arXiv:2304.11219v1 [cs.PF] 21 Apr 2023

LightningSim：面向高级综合的快速、准确的基于跟踪的仿真

Rishov Sarkar、Cong Hao，佐治亚理工学院电气与计算机工程学院，
rishov.sarkar@gatech.edu，callie.hao@ece.gatech.edu

摘要——高级综合使硬件设计人员能够使用 C/C++ 创建复杂的 RTL 设计。传统的 HLS 工作流程包括多轮 C/C++ 仿真，用于部分功能验证，以及 HLS 综合，用于粗略的时序估算。然而，C/C++ 仿真和 HLS 综合的估算都无法反映 FIFO 交互和流水线停顿等复杂行为，因此难以发现死锁和延迟开销等问题。这些问题只有通过 C/RTL 协同仿真才能揭示，但其速度通常比 C/C++ 仿真或 HLS 综合慢几个数量级，远不足以纳入编辑-运行开发周期。为解决这一问题，我们提出 LightningSim，这是一款面向 HLS 的快速仿真工具，兼具原生 C/C++ 的速度与 C/RTL 协同仿真的准确性。LightningSim 直接处理 LLVM 中间表示（IR）代码，准确仿真硬件设计的动态行为。首先，它跟踪 LLVM IR 执行，以捕获运行时信息；其次，将静态 HLS 调度信息映射到跟踪中，以仿真动态行为；第三，根据函数间交互计算停顿和死锁，从而得到精确的周期数。在 33 项基准测试上的评估表明，LightningSim 的时序估算准确率达 99.9%，速度最高可快于 RTL 仿真 95× 倍。我们的代码已在 GitHub 上公开。[1]

I. 引言

高级综合（HLS）工具可将高级软件编程语言（通常为 C 或 C++）中注释了硬件专用指令（例如“编译指示”）的代码，转换为 Verilog 或 VHDL 的 RTL 代码。目前，HLS 工具能够通过编译并执行 HLS 源代码，进行 C/C++ 仿真，从而快速验证功能。然而，由于缺少循环次数、分支和停顿等运行时信息，HLS 工具提供的延迟估算通常不够准确 [1]，[3]，[8]，[9]。要获得周期精确时序（例如操作、函数和整个程序的时钟周期数），并检查死锁等设计问题，则需要进行完整的 RTL 仿真，

这通常比 C/C++ 仿真慢数百到数千倍，复杂设计的仿真可能需要数小时甚至数天。如何在不运行 RTL 仿真的情况下尽可能快速地获取准确的延迟信息，对于敏捷硬件开发至关重要，也备受关注。

为应对这一挑战，我们注意到，为了从 C/C++ 代码生成 RTL，高层次综合工具必须先将源代码编译为 LLVM [5]中间表示（IR）操作，并在函数和循环中对其进行精确调度，从而能够提供丰富（但不完整的）

[1]https://github.com/sharc-lab/LightningSim

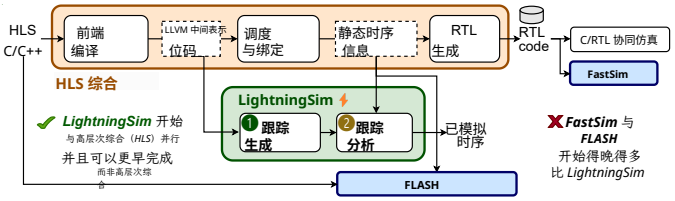

图 1. LightningSim 与 HLS 以及现有仿真工具 FLASH [3] 和 FastSim [1]的比较。LightningSim 可以与 HLS 并行启动，并且可以比 HLS 更早完成，而 FastSim 和 FLASH 的启动时间则晚得多。

延迟信息。延迟变化可归因于设计的动态特性，包括循环迭代次数、分支条件，以及先进先出队列和 AXI 接口中的停顿。幸运的是，无需调用 *RTL* 仿真，仅通过对 *C/C++* 代码进行插桩即可模拟此类动态行为。根据 *IR* 执行轨迹，我们可以精确推断从 C/C++ 代码生成的每条 LLVM IR 指令的开始周期和周期数，从而计算整个程序的准确延迟。

受加快仿真需求及其所带来机遇的启发，我们提出了 **LightningSim**，这是首个基于 Vitis HLS 的跟踪模拟器。[2] 它能在 HLS 完成前端编译和调度后立即提供整个程序准确的时钟周期数，其结果与 RTL 仿真相似（准确率达 99.9%），甚至早于 RTL 代码生成。图 1 展示了它与 HLS 的交互方式，并将其与两种现有的快速仿真工具 FastSim [1]和 FLASH [3]进行了比较。值得注意的是，LLVM 中间表示在前端编译后生成时，LightningSim 就可以与 HLS 并行启动；而 FLASH [3]必须等待调度完成，FastSim [1]则必须等到 RTL 生成完成后才能启动。

除了能够提前启动之外，LightningSim 最显著的优势是采用了一种新颖的**解耦**两阶段仿真：**中间表示跟踪生成**和**IR 轨迹分析**，如图 2 所示。首先，IR 跟踪生成阶段接收 HLS 生成的 LLVM 中间表示，通过即时补全未定义函数来支持其执行和跟踪，然后在 CPU 上执行它，以收集一条跟踪，即 LLVM IR 指令的完整执行历史。其次，跟踪分析 阶段将从 HLS 中提取的静态调度信息映射到生成的动态轨迹上，以

[2]虽然 LightningSim 基于面向 FPGA 的 Vitis HLS [10]，但经过适当的工程工作，应该可以轻松应用于其他高层次综合工具。

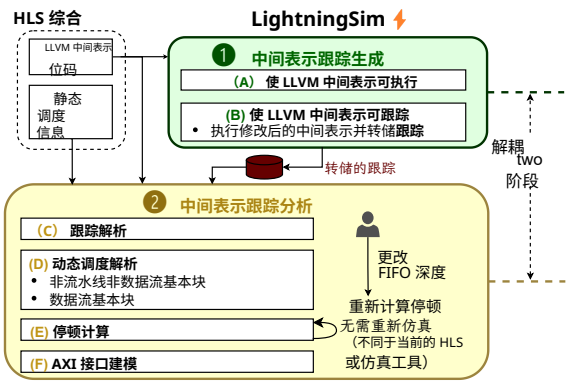

图 2. LightningSim 解耦两阶段仿真流程概览：中间表示跟踪生成与跟踪分析。

计算最终的延迟。此阶段能够捕捉程序在运行时的复杂行为，包括流水线循环、数据流、FIFO/AXI 停顿、分支等。将跟踪生成与分析解耦后，便可实现增量仿真：如果硬件配置（例如 FIFO 深度）发生变化，只需重新运行跟踪分析，而无需重新执行整个 HLS 综合和跟踪生成。与现有 HLS 工具和以往的仿真器相比，这是 LightningSim 的一项独特优势：如果仿真基于生成的 RTL，那么 FIFO 深度一旦发生变化，就必须重新生成并重新仿真整份 RTL 代码。

如表 I 所总结，我们突出介绍了相较于现有方法的贡献和优势：

- **“类似 C 语言”的仿真速度与“类似 RTL”的准确性。** LightningSim 对顺序执行的 LLVM 代码进行轻量级插桩，其性能几乎与直接在 CPU 上编译并运行 C/C++ 程序相同。此外，LightningSim 得到的周期数与 RTL 仿真获得的几乎相同。在 33 个基准测试中，包括最先进的机器学习加速器，LightningSim 的时钟周期数准确性达到 RTL 仿真的 99.9%；加速比最高可达 95× ，平均为 20.8×。
- **严谨的动态行为建模。** LightningSim 利用高层次综合提供的静态调度信息和动态中间表示执行轨迹，忠实模拟动态程序行为，包括先进先出队列停顿、死锁、流水线循环、数据流、分支等。
- **解耦的、基于跟踪的增量仿真分析。** 得益于将跟踪生成与分析解耦，LightningSim 可以在单次仿真运行后，灵活快速地事后更改仿真的硬件参数事后。据我们所知，LightningSim 是**首个** 仅凭一次仿真即可模拟任意先进先出队列深度的工具，因此无需重新运行高层次综合，即可调整或建议最优先进先出队列深度并检测死锁。
- **可与 HLS 综合并行进行。** 据我们所知，LightningSim 是首个能够在 HLS 综合完成之前，甚至在调度完成之前，就开始模拟的工具。中间表示跟踪生成步骤可在前端 LLVM 编译完成后立即开始，而前端 LLVM 编译会先于 HLS 调度、绑定和 RTL 生成完成。在 33 个基准测试中，有 13 个基准测试中，*LightningSim* 比 *HLS* 综合更早完成，尤其是那些 RTL 生成耗时较长的复杂基准测试。
- **外部存储器访问的时序模型。** FPGA 设计通常使用 AXI 协议从外部存储器——动态随机存取存储器（DRAM）中读取数据并向其中写入数据。LightningSim 为此类访问提供了高度精确的时序模型。
- **开源，使用轻松便捷。** LightningSim 可通过 GitHub[1] 公开获取。完成安装后，只需运行一条 Python 命令，即可轻松将其应用于现有的 Vitis HLS 项目，而无需重新运行 HLS 综合。

表 I LightningSim 与现有高层次综合软件仿真器的比较。

特性	FastSim [1]	FLASH [3]	LightningSim
先进先出队列周期精准	✓	✓	✓
外部存储器建模	✓	✗	✓
快速增量仿真	✗	✗	✓
可与高层次综合并行化	✗	◐[1]	✓
事件驱动仿真	✗	✗	✓
开源	◐[2]	✗	✓

[1]FLASH 必须等到高层次综合调度完成后才能运行，而 LightningSim 则无需等待。[2]截至撰写本文时，FastSim 的公开代码库（https://github.com/ckarfa/ FastSim）仅包含预先生成的示例，不包含 FastSim 工具的代码。

II. 背景与动机

A. 高级综合执行流程与局限性

高级综合的执行流程。我们以赛灵思 Vitis HLS（前身为 Vivado HLS） [10] 作为典型示例，但相同概念也适用于大多数其他 HLS 工具。HLS 综合通常包括以下步骤：

a) 前端编译：HLS 首先将输入代码编译为较低层次的中间表示，通常是 LLVM 中间表示 [5]。随后对该中间表示执行优化遍，包括 HLS 专用的自定义优化遍，以应用循环展开和数组分区等指令。

b) 调度与绑定：随后对 LLVM 中间表示中的每个函数进行调度与绑定。在调度过程中，HLS 设计有限状态机（FSM），并构建流水线和数据流，以控制每条 LLVM 指令的执行。在绑定过程中，HLS 分配硬件资源来实现有限状态机和所有指令。

c) RTL 生成：最后，HLS 输出 RTL 代码（Verilog 和 VHDL）以及报告，其中包含基于静态分析得出的资源利用率估算和粗略的时序估算。

高级综合的局限性。尽管高层次综合可通过快速运行 C/C++ 仿真来验证功能，但仅靠静态分析无法提供准确的延迟信息。正如先前研究所指出的 [1], [3], [8], , 高层次综合生成的延迟报告可能极不准确，这是由于其】【。

缺乏循环次数、分支和先进先出队列停顿等运行时信息。为了获得准确的延迟，可以运行“C/RTL 协同仿真”（又称“协同仿真”或“协同仿真”），在其中对生成的 RTL 代码进行仿真，以获得周期精确时序并进行死锁检测。然而，协同仿真比高层次综合慢几个数量级，通常需要数小时到数天，这妨碍了硬件设计工作流程的灵活性。

B. 现有工作与关键洞见

我们指出了两项专注于高层次综合仿真加速的先前工作：FastSim [1]和 FLASH [3]。FastSim 将生成的 Verilog 代码转换回等效的 C++ ，并针对有限状态机（有限状态机）等经常出现的结构进行优化，以实现更快的仿真。FLASH 利用从高层次综合综合中提取的调度信息以及输入 C/C++ 代码，生成一个 先进先出队列通信周期精确 （FCCA）C 模型。这两种方法都依赖于高层次综合调度或 RTL 生成之后 的信息。

相比之下，我们发现了一种新方法的机会：**将 HLS 调度信息直接映射到 LLVM IR 执行**。FLASH 的作者也曾谈到基于 LLVM IR 进行仿真的可能性，尽管他们选择了 C 级仿真以获得更好的可读性 [3]。在这项工作中，我们主张采用 IR 级仿真，理由如下。首先，借助 HLS 前端编译，LLVM IR 已包含循环展开和数组分区等硬件信息，因此能够更贴近实际硬件行为进行仿真，避免重复 HLS 工具已完成的工作。其次，我们可以揭示原始 C/C++ 代码与 LLVM 代码在行为上的任何差异，从而实现比单独使用 C 仿真更全面的功能验证。例如，我们的仿真会考虑嵌套循环展平优化；该优化会将内层循环外部的变量定义提升到循环内部。第三，由于 HLS 调度信息将基于 LLVM IR，我们无需等待调度完成，IR 一经生成即可执行。这也支持解耦式执行，稍后将对此进行讨论。第四，处理 IR 使 LightningSim 不受高级语言语法的限制，因此能够更灵活地适配其他 HLS 工具。

C. 即时仿真与基于跟踪的仿真

虽然我们提出了将 HLS 调度信息映射到 LLVM 中间表示的方法，但仍有两种可行方案：一步式即时仿真和解耦式基于跟踪的仿真。LightningSim 采用后者，它在两个方面明显优于前者：仿真启动时间，以及硬件配置变更时进行部分仿真。

首先，由于即时仿真会执行 LLVM 中间表示指令，并利用硬件调度信息同时计算延迟，因此在调度完成之前无法开始仿真。相比之下，在解耦的基于跟踪方法中，只要高层次综合前端编译完成，跟踪生成就可以开始，而这通常只需几秒钟。其次，由于即时仿真会在执行过程中计算延迟

因此，对仿真参数进行任何调整（例如 FIFO 深度）都需要重新启动整个仿真流程，包括使用新参数重新编译 LLVM 中间表示，以及重新运行编译后的中间表示二进制文件。相比之下，解耦的基于跟踪方法无需在第一步中重新运行生成的跟踪，而是可以直接执行第二步，使用新参数分析该跟踪。这对于包含 FIFO 和数据流的设计尤其有用：LightningSim 无需反复运行 HLS 综合和协同仿真，即可确定最优 FIFO 深度。

III. LIGHTNINGSIM 概述与挑战

图 2 展示了 LightningSim 的概览，其中包括两个解耦的阶段： 中间表示跟踪生成和 中间表示轨迹分析。首先，LightningSim 将 LLVM 中间表示编译为可执行二进制文件并运行它，以生成执行跟踪。其次，LightningSim 分析该跟踪，并以层次化方式计算整个程序的延迟，计算粒度从指令、基本块到函数。在这两个阶段中，我们发现了以下主要挑战：

1)**支持 LLVM 中间表示的执行**。虽然 LLVM 项目提供了后端基础设施，能够将中间表示代码编译为适用于主流 CPU 架构的本机机器码， [5],但 HLS 生成的中间表示代码仅供内部使用，因此由于存在未定义或虚设函数，无法直接编译为软件可执行文件。我们必须即时定义或重新定义这类函数，才能支持执行。

2) **启用 LLVM 中间表示跟踪**。HLS 工具提供的调度信息以函数（对应一个硬件模块）中每条 LLVM 中间表示指令的硬件起始和结束阶段（大致相当于时钟周期）表示。为了准确统计时钟周期，必须跟踪 LLVM 中间表示指令的执行顺序，以及其他可能影响时序和停顿的事件，例如从先进先出队列和 AXI 接口读取或写入数据。

3) **动态调度解析**。HLS 仅为每个模块提供静态调度，但不会考虑动态程序行为。在 RTL 生成过程中，静态调度中的每个阶段都会映射为有限状态机（FSM）的一个状态，但状态并非按顺序执行：有限状态机可能因分支和多次迭代的循环而跳过某些状态。流水线循环使问题更加复杂，因为不同循环迭代中的阶段执行可能重叠。因此，我们必须根据静态调度和指令执行跟踪重新计算动态调度信息。

4) **停顿计算**。 动态调度的每个阶段都对应一个时钟周期，但停顿时除外。尝试从空 FIFO 读取、向已满 FIFO 写入，或从尚未准备就绪的 AXI 接口读取等操作时，都可能发生停顿。因此，我们必须跟踪每个 FIFO 和 AXI 接口的状态，并根据动态调度计算停顿周期。

表 II 已实现的用于启用中间表示执行的高层次综合专用函数。

缺失的内建函数	llvm.part.select.i*N*.i*N*(...) llvm.part.set.i*M*.i*M*.i*N*(...)
缺失的函数	_autotb_FifoRead_i*N*(...) _autotb_FifoWrite_i*N*(...) _ssdm_op_ReadReq.m_axi.p1i*N*(...) _ssdm_op_Read.m_axi.p1i*N*(...) _ssdm_op_WriteReq.m_axi.p1i*N*(...) _ssdm_op_Write.m_axi.p1i*N*(...) _ssdm_op_WriteResp.m_axi.p1i*N*(...) _ssdm_SpecMemSelectRead(...)

5) AXI 建模。 对于通过 AXI 接口进行的外部存储器访问，停顿计算尤其具有挑战性，因为其内部行为复杂，难以准确预测 AXI 事务的时序；FLASH [3] 没有针对这一点的时序模型。我们必须为 AXI 事务设计一个模型，使其尽可能接近高层次综合生成的 RTL 设计。

在下一节中，我们将讨论 LightningSim 应对这些挑战的两个阶段。

IV. LIGHTNINGSIM 技术

如图 2 所示，跟踪生成阶段分为两个步骤：**跟踪生成**有两个步骤：**(A)** 使 LLVM 中间表示可执行；**(B)** 使 LLVM 中间表示可跟踪，执行该表示，然后转储跟踪。跟踪分析阶段分为四个步骤：**(C)** 跟踪解析；**(D)** 动态调度解析；**(E)** 停顿计算；**(F)** AXI 接口建模。

A. 使中间表示可执行

高层次综合前端编译会生成用于调度、绑定和 RTL 生成的 LLVM 中间表示代码。LightningSim 提取该中间表示代码，并为 CPU 执行做好准备。

LLVM 项目内置了代码生成后端，可将中间表示代码编译为适用于现代 CPU 架构（包括 x86 和 x86-64 [5]）的本机机器码。然而，由于高层次综合生成的 LLVM 中间表示代码并非用于生成 CPU 机器码，因此存在必须定义或重新实现的未定义函数和缺失的内建函数即时。新实现的高层次综合专用内建函数和函数汇总于表 II。

*1)*实现缺失的内建函数：按位运算（如位选择和位操作）是常见且高效的硬件操作。Vitis HLS 编译器频繁使用位操作内建函数 llvm.part.select.* 和 llvm.part.set.*，但最新版本的 LLVM 中缺少这些函数的实现。因此，我们通过在 LLVM SelectionDAG 中为每个内建函数定义一个降低操作来重新实现它们。每个降低操作都会将该内建函数替换为一系列位操作，这些操作仍受 LLVM 代码生成器支持。

LightningSim 使用 Clang 编译 HLS 生成的中间表示，以正确处理这些内建函数。

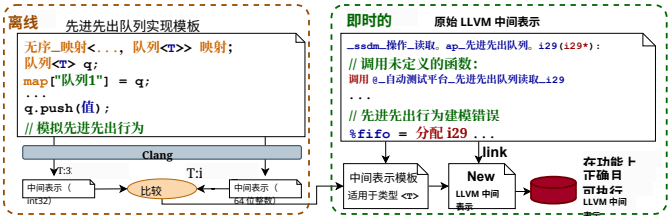

图 3。LightningSim 会将未定义或定义不正确的函数即时实现为新的中间表示，然后从原始中间表示链接过去，以构建功能正确且可执行的中间表示代码。

*2)*实现并链接缺失的函数：Vitis HLS 生成的 LLVM 中间表示包含对未定义或虚设函数的引用，这可能会妨碍软件可执行文件的生成，也无法准确反映 HLS 设计的行为。为了执行中间表示，我们必须即时重新定义这些函数，因为其中一些函数可能具有无法预先定义的任意参数。LightningSim 实现的函数完整列表汇总于表 II，其中名称以 _ssdm 或 _autotb 开头的函数由 HLS 特殊处理，但没有正确的实现。

我们以任意精度先进先出队列实现为例进行说明。如图 3 所示，原始 LLVM 中间表示函数 _ssdm_op_Read.ap_fifo. i29(i29*) 仅调用了一次未定义函数_autotb_FifoRead_i29(i29*)，该调用被解释为从 29 位宽先进先出队列中读取数据。为了准确计算设计延迟，在中间表示执行期间，必须如实体现先进先出队列行为（入队、出队）。然而，在原始 LLVM 中间表示中，它仅表示为指向 29 位整数的指针，即 %fifo = alloca i29，因此无法正确模拟先进先出行为。另一项挑战是，高层次综合允许使用任意精度数据类型的先进先出队列，而这些队列与原生 C++ 不兼容。

为正确模拟任意精度数据类型的先进先出行为，我们提出了一个模板化的 C++ 文件，LightningSim 会动态将其编译为 LLVM 中间表示。如图 3 所示，整个过程分为离线和在线两个步骤。首先，在执行之前，LightningSim 使用原生 C++ 数据结构 std::queue 表示先进先出队列，并使用 std::unordered_map 将所有先进先出队列指针映射到实际的 C++ 队列。此外，每次写入和读取都会连同先进先出队列指针一起记录到一个特殊文件中，供 LightningSim 后续进行跟踪分析。为了支持任意精度数据类型，我们还创建了一个可适配任意位宽 N 的 LLVM 中间表示模板。我们首先分别针对 $N = 32$ 和 $N = 64$ 编译两个版本的函数，生成中间表示；它们对应的原生 C++ 类型分别为 int32_t 和 int64_t。通过比较这两个函数之间的差异，我们可以确定当 N 为任意数值时所需的修改，并据此生成正确的中间表示函数。其次，在执行过程中，LightningSim 读取该模板，针对给定的 N 实例化模板，单独编译新生成的中间表示代码，并将其与原始 LLVM 中间表示链接，最后

构建出功能正确且可执行的中间表示代码。

对于其他未定义或不可用的函数（例如 AXI 读写函数），LightningSim 也采用了类似的方法。HLS 生成的中间表示中的定义具有弱链接，而 LightningSim 的定义具有外部链接，因此在链接时会优先采用 LightningSim 的定义，从而在仿真时提供正确的行为。

B. 使中间表示可跟踪

为了准确表示硬件时序，LightningSim 需要了解 CPU 仿真期间执行了哪些特定的 LLVM 中间表示指令，以及这些指令的执行顺序。例如，需要了解执行了哪些条件语句，以及每个循环的迭代次数。LightningSim 不会在中间表示中逐一识别和处理每种控制流结构，而是采用一种通用方法来跟踪所有控制流的执行。

在 LLVM 中间表示及类似的低级语言中，每个函数都由组织在 基本块 中的指令组成；基本块是一组具有单一入口点和单一出口点的指令。入口点可以是 br（分支）和 switch 的跳转目标；出口点（又称 终结指令）可以是 br、switch 或 ret（返回）。

基本块的执行等价于其中所有指令的顺序执行。为了按照精确顺序跟踪已执行的指令，我们只需跟踪已执行的基本块。这是一种广为人知的 CPU 指令跟踪方法 [2]。

LightningSim 使用自定义 LLVM 遍遍历所有基本块，并添加一个调用，调用中包含可唯一标识各个基本块的参数，该调用指向特定的跟踪函数。该跟踪函数定义在一个独立的 C++文件中，该文件随 LightningSim 一同提供并参与链接；它负责记录每个基本块的执行情况，并将其记录为“跟踪基本块”，其中包含函数名称和该函数内的基本块编号。

经过 LLVM 遍以及第 IV-A 节所述的流程后，中间表示代码可在 CPU 上执行。生成的跟踪包含第 IV-A2 节所述新定义函数的跟踪日志，从而支持 LightningSim 的第二阶段：跟踪分析。

C. 跟踪解析

跟踪解析为第二阶段——跟踪分析做好准备。执行后生成的跟踪是“扁平的”，如图 4 所示的示例，即跟踪数据。扁平的跟踪数据包括
指令的确切顺序、读取和写入的 FIFO，以及通过 AXI 接口读取和写入的片外 DRAM 地址。然而，由于 HLS 中的函数会变成并行运行的硬件模块，我们必须隔离每次函数调用的跟踪数据，以便独立处理，这就需要进行跟踪解析。因此，LightningSim 将扁平跟踪转换为函数调用的层次结构，如图 4 下半部分所示。解析后，每次调用由一个基本块列表组成，其中每个基本块都具有

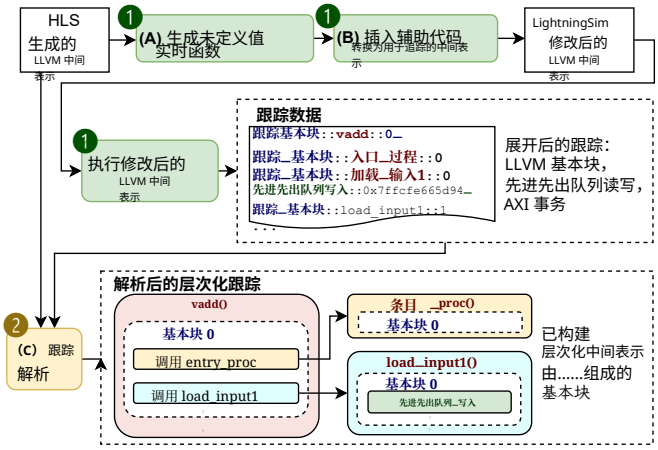

图 4. LightningSim 将扁平的执行跟踪解析为层次化数据结构，其中包含每个函数调用中的基本块。每个函数都包含 FIFO 操作、AXI 事务和函数子调用等事件。每个子调用都会递归解析为自己的层次化数据结构。

将 AXI、FIFO 和/或子调用指令映射到跟踪中对应行的关系。对于子调用，该结构以递归格式包含一个嵌套的基本块列表。

D. 动态调度解析

动态调度解析是 LightningSim 中的关键步骤。如图 5 所示，它使用由高层次综合生成的**静态调度**和 LightningSim 第一阶段生成的跟踪，来确定每条指令和每个函数的**动态调度**。随后，在完成停顿计算和 AXI 建模后，统一计算**最终时钟周期数**。

1) 静态调度与动态调度之间的差异：前端编译和调度完成后，HLS 会为每个函数生成静态调度，并用阶段进行标注，指出每条指令的起始阶段和结束阶段。在生成的 RTL 中，每个函数都映射为一个由有限状态机（FSM）控制的硬件模块。**每个阶段通常对应一个时钟周期，但由于程序的动态行为**，也存在若干例外，包括先进先出队列停顿、对其他函数的子调用，以及流水线和数据流的特殊行为。具体而言，在流水线中，多个阶段可以同时执行；在数据流中，数据流进程（子调用）的开始和结束由其输入和输出决定，而非其在调度中的位置；在循环中，循环体可以执行多次。因此，要准确计算时钟周期，LightningSim 需要根据已执行轨迹，将每个函数的静态调度解析为动态调度，其中阶段编号单调递增。

LightningSim 跟踪解析的基本单位是基本块（BB）：跟踪中的每个基本块实例都可以关联到 HLS 生成的调度中的一组“静态阶段”，以及根据仿真行为确定的一组“动态阶段”。来自 HLS 的信息有助于识别哪些基本块属于

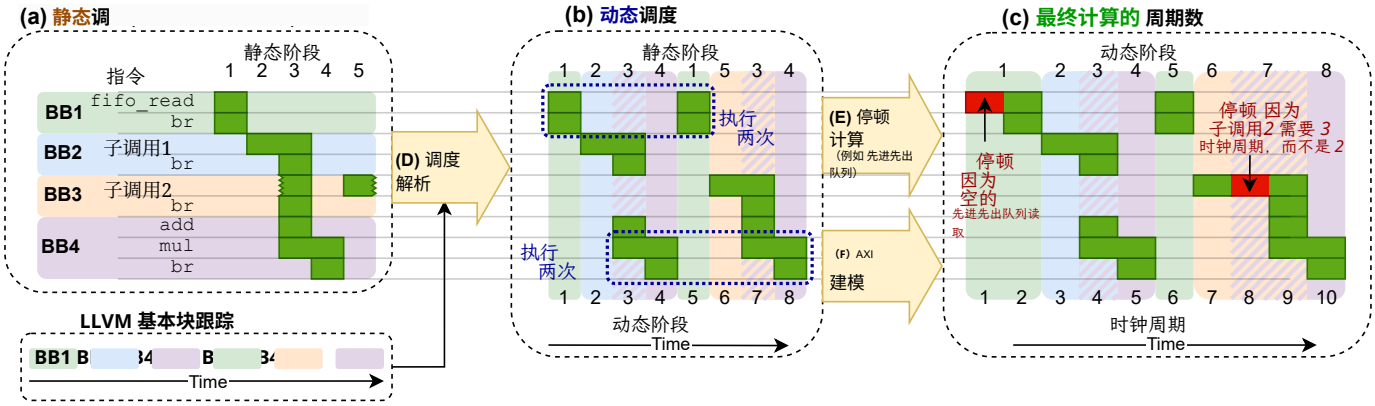

图 5. LightningSim 对单个函数调用进行跟踪分析的过程示例。利用仿真期间生成的 LLVM 基本块跟踪（§IV-B）以及 HLS 综合得到的静态调度数据，调度解析（§IV-D）解析出动态调度，其中动态阶段随时间单调递增。随后，停顿计算（§IV-E）确定每个阶段的停顿周期。

算法 1 动态调度解析

要求： 基本块的静态起止阶段、已执行轨迹、流水线信息
确保： 每个基本块的动态起止阶段

```
bb_prev ← ∅
基本块 prev.static 起始 ← 0，基本块 prev.static 结束 ← 0
基本块 prev.dynamic 起始 ← 0，基本块 prev.dynamic 结束 ← 0
遍历基本块 ∈ 跟踪（所有基本块）执行
    delay ← 基本块.static 起始 − 基本块 prev.static 结束
    如果非流水线则
        delay ← max(delay, 1)
    结束 if
    如果基本块启动一次新迭代则
        如果 非流水线则
            延迟 ← 1          ▷当前基本块在没有重叠的情况下开始
        else
            延迟 ←延迟 + 流水线.启动间隔          ▷重叠，但增加启动间隔
        结束 if
    结束 if
    基本块.dynamic 开始 ←基本块 prev.dynamic 结束 + 延迟
    基本块.dynamic 结束 ← 基本块.dynamic 开始 + 跨度 − 1
结束 for
```

流水线和数据流区域，可分别处理三种情况：非流水线非数据流基本块、流水线基本块和数据流基本块。

2) 解析非流水线非数据流基本块：LightningSim 通过更新跟踪中每个基本块的三个状态变量，迭代地执行跟踪解析。每个跟踪解析过程都在单个函数内进行。这三个状态变量是：

- 静态起止阶段：当前基本块在静态调度中开始/结束的阶段。这些信息取自 HLS生成的调度信息。
- 动态起止阶段：当前基本块在动态调度中开始/结束的单调递增阶段。这是 LightningSim 为每个基本块的每次出现计算的值。
- 新迭代：一个标志，用于指示基本块是否开始新的循环迭代。如果它是新迭代中的第一个基本块，则会立即开始执行，不会有延迟，

否则，它将遵循既定的静态调度。

算法 1 展示了解析动态调度的过程。我们使用图 5 中的示例逐步说明计算过程。

输入。 如图 5(a) 所示，共有四个基本块，从 BB1 到 BB4，每个基本块包含若干条指令。每条指令都关联一个由 HLS 调度生成的静态阶段。同时，LightningSim 还生成了这些 LLVM 基本块的跟踪。在此跟踪中，BB1 和 BB4 都执行了两次。基于该静态调度和跟踪，我们计算动态调度。

基本块静态起止阶段和跨度。 当跟踪中遇到一个新的基本块时，我们首先确定其静态起止阶段。通常，基本块的静态结束阶段是该基本块终结指令（例如 br）所在的静态阶段。例如，BB1 的静态结束阶段为 1，而 BB2 的静态结束阶段为 3。每个基本块还与一个跨度相关联，即该基本块处于活动状态的阶段数。例如，BB1 的跨度为 1，而 BB2 的跨度为 2。

我们注意到一种特殊情况：基本块的静态起始阶段不一定始终是该基本块中所有指令的静态起始阶段中编号最小的阶段。在图 5 所示的示例中，基本块 BB3 跨越第 3 阶段和第 5 阶段，但它实际上从第 5 阶段开始，并在第 3 阶段结束。LightningSim 会识别这些特殊情况，并使用其实际起始阶段。

基本块延迟。 要确定基本块的动态起止阶段，我们必须先知道该基本块在前一个基本块执行完毕后还需等待多少个时钟周期。此值记为延迟，计算方式为 $bb.static\ start - bb\ prev.static\ end$（算法 1 第 5 行）。如果延迟为 1，则表示在跟踪中，当前基本块可以在前一个基本块执行完毕后立即开始。如果延迟为零或负数，则表示当前基本块可以与前一个基本块重叠执行。如果延迟大于 1，我们总是将其限制为 1，因为这表示存在有限状态机能够在非流水线循环中跳过的空阶段（第 7 行）。

基本块的动态起止阶段。 根据前一个基本块的动态结束时间和当前基本块的延迟，计算当前基本块的

动态起始时间为 $bb\ prev.dynamic\ end + delay$（第 16 行），动态结束时间增加一个跨度（第 17 行）。

❒ **示例** 见图 5。**BB1**。 作为跟踪中的第一个基本块，BB1 的静态和动态起止阶段均为 1。**BB2**。 BB2 紧随 BB1，其静态起始阶段和结束阶段分别为 2 和 3，跨度为 2。其延迟为 1，计算方式为 2（BB2 的静态起始阶段）减去 1（BB1 的静态结束阶段）。BB2 的动态起始阶段为 2，计算方式为 1（BB1 的动态结束阶段）加 1（延迟），动态结束阶段为 3。**BB4**。 其延迟为 $3-3=0$，这意味着从静态调度中可以看出，BB4 与其前一个基本块 BB2 有一个阶段的重叠。因此，其动态起始阶段为 $3+0=3$，动态结束阶段为 4。

新迭代标志。 在非流水线循环中，前一次迭代的最后一个基本块一执行完毕，新一轮迭代就会立即开始（流水线循环将在后文讨论）。为对此情况建模，我们维护一个新迭代 标志。设置此标志后，启动新一轮迭代的第一个基本块将比其前一个基本块晚一个阶段开始执行，即强制将延迟 设为 1，以确保既无重叠，也无空状态。当前基本块更新后，会清除此标志，以确保下一轮迭代符合要求。

❒ **示例**见图 5。**BB1** （第二次）。在此跟踪中，BB1 第二次出现，表明新迭代开始，因此 BB1 可以在 BB4 完成后立即启动。因此，BB1 的第二次动态启动时间为 $4+1=5$，如图 5(b) 所示。BB1 更新后，新迭代标志被清除。**BB3**。 接下来，BB3 的延迟为 4，计算方式为 5（BB3 静态启动时间）减去 1（BB1 静态结束时间）；为模拟有限状态机跳过空阶段的行为，应将其限制为 1。因此，其动态启动时间为 $5+1=6$，结束时间为 7。**BB4**。 由于 BB4 仍处于第二次迭代中，因此遵循静态调度。因此，其动态启动时间为 $7+0=7$。如图 5(b) 底部所示，动态阶段范围为 1 到 8。

3) 解析流水线基本块： 在流水线循环中，每次循环迭代都会延迟一个启动间隔 （启动间隔），即一次迭代开始到下一次迭代开始之间的时钟周期数（动态阶段）。例如，如果循环的启动间隔为 2，那么第一次迭代将在动态阶段 0 开始，第二次在动态阶段 2 开始，第三次在动态阶段 4 开始，依此类推。因此，与非流水线循环相比，处理流水线循环中的基本块有以下几个主要区别。

首先，延迟 计算方式有所不同。对于流水线中的基本块，LightningSim 不会将延迟值限制为零，因为这表示条件语句未执行。在非流水线循环中，有限状态机可以跳过未执行的阶段；然而，在流水线循环中，每次迭代都必须具有固定数量的阶段，因此不能跳过任何阶段。其次，根据 新迭代标志开始新一轮迭代时，延迟值将设为循环启动间隔，这与流水线循环的行为一致。第三，当 LightningSim 遇到流水线循环外的基本块时，会将其 静态阶段和动态阶段 重置为循环内部遇到的最大静态阶段和动态阶段，以确保流水线阶段不会与非流水线阶段重叠。

4) 解析数据流基本块： 数据流区域内指令的静态起始和结束阶段无法准确反映其在硬件中的行为。相反，数据流进程的起始和结束由其输入和输出驱动。因此，LightningSim 会根据数据流区域内指令的输入和输出通道以及数据流进程，重新计算这些指令的静态起始和结束阶段。

起始阶段的规则如下：(1) 如果数据流进程没有输入，则会立即在阶段 0 开始。(2) 如果数据流进程有任何标量输入——即单个值而非通道的输入——则会在所有标量输入产生后的一个阶段开始。(3) 否则，数据流进程的起始由起始阶段传播决定：该进程会在其任一输入进程开始后的一个阶段开始。

对于结束阶段，规则如下：(1) 如果数据流进程有任何输出，且这些输出全部是标量输出（单值，而非通道），则该进程在其开始的同一阶段结束。(2) 否则，该进程会在数据流区域结束时与其他所有进程同步——其结束阶段与所有其他非标量输出数据流进程的结束阶段相同（由其中任一进程的最晚开始阶段确定）。

通过使用此模型重新计算数据流区域中每条指令的静态开始阶段和结束阶段，我们可以像算法 1 中处理非流水线、非数据流区域那样，复用相同的数据流解析流程。

E. 停顿计算

动态调度会为每条指令生成一个动态阶段。然而，要精确计算时钟周期数，仍有两个问题需要解决。首先，有些阶段可能需要一个以上的周期才能完成，因为停顿会造成延迟；例如，当先进先出队列或 AXI 接口未能在高层次综合设计预期的时间就绪时。此外，先进先出队列深度过小可能导致死锁，这必须在仿真期间检测。其次，在任何函数中，停顿都可能延迟任一子调用的开始，因此被调用函数的时序取决于调用方。因此，必须以分层且全局的方式计算停顿：一个函数的停顿可能需要传播到其他函数以及其自身的调用方或被调用函数。

停顿模拟器变量。LightningSim 是一种基于事件的模拟器，用于跟踪每个函数的开始周期和结束周期。它通过首先为顶层 HLS 内核函数启动一个新的模拟器，随后在跟踪中遇到其他函数时为其启动模拟器，从而创建一组活动模拟器。每个模拟器都会维护以下变量：

- 开始周期 和当前周期，对于顶层函数，两者均初始化为零。任何时候，调用方的当前周期都将是其被调用函数的开始周期。
- 停顿事件：模拟器遇到停顿，可能是对子调用函数的调用，也可能是对先进先出队列或 AXI 总线执行 I/O 操作。
- 停顿解除：子调用完成，或先进先出队列/AXI 准备就绪时，停顿将解除。

每个仿真器都以先前步骤中解析后的动态跟踪记录为输入，并生成一系列仿真事件

在此函数中可能影响其他函数的内容。每个事件都映射到一个动态阶段。这一系列仿真事件包括以下内容：

- 子调用开始：子调用开始执行时所处的动态阶段。在 RTL 中，这对应于调用方首次断言被调用函数的 ap_start 信号时所处的有限状态机阶段。
- 子调用结束：子调用所跨越的最后一个动态阶段。在硬件中，有限状态机不会前进到此阶段之后，直到被调用函数断言其 ap_done 信号。
- *I/O* 事件：在先进先出队列或 AXI 总线上发生 I/O 操作时所处的动态阶段。硬件有限状态机不会前进到此阶段之后，直到 I/O 操作成功完成。

模拟器工作流程。从顶层函数开始，LightningSim 依次处理每个活动模拟器的模拟事件列表（最初只有顶层模拟器）。在一个模拟器中，每个动态阶段的当前周期都会递增 1。如果遇到子调用开始事件，且被调用函数的模拟器尚不存在，则会创建一个新的模拟器，其开始周期设为调用方的当前周期。这个新模拟器也会添加到 LightningSim 的活动模拟器集合中。

遍历每个活动模拟器的事件时，如果遇到停顿事件，LightningSim 必须确定该停顿的释放周期。如果停顿事件是子调用，则被调用函数执行完毕时可以释放停顿；如果停顿事件是 FIFO/AXI 操作，则 I/O 就绪时可以释放停顿。LightningSim 为每个 FIFO 和 AXI 接口维护数据结构，以精确模拟 I/O 状态。停顿事件结束后，LightningSim 会更新所有因该事件而受阻的模拟器的当前周期值，同时更新 FIFO/AXI 接口的数据结构。此过程会持续进行，直到所有模拟器都执行完毕。完成后，LightningSim 已计算出设计中每个被调用函数的开始和结束周期。

❒ **示例** 见图 5。在此示例中，我们展示了两个停顿事件：一个是 BB1 因 FIFO 读取事件而发生的停顿，另一个是 BB3 因子调用事件而发生的停顿。假设 BB1 因 FIFO 读取而停顿一个周期，BB3 则因遇到对另一个函数的子调用而停顿一个周期，而该函数调用需要 3 个周期。最后，虽然动态阶段从 1 到 8，但由于这些停顿，精确的时钟周期范围是从 1 到 10。

死锁检测。LightningSim 能够在仿真期间使用一种受缓冲区锁相关研究 [6] 启发的方法检测任何死锁。如果 LightningSim 无法计算任一停顿事件的停顿解除周期，则表明该硬件设计将发生死锁。这可能是由 FIFO 输入/输出中的循环依赖导致的：例如，模块 A 被阻塞，无法向已满的 FIFO X 写入数据，而该 FIFO 随后将由模块 B 读取；模块 B 被阻塞，无法从空 FIFO Y 读取数据，而该 FIFO 预计将由模块 A 写入。在这种情况下，LightningSim 会识别出所有仿真器都无法继续运行，即发生了死锁。

增量仿真。值得注意的是，一旦 LightningSim 检测到死锁，就能够通过更改 FIFO 深度，快速提出解决方案，并仅增量重跑停顿计算步骤，同时保持已执行轨迹和动态解析的调度不变。由于停顿计算非常高效（只需几秒），LightningSim 可以通过在仿真深度不受限制时计算观测到的最大深度，快速确定**最佳**FIFO 深度。

这与 HLS 工具和现有仿真器有着根本区别：一旦更改 FIFO 深度，它们就需要完整地重新综合，并重新执行整个 RTL 生成和仿真流程；而 LightningSim 无需如此，这得益于其**解耦的**两个阶段。

F. AXI 接口建模

HLS 生成的设计为每个 AXI 接口都集成了自己的 AXI 控制器，用于协调 HLS 设计与系统其余部分之间所有 AXI 总线上的请求和响应。例如，这些控制器会将 HLS 设计内部生成的长 AXI 突发拆分为多个符合规范的 AXI 突发，确保它们不会跨越 4 KB 地址边界。由于 AXI 控制器仅在生成的 RTL 中实现，LightningSim 会使用详细模型来计算 AXI 输入/输出事件的停顿周期，尽可能贴近 AXI 控制器复杂的 RTL 描述行为。

对于高层次综合设计中发出的每个 AXI 读取或写入请求，LightningSim 首先计算该请求的 突发计数，即如果请求跨越 4 KB 地址边界，完成该请求所需的突发次数。为模拟 fifo_rctl 的行为（它最多可同时容纳 16 个未完成突发），LightningSim 会跟踪*rctl* 深度：每个未完成请求都会使其按突发计数递增；该请求的所有读/写数据均被消耗后，则按相同数量递减。任何会使 *rctl* 深度 超过 16 个未完成突发的请求，都会排入待处理 队列，并在 *rctl* 深度 降低到足以容纳这些请求时发出。

AXI 读取和写响应的开销以固定数量的时钟周期建模，该数量通过经验确定，并叠加在高层次综合设计使用 # pragma HLS interface 指定的 AXI 接口延迟之上。这种方式能够准确模拟 Vitis HLS C/RTL 协同仿真的行为；由于高层次综合生成的 AXI 控制器和 SystemVerilog AXI 测试平台中的内部逻辑，每个 AXI 事务都会产生固定数量的额外时钟周期。

V. 执行时间

LightningSim 的时间复杂度大致可表示为（编译和运行 LLVM 中间表示所需的时间） + O（跟踪中的基本块数量） + O（跟踪中的函数调用数量） + O（跟踪中的 FIFO/AXI 读写数量），由以下各项之和推导得出：

1) 跟踪生成。这完全取决于编译和执行 LLVM 中间表示所需的时间。**2) 跟踪解析**。其耗时与跟踪中的条目数量呈线性关系。跟踪中针对以下每种事件都包含一个条目：

- 执行一个基本块
- 对先进先出队列进行读取或写入

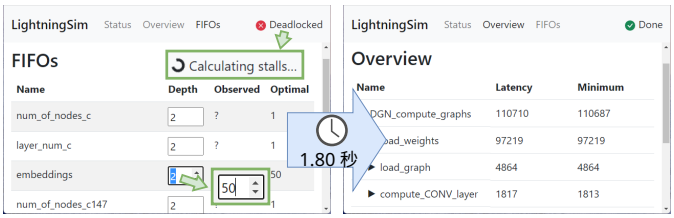

图 6。LightningSim 用户界面。FIFO 深度可以即时更改。此示例展示了 FlowGNN DGN [7], 的界面，停顿重新计算耗时 1.80 秒。

- 一项 AXI 总线操作（读取、写入、读请求、写请求或写响应）

3) **调度解析**。 如上所述，这与跟踪事件的数量也呈线性关系。

4) **停顿计算**。 这与跟踪中的仿真事件 数量大致呈线性关系（见第 IV-E 节）。这包括：

- 子调用开始（位于函数调用图中的任意位置）
- 子调用结束（位于函数调用图中的任意位置）
- 对先进先出队列进行读取或写入
- 一项 AXI 总线操作（读取、写入、读取请求、写入请求或写入响应）

更改先进先出队列深度时，只需重新计算最后一步，因此时间复杂度约为 O（跟踪中的函数调用次数） + O（跟踪中的 FIFO/AXI 读写次数）。

VI. LIGHTNINGSIM 使用方法

现在介绍 LightningSim 的使用流程。LightningSim 是开源软件，可按照我们代码仓库中的说明下载并安装。[1]

LightningSim 旨在实现一键式便捷操作。它以 Python CLI 脚本的形式提供，调用方式如下：./main.py / path/to/Vitis/HLS/project/ solution1。此操作会启动 Web 服务器，提供 Web 用户界面，让用户查看 LightningSim 流程的进度并控制其执行，如图 6 所示。

LightningSim 直接从 Vitis HLS 解目录读取所有项目设置，包括测试平台文件、编译标志等，以尽可能贴近 Vitis HLS 的 C 仿真设置。这意味着，现有的 Vitis HLS 项目无需进行**任何额外配置**即可使用 LightningSim。

默认情况下，LightningSim 会等待下一次 HLS 综合运行启动后再开始仿真；不过，如果用户想使用已完成的 HLS 综合结果运行仿真，也可以跳过此步骤。在每个阶段，LightningSim 只等待必要的 HLS 生成文件，因此 HLS 的架构综合阶段完成后，它便可立即开始 LLVM 插桩；HLS 调度数据生成后，也可立即开始跟踪分析。

LightningSim仿真完成后，网页界面的概览选项卡会以分层树状视图显示每次函数调用的时钟周期延迟，以及每次调用的最小 延迟。计算最小延迟时，假设所有先进先出队列的深度均为无限，并据此确定最终延迟；通过调整先进先出队列深度即可获得这一可能的最小延迟。LightningSim 基于跟踪的分析方法只需使用不同参数重新运行停顿计算，即可高效地计算出这一最小延迟。

Web 用户界面还提供一个先进先出队列选项卡，其中的表格列出了设计中每个先进先出队列的信息。每个先进先出队列都列有名称、深度、观测深度和最优深度。观测深度表示仿真期间该先进先出队列在任意时钟周期出现过的最大队列长度；最优深度则是对所有先进先出队列均不设上限的仿真变体所得到的相同测量值，该值用于计算前文所述的最小延迟。

可编辑的数值字段允许用户调整仿真中每个先进先出队列的深度。调整深度后，LightningSim 会在后台自动重新计算延迟和观测到的先进先出队列深度，**无需重新运行整个仿真**。这样，用户可以快速探索先进先出队列深度对硬件设计速度的影响。

VII. 结果

A. 评估设置

我们在一台运行 Red Hat Enterprise Linux 7.9 的服务器上评估 LightningSim，使用的是 2021.1 版赛灵思工具，其中包括 Xilinx Vitis HLS 2021.1。该服务器配备 64 核 Intel Xeon Gold 6226R CPU 和 502 GiB 内存。对于每个测试用例，我们将 LightningSim 的周期数和运行时间与使用 Vitis HLS 进行综合以及使用 Vivado XSIM 进行 C/RTL 协同仿真的结果进行比较。

B. 基准测试

我们使用 33 个基准测试，全面评估 LightningSim 的准确性和性能。如表 III 第二列所示，每个基准测试最多包含五种用于评估 LightningSim 的高级特性，包括：**C** 表示对其他函数的子调用；**P** 表示流水线循环；**D** 表示数据流区域；**F** 表示先进先出队列流；**A** 表示 AXI 主接口。部分基准测试来自赛灵思示例代码库 [11]，[12]，其中包含许多用于测试各类高层次综合特性的小型基准测试。我们还使用了 Kastner 等人的著作《*FPGA* 并行编程》 [4], 中的若干 Vitis HLS 项目示例，该书包含多种流行算法的高层次综合实现。最后，为了进行全面测试，我们使用了 FlowGNN [7], 中的五种设计。这些设计是最新的开源高层次综合设计，分别实现了五种图神经网络（GNN）加速器，并广泛采用数据流、流水线、数据流传输和 AXI 接口。FlowGNN 中的每种 GNN 实现均包含超过 1500 行代码和 23 个函数。

C. 性能与准确性

性能和准确性结果如表 III 所示。我们不区分仿真准备和执行：LightningSim 报告的是端到端 时序。

表 III LightningSim 在各种基准测试中的速度和准确性指标。

基准测试	特性 C	P	D	F	A	时间 (s) HLS	LS ‖ 高层次综合	协同仿真	LS	LS Inc	时钟周期 协同仿真	LS	HLS
定点平方根 [11]	✗	✓	✗	✗	✗	9.83	13.90 (+41.5%)	27.25	4.87 (5.6×)	0.00	30	30 (±0.0%)	32 (+6.7%)
有限脉冲响应滤波器 [11]	✗	✓	✗	✗	✗	7.60	8.52 (+12.0%)	20.12	1.76 (11.5×)	0.00	172	172 (±0.0%)	174 (+1.2%)
定点滑动窗口卷积 [11]	✗		✓	✗	✗	12.00	14.20 (+18.3%)	28.30	3.14 (9.0×)	0.00	35	35 (±0.0%)	37 (+5.7%)
浮点转换 [11]	✗	✓	✗	✗	✗	8.34	9.29 (+11.5%)	49.78	1.80 (27.6×)	0.00	35	35 (±0.0%)	37 (+5.7%)
任意精度算术逻辑单元 [11]	✗	✗	✗	✗	✗	9.37	10.18 (+8.5%)	24.17	1.59 (15.2×)	0.00	36	36 (±0.0%)	36 (±0.0%)
并行循环 [11]	✓	✓	✗	✗	✗	10.97	11.83 (+7.9%)	26.81	1.70 (15.7×)	0.00	32	32 (±0.0%)	34 (+6.2%)
不完美循环 [11]	✓	✓	✗	✗	✗	11.95	11.94 (−0.1%)	25.80	1.66 (15.5×)	0.00	34	34 (±0.0%)	36 (+5.9%)
具有最大界限的循环 [11]	✗	✓	✗	✗	✗	10.86	12.04 (+10.9%)	24.76	1.71 (14.4×)	0.00	31	31 (±0.0%)	33 (+6.5%)
完美嵌套循环 [11]	✗	✓	✗	✗	✗	11.48	12.30 (+7.1%)	24.76	1.70 (14.6×)	0.00	406	406 (±0.0%)	408 (+0.5%)
流水线嵌套循环 [11]	✗	✓	✗	✗	✗	11.34	12.42 (+9.5%)	24.92	1.67 (14.9×)	0.00	405	405 (±0.0%)	405 (±0.0%)
顺序累加器 [11]	✓	✓	✗	✗	✗	11.46	12.14 (+5.9%)	26.59	1.76 (15.1×)	0.00	32	32 (±0.0%)	34 (+6.2%)
累加器 +断言 [11]	✓	✓	✗	✗	✗	11.62	12.39 (+6.6%)	27.13	1.74 (15.5×)	0.00	33	33 (±0.0%)	259 (+684.8%)
累加器 +数据流 [11]	✓	✓	✓	✗	✗	11.99	12.50 (+4.3%)	27.26	1.79 (15.2×)	0.00	31	31 (±0.0%)	33 (+6.5%)
静态内存示例 [11]	✓	✓	✗	✗	✗	6.18	6.92 (+12.0%)	33.23	1.65 (20.2×)	0.00	66	66 (±0.0%)	70 (+6.1%)
指针类型转换示例 [11]	✗	✓	✗	✗	✗	5.01	6.03 (+20.4%)	32.55	1.58 (20.6×)	0.00	408	408 (±0.0%)	410 (+0.5%)
双重指针示例 [11]	✓	✓	✗	✗	✗	5.57	6.17 (+10.7%)	31.70	1.47 (21.5×)	0.00	25	25 (±0.0%)	29 (+16.0%)
AXI4 主设备 [11]	✓	✓	✗	✗	✓	6.36	6.63 (+4.3%)	21.06	1.64 (12.9×)	0.00	178	177 (−0.6%)	176 (−1.1%)
不含侧通道的 AXIS [11]	✗	✓	✗	✗	✗	5.29	6.34 (+19.8%)	19.12	1.51 (12.7×)	0.00	52	51 (−1.9%)	53 (+1.9%)
多次数组访问 [11]	✗	✓	✗	✗	✗	11.05	12.03 (+8.8%)	24.32	1.59 (15.3×)	0.00	252	252 (±0.0%)	254 (+0.8%)
已解析的数组访问 [11]	✓	✓	✗	✗	✗	10.55	11.55 (+9.5%)	24.36	1.61 (15.2×)	0.00	131	131 (±0.0%)	133 (+1.5%)
带 ECC 的 URAM [11]	✓	✓	✗	✗	✗	7.32	6.38 (−12.9%)	22.07	1.71 (12.9×)	0.00	115	115 (±0.0%)	121 (+5.2%)
定点汉明 [11]	✗	✓	✗	✗	✗	8.71	9.73 (+11.7%)	33.28	1.87 (17.8×)	0.00	259	259 (±0.0%)	261 (+0.8%)
未优化的 FFT [4]	✓	✓	✗	✗	✗	15.90	14.89 (−6.3%)	153.53	4.28 (35.9×)	1.85	261,781	261,150 (−0.2%)	?
多阶段 FFT [4]	✓	✓	✓	✗	✗	17.33	12.95 (−25.3%)	61.43	2.07 (29.7×)	0.00	3,770	3,772 (+0.1%)	?
霍夫曼编码 [4]	✓	✓	✓	✗	✗	43.82	38.79 (−11.5%)	46.89	2.29 (20.5×)	0.01	10,283	10,272 (−0.1%)	?
矩阵乘法 [4]	✗	✓	✗	✗	✗	6.87	6.81 (−0.8%)	26.33	1.84 (14.3×)	0.00	1,036	1,036 (±0.0%)	1,038 (+0.2%)
并行归并排序 [4]	✓	✓	✓	✗	✗	8.04	6.30 (−21.7%)	48.79	1.59 (30.7×)	0.00	131	131 (±0.0%)	139 (+6.1%)
使用流的向量加法 [12]	✓	✓	✓	✓	✓	11.04	10.01 (−9.3%)	27.21	3.74 (7.3×)	0.36	4,261	4,261 (±0.0%)	4,242 (−0.4%)
FlowGNN GIN [7]	✓	✓	✓	✓	✓	824.12	566.17 (−31.3%)	4219.85	44.01 (95.9×)	2.62	260,359	260,337 (−0.0%)	216,007 (−17.0%)
FlowGNN GCN [7]	✓	✓	✓	✓	✓	3224.02	3022.28 (−6.3%)	534.33	53.65 (10.0×)	19.67	112,836	112,561 (−0.2%)	68,170 (−39.6%)
FlowGNN GAT [7]	✓	✓	✓	✓	✓	965.50	723.50 (−25.1%)	838.24	42.87 (19.6×)	14.15	17,282	17,282 (±0.0%)	16,102 (−6.8%)
FlowGNN PNA [7]	✓	✓	✓	✓	✓	820.31	592.29 (−27.8%)	3285.45	56.02 (58.7×)	8.54	344,206	344,206 (±0.0%)	?
FlowGNN DGN [7]	✓	✓	✓	✓	✓	848.94	606.15 (−28.6%)	996.13	38.46 (25.9×)	1.80	110,710	110,710 (±0.0%)	93,846 (−15.2%)

特征 表示基准测试是否具有：**C**：对其他函数的子**调用**；**P**：**流水线化的**循环；**D**：**数据流**区域；**F**：**先进先出队列**流；**A**：AXI 接口。**HLS**：HLS 综合所需时间。**LS ‖HLS**：从 HLS 综合开始到 LightningSim 并行完成所需时间（与单独进行 HLS 综合相比）。**协同仿真**：C/RTL 协同仿真所需时间。**LS**：HLS 综合完成后，LightningSim 端到端仿真所需时间（与 C/RTL 协同仿真相比）。**LS 增量计算**：更改 FIFO 深度后，LightningSim 增量重新计算停顿所需时间。**协同仿真周期数**：C/RTL 协同仿真报告的时钟周期数。**LS 周期数**：LightningSim 报告的时钟周期数（与 C/RTL 协同仿真相比）。**HLS 周期数**：HLS 综合报告（如有）提供的时钟周期数（与 C/RTL 协同仿真相比）。

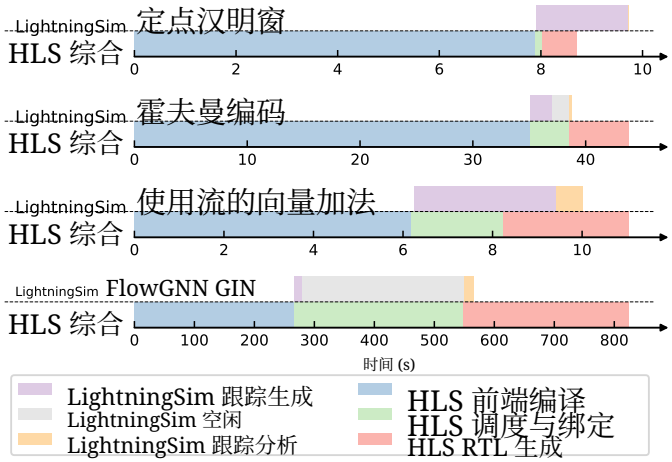

图 7. LightningSim 与高层次综合在不同基准测试中的并行执行时间线。对于复杂设计，LightningSim 的并行性尤其有优势，因为 HLS 调度/绑定和 RTL 生成需要数分钟。

该表显示了两个类别：以秒为单位的**时间**，以及端到端**时钟周期**。

时间。首先，与 C/RTL 协同仿真相比，LightningSim 实现了 5.6× 到 95.9× 倍的加速（**LS** 对比**协同仿真**）。对于 FlowGNN 模型等较大的设计，加速效果通常更为显著。该比较不包括 HLS 综合时间；也就是说，在 LightningSim 和协同仿真开始之前，我们假设 HLS 综合已经完成。

其次，当 LightningSim 与 HLS 综合并行执行时，在 33 个案例中有 13 个案例里 LightningSim 比 HLS 综合更早完成，如列 **LS ‖HLS** 中的负值所示。对于 FlowGNN 等较大的设计，LightningSim 总是比 HLS 综合更早完成。在所有 HLS 综合先于 LightningSim 完成的案例中，从综合开始到综合和 LightningSim 均完成，时间都不超过 15 秒。

第三，LightningSim 在增量停顿重新计算（**LS Inc**）方面实现了最大的加速效果。此步骤会更改 FIFO 深度，并重新执行仿真以计算停顿和时钟周期。在所有测试案例中，该过程耗时均少于 20 秒，使硬件设计人员能够以前所未有的速度评估流式数据流设计中的死锁和停顿。

遗憾的是，截至撰写本文时，FLASH [3]和 FastSim [1] 均未开源其工具，因此难以进行比较。不过，我们注意到，FLASH 的仿真准备 + 仿真时间平均比单独进行 HLS 综合慢 58%，显著高于 LightningSim 平均仅比综合慢 1.04% 的表现。

周期数。与 C/RTL 协同仿真生成的周期数相比，LightningSim 生成的周期数结果平均准确率为 99.9%。在 33 个案例中，LightningSim 有 26 个达到 100% 准确率。其余 7 个案例中，两个案例的误差分别为 1.9% 和 0.6%，但都只相差 1 个周期；另外五个案例的误差均在 0.2% 以内。

D. 高层次综合中的并行性

图 7 展示了 LightningSim 与高层次综合并行执行若干基准测试时的详细时间线。对于 FlowGNN GIN 这类大型设计，综合流程中的每一步都需要数分钟，因此 LightningSim 的效率优势最为明显。值得注意的是，在 FlowGNN GIN 中，跟踪生成和跟踪分析所需时间都短得可以忽略不计，而大部分时间都耗在等待高层次综合调度完成上。对于较小的设计，LightningSim 通常也能比综合更快，例如霍夫曼编码或带流的向量加法。

VIII. 结论

在本工作中，我们提出了 LightningSim，这是一种快速、准确的基于跟踪的 HLS 模拟器。LightningSim 将 HLS 综合过程中生成的调度信息映射到 HLS 生成的 LLVM 位码的执行上。它通过跟踪生成和跟踪分析两个阶段实现这一过程，从而具备独特的灵活性，并支持诸如与 HLS 综合并行执行，以及在无需重新仿真的情况下使用不同 FIFO 深度重新计算时序等实用功能。

LightningSim 为未来工作留下了若干发展方向。目前尚不支持某些 HLS 功能，例如流水线回退和非确定性结构。此外，更精确的 AXI 模型将使 LightningSim 能够在更多 HLS 设计中实现周期精确仿真。

LightningSim 代表着敏捷硬件开发方向上的一步，有助于弥合软件工程与硬件设计之间的开发者体验（DX）差距，并推动加速器设计民主化；通过实现更快速的设计迭代，让更多人能够自信地设计专用硬件。

致谢

本研究部分获得佐治亚理工学院新型计算层次结构研究中心（CRNCH）和2022年高通创新奖学金项目的支持。

参考文献

[1] M. Abderehman、J. Patidar、J. Oza、Y. Nigam、T. A. Khader 和 C. Karfa，“FastSim：一种用于高级综合的快速仿真框架”，*IEEE* 集成电路与系统计算机辅助设计汇刊，第 41 卷，第 5 期，第 1371–1385 页，2022年5月。

[2] T. Ball 和 J. R. Larus，“优化分析与跟踪程序”，*ACM* 编程语言与系统汇刊，第 16 卷，第 4 期，第 1319–1360 页，1994 年 7 月。[3] Y.-K. Choi、Y. Chi、J. Wang 和 J. Cong，“FLASH：快速、并行且准确的高层次综合模拟器”，*IEEE* 集成电路与系统计算机辅助设计汇刊，第 39 卷，第 12 期，第 4828–4841 页，2020 年 12 月。[4] R. Kastner、J. Matai 和 S. Neuendorffer，“FPGA 并行编程”，2018 年 5 月。[5] C. Lattner 和 V. Adve，“LLVM：用于终身程序分析与转换的编译框架”，载于 *2004* 年国际代码生成与优化研讨会。*CGO 2004*。，2004 年 3 月，第 75–86 页。[6] T. M. Parks，“进程网络的有界调度”，博士学位论文，加利福尼亚大学伯克利分校，加利福尼亚州伯克利，1995 年 12 月。[7] R. Sarkar、S. Abi-Karam、Y. He、L. Sathidevi 和 C. Hao，“FlowGNN：用于实时且与工作负载无关的图神经网络推理的数据流架构”，载于 *2023 IEEE* 高性能计算机体系结构国际研讨会（*HPCA*）。加拿大魁北克省蒙特利尔：IEEE，2023 年 2 月，第 1099–1112 页。[8] A. Sohrabizadeh、Y. Bai、Y. Sun 和 J. Cong，“利用图神经网络实现自动化 FPGA 加速器优化”，2021 年 11 月。[9] N. Wu、Y. Xie 和 C. Hao，“IRONMAN：通过强化学习在高层次综合中进行 GNN 辅助的设计空间探索”，载于 *2021* 年大湖区 *VLSI* 研讨会论文集，GLSVLSI ’21 系列。美国纽约州纽约：计算机协会，2021 年 6 月，第 39–44 页。[10] 赛灵思，“Vitis 高层次综合”，
h
t
t
ps://
www.xilinx.com/products/design-tools/vivado/high-level-design.html。[11] ——，“Vitis HLS 基础示例”，GitHub，2021 年 4 月。[12] ——，“Vitis 加速示例代码库”，GitHub，2022年。