TimeTravelContainer：用于有状态智能体探索的实时分叉沙箱

Zizheng Guo、Yanqing Zhang、Nathaniel Pinckney、Cunxi Yu、Brucek Khailany 英伟达公司{zizhengg,yanqingz,cunxiy,bkhailany}@nvidia.com

arXiv:2611.32255v1 [cs.OS] 24 Nov 2026

摘要—驱动有状态工具（例如交互式 shell、设计环境、仿真引擎）的大语言模型智能体，经常需要克隆、回溯并并行运行实时会话，以支持探索。然而，如今克隆实时会话需要将工具的内存状态序列化到磁盘，或从头重新启动；这两种方式对于交互式探索而言都慢得难以接受。现有基于微型虚拟机或内核补丁的智能体沙箱，要么将客户机的完整内存序列化到存储设备，要么需要特权部署，因此难以用于共享计算集群、个人计算机及其他受限环境。我们提出 TimeTravelContainer (TTC)，这是一种无根容器运行时，能够对整个运行中的容器进行实时分叉，包括进程、内存、文件系统和网络，并采用写时复制（CoW）语义。由于工具会话的大部分内存通常在分叉点之后为只读，克隆体会自动共享这些内存，只有发生分歧的页面才会占用额外内存。TTC 无需内核修改、硬件虚拟化或根权限，可在包括 Linux、macOS（Apple Silicon）和 Windows（WSL 2）在内的所有主流平台上运行，因此可作为智能体运行框架的便携式基础。我们在两种智能体驱动的电子设计自动化（EDA）流程上评估 TTC，并将其作为有代表性的有状态工具工作负载：MAGE (Innovus) 和 EvoDRC (KLayout)。TTC 实现了 $84.1\times$环境克隆加速比，31–72% 的内存缩减，以及 $1.7\text{–}9.4\times$ 端到端加速比，表明实时分叉消除了有状态工具环境中智能体驱动探索的扩展瓶颈。https://github.com/NVlabs/TimeTravelContainer

I. 引言

大语言模型智能体越来越有能力通过多轮交互来驱动复杂的有状态工具。在软件工程中，智能体会打开编辑器和终端、提出代码更改、运行测试，并根据反馈反复迭代 [1]，[2]。在电子设计自动化 (EDA) 中，智能体会驱动交互式工具 shell 来优化电路 [3], [4], [5]。机器人仿真、数据库调优、科学计算以及智能体需要与交互式工具保持长期会话的各种场景中，也存在类似模式。

此类智能体的一项关键能力是有状态探索：能够克隆正在运行的工具会话，并行尝试替代操作，同时回滚到更早的状态而不会丢失工作成果。这与树搜索、蒙特卡洛展开和推测执行直接对应，而这些方法的有效性取决于创建和管理替代执行状态的成本。为了让智能体实现超人的周转速度和质量，承载工具会话的沙箱必须支持即时克隆、低成本分支和透明隔离。

如今的智能体运行框架对此支持不足。克隆工具会话目前只能通过以下方式实现：将工具状态保存到磁盘上的检查点并恢复，或从已保存的快照重新启动工具。现有的智能体沙箱 [6]，[7] 可提供隔离，但没有克隆实时会话的机制。基于微型虚拟机的平台 [8], [9], [10] 支持虚拟机快照，但会将完整的客户机内存映像序列化到存储中；对于常驻内存达数十吉字节的工具而言，这并不实用。内核级方法 [11], [12]能够实现更快的分叉，但需要定制内核模块或提升的权限，因此不适用于共享计算集群、个人笔记本电脑和边缘设备：在这些环境中，安装内核模块要么复杂且不安全，要么即便获准也可能需要数周的 IT 审批。由于沙箱将由自动运行智能体直接调用，使用非标准内核模块还会增加遭受外部恶意攻击的风险。缺少实时分叉支持，迫使智能体只能按顺序探索，浪费可加速搜索的并行性；或者为每次克隆付出高昂的延迟和内存开销。

我们介绍 *TimeTravelContainer*（TTC），一种具备实时分叉原语的无根 Linux 容器运行时。TTC 构建于 gVisor 的 [13] 用户空间内核之上，能够以写时复制（CoW）语义克隆整个运行中的容器，包括所有进程、线程、内存、文件系统和内部网络状态。由于所有操作系统状态都由用户空间管理，TTC 无需内核补丁、硬件虚拟化或根权限即可实现容器级分叉，而且自身也能运行在另一个沙箱内。TTC 可原生运行于 Linux、Apple Silicon Mac（通过 Hypervisor.framework）和 Windows（通过 WSL 2），为跨平台的智能体运行框架提供可移植的基础，适用于从云集群到个人计算机。一次 fork 会创建共享同一因果前缀的独立克隆：分叉点之前的工具状态仍保持共享；只有克隆实际修改的内存页才会被分配。这使 fork 的成本不受常驻内存大小影响，而仅与后续的分歧程度成正比。

我们在两种智能体电子设计自动化流程上评估 TTC，将其作为具有代表性的

有状态工具工作负载。电子设计自动化是对实时分叉的压力测试：工具会维护数十至数百 GB 的常驻内存，其中主要是只读数据结构；它们以无法修改的专有二进制文件形式运行，并在通常无法获得根权限的共享计算集群上执行。我们将 TTC 集成到 MAGE [4], 中，用于驱动 Cadence Innovus 进行交互式宏单元布局；还集成到 EvoDRC [5], 中，该工具使用 KLayout 以迭代方式修复设计规则违规。我们的贡献如下：

1) **适用于智能体沙箱的可移植实时分叉原语**。TTC 提供基于内存的写时复制容器分叉，作为一种无需 root 权限的非特权运行时，无需内核修改，并可在 Linux、macOS 和 Windows 上运行，从而弥补当前智能体基础设施的性能差距（表 I）。

2)**集成到两种智能体驱动的电路设计流程中**。我们展示了 TTC 如何取代 MAGE/Innovus 中基于文件的检查点，以及 EvoDRC/KLayout 中的工具重新启动，从而无需修改工具本身即可实现透明克隆和并行探索。

3) **系统化评估**。在 MAGE 上，TTC 将环境克隆速度提升了 $84.1\times$（86.5 秒 $\rightarrow$ 1.03 秒），并在并发实例间将驻留内存缩减 31–72%。在 EvoDRC 上，TTC 在原始小型基准测试中实现了 1.8–$2.1\times$ 的端到端加速，在大型设计上实现了 8–$9.4\times$ 的加速，表明实时分叉消除了此前限制智能体驱动探索范围的扩展性障碍。

II. 初步介绍

A. 智能体工作流中的有状态探索

大语言模型智能体日益广泛地在各个领域操作有状态的交互式工具。在软件工程中，智能体会打开编辑器、终端和调试器，以提出补丁、运行测试并根据反馈迭代 [1], [2]。在电子设计自动化（电子设计自动化）领域，智能体已应用于芯片设计流程的各个环节：RTL生成与修复 [14], [15], [16], 验证与测试平台构建 [17], [18], [19], 逻辑综合优化 [20], [21], [22], 物理实现 [3]，[23], [24], 设计规则违规修复 [5], [25], 以及端到端 ASIC流程 [26]。类似的有状态交互模式也出现在机器人仿真、数据库管理、科学计算，以及任何智能体需要与交互式工具保持长期会话的领域。

这些领域中的常见模式是有状态探索：智能体开启一个交互式工具会话，逐步提出更改、评估反馈，并决定继续、回溯或克隆出并行的备选方案。这类探索的成效受管理不同执行状态的成本所限制。如果回溯必须依赖基于文件的检查点，并从头重新启动工具，智能体就只能探索更少的备选方案，或者付出巨大的实际运行时间和内存代价。这一瓶颈促使人们为智能体沙箱开发一种通用的实时分叉原语。

B. 智能体沙箱与实时分叉

智能体运行框架在沙箱环境中执行工具调用，以实现隔离和可复现性。大多数沙箱优先考虑安全性，即防止智能体逃逸主机，而非执行灵活性，例如克隆和恢复实时状态。Docker [6]和 OCI 容器提供文件系统和进程隔离，但没有克隆运行中容器的机制。SWE-agent [1] 和 Open-Hands [2]将智能体会话封装在 Docker 中；要进行分叉，必须先停止容器、提交更改并重新启动，这会丢弃内存中的状态。E2B [7]提供云端沙箱和文件系统快照，可将状态序列化到磁盘，但无法捕获运行中的进程内存。

虚拟化方法。 越来越多的研究致力于实现执行状态灵活性。AgentENV [8], CubeSand- box [9], 和 forkd [10]基于微型虚拟机虚拟化（KVM/Firecracker），支持对虚拟机进行快照或分叉。然而，它们的分叉原语会将完整的客户机内存映像 序列化到存储中：必须先写出虚拟机状态，再读回以创建克隆。有状态工具通常会维护数 GB 到数十 GB 的常驻内存，其中大部分是无需复制的只读数据（已加载模型、已解析数据库、缓存索引）。因此，基于磁盘的虚拟机快照无法满足交互式智能体探索所需的亚秒级克隆速度。

内核级方法。 近期的操作系统研究直接解决了实时分叉问题。TClone [11] 向 Linux 注入内核模块，以实现跨容器的写时复制（CoW）页表共享，并扩展 CRIU [27]，使其能够跨命名空间重建进程树，而无需将内存序列化到磁盘；与原版 CRIU 相比，其分叉延迟更低。DeltaBox [12] 提出了增量检查点抽象，并通过从冻结的模板进程分叉来加速回滚。Fork-Explore-Commit [28] 引入了基于 FUSE 的写时复制文件系统，为每个智能体克隆提供隔离的工作区，但它在文件系统层面运行，无法分叉实时进程状态。这些内核模块或内核补丁方法能够实现快速分叉，但要求部署定制内核或特权模块，因此在共享计算集群、托管云环境、个人计算机以及智能体日益运行的移动设备或边缘设备上的适用性受到限制。

Unix `fork()` 及其局限性。 fork() 系统调用 [29]通过写时复制（CoW）创建一个与父进程共享地址空间的子进程，因此无论内存大小如何，几乎都能瞬间完成。不过，fork() 无法直接克隆复杂的应用环境：它只会复制调用线程，导致子进程中的所有其他线程终止；它不会保留进程 ID、父子关系，

表 I：智能体执行环境比较。

平台	实时分叉支持	内存中 CoW	未修改内核	未特权化
Docker [6]	✗	✗	✓	✗
SWE-agent [1]	✗	✗	✓	✗
OpenHands [2]	✗	✗	✓	✗
E2B [7]	✗	✗	✓	✓
AgentENV [8]	✓	✗	✗	✗
CubeSandbox [9]	✓	✗	✗	✗
forkd [10]	✓	✗	✗	✗
TClone [11]	✓	✓	✗	✗
DeltaBox [12]	✓	✓	✗	✗
FEC [28]	✓*	✗	✓	✓
gVisor [13]	✗	✗	✓	✓
TTC（我们的方法）	✓	✓	✓	✓

*仅限文件系统；不会分叉运行中的进程状态。

或会话/组成员关系；也不会复制指向容器本地资源（如内部套接字或伪终端）的已打开文件描述符。多线程工具包含后台线程、进程间通信通道和多进程架构，仅仅注入 fork() 调用就会导致崩溃或死锁。

CRIU（用户空间检查点/恢复） [27] 通过捕获进程树的完整状态来解决多进程重建问题，其中包括线程、进程标识符、文件描述符、信号处理程序和命名空间成员关系。然而，它需要将整个内存映像序列化到磁盘。恢复需要提升的权限（CAP_CHECKPOINT_RESTORE 或 CAP_SYS_ADMIN），而且恢复的副本会独立实例化每个内存页，包括本可共享的大型只读区域，无法与原始副本进行写时复制共享。

gVisor 与 TTC。gVisor [13] 完全在用户空间中重新实现 Linux 内核接口，将其作为沙箱进程（即“Sentry”）运行，在无需内核修改或硬件虚拟化的情况下拦截应用程序的系统调用。由于 Sentry 在自己的地址空间中管理所有操作系统状态，包括页表、文件描述符、进程标识符和套接字，因此它可以向客户进程提供一个完整的 Linux 内核，同时为安全目的实现自身的抽象层。我们发现，gVisor 的抽象能力超出了沙箱安全的需求，还可用作通用的内存和操作系统资源虚拟化层。TimeTravelContainer（TTC）在 gVisor 的基础上扩展了智能体所需的基本原语：容器级分叉，可通过写时复制语义克隆完整环境（进程、线程、文件系统、网络和内存），且无需内核补丁、KVM 或提升的权限。第三节将详述技术细节。表 I 对比了 TTC 与现有平台。

III. TIMETRAVELCONTAINER

图 1 展示了 TimeTravelContainer (TTC) 的体系结构。沙箱族将同一容器的所有克隆组织成类 Git 分支。这些分支共享单个平台实例（守护进程）和页面池，但每个分支都拥有独立的内核对象图，其中包含各自的进程树、文件描述符、虚拟文件系统和网络协议栈。以下各小节将介绍 TTC 如何实现实时分叉、写时复制资源共享、生命周期管理、跨平台可移植性和跨节点扩展。

A. 用户空间内核：可移植的基础

智能体运行框架必须能在各种环境中运行：共享计算集群、托管云、个人笔记本电脑，甚至其他沙箱内部。正如先前工作所要求的那样，需要内核模块、硬件虚拟化或根权限 [11], [12], [8], ，这会将部署限制在允许进行此类修改的环境中。

TTC 采用 gVisor 的用户空间内核 [13] 作为基础。gVisor 的用户空间内核（称为 *Sentry*）是一个非特权的单一进程，它会拦截来自客户应用程序的每个系统调用和缺页异常，并模拟完整的 Linux 内核接口：进程管理、虚拟内存、文件系统、网络功能和信号。每个客户进程，无论是多线程应用服务器、数据库还是电子设计自动化工具，看到的都是完整且未经修改的 Linux 环境。

由于所有操作系统状态都位于 Sentry 的地址空间内，而非宿主内核数据结构中，TTC 可以将容器级分叉、暂停、去水合和迁移实现为对其自身进程内数据的操作，无需内核协作。这些操作对客户应用程序不可见，也无需与其协同设计：工具运行未经修改的二进制文件，并不知道自己的环境已被分叉或迁移。

B. 并行探索的写时复制内存

并行探索意味着运行同一工具的多个副本。简单地为每个副本生成其自身的完整内存映像。单个交互式会话可能会使用数 GB 到数十 GB 的驻留内存。运行四个并行分支会按比例放大内存需求，很快耗尽节点容量。

实际上，这些内存中的大部分在各分支之间并不会发生变化：已解析的数据结构、已加载的库、缓存索引，以及早于分叉点的工具状态都只是只读的。TTC 利用这一点，在各分支之间透明地共享相同的页面。

在 fork 时，TTC 会将源分支中的每个内存页标记为写时复制：逻辑上可写，但在硬件中映射为只读。当某个分支首次写入共享页面时，硬件内存管理单元会生成缺页异常。TTC 截获该异常，分配一个新的 4 KiB 页面，复制原始内容，并将新页面重新映射为可写。此后，对该页面的读写完全由硬件内存管理单元处理，无需 TTC 介入。因此，写时复制回收成本是每个发生分歧的页面一次性的开销，而稳态内存开销仅与

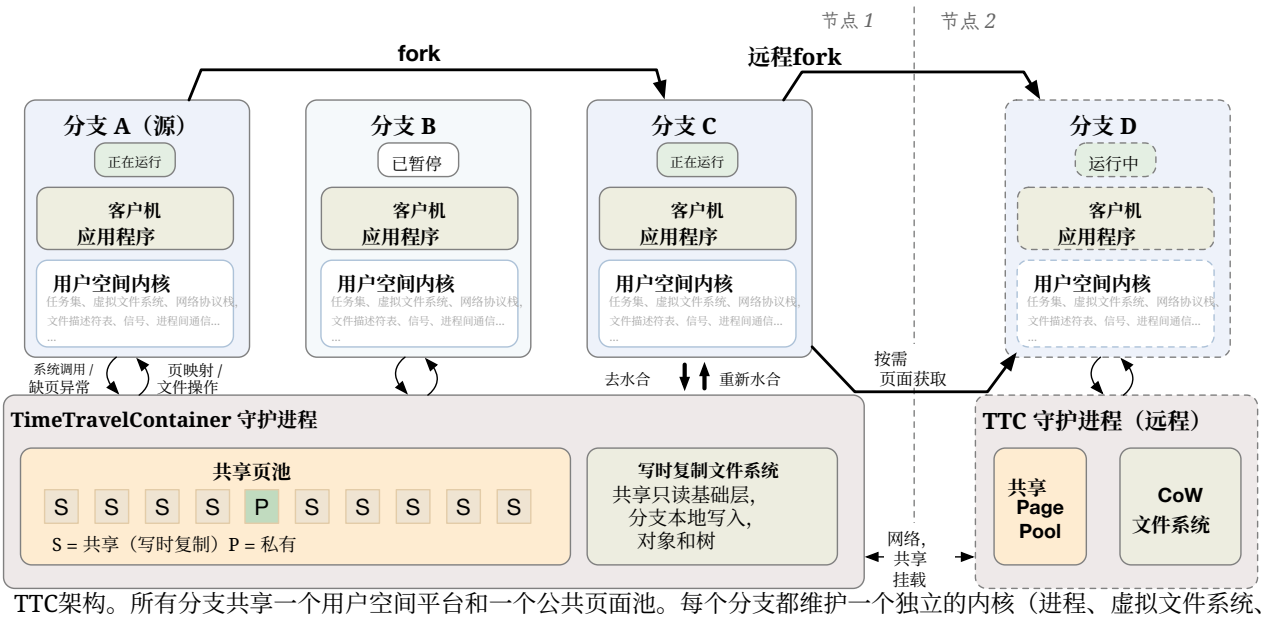


图 1：TTC架构。所有分支共享一个用户空间平台和一个公共页面池。每个分支都维护一个独立的内核（进程、虚拟文件系统、网络）。标记为 S 的页面通过写时复制共享；标记为 P 的页面已发生分歧，属于分支私有页面。图中所示操作包括：**fork** （即时的内存克隆）、**暂停** （释放 CPU，内存驻留）、**去水合**（将私有页面写入磁盘），以及**远程fork** （跨节点克隆，并按需获取页面）。

分支之间实际存在差异的页面数量成正比，而不是与总驻留大小成正比。

C. 用于数据隔离的写时复制文件系统

有状态工具会话通常涉及大量磁盘数据：设计文件、模型权重、数据库快照、中间结果和配置文件。智能体和工具会读取和写入这些文件，并发分支之间不得相互干扰。

在 SWE-agent [1] 和 OpenHands [2], 等软件工程智能体运行框架中，通常通过 Git工作树实现文件隔离，但这会复制整个工作树。这种方法在用于通用有状态工具时有两个局限：无论文件是否会被修改，都会复制所有文件，因而占用大量存储空间；而且它最适用于由 Git 跟踪的 ASCII 文本文件，但许多工作负载主要使用二进制文件或不适合纳入 Git 的大型构件。

相反，TTC为每个分支提供一个虚拟写时复制文件树。每项文件操作都会在用户空间内核层面被拦截，因为Sentry已经拥有客户进程所交互的整个文件系统接口。未修改的文件通过引用在各分支之间共享。当某个分支首次写入文件时，TTC会创建一个分支本地副本（如果可用，则使用主机文件系统引用链接，否则复制整个文件），并将后续访问重定向到该本地副本。这不需要FUSE模块或内核文件系统驱动，因为TTC已经调解了所有来宾I/O。

D. 空闲分支的资源管理

容器分支经常会变为空闲状态。子代理可能已经完成探索、等待大语言模型推理，或者仅仅作为一个干净的回滚点，由主代理保留，以防后续尝试失败。如果继续运行，这些空闲分支仍会消耗CPU时间和内存。

TTC 提供了两种分级的资源释放机制：

暂停会停止分支中的所有线程并释放 CPU，但会让分支的内存保持驻留。恢复是即时的：线程会从停止的位置准确地继续运行，无需重新加载。这适用于短暂的空闲时段，例如等待大语言模型推理。

去水合更进一步：它会将分支的私有页写入持久化存储，并释放相应的物理内存，使分支完全进入冷态。两项设计决策让去水合切实可行：

首先，从工具的角度来看，再水合是即时的。恢复去水合分支时，TTC 会在所有内存页从磁盘加载完毕之前立即启动其进程。进程访问的页面会优先按需获取；其余页面则在后台异步加载。工具无需等待批量恢复完成，即可从去水合后的第一条指令继续执行。

其次，去水合会保留写时复制关系。只有去水合分支独有的页面，即已经与其他活动分支分化的页面，才会

被保存。共享页面仍保留在页面池中，供仍需要它们的活动分支引用。之后重新水合该分支时，它会重新建立对共享页面的写时复制引用，而不是将其物化为独立副本。因此，去水合再重新水合的过程不会增加内存使用量。

E. 跨平台可移植性

智能体运行环境日益多样化，包括云集群、本地服务器、开发者笔记本电脑和边缘设备。依赖单一操作系统或体系结构的实时分叉原语会限制其采用。

TTC 通过用户空间内核设计实现跨平台可移植性。在**Linux 主机**（内核 ≥3.17）上，TTC 通过 seccomp 拦截系统调用，无需硬件虚拟化或嵌套虚拟化。在支持硬件虚拟化的环境中，还可选用 KVM 后端以降低系统调用延迟。在 **Apple Silicon Mac**（macOS 15.0+）上，TTC 使用原生 Hypervisor.framework 后端：它在虚拟机管理程序内作为面向客户机的内核运行，无需启动 Linux 虚拟机即可提供完整的 Linux 环境。在**Windows**上，TTC 通过 WSL 2 运行。在所有情况下，都全面支持相同的容器镜像、分叉语义和命令行界面。

TTC 是一个可扩展的平台，用户可以在其中添加自己的物理硬件抽象，并定义其分叉语义。TTC 内置的示例包括：在 Linux 上支持容器内 Vulkan 和 OpenGL 图形加速，以及 CUDA 通用 GPU 计算，因此可以启动可分叉、支持图形加速的安卓 [30]，也可以在可分叉容器内运行 GPU 加速推理（例如，使用 Vulkan 和 CUDA 后端的 llama.cpp）。这种可扩展性使 TTC 能够适配需要专用硬件的工作负载，涵盖使用计算加速器的电子设计自动化工具，以及驱动图形应用程序的智能体。

对可移植性的特别关注，使 TTC 能够成为必须跨平台运行的智能体运行框架的基础：无论是在无法获得根权限的共享电子设计自动化计算集群上、研究人员开发期间使用的 MacBook 上，还是在收集智能体轨迹以生成持续训练数据的边缘设备上。

F. 面向集群规模探索的跨节点分叉

有状态工具需要大量资源，而生产工作负载通常运行在由多个计算节点组成的集群上。当并行探索超出单个节点的内存或 CPU 容量时，分支必须转移到其他节点。

TTC 支持 远程*fork*：一个节点上的分支可以被 fork，以便在另一台物理节点上创建新分支；或者，现有分支可以在节点之间迁移，整个过程都不会中断正在运行的进程。目标节点会立即启动新分支，其初始内存映像为空，并按需获取页面：每个 1MiB 内存块都从源节点或共享存储（例如网络文件系统或 Lustre）中读取。

文件系统层遵循相同的模式。共享基础镜像中的文件对象可通过共享存储挂载访问。分支本地文件修改会与内存块一同延迟传输。这种设计意味着，远程分叉的实际耗时主要取决于目标分支实际访问的页面，而非源分支的总驻留大小。

IV. 实验结果

我们在两种智能体电子设计自动化（EDA）流程上评估 TTC，将其作为有状态工具工作负载的代表。电子设计自动化工具是检验实时分叉能力的严苛案例：它们会占用数十至数百 GB 的常驻随机存取存储器，以无法插桩的专有二进制文件形式运行，并在无法获得根权限的共享计算集群上操作。我们使用 MAGE [4], ，其中大语言模型智能体驱动 Cadence Innovus 进行交互式宏单元布局；以及 EvoDRC [5], ，其中进化智能体驱动 KLayout 进行设计规则检查违规修复。这两种流程代表了不同的集成模式：替代检查点/恢复（MAGE）和消除工具重新启动（EvoDRC）。

实验设置。所有实验均在服务器上运行，每台服务器配备两颗英特尔至强铂金 8270 处理器（2.70 GHz，52 个物理核心）和 755 GiB 随机存取存储器，并运行 Rocky Linux 8.9 和内核 6.1.62。每项工具操作使用 8 个工作线程，每个节点的总并发数上限为 52 个线程。我们使用了 MAGE 修订版 27c3ed7 [1]和 Cadence Innovus 23.37-s090 1，以及 EvoDRC 修订版 69a138b [2] 和 KLayout 0.30.12。所有实验都重放固定的工具工作负载，不调用大语言模型，因此报告的时间不包括大语言模型推理延迟。

A. 基准测试

表 II 总结了我们实验中使用的设计。MAGE 基准测试（Ariane133 和 BlackParrot）是来自 MAGE 流程的大型设计。EvoDRC 基准测试包括 DAC'26 设计规则检查基准套件 [31]中的七个小型区块（Block1–7），以及我们根据 ResizerA-gent [32] 改编的三个大型设计（高级加密标准、Ibex、JPEG）。

B. MAGE：克隆加速比

我们比较了两种创建已准备好的 Innovus 会话的独立可用副本的方法。在传统的 MAGE 工作流程中，代理会将 Innovus 检查点保存到磁盘，启动一个新的 Innovus 进程，然后恢复该检查点；保存和加载都受 I/O 限制，并且耗时随磁盘映像大小增长。使用 TTC 时，代理

[1]https://github.com/ABKGroup/MAGE [2] https://github.com/ASU-VDA-Lab/EvoDRC

表 II：基准测试设计。

Flow	设计	工艺	单元	Nets
MAGE	Ariane133 Nangate45 118594 172525			
MAGE	BlackParrot Nangate45 719 401 1354 154			
DAC26 设计规则检查 区块1		ASAP7	143	120
DAC26 设计规则检查 区块2		ASAP7	62	43
DAC26 设计规则检查 区块3		ASAP7	76	62
DAC26 设计规则检查 区块4		ASAP7	107	81
DAC26 设计规则检查 区块5		ASAP7	47	28
DAC26 设计规则检查 区块6		ASAP7	155	101
DAC26 设计规则检查 区块7		ASAP7	574	410
ResizerAgent 高级加密标准		ASAP7	31 163	18 612
ResizerAgent Ibex		ASAP7	32 705	23 029
ResizerAgent JPEG		ASAP7	75 803	61 925

表 III：MAGE/Innovus：克隆延迟对比。

设计	保存 (s)	加载 (s)	总计 (s)	TTC fork (s)	加速比-up
Ariane133	11.4	75.1	86.5	1.03	84.1×
BlackParrot	22.3	163.4	185.7	3.01	61.8×

则会 fork 运行中的容器，保留已初始化的 Innovus 进程，并通过写时复制共享其内存。两种路径均在等效的就绪查询成功后完成；我们测量从启动克隆到就绪之间的总耗时。这些结果隔离了克隆原语，不包括 MAGE 代理的端到端运行时间，而目前该时间主要受 Innovus 布线影响。

表 III 报告了实验结果。在 Ariane133 上，TTC 将克隆延迟从 86.5 s 降至 1.03s，提速 84.1×。对于规模更大的 BlackParrot 设计（719 K 个单元），检查点/恢复需要 185.7 s；TTC fork 仅需 3.0 s，提速61.8× 。TTC 的 fork 开销主要来自元数据序列化，且在很大程度上与常驻内存大小无关。fork 之后，每个子进程独立执行连通性检查、设计规则检查（DRC），以及三轮编辑与时序报告周期，以验证隔离运行的正确性，并在现实工作负载下测量内存差异（表 IV）。

C. MAGE：内存缩减

表 IV 报告了运行多个并发 Innovus 分支时的总驻留内存。在检查点/恢复基线中，每个分支都会生成独立的内存映像，因此总内存会随分支数量线性增长。使用 TTC 时，各分支通过写时复制共享只读页面：只有分支实际修改的页面才会被复制。

在 Ariane133 上使用 2 个分支时，TTC 在 fork 后立即（“就绪”状态）将内存占用降低了 37%。在分支经过连通性检查、设计规则检查以及编辑—时序报告周期而产生分歧后，TTC 仍可节省 31%，因为大部分内存（网表图、库和工艺数据）仍保持共享。使用 6 个分支时，节省幅度更加明显：fork 时为 70%，产生分歧后为 56%。在规模更大的 BlackParrot 设计上也呈现相同趋势：使用检查点/恢复需要 62.4 吉字节

表 IV：MAGE/Innovus：并发分支的总驻留内存（GB）。“节省”表示通过写时复制共享减少的内存占比。

分支	分歧步骤	检查点	TTC	节省
Ariane133（单个实例：*5.0 GB*）				
2	就绪	9.1	5.8	37%
	连接 + 设计规则检查	9.6	6.5	33%
	3× 编辑+报告	9.9	6.8	31%
6	就绪	25.1	7.5	70%
	连接 + 设计规则检查	27.7	12.3	56%
	3×编辑+报告	27.9	12.2	56%
BlackParrot（单个实例：*8.8* 吉字节）				
2	就绪	17.1	10.6	38%
	连接 + 设计规则检查	19.0	14.4	24%
	3× 编辑+报告	20.5	15.5	24%
6	就绪	50.1	14.0	72%
	连接 + 设计规则检查	59.7	36.1	40%
	3× 编辑+rpt	62.4	40.9	34%

而在完全产生分歧后，使用 TTC 仅需 40.9 吉字节，减少了 34%。

这些结果证实，写时复制共享对于电子设计自动化工作负载是有效的：驻留内存的大部分由只读数据结构组成（分叉点之前就已存在的网表、库和布线数据库），只有每个分支所做的增量修改会占用额外内存。

D. EvoDRC：端到端工具调用加速

在传统的 EvoDRC 流程中，每次候选方案评估都会准备一个 Python 布局上下文，并启动一个全新的 KLayout 进程进行设计规则检查。借助 TTC，我们只需准备一次上下文并启动一个持久运行的 KLayout 进程，然后针对每次候选方案评估对该环境执行 fork。每个子进程都会继承完全初始化的状态，同时其修改与父进程保持隔离；候选方案渲染和布局加载仍属于每个子进程的评估步骤。我们验证了两种流程中的几何图形、设计规则检查报告和修复判定结果一致。

表 V 报告了两种设置下的端到端运行时间：单次进化调用（进行一代变异）和连续 10 次调用（完整的修复过程）。

在小型区块（Block1–7）上，TTC 每次调用可实现 1.8–2.3× 倍加速。这些区块的 KLayout 启动开销较低，因此每次 fork 节省的绝对时间较少，但相对提升始终稳定。调用 10 次时，加速效果相近（1.8–2.1×），这证实了每轮节省的时间会线性累积。

在大型设计中，这一优势会变得非常显著。在 AES（31K 个单元）上，TTC 单次调用的速度提升达到 31.9× ，连续调用 10 次则达到 8.0× 。在 Ibex 和 JPEG 上，单次调用的速度分别提升 103× 和 173×。这些设计的版图数据库很大，导致重新启动 KLayout 的代价极高；TTC 的实时分叉则完全绕过了这项开销。连续调用 10 次的速度提升（9.7× 和 9.4×）比例较低，因为其中计入了会话开始时的一次性初始化

表 V：EvoDRC/KLayout：端到端工具调用运行时比较。

	单次调用			10 次调用		
设计	原始 (s)	TTC (s)	加速比 ×	原始 (s)	TTC (s)	加速比 ×
区块1	18.3	10.0	1.8	185	104	1.8
区块2	13.5	7.1	1.9	136	72	1.9
区块3	14.6	7.7	1.9	168	80	2.1
区块4	16.9	9.1	1.9	169	92	1.8
区块5	13.1	6.6	2.0	135	68	2.0
区块6	18.9	10.6	1.8	191	112	1.7
区块7	64.4	27.6	2.3	652	310	2.1
AES	1 318	41.3	31.9	13 823	1725	8.0
Ibex	3 088	29.8	103.5	30 879	3179	9.7
JPEG	8 527	49.3	173.0	85 273	9032	9.4

开销，以展示端到端的速度提升，但节省的绝对时间仍然非常可观：JPEG 的耗时从 23.7 小时降至 2.5 小时。

这些结果表明，TTC 消除了智能体驱动的探索中的扩展障碍。在工具启动成本适中的小型设计上，TTC 带来了稳定的∼2× 改进。对于大型设计，节省的时间十分可观：原本需要数天的运行缩短为不到一小时的实验，并使这些案例中的智能体迭代探索首次变得切实可行。

E. 讨论

MAGE 和 EvoDRC 的结果展示了 TTC 价值中相互补充的方面。

克隆成本。 MAGE 表明，与 Innovus 检查点/恢复相比，TTC 将单次克隆延迟降低了 61–84× 。这一加速是任何采用分支策略的智能体的基础，包括树搜索、蒙特卡洛展开或试探性探索。尽管 MAGE 当前的端到端运行时间主要耗费在 Innovus 布线上（每次布线需数分钟），但随着智能体转向更细粒度的探索、缩短每个分支的工作时间，克隆加速比将变得至关重要。

内存效率。 写时复制共享可将常驻内存减少 30–70%，具体取决于分歧程度。在固定内存预算下，这直接转化为更高的分支并行度：在 Ariane133 上运行 6 个分支时，检查点/恢复需要 25–28 吉字节，而 TTC 仅需 7.5–12.2 吉字节，因此可以运行更多分支，或在同一节点上为其他工作负载留出空间。

端到端影响。 EvoDRC 表明，当每个分支的工作相对于工具启动成本较短时，TTC 的实时分叉可带来变革性的端到端加速。在加载 KLayout 占主导的大型设计中，TTC 可实现 31–173× 的单次调用加速。TTC 使得在大规模场景下，能够基于沉重的工具状态进行短时的分支探索；这尤其符合大语言模型智能体的优势，因为它们倾向于快速试验并立即获得反馈。

超越电子设计自动化的泛化能力。尽管我们的评估使用了电子设计自动化工具，但使实时分叉发挥作用的特性在此：以读取为主的内存占用较大、工具初始化成本高昂，以及需要隔离式并行探索，这些特性在许多有状态智能体工作负载中都很常见。探索数吉字节数据库中的查询计划或模式迁移的数据库智能体，也面临同样的克隆或重新启动的两难选择。在物理引擎中评估动作序列的机器人仿真智能体，可以通过在仿真运行中途分叉仿真器获益，而不必从初始状态重新回放。软件工程智能体已经使用文件级隔离（Git工作树），但若能分叉整个环境，包括运行中的服务器、数据库和构建缓存，就可以进行更丰富的探索。TTC 的写时复制 fork 与工具无关：它在应用层之下运行，无需对客户工具进行插桩或协同设计，其优势会随各分支间保持共享的内存比例增加而扩大。此外，TTC 的跨平台可移植性（Linux、macOS、Windows）和无 root 部署，使个人计算机和边缘设备也能享有这些优势，从而有望在各种现实世界任务中收集分支智能体轨迹，用于训练持续改进的智能体模型。

V. 结论

我们介绍了 TimeTravelContainer（TTC），这是一种无需 root 权限的非特权容器运行时，能够以写时复制语义对运行中的工具会话进行实时分叉。TTC 无需修改内核、硬件虚拟化或根权限，可在所有主流平台上运行，为跨平台的智能体运行框架提供了可移植的基础。

在两个具有代表性的智能体电子设计自动化流程（即有状态工具工作负载）上进行评估后，TTC 相较于 Innovus 检查点/恢复实现了 $84.1\times$ 的克隆加速比，并将并发分支的内存占用降低了 31–72%，端到端速度最高提升 $9.4\times$ ，这表明实时分叉消除了智能体驱动探索面临的一项根本性扩展障碍。

除电子设计自动化外，TTC 还支持更广泛的智能体策略，将多轮交互与并行探索相结合。在任何由智能体驱动有状态、长期运行工具会话的领域——包括软件工程、数据库管理、机器人仿真和科学计算——都能受益于即时且节省内存的分叉。由于 TTC 的分叉原语与工具无关，且其部署无需特权，因此适用范围涵盖从共享计算集群到个人计算机和边缘设备的各种环境，为在多样化的现实世界任务中收集分支智能体轨迹创造了可能，从而训练持续改进的智能体模型。

参考文献

[1] J. Yang、C. E. Jimenez、A. Wettig、K. Lieret、S. Yao、K. Narasimhan 和 O. Press，“SWE-agent：智能体—计算机接口实现自动化软件工程”，载于神经信息处理系统进展（*NeurIPS*），2024。

[2] X. Wang、B. Li、Y. Song、F. F. Xu、X. Tang、M. Zhuge、J. Pan、Y. Song、B. Li、J. Singh、H. H. Tran、F. Li、R. Ma、M. Zheng、B. Qian、Y. Shao、N. Muennighoff、Y. Zhang、B. Hui、J. Lin、R. Brennan、H. Peng、H. Ji 和 G. Neubig，“OpenHands：面向人工智能软件开发者的通用型智能体开放平台”，arXiv预印本 arXiv:2407 16741，2024。。[3] H. Wu、Z. He、X. Zhang、X. Yao、S. Zheng、H. Zheng 和 B. Yu，“ChatEDA：由大语言模型驱动的电子设计自动化自主智能体 ”，IEEE集成电路与系统计算机辅助设计汇刊，2024，arXiv:2308.10204。[4] A. B. Kahng、S. Kundu 和 B. Pramanik，“MAGE：通过智能体多模态推理实现类人宏单元布局”，arXiv预印本 arXiv:2607.18536，2026。[5] B.-Y. Wu、C.-T. Ho、H. Yang、B. Khailany 和 V. A. Chhabria，“EvoDRC：用于自动设计规则检查违规修复的自我进化智能体框架”，载于亚洲及南太平洋设计自动化会议（ASP-DAC），2027，arXiv:2607.20019。[6]“Docker：加速容器应用开发”，https://www.docker.com。[7] “E2B”，https://e2b.dev，2024。[8] “AgentENV”，https://github.com/kvcache-ai/AgentENV，2026。[9] “CubeSandbox”，https://github.com/TencentCloud/CubeSandbox，2025。[10] “forkd”，https://github.com/deeplethe/forkd，2025。[11] Y. Huang、V. Srivatsa、A. Asch、H. T. Patwa 和 Y. Zhang，“TClone：面向计算机操作智能体的实时图形用户界面环境低延迟 fork”，arXiv预印本 arXiv:2605.17320，2026。[12] Y. Dong、J. He、S. Liu、Y. Hou、D. Du、Z. Xu、S. Yu、B. Yang、Y. Xia 和 H. Chen，“DeltaBox：通过毫秒级沙箱检查点与回滚扩展有状态人工智能智能体”，arXiv预印本 arXiv:2605 22781，2026。。[13] Google LLC，“gVisor”，https://gvisor.dev，2018。[14] C.-T. Ho、H. Ren 和 B. Khailany，“VerilogCoder：采用基于图的规划和基于抽象语法树的波形追踪工具的自主 Verilog 编码智能体”，载于AAAI人工智能会议，2025，arXiv:2408.08927。[15] S. Thakur、J. Blocklove、H. Pearce、B. Tan、S. Garg 和 R. Karri，“AutoChip：利用大语言模型反馈自动生成硬件描述语言”，arXiv预印本 arXiv:2311.04887，2023。[16] Y.-D. Tsai、M. Liu 和 H. Ren，“RTLFixer：利用大语言模型自动修复 RTL 语法错误”，载于设计自动化会议（DAC），2024，arXiv:2311.16543。[17] R. Qiu、G. L. Zhang、R. Drechsler、U. Schlichtmann 和 B. Li，“AutoBench：使用大语言模型为硬件描述语言设计自动生成并评估测试平台”，载于面向CAD的机器学习（MLCAD），2024，arXiv:2407.03891。[18] ——，“CorrectBench：使用大语言模型为硬件描述语言设计自动生成测试平台并进行功能自我纠正”，载于欧洲设计、自动化与测试会议（DATE），2025，arXiv:2411.08510。[19] W. Fang、M. Li、M. Li、Z. Yan、S. Liu、H. Zhang 和 Z. Xie，“AssertLLM：通过多大语言模型根据设计规范生成并评估硬件验证断言”，载于亚洲及南太平洋设计自动化会议（ASP-DAC），2025，arXiv:2402.00386。[20] A. Hosny、S. Hashemi、M. Shalan 和 S. Reda，“DRiLLS：用于逻辑综合的深度强化学习”，载于亚洲及南太平洋设计自动化会议（ASP-DAC），2020。[21] A. Grosnit、C. Malherbe、R. Tutunov、X. Wan、J. Wang 和 H. Bou Ammar，“BOiLS：面向逻辑综合的贝叶斯优化”，载于欧洲设计、自动化与测试会议（DATE），2022，arXiv:2111.06178。[22] X. Yao、Y. Wang、X. Li、Y. Lian、R. Chen、L. Chen、M. Yuan、H. Xu 和 B. Yu，“RTLRewriter：大模型辅助 RTL 代码优化的方法”，载于IEEE/ACM计算机辅助设计国际会议（ICCAD），2024，arXiv:2409.11414。[23] A. Ghose、A. B. Kahng、S. Kundu 和 Z. Wang，“ORFS-agent：用于芯片设计优化的工具使用型智能体”，载于面向CAD的机器学习（MLCAD），2025，arXiv:2506.08332。[24] H. Wu、H. Zheng、Z. He 和 B. Yu，“殊途同归：面向电子设计自动化的基于大语言模型的多智能体协作系统”，arXiv预印本 arXiv:2502.10857，2025。[25] C.-C. Chang、C.-T. Ho、Y. Li、Y. Chen 和 H. Ren，“DRC-Coder：使用大语言模型自主代理自动生成 DRC 检查器代码”，发表于*ACM/IEEE* 国际物理设计研讨会论文集（*ISPD*），2025 年，arXiv:2412.05311。[26] A. Allam、Y. Mansour 和 M. Shalan，“ASIC-agent：用于 ASIC 设计并经过基准评估的自主多智能体系统”，发表于 *IEEE*大语言模型辅助设计国际会议（*ICLAD*），2025 年，arXiv:2508.15940。[27] “CRIU：用户空间检查点/恢复”，https://criu.org。[28] C. Wang 和 Y. Zheng，“分叉、探索、提交：用于智能体探索的操作系统原语”，发表于*ASPLOS AgenticOS*研讨会，2026 年，arXiv:2602.08199。[29] A. Silberschatz、P. B. Galvin 和 G. Gagne，操作系统概念，第 10 版。Wiley，2018 年。[30] “ReDroid：Linux 上 GPU 加速的安卓容器”，https://github.com/remote-android/redroid-doc，2024 年。[31] B.-Y. Wu、C.-T. Ho、H. Yang、C.-C. Chang、A. Akkur、B. Khailany 和 V. A. Chhabria，“面向数字VLSI设计中DRC挑战的智能体解决方案”，发表于设计自动化会议（*DAC*）特别研究环节，2026 年。[32] V. A. Chhabria、A. Dey、A. B. Kahng、I. Savidis、P. Shrestha 和 B.-Y. Wu，“ResizerAgent：OpenROAD重定时器中用于时序优化的智能体策略选择”，发表于面向*CAD*的机器学习（*MLCAD*），2026 年。