against $store_1$—the *load* will wait for $store_1$ to move its value to its pending buffer.

With forwarding, some WAW checks cannot be pruned anymore, because load RAW checks do not use store ACKs. In our above example, if all operations are in the same loop, then the $store_0$ WAW check against $store_1$ cannot be pruned, because the *load* ACK might be updated as a result of store forwarding from $store_1$, with the forwarded value not yet ACKed in $store_1$.

5.6 Intra-Loop RAW Hazards

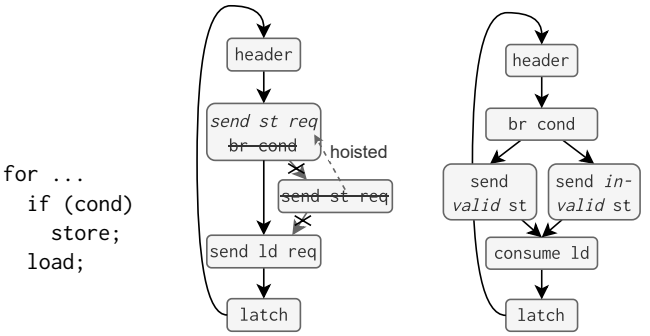
A timely disambiguation of RAW hazards, where both the load and store are in the same loop PE, is crucial since any unnecessary stalls would be repeated on every iteration, resulting in a large throughput reduction. As our evaluation in section 7 will show, store-to-load forwarding becomes crucial in intra-loop RAW dependencies.

In addition to forwarding, there is another term needed in the RAW Hazard Safety Check to make intra-loop RAW hazard checks timely. Consider this simple code:

```
for (i = 0; i < N; ++i)
    d = data[i];
    data[i] = work(d);
```

The load and store address distribution is $\{0, 1, 2, \ldots\}$—there is no actual RAW hazard, but assume that we do not know this at compile time. In this situation, the RAW Hazard Safety Check for a given load at iteration k will only succeed once the next store request in the DU is for iteration $k-1$ and there are no outstanding store ACKs. If the next store request is for an earlier iteration, e.g., an earlier store request is waiting for its store value, then the load would have to be stalled, even though it would be perfectly safe to execute it.

We solve this issue by adding a *NoDependence* single-bit term to the RAW Hazard Safety Check. For each intra-loop RAW hazard pair, *NoDependence* is set in the AGU to the result of $req.address_{load} > req.address_{store}$, where $req.address_{load}$ is the next load address to be sent to the DU, and $req.address_{store}$ is the most recent store address that was sent to the DU. When *NoDependence* is true, and the No Address Reset Check evaluates to true, then the load can be deemed safe since the monotonicity property implies that all store addresses up to $req.schedule_{load}$ are lower than $req.address_{load}$.



(a) Store under *if*-condition. **(b) AGU speculates requests.** **(c) CU kills misspeculations.**

Figure 6: Memory requests in *if*-conditions are speculated.

Note that a similar check is not needed for intra-loop WAW dependencies, since stores do not stall the datapath if sufficient buffering is provided for the store values.

6 Handling Control Flow

The Hazard Safety Check relies on the ability of the DU to detect that a given memory operation has completed a certain schedule time range or a certain address range. This assumes that AGUs supply an operation's schedule and address for every loop iteration. This assumption is broken by operations inside *if*-conditions, which can lead to a deadlock. Consider the code in figure 6(a). If the *if*-condition in this loop is never true, then the store will never update its ACK address and schedule, and thus the RAW Hazard Safety Check in the DU would never succeed. Eventually, the AGU would fill the load request FIFO, resulting in a deadlock.

This could be avoided by using separate AGUs for each memory operation—the store AGU would be guaranteed to at least send a final sentinel value, which would eventually cause the RAW hazard check to succeed. However, this would again mean that some loops need to run to completion before the check can be performed.

A better approach is to *speculatively* send memory requests. We adapt the work presented in [62] to implement speculation in a DAE architecture. In our example, the store request can be hoisted out of the *if*-condition in the AGU. Then, the store values going to the DU from the CU can be tagged with a *valid* bit that signals if the value should be committed or not, depending on the actual control flow at runtime. Figure 6 shows the AGU and CU control-flow graphs that implement such speculation.

Previous work used speculation to remove loss-of-decoupling (LoD) problems in DAE architectures [32, 33, 61]. A LoD arises when the AGU has dependencies on values that have to be loaded from a DU or calculated by a CU, preventing the AGU from running ahead [8]. Our approach is the same as previous work, but we apply it to all *if*-conditions with the goal of producing an $(address, schedule)$ pair for each loop iteration in the AGU. As a side benefit, speculation also makes us immune to the control-dependency LoD problem.

Mis-speculated loads are executed normally in the DU. The read in the CU CFG is moved to the same location where it was speculated in the AGU. This guarantees that the order of load requests

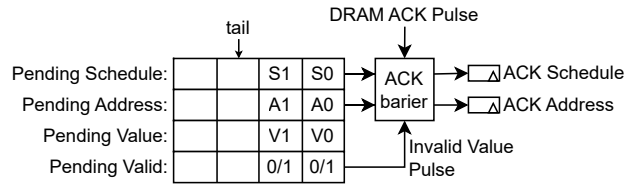


Figure 7: Handling of mis-speculated stores in the DU. Before being moved to the pending buffer, invalid stores are also checked for safety to uphold the transitive property of the Hazard Safety Check. They do not submit DRAM requests. When reaching the head of the pending buffer, they update the ACK registers without having to wait for an ACK.

made from the AGU is the same as the order of load value consumption in the CU, on every CFG path. After reading a speculated load value, the CU can simply not use it if it takes a CFG path where the load value is not needed. Since the basic block location of the speculated load value consumption changes, we also need to adjust any ϕ-nodes that use the load value.

Mis-speculated stores are detected using the *valid* bit in store values coming from the CU. Invalid stores are never committed to memory—there is no need for costly rollbacks. However, invalid stores should eventually update the ACK registers to signal that a given time and address range was completed by the store. Figure 7 shows our approach to this.

If a whole loop with memory operations is under an *if*-condition, then we fold the *if*-condition into the loop body and execute the whole loop speculatively. This was not a performance problem in our evaluation, but future work could investigate a whole loop speculation scheme that does not require executing all loop iterations.

7 Evaluation

We implemented our compiler/hardware co-design in the Intel HLS compiler [36]. Figure 8 shows our tool flow. Our implementation and evaluation are publicly available [59].

7.1 Methodology

We evaluate dynamic loop fusion on ten benchmarks where there is a possibility for parallelism across loops that is not exploited by current static and dynamic HLS tools. All baselines use the Intel HLS compiler:

- `STA`: baseline Intel HLS compiler performing automatic static loop fusion. This approach uses the same dynamically coalescing LSU as our DU.
- `LSQ`: an implementation of dynamic scheduling within the Intel HLS compiler [60]. An LSQ is used for memory accesses, but without support for dynamic coalescing. This approach is representative of all current LSQ implementations in HLS [20, 21, 29, 34, 38, 61].
- `FUS1`: the dynamic loop fusion approach described in this paper, but with no store-to-load forwarding.
- `FUS2`: `FUS1` with store-to-load forwarding enabled.

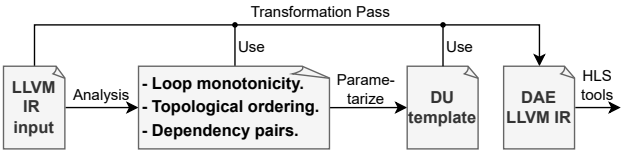


Figure 8: Our compiler/hardware co-design flow. We use the Intel HLS tool in this paper. Our DU is parametrized by the number of loads and stores. The DU disambiguation logic is parameterized for each hazard pair (dependency source and destination) based on the loop nest monotonicity of the dependency source; and the relative topological ordering of the dependency source relative to the destination.

We execute our benchmarks in hardware on the Altera Arria 10 GX1150 FPGA board [37] with 2 banks of DDR4 memory (the memory controller uses two 512-bit channels). We use large datasets to ensure data is distributed across DRAM pages, resulting in variable latency. Each code is executed three times and the minimum time is reported. Area, reported as Adaptive Logic Modules (ALMs) [35], and frequency are taken from Quartus 19.2 reports after place and route. Our approach does not increase DSPs and BRAMs.

7.2 Benchmarks

We use irregular codes from dynamic HLS research [13, 40, 60], choosing codes where there are sibling loops that can benefit from our dynamic loop fusion. For some benchmarks, we unroll outer loops to expose two inner loops that can be dynamically fused; or we compose multiple kernels to simulate applications composed of multiple tasks. Some codes have address expressions that can be analyzed for monotonicity; some codes use data-dependent accesses that are asserted to be monotonic by the programmer. We now list our benchmarks and the parameters used in this evaluation:

- **RAWloop, WARloop, WAWloop:** each benchmark has two loops, each with one memory access, forming a RAW, WAR, or WAW dependency across loops. We use these benchmarks to compare our speedup to the maximum theoretical speedup. Complexity $O(n)$. We set $n = 10\,000\,000$.
- **bnn:** one layer of a sparse binarized neural network. There are two loops, both with data-dependent accesses that prevent fusion. We mark the inner loops as monotonic since we know that the sparse representation is monotonic. Complexity $O(n^2)$. We set $n = 10\,000$.
- **pagerank:** uses a compressed sparse row (CSR) format to iterate over the graph. Another two loops in the algorithm have a regular access pattern, but they cannot be fused because the irregular loop is between them. Complexity $O(iters\ (nodes + edges))$. We set $iters = 10$, $nodes = 325\,729$, $edges = 1\,497\,134$ using the *web-NotreDame* graph [42].
- **fft:** an FFT with the middle loop unrolled by a factor of two. The non-affine accesses prevent loop fusion. The `LSQ` and `STA` approach is equivalent for fft, because there are no hazards within loops that would need an LSQ. Complexity $O(n \log n)$. We set $n = 1\,048\,576$.
- **matpower:** sparse matrix power using the CSR format with the outer loop unrolled by a factor of 2. Complexity $O(nz^3)$,

Table 1: Performance, area usage, and circuit frequency of the STA, LSQ [60], FUS1, and FUS2 approaches. The second column reports the number of PEs and DUs generated by our FUS approach, together with loads and stores per DU.

Kernel	Number of				Area in 1000s of ALMs				Freq in MHz				Time in seconds			
	PE	DU	LD	ST	STA	LSQ	FUS1	FUS2	STA	LSQ	FUS1	FUS2	STA	LSQ	FUS1	FUS2
RAWloop	2	1	1	1	78	79.6	82.5	83.3	304	268	263	239	6.8	33.3	3.9	4.4
WARloop	2	1	1	1	78.1	79.6	82.2	82.2	279	264	261	261	7.1	33.5	4.1	4.1
WAWloop	2	1	1	1	78.3	80.8	88.4	88.4	294	269	251	251	6.8	7.5	4.1	4.1
bnn	2	1	2	2	78.9	85.1	93.5	95.2	279	244	266	257	39.2	3.2	1.6	1.6
pagerank	3	2	2/1	2/1	81.5	87.8	114.1	115.2	262	237	246	246	35.7	0.8	1.6	0.7
fft	2	2	4/4	4/4	102.7	102.7	150.4	152.2	246	246	221	219	7.8	7.8	2.8	1.7
matpower	2	1	4	2	82.1	97.6	105.4	108.6	274	193	260	257	18	3.7	12.3	1.6
hist+add	3	2	2/2	1/1	79.2	87.9	97.0	99.3	286	220	282	270	3.9	1	0.2	0.2
tanh+spmv	2	2	2/1	1/1	80.2	93.1	99.5	101.8	274	225	260	264	4.4	0.9	0.5	0.5
				Harmonic Mean:	1	1.07	1.22	1.24	1	0.86	0.92	0.9	1	0.12	0.1	0.07

where nz is the number of non-zero matrix values. We set $nz = 4\,096$.

- **hist+add:** addition of two histograms. The STA approach can fuse the two histogram loops, but not the addition. Three $O(n)$ loops. We set $n = 10\,000\,000$.
- **tanh+spmv:** *tanh* applied to a vector before it is used in a COO sparse matrix-vector multiplication. The *tanh* loop has a store in an *if*-condition, which we speculate. One $O(n)$ loop followed by a $O(nz)$ loop, where nz is the number of non-zero matrix values. We set $n = 10\,000$, $nz \simeq 10\,000$.

7.3 Results

Table 1 shows the area and performance results for the four approaches that we evaluate. Dynamic loop fusion with forwarding is on average **14× faster than static HLS** and **4× faster than dynamic HLS** that uses an LSQ.

7.3.1 Theoretical Speedup. The RAW/WAR/WAW loop benchmarks have a theoretical speedup of 2×, but FUS2 achieves a speedup of around 1.7×. The lower speedup is due to the lower FUS2 circuit frequency on these benchmarks. The LSQ approach sees a slowdown relative to STA in the RAW/WAR loop benchmarks, because it cannot use a dynamically bursting LSU which stalls the load loop significantly (the LSQ used in [60] uses a non-bursting LSU to gurantee that hazards are not violated [61]). Store loops, e.g., WAWloop, do not suffer as much from a lack of bursting in the LSQ approach, because stores do not stall the LSQ pipeline.

7.3.2 Store-to-Load Forwarding Impact. We observe that forwarding has no observable benefit on codes where the forwarding happens across loops, e.g., RAWloop. This is expected in our evaluation setup, since without forwarding, the only penalty is an initial wait for the store ACK to be updated. Forwarding across loops may become beneficial if the DRAM bandwidth becomes a bottleneck, which is likely to occur in practice once data parallelism is exploited. Forwarding becomes crucial if the store and load are in the same loop and the dependency distance is lower than the store latency (e.g., fft, matpower, or pagerank). Future work could use a more precise cost model and enable forwarding only where beneficial, e.g., always use forwarding for RAW dependencies inside loops, but for RAW dependencies across loops enable it only once the memory bandwidth is saturated.

7.3.3 Which Codes Benefit from Dynamic Loop Fusion? It only makes sense to fuse loops with similar time complexities. Consider the pagerank benchmark as an example where fusion offers only a modest 1.1× speedup over the LSQ approach. The code consists of two $O(n)$ loops which go over graph nodes and one $O(n^2)$ loop which goes over edges. Even if all three loops are fused, the runtime will still be dominated by the $O(n^2)$ loop. We used the web-Google graph [42] with 875,713 nodes and 5,105,039 edges, which only has a theoretical speedup of ≈ 1.3 over LSQ.

We see the biggest benefit of using dynamic loop fusion in the ability to unroll outer loops of irregular codes without having to worry about breaking data dependencies (e.g., fft and matpower), and in the ability to perform task fusion at a fine-grained level (e.g., hist+add and tanh+spmv).

7.3.4 Area Overhead. Dynamic loop fusion with forwarding comes at an average area increase of 24% and frequency degradation of 9% over static HLS. The most area-hungry component is the dynamically coalescing LSU. The STA approach also uses the costly coalescing LSUs, which amortizes the area overhead of fusion. The LSQ approach uses a simpler LSU, which explains its low area overhead.

For example, in the RAWloop benchmark, the FUS2 DU consumes 1,550 ALMs (1,200 of which are dedicated to the pending buffers and its associative searching), whereas a single load LSU consumes 2,840 ALMs and the DRAM interconnect consumes 68,089 ALMs. If the OpenCL kernel runtime and DRAM interconnect are not counted, then our area overhead of dynamic loop fusion with forwarding increases to 2.1×. However, codes not using DRAM will not need the area budget for pending buffers, resulting in an overhead closer to what we report in table 1.

Hazard pairs pruning has a large impact on the area and critical path of codes with many loads and stores. For example, the FFT code uses two DUs, each with 4 loads and stores. The unpruned FFT FUS2 version uses 32% more area and achieves a 28% lower frequency than the pruned version.

8 Limitations and Future Work

Our dynamic loop fusion approach can be integrated with common loop transformations used in HLS. For example, loop tiling does not break the monotonicity of inner loops. Dataflow designs—concurrent loops communicating via FIFOs–are automatically generated by our compiler given the program loop forest, as shown in Figure 3. Loop unrolling—replicating the datapath of the inner loop—is also compatible with our DU, since we do not impose any limits on the number of memory ports. However, our current implementation does not work with automatic loop unroll pragmas, and manual unrolling is needed instead. This is because unrolling pragmas are typically implemented in the closed-source back-end of vendor compilers, and our compiler passes operate on LLVM IR in the middle-end. We do not study unrolling in this work, because the types of irregular codes that we consider in this paper do not lend themselves to the same automatic parallelization as regular codes, e.g., due to unpredictable loop-carried dependencies.

In this work, we consider DRAM streaming applications, relying on a dynamically bursting and coalescing LSU to discover memory parallelism at runtime and on store-to-load forwarding to increase temporal locality. In our current design, the amount of on-chip data reuse is limited by the size of the pending buffers, which have to be kept small to make associative searching feasible. A cache memory hierarchy implemented in BRAM could further decrease the number of DRAM requests and increase temporal locality. Recent work has advanced the state-of-the-art of non-blocking caches on FPGAs by storing Miss Status Holding Registers (MSHRs) in BRAM and using hash-based, instead of associative, searching [3, 72]. In a DU with cache, the pending buffers could be changed to MSHRs with added schedule information and our store-to-load forwarding could be removed altogether, since temporal locality would be provided by the cache.

Using a BRAM-based cache and loop unrolling are orthogonal goals—supporting multiple memory ports is cheaper to do in BRAM than in DRAM, both in terms of the circuit area and available bandwidth. However, since the automatic partitioning of BRAM into multiple banks cannot be performed for irregular code, a memory arbiter would have to be integrated to support multiple BRAM ports, as for example in [14, 73, 74].

9 Related Work

Our loop monotonicity analysis benefits from decades of research on abstract interpretation of recurrences [2, 5, 9, 43, 49, 64]. Loop monotonicity has first been exploited in a practical setting by Gupta *et al.* to synthesize race detection runtime checks in fork-join parallel programs [31]. However, they did not consider shared loops and non-monotonic outer loops.

We discussed the basics of the polyhedral compiler transformation framework in section 2.2, stating that codes with non-affine memory addresses or loop bounds cannot be analyzed. Recent work has combined the Inspector/Executor (I/E) approach [24, 46, 56] with polyhedral transformations [54, 58, 67]. The idea behind the I/E approach is to generate inspector code which gathers values of variables unknown at compile-time; and/or rearranges data structures in memory for better locality and to increase dependency distances. The small overhead of the inspector code is offset by the throughput improvement obtained in the executor code. For example, Strout *el al.* proposed the Sparse Polyhedral Framework which uses "uninterpretable functions" to represent non-affine terms such as data-dependent memory accesses [58]. By proving basic properties about an uninterpretable function in the inspector code (e.g., monotonicity) a large amount of potential data dependencies can be ruled out, allowing the executor code to exploit more parallelism [66]. Most recently, [18] proposed sparse fusion, an I/E technique that inspects the access patterns of multiple irregular loops and then creates an execution schedule that allows sibling loops to execute in parallel. Although we share the same goal as these works, our approach is fundamentally different. Instead of relying on inspector code to discover data dependencies, we resolve data dependencies "on the fly" in our DU. We also provide a finer-grained monotonicity compiler analysis, discovering which specific loop depths cause an address expression to visit an earlier value, rather than deciding on the monotonicity of the entire loop nest expression.

All previous work on dynamic memory disambiguation in HLS sequentializes loops that share a data dependency [1, 25, 29, 38, 61]. Cheng *el al.* investigated compile time checks to prove that two loops do not access the same memory locations [16]—their approach is the same as existing polyhedral optimizers, but uses a different formulation. Others have exploited the SCEV framework to augment the static analysis with dynamic checks in HLS [23, 44, 45]—these approaches are similar to multi-versioned SIMD CPU code, where the fast (SIMD) path is taken if a set of conditions evaluates to true at runtime. All these works either only improve the throughput of single loops, or execute separate loops in parallel only if all iterations are independent.

Winterstein *et al.* [68] used symbolic execution, based on separation logic, to prove the absence of aliasing when unrolling irregular loops into multiple PEs. Later, they expanded the work to support aliasing limited to commutative operations by using locks to access a shared memory space [70]. They rely on the commutative property because they cannot guarantee sequential consistency of accesses to the same memory spaces, as we have proposed here.

10 Conclusions

We have presented dynamic loop fusion, a compiler/hardware co-design approach that enables dynamic memory disambiguation across monotonic loops without the need for address history searches. Our hazard detection logic is enabled by a novel program-order schedule representation, and by assuming monotonically non-decreasing addresses are in inner loops. We have presented a compiler analysis, based on the chain of recurrences formalism, to detect loop monotonicity. We have also shown that most codes contain addresses that are monotonic, making our approach applicable to a large class of applications. On an evaluation of 10 irregular codes, dynamic loop fusion provided an average speedup of 14× over static HLS and 4× over dynamic HLS.

Acknowledgments

This work was partly supported by a PhD scholarship from the UK EPSRC. We thank the anonymous reviewers and our shepherd Louis-Noël Pouchet for improving this paper. We thank Intel for free access to FPGAs through their FPGA DevCloud.

References

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

[2] Zahira Ammarguellat and Williams Ludwell Harrison III. 1990. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 283–295.

[3] Mikhail Asiatici and Paolo Ienne. 2019. Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 310–319. https://doi.org/10.1145/3289602.3293901

[4] Mikhail Asiatici and Paolo Ienne. 2021. Request, Coalesce, Serve, and Forget: Miss-Optimized Memory Systems for Bandwidth-Bound Cache-Unfriendly Applications on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 15, 2, Article 13 (dec 2021), 33 pages. https://doi.org/10.1145/3466823

[5] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. 1994. Chains of Recurrences—a Method to Expedite the Evaluation of Closed-Form Functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. https://doi.org/10.1145/190347.190423

[6] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *Compiler Construction*, Rajiv Gupta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–303.

[7] Daniel Berlin and David Edelsohn. 2004. High-level loop optimizations for GCC. *Proceedings of the 2004 GCC Developers Summit* (01 2004).

[8] 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

[9] W. Blume and R. Eigenmann. 1994. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, Vol. 2. 233–238. https://doi.org/10.1109/ICPP.1994.59

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.* 43, 6 (June 2008). https://doi.org/10.1145/1379022.1375595

[11] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. 2014. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2014.6927490

[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. https://doi.org/10.1109/MICRO.2016.7783749

[13] Jianyi Cheng. 2019. JianyiCheng: HLS_Benchmarks_First_Release. https://doi.org/10.5281/zenodo.3561115

[14] Jianyi Cheng, Shane T. Fleming, Yu Ting Chen, Jason Anderson, John Wickerson, and George A. Constantinides. 2022. Efficient Memory Arbitration in High-Level Synthesis From Multi-Threaded Code. *IEEE Trans. Comput.* 71, 4 (2022), 933–946. https://doi.org/10.1109/TC.2021.3066466

[15] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. 2022. DASS: Combining Dynamic amp; Static Scheduling in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022). https://doi.org/10.1109/TCAD.2021.3065902

[16] Jianyi Cheng, Lana Josipović, George A. Constantinides, and John Wickerson. 2022. Dynamic Inter-Block Scheduling for HLS. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 243–252. https://doi.org/10.1109/FPL57034.2022.00045

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

[18] Kazem Cheshmi, Michelle Strout, and Maryam Mehri Dehnavi. 2023. Runtime Composition of Iterations for Fusing Loop-carried Sparse Dependence. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Article 89, 15 pages. https://doi.org/10.1145/3581784.3607097

[19] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 204–213. https://doi.org/10.1109/FCCM51124.2021.00032

[20] Steve Dai, Gai Liu, Ritchie Zhao, and Zhiru Zhang. 2017. Enabling adaptive loop pipelining in high-level synthesis. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 131–135. https://doi.org/10.1109/ACSSC.2017.8335152

[21] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. 2017. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 189–194. https://doi.org/10.1145/3020078.3021754

[22] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. 2021. *StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems*. IEEE Press, 315–326. https://doi.org/10.1109/CGO51591.2021.9370315

[23] Florian Dewald, Johanna Rohde, Christian Hochberger, and Heiko Mantel. 2022. Improving Loop Parallelization by a Combination of Static and Dynamic Analyses in HLS. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 31 (feb 2022), 31 pages. https://doi.org/10.1145/3501801

[24] Chen Ding and Ken Kennedy. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 229–241. https://doi.org/10.1145/301618.301670

[25] Ayatallah Elakhras, Riya Sawhney, Andrea Guerrieri, Lana Josipovic, and Paolo Ienne. 2023. Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 39–45. https://doi.org/10.1145/3543622.3573050

[26] Kermin Elliott Fleming. 2013. *Scalable reconfigurable computing leveraging latency-insensitive channels*. Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. https://hdl.handle.net/1721.1/79212

[27] 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

[28] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming* 34 (2006), 261–317.

[29] 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

[30] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 04 (2012). https://doi.org/10.1142/S0129626412500107

[31] Rajiv Gupta and Madalene Spezialetti. 1991. Loop monotonic computations: an approach for the efficient run-time detection of races. In *Proceedings of the Symposium on Testing, Analysis, and Verification* (Victoria, British Columbia, Canada) *(TAV4)*. Association for Computing Machinery, New York, NY, USA, 98–111. https://doi.org/10.1145/120807.120816

[32] 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

[33] 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

[34] Jing Huang, Yuanjie Huang, Olivier Temam, Paolo Ienne, Yunji Chen, and Chengyong Wu. 2014. A low-cost memory interface for high-throughput accelerators. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (New Delhi, India) *(CASES '14)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/2656106.2656109

[35] 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.

[36] Intel. 2024. Compiler Handbook for Intel FPGAs. https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/developer-guide/2024-1/intel-oneapi-dpc-c-compiler-handbook-for-intel.html. Accessed: 2024-08-02.

[37] Intel. 2024. Intel® PAC with Intel® Arria® 10 GX FPGA - Product Specification. https://www.intel.com/content/www/us/en/products/sku/149169/intel-pac-with-intel-arria-10-gx-fpga/specifications.html. [Accessed 18-09-2024].

[38] Lana Josipovic, Philip Brisk, and Paolo Ienne. 2017. An Out-of-Order Load-Store Queue for Spatial Computing. *ACM Transactions on Embedded Computing Systems* (2017). https://doi.org/10.1145/3126525

[39] Lana Josipović, Atri Bhattacharyya, Andrea Guerrieri, and Paolo Ienne. 2019. Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs. In *2019*

International Conference on Field-Programmable Technology. https://doi.org/10.1109/ICFPT47387.2019.00031

[40] 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

[41] Ken Kennedy and John R. Allen. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[42] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[43] Yuan Lin and David Padua. 2000. Compiler analysis of irregular memory accesses. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/349299.349322

[44] Junyi Liu, Samuel Bayliss, and George A. Constantinides. 2015. Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 159–162. https://doi.org/10.1109/FCCM.2015.31

[45] Junyi Liu, John Wickerson, Samuel Bayliss, and George A. Constantinides. 2018. Polyhedral-Based Dynamic Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 9 (2018), 1802–1815. https://doi.org/10.1109/TCAD.2017.2783363

[46] N. Mitchell, L. Carter, and J. Ferrante. 1999. Localizing non-affine array references. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*. 192–202. https://doi.org/10.1109/PACT.1999.807526

[47] 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

[48] Sebastian Pop, Philippe Clauss, Albert Cohen, Vincent Loechner, and Georges-André Silber. 2004. Fast recognition of scalar evolutions on three-address ssa code. *CRI/ENSMP Research Report, A/354/CRI* (2004), 1–28.

[49] Sebastian Pop, Albert Cohen, and Georges-André Silber. 2005. Induction Variable Analysis with Delayed Abstractions. In *High Performance Embedded Architectures and Compilers*, Tom Conte, Nacho Navarro, Wen-mei W. Hwu, Mateo Valero, and Theo Ungerer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 218–232.

[50] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: convexity, pruning and optimization. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. https://doi.org/10.1145/1926385.1926449

[51] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. 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* (Virtual, USA) *(ASPLOS '21)*. 600–615. https://doi.org/10.1145/3445814.3446723

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

[53] 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

[54] Mahesh Ravishankar, John Eisenlohr, Louis-Noel Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2012. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.30

[55] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou, and Jason Cong. 2018. ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 9–16. https://doi.org/10.1109/FCCM.2018.00011

[56] J.H. Saltz, R. Mirchandaney, and K. Crowley. 1991. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.* 40, 5 (1991), 603–612. https://doi.org/10.1109/12.88484

[57] 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.

[58] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721

[59] Robert Szafarczyk. 2024. *Dynamic Loop Fusion in HLS - FPGA25 artifact*. https://doi.org/10.5281/zenodo.14383450

[60] 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. https://doi.org/10.1109/FPL60245.2023.00009

[61] 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. https://doi.org/10.1109/ICFPT59805.2023.00018

[62] 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* (Las Vegas, United States) *(CC 2025)*. To Appear.

[63] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A Framework for Massively Parallel Streaming on FPGAs. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 639–651. https://doi.org/10.1145/3373376.3378495

[64] Robert A. van Engelen, J. Birch, Y. Shou, B. Walsh, and Kyle A. Gallivan. 2004. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the 18th Annual International Conference on Supercomputing* (Malo, France) *(ICS '04)*. Association for Computing Machinery, New York, NY, USA, 106–115. https://doi.org/10.1145/1006209.1006226

[65] Wim Vanderbauwhede, Syed Waqar Nabi, and Cristian Urlea. 2019. Type-Driven Automated Program Transformations and Cost Modelling for Optimising Streaming Programs on FPGAs. *International Journal of Parallel Programming* 47 (02 2019). https://doi.org/10.1007/s10766-018-0572-z

[66] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating Wavefront Parallelization for Sparse Matrix Computations. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 480–491. https://doi.org/10.1109/SC.2016.40

[67] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2018. Non-affine Extensions to Polyhedral Code Generation. 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, 185–194. https://doi.org/10.1145/2544137.2544141

[68] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. 2014. Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 1–8. https://doi.org/10.1109/FCCM.2014.11

[69] Felix Winterstein and George Constantinides. 2017. Pass a pointer: Exploring shared virtual memory abstractions in OpenCL tools for FPGAs. In *2017 International Conference on Field Programmable Technology (ICFPT)*. 104–111. https://doi.org/10.1109/FPT.2017.8280127

[70] Felix Winterstein, Kermin Fleming, Hsin-Jung Yang, Samuel Bayliss, and George Constantinides. 2015. MATCHUP: Memory Abstractions for Heap Manipulating Programs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. Association for Computing Machinery, New York, NY, USA, 136–145. https://doi.org/10.1145/2684746.2689073

[71] Peng Wu, Albert Cohen, Jay Hoeflinger, and David Padua. 2001. Monotonic evolution: an alternative to induction variable substitution for dependence analysis. In *Proceedings of the 15th International Conference on Supercomputing* (Sorrento, Italy) *(ICS '01)*. Association for Computing Machinery, New York, NY, USA, 78–91. https://doi.org/10.1145/377792.377809

[72] Shaoxian Xu, Sitong Lu, Zhiyuan Shao, Xiaofei Liao, and Hai Jin. 2024. MiCache: An MSHR-inclusive Non-blocking Cache Design for FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 22–32. https://doi.org/10.1145/3626202.3637571

[73] Hsin Jung Yang, Kermin Fleming, Michael Adler, and Joel Emer. 2014. LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 117–124. https://doi.org/10.1109/FCCM.2014.43

[74] Hsin-Jung Yang, Kermin Fleming, Michael Adler, Felix Winterstein, and Joel Emer. 2016. LMC: Automatic Resource-Aware Program-Optimized Memory Partitioning. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*. Association for Computing Machinery, New York, NY, USA, 128–137. https://doi.org/10.1145/2847263.2847283