A Position on Tropical-Time for HLS

Sijie Kong
University of California, Santa Barbara
USA

Jonathan Balkind
University of California, Santa Barbara
USA

1 INTRODUCTION

High-level synthesis (HLS) has long promised to make hardware design more productive by importing software-style abstraction and tooling. Its value proposition is well understood: improved developer productivity, portability across targets, faster iteration, and simplified verification and integration through mature compilation pipelines. Most modern HLS systems are organized around a *control–dataflow graph* (CDFG) representation, which factors programs into control constructs (loops and branches), functional decomposition, and variable-level dependencies. This compact abstraction preserves enough structure to enable scheduling, pipelining, and resource mapping, while intentionally hiding many low-level microarchitectural details.

Yet the same abstraction that makes HLS scalable also creates a persistent barrier: limited control over time. By construction, CDFG-style languages obscure cycle-level behaviors and the exact temporal relationship between interacting components. In practice, designers who need predictable, cycle-accurate interfaces often fall back to RTL, or rely on fragile manual guidance. Prior work has repeatedly shown that cycle-accurate design can be especially efficient for stream-based accelerators, where explicit cycle alignment can eliminate unnecessary buffering and module-level handshaking [6, 9]. Cycle-accurate specifications also ease interoperability with highly optimized blocks—for example, integrating FFT engines or systolic arrays into larger systems—by making latency and interface timing explicit rather than emergent.

Existing approaches provide only partial relief. Commercial flows often depend on directives (e.g., pragmas) to steer timing and microarchitecture [1, 7]; however, the resulting implementations can be unpredictable and difficult to tune toward a specific intended structure [5]. Other systems elevate specialized microarchitectures to first-class language constructs [2], improving predictability for those patterns but constraining generality and limiting reuse across domains. Meanwhile, several proposals introduce languages with explicit time or event concepts [10, 12], but they largely remain at the RTL level. Even in HLS-adjacent work that incorporates time, the expressiveness is often narrow: the time model in HIR [9] and Aetherling [6], for instance, does not capture general cycle-accurate behaviors needed for broad design patterns.

This paper argues for a different point in the design space. Rather than proposing yet another language or toolchain, we take a position on a missing intermediate abstraction: a principled way to attach and manipulate timing information within a CDFG-style intermediate representation (IR) without collapsing back to RTL. Concretely, we advocate *tropical algebra* as a foundation for time annotations in HLS IRs.

Our position is that tropical-time provides a compelling set of properties for bridging software-style CDFG abstraction and hardware-style cycle accuracy. Specifically, we argue that tropical-time is: (1) *expressive* enough to describe cycle-accurate behaviors, including static and dynamic latency relationships; (2) *compatible* with the CDFG abstraction, so time annotations can be added alongside the existing dependency analysis rather than replacing it; (3) *exploration-friendly*, leaving sufficient room for scheduling, pipelining, and resource trade-offs instead of hard-coding microarchitectures; and (4) *verifiable*, supporting reasoning about time properties.

By focusing on an IR-level notion of time, we aim to clarify a path toward HLS systems that preserve the productivity benefits of CDFG-based compilation while providing designers and compilers with explicit, analyzable control over temporal behavior.

2 THE TROPICAL-TIME IR

Tropical-time is inspired by the tropical semiring [11] but slightly different. It treats *time* as symbolic expressions

$$e ::= T \mid e + n \mid \max(e_1, e_2) \qquad (T \text{ a symbolic time},\ n \in \mathbb{N}),$$

i.e., it supports (i) *constant delay* via $e+n$ and (ii) *join* via $\max(e_1, e_2)$. We extend a conventional CDFG with *timed regions* annotated by half-open intervals $[T_{\text{start}}, T_{\text{end}})$, meaning the region may start at T_{start} and must finish before T_{end}. The IR provides time-aware operations. A `wait` monitors an event starting at time T and ending at the (symbolic) trigger time T'. A `sample` reads an input port at a given time and produces a value. The following program waits for two independent valid signals, samples their inputs at the trigger times, and performs computation once both values are available (Figure 1):

```
port valid1, valid2, input1, input2;
var  val1, val2, outval;
@[T,  T1) { wait(valid1); }
@[T,  T2) { wait(valid2); }
@[T1, T1+1) { val1 <- sample(input1); }
@[T2, T2+1) { val2 <- sample(input2); }
@[max(T1,T2)+1, T') {
  // use val1 and val2 in CDFG-style code
  outval = val1 + val2;
  // ... (other computation to be scheduled)
}
```

To give each tropical-time program a unique SSA meaning, we run a *liveness check* over annotated time intervals. Every definition and use carries an interval. For a valid use of v in $[T_s^u, T_e^u)$, there must exist exactly one definition of v in $[T_s^d, T_e^d)$ that (i) happens-before the use, i.e., $T_e^d \le T_s^u$, and (ii) is the latest reaching definition among all defs of v, i.e., $\forall [T_s^{d'}, T_e^{d'}) \in \text{Defs}(v)$, $(T_e^{d'} \le T_s^u) \Rightarrow (T_e^{d'} \le T_s^d)$. We also require def/use endpoints to be comparable in the IR's time order, avoiding ambiguity between symbolic times. The check is efficient: for each variable, scan defs/uses in time order

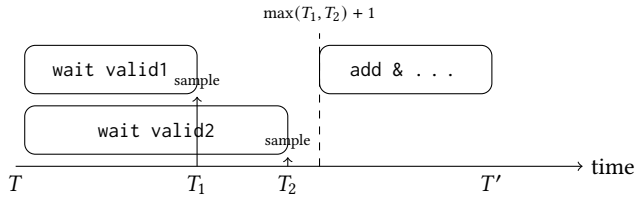

Figure 1: A possible execution.

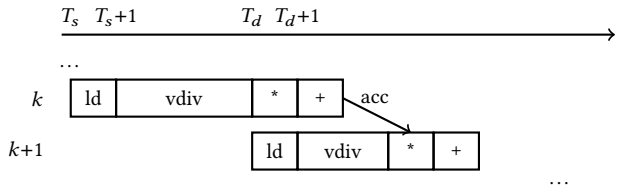

Figure 2: A variable-latency pipeline schedule: iteration k+1 can start at T_d.

while maintaining the most recent reaching def. Many hardware-centric analyses (e.g., resource usage over time, port occupancy) reduce to the same core task: whether the relevant liveness intervals are disjoint; we leave these extensions to future work.

Time intervals also expose optimization freedom by making non-overlap explicit. For example, `@[T, T+1){s0=a0+b0;}` and `@[T+1, T+2){s1=a1+b1;}` are disjoint, so a compiler may safely map both additions onto the same physical adder via time-multiplexing. In contrast, `@[T1,T1+1){...}` and `@[T2,T2+1){...}` do not establish an order between T_1 and T_2; without comparability, the compiler must conservatively assume the intervals may overlap and therefore cannot share a single adder.

3 LOOPS

Before introducing loop construct in tropical-time IR, we define *loop validity*, focusing only on timing well-formedness: (1) the loop cannot "peek" into the future, and (2) it admits a finite-state implementation (it never requires tracking infinitely many in-flight iterations). Concretely, let iterations be indexed by $i \in \mathbb{N}$ with start/end times $T_s^{(i)}$ and $T_e^{(i)}$. A loop is valid iff (i) iteration starts are strictly ordered, i.e., $\forall i,\ T_s^{(i+1)} > T_s^{(i)}$, and (ii) in-flight concurrency is finite at every concrete time, i.e., $\forall t,\ \left|\left\{ i : T_s^{(i)} \le t \wedge T_e^{(i)} > t \right\}\right| < \infty$. Since (ii) is hard to prove directly, we check the sufficient "finite-lag" condition (iii) $\forall i,\ \exists n \in \mathbb{N},\ T_s^{(i+1)} + n \ge T_e^{(i)}$; together with (i), it bounds how many later starts can occur before $T_e^{(i)}$ and thus implies (ii). Exposing next iteration's start $T_s^{(i+1)}$ as an explicit user defined expression based on current iteration makes such finite-lag obligations routine for common pipelined patterns (Figure 3).

We use the following tropical-time loop construct, where each iteration k has an explicit start time $T_s(k)$, and the next start $T_s(k{+}1)$ may be defined from events in the current iteration. For simplicity, we assume `*` and `+` have latency/II of 1, while `vdiv` has variable latency (at least 1 cycle); the body is otherwise fixed-scheduled. Intuitively, each iteration loads `A/B/C`, launches a division, waits for it to complete, updates the loop-carried accumulator `acc`, and only then starts the next iteration (Figure 2):

```
var i, acc;
loop (%k : nat) { // %k is the iter index
  var a, b, c, d, e;
  @[Ts(%k),Ts(%k)+1) {a=load A[i]; b=load B[i]; c=load C[i];}
  @[Ts(%k), Ts(%k)+1) { i = i + 1; }
  @[Ts(%k)+1, Td(%k)) { d = vdiv(a, b); }
  @[Td(%k), Td(%k)+1) { e = acc * d; }
  @[Td(%k)+1, Td(%k)+2) { acc = e + c; }
  Ts(%k+1).set(Td(%k)); // define start of next iter
}
```

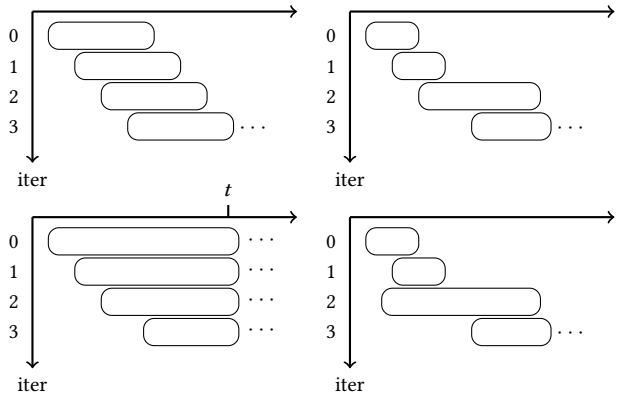

Figure 3: Four loop timing patterns in tropical-time. Top-left: a static pipeline with fixed II and latency. Top-right: a dynamic pipeline whose issue times and latencies may vary, yet remains time-valid when (i) and (iii) hold. Bottom-left: an invalid loop violating (ii) at time t. Bottom-right: an invalid loop violating (i) at iteration 1 and 2.

The only loop-carried dependence is `acc`, and the IR can infer it via interval-based def–use analysis: iteration k writes `acc` in $[T_d(k){+}1,\ T_d(k){+}2)$, while iteration k+1 starts at $T_s(k{+}1) = T_d(k)$, so any use scheduled at or after $T_d(k) + 2$ is forced to read the unique latest reaching def. The loop is also well-formed: $T_s(k{+}1)$ is derived from a completion event (with $T_d(k) \ge T_s(k){+}2$), so starts strictly increase (no future peeking), and the lag of each iteration is 2. The same interval reasoning supports checking hardware-related properties—e.g., that `vdiv` invocations across iterations are non-overlapping.

4 PROTOTYPE AND EARLY RESULTS

We built a minimal prototype of tropical-time IR as a Python-embedded DSL. The frontend allows user-omitted "time holes," and emits tropical-time IR under a simple operator model (e.g., mul: II= 1, lat= 3; add: II= 1, lat= 1). We extract timing constraints, solve them with an ASAP heuristic [3] and ILP [8], then generate a scheduled FSM and lower to PyRTL [4] for simulation and Verilog. The prototype currently supports only sequential execution and statically scheduled pipelines, with minimal optimization; its main goal is to exercise the IR checks (interval def–use uniqueness and loop-validity). In an early test, `dot2` ($a_0b_0 + a_1b_1$) synthesizes with expected timing and basic resource sharing (one multiplier for II= 2, lat= 4; two for II= 1, lat= 4).

REFERENCES

[1] Alain Darte AMD. [n. d.]. AMD Vitis™ High-Level Synthesis (HLS) Tool: Principles and Evolution. ([n. d.]).

[2] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A programming model for composable accelerator design. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 593–620.

[3] Jason Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of the 43rd annual Design Automation Conference*. 433–438.

[4] Deeksha Dangwal, Georgios Tzimpragos, and Timothy Sherwood. 2020. Agile hardware development and instrumentation with PyRTL. *IEEE Micro* 40, 4 (2020), 76–84.

[5] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. 2020. Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 244–254.

[6] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 408–422.

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

[8] Gurobi Optimization, LLC. 2026. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[9] Kingshuk Majumder and Uday Bondhugula. 2023. HIR: An mlir-based intermediate representation for hardware accelerator description. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 189–201.

[10] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular hardware design with timeline types. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 343–367.

[11] Jean-Eric Pin. 1998. Tropical semirings. *Idempotency (Bristol, 1994)* (1998), 50–69.

[12] Jason Zhijingcheng Yu, Aditya Ranjan Jha, Umang Mathur, Trevor E Carlson, and Prateek Saxena. 2025. Anvil: A General-Purpose Timing-Safe Hardware Description Language. *arXiv preprint arXiv:2503.19447* (2025).