Unifying HLS and RTL with Timeline Types

Jingtao Xia
UC Santa Barbara
USA

Ayana Alemayehu
MIT
USA

Sijie Kong
UC Santa Barbara
USA

Ben Hardekopf
UC Santa Barbara
USA

Rachit Nigam
MIT CSAIL
USA

Jonathan Balkind
UC Santa Barbara
USA

ABSTRACT

Hardware design today is divided between two paradigms: RTL offers precise control over timing and resources but demands substantial manual effort, while HLS provides automation at the cost of predictability and fine-grained control. We argue that this dichotomy is unnecessary. This paper proposes a unifying design approach for latency-sensitive and latency-abstract hardware based on *timeline types*. By extending Filament's timeline type system with scheduler-determined parameters, denoted by question marks, designers can selectively leave binding and scheduling decisions partially open to automation while retaining explicit control where needed. This enables a continuous spectrum between fully manual RTL design and fully automated HLS design within a single language framework, making it possible to combine fine-grained control with automation.

1 INTRODUCTION

Modern hardware design forces an uncomfortable choice. Register-transfer level (RTL) design offers precise control over timing, resource allocation, and microarchitecture, but demands substantial manual effort. High-level synthesis (HLS) automates allocation, binding, and scheduling, yet often yields designs whose performance and resource usage are difficult to predict or fine-tune. Designers who require both productivity and control are thus forced to switch between paradigms, or accept an unsatisfactory compromise.

We argue that this dichotomy can be resolved within a single design framework. Our starting point is Filament's *timeline type* system [7], which provides static, compositional guarantees of timing correctness by encoding the clock-cycle intervals during which a signal is available or required. Intuitively, the timing variables used by timeline types align closely with the scheduling variables manipulated by modern HLS schedulers, which are typically solved using systems of difference constraints (SDC) [2, 3, 9]. This correspondence suggests a natural path toward unification. We extend Filament's timeline types with *question marks*, allowing selected timing, binding, and resource decisions to be left unspecified using a question mark (?) and resolved automatically by a scheduler, while others remain explicitly controlled by the designer. In contrast to traditional HLS approaches that rely on external pragmas or tool-specific directives, our approach internalizes scheduling intent into the language itself, enabling fine-grained control and automation to coexist in a principled, compositional manner. From an HLS perspective, this amounts to making allocation, binding, and scheduling decisions explicitly and partially specifiable at the language level, rather than implicitly encoded through pragmas or left entirely to the tool.

In this paper, we present a position arguing that partial specification over timeline-based timing constraints provides a principled foundation for unifying RTL and HLS design.

2 TIMELINE TYPES WITH QUESTION MARKS

Filament's timeline types encode when signals are available using event-relative intervals (e.g., `['G, 'G+4]` means available from cycle `'G` to `'G+4`). In Filament, component instances are declared explicitly, and each use of an instance (an *invocation*) specifies a concrete start time. The type system ensures that invocations on the same instance do not overlap. We extend Filament to support partial specification of the three core HLS aspects: allocation, binding, and scheduling.

Allocation. In Filament, instances are declared individually. We generalize this with *pool* declarations, which specify a set of component instances available within a module (e.g., `pool M : Mult[32] * 2` declares two 32-bit multipliers). The pool size can also be left unspecified (`* ?`), allowing the solver to determine resource usage.

Binding. Binding selects which instance from a pool an operation uses. This can be fully unspecified (`M[?]`, HLS-style), leaving the choice to the solver, or fully specified (`M[0]`, HDL-style). *Named binding variables* (`M[?u]`) let multiple operations share the same instance without specifying which one.

Scheduling. An invocation combines a bound instance with a start time. In Filament, start times are concrete offsets from a global event (e.g., `<'G+3>` starts 3 cycles after event `'G`). We extend this so that timing can be fully unspecified (`<?>`, HLS-style), leaving all decisions to the solver, or fully specified with a concrete offset (HDL-style). For richer control, we provide: *named scheduling variables* (e.g., `?t`) let multiple invocations reference a shared symbolic time, expressing relative constraints (e.g., `<?t>` and `<?t+3>` must be 3 cycles apart); *ranges* bound the offset from either a global event (`<'G+[1..2]>`) or a named scheduling variable (`<?t+[0..2]>`).

3 EXAMPLE: DOT PRODUCT

Consider computing $\sum_{i=1}^{4} a_i \cdot b_i$ using two multipliers (latency 3) and two adders (latency 1). For simplicity, we assume a non-pipelined schedule and model only operator latency (no II), treating all functional units as single-issue. The following code illustrates the use of question marks for allocation, binding, and scheduling:

```
comp dot_product<'G>(
  a1: ['G,'G+4] 32, b1: ['G,'G+4] 32,
  a2: ['G,'G+4] 32, b2: ['G,'G+4] 32,
  a3: ['G,'G+4] 32, b3: ['G,'G+4] 32,
  a4: ['G,'G+4] 32, b4: ['G,'G+4] 32,
) -> (out: ['G+8,'G+9] 32) {
  pool M : Mult[32] * 2;  // two multipliers
  pool A : Add[32]  * 2;  // two adders
  // mul1,mul2 share unit [?u1]; mul2 starts 3 cycles
  ↪  after mul1
  mul1 := M[?u1]<?t_mul>(a1, b1);
  mul2 := M[?u1]<?t_mul+3>(a2, b2);
  // mul3,mul4 share unit [?u2]; mul4 has fixed timing
  mul3 := M[?u2]<?>(a3, b3);
  mul4 := M[?u2]<'G+3>(a4, b4);
  // Partial sums with timing constraints
  add1 := A[0]<?t_add>(mul1.out, mul2.out);
  add2 := A[?]<?t_add+[0..2]>(mul3.out, mul4.out);
  // result: fully automatic
  result := A[?]<?>(add1.out, add2.out);
  out = result.out;
}
```

This example demonstrates question marks across all three dimensions:

- **Allocation:** Pools M and A declare two multipliers and two adders available.
- **Binding:** Operations mul1/mul2 share one multiplier via [?u1]; mul3/mul4 share another via [?u2]. The adder for add1 is fixed to A[0]; others are left to the scheduler.
- **Scheduling:** Operations mul1 and mul2 are relatively constrained: mul2 starts 3 cycles after mul1 via <?t_mul+3>. Operation mul4 is fixed at 'G+3; mul3 is fully flexible. The adder add2 must start within 2 cycles of add1.

The result addition leaves both binding and timing fully unspecified, delegating all decisions to the scheduler. Notably, these constraints are expressed directly in the language rather than through external pragmas.

Constraint Generation. Question marks induce a constraint system over time, binding, and scheduling variables. Timeline types already encode timing relationships as difference constraints, which align naturally with the System of Difference Constraints (SDC) formulation used by HLS schedulers. Our extension adds flexibility: fully specified values fix constraints, fully unspecified values introduce free variables, and named variables create relationships between operators. In addition, we incorporate resource constraints that enforce non-overlap between invocations of the same functional unit, data-dependency constraints that preserve def–use ordering, and timing constraints on input availability and output production. The resulting constraint system can be solved to produce a fully specified schedule.

Scheduled Output. The output is fully-specified Filament code:

```
comp dot_product<'G>(...) -> (...) {
  mul1 := M[1]<'G>(a1, b1);
  mul2 := M[1]<'G+3>(a2, b2);
  mul3 := M[0]<'G>(a3, b3);
  mul4 := M[0]<'G+3>(a4, b4);
  add1 := A[0]<'G+6>(mul1.out, mul2.out);
  add2 := A[1]<'G+6>(mul3.out, mul4.out);
  result := A[0]<'G+7>(add1.out, add2.out);
  ...
}
```

The scheduled output is a standard Filament module with concrete timeline types, enabling it to be composed with hand-written HDL in a type-safe manner.

4 ANOTHER VIEW: FROM XLS

While we present our approach starting from Filament, an HDL with timeline types, the same unification idea can be approached from the HLS side. Ongoing work explores adding timeline type annotations to XLS [4], an HLS framework from Google. By exposing scheduling decisions through timeline types, designers gain visibility and control over what are traditionally opaque, automatic decisions. This suggests that timeline-type-based partial specification is not tied to one language or direction, but can serve as a general interface between manual and automated hardware design.

5 DISCUSSION

Incremental Design and Debugging. Partial specification supports incremental workflows. Designers can start with fully automatic scheduling (all question marks) to explore the design space, then progressively fix decisions as they tune for performance or debug timing issues. Conversely, starting from a fully specified HDL design, they can selectively relax constraints to discover optimization opportunities.

Comparison to Pragmas. Traditional HLS tools use pragmas or directives to guide scheduling, but these are external hints disconnected from the program's semantics. Moreover, pragmas are often advisory: tools may silently ignore them if constraints cannot be satisfied, making failures difficult to diagnose [1, 5, 6]. Our approach integrates scheduling constraints into the type system, making them first-class and checkable — the solver either satisfies them or reports a clear failure.

Compositionality. A key advantage over traditional HLS is that scheduled modules become standard Filament components with concrete timeline types. When an HLS-generated module is used in a larger hand-written design, the type system verifies that the caller respects the module's timing requirements [8]. This compositional guarantee is absent in conventional HLS, where timing properties are internal to the tool and lost at module boundaries.

Efficiency Limitations. While timing constraints map naturally to SDC, our current prototype uses SMT solving to jointly handle binding and scheduling, which is more expensive than the heuristic-based approaches [3, 9] used in production HLS tools. Exploring more efficient algorithms for this combined problem is future work.

REFERENCES

[1] AMD Xilinx. 2024. *Vitis High-Level Synthesis User Guide*. https://docs.amd.com/r/en-US/ug1399-vitis-hls UG1399 (v2024.1).

[2] J. Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *2006 43rd ACM/IEEE Design Automation Conference*. 433–438. doi:10.1145/1146909.1147025

[3] Steve Dai and Zhiru Zhang. 2019. Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC ’19)*. Association for Computing Machinery, New York, NY, USA, Article 127, 6 pages. doi:10.1145/3316781.3317842

[4] Google LLC. 2026. XLS: Accelerated HW Synthesis. https://google.github.io/xls/. Accessed 2026-01-29.

[5] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D. Hämäläinen. 2019. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2019), 898–911. doi:10.1109/TCAD.2018.2834439

[6] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604. doi:10.1109/TCAD.2015.2513673

[7] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (June 2023), 25 pages. doi:10.1145/3591234

[8] Rachit Nigam, Ethan Gabizon, Edmund Lam, Carolyn Zech, Jonathan Balkind, and Adrian Sampson. 2026. Parameterized Hardware Design with Latency-Abstract Interfaces. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://people.csail.mit.edu/rachit/files/pubs/lilac.pdf To appear.

[9] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 211–218. doi:10.1109/ICCAD.2013.6691121