Predictable Accelerator Design with Time-Sensitive Affine Types

Anonymous Author(s)

Abstract

While field-programmable gate arrays (FPGAs) provide an opportunity to co-design applications with hardware accelerators, they remain difficult to program. *High-level synthesis* (HLS) tools promise to raise the level of abstraction by compiling C or C++ to accelerator designs. Repurposing legacy software languages, however, requires complex heuristics to automatically map unrestricted imperative code onto hardware structures. We find that the black-box heuristics in HLS tools can be *unpredictable:* changing parameters in the program that should improve performance can counterintuitively yield slower and larger FPGA implementations.

This paper proposes a type system that restricts HLS to programs that can predictably compile to hardware accelerators. The key idea is to model consumable hardware resources with a *time-sensitive affine type system* that prevents conflicting simultaneous uses of the same hardware structure. We implement the type system in Dahlia, a programming language that compiles to HLS C++, and evaluate how its type system can reduce the size of HLS parameter spaces while accepting Pareto-optimal designs.

1 Introduction

While Moore's law may not be dead yet, its stalled returns for traditional CPUs have sparked renewed interest in specialized hardware accelerators [28], for domains from machine learning [31] to genomics [53]. Reconfigurable hardware—namely, field-programmable gate arrays (FPGAs)—offer some of the benefits of specialization without the cost of custom silicon. FPGAs can accelerate code in domains from databases [12] to genomics [11] and have driven vast efficiency improvements in Microsoft's datacenters [43, 20].

However, FPGAs are hard to program. The gold-standard programming model for FPGAs is register transfer level (RTL) design in hardware description languages such as Verilog, VHDL, Bluespec, and Chisel [38, 5]. RTL requires digital design expertise: akin to assembly languages for CPUs, RTL is irreplaceable for manual performance tuning, but it is too explicit and verbose for rapid iteration [50].

FPGA vendors offer *high-level synthesis* (HLS) or "C-to-gates" tools [55, 17, 40, 10] that translate annotated subsets of C and C++ to RTL. Repurposing a legacy software languages, however, has drawbacks: the resulting language subset is small and difficult to specify, and minor code edits can cause large swings in hardware efficiency. We find empirically that smoothly changing source-level hints can cause wild variations in accelerator performance. Semantically, *there is no HLS programming language:* there is only the subset of C++ that a particular version of a particular compiler supports.

This paper describes a type system that restricts HLS to programs whose hardware implementation is clear. The goal is *predictable* architecture generation: the architectural implications are manifest in the source code, and costly implementation decisions require explicit permission from the programmer. Instead of silently generating bad hardware for difficult input programs, the type system yields errors that help guide the programmer toward a better design. The result is a language that can express a subset of the architectures that HLS can. In return, the language makes the architectural implications manifest in the source code and lets programmers make trade-offs in a pruned design space.

The central insight is that an affine type system [51] can model the restrictions of hardware implementation. Components in a hardware design are finite and expendable: a subcircuit or a memory can only do one thing at a time, so a program needs to avoid conflicting uses of any given component. Previous research has shown how to apply substructural type systems to model classic computational resources such as memory allocations and file handles [24, 7, 36, 51] and to enforce exclusion for safe shared-memory parallelism [23, 6, 14]. Unlike those classic resources, however, the availability of hardware components changes with time. We extend affine types with *time sensitivity* using the insight that repeated uses of the same resources is safe as long as they are temporally separated, i.e., the circuits using the resources do not run simultaneously.

We describe Dahlia, a programming language for predictable accelerator design. Dahlia differs from HLS languages in two ways: 1) Dahlia makes the hardware implementation for each language construct manifest in the source code instead of leaving this decision up to the HLS middle-end and 2) Dahlia uses its *time-sensitive affine types* to reason about the hardware constraints and reject programs when the program would require complex transformation to be implemented in hardware. We implement a compiler for Dahlia that emits annotated C++ for a commercial HLS toolchain. Instead of generating RTL directly, Dahlia adds predictability to the existing high-performance optimizations that HLS tools offer. Dahlia makes it easier for programmers to understand *why* certain design parameters cause unpredictable

PLDI'20, June 15–20, 2020, London, United Kingdom
2018.

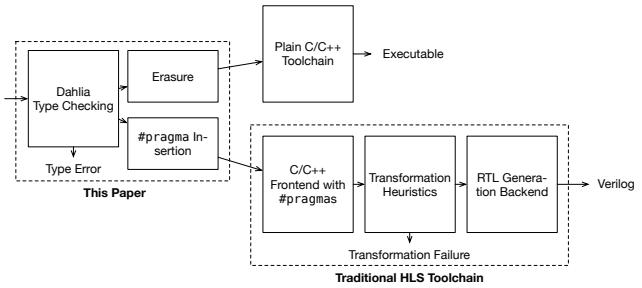


Figure 1. Overview of a traditional high-level synthesis toolchain and how Dahlia layers type safety on top.

hardware generation through the lens of resource and scheduling constraints. We show in this paper that predictability pitfalls exist in both industrial and recent academic tools, and Dahlia's reasoning can help alleviate these issues.

The contributions of this paper are:

- We identify predictability pitfalls in HLS and measure their effects in an industrial tool in Section 2.
- We design Dahlia, a language we describe in Section 3 that restricts HLS to predictable design spaces by modeling the constraints of hardware generation using *time-sensitive affine types*.
- We formalize a time-sensitive affine type system and prove syntactic type soundness in Section 4.
- We empirically demonstrate Dahlia's effectiveness in rejecting unpredictable design points and its ability to make area–performance trade-offs in common accelerator designs in Section 5.

2 Predictability Pitfalls in Traditional HLS

Figure 1 depicts the design of a traditional high-level synthesis (HLS) compiler. A typical HLS tool adopts an existing open-source C/C++ frontend and adds a set of *transformation heuristics* that attempt to map software constructs onto hardware elements along with a backend that generates RTL code [16, 10]. The transformation step typically relies on a constraint solver, such as an LP or SAT solver, to satisfy resource, layout, and timing requirements [25, 18]. Programmers can add `#pragma` hints to guide the transformation—for example, to duplicate loop bodies or to share functional units.

HLS tools are best-effort compilers: they make a heuristic effort to translate *any* valid C/C++ program to RTL, regardless of the consequences for the generated accelerator architecture. Sometimes, the mapping constraints are unsatisfiable, so the compiler selectively ignores some `#pragma` hints or issues an error. The generated accelerator's efficiency depends on the interaction between the code, the hints, and the transformation heuristics that use them.

The standard approach prioritizes automation over predictability. Small code changes can yield large shifts in the generated architecture. When performance is poor, the compiler provides little guidance about how to improve it. Pruning such *unpredictable* points from the design space would let programmers explore smaller, smoother parameter spaces.

```
1 int m1[512][512], m2[512][512], prod[512][512];
2 int sum;
3 for (int i = 0; i < 512; i++) {
4   for (int j = 0; j < 512; j++) {
5     sum = 0;
6     for (int k = 0; k < 512; k++) {
7       sum += m1[i][k] * m2[k][j];
8     }
9     prod[i][j] = sum; } }
```

Figure 2. Dense matrix multiplication in HLS-friendly C.

2.1 An Example in HLS

Programming with HLS centers on arrays and loops, which correspond to memory banks and logic blocks. Figure 2 shows the C code for a matrix multiplication kernel. This section imagines the journey of a programmer attempting to use HLS to generate a fast FPGA-based accelerator from this code. We use Xilinx's SDAccel compiler (v2018.3.op) and target UltraScale+ VU9P FGPA on the AWS F1 platform to perform the experiments mentioned in the section.

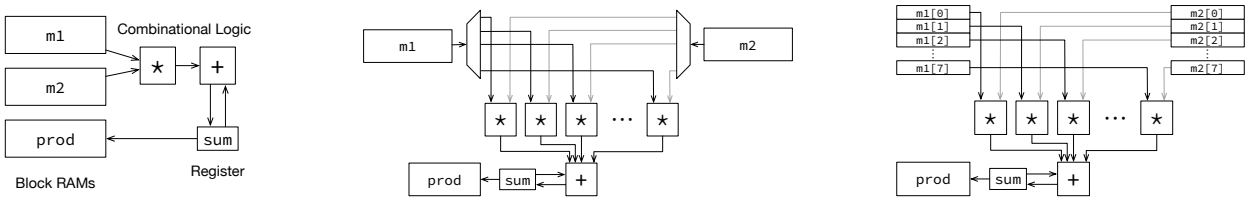
Initial accelerator. Our imaginary programmer might first try compiling the code in Figure 2 verbatim. The HLS tool maps the arrays `m1`, `m2`, and `prod` onto on-chip memories. FPGAs have SRAM arrays, called *block RAMs* (BRAMs), that the compiler allocates for this purpose. The loop body becomes combinational logic consisting of a multiplier, an adder, and an accumulator register. Figure 3a depicts this configuration.

This design, while functional, does not harness any parallelism that an FPGA can offer. The two key metrics for evaluating an accelerator design are performance and area, i.e., the amount of physical chip resources that the accelerator occupies. This initial configuration computes the matrix product in 841.1 ms and occupies 2355 of the device's lookup tables (LUTs). However, the target FPGA has over 1 million LUTs on the device fabric, so the programmer's next job is to expend more of the FPGA area to improve performance.

Loop unrolling. The standard tool that HLS offers for expressing parallelism is an `UNROLL` annotation, which duplicates the logic for a loop body. A programmer might attempt to obtain a better accelerator design by adding this annotation in the innermost loop on lines 6–8

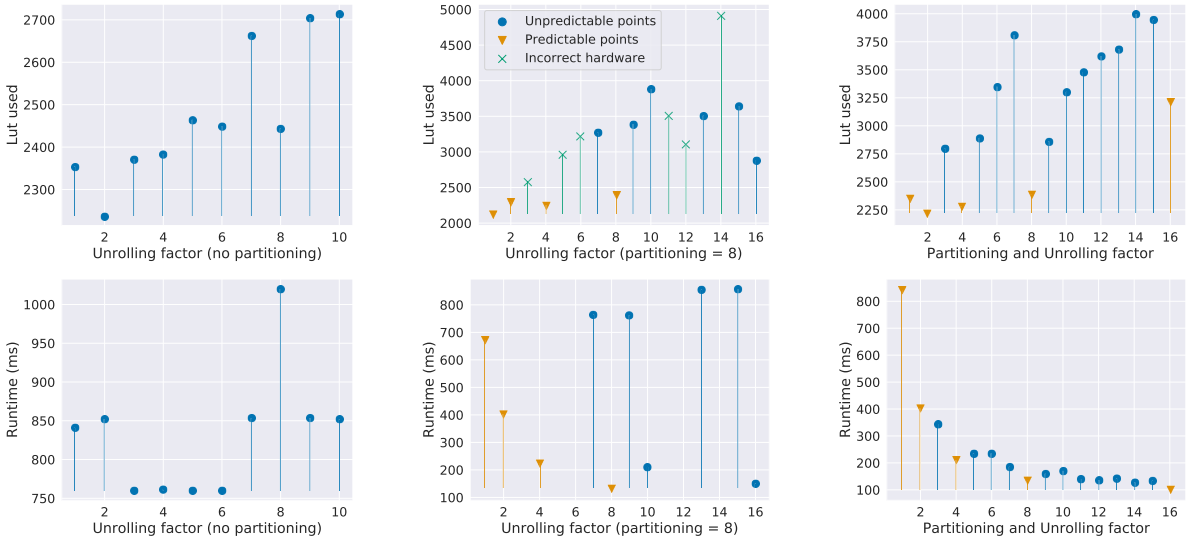
```
#pragma HLS UNROLL FACTOR=8
```

This unrolling directive instructs the HLS tool to create 8 copies of the multiplier and adder, called *processing elements* (PEs), and attempt to run them in parallel. Loop unrolling represents an area–performance trade-off: programmers can reasonably expect greater unrolling factors to consume more of the FPGA chip but yield lower-latency execution.



(a) The original code. **(b)** With unrolling. **(c)** With unrolling and banking.

Figure 3. Three accelerator implementations of the matrix multiplication in Figure 2.



(a) Unrolling without partitioning **(b)** Unrolling with 8-way partitioning **(c)** Unrolling and banking in lockstep

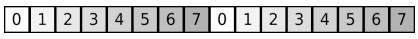
Figure 4. Look-up table count (top) and execution latency (bottom) for the kernel in Figure 2 with varying parameters.

The `UNROLL` directive alone, however, fails to achieve this effect. Figure 4a shows the effect of various unrolling factors on this code in area (LUT count) and performance (latency). There is no clear trend: greater unrolling yields unpredictably better and worse designs. The problem is that the accelerator's memories now bottleneck the parallelism provided by the PEs. The BRAMs in an FPGA have a fixed, small number of *ports*, so they can only service one or two reads or writes at a time. So while the HLS tool obeys the programmer's `UNROLL` request to duplicate PEs, its scheduling must serialize their execution. Figure 3b shows how the HLS tool must insert additional *multiplexing* hardware to connect the multipliers to the single-ported memories. The additional hardware and the lack of parallelism yields the unpredictable performance and area for different PE counts.

Memory banking to match parallelism. To achieve proper speedups from parallelism, accelerators need to use multiple memories. HLS tools provide annotations to *partition* arrays, allocating multiple BRAMs and increasing the access throughput. The programmer can insert these partitioning annotations to allocate 8 BRAMs per input memory:

```
#pragma HLS ARRAY_PARTITION VARIABLE=m1 FACTOR=8
```

```
#pragma HLS ARRAY_PARTITION VARIABLE=m2 FACTOR=8
```



A banked memory consists of several physical memories, each of which stores a subset of the array's data. The compiler partitions the array using a "round-robin" policy to enable parallel access. In this example, elements 0 and 8 go in bank 0, elements 1 and 9 go in bank 1, etc.

Figure 3c shows the resulting architecture, which requires no muxing and allows memory parallel access.

Combining banking and unrolling, however, unearths another source of unpredictable performance. While the HLS tool produces a good result when both the banking factors and the loop unrolling factor are 8, other design choices perform worse. Figure 4b shows the effect of varying the unrolling factor while keeping the arrays partitioned with factor 8. Again, the area and performance varies unpredictably with the unrolling factor. Reducing the unrolling factor from 9 to 8 can counter-intuitively *improve* both performance and area. In our experiments, some unrolling factors yield hardware that produces incorrect results. (We show the area but omit the running time for these configurations.)

The problem is that some partitioning/unrolling combinations yield much simpler hardware than others. When both the unrolling and the banking factors are 8, each parallel PE need only access a single bank, as in Figure 3c. The first PE needs to access elements 0, 8, 16, and so on—and because the array elements are "striped" across the banks, all of these values live in the first bank. With unrolling factor 9, however, the first PE needs to access values from *every* bank, which requires complicated memory indirection hardware. With unrolling factor 4, the indirection cost is smaller—the first PE needs to access only bank 0 and bank 4.

From the programmer's perspective, the HLS compiler silently enforces an unwritten rule: *When the unrolling factor divides the banking factor, the area is good and parallelism predictably improves performance. Otherwise, all bets are off.* Figure 4b labels the points where the unrolling factor divides the banking factor as *predictable points*. The HLS compiler emits no errors or warnings for any parameter setting.

Banking vs. array size. Even if we imagine that a programmer carefully ensures that banking factors exactly match unrolling factors, another pitfall awaits them when choosing the amount of parallelism. Figure 4c shows the effects of varying the banking and unrolling factor in our kernel *together*. The LUT count again varies wildly.

The problem is that, when the banking and unrolling factors do not evenly divide the sizes of the arrays involved, the accelerator needs extra hardware to cope with the "leftover" elements. The memory banks are unevenly sized, and the PEs need extra hardware to selectively disable themselves on the final iteration to avoid out-of-bounds memory accesses.

Again, there is a predictable subset of design points when the programmer obeys the unwritten rule: *An array's banking factor should divide the array size.* Figure 4c highlights the *predictable points* that follow this rule. The performance reliably improves with increasing parallelism and the area cost scales proportionally.

2.2 Enforcing the Unwritten Rules

The underlying problem in each of these sources of unpredictability is that the traditional design HLS tools prioritizes automation over programmer control. While automation can seem convenient, mapping heuristics give rise to implicit rules that, when violated, silently produce bad hardware instead of rejecting the program.

This paper instead prioritizes the predictability of hardware generation and making architectural decisions obvious in the source code. HLS tools *already* contain such a predictable and understandable language. By modeling resource constraints, we can separate out this well-behaved subset from the unpredictable parts. Figure 1 shows how our checker augments a traditional HLS toolchain by lifting hidden compiler reasoning into the source code and rejecting potentially unpredictable programs.

The challenge, however, is that the "unwritten rules" of HLS are never explicitly encoded anywhere—they arise implicitly from non-local interactions between program structure, hints, and heuristics. A naïve syntactic enforcement strategy would be too conservative—it would struggle to allow flexible, fine-grained sharing of hardware resources.

We design a type system that models the constraints of hardware implementation to enforce these constraints in a composable, formal way. Our type system addresses *target-independent* issues—it prevents problems that would occur even on an arbitrarily large FPGA. We do not attempt to rule out resource exhaustion problems because they would tie programs to specific target devices. We see that kind of quantitative resource reasoning as important future work.

3 The Dahlia Language

Dahlia's type system enforces a safety property: that the number of simultaneous reads and writes to a given memory bank may not exceed the number of ports. While traditional HLS tools enforce this requirement with scheduling heuristics, Dahlia enforces it at the source level using types.

The key ideas in Dahlia are (1) using substructural typing to reason about consumable hardware resources and (2) expressing time ordering in the language to reason about when resources are available. This section describes these two core features (Sections 3.1 and 3.2) and then shows how Dahlia builds on them to yield a language that is flexible enough to express real programs (Sections 3.3–3.6).

3.1 Affine Memory Types

The foundation of Dahlia's type system is its reasoning about memories. The problem in Section 2.1's example is conflicting simultaneous accesses to the design's memories. The number of reads and writes supported by a memory per cycle is limited by the number of ports in the memory. HLS tools automatically detect potential read/write conflicts and schedule accesses across clock cycles to avoid errors. Dahlia instead makes this reasoning about conflicts explicit by enforcing an affine restriction on memories.

Memories are defined by giving their type and size:

```
let A: float[10];
```

The type of `A` is `mem float[10]` which is a memory with 10 floating point elements and a single read/write port. Each Dahlia memory corresponds to an on-chip BRAM in the FPGA. A memory resembles a C or Java array: programs can read and mutate its contents by subscripting, as in `A[5] := 4.2`. Because they represent static physical resources in the generated hardware, memory types differ from plain value types like `float` by preventing duplication and aliasing:

```
let x = A[0];  // OK: x is a float.
let B = A;     // Error: cannot copy memories.
```

The affine restriction on memories disallows reads and writes to a memory at the same time:

```
let x = A[0];  // OK
A[1] := 1      // Error: Previous read consumed A.
```

While typechecking A, the Dahlia compiler removes A from the typing context. Subsequent uses of A are errors, with one exception: identical reads to the same memory location are allowed. This program is valid, for example:

```
let x = A[0];
let y = A[0];  // OK: Reading the same address.
```

The type system uses access capabilities to check reads and writes [19, 22]. A read expression such as A[0] acquires a *non-affine read capability* for index 0 in the current scope, which permits unlimited reads to the same location but prevents the acquisition of other capabilities for A. The generated hardware reads once from A and distributes the result to both variables x and y, as in this equivalent code:

```
let tmp = A[0]; let x = tmp; let y = tmp;
```

However, memory writes use *affine write capabilities*, which are use-once resources: multiple simultaneous writes to the same memory location remain illegal.

3.2 Ordered and Unordered Composition

A key HLS optimization is parallelizing execution of independent code. This optimization lets HLS compilers parallelize and reorder dependency-free statements connected by ; when the hardware constraints allow it—critically, when they do not need to access the same memory banks.

Dahlia makes these parallelism opportunities explicit by distinguishing between *ordered* and *unordered* composition. The C-style ; connector is unordered: the compiler is free to reorder and parallelize the statements on either side while respecting their data dependencies. A second connector, ---, is ordered: in A --- B, statement A must execute before B.

Dahlia prevents resource conflicts in unordered composition but allows two statements in ordered composition to use the same resources. For example, Dahlia accepts this program that would be illegal when joined by the ; connector:

```
let x = A[0]
---
A[1] := 1
```

In typechecking, ordered composition *restores* the affine resources that were consumed in the first command before checking the second command. The capabilities for all memories are discarded, and the program can acquire fresh capabilities to read and write any memory.

Together, ordered and unordered composition can express complex concurrent designs:

```
let A: float[10];  let B: float[10];
{
  let x = A[0] + 1
  ---
  B[1] := A[1] + x  // OK
};
let y = B[0];  // Error: B already consumed.
```

The statements composed with --- are ordered with each other but *unordered* with the last line. The read therefore must not conflict with either of the first two statements.

Logical time. From the programmer's perspective, a chain of ordered computations executes over a series of *logical time steps*. Logical time in Dahlia does not directly reflect physical time (i.e., clock cycles). Instead, the HLS backend is responsible for allocating cycles to logical time steps in a way that preserves the ordering of memory accesses. For example, a long logical time step containing an integer division might require multiple clock cycles to complete, and the compiler may optimize away unneeded time steps that do not separate memory accesses. Regardless of optimizations, however, a well-typed Dahlia program requires at least enough ordered composition to ensure that memory accesses do not conflict.

Local variables as wires & registers. Local variables, defined using the let construct, do not share the affine restrictions of memories. Programs can freely read and write to local variables without restriction, and unordered composition respects the dependencies induced by local variables:

```
let x = 0;  x := x + 1;  let y = x;  // All OK
```

In hardware, local variables manifest as wires or registers. The choice depends on the allocation of physical clock cycles: values that persist across clock cycles require registers. Consider this example consisting of two logical time steps:

```
let x = A[0] + 1  ---  B[0] := A[1] + x
```

The compiler must implement the two logical time steps in different clock cycles, so it must use a register to hold x. In the absence of optimizations, registers appear whenever a variable's live range crosses a logical time step boundary. Therefore, programmers can minimize the use of registers by reducing the live ranges of variables or by reducing the amount of sequential composition.

3.3 Memory Banking

As Section 2.1 details, HLS tools can *bank* memories into disjoint components to allow parallel access. Dahlia memory declarations support bank annotations:

```
let A: float[8 bank 4];
```

In a memory type mem t[n bank m], the banking factor m must evenly divide the size n to yield equally-sized banks. HLS tools, in contrast, allow uneven banking and silently insert additional hardware to account for it (see Section 2.1).

Affine restrictions for banks. Dahlia tracks an affine resource for each memory bank. To physically address a bank, the syntax $M\{b\}[i]$ denotes the ith element of M's bth bank. This program is legal, for example:

```
let A: float[10 bank 2];
```

```
A{0}[0] := 1;
A{1}[0] := 2;  // OK: Accessing a different bank.
```

To enforce its safety property, Dahlia tracks an affine resource for each memory bank. Dahlia also needs to conservatively approximate the banks consumed by every memory access expression. Dahlia supports logical indexing into banked arrays using the syntax $M[n]$ for literals n. For example, `A[1]` is equivalent to `A{1}[0]` above. Because the index is static, the type checker can automatically deduce the bank and offset.

Multi-ported memories. Dahlia also supports reasoning about multi-ported memories:

```
let A: float{2}[10];
```

which declares a memory where every bank gets two two read/write ports. Dahlia extends its affine reasoning to allow each bank to provide up to k resources where k is the number of ports in a memory. This allows multi-ported memories to provide multiple read/write capabilities in logical timestep. For example, Dahlia accepts the following program:

```
let A: float{2}[10];
let x = A[0];
A[1] := x + 1;
```

Dahlia does not guarantee data-race freedom in presence of multiported memories. Programs are free to write to and read from the same memory location in the same logical timestep and should expect the semantics guranteed by the underlying memory technology. A more precise type system/static analysis can be built on top of Dahlia to provide data-race freedom.

Multi-dimensional banking. Banking generalizes to multi-dimensional arrays. Every dimension can have an independent banking factor. This two-dimensional memory has two banks in each dimension, for a total of $2 \times 2 = 4$ banks:

```
let M: float[4 bank 2][4 bank 2];
```

0	1	0	1
2	3	2	3
0	1	0	1
2	3	2	3

The physical and logical memory access syntax similarly generalizes to multiple dimensions. For example, `M{3}[0]` represents the element logically located at `M[1][1]`.

3.4 Loops and Unrolling

Fine-grained parallelism is an essential optimization in hardware accelerator design. Accelerator designers duplicate a block of logic to trade off area for performance: n copies of the same logic consume n times as much area while offering a theoretical n-way speedup. Dahlia syntactically separates out parallelizable *doall* `for` loops, which must not have any cross-iteration dependencies, and sequential `while`, which may have dependencies but are not parallelizable. Programmers can mark `for` loops with an `unroll` factor to duplicate the loop body logic and run it in parallel:

```
for (let i = 0..10) unroll 2 { f(i) }
```

This loop is equivalent to a sequential one that iterates half as many times and composes two copies of the body in parallel:

```
for (let i = 0..5) {  f((2*i) + 0);  f((2*i) + 1)  }
```

The doall restriction is important because it allows the compiler to run the two copies of the loop body in parallel using unordered composition. In traditional HLS tools, a loop unrolling annotation such as `#pragma HLS unroll` is always allowed—even when the loop body makes parallelization difficult or impossible. The toolchain will replicate the loop body and rely on complex analysis and resource scheduling to optimize the unrolled loop body as well as it can.

Resource conflicts in unrolled loops are errors. For example, this unrolled loop is illegal because it accesses an unbanked array in parallel:

```
let A: float[10];
for (let i = 0..10) unroll 2 {
  A[i] := compute(i)  // Error: Insufficient banks.
}
```

Unrolled memory accesses. To type check memory accesses within unrolled loops, Dahlia uses special *index types* for loop iterators. Index types generalize integers to encode information about loop unrolling. In this example:

```
for (let i = 0..8) unroll 4 { A[i] }
```

The iterator `i` gets the type `idx{0..4}`, indicating that accessing an array at `i` will consume banks 0, 1, 2, and 3. Type checking a memory access with `i` consumes all banks indicated by its index type.

Unrolling and ordered composition. Loop unrolling has a subtle interaction with ordered composition. In a loop body containing `---`, like this:

```
let A: float[10 bank 2];
for (let i = 0..10) unroll 2 {
  let x = A[i]
  ---
  f(x, A[0])
}
```

A naive interpretation would use parallel composition to join the loop bodies at the top level:

```
for (let i = 0..5) {
  { let x0 = A[2*i]      ---  f(x0, A[0]) };
  { let x1 = A[2*i + 1]  ---  f(x1, A[0]) }  }
```

However, this interpretation is too restrictive. It requires *all* time steps in each loop body to avoid conflicts with all other time steps. This example would be illegal because the access to `A[i]` in the first time step may conflict with the access to `A[0]` in the second time step. Instead, Dahlia reasons about unrolled loops *in lockstep* by parallelizing *within* each logical time step. The loop above is equivalent to:

```
for (let i = 0..5) {
```

```
  { let x0 = A[2*i]; let x1 = A[2*i + 1] }
  ---
  { f(x0, A[0]);     f(x1, A[0]) } }
```

The lockstep semantics permits this unrolling because conflicts need only be avoided between unrolled copies of the same logical time step. HLS tools must enforce a similar restriction, but leave the choice to black-box heuristics.

Nested unrolling. In nested loops, unrolled iterators can separately access dimensions of a multi-dimensional array. Nested loops also interact with Dahlia's read and write capabilities. In this program:

```
let A: float[8 bank 4][10 bank 5];
for (let i = 0..8) {
  for (let j = 0..10) unroll 5 {
    let x = A[i][0]
    ---
    A[i][0] := j;  // Error: Insufficient write
        capabilities.
  } }
```

The read to array $A[i][0]$ can be proved to be safe because after desugaring, the reads turn into:

```
let x0 = A[i][0];  let x1 = A[i][0] ...
```

This is safe because the first read acquires a *read capability* for indices `i` and `0`, causing the subsequent copies to be safe. Architecturally, it corresponds to the single read being *fanned out* to each parallel PE.

However, the write desugars to:

```
A[i][0] := j; A[i][0] := j + 1 ...
```

which causes a write conflict in the hardware.

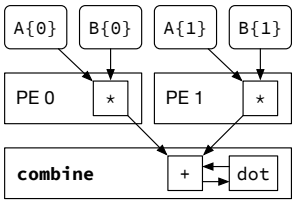
3.5 Combine Blocks for Reduction

While Dahlia's `for` loops prevent cross-iteration dependencies, accelerators often need to reduce the results of parallel loop iterations. In traditional HLS, loops can freely include dependent operations, as in this dot product:

```
for (let i = 0..10) unroll 2 { dot += A[i] * B[i]; }
```

However, the `+=` update silently introduces a dependency between every iteration. HLS tools heuristically analyze loops to extract and serialize dependent portions. In Dahlia, programmers explicitly distinguish the non-parallelizable reduction components of `for` loops. Each `for` can have an optional `combine` block that contains sequential code to run after each unrolled iteration group of the main loop body. For example, this loop is legal and generates the hardware:

```
for (let i = 0..10)
unroll 2 {
  let v = A[i] * B[i];
} combine {
  dot += v;
}
```



There are two copies of the loop body that run in parallel and feed into a single reduction tree for the `combine` block.

The type checker gives special treatment to variables like `v` that are defined in `for` bodies and used in `combine` blocks. In the context of the `combine` block, `v` is a *combine register*, which is a tuple containing all values produced for `v` in the unrolled loop bodies. Dahlia defines a class of functions called *reducers* that take a combine register and return a single value (similar to a functional fold). Dahlia defines `+=`, `-=`, `*=`, `/=` as built-in reducers with infix syntax.

3.6 Memory Views for Flexible Iteration

To predictably generate hardware for parallel accesses, Dahlia statically calculates banks accessed by each PE and guarantees that they are distinct. Figure 5a shows the kind of hardware generated by this restriction—each PE is directly connected to a bank.

To enforce this hardware generation, Dahlia only allows simple indexing expressions like `A[i]` and `A[4]`, and rejects arbitrary index calculations like `A[2*i]`. General indexing expressions can require complex indirection hardware to allow any PE to access any memory bank. An access like `A[i*i]`, for example, makes it difficult to deduce which bank it would read on which iteration. For simple expressions like `A[j+8]`, however, the bank stride pattern is clear. Traditional HLS tools make a best-effort attempt to deduce access patterns, but subtle changes in the code can unpredictable prevent the analysis and generate bad hardware.

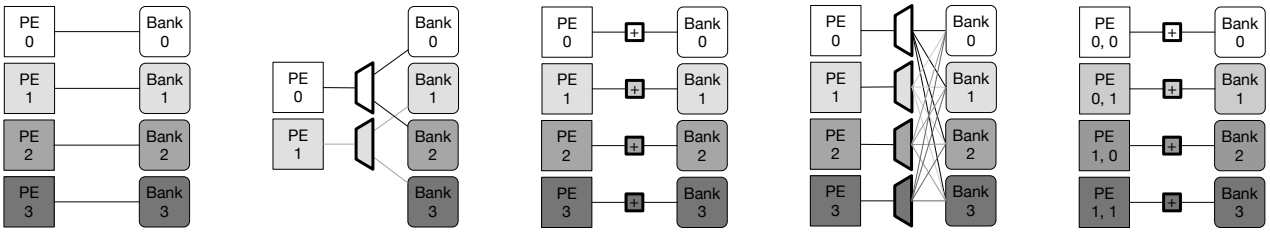
Dahlia programmers use *memory views* to convince the Dahlia compiler that a parallel access will be predictable. They key idea is to offer different logical arrangements of the same underlying physical memory. A view provides a different memory type for a given input memory to allow a different pattern of parallel access. Each view comes with a hardware cost. Dahlia compiles memory views into direct memory accesses before emitting HLS C++ code.

The rest of this section describes Dahlia's memory views and explains the hardware cost for each.

Shrink. To directly connect PEs to memory banks, Dahlia requires the unrolling factor to match the banking factor. To allow lower unrolling factors, Dahlia provides *shrink views*, which reduce the banking factors of an underlying memory by an integer factor. For example:

```
let A: float[8 bank 4];
view sh = shrink A[by 2];
for (let i = 0..8) unroll 2 {
  sh[i]; // OK: sh has 2 banks.
}
```

Dahlia allows `sh[i]` here because each PE will access a distinct set of banks. The first PE accesses banks 0 and 2 while the second PE accesses banks 1 and 3. The hardware cost of a shrink view, as Figure 5b illustrates, is the additional multiplexing to select the right bank on every iteration.



(a) No view. **(b)** Shrink view. **(c)** Suffix view. **(d)** Shift view. **(e)** Split view.

Figure 5. Hardware schematics for each kind of memory view. Heavy outlines indicate added hardware cost.

Suffix and prefix. A second kind of view lets programs create small slices of a larger memory. Dahlia divides the problem into suffixing (taking the last n elements) and prefixing (taking the first n elements). Dahlia also distinguishes between suffixes that it can implement efficiently and costlier ones. An efficient *aligned* suffix view uses this syntax:

```
view v = suffix M[by k * e];
```

where `v` starts at element $k \times e$ of the memory `M`. Critically, k must be the banking factor of `M`. Aligning the suffix to the banking factor ensures that Dahlia can still directly connect PEs to banks, as Figure 5c illustrates. This works because Dahlia statically knows the bank of `M` for every index in `v`: namely, `v` has the same banking factor as `M` and the bank for every element in `v` is equal to the bank in the underlying memory. For example, generating suffixes in a loop results in this pattern, where the digits in each cell are bank numbers and the heavy outline indicates the view:

```
let A: float[6 bank 2];
for (let i = 0..4) {
  view s = suffix A[by 2 * i];
  s[j]; }
```

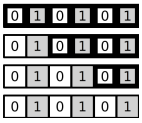


Figure 5c shows that the hardware cost of using an aligned suffix view consists only of an additional hardware cost is an *address adapter* that offsets the address into a bank. There is no additional hardware required to select the right bank.

Prefix views have no hardware cost: they change neither the index nor the bank number for any access. They statically restrict accesses by producing a memory with a smaller size.

Shift. Shifted suffixes are like standard suffixes but allow unrestricted offset expressions. Dahlia disallows parallelizing the context that creates *shift views*. This is because proving that two arbitrary views do not overlap is undecidable in general. The hardware cost for a shifted view (Figure 5d) is much higher because the HLS compiler has to assume that any PE might use any memory bank and generates hardware to connect every PE to every bank. Even in this worst-case scenario, Dahlia can reason about the disjointness of bank accesses for a given suffix. This loop is legal, for example:

```
let A: float[12 bank 4];
for (let i = 0..3) {
  view r = shift A[by i*i];
  for (let j = 0..4) unroll 4 {
    r[j]; } }
```

And the access in the loop is equivalent to `A[i*i + j]`.

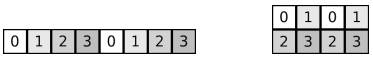
Split. Some nested iteration patterns can be parallelized at two levels: globally, over an entire array, and locally, over a smaller window. This pattern arises in blocked computations, such as this dot product loop in C++:

```
float A[8], B[8], sum = 0.0;
for (int i = 0; i < 4; i++) {
  for (int j = 0; j < 2; j++) {
    int v = A[2*i + j] * B[2*i + j];
    sum += v; } }
```

Both the inner loop and the outer loop represent opportunities for parallelization. To allow unrolling each loop, Dahlia requires banking corresponding to the unrolling factor.

To allow unrolling at both the global and local level, Dahlia has to guarantee that each window is disjoint and each element within the window touches distinct elements.

Split views allow for this reasoning. The key idea is to create a logically *more* dimensions than the physical memory and reusing Dahlia's reasoning for multidimensional memories to prove safety for such accesses. A *split view* transforms the one-dimensional memory (left) into a two-dimensional memory (right).



Each row contains logical chunks for the computation. Using these split-view declarations:

```
view split_A = split A[by 2];
view split_B = split B[by 2];
```

Each view has type `mem float[4 bank 2][2 bank 2]`. The above example can now unroll both loops:

```
for (let i = 0..4) unroll 2 {
  for (let j = 0..2) unroll 2 {
    let v = split_A[i][j] * split_B[i][j];
  } combine {
    sum += v; } }
```

As Figure 5e illustrates, split views have similar cost to aligned suffix views: they require no bank indirection hardware because the bank index is always known statically. They

$$
\begin{array}{l}
x \in \text{variables} \qquad a \in \text{memories} \\
n \in \text{numbers} \qquad b ::= \textbf{true} \mid \textbf{false} \qquad v ::= n \mid b \\
e ::= v \mid \textbf{bop}\ e_1\ e_2 \mid x \mid a[e] \\
c ::= e \mid \textbf{let}\ x = e \mid c_1 \text{ —— } c_2 \mid c_1\ ;\ c_2 \mid \textbf{if}\ e\ c_1\ c_2 \mid \\
\qquad \textbf{while}\ e\ c \mid x := e \mid a[e_1] := e_2 \mid \textbf{skip} \\
\tau ::= \textbf{bit}\langle n \rangle \mid \textbf{float} \mid \textbf{bool} \mid \textbf{mem}\ \tau[n_1\ \textbf{bank}\ n_2]
\end{array}
$$

Figure 6. Abstract syntax for the Filament core language.

require an address adapter to compute the address within the bank from the separate coordinates.

4 Formalism

This section formalizes the time-sensitive affine type system that underlies Dahlia in a core language, Filament. We give both a large-step semantics, which is more intelligible, and a small-step semantics, which enables a soundness proof.

4.1 Syntax

Figure 6 lists the grammar for Filament. Filament statements c resemble a typical imperative language: there are expressions, variable declarations, conditions, and simple sequential iteration via `while`. Filament has ordered composition c_1 —— c_2 and unordered composition $c_1\ ;\ c_2$. It separates memories a and variables x into separate syntactic categories. Filament programs can only declare the latter: a program runs with a fixed set of available memories.

4.2 Large-Step Semantics

Filament's large-step operational semantics is a *checked semantics* that enforces Dahlia's safety condition by explicitly tracking and getting stuck when it would otherwise require two conflicting accesses. Our type system (Section 4.3) aims to rule out these conflicts.

The semantics uses an environment σ mapping variable and memory names to values, which may be primitive values or memories, which in turn map indices to primitive values. A second context, ρ, is the set of the memories that the program has accessed. ρ starts empty and accumulates memories as the program reads and writes them.

The operational semantics consists of an expression judgment $\sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, v$ and a command judgment $\sigma_1, \rho_1, c \Downarrow \sigma_2, \rho_2$. We describe some relevant rules here, and the supplementary material lists the full semantics and proof [2].

Memory accesses. Memories in Filament are mutable stores of values. Banked memories in Dahlia can be built up using these simpler memories. The rule for a memory read expression $a[n]$ requires that a not already be present in ρ, which would indicate that the memory was previously consumed:

$$\frac{a \notin \rho_1 \qquad \sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, n \qquad \sigma_2(a)(n) = v}{\sigma_1, \rho_1, a[e] \Downarrow \sigma_2, \rho_2 \cup \{a\}, v}$$

Composition. Unordered composition accumulates the resource demands of two commands by threading ρ through:

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_2, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1\ ;\ c_2 \Downarrow \sigma_3, \rho_3}$$

If both commands read or write the same memory, they will conflict in ρ. Ordered composition runs each command in the same initial ρ environment and merges the resulting ρ:

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_1, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \text{ —— } c_2 \Downarrow \sigma_3, \rho_2 \cup \rho_3}$$

4.3 Type System

The typing judgments have the form $\Gamma_1, \Delta_1 \vdash c \dashv \Gamma_2, \Delta_2$ and $\Gamma, \Delta_1 \vdash e : \tau \dashv \Delta_2$. Γ is a standard typing context for variables and Δ is the affine context for memories.

Affine memory accesses. Memories are affine resources. The rules for reads and writes check the type of the index in Γ and remove the memory from Δ:

$$\frac{\Gamma, \Delta_1 \vdash e_1 : \textbf{bit}\langle n \rangle \dashv \Delta_2 \qquad \Delta_2 = \Delta_3 \cup \{a \mapsto \textbf{mem}\ \tau[n_1]\}}{\Gamma, \Delta_1 \vdash a[e] : \tau \dashv \Delta_3}$$

Composition. The unordered composition rule checks the first statement in the initial contexts and uses the resulting contexts to check the second statement:

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_2 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1\ ;\ c_2 \dashv \Gamma_3, \Delta_3}$$

Ordered composition checks both commands with the same resources:

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_1 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1 \text{ —— } c_2 \dashv \Gamma_3, \Delta_2 \cap \Delta_3}$$

The rule merges the resulting Δ contexts with set intersection to yield the resources not consumed by either statement.

4.4 Small-Step Semantics

We also define a small-step operational semantics for Filament upon which we build a proof of soundness. The semantics uses the same environment σ and memory context ρ as the big-step semantics.

The semantics consists of judgments $\sigma_1, \rho_1, e \longrightarrow \sigma_2, \rho_2, e'$ and $\sigma_1, \rho_1, c \longrightarrow \sigma_2, \rho_2, c'$. We state that the small-step semantics is equivalent to the big-step semantics, though several big-step judgments are represented by multiple small-step judgments each. Additionally, modeling sequential composition c_1 —— c_2 necessitates an intermediate command form $c_1 \overset{\rho}{\sim} c_2$ to correctly thread ρ to c_1 and c_2.

4.5 Soundness Theorem

We state a soundness theorem for Filament's type system with respect to its checked small-step operational semantics.

Theorem. *If* $\emptyset, \Delta^* \vdash c \dashv \Gamma_2, \Delta_2$, *then* $\emptyset, \emptyset, c \overset{*}{\longrightarrow} \sigma, \rho, \textbf{skip}$ *or* c *diverges.*

where Δ^* is the initial affine context of memories available to a program. The theorem implies that the type system rules out stuckness due to memory conflicts in ρ. We prove this using progress and preservation lemmas:

Lemma 1 (Progress). *If* $\Gamma, \Delta \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\Gamma, \Delta \sim \sigma, \rho$, *then* $\sigma, \rho, c \longrightarrow \sigma', \rho', c'$ *or* $c =$ **skip**.

Lemma 2 (Preservation). *If* $\Gamma, \Delta \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\Gamma, \Delta \sim \sigma, \rho$, *and* $\sigma, \rho, c \longrightarrow \sigma', \rho', c'$, *then* $\Gamma', \Delta' \vdash c' \dashv \Gamma_2', \Delta_2'$ *and* $\Gamma', \Delta' \sim \sigma', \rho'$.

In these lemmas, $\Gamma, \Delta \sim \sigma, \rho$ is a well-formedness judgment stating that all variables in Γ are in σ and all memories in Δ are not in ρ. We prove the lemmas by induction on the small-step relation in the supplementary material.

5 Evaluation

Our evaluation measures whether Dahlia's restrictions can improve predictability without sacrificing too much sheer performance. We conduct two experiments: (1) We perform an exhaustive design space exploration for one kernel to determine how well the restricted design points compare to the much larger unrestricted parameter space. (2) We port the MachSuite benchmarks [46] and, where Dahlia yields a meaningful design space, perform a parameter sweep.

5.1 Implementation and Experimental Setup

We implemented a Dahlia compiler in 5200 LoC of Scala. The compiler checks Dahlia programs and generates C++ code using Xilinx Vivado HLS's `#pragma` directives [55]. We execute benchmarks on AWS F1 instances [1] with 8 vCPUs, 122 GB of main memory, and a Xilinx UltraScale+ VU9P. We use the SDAccel development environment [54] and synthesize the benchmarks with a target clock period of 250 MHz.

5.2 Case Study: Unrestricted DSE vs. Dahlia

In this section, we conduct an exhaustive design-space exploration (DSE) of a single benchmark as a case study. Without Dahlia, the HLS design space is extremely large—we study how the smaller Dahlia-restricted design space compares.

We select a blocked matrix multiplication kernel (gemm-blocked from MachSuite) for its large but tractable design space. The kernel has 3 two-dimensional arrays (two operands and the output product) and 5 nested loops, of which the inner 3 are parallelizable. We define parameters for the 6 banking factors (two dimensions for each memory) and 3 unrolling factors. (A full code listing appears in the supplementary material [2].) We explore a design space with banking factors of 1–4 and unrolling factors of 1, 2, 4, 6, and 8. This design space consists of 32,000 distinct configurations.

We exhaustively evaluated the entire design space using Vivado HLS's estimation mode, which required a total of 2,666 compute hours. We identify Pareto-optimal configurations according to their estimated cycle latency and number of lookup tables (LUTs), flip flops (FFs), block RAMs (BRAMs), and arithmetic units (DSPs).

Dahlia accepts 354 configurations, or about 1.1% of the unrestricted design space. The smaller space is only useful if it includes *useful* design points—a broad range of Pareto-optimal configurations. Figures 7a and 7c show the Pareto-optimal points and the subset that Dahlia accepts, respectively. (Pareto optimality is determined using all objectives, but the plot shows only two: LUTs and latency.) The Dahlia-accepted points lie primarily on the Pareto frontier. The optimal points that Dahlia rejects expend a large number of LUTs to reduce BRAM consumption.

5.3 Dahlia-Directed DSE & Programmability

We port benchmarks from an HLS benchmark suite, MachSuite [46], to study Dahlia's flexibility. Of the 19 MachSuite benchmarks, one contains a correctness bug and two fail to synthesize correctly in Vivado, indicating a bug in the tools. We successfully ported all 16 of the remaining benchmarks.

From these, we select 3 benchmarks that exhibit the kind of fine-grained, loop-level parallelism that Dahlia targets as case studies: sencil2d, md-knn, and md-grid. As the previous section illustrates, an unrestricted DSE is intractable for even modestly sized benchmarks, so we instead measure the breadth and performance of the much smaller space of configurations that Dahlia accepts. For each benchmark, we find all optimization parameters available in the Dahlia port and define a search space for each. The Dahlia type checker rejects some design points, and we measure the remaining space. We use Vivado HLS's estimation mode to measure the resource counts and estimated latency for each accepted point. Figure 8 depicts the Pareto-optimal points in each space. In each plot, we also highlight the effect a single parameter has on the results.

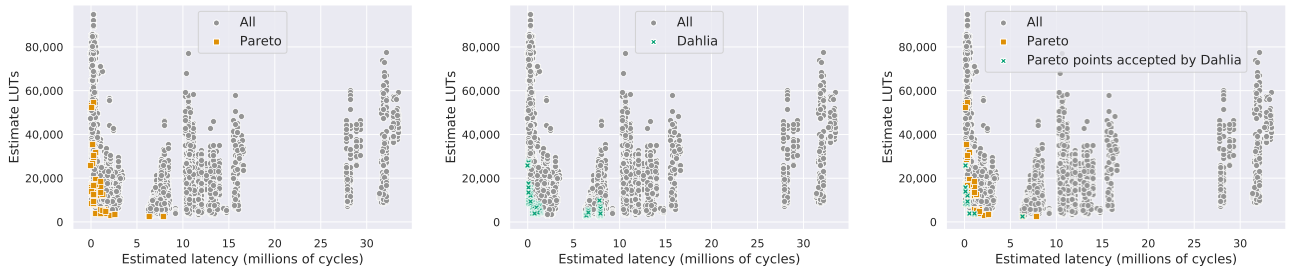
The rest of this section reports quantitatively on each benchmark's design space and reports qualitatively on the programming experience during the port from C to Dahlia.

stencil2d. MachSuite's stencil2d is a filter operation with four nested loops. The outer loops scan over the input matrix and the inner loops apply a 3×3 filter. Our Dahlia port unrolls the inner two loops and banks both input memories. We use unrolling factors from 1 to 3 and bank each dimension of the input array by factors 1 to 6. The resulting design space has 2,916 points. Dahlia accepts 18 of these points (0.6%).

Section 5.3 shows the Pareto-optimal Dahlia-accepted points. The figure uses color to show the unrolling factor for the innermost loop. This unrolling factor has a large effect on the design's performance, while banking factors and the other loop explain the rest of the variation.

The original C code uses single-dimensional arrays and uses index arithmetic to treat them as matrices:

```
for (r=0; r<row_size-2; r++) {
  for (c=0; c<col_size-2; c++) {
```



(a) Pareto optimal points **(b)** Points accepted by Dahlia **(c)** Pareto points accepted by Dahlia

Figure 7. Results from exhaustive design space exploration for gemm-blocked.

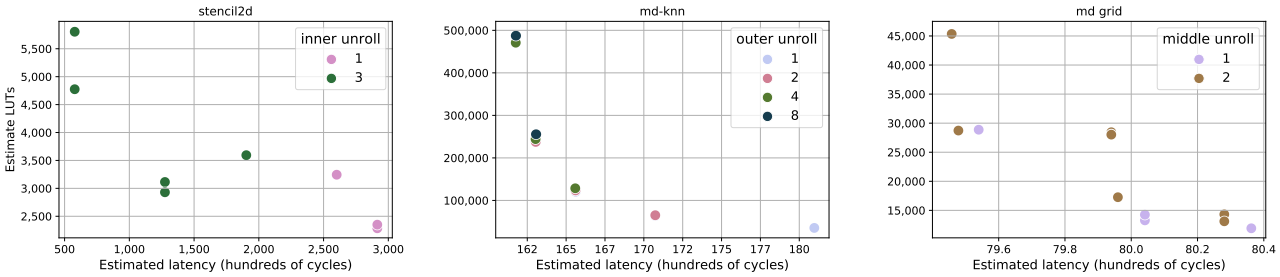


Figure 8. The design spaces for three MachSuite benchmarks. Each uses a color to highlight one design parameter.

```
    for (k1=0; k1<3; k1++) {
      for (k2=0; k2<3; k2++) {
        mul = filter[k1*3 + k2] *
              orig[(r+k1)*col_size + c+k2];
```

In the Dahlia port, we must use proper two-dimensional arrays because the compiler rejects arbitrary indexing expressions. Using views, programmers can decouple the storage format from the iteration pattern. To express the accesses to the input matrix `orig`, we create a shifted suffix view (Section 3.6) for the current window:

```
for (let r = 0..126) {
  for (let c = 0..62) {
    view window = shift orig[by r][by c];
    for (let k1 = 0..3) unroll 3 {
      for (let k2 = 0..3) unroll 3 {
        let mul = filter[k1][k2] * window[k1][k2];
```

The view makes the code's logic more obvious while allowing the Dahlia type checker to allow unrolling on the inner two loops. It also clarifies why parallelizing the outer loops would be undesirable: the parallel views would require overlapping regions of the input array, introducing a bank conflict.

md-knn. The md-knn benchmark implements an n-body molecular dynamics simulation with a k-nearest neighbors kernel. The MachSuite implementation uses data-dependent loads in its main loop, which naïvely seems to prevent parallelization. In our Dahlia port, however, we hoist this serial section into a separate loop that runs before the main, parallelizable computation. Dahlia's type system helped guide the programmer toward a version of the benchmark where the benefits from parallelization are clear.

For each of the program's four memories, we used banking factors from 1 to 4. We unrolled each of the two nested loops with factors from 1 to 8. The full space has 16,384 points, of which Dahlia accepts 525 (3%).

Section 5.3 shows the latency of the Pareto-optimal designs. The visualization omits a tight cluster of 189 outlier points (36% of the Pareto-optimal configurations) with much higher latency (~130, 000 cycles), and marginally better resource usage, to make it easier to see the trend. The color shows the unrolling factor of the outer loop. In this kernel, the dominant effect is the memory banking (not shown in the figure), which determines which cluster the results fall into. The unrolling factor is a second-order effect that expends LUTs to achieve a small increase in performance.

md-grid. Another algorithm for the same molecular dynamics problem, md-grid, uses a different strategy based on a 3D grid implemented with several 4-dimensional arrays. It calculates forces between neighboring grid cells. Of its 6 nested loops, the outer three are parallelizable. We use banking factors of 1 to 4 for each dimension of each array, and we try unrolling factors from 1 to 8 for both loops. The full space has 21,952 points, of which Dahlia accepts 81 (0.4%).

Section 5.3 again shows the Pareto-optimal design points. The *innermost* loop unrolling factor (not shown in the figure) determines which of three coarse regimes the design falls into. The color shows the *second* loop unrolling factor, which

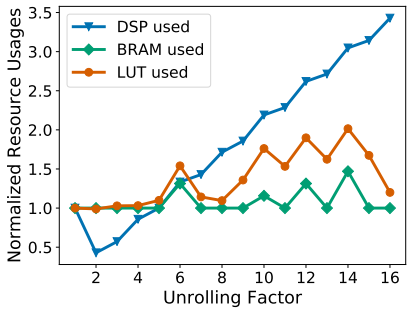


Figure 9. Resource utilization for gemm-ncubed in Spatial normalized to no unrolling.

determines a second-order area–latency trade-off within each regime. In both cases, unrolling has a predictable effect on the quality of the design: more unrolling attains better performance at the cost of area.

6 Related Work

Dahlia builds on a long history of work on safe systems programming. Substructural type systems are known to be a good fit for controlling system resources [7, 24, 51, 14, 36]. Dahlia's enforcement of exclusive memory access resembles work on race-free parallel programming using type and effect systems [8] or concurrent separation logic [39]. Safe parallelism on CPUs focuses on data races where concurrent reads and writes to a memory are unsynchronized. Conflicts in Dahlia are different: *any* simultaneous pair of accesses to the same *bank* is illegal. The distinction influences Dahlia's capability system and its memory views, which cope with the arrangement of arrays into parallel memory banks.

Dahlia takes inspiration from other approaches to improving the accelerator design process, including HDLs, HLS, DSLs, and other recent accelerator design languages.

Spatial. Spatial [32] is a language for designing accelerators that builds on *parallel patterns* [41], which are flexible hardware templates. Spatial adds some automation beyond traditional HLS: it infers a banking strategy given some parallel accesses. Like HLS, Spatial designs can be unpredictable. Figure 9 shows resource usage for the matrix multiplication kernel from Section 2 written in Spatial. (A full experimental setup appears in the supplementary material [2].) For unrolling factors that do not evenly divide the memory size, Spatial will sometimes infer a banking factor that is not equal to the unrolling factor. In these cases, the resource usage abruptly increases. A type system like Dahlia could help address these predictability pitfalls in Spatial.

Better HDLs. Modern hardware description languages [5, 35, 15, 4, 52, 30, 38] aim to address the shortcomings of Verilog and VHDL. These languages target register transfer level (RTL) design. Dahlia targets a different level of abstraction and a different use case: it uses an imperative programming model and focuses exclusively on computational accelerators. Dahlia is not a good language for implementing a CPU, for example. Its focus on acceleration requires the language and semantics to more closely resemble software languages.

Traditional HLS. Existing commercial [55, 29, 37, 9] and academic [44, 10, 40, 56] high-level synthesis (HLS) tools compile subsets of C, C++, OpenCL, or SystemC to RTL. While their powerful heuristics can be effective, when they fail, programmers have little insight into what went wrong or how to fix it [34]. Dahlia represents an alternative approach that prioritizes programmer control over black-box optimization.

Targeting hardware from DSLs. Compilers to FPGAs and ASICs exist for DSLs for image processing [26, 27, 42, 48] and machine learning [21, 49]. Dahlia is not a DSL: it is a general language for implementing accelerators. While DSLs offer advantages in productivity and compilation for individual application domains, they do not obviate the need for general languages to fill in the gaps between popular domains, to offer greater programmer control when appropriate, and to serve as a compilation target for multiple DSLs.

Accelerator design languages. Some recent languages also target general accelerator design. HeteroCL [33] uses a Halide-like [45] scheduling language to describe how to map algorithms onto HLS-like hardware optimizations, and T2S [47] similarly lets programs describe how generate a spatial implementation. Lime [3] extends Java to express target-independent streaming accelerators. CoRAM [13] is not a just a language; it extends FPGAs with a programmable memory interface that adapts memory accesses, akin to Dahlia's memory views. Dahlia's focus on predictability and type-driven design makes it unique, as far as we are aware.

7 Conclusion

Dahlia exposes predictability as a new design goal for HLS tools. Predictability comes at a cost—it can rule out design points that perform surprisingly well because of a subtle convergence of heuristics. We see these outliers as a worthy sacrifice in exchange for an intelligible programming model.

Future work should extend Dahlia with parametric polymorphism to express interdependencies between parameters. While Dahlia's design spaces are already smaller than in unrestricted HLS, polymorphism could help scale DSE to large designs comprising many interconnected subcomponents.

References

[1] Amazon Web Services. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[2] Anonymous for double blind review. Predictable Accelerator Design with Time-Sensitive Affine Types: Supplemental Material.

[3] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[4] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware Using

Haskell. In *Euromicro Conference on Digital System Design: Architectures, Methods and Tools*.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Design Automation Conference (DAC)*.

[6] Henry G. Baker. 1995. "Use-once" Variables and Linear Objects: Storage Management, Reflection and Multi-threading. *SIGPLAN Notices* 30, 1 (Jan. 1995), 45–52.

[7] J Bernardy, Mathieu Boespflug, Ryan Newton, Simon L. Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[8] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[9] Cadence. Stratus High-Level Synthesis. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[10] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[11] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. 2016. When Apache Spark meets FPGAs: a case study for next-generation DNA sequencing acceleration. In *IEEE International Conference on Cloud Computing (CloudCom)*.

[12] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: big data on little clients. In *International Symposium on Computer Architecture (ISCA)*.

[13] Eric S Chung, James C Hoe, and Ken Mai. 2011. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Field programmable gate arrays (FPGA)*.

[14] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*.

[15] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. 2017. A Pythonic approach for rapid hardware prototyping and instrumentation. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

[16] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. 2006. Platform-Based Behavior-Level and System-Level Synthesis. In *International SoC Conference*.

[17] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 30, 4 (April 2011), 473–491.

[18] J. Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Design Automation Conference (DAC)*.

[19] Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear regions are all you need. In *European Symposium on Programming (ESOP)*.

[20] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-scale DNN Processor for Real-time AI. In *International Symposium on Computer Architecture (ISCA)*.

[21] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. 2014. Hardware system synthesis from Domain-Specific Languages. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

[22] Colin S Gordon, Michael D Ernst, and Dan Grossman. 2013. Rely-guarantee references for refinement types over aliased mutable data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[23] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[24] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based Memory Management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[25] S Gupta, Renu Gupta, Nikil Dutt, and Alex Nicolau. 2004. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer.

[26] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics* 33, 4 (2014).

[27] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics* 35, 4 (2016).

[28] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Communications of the ACM (CACM)* 62, 2 (Jan. 2019), 48–60.

[29] Intel. Intel High Level Synthesis Compiler. https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html

[30] Jane Street. HardCaml: Register Transfer Level Hardware Design in OCaml. https://github.com/janestreet/hardcaml.

[31] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture (ISCA)*.

[32] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: a language and compiler for application accelerators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[33] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *International Symposium on Field-Programmable Gate Arrays*

(FPGA).

[34] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N Do, and Deming Chen. 2012. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering* (2012).

[35] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[36] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *High Integrity Language Technology (HILT)*.

[37] Mentor Graphics. Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.

[38] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Conference on Formal Methods and Models for Co-Design (MEMOCODE)*.

[39] Peter W. O'Hearn. 2007. Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science* 375 (April 2007), 271–307.

[40] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

[41] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating configurable hardware from parallel patterns. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 651–665.

[42] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)* (2017).

[43] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Y. Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *International Symposium on Computer Architecture (ISCA)*.

[44] Andrew R Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. 2008. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[45] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[46] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*.

[47] Hongbo Rong. Programmatic Control of a Compiler for Generating High-performance Spatial Hardware. arXiv preprint 1711.07606. https://arxiv.org/abs/1711.07606.

[48] Jeff Setter. Halide-to-Hardware. https://github.com/jeffsetter/Halide-to-Hardware.

[49] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[50] Stuart Sutherland, Don Mills, and Chris Spear. 2007. Gotcha Again: More Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know. In *Synopsys Users Group (SNUG) San Jose*. https://lcdm-eng.com/papers/snug07$_{\mathrm{V}}$erilog%20Gotchas%20Part2.pdf

[51] Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[52] Lenny Truong and Pat Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity.

[53] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A Genomics Co-processor Provides Up to 15,000X Acceleration on Long Read Assembly. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[54] Xilinx Inc. SDAccel: Enabling Hardware-Accelerated Software. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[55] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. UG902 (v2017.2) June 7, 2017. https://www.xilinx.com/support/documentation/sw$_{\mathrm{m}}$anuals/xilinx2017$_{2}$/ug902-vivado-high-level-synthesis.pdf.

[56] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*. 99–112.