使用 Lingua Franca 实现高性能确定性并发

Christian Menard
christian.menard@tu-dresden.de
德累斯顿工业大学，德国

Marten Lohstroh
marten@berkeley.edu 加州大学伯克利分校，美国

Soroush Bateni
soroush@berkely.edu 加州大学伯克利分校，美国

Matthew Chorlian
mattchorlian@berkeley.edu 加州大学伯克利分校，美国

Arthur Deng
langxing.deng@berkeley.edu 加州大学伯克利分校，美国

Peter Donovan
peterdonovan@berkeley.edu 加州大学伯克利分校，美国

Clément Fournier
clement.fournier@tu-dresden.de
德累斯顿工业大学，德国

Shaokai Lin
shaokai@berkeley.edu 加州大学伯克利分校，美国

Felix Suchert
felix.suchert@tu-dresden.de
德累斯顿工业大学，德国

Tassilo Tanneberger
tassilo.tanneberger@tu-dresden.de 德累斯顿工业大学，德国

Hokeun Kim
hokeun@hanyang.ac.kr 汉阳大学，韩国

Jeronimo Castrillon
jeronimo.castrillon@tu-dresden.de 德累斯顿工业大学，德国

Edward A. Lee
eal@berkeley.edu 加州大学伯克利分校，美国

arXiv:2301.02444v2 [cs.PL] 9 Jan 2023

摘要

Actor 框架及类似的响应式编程技术广泛用于构建并发系统。它们承诺高效运行，并能很好地扩展到分布式系统中的大量核心或节点。然而，它们也使程序员面临非确定性，这往往导致实现难以理解、调试和测试。近期提出的反应器模型是一种很有前景的替代方案，能够实现高效的确定性并发。本文表明，确定性既不会降低表达能力，也不会损害性能。为此，我们评估了 Lingua Franca（LF）：这是一种面向反应器的协调语言，它为主流编程语言配备了一种并发模型，可自动利用那些不会引入非确定性的并行机会。我们对 Savina 基准测试套件的实现表明，就执行时间而言，LF 程序的运行时性能甚至超过了流行且经过高度优化的 actor 框架。我们将其与 Akka 和 CAF 进行比较，LF 的性能分别优于二者 $1.86x$ 和 $1.42x$。

CCS 概念：• **计算方法 → 并发编程语言**；• **软件及其工程 →运行时环境**；**源代码生成**。

关键词：协调、并发、确定性、性能

1 引言

研究编程语言语义的理论学家早已认识到确定性的价值，以及非确定性在编程语言中的表达能力。然而在实践中，如今非确定性会悄然进入编程语言和框架，并非为了发挥其表达能力，而是因为人们普遍认为，要在并行硬件上获得良好性能，就离不开非确定性。本文表明，实现高性能并不必以牺牲确定性为代价。我们将重点考察 actor 框架；这类框架在许多要求极高的应用中广受欢迎且成效显著，但它们允许的非确定性往往并非应用实际所需。

利用多核机器等并行硬件来提升性能，只有在程序展现出并发性时才有可能。常见的并发抽象包括线程 [18], 远程过程调用 [63], 发布-订阅 [27], 面向服务的架构 [64], 和参与者 [1, 33]。这些模型各有优点，但都会引入非确定性：对于给定的状态和输入，程序的行为并非唯一确定。虽然非确定性在某些应用中很有用，但大多数编程任务都更受益于可重复的行为。确定性程序更易于理解、调试和测试（对于每个测试向量，都有一个已知的正确响应）。对于非确定性程序，问题行为可能更难发现，因为它们可能只占据状态空间 [37] 中很小的一部分。而且，重现故障可能极其困难 [40, 48, 61]

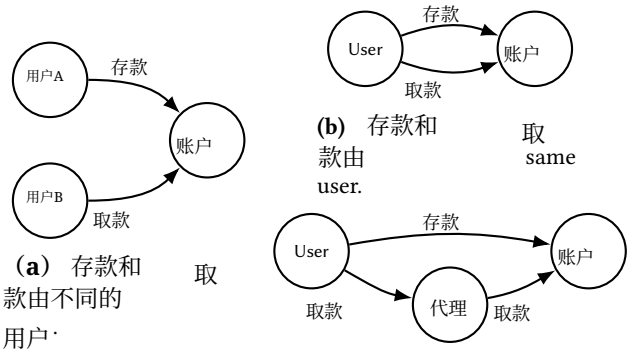
因为它们可能只会在系统处于特定负载水平时出现 [67]。

确定性是一个微妙的概念 [41]。本文关注程序确定性的一种特定形式：如果给定相同的输入，程序始终产生相同的输出，那么该程序就是确定性的。这个定义并不要求操作必须按特定顺序执行，因此与并发和并行执行并不矛盾。即使使用线程、actor 和异步远程过程调用等非确定性抽象，实现这种形式的确定性也是可能的，但往往并不容易。对于足够简单的程序，例如由一串 actor 构成的程序，只要通信可靠，执行就是确定性的。本文中用于对比的一些基准测试就具有这种确定性。不过，正如我们将要展示的，即使通信结构稍微复杂一些，也会导致非确定性，而且这种非确定性可能难以消除。

本文评估了一种用于描述并发软件的基于语言的协调方法，该方法默认保持确定性，只有在程序员明确引入非确定性时才允许其出现。协调语言 Lingua Franca（LF） [53], 基于一种称为反应器的并发计算模型 [50, 51],，通过分析程序结构并确保在运行时正确处理数据依赖来实现这一点。LF 程序定义了称为“反应器”的反应式软件组件，并提供运算符以分层（通过包含关系）和双边（通过连接）组合这些组件。由于该语言同时支持确定性和非确定性并发，因此为探索确定性对性能的影响提供了理想的环境。

LF 确定性子集的语义可以看作 actor 的一种确定性变体 [1, 33, 52]。本文表明，它在并行硬件上的性能可与流行的非确定性 actor 实现相媲美。与 Akka [66] 和 CAF [16]（本文用于比较的框架）类似，LF 会协调以常规编程语言编写的代码块的执行，让程序员能够使用自己熟悉的语言、库和工具。与用于比较的框架不同，LF 支持多种语言。目前，它支持 C、C++、Python、TypeScript 和 Rust。本文重点关注 C++目标的运行时性能。作为本文的一项核心贡献，我们对该目标进行了优化，使其能够高效地利用并行硬件上的并发。早期工作 [53]仅根据 LF 的 C 目标报告了初步的性能表现；该目标主要面向嵌入式系统。

LF 并发模型的核心是逻辑时间模型，它清晰地定义了同时性，并基于因果关系接口进行依赖分析以避免死锁 [47]。正是这种定时语义使 LF 能够实现高效的确定性并发。然而，我们用来比较的基准测试是为评估 actor 框架而创建的，



（a） 存款和取款由不同的用户·

(b) 存款和取款由user.

（c）通过代理发出的取款。

图1. 可能呈现非确定性行为的 actor 程序示例。

而这些框架没有时间语义。这些基准测试都没有利用 LF 与时间相关的特性；时间语义仅用于实现确定性。

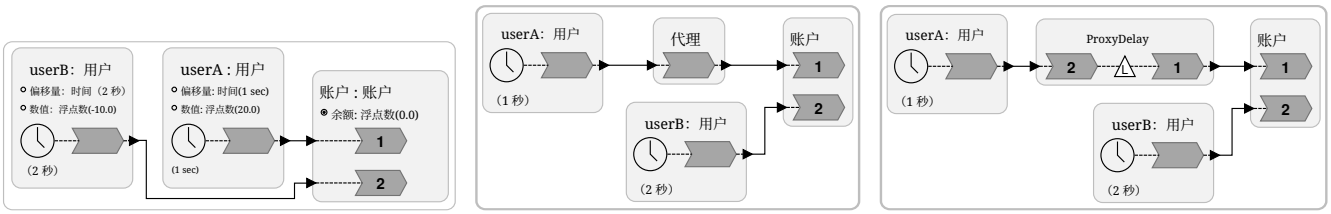
贡献。 我们表明，Lingua Franca 中实现的面向反应器范式能够在不放弃确定性的前提下，高效利用并行硬件。为此，我们解释了 LF 程序展现并发的机制；提出了一种支持定义可扩展程序的语言扩展；并介绍了一种经过优化的 LF C++ 运行时，可实现高效的并行执行。我们还基于 Savina 基准测试套件 [34],进行了广泛评估，结果表明，我们的 LF 运行时优于 Akka 和 CAF，分别高出1.86x 和 1.42x。

概要。 我们首先阐述本工作的动机（第2节），然后介绍 LF（第3节）。接着，我们详细介绍 LF 中的并发，并介绍我们优化后的 C++ 运行时（第4节）。随后，我们报告基准测试结果（第5节）、讨论相关工作（第6节），最后总结全文（第7节）。

2 动机

Actor 模型因其能够让程序员轻松表达并发、提供出色的执行性能，并良好地扩展到大型数据集和复杂应用程序，而被广泛认可并应用于生产环境。此外，与基于线程的程序相比，Actor 语义能够避免低级 数据竞争。然而，与大多数消息传递范式一样，actor 也会让程序员面临以高级 数据竞争形式出现的不确定性 [76], ；随着程序复杂性的增加，这一问题会变得更难以管理。

来看图1a中的简单示例。Account actor 管理一个由两位用户交互操作的银行账户的余额。用户A发送一条存款消息，以增加账户余额；用户B发送一条取款消息，以减少账户余额。如果我们假设



（**a**）简单的 LF 实现。 **(b)** 添加代理反应器。 **(c)** 添加引入逻辑延迟的代理。

图 2。 图1a所示 actor 程序及其变体的 LF 实现示例 .

余额初始化为 0，且只有在取款后余额不会变成负数时，账户才会批准取款，那么可能出现两种行为。如果先处理用户A的消息，则会批准用户B的取款。如果先处理用户B的消息，则会拒绝其取款。Actor 模型并未赋予消息顺序任何含义。因此，这个示例不存在明确定义的正确行为。

读者可能会反驳，对于图1a所示的应用，事务顺序本质上具有非确定性，而软件框架引入的任何额外非确定性都无关紧要。然而，如果我们关注可测试性，就会发现即使输入完全相同，也可能产生不同的结果，从而增加测试难度。如果我们关注一致性，同一事件的不同观察者可能看到不同的行为，这就会带来问题。在数据库中，通常会为外部输入分配时间戳，随后将这些时间戳视为输入的语义属性，并根据这些时间戳定义数据库的行为。本文采用这一观点，并依据 Lee [41] 给出的确定性定义：“确定性是模型的属性，而非物理实现的属性”，以及“如果给定提供给模型的所有输入，模型恰好定义一种可能的行为，那么该模型就是确定性的。”如果我们将图1a中的“输入”定义为带时间戳的用户查询，将“行为”定义为账户采取的动作序列，那么要求确定性是合理的。

请看图1b，其中只有一个用户。即使这位用户先发送存款消息，再发送取款消息，Actor 模型也无法保证接收 actor 会按此顺序看到并处理收到的消息。虽然有些 actor 框架（例如 Akka 和 Erlang）保证按序消息传递，但另一些框架（例如 AmbientTalk [77],）则明确不作此保证。然而，即使框架保证点对点按序消息传递，这一特性也不具传递性。如果我们像图1c所示添加一个代理，就无法对账户接收消息的顺序作出任何假设。这个例子进一步说明，组合 actor 可能会产生意想不到的副作用。

因此，使用 actor 实现实际并发问题的解决方案可能颇具挑战。即使是上文讨论的看似简单的并发问题，也需要遵循严格的编程规范，而且解决方案通常也难以维护，并且往往缺乏模块化。此外，actor 框架固有的非确定性使得验证此类解决方案十分困难。错误行为可能只在部分执行中出现，因此集成测试无法可靠地检测到这类“海森堡错误” [61]。

在一项近期研究中，Bagherzadeh 等人 [4] 分析了在 StackOverflow 或 GitHub 上讨论的 Akka 程序中的缺陷，并发现其中 14.6% 是由竞态导致的。因此，在 Akka 程序中，高层竞态是仅次于程序逻辑错误的第二常见缺陷原因。在一项针对 12 个基于 actor 的生产系统的类似研究中，Hedden 和 Zhao [32] 发现，报告的缺陷中有 3.2% 是由糟糕的消息排序导致的，4.8% 是由不正确的协调机制导致的，4.8% 是由关闭时协调错误导致的，2.4% 是由启动时协调错误导致的。请注意，这些数字仅涵盖他们研究项目中已知的缺陷；正如作者所指出的，报告的消息排序缺陷大多来自 Gatling 项目，因为该项目已集成名为 Bita [73] 的调试工具，专门用于检测此类缺陷。我们怀疑，不使用专用调试工具的项目中还存在更多未被发现的缺陷。

actor 社区通过引入更好的工具来分析和调试 actor 程序，解决了 actor 固有的非确定性及其导致的错误。这些工具包括 TransPDOR [72], Bita [73], Actoverse [69], iDeA [55],CauDEr [39],和多重宇宙调试 [75]。尽管这些解决方案很有价值，但我们认为，用于表达并发程序的编程模型应默认提供确定性语义，并允许程序员仅在希望引入非确定性且了解其无害之处时才这样做。在这种情况下，仍可使用上述针对非确定性行为的工具来调试实现。

实现确定性并发有多种方法，包括 Kahn 进程网络 [35, 36], 多种数据流模型 [21, 43, 62], 物理异步、逻辑同步模型 [68], 同步反应式语言 [8, 24],以及离散事件系统 [15, 23, 46, 79]。Lohstroh 等人 [53]将反应器模型与这些方法逐一比较，指出该模型兼具它们的许多优点，且

缺陷更少。Lingua Franca 基于这一反应器模型构建，因为它比某些替代方案（例如 Kahn 网络）表达能力更强，而且在风格上接近 actor 模型，而 actor 模型已被证明在实践中行之有效。本文表明，由此实现的确定性不会带来性能损失，反而在大多数情况下有助于提升性能。

3 Lingua Franca 简介

Lingua Franca（LF）基于一种相对较新的面向反应器的编程范式。直观地说，我们可以将反应器描述为具有离散事件执行语义、并显式声明端口和连接的确定性 actor。程序使用逻辑时间线对事件排序，并确保执行具有确定性。作为一种多语言编程语言，LF 会纳入目标编程语言的代码，以实现各组件的逻辑。LF 本身只关注程序的协调方面。

本节将介绍反应器和 LF 的核心概念。不过请注意，全面讨论 LF（包括其语法和工具链）超出了本文的范围。我们将以 LF 自动根据 LF 源代码合成的程序图示表示为基础展开讨论 [78]。Lohstroh 等人 [53]对 LF 的文本语法作了完整介绍。

3.1 LF 示例

图1a 中存款/取款示例的 LF 实现如图 2a 所示。该程序由三个 **反应器实例**userA、userB 和 account 组成，在图中显示为浅灰色方框。请注意，userA 和 userB 是同一个 **反应器类** 用户的实例，因此具有相同的结构和功能。图中，黑色三角形表示**端口**。在此示例中，两个用户都具有一个输出端口，分别连接到账户的一个输入端口。这些端口和连接使用户能够向账户发送请求。

在 LF 中，所有计算都在称为**反应**的响应式代码段中执行，这些代码段使用任意目标语言实现。图中，反应用深灰色的箭头形符号表示。所有反应都必须显式声明其触发器、其他依赖项以及潜在副作用。在图 2a 的示例中，两个用户都定义了一个由**计时器**触发的反应。计时器是 LF 的一种构造，用于按固定间隔或在特定时间产生事件。userA 的计时器配置为在程序启动后一秒触发事件；userB 的计时器则配置为在程序启动后两秒触发事件。相应的反应只需设置用户的输出端口，即可发送存款或取款请求。

Account 反应器定义了两个反应，分别对应它的两个输入。这两个反应都只会尝试将请求的变更应用到余额上，余额存储在一个状态变量中，该变量属于反应器实例。请注意，反应器可以定义任意状态变量，供其自身的所有反应访问（但不包括所包含反应器的反应）。除了状态变量之外，反应器还可以定义可在实例化时设置的参数。借助这一机制，用户反应器可以重复使用，因为计时器触发的确切时间（偏移量）以及要取出或存入的金额（数值）都可以在实例化时配置。

读者可能会注意到，账户中的分离反应逻辑重复，并不是切实可行的解决方案，尤其是在用户众多的情况下。我们选择这种表示方式，是为了让讲解更简单，并避免详细讨论 LF 语法。实际上，在 LF 中，单个反应可以绑定到任意数量的上游端口。

程序执行时，会等待 1 秒，然后触发 userA 反应器的计时器并调用其反应。该反应产生的事件将触发账户中的反应 1；在第一个反应完成后，反应 1 会立即被调用。程序启动两秒后，userB 将执行反应，随后触发账户中的反应 2。在此示例中，存款事件（+20.0）早于取款事件（-10.0）发生，因此我们的执行语义会确保账户先处理存款事件，再处理取款事件，这意味着余额不会变成负数。在更贴近实际的实现中，两个用户会零星地产生事件，并且其反应会由物理动作而非计时器触发（见第 3.3 节）。不过，使用计时器能让讲解大为简化，因为我们只需考虑一条对事件进行排序的逻辑时间线 。此外，还可以利用此类计时器创建回归测试，用特定的输入时序验证程序执行。

请注意，即使两个事件在逻辑上同时发生，也就是说，Account 反应器中的两个反应都在同一逻辑时间被触发，最终程序仍将具有确定性。同一标签内的所有反应都按照明确定义的优先关系执行。具体而言，同一反应器内的任何反应都互斥，并按照 LF 代码中反应的词法声明顺序执行。图 2 中的示意图也通过反应旁显示的数字体现了这一顺序。关于反应优先关系的更多细节，请参见第 4.1 节。

若要有意改变事件发生的顺序，可以在程序中使用**逻辑动作**引入逻辑延迟，如图 2c 所示。在图中，动作以小白色三角形表示。与端口不同，端口允许事件在不同反应器之间以逻辑上的瞬时方式转发，而逻辑动作则提供了一种机制，用于在稍后的（逻辑）时间调度新事件。收到输入后，ProxyDelay 反应器的反应 2 会被触发，并按可配置的延迟调度其逻辑动作

。这会创建一个新事件；处理该事件时，将触发 ProxyDelay 反应器的反应 1，后者会取回原始数值并将其转发到输出端口。

如果我们假设调度逻辑动作时采用 2 秒的延迟，那么 userA 发出的存款消息将在启动 3 秒后才到达账户。因此，存款消息会在 userB 的取款消息之后 处理，导致 B 的请求被拒绝。

值得注意的是，无论反应的物理执行时间如何，文中讨论的所有示例都是确定性的，因为所有事件都沿着一条逻辑时间线明确排序。另一方面，事件的物理时序只是近似的。本文的贡献在于证明，这种确定性并不一定会降低性能，而且对无需显式定时的应用也很有用。

3.2 逻辑时间与物理时间

所有事件都关联一个**标签**，运行时会用它在逻辑时间线上对事件进行排序。在对时间敏感的应用中，标签不仅用于逻辑排序，还与物理时间相关联。默认情况下，只有当当前物理时间 T 大于标签 t 的时间值$T > t$时，运行时才会处理与该标签关联的事件。我们称逻辑时间在“追赶”物理时间。反应器模型中物理时间与逻辑时间之间的关系，赋予了逻辑延迟有用的语义，也使得截止时间得以定义。这种定时语义对于运行于信息物理系统中的软件尤其有用。若想更深入地了解 LF 的定时语义，感兴趣的读者可参阅 [54]。

如果应用不需要任何物理时间属性，就可以关闭物理时间与逻辑时间的并行；此时，标签仅用于保持确定性，而不用于控制时序。此外，LF 程序员无需显式控制其程序的时序方面。例如，调度动作时可以省略延迟，此时运行时将使用下一个可用标签。因此，即使是无定时的一般用途程序，也能受益于 LF 的定时语义所实现的确定性并发。

3.3 异步性与刻意的非确定性

反应器模型区分逻辑动作和物理动作。逻辑动作总是相对于当前标签，以同步方式调度。**物理动作**可以从异步上下文中调度；其事件的逻辑时间根据当前读取的物理时间确定。物理动作是处理源自物理过程的偶发输入的关键机制（例如用户发起取款或存款请求）。

将标签分配给物理动作的过程具有非确定性，因为它并非由程序定义。

然而，一旦这些标签被分配，例如分配给用户的存款或取款请求，事件的处理就是确定性的，并且按照标签顺序进行。因此，分配给外部发起事件的标签被视为输入的一部分；给定这一输入，程序仍保持确定性。这种方法为确定性的、因而可测试的程序逻辑划定了清晰的边界，同时允许它与零星的外部输入交互。

物理动作也可以在程序本身中使用，例如，为从另一个反应器接收的消息非确定性地分配一个新标签。在这种用法中，物理动作提供了一种有意将类似 actor 的非确定性引入程序的方式。

4 高效的确定性并发

LF 程序默认具有确定性。这一特性继承自 LF 所实现的反应器模型。Lohstroh 等人 [53] 解释了反应器为何表现出确定性。他们的论证可以适用于 Lingua Franca 语言的具体语境，但这超出了本文的讨论范围。反应器也支持并发；正如本文所展示的，所暴露的并发足以让运行时系统有效利用多核硬件，其性能可与根本上采用异步和非确定性设计的 actor 框架相匹敌，甚至更胜一筹。本节首先具体说明并发是如何暴露出来的，然后更深入地介绍我们的 C++运行时的实现方式，以及它如何利用并行硬件。

4.1 并行性

使用静态声明的端口和连接作为反应器之间的接口，并声明反应依赖关系，这使反应器区别于 actor 或其他更具动态性的模型；在这些模型中，通信完全基于地址。反应器程序的固定拓扑灵活性较低，也限制了运行时适应，但同时带来了两项关键优势。首先，它实现了组件功能与组件组合的关注点分离。其次，它在接口层面明确指出了组件之间存在的依赖关系。因此，可以为反应器的任意组合推导出依赖图。依赖图是一个**无环优先关系图（APG）**，它将所有反应组织成偏序，刻画了为确保反应器程序的执行产生确定性结果而必须遵守的所有调度约束。由于无论触发反应时执行的代码内容如何，该图都始终有效，因此可以将反应视为黑盒。正是这一特性使 LF 能够支持多种编程语言，并揭示应用程序中的并发性。

图 3 展示了图 2c 所示程序的依赖图。实线箭头表示因一个反应（可能）向另一个反应发送数据而产生的依赖项

。虚线箭头表示因两个反应属于同一反应器而产生的依赖项。与 actor 的行为类似，同一反应器中的反应互斥。执行顺序定义明确，由 LF 代码中反应的词法声明顺序决定。图 2 中反应标签里的数字也标示了这一顺序。

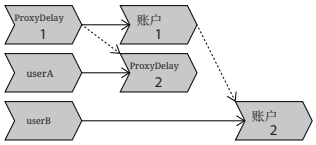


图 3。 图 2c 中程序的反应图。

依赖图精确规定了反应的执行顺序。独立的反应可以并行执行，而不会破坏确定性。例如，图 3 中的 APG 表明，ProxyDelay 的反应 1 以及 userA 和 userB 的反应都可以并行执行。请注意，依赖图必须是无环的，因为任何环都会违反因果关系。LF 编译器确保有效的程序具有无环的依赖图。LF 程序中的任何依赖环都可以通过引入一个逻辑动作，并用它在未来标签处安排一个新事件来解决。

由于 LF 中的所有依赖关系都是静态声明的，因此与 actor 和类似模型相比，LF 缺乏运行时灵活性。反应器模型通过**变异**弥补了这一点，这些变异支持在运行时调整反应器拓扑及其隐含的依赖图。然而，LF 尚未完全实现变异，而讨论变异也超出了本文的范围。

4.2 可扩展的连接模式

对于较大的程序，逐个创建反应器实例、端口和连接会变得十分繁琐。为解决这一问题，我们扩展了 LF 语法，使其能够一次创建多个端口或反应器实例。此外，我们重载了 LF 的连接运算符，以便一次通过多端口和组创建多个连接。借助这一机制，只需一行代码即可实现各种复杂的连接模式；由于它完全可参数化，LF 程序无需重新编译即可灵活扩展，以适应给定的问题规模。

考虑一个简单的 LF 分叉-汇合程序：[1]

```
1 反应器 Src(w: 整数(3)) { 2 输出[w] out: 整数 3 } 4 反应器 Worker {
5 输
(w:
wr
w}
.输入17 }
```

该程序定义了一个Src、一个Worker和一个Sink反应器，以及一个未命名的main反应器，用于组装该程序。Worker定义了两个类型为整数的单端口，分别名为输入和输出。Src 和Sink 都使用我们的语法扩展，各自定义了一个宽度为w的**多端口**，其中w 是参数，默认值为 3。主反应器会创建所有反应器实例和连接。具体而言，它会创建 Src和Sink的两个单独实例，并使用我们的语法扩展创建一个宽度为w 的工作线程反应器**银行**（第 14 行）。两条连接语句（第 15、16 行）分别建立w 个连接，对应多端口与银行实例的每一对。由此形成的连接模式如图 4a 所示。请注意，工作线程的确切数量可通过主反应器的w 参数进行配置；执行程序时即可指定该参数，无需重新编译。因此，该程序可配置为使用任意数量的工作线程。

在此示例中，源反应器会生成三个彼此独立的数值并发送给工作线程。相反，如果我们想将单个数值广播给所有工作线程，则可以使用广播语法 `(...)+`。将源反应器配置为使用单个输出（在第 12 行设置 `w=1` ），并将第 15 行改为 `(src.out)+ -> wrk.in`，即可创建图 4b 所示的模式。

另一种常见模式是级联组合，它可以方便地用 LF 语法表示，如以下程序所示：

```
1 主反应器(n: 整数(2)) {2 src = new Src(w = 1)
3
W
```

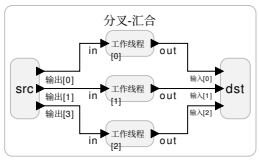
连接运算符会对左侧和右侧列出的所有端口进行排序，并将左侧第 *n*个端口连接到右侧第 *n*个端口。只需将第 5 行连接语句左侧偏移一个源端口，并在右侧追加 Sink 端口，就能有效地安排这些连接，形成图 4c 所示的级联。

连接运算符还会连接组内的多端口。在这种情况下，运算符会隐式展开连接两侧的所有端口实例，形成一个扁平的端口列表。展开时，先列出第一个组实例的所有端口，再列出第二个实例的所有端口，依此类推。请看以下程序：

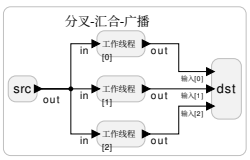
```
1 reactor Node(w: int(3)) { 5 main reactor(w: int(3)) {
2   input[w] in: int        6   node = new[w] Node(w=w)
3   output[w] out: int      7   node.out -> node.in
4 }                         8 }
```

这将生成图 4d 所示的模式，但它不太实用。在第 7 行连接的任意一端使用 交错 修饰符，我们便可以修改展开

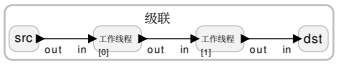
[1]为简洁起见，省略了实现细节。有关 LF 语法的介绍，请参阅 [53]。



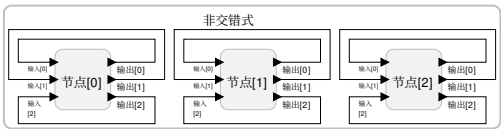
(a) 多端口连接



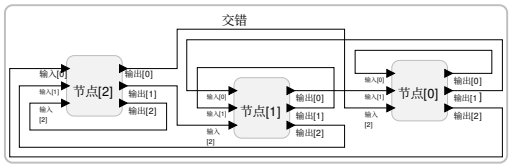
(b) 广播连接



(c) 级联组合



(d)银行内的多端口：直接连接



(e)银行内的多端口：交错连接

图 4。Lingua Franca 中的各种连接模式

策略：先列出所有银行实例中的所有第一个端口实例，然后列出所有银行实例中的第二个端口实例，依此类推。这样便会生成图 4e 所示的全连接模式。

本节讨论的所有模式都广泛应用于第 5 节的基准测试实现中。

4.3 运行时实现

每个 LF 程序的执行都由一个运行时控制。最重要的是，运行时包含一个调度器，用于跟踪所有已调度的未来事件、控制逻辑时间的推进，并按照依赖图指定的顺序调用任何已触发的反应，同时力求尽可能发挥并行性。Lohstroh 等人已经为反应器程序 [51]概述了一种简单的调度算法。在本节中，我们将介绍该调度算法的 C++ 实现，旨在发挥并行性，同时将同步开销降至最低，并避免对共享资源的争用。

图 5 概述了我们的运行时所使用的调度机制。调度器会跟踪未来事件，并严格按照标签顺序处理这些事件。处理事件时，调度器首先确定由该事件触发的所有反应，并将其存入事件队列 和反应队列。对于反应队列中所有依赖项均已满足的反应（由 APG 指示），调度器会将其转发到就绪队列，然后由工作线程取出执行。如果执行的反应通过设置端口触发了其他反应，这些反应也会被插入反应队列。如果反应

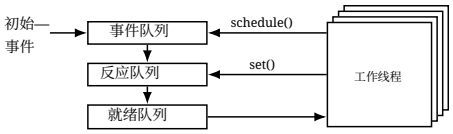


图 5。LF 运行时中的调度机制。

通过动作安排了未来事件，这些新事件就会被插入事件队列。请注意，调度器始终会等待当前标签下的所有反应处理完毕，然后才推进到下一个标签并触发新的反应。

调度器最重要的任务是决定何时将任意给定的反应从反应队列移至就绪队列。由于 APG 精确地定义了反应的顺序约束，反应调度与基于 DAG 的调度策略 [2, 38] 密切相关。然而，APG 并不等同于任务图，因为其中可能包含无需执行的反应。通常，在特定标签触发的反应仅占一小部分。更多此外，对于给定的标签，我们无法预先准确知道哪些反应会被触发，因为反应可能会也可能不会通过其声明的端口发送消息。因此，无法预先计算出最优调度。

要判断某个已触发的反应是否已准备好执行，我们需要检查它是否依赖于其他任何已触发或正在执行的反应。为避免在运行时遍历 APG，我们采用一种简单的启发式方法。具体而言，我们为每个反应分配一个层级（即 [38] 中定义的顶层）。处于同一层级的反应彼此没有依赖关系，因此可以并行执行。随后，我们的调度器按层级顺序处理反应。某一层级中的所有反应处理完毕后，下一层级中所有已触发的反应都会移入就绪队列。这种方法避免了在执行期间分析 APG，但也未能充分利用所有并行执行机会。例如，该方法不会让 ProxyDelay 的反应 2 与 Account 的反应 2 并行执行。尽管如此，我们的评估表明，在大多数情况下，这种策略足以高效利用并行性。鉴于 DAG 调度方面已有大量研究，我们相信未来可以采用更复杂的策略，将错失的并行机会也纳入考虑。

我们的调度方法还有一项局限：调度器只会执行在同一标签下触发的反应。具体而言，对于不使用逻辑延迟来创建独立流水线阶段的程序，这可能会妨碍对流水线并行的利用。不过，正如 Bateni 等人所讨论的，可以通过采用联邦执行策略来克服这一局限。 [7]

虽然 [51] 中概述并在本节中进一步完善的调度算法实现起来相对简单，但要获得有竞争力的性能，还需要进一步优化。下面我们将详细介绍 C++ 运行时所采用的几项最重要的优化。

协调工作线程。 在上述讨论中，我们从概念上将调度器与工作线程区分开来。然而，在实际实现中，使用中央调度器和独立的工作线程会带来显著的同步开销。因此，在我们的实现中，任何工作线程都可以成为调度器，将就绪的反应移入就绪队列；如果所有反应均已处理完毕，还可以将逻辑时间推进到下一个标签。此外，我们利用这样一个事实：在任意时刻，我们都知道需要并行执行的反应数量，并使用计数信号量来控制活动工作线程的数量。

无锁数据结构。用于簿记（例如，记录所有已设置端口的列表）的三个队列和其他数据结构由多个工作线程共享。由于共享资源上的争用严重，使用互斥锁进行同步被证明效率低下，尤其是在许多并行反应设置端口或调度动作时。因此，在可行的情况下，我们采用无锁数据结构。例如，就绪队列采用固定大小的缓冲区，并配有一个原子计数器。由于我们确切知道最多可以并行运行多少个反应（即 APG 中处于同一层级的反应数量上限），因此可以固定队列的大小。每当有新的反应被移入反应队列时，原子计数器就会被设为队列中的反应数量。每当工作线程尝试执行一个反应时，就会以原子方式递减该计数器。如果结果为负数，则队列为空；否则，该结果会给出要从缓冲区读取的索引。

我们进一步利用有关反应器程序执行过程的知识。例如，只有在所有反应都处理完毕后，调度器才会推进逻辑时间。由于所有工作线程都在等待新的反应，因此这一操作无需额外同步即可安全执行。

稀疏多端口。 在多个反应器中的反应可能触发同一反应的程序中（例如，包含任意数量用户的账户），被触发的反应通常需要知道哪些端口实际已设置（即包含数据）。如果上游反应器很多而通信稀疏，逐一检查所有端口是否有数据可能效率低下。为此，我们提供了一个应用程序接口，用于获取仅包含已设置端口的迭代器。请注意，actor 中不会出现这个问题，因为其中不存在端口，消息会逐条处理，且只考虑实际发送的消息。

5 性能评估

Actor 模型被广泛用于编写大型并发应用程序，而 C++ Actor Framework (CAF) [16] 和 Akka [66] 等实现因能够高效利用更多线程而闻名。与 actor 相比，LF 施加了各种限制，形成了一种允许更少行为的计算模型。在本节中，我们将展示这些限制并不一定会带来额外开销或更长的执行时间。事实上，LF 在许多基准测试中明显更快。

5.1 方法论

我们的评估基于面向 actor 语言和框架的 Savina 基准测试套件 [34]。尽管该套件存在若干问题，正如 Blessing 等人在 [13],中所讨论的那样，Savina 涵盖了广泛的模式；据我们所知，它是已发表的最全面的 actor 框架基准测试套件。Savina 基准测试套件包含所有基准测试的 Akka 实现。大多数 Savina 基准测试也提供 CAF 实现。[2]

我们将 30 个 Savina 基准测试中的 22 个移植到了 LF 的 C++ 目标。[3] 由于 actor 模型与反应器模型之间存在根本差异，移植基准测试的过程并不总是直截了当。我们的目标是尽可能贴近原始工作负载，并考虑各个基准测试背后的意图。我们没有实现 Fork Join（Actor 创建）、斐波那契、快速排序、双调排序、埃拉托斯特尼筛法、不平衡蛛网树、在线设施选址和逐次超松弛基准测试，因为它们要求具备动态创建 actor 的能力。在反应器模型中，这可以通过能够修改反应器拓扑的变异来实现 [50, 51]。然而，LF 尚未完全实现变异，而关于支持变异的语言级构造的讨论超出了本文的范围。

此外，我们在本文中省略了 A* 搜索和逻辑斯蒂映射序列基准测试。A* 搜索存在严重的竞态条件，导致执行时间大幅波动 [13]。逻辑斯蒂映射序列因违反 actor 语义并且需要显式同步而被省略 [13]。因此，CAF 实现需要使用阻塞调用，这使其速度至少比其他实现慢两个数量级。由于这并非 CAF 本身的问题，而是基准测试设计的问题，我们省略逻辑斯蒂映射序列，以避免使分析结果产生偏差。

图 6 展示了使用 Akka、CAF 和 LF 的 C++ 目标对所有受支持的基准测试进行测量后得到的结果。图中显示了每项基准测试在不同工作线程数量下的平均执行时间（包括 99% 置信区间）。CAF 并未实现所有基准测试，因此部分图中没有 CAF 的数据。

所有测量均在一台工作站上进行，该工作站配备 Intel Core i9-10900K 处理器和 32 GiB DDR4-2933 内存，运行 Ubuntu 22.04，并使用 CAF 版本 17.6 和 Akka 版本 2.6.17。按照 Savina 的方法论，测量不包括初始化和清理。每次测量包含 32 次迭代。前两次迭代

[2] https://github.com/woelke/savina [3] 源代码可在 https://github.com/lf-lang/benchmarks-lingua- franca 获取

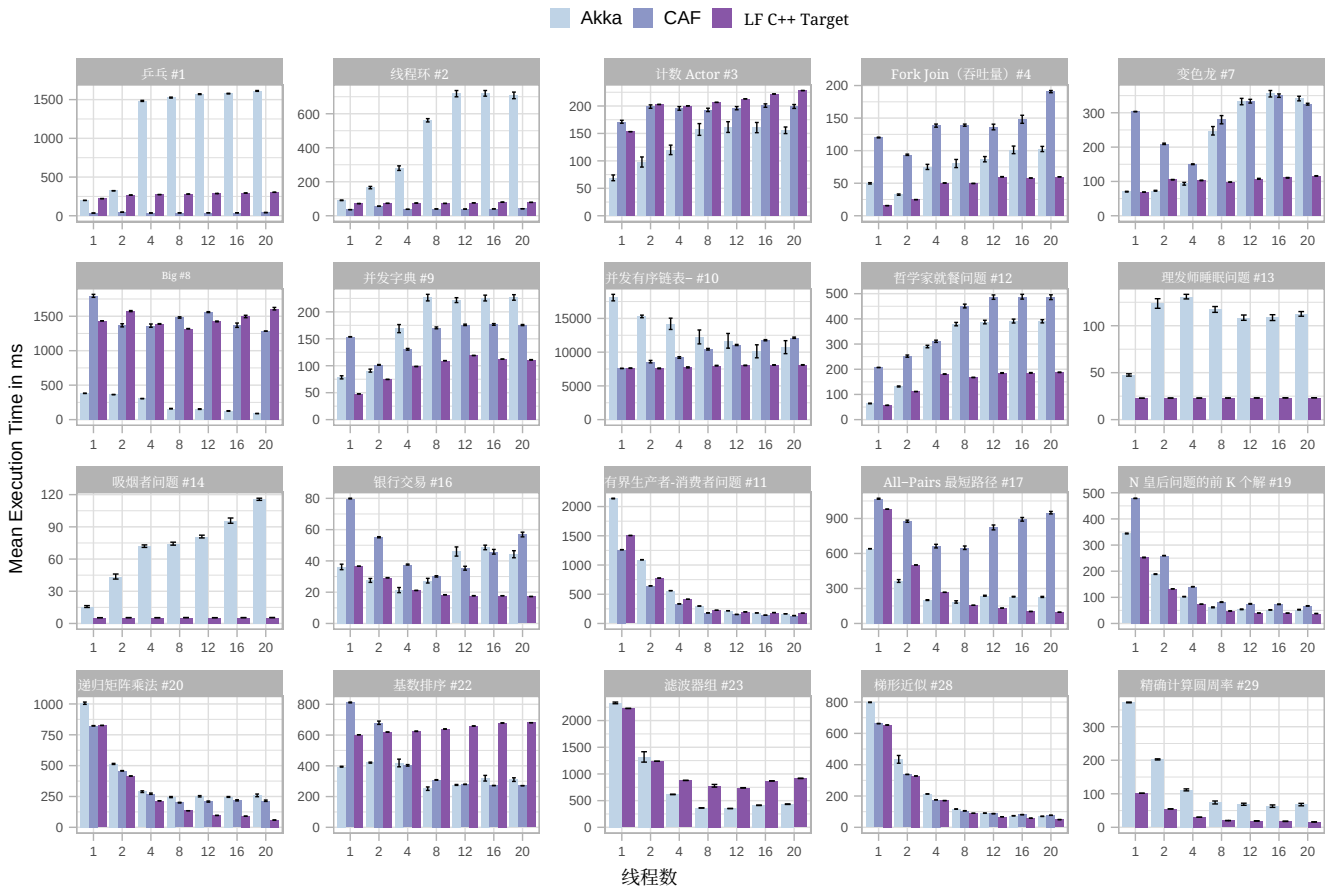


图 6. 在不同工作线程数量下测得的以 LF、CAF 和 Akka 实现的各类 Savinabenchmarks 的平均执行时间和 99% 置信区间。带有 # 前缀的数字是基准测试编号，详见 [34]。

不纳入我们的分析，而是用于预热缓存和 Java 虚拟机（针对 Akka）。

5.2 讨论

图 6 中的前六张图属于 Savina 基准测试套件中的微基准测试。这些测试旨在揭示用于交换消息和进行调度的协议开销。总体而言，我们的 C++运行时性能与 Akka 和 CAF 相当。在乒乓和线程环基准测试中，我们的实现明显快于 Akka，但仍不及 CAF。在计数 Actor 和 Big 基准测试中，Akka 表现更好，而 LF 的性能略逊于 CAF。在 Fork Join 和变色龙基准测试中，LF 实现优于 Akka 和 CAF，尤其是在使用更多工作线程时。

接下来的六张图（从并发字典到银行交易）属于并发基准测试。LF 在所有并发基准测试中都显著优于 CAF 和 Akka（尤其是工作线程数量较多时）。这凸显了在 LF 中，并发行为可以自然地表达，并且能够高效执行。另一方面，这些基准测试的 Actor 实现必须显式进行同步，并采用可能开销很大的协议（例如，通过发送确认消息），或者实现其他会违反 Actor 语义的（阻塞式）同步策略 [13]。

其余图表属于 Savina 基准测试套件[4]中的并行基准测试。正如第 4.3 节所述，基数排序和滤波器组在一定程度上受我们调度器效率低下的影响。在这些特定基准测试中，我们的简单算法导致执行并非最优，因为某些反应的执行时间晚于可能的时间。我们将在未来工作中修订这一算法。不过，其余并行基准测试表明，LF 能够高效地实现并行算法。我们的 LF 实现与 Akka 和 CAF 不相上下，并且随着线程数增加，扩展性良好。

平均而言，LF 的性能优于 Akka 和 CAF。使用 20 个线程时，C++ 运行时的速度比 Akka 快 $1.85x$，比 CAF 快 $1.42x$。这些加速比是根据各项基准测试的加速比几何平均数计算得出的。我们得出结论，LF 能够与现代且经过高度优化的

[4]生产者-消费者实际上被列为并发基准测试，但我们认为它更适合归入并行基准测试。

actor 框架（如 Akka 和 CAF）相媲美，甚至胜过它们。尤其是在需要同步的工作负载下，LF 的性能显著优于 Actor 实现。LF 在利用并行性方面与 actor 框架同样高效，并且能够很好地扩展到更多线程。总之，LF 提供的确定性并发不会损害性能，反而能实现更高效的实现。这在一定程度上得益于调度器能够洞察程序结构；与 Savina 的许多 actor 基准测试不同，LF 可以避免显式同步。

C++ 与 Scala（Akka）之间的性能比较需要谨慎看待，因为库实现的差异以及 Java 虚拟机的行为等其他因素都可能影响性能。例如，在 Pi Precision 基准测试中，Akka 与我们的实现之间存在巨大差距，这是因为 Scala/Java 对大数的表示效率较低。不过，Savina 基准测试套件中的其他基准测试不依赖外部库，设计上也更便于在不同语言之间移植。还需注意，在所有基准测试中，针对 20 个线程，CAF 相较于 Akka 的平均加速比仅为 $1.09x$，且在 16 项基准测试中的 9 项表现不如 Akka。对于单线程执行，Akka 在 10 项基准测试中优于 CAF，平均加速比达到 1.33 倍。这表明，已实现的 Scala 工作负载与 C++实现具有可比性。即使考虑到 Java 虚拟机可能造成的偏差，我们的结果仍清楚表明，LF 能够与研究现状中的 actor 框架相竞争。

为了更好地了解第 4.3 节所讨论的优化所产生的影响，图 7 还展示了我们的运行时在 20 个工作线程下，相较于优化程度较低的运行时所获得的加速比。作为基线的是我们较早版本的运行时。它也经过了一定优化：我们通过代码剖析找出明显的瓶颈，并采用常见的代码优化技术消除了这些瓶颈；但它不包含本文讨论的优化。我们的优化带来的整体平均加速比（几何平均数）为 $2.18x$。具体而言，Big 和 Bank Transaction 从我们针对稀疏通信模式所做的优化中获益显著。并发基准测试（例如并发字典和哲学家就餐问题）的性能提升，大多归功于共享资源争用的减少。

然而，并非所有基准测试都能从我们的优化中受益。乒乓和计数 Actor 的性能下降表明，为了实现高效的并行执行而进行优化，也会给简单的顺序程序带来代价。

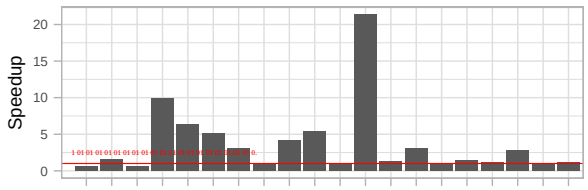


图 7. 与未优化版本相比，我们优化后的 C++运行时在使用 20 个工作线程时所实现的加速比。

6 相关工作

LF 与围绕 Hewitt 的 Actor 模型发展起来的语言和框架密切相关 [1, 33], , 包括 Akka [66],、CAF [16], 、Ray [60], 、Erlang [3], 、Rebeca [70], 、P [22] 和 Pony[17]。此外，响应式编程技术 [5], （如 ReactiveX [56]和 Reactors.IO [65] 等框架所采用的技术）也与 LF 密切相关，但事件循环等语言级构造 [74], 也与 LF 密切相关。虽然 actor 和响应式编程具有良好的弹性和可扩展性，但代价是非确定性，这使得程序出了名地难以测试和调试 [6, 75]。如果语言、框架和库不强制遵循底层模型，反而让程序员有机会破坏其语义 [71], 还会引发更多问题。Pony 通过采用类似 Rust 的强类型系统，在编译时防止数据竞争，从而解决了后一个问题。Rebeca 提供了一套形式化方法和模型检验技术，用于分析和验证参与者网络。虽然这可以增强人们对实现正确性的信心，但寻找正确实现的责任仍在程序员身上。P 更进一步，它还配备了高效的运行时系统和编译器，能够生成正确性由构造保证且性能合理的代码。

Blessing 等人提出了一种策略，将 actor 间的通信映射到树形拓扑，以保证消息的因果顺序 [12]。采用类似的方法，Sang 等人利用有向无环图拓扑实现事件处理的可串行化性。Orleans [14] 也基于类似 actor 的模型，并为事务提供原子性保证。最后，Field 等人定义的反应器 [28] 是一种与 actor 密切相关的模型，同时支持同步通信和异步通信，并提供原子性保证。所有这些策略在分布式场景中都很有用，尤其是在存在网络故障的情况下。不过，本文关注的是单主机上的执行。此外，LF 所提供的确定性保证更强。尽管如此，这些技术与 LF 密切相关，可用于确保分布式 LF 程序的容错执行。不过，我们认为 LF 提供了更通用的解决方案，因为程序员可以在分布式环境中明确权衡一致性与可用性 [42],，因此可以根据具体的应用需求调整解决方案。

数据流模型 [11, 21, 44] 和进程网络 [35,45]通过创建具有确定性语义的 actor 静态连接网络，提供确定性并发。

这些模型有助于改进静态分析和优化 [30], , 但会限制应用程序的灵活性以及对外部事件作出响应的能力。Ohua [26] 是另一种通过数据流提供并行性的语言，与 LF 类似，它能够与现有的高级语言集成。然而，它未能为各个节点提供协调机制，也不提供定时语义。

确定性并发也见于 Esterel [10],Lustre [31],和 SIGNAL [9]等同步语言，以及 Fran [25],FrTime [19], 和 Elm [20]等函数式响应式编程（FRP）语言。然而，这些语言难以用于通用编程，因为它们要求使用纯函数, 而且缺乏适用范围广泛的库。直到最近，才有人在 Esterel [29]的形式语义以及分布式数据流 [59]中考虑副作用。在 Lingua Franca 中，可以将任意代码嵌入反应中，并从流行通用语言的现有库中受益。

另一种有趣的方法是使用 DThreads [49] 或 Conse- quence [57], 等确定性多线程库，它们会为并发存储操作强制规定一个全序。近期研究在避免传统确定性多线程技术 [58]的瓶颈方面取得了显著进展。然而，我们认为，基于 [40]中概述的原因，线程并不是一种方便的并发模型。此外，线程无法实现透明的分布式执行, 而（重新）actor 则可以。

7 结论

与用于异步并发的 actor 及相关模型不同，LF 默认强制保证确定性，只有在刻意引入时才会呈现异步行为。基于 LF 的 C++ 目标平台所做的评估表明，这种确定性模型不会妨碍性能。相反，与 Akka 相比，我们的平均加速比为 $1.85x$，与 CAF 相比则为 $1.42x$。借助 LF, 我们成功兼顾了可复现（且可测试）的行为与良好的性能。不过，我们相对简单的调度策略可能仍有很大的改进空间。未来工作将探索更先进的调度算法，以便在运行时充分利用更多并行性。我们还计划提供对变异的完整运行时支持，并实现其余需要变异的 Savina 基准测试。最后需要指出的是，我们实现的 Savina 基准测试套件不仅可用于将 LF 与面向参与者的框架进行比较, 也表明 LF 虽然仍处于起步阶段，但已经适用于解决实际问题。

致谢

本文的研究部分得到了德国联邦教育与研究部（BMBF）的资助, 资助项目包括 Software Campus（01IS12051）以及“Souverän. Digital. Vernetzt.”计划中的联合项目 6G-life（16KISK001K）。

这项研究还获得了美国国家科学基金会（NSF）项目资助（编号 #CNS-1836601，项目“协调安全与互联网”）、由电装、福特、西门子和丰田支持的 iCyPhy研究中心（工业信息物理系统）以及韩国国家研究基金会（NRF）的资助（编号 NRF-2022R1F1A1065201）。

参考文献

[1] Gul A. Agha、Ian A. Mason、Scott F. Smith 和 Carolyn L. Talcott。1997年。演员计算基础。《函数式编程杂志》7，1（1997年），1–72。[2] Mahfooz Alam、As If 和 Ankur Varshney。2018年。同构与异构系统的动态调度算法综述。

73 83 10 1007 978 981 10 8533 8 – 。https://doi.org/ 。/ - - [3] - - Joe Armstrong 、 Robert Virding 、 Claes Wikström 和 Mike Williams。1996年。*Erlang*并发编程（第二版）。Prentice Hall。[4] Mehdi Bagherzadeh、Nicholas Fireman、Anas Shawesh 和 Raffi Khatchadourian。2020年。Actor并发错误：对症状、根本原因、应用程序接口用法及差异的全面研究。《*ACM*程序设计语言》4，OOPSLA，文章214（2020年11月），32页。https://doi.org/10.1145/3428282[5] Engineer Bainomugisha、Andoni Lombide Carreton、Tom van Cutsem、Stijn Mostinckx 和 Wolfgang Meuter。2013年。响应式编程综述。《*ACM*计算机调查》45，4，文章52（2013年8月），34页。https://doi.org/ 10 1145 2501654 2501666 。/ 。[6] Herman Banken、Erik Meijer 和 Georgios Gousios。2018年。调试响应式程序中的数据流。载于*2018*年*IEEE/ACM*第*40*届软件工程国际会议（*ICSE*）。IEEE，752–763。[7] Soroush Bateni、Marten Lohstroh、Hou Seng Wong、Rohan Tabish、Hokeun Kim、Shaokai Lin、Christian Menard、Cong Liu 和 Edward A. Lee。2022年。Xronos：面向安全关键分布式嵌入式系统的可预测协调。https://doi.org/10.48550/ARXIV.2207.09555[8] Albert Benveniste 和 Gérard Berry。1991年。响应式与实时系统的同步方法。《*IEEE*论文集》79，9（1991年），1270–1282。[9] Albert Benveniste 和 Paul Le Guernic。1990年。混合动力系统理论与SIGNAL语言。《*IEEE*自动控制汇刊》35，5（1990年），525–546。[10] Gérard Berry 和 Georges Gonthier。1992年。Esterel同步编程语言：设计、语义与实现。《计算机编程科学》19，2（1992年），87–152。http://citeseerx.ist.psu.edu/viewdoc/summary?doi =10 1 1 17 5606 。 。 。 。G. Bilsen、M. Engels、R. Lauwereins 和 J. A. Peperstraete。1994年。多速率与周期静态DSP应用的静态调度。载于超大规模集成电路信号处理研讨会。IEEE出版社。[11] [12] Sebastian Blessing、Sylvan Clebsch 和 Sophia Drossopoulou。2017年。因果消息传递的树形拓扑。载于第*7*届*ACM SIGPLAN*基于*Actor*、智能体和去中心化控制编程国际研讨会论文集（加拿大不列颠哥伦比亚省温哥华）*(AGERE2017)*。计算机协会，纽约州纽约市，美国，

1 10 10 1145 3141834 3141835 – 。https://doi.org/ 。/ [13] 。Sebastian Blessing 、 Kiko Fernandez-Reyes 、 Albert Mingkun Yang、Sophia Drossopoulou 和 Tobias Wrigstad。2019年。运行吧，Actor，运行吧：迈向跨演员语言基准测试。载于第*9*届*ACM SIGPLAN*基于*Actor*、智能体和去中心化控制编程国际研讨会论文集（希腊雅典）*(AGERE 2019)*。计算机协会，纽约州纽约市，美国，

41 50 10 1145 3358499 3361224 – 。https://doi.org/ 。/ 。[14] Sergey Bykov、Alan Geller、Gabriel Kliot、James R. Larus、Ravi Pandya 和 Jorgen Thelin。2011年。Orleans：人人可用的云计算。载于第*2*届*ACM*云计算研讨会论文集（葡萄牙卡斯凯什）*(SOCC'11)*。计算机协会，纽约州纽约市，美国，文章16，14页。https://doi.org/10.1145/2038916.2038932

[15] C. G. Cassandras。1993年。离散事件系统：建模与性能分析。Irwin。[16] Dominik Charousset、Raphael Hiesgen 和 Thomas C. Schmidt。2016年。重新审视C++中的Actor编程。《计算机语言、系统与结构》45（2016年4月），105–131。[17] Sylvan Clebsch、Juliana Franco、Sophia Drossopoulou、Albert Mingkun Yang、Tobias Wrigstad 和 Jan Vitek。2017年。Orca：面向Actor语言的垃圾回收与类型系统协同设计。《ACM程序设计语言》1，OOPSLA，文章 72 2017 28 10 1145 3133896（10月），[18] 页数。https://doi.org/ 。/ Eric C Cooper 1988 和 Richard P Draves。 。C线程。技术报告CMU-CS-88-154。[19] Gregory H Cooper 和 Shriram Krishnamurthi。2006年。在按值调用语言中嵌入动态数据流。载于欧洲编程研讨会。Springer，294–308。[20] Evan Czaplicki 和 Stephen N Chong。2013年。面向图形用户界面的异步函数式响应式编程。载于第34届ACM SIGPLAN编程语言设计与实现会议论文集（PLDI'13）。ACM出版社。[21] Jack B. Dennis。1974年。第一版数据流过程语言。报告MAC TM61。麻省理工学院计算机科学实验室。[22] Ankush Desai、Vivek Gupta、Ethan Jackson、Shaz Qadeer、Sriram Rajamani 和 Damien Zufferey。2012年。P：安全的异步事件驱动编程。报告。微软研究院。[23] Stephen Edwards 和 John Hui。2020年。稀疏同步模型。载于规范与设计语言论坛（FDL），德国基尔，9月15–17日。IEEE，1–8。[24] Stephen A. Edwards 和 Edward A. Lee。2003年。同步框图语言的语义与执行。《计算机编程科学》48，1（2003年），21–42。https://doi.org/10.1016/S 0167 6423 02 00096 5 - ( ) - Conal Elliott 和 Paul Hudak。1997年。[25] 函数式响应式动画。载于《ACM SIGPLAN通报》第32卷。ACM，263–273。[26] Sebastian Ertel、Justus Adam 和 Jeronimo Castrillon。2018年。支持大数据系统中的细粒度数据流并行。载于第9届多核与众核编程模型及应用国际研讨会论文集。41–50。[27] Patrick Th. Eugster、Pascal A. Felber、Rachid Gerraoui 和 Anne-Marie Kermarrec。2003年。发布/订阅的多种面貌。《计算机调查》 35, 2 2003 114 131 10 1145 857076 857078 （ ），[28] – 。https://doi.org/ 。/ 。John Field、Maria-Cristina 2009 Marinescu 和 Christian Stefansen。 。反应器：面向分布式应用的面向数据的同步/异步编程模型。《理论计算机科学》410，2（2009年），168–201。https://doi.org/10.1016/j.tcs.2008.09.052分布式计算技术。[29] Spencer P Florence、Shu-Hung You、Jesse A Tov 和 Robert Bruce Findler。2019年。Esterel演算：能则能，不能则不能。《ACM编程语言论文集》3，POPL（2019年），1–29。[30] Marc Geilen、Twan Basten 和 Sander Stuijk。2005年。通过模型检验最小化同步数据流图的缓冲区需求。载于设计自动化会议（DAC）。ACM，819–824。https://doi.org/ 10 1145 1065579 1065796 。/ 。Nicholas Halbwachs、[31] Paul Caspi、Pascal Raymond 和 Daniel Pilaud。1991年。同步数据流编程语言LUSTRE。《IEEE论文集》79，9（1991年），1305–1320。[32] Brandon Hedden 和 Xinghui Zhao。2018年。Actor系统中的缺陷综合研究。载于第47届并行处理国际会议论文集（美国俄勒冈州尤金）（ICPP 2018）。计算机协会，纽约州纽约市，美国，文章56，9页。https://doi.org/ 10 1145 3225058 3225139 。/ 。Carl Hewitt。 。Actor [33] 计算模型：可扩展的 2010 稳健信息系统。arXiv 预印本 arXiv:1008.1459（2010年）。[34] Shams M. Imam 和 Vivek Sarkar。2014年。Savina——Actor基准测试套件：支持对Actor库进行实证评估。载于论文集 第*4*届基于参与者、代理与分布式控制的国际研讨会论文集（美国俄勒冈州波特兰）（*AGERE! '14*）。计算机协会，美国纽约州纽约市，67–80。https://doi.org/ 10 1145 2687357 2687368 。/。Gilles Kahn。并行程序设计简单语言的语义。载于[35] 1974 *IFIP 74* 大会论文集。北荷兰出版公司，471–475。[36] Gilles Kahn 和 D. B. MacQueen。1977年。协程与并行进程网络。载于信息处理，B. Gilchrist（编）。北荷兰出版公司，993–998。[37] Dieter Kranzlmüller 和 Martin Schulz。2002年。消息传递程序中的非确定性札记。载于并行虚拟机与消息传递接口的最新进展，Dieter Kranzlmüller、Jens Volkert、Peter Kacsuk 和 Jack Dongarra（编）。Springer Berlin Heidelberg，柏林，海德堡，357–367。[38] Yu-Kwong Kwok 和 Ishfaq Ahmad。1999年。将有向任务图分配到多处理器的静态调度算法。*ACM Comput. Surv.* 31卷，4期（1999年12月），406–471。https://doi.org/10.1145/344588。344618[39] Ivan Lanese、Naoki Nishida、Adrián Palacios 和 Germán Vidal。2018年。CauDEr：面向 Erlang 的因果一致可逆调试器。载于函数式与逻辑编程，John P. Gallagher 和 Martin Sulzmann（编）。Springer International Publishing，Cham，247–263。[40] Edward A. Lee。2006年。线程的问题。《计算机》39卷，5期（ 2006 33 42 10 1109 2006 180 ），–。https://doi.org/。/ MC.。Edward A. Lee。[41] 2021 确定性。*ACM* 嵌入式计算系统汇刊（*TECS*）20卷，5期（2021年7月），1–34。https://doi.org/10.1145/3453652[42] Edward A. Lee、Soroush Bateni、Shaokai Lin、Marten Lohstroh 和 Christian Menard。2021年。量化并推广 CAP 定理。*CoRR* abs/2109.07771（2021）。arXiv:2109.07771 https://arxiv.org/abs/ 2109 07771 。Edward A. Lee 和 Eleftherios Matsikoudis[43]。2009年。带触发的数据流语义。剑桥大学出版社。

h
t
t
p://
ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/

[44] E. A. Lee 和 D. G. Messerschmitt。1987年。同步数据流。*Proc. IEEE* 75, 9 1987 1235 1245 10 1109 1987 13876 ( )，–。[45] https://doi.org/。/PROC.。E. A. Lee 1995 和 T. M. Parks。数据流进程网络。*Proc.IEEE* 83, 5 1995 773 801 10 1109 5 381846 ( )，–。https://doi.org/。/。Edward A. Lee [46] 2007 和 Haiyang Zheng。利用同步语言原理对嵌入式系统进行异构建模与设计。载于*EMSOFT*。ACM，114–123。https://doi.org/10.1145/1289927.1289949[47] Edward A. Lee、Haiyang Zheng 和 Ye Zhou。2005年。因果关系接口与组合式因果关系分析。载于接口技术基础（*FIT*），*CONCUR* 卫星研讨会。[48] Bozhen Liu、Peiming Liu、Yanze Li、Chia-Che Tsai、Dilma Da Silva 和 Jeff Huang。2021年。当线程遇上事件：基于起源的高效且精确的静态数据竞争检测。载于第*42*届 *ACM SIGPLAN*编程语言设计与实现国际会议论文集。计算机协会，美国纽约州纽约市。https://doi.org/ 10 1145 3453483 3454073 。/。Tongping Liu、[49] Charlie Curtsinger 和 Emery D. Berger。2011年。Dthreads：高效的确定性多线程。载于第二十三届 *ACM*操作系统原理研讨会论文集（葡萄牙卡斯凯什）（*SOSP '11*）。计算机协会，美国纽约州纽约市， 327 336 10 1145 2043556 2043587 –。https://doi.org/。/。[50] Marten Lohstroh。2020 反应器：反应式系统并发计算的确定性模型。博士论文。加州大学伯克利分校 EECS 系。http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html

[51] Marten Lohstroh、Íñigo Íncer Romeo、Andrés Goens、Patricia Derler、Jeronimo Castrillon、Edward A. Lee 和 Alberto Sangiovanni-Vincentelli。2019年。反应器：一种用于可组合反应式系统的确定性模型。载于第*8*届国际网络物理系统模型化设计研讨会（*CyPhy'19*），第11971卷（计算机科学讲义）。施普林格出版社，27。[52] Marten Lohstroh 和 Edward A. Lee。2019年。确定性 Actor。载于规范与设计语言论坛（*FDL*）。[53] Marten Lohstroh、Christian Menard、Soroush Bateni 和 Edward A. Lee。2021年。迈向确定性并发系统的通用语言。*ACM* 嵌入式计算系统汇刊（*TECS*），*FDL'19* 专刊20卷，4期（2021年5月），第36篇文章。https://doi.org/10.1145/3448128[54] Marten Lohstroh、Christian Menard、Alexander Schulz-Rosengarten、Matthew Weber、Jeronimo Castrillon 和 Edward A. Lee。2020年。一种用于跨多条时间线进行确定性协调的语言。载于规范与设计语言论坛（*FDL*）。IEEE。https://doi.org/ 10 1109 50818 2020 9232939 。/FDL。。[55] Aman Shankar Mathur、Burcu Kulahcioglu Ozkan 和 Rupak Majumdar。2018年。IDeA：面向 Actor 的沉浸式调试器。载于第*17*届 *ACM SIGPLAN Erlang* 国际研讨会论文集（美国密苏里州圣路易斯）（*Erlang 2018*）。计算机协会，美国纽约州纽约市，1 12 10 1145 3239332 3242762 –。[56] 2010 https://doi.org/。/。Erik Meijer。响应式扩展（Rx）：解决异步编程的烦恼。载于*ACM SIGPLAN* 函数式编程商业用户研讨会（马里兰州巴尔的摩）（*CUFP '10*）。ACM，美国纽约州纽约市，第11篇文章，共1页。[57] Timothy Merrifield、Joseph Devietti 和 Jakob Eriksson。2015年。基于全存储顺序一致性的高性能确定性。载于第十届欧洲计算机系统会议论文集（法国波尔多）（*EuroSys '15*）。计算机协会，美国纽约州纽约市，第31篇文章，共13页。https://doi.org/10.1145/2741948.2741960 [58] Timothy Merrifield、Sepideh Roghanchi、Joseph Devietti 和 Jakob Eriksson。2019年。用于加速确定性多线程的惰性确定性。载于第二十四届编程语言与操作系统体系结构支持国际会议论文集（美国罗得岛州普罗维登斯）（*ASPLOS '19*）。计算机协会，美国纽约州纽约市，879–891。https://doi.org/10.1145/3297858.3304047[59] Ragnar Mogk、Joscha Drechsler、Guido Salvaneschi 和 Mira Mezini。2019年。面向分布式交互应用的容错编程模型。《*ACM*编程语言论文集》OOPSLA（2019），1–29。[60] Philipp Moritz、Robert Nishihara、Stephanie Wang、Alexey Tumanov、Richard Liaw、Eric Liang、William Paul、Michael I. Jordan 和 Ion Stoica。2017年。Ray：面向新兴 AI 应用的分布式框架。*CoRR* abs/1712.05889（2017）。arXiv:1712.05889 http://arxiv.org/abs/ 1712 05889 。[61] Madanlal Musuvathi、Shaz Qadeer、Thomas Ball、Gerard Basler、Piramanayagam Arumuga Nainar 和 Iulian Neamtiu。2008年。在并发程序中发现并重现海森堡错误。载于*OSDI*，第8卷。[62] Walid A. Najjar、Edward A. Lee 和 Guang R. Gao。1999年。数据流计算模型的进展。*Parallel Comput.* 25卷，13-14期（1999年12月），1907–1929。[63] Bruce Jay Nelson。1981年。远程过程调用。博士论文。美国。[64] R. Perrey 和 M. Lycett。2003年。面向服务的体系结构。载于*2003*年应用与互联网专题研讨会。IEEE，116–119。https://doi.org/ 10 1109 2003 1210138 。/SAINTW。。[65] 2018 Aleksandar Prokopec。反应器编程模型的可插拔调度。载于使用 *Actor* 编程：研究现状与研究展望，Alessandro Ricci 和 Philipp Haller（编）。Springer International Publishing，125–154。[66] Raymond Roestenburg、Rob Bakker 和 Rob Williams。2016年。《*Akka* 实战》。Manning Publications Co. [67] Koushik Sen。2008年。并发程序的竞态引导随机测试。*SIGPLAN* 通讯 43卷第6期（2008年6月），11–21。https://doi.org/10. 1145/1379022.1375584[68] Lui Sha、Abdullah Al-Nayeem、Mu Sun、José Meseguer 和 Pete C. Olveczky。2009年。*PALS*：物理异步、逻辑同步系统。技术报告。伊利诺伊大学。http://hdl.handle.net/ 2142/11897[69] Kazuhiro Shibanai 和 Takuo Watanabe。2017年。Actoverse：面向 actor 的可逆调试器。载于第*7*届 *ACM SIGPLAN* 演员、智能体与分布式控制编程国际研讨会论文集（加拿大不列颠哥伦比亚省温哥华）（*AGERE 2017*）。计算机协会，美国纽约州纽约市，50–57。https://doi.org/ 10 1145 3141834 3141840 。/。Marjan Sirjani、Ali [70] Movaghar、Amin Shali 和 Frank S. de Boer。2004年。使用 Rebeca 对反应式系统进行建模与验证。*Fundam. Inform.* 63卷第4期（2004年），385–410。[71] Samira Tasharofi、Peter Dinges 和 Ralph E Johnson。2013年。为什么 Scala 开发者会将 Actor 模型与其他并发模型混用？载于欧洲面向对象编程大会。Springer，302–326。[72] Samira Tasharofi、Rajesh K. Karmani、Steven Lauterburg、Axel Legay、Darko Marinov 和 Gul Agha。2012年。TransDPOR：一种用于测试 actor 程序的新型动态偏序约简技术。载于分布式系统形式化技术，Holger Giese 和 Grigore Rosu（编）。Springer Berlin Heidelberg，柏林，海德堡，219–234。[73] Samira Tasharofi、Michael Pradel、Yu Lin 和 Ralph Johnson。2013年。Bita：覆盖率引导的 actor 程序自动化测试。载于*2013*年第*28*届 *IEEE/ACM* 自动化软件工程国际会议（*ASE*）。114 124 10 1109 2013 6693072 –。[74] 2010 https://doi.org/。/ASE。/。Stefan Tilkov 和 Steve Vinoski。/。Node.js：使用 JavaScript 构建高性能网络程序。*IEEE Internet Computing* 14卷第6期（2010年），80–83。[75] Carmen Torres Lopez、Robbert Gurdeep Singh、Stefan Marr、Elisa Gonzalez Boix 和 Christophe Scholliers。2019年。多重宇宙调试：针对非确定性程序的非确定性调试（崭新创意论文）。载于第*33*届欧洲面向对象编程大会（*ECOOP 2019*）（《信息学讲义丛书》（*LIPIcs*），第*134*卷），Alastair F. Donaldson（编）。Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik，德国 Dagstuhl，27:1–27:30。https://doi.org/10.4230/LIPIcs.ECOOP.2019.27[76] Carmen Torres Lopez、Stefan Marr、Elisa Gonzalez Boix 和 Hanspeter Mössenböck。2018年。使用 *Actor* 编程：研究现状与研究展望。Springer International Publishing，Cham，第A章：Actor 程序并发错误与高级开发支持研究，155–185。https://doi.org/10.1007/978-3-030-00302-9 6_[77] Tom Van Cutsem、Elisa Gonzalez Boix、Christophe Scholliers、Andoni Lombide Carreton、Dries Harnie、Kevin Pinte 和 Wolfgang De Meuter。2014年。AmbientTalk：使用 actor 编写响应式移动点对点应用程序。《计算机语言、系统与结构》40卷第3–4期（2014年），112–136。[78] Reinhard von Hanxleden、Edward A. Lee、Hauke Fuhrmann、Alexander Schulz-Rosengarten、Sören Domrös、Marten Lohstroh、Soroush Bateni 和 Christian Menard。2022年。十二年后的 Pragmatics：Lingua Franca 报告。载于形式化方法、验证与确认的应用：验证原理：第*11*届形式化方法应用国际研讨会（*ISoLA 2022*），Tiziana Margaria 和 Bernhard Steffen（编）。Springer Nature Switzerland，60–89。[79] Bernard Zeigler。1976年。建模与仿真理论。Wiley Interscience，纽约。