探索 WebAssembly 在高性能计算中的应用

Mohak Chadha、Nils Krueger、Jophin John、Anshul Jindal、Michael Gerndt
{firstname.lastname}@tum.de 计算机架构与并行系统教席，慕尼黑工业大学，德国

Shajulin Benedict
shajulin@iiitkottayam.ac.in 计算机科学与工程系，印度科塔亚姆信息技术学院，喀拉拉邦

arXiv:2301.03982v1 [cs.DC] 10 Jan 2023

摘要

基于 Linux 内核提供的 命名空间 的容器化方法，在高性能计算社区中越来越受欢迎，既可用于隔离应用程序，也可用于打包和分发应用程序。然而，这些方法在高性能计算系统中的采用和使用面临多项挑战，包括难以在高性能计算资源上以非特权方式运行和构建科学应用容器镜像、高性能计算架构日益异构，以及无法访问仅在高性能计算系统上提供的专用网络库。基于容器的高性能计算应用开发所面临的这些挑战，与一种名为 *WebAssembly*（Wasm）的新型通用中间二进制格式所具备的诸多优势高度契合。这些优势包括轻量级的用户空间隔离机制，以及跨操作系统和处理器架构的可移植性。本文探索将 Wasm 用作基于消息传递接口的高性能计算应用的分发格式。为此，我们提出了 *MPIWasm*，这是一种面向基于消息传递接口的高性能计算应用的新型 Wasm 嵌入器，能够实现 Wasm 代码的高性能执行，消息传递接口调用开销低，并支持高性能计算系统上的高性能网络互连。我们在生产级高性能计算系统和 AWS Graviton2节点上，使用标准化高性能计算基准测试评估了*MPIWasm* 的性能和开销。实验结果表明，*MPIWasm* 在所有场景中的原生应用程序性能均具有竞争力。此外，我们观察到，与不同标准化基准测试所使用的静态链接二进制文件相比，Wasm 二进制文件平均小 139.5 倍。

CCS 概念： • **软件及其工程** → **进程管理**。

关键词：WebAssembly、Wasmer、Wasm、消息传递接口、高性能计算

允许免费为个人或课堂使用而制作本作品全部或部分内容的数字或纸质副本，但前提是不得为营利或商业利益而制作或分发副本，并且副本须附带本声明和首页上的完整引文。对于本作品中由 ACM 以外的其他方拥有版权的组件，必须尊重其版权。注明出处后，可以摘录。若要以其他方式复制、再版、发布到服务器或分发至列表，须事先获得明确许可和/或支付费用。请通过 permissions@acm.org 联系我们申请许可。*PPoPP '23，2023年2月25日至3月1日，加拿大魁北克省蒙特利尔*© 2023年 计算机协会。ACM ISBN 979-8-4007-0015-6/23/02。.. $15.00https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM参考文献格式：

Mohak Chadha、Nils Krueger、Jophin John、Anshul Jindal、Michael Gerndt 和 Shajulin Benedict。2023年。《探索 WebAssembly 在高性能计算中的应用》。载于 第 *28* 届 *ACM SIGPLAN* 并行编程原理与实践年会（*PPoPP '23*），*2023*年*2*月*25*日至*3*月*1*日，加拿大魁北克省蒙特利尔。ACM，纽约，美国，16页。https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 引言

Linux 容器凭借其可移植性和高可用性，已成为在云环境中开发、测试和部署从企业应用到 Web 服务等各类应用程序的事实标准 [29]。这是因为容器能够将应用程序及其定制的软件依赖项打包成一个整体，制作成易于部署的镜像。受其在云端广受欢迎的推动，容器也日益受到高性能计算社区的关注 [30, 74, 90]。对于高性能计算系统，容器为用户提供了灵活性，并允许他们为大型科学应用定义定制的软件栈，即 用户定义的软件栈 (UDSS)。此外，容器还能提供易于创建、可靠且可验证的环境，以便将来复现。为此，已有多种面向高性能计算的容器化解决方案问世，例如 Charliecloud [72], Shifter [49], Singularity [58], Podman [48], 和 Sarus [33]。与以往的方法不同，本文探讨采用一种名为 *WebAssembly* (Wasm) [52],的新技术 作为 Linux 容器的替代方案 [81], , 用于打包和分发高性能计算应用。

尽管容器日益普及，但容器在高性能计算系统中的采用和使用仍受到显著限制 [36]。这可归因于用户在高性能计算系统上为应用程序运行和构建容器镜像时普遍面临的诸多挑战。对于运行容器，大多数容器化解决方案都需要 root 权限；但由于高性能计算环境中的共享文件系统及其 UNIX权限，普通高性能计算用户无法获得此类权限。虽然 Singularity [58] 和 Podman [48] 等面向高性能计算的容器化解决方案支持无*root*权限容器，并通过 Fakeroot [61], 实现这一功能，但它们目前的实现并不支持 GPFS 等高性能计算系统中常见的分布式文件系统 [70, 80]。此外，正如 [71],所指出的，由无特权（普通）用户在高性能计算资源上构建符合开放容器倡议（OCI） [69] 的容器镜像

（即应用程序最终运行所在系统上的用户）非常困难，需要超级计算中心提供支持。这是因为 Docker [43] 等大多数容器构建解决方案也需要 root 权限。因此，大多数用户会使用自己的本地系统来构建或开发应用程序容器镜像，然后将构建好的镜像传输到高性能计算系统的登录/前端节点上执行。然而，这种做法会给基于容器的高性能计算应用开发带来若干问题。首先，高性能计算节点的异构性日益增强 [11]，采用了不同的处理器架构，例如 `x86_64` 或 `aarch64`，并配备了图形处理器等专用加速器。由于性能对高性能计算至关重要，使用特定处理器的微架构特性来编译应用程序十分重要。虽然可以通过 cross-compiling 高性能计算应用，或使用 QEMU 进行仿真，借助 `build-x` [44]，等插件为多个平台构建容器镜像，但高性能计算应用的构建流程并未广泛支持这种做法，而且还需要具备特定的 Linux 内核功能（`binfmt_misc` [62]）。此外，测试和开发高性能计算应用只能让用户深入了解目标系统。另外，大多数容器镜像的大小从数 MiB 到数 GiB 不等。因此，频繁地从本地系统向高性能计算系统进行网络传输会十分繁琐。其次，构建高性能计算应用需要访问专用网络库，并需要使用本地用户系统上没有的编译器许可证。最后，尽管不同容器化解决方案对容器化应用程序的性能几乎没有影响 [72, 75, 84]，但构建高性能计算应用的高性能容器镜像并非易事：它不仅学习门槛高，还要求用户了解目标系统上的特定消息传递接口库版本（例如 Open-MPI [10] 4.0）以及高性能网络互连硬件（例如 Intel OmniPath [4]）和库（例如 Intel Performance Scaled Messaging [5]）。基于容器的高性能计算应用开发所面临的这些挑战，与 Wasm [52] 旨在解决的诸多优势和核心问题密切相关。

Wasm 是一种低级、静态类型的通用二进制结构格式，用于在虚拟机中实现内存安全的沙箱化执行。它可在现代处理器架构和操作系统之间移植，执行速度快，并采用低级内存模型 [52]。尽管 Wasm 最初旨在网页浏览器中执行，但凭借其简洁性和通用性，已在非 Web 领域得到广泛采用和应用，例如无服务器计算 [78]，、边缘计算 [47, 53],和物联网 [51]。它不需要垃圾回收，并被设计为通用编译目标，能够成熟地支持采用 LLVM [59] 前端的编程语言，例如 C 语言、 C++、 C# 和 Rust [32, 35, 40, 45, 77]。

图1 展示了在高性能计算中使用 Wasm 的一般工作流程。开发人员可以在本地系统上提前将高性能计算应用编译为 Wasm（提前编译，AoT），然后

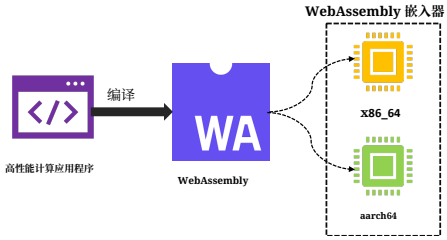


图 1. 高性能计算应用可以编译为*WebAssembly*，并分发到多个平台，在这些平台上，由支持它的*WebAssembly* 嵌入器高效执行。

将其分发到多个平台，而不是分发源代码或构建应用程序容器。通常，与原生`x86_64` 二进制文件 [52, 56, 91]相比，Wasm 二进制文件体积更小。随后，可使用独立的*Wasm* 嵌入器 [52]在任何平台上执行生成的二进制文件。*Wasm* 嵌入器主要有两项用途。首先，它为在平台上运行 Wasm 二进制文件提供隔离的执行环境。与采用不同 Linux 命名空间 [71] 实现隔离和安全性的基于容器的方法相比，Wasm 基于软件故障隔离（SFI） [85] 和控制流完整性（§2.2）在应用程序级别提供轻量级隔离。其次，它负责将 Wasm 二进制文件编译为本机机器码：既可以在执行时通过即时编译（JIT）引擎进行编译，也可以通过相同的 JIT 引擎或提前编译器进行提前编译（AoT）。请注意，普通用户即可执行 Wasm 二进制文件，且无需任何特权。目前已有若干开源独立嵌入器，例如`Wasmer` [87], `Wasmtime` [37], 和`Wasm3` [79] 。然而，它们都不支持执行高性能计算应用。

作为将 Wasm 引入高性能计算生态系统的第一步，本文仅关注基于消息传递接口 [6] 的高性能计算应用。我们选择消息传递接口，是因为它在高性能计算社区 [34]中广为人知且影响深远。为此，我们的主要贡献如下：

- 我们实现并推出了*MPIWasm*，这是一款基于 Wasmer [87]、面向基于 MPI 的高性能计算应用的新型*Wasm* 嵌入器 。*MPIWasm*能够高性能地执行 Wasm 代码，通过零拷贝内存操作降低消息传递接口调用的开销，并支持 Intel OmniPath [4] 等高性能网络互连。
- 我们通过在生产级高性能计算系统和 AWS Graviton2 [1] 节点上使用标准化高性能计算基准测试进行大量实验，分别展示了*MPIWasm* 的低开销和性能；这些节点分别基于 `x86_64` 和 aarch64 架构。
- 我们详细探讨了在高性能计算生态系统中使用 Wasm 的各种潜在未来方向。

本文其余部分结构如下。§2 详细介绍 Wasm。§3 中，我们详细介绍嵌入器*MPIWasm* 。实验结果见 §4。§5中，我们描述了在高性能计算生态系统中使用 Wasm 的各种潜在方向。§6 介绍了与我们工作相关的一些已有方法。§7总结全文并展望未来。

2 WebAssembly 入门

2.1 WebAssembly 概述

WebAssembly（Wasm）于2015年推出，作为 JavaScript 的替代方案，用于基于网页浏览器的应用程序。它取代了 asm.js [67], , 这是 Mozilla 先前的一次尝试，重点是 JavaScript 代码的一个子集，该子集可以进行提前编译优化。

应用程序编译为 Wasm 后，生成的二进制文件称为模块。Wasm 模块包含函数定义、全局变量声明、表以及线性内存地址空间。Wasm 中的所有应用程序代码都组织在函数中。Wasm 中的概念机器基于栈，不包含寄存器，因此所有指令都会从机器的栈中弹出操作数。不过，由于应用程序的控制流是模块的显式组成部分，且 Wasm 操作具有类型，因此可以静态预测程序中任意位置的栈布局，从而使编译器能够将栈语义转换为基于寄存器的指令集。与其他高级编程语言类似，Wasm 允许定义不限定于特定函数或代码块的全局变量。Wasm 模块中的表用于存储函数引用 [52]。Wasm 指令集架构目前仅支持四种变量数据类型：（i）i32，即 32 位整数；（ii）i64，即 64 位整数；（iii）f32，即 32 位 IEEE 754 浮点数；（iv）f64，即 64 位 IEEE 754 浮点数。构造复合类型时，通常会组合使用这些基本类型。

Wasm 可通过导入/导出系统，让模块与其嵌入器共享数据和代码。Wasm 模块中可能出现的所有函数定义，都可以从嵌入器导入，而不必在模块内部定义。同样，模块中存在的函数定义也可以导出，以便嵌入器使用（§2.3）。

2.2 WebAssembly 安全与沙箱模型

Wasm 使用软件故障隔离技术（SFI） [85]来对正在执行的 Wasm 模块进行沙箱隔离。默认情况下，Wasm 模块无法与宿主系统交互，也无法执行任何类型的输入/输出操作。Wasm 模块的代码若要发起任何系统交互，都必须通过从嵌入器（§2.1）导入的函数来完成。因此，嵌入器既可以充当转换层，也可以充当仲裁者，以强制执行隔离要求。作为转换器，它

```
1  (类型 (;1;) (函数 (参数 i32)(结果 i32)))
2  ...
3  (类型 (;5;) (函数 (参数 i32 i32) (结果 i32)))
4  ...
5  (类型 (;14;) (函数 (参数 i32 i32 i32 i32 i32 i32)
6                     (结果 i32)))
7  (类型 (;15;) (函数 (参数 i32 i32 i32 i32) (结果 i32)))
8  ...
9  (导入 "wasi_snapshot_preview1""path_open"
10       (函数 $__wasi_path_open(类型 22)))
11 (导入 "wasi_snapshot_preview1""fd_close"
12       (函数 $__wasi_fd_close(类型 1)))
13 (导入 "wasi_snapshot_preview1""文件描述符定位"
14       (函数 $__wasi_fd_seek(类型 23)))
15 (导入 "wasi_snapshot_preview1""fd_read"
16       (函数 $__wasi_fd_read (类型 15)))
17 (导入 "wasi_snapshot_preview1""proc_exit"
18       (函数 $__wasi_proc_exit(类型 0)))
19 ...
20 (导出 "_start"(函数 $_start))
21 (导出 "内存"(内存 0))
```

列表 1. 使用 ++ WASI-SDK 编译的 C 语言应用程序的 Wasm 模块示例，采用 WASI-SDK WebAssembly 文本格式（WAT） [68]。省略号表示为简洁起见而省略的部分。

可以为 Wasm 模块提供通用接口，即使底层系统可能具有不同的原生接口；作为仲裁者，嵌入器则可以根据应用级安全策略，限制 Wasm 模块对系统资源的访问。例如，嵌入器可以仅允许对特定目录中的文件进行文件输入/输出，从而将 Wasm 模块与文件系统的其余部分隔离开来。虽然这种做法在原理上类似于 Linux 上的 Seccomp-BPF 等内核级系统调用过滤技术 [83]，但在应用程序级别执行此类过滤，可以定义语义上更有意义的策略。

在 Wasm 中，所有内存访问都局限于模块的线性内存，且与代码空间分离。目前，Wasm 规范 [52] 支持使用 32位地址对模块可访问的内存进行索引。虽然这将单个模块的内存限制在 4GiB，但也能在运行时通过硬件加速对内存访问进行边界检查 [42]。如果嵌入器是一个具有 64位内存地址空间的进程，它就可以在自己的内存空间中安全地执行不可信的WebAssembly模块，而无需额外隔离，只需为该模块预留一段连续的虚拟内存即可。该范围内并非所有页都需要映射到物理内存，只需映射足够数量的页，以容纳模块在某一时刻所使用的内存即可。这确保了 Wasm 模块只能在自己的执行环境中运行，且无法破坏嵌入器的内存，因为任何越界内存访问都会导致页错误，随后可由嵌入器处理。此外，由于 Wasm 规范 [52]中的内存指令使用偏移量，因此在 Wasm 中无法读写任意内存位置。

在 C 语言程序生成的汇编代码中，函数调用通常表示为一条跳转指令，跳转到该函数第一条指令的地址。典型的漏洞利用方式是更改这个地址，从而控制程序的控制流。

```
1  typedef int MPI_Comm;
2  typedef int MPI_Datatype;
3  ...
4  int MPI_Init(整型* argc, CHAR*** argv);
5  int MPI_Finalize(void);
6  int MPI_Send(
7      const void* buf, int count, MPI_Datatype datatype,
8      int dest, int tag, MPI_Comm 通信器
9  );
10 整数 MPI_Recv(
11     无效* 缓冲区, int 计数, MPI_Datatype 数据类型,
12     整数 源, 整数 标签, MPI_Comm 通信器, MPI_Status* 状态
13 );
```

清单 2. 自定义 *MPIWasm* mpi.h 头文件摘录。

然而，Wasm 中的此类漏洞利用无法得逞，因为它具有控制流完整性，能够强制执行结构化的程序控制流。这主要有两个原因。首先，在 Wasm 中，函数由表中的一个索引表示（§2.1），这为表示函数地址增加了一层间接寻址。其次，Wasm 规范禁止构造任意内存地址 [42]，而且嵌入器与模块的内存相互隔离，因此无法覆盖函数指令。

2.3 WebAssembly 系统接口

由于 Wasm 最初是为网页浏览器设计的，因此面向 POSIX 环境、并能让 Wasm 模块在这些环境中执行的系统接口，并未纳入原始规范 [52]。为了解决这一问题，人们制定了 WebAssembly 系统接口（WASI）规范 [89]。WASI 规定了嵌入器为执行大多数 POSIX 应用程序所需实现的接口。实现 WASI 规范的嵌入器将能够运行任何使用 WASI-SDK [28]编译的通用应用程序。WASI-SDK 包含 Clang 编译器及其自带的、基于*musl libc*的 C 库；这些组件会调用从嵌入器导入的 WASI 系统调用，而不依赖 Linux 系统调用 [22]。请注意，由于*glibc* [26] 在 Linux 系统中无处不在，一些应用程序已开始依赖 glibc 特有的函数或行为。这些应用程序需要经过修改，才能编译为符合 WASI 的 Wasm 模块。

清单1 展示了一个使用 C语言++ 应用程序并通过 `WASI-SDK` 编译得到的 Wasm 模块，其采用 WebAssembly 文本格式（WAT）。WAT 是一种人类可读的格式，可供开发人员检查 Wasm 模块的源代码。可以看到，该模块包含若干个以整数作为参数和返回类型的函数（第 1-7 行）（§2.1），导入了 WASI 函数（第 9-18 行），并导出了其 `_start`（主函数）和内存（第 20-21 行）。导出这两个定义后，执行此模块的嵌入器便可以调用其入口函数，并对模块的线性内存进行读写。第 9-16 行的导入语句使 Wasm 模块能够打开并读取文件，而函数 `proc_exit` 则由嵌入器用于处理应用程序的终止，例如释放为模块预留的内存。若要让该模块

```
1  (导入 "env" "MPI_Init" (
2      func $MPI_Init (参数 i32 i32)(结果 i32)
3  ))
4  (导入 "环境" "MPI_Finalize" (函数 $MPI_Finalize (结果 i32)))
5  (导入 "env" "MPI_Send" (
6      func $MPI_Send (参数 i32 i32 i32 i32 i32 i32) (结果 i32)
7  ))
8  (导入 "env" "MPI_Recv" (
9      func $MPI_Recv (参数 i32 i32 i32 i32 i32 i32 i32)
10                    (结果 i32)
11 ))
```

清单 3. 清单 2 中所示函数对应的模块导入的 WAT 表示2。

执行，嵌入器必须实现导入的函数。

3 MPIWasm

在本节中，我们将详细介绍 MPIWasm——我们的嵌入器，用于执行调用消息传递接口标准中函数的 Wasm 模块。

3.1 概述

MPIWasm 的目的是支持在高性能计算系统上执行编译为 *Wasm* 的 *MPI* 应用程序。为促进其在高性能计算环境中的采用并确保其适用性，它 *(i)* 支持高性能执行编译为 *Wasm* 的、基于 *MPI* 的高性能计算应用程序（§3.3），(ii) 通过零拷贝内存操作降低消息传递接口调用的开销（§3.6），以及 (iii) 支持 InfiniBand 等高性能互连 [64] 和 Intel OmniPath [4]。高性能计算系统上的 MPI 库利用这些网络互连实现高性能进程间通信。为立即支持现代高性能计算系统中已有的网络互连，*MPIWasm*会在运行时链接到目标高性能计算系统上的 MPI 库，并在 Wasm 模块与主机[1] MPI 库之间提供转换层。因此，开发者无需了解目标高性能计算系统上使用的具体网络库或网络互连。根据所使用的主机 MPI 库（如 OpenMPI）[10] 或 MPICH [8]，*MPIWasm*需要分别构建。目前 MPIWasm支持这两个库。

我们的嵌入器目前支持执行以 C语言/C++ 编写且符合 MPI-2.2 标准的应用程序 [65]。我们有意在未来集成对 MPI-3.1 [66] 的支持，但这超出了本文的范围。我们选择专注于 C语言/C++ 应用程序，因为自 [59]llvm-8以来，LLVM/Clang 项目中的 Wasm 后端已趋于稳定和成熟。作为*MPIWasm*的基础，我们采用了名为`Wasmer` [87]的开源 Wasm 嵌入器。Wasmer 支持在三大主流平台上执行 Wasm 模块，即 Linux、Windows 和 macOS，并同时支持`x86_64` 和 `aarch64` 指令集架构。此外，它实现了 WASI 规范（§2.3），并提供

[1]我们将 target 和 host 这两个术语互换使用，指运行 *Wasm*模块的系统。

便捷的机制，用于定义提供给模块的其他函数。嵌入器功能的这种动态扩展，使我们能够将 MPI 函数添加到其提供给 Wasm 模块的功能中。为实现 *MPIWasm*，我们使用 Rust 编程语言，原因有二。首先，它在保证内存安全的同时，能够提供与 C语言/C++ 相当的高性能 [82]。其次，它在嵌入Wasmer 并将其用作库方面，拥有完善的支持和文档。

3.2 将 C/C++消息传递接口应用程序编译为 Wasm

大多数 MPI 应用程序都需要其执行环境提供 POSIX 功能，例如读取和写入文件描述符的能力。WASI（§2.3）定义了 WebAssembly 导出项，使以其为目标的 Wasm 模块能够调用 POSIX 系统所提供的 C 标准库中的大多数函数。为此，WASI-SDK [28]结合了 clang 编译器和 wasi-libc C 库，使仅使用 POSIX 函数而不依赖其他库的 C 语言应用程序能够编译为 Wasm。标准版 WASI-SDK 不支持编译 C 语言 MPI 应用程序。为此，我们实现了一个自定义的 mpi.h MPI 头文件，并将其添加到 WASI-SDK 中。该头文件包含不同 MPI 类型的定义，例如 MPI_Op、MPI_Comm 和 MPI_Datatype，以及 MPI_Status 结构体的定义。此外，它还根据 MPI-2.2 [65] 标准定义了 MPI 函数的签名。清单 2 展示了该头文件的一个节选。这是传统 MPI 库头文件的精简版本，其中大多数类型都定义为整数（§3.6）。将我们的头文件与 WASI-SDK 结合使用后，符合 MPI-2.2 标准的 C 语言 MPI 应用程序即可编译为 Wasm。此外，为了便于使用并推动采用，我们实现了一个自定义的基于 Python 的工具，以简化 MPI 应用程序的整个编译流程。清单 3 展示了与清单 2 中所示函数相对应的 Wasm 模块中不同的 MPI 专用导入项。MPIWasm 为这些导入项提供定义，从而支持基于 MPI 的高性能计算应用的执行。此外，它还支持 WASI 规范，使 MPI 应用程序能够使用 POSIX 功能。

3.3 使用高性能执行 Wasm 代码

Wasm 模块有多种执行策略，包括使用解释器 [79], 提前编译（AoT） [37], 或即时编译（JIT） [37]。然而，对于高性能计算系统而言，最有用的方法是在执行应用程序之前，将 Wasm 指令（Wasm 指令集架构）转换为主机的原生指令集，即提前编译。为此，*MPIwasm* 基于 Wasmer 提供的代码生成基础设施构建。 [87] Wasmer 目前支持三种编译器后端，即 Singlepass、[86],Cranelift [3], 和 LLVM。 [59]SinglePass 编译器旨在以线性时间生成机器码，且不会执行太多代码优化。Cranelift 编译器完全基于 Rust，且与 LLVM 类似。使用 Cranelift 时，WASM 指令首先被转换为 Cranelift 的中间表示（IR），即 Cranelift-IR；随后，在考虑微架构特定优化的基础上，将其转换为主机的原生指令集。另一方面，使用 LLVM 时，Wasm 指令集架构首先被转换为 LLVM-IR，然后生成本机机器码。Cranelift-IR 与 LLVM-IR 类似，但抽象层级更低，因此会妨碍中层代码优化。[2]在编译过程结束时，三种编译器都会生成一个共享对象，可通过 libloading 库使用快速的 dlopen 调用加载。 [27]

表 1. 比较 Wasmer [87]所支持的不同编译器后端在编译耗时和性能方面的表现（针对 HPCG [73] Wasm 模块）。该 Wasm 模块使用我们的 WASI-SDK (§3.3)生成，并通过 *MPIWasm* 在x86_64 系统上执行。

编译器	编译耗时（毫秒）	单核性能（每秒十亿次浮点运算）
Singlepass [86]	52 0.3769	
Cranelift [3]	150	1.3240
LLVM [59]	2811	1.5426

表1 展示了 Wasmer 支持的三种不同编译器在 HPCG 基准测试中的编译时间和运行时性能对比。虽然 LLVM 编译 Wasm 模块的速度最慢，但它使 HPCG 应用程序获得了最快的运行时性能。因此，我们选择 LLVM 作为 *MPIWasm*中的编译器后端。为抵消 LLVM 相较于另外两种编译器所需的较长编译时间，我们为生成的机器码实现了一种缓存机制。该缓存机制基于 FileSystemCache [46]，由 Wasmer提供。在我们的实现中，我们使用 Blake-3 哈希函数 [2]为每个 Wasm 模块生成哈希值。此外，我们将 LLVM生成的共享对象以该哈希值为名存储在本地文件系统中。因此，对 Wasm 模块的任何更改都会生成新的哈希值，并触发模块的重新编译。为此，在使用 *MPIWasm* 的系统上重复执行同一应用程序，不会产生执行时的重新编译开销。

3.4 使用 MPIWasm 实现文件系统隔离

由于在 Wasm 中，应用程序的所有系统交互都必须通过调用嵌入器实现的函数来完成（§2.2，§2.3），嵌入器因而能够对这些函数的使用施加额外限制，并检查传入的参数。在 Wasmer 中，所有导出函数

[2]关于 Cranelift-IR 和 LLVM-IR 的更详细讨论可参见此处 [25]。

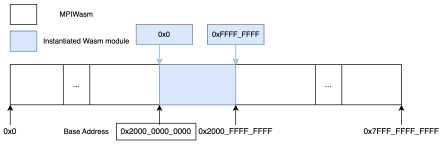


图 2. 已实例化 Wasm 模块的 *MPIWasm* 内存地址空间。所有针对 Wasm 模块线性地址空间的内存访问指令，其偏移量均相对于基地址给出。

处理文件输入/输出的函数会自行处理权限，与操作系统采用的权限机制相互独立。这种在进程内对文件系统访问进行的间接寻址，使 Wasmer 能够向 Wasm 模块呈现一棵虚拟目录树，其中只包含该模块获准访问的目录。此外，单个目录的访问权限可以比运行嵌入器的用户所拥有的权限更加严格。例如，用户可以对自己的主目录及其所有子目录拥有读写权限，但只向由嵌入器执行的 Wasm 模块授予其中某个特定子目录的只读访问权限。*MPIWasm* 通过 -d 标志提供此隔离功能，该标志向 Wasm 模块授予对指定目录的读写访问权限。请注意，暴露目录的完整绝对路径不会呈现给 Wasm 模块。在呈现给它的虚拟目录树中，所有获准访问的子目录都是根目录的直接子目录。通过这种方式映射目录路径，可以避免暴露目录完整路径中包含的信息，例如主目录路径中的用户名。

3.5 将 Wasm 转换为主机内存地址 ress

Wasm 安全模型的一项重要内容是将主机与模块的线性内存地址空间分隔开（§2.2）。由于确保能力限制得到遵守是 Wasm 嵌入器的职责，因此保护其数据结构，避免模块代码意外或恶意访问，至关重要。然而，这种分隔给支持 MPI 应用程序带来了挑战，因为 MPI API 要求库能够直接读写应用程序的内存。基于 Wasm 执行的 MPI 应用程序只能提供其自身线性内存地址空间中的内存地址，而目标 MPI 库需要的是主机内存地址空间中的地址。

执行 Wasm 模块时，*MPIWasm* 会在自身的地址空间中预留一部分供 Wasm 模块使用。因此，此范围内的每个字节都可以通过模块内存空间中的内存地址或嵌入器（宿主）的内存空间中的内存地址进行寻址。此外，在实例化模块的线性内存时，*MPIWasm* 会记录其基地址。随后，便可以将线性地址空间中的地址转换为 Wasm 模块的地址空间，反之亦然：将线性地址空间中的地址视为相对于模块基地址的偏移量即可。图2对此进行了说明。具体来说，*MPIWasm* 会直接将指向模块线性地址空间的 32 位 Wasm 指针转换为指向嵌入器地址空间的 64 位指针，反之亦然。为此，*MPIWasm* 会直接使用宿主系统上的 MPI 库，而无需将任何数据从模块的地址空间复制到其他位置，也就是说，它支持零拷贝内存操作。

我们的内存地址转换机制不会破坏内存安全，原因如下：(i) 恶意 Wasm 模块无法破坏控制流完整性（§2.2），且 (ii) 由于线性内存的大小始终已知，*MPIWasm*可以对所有内存访问执行运行时边界检查。因此，除非明确授予模块访问权限，否则模块无法访问嵌入器或底层操作系统的内存。

3.6 转换 MPI 数据类型

MPI 以库的形式实现，其中最常用的是 OpenMPI [10]，MPICH [8], 和 MVAPICH [9]。因此，它无法保证不同库之间具有应用程序二进制接口（ABI）和互操作性。这意味着更换 MPI 实现需要重新编译整个应用程序代码。ABI 不兼容的原因之一是，MPI 标准未为其数据类型（例如 `MPI_Op` ）规定明确的类型，其实现完全由 MPI 库决定。然而，由于 Wasm 模块不仅要能在不同 MPI 库之间移植，还要能在不同 CPU 架构之间移植，因此有必要在主机 MPI 库使用的数据类型与 *MPIWasm*向 Wasm 模块公开的数据类型之间添加一个抽象层。由于大多数 MPI 数据类型（例如`MPI_Comm`、`MPI_Datatype` 和 `MPI_Op`）对于应用程序而言是不透明的，且仅用作 MPI 函数的参数，因此可以实现这一抽象层。 *MPI-Wasm* 将大多数 MPI 数据类型定义为 Wasm 模块视角下的 32 位整数（列表 2），并将这些数据类型透明地转换为宿主端等效项（§3.7）。我们使用整数作为数据类型，因为 *MPIWasm* 在内部使用标识符来标识它代表模块创建的数据结构，以便与主机 MPI 库通信。

3.7 在*MPIWasm*中实现消息传递接口函数

Wasm 导入项通过要导入的定义的命名空间和名称来引用。默认情况下，编译 C/C++ 应用程序以生成 Wasm 时，所有未定义的符号都会作为模块导入项解析到`env` 命名空间中。清单3也展示了与*MPI* 标准相关的函数导入项。*MPI-Wasm* 为所有这些函数提供了定义，函数名称与原始 MPI 函数相同，并将它们导出到 `env` 命名空间。为实现这些函数，我们

将 §3.5 和 §3.6 中所述的内存地址转换与 MPI 数据类型转换结合起来。为此，我们维护了一个名为Env 的结构，用于存储这些转换所需的全局状态。该结构包含有关分配给 Wasm 模块的内存及其基址指针（§3.5）的信息，以及模块使用的不同数据类型（例如 MPI_Comm ）的信息。为直接使用主机 MPI 库，我们在MPI-Wasm [7] 中使用项目*rsmpi*。rsmpi 为 Rust 提供 MPI 绑定，并支持 OpenMPI [10] 和 MPICH [8]。它使用 rust-bindgen项目生成针对特定 MPI 库定制的外部函数接口。MPIWasm 中的每个 MPI 函数 都会使用适当的参数直接调用rsmpi中的对应函数。

对于 MPI-2.2 标准中的大多数函数，*MPI-Wasm* 会直接将执行委托给主机 MPI 库，但 MPI 函数 MPI_Alloc_mem 和 MPI_Free_mem 的实现方式有所不同。借助这些函数，可以分配供其他 MPI 函数使用的内存。当 Wasm 模块调用 MPI_Alloc_mem 时，它期望得到一个位于模块地址空间中的 32 位内存地址；而调用主机 MPI 库的 MPI_Alloc_mem 函数时，返回的却是一个位于嵌入器地址空间中的 64 位内存地址，该地址不在为 Wasm 模块预留的内存块内。为解决这一问题，*MPIWasm*仅在 Wasm 模块定义并导出 malloc 和 free 函数时，才支持 MPI_Alloc_mem 和 MPI_Free_mem。调用 MPI_Alloc_mem 时，*MPIWasm* 只需调用导出的 malloc，即可获得合适的 32 位模块内存地址。随后，该地址可用作 MPI_Alloc_mem 的返回值。我们以类似的方式实现 MPI_Free_mem。

3.8 局限性

Wasm 规范目前假定 [52]执行环境中多字节值采用小端字节序。通过允许主机 MPI 库直接访问 Wasm 模块的内存，我们假定模块地址空间和嵌入器地址空间中的值采用相同的字节序。因此，*MPIWasm* 不支持大端 CPU 架构。这并不是一个缺点，因为高性能计算系统中的大多数处理器架构都采用小端字节序。此外，由于 Wasm 模块当前采用线性的 32 位内存空间，编译为 Wasm 的高性能计算应用的内存不能超过 4GiB。对 64 位内存地址的支持是 Wasm 规范发展的重要里程碑，Wasm Memory64 提案 [20], 对此进行了重点说明，但不在本工作的范围内。

4 实验结果

本节展示嵌入式 MPIWasm 在不同处理器架构上的性能结果。对于

```
1    mpirun -np  <进程数 > ./mpiWasm mpi-app.wasm  <参数 >
```

代码清单 4. 使用 *MPIWasm* 执行编译为 Wasm 的 MPI 应用程序。

表 2. 比较不同 MPI 应用程序的原生动态链接、静态链接和 Wasm 二进制文件的大小。原生应用程序针对 x86_64 架构进行编译。

应用程序	原生动态大小（KiB）	原生静态大小（MiB）	Wasm 大小（KiB）
英特尔 MPI 基准测试 [55]。	1087	27	893
HPCG [73]。	164	26	722
IOR [60]。	364	16	315.32
IS [31]。	36	15	57.88
DT [31]。	40	15	49.51

在所有实验中，我们遵循报告结果的最佳实践 [54]。

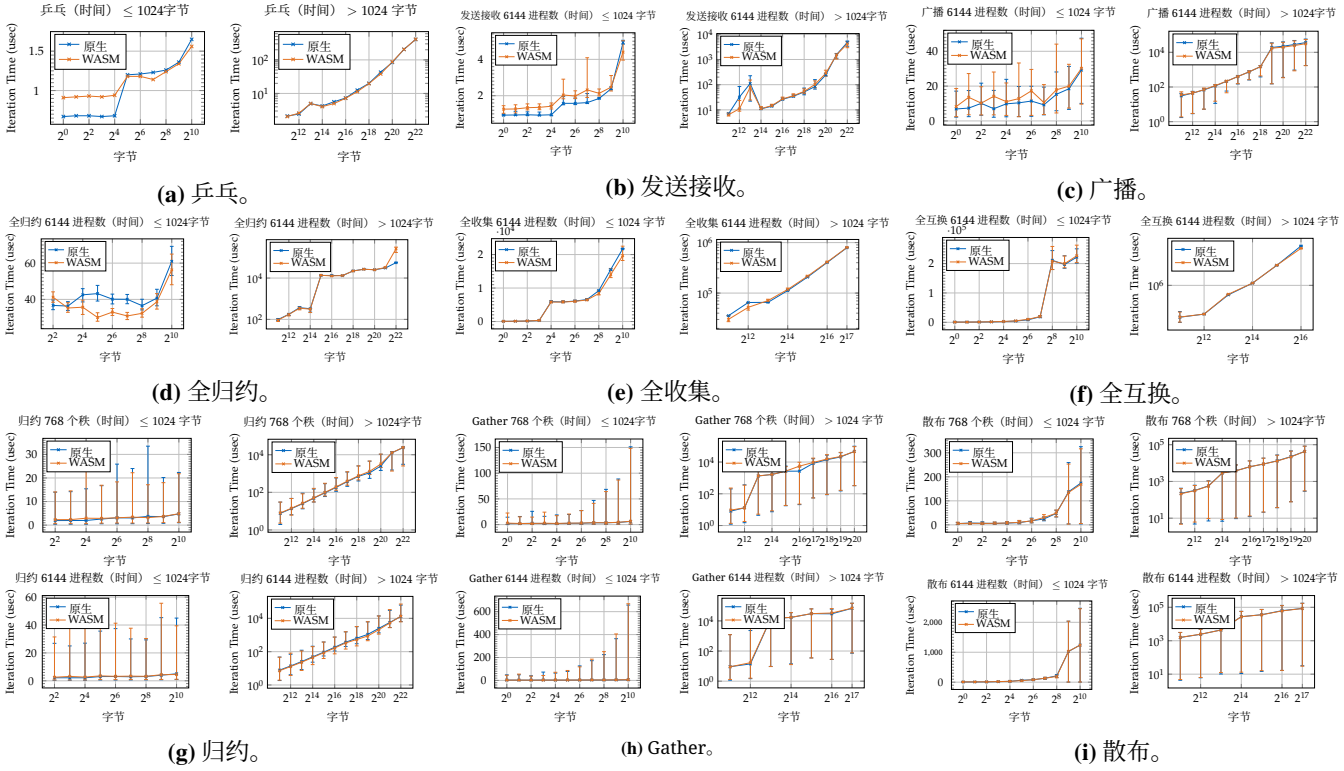
4.1 系统描述

为了分析我们实现的 Wasm 嵌入器的性能，我们使用了两个系统。首先是位于我们研究所的生产级高性能计算集群，即 SuperMUC-NG。其次是配备 Graviton2 处理器的 AWS EC2 虚拟机（ VM ）实例 [1]。我们的高性能计算集群包含八个岛，共有 6480 个基于英特尔 Skylake-SP 架构的计算节点。每个计算节点有两个处理器插槽，配备两颗英特尔至强白金 8174 处理器，每颗处理器有 24 个核心，主存总容量为 96 GiB。每个核心的标称运行频率为 3.10 GHz。系统已禁用超线程和 Turbo Boost。系统内部互连采用高速 Intel Omni-Path [4] 网络，带宽为 100 Gbit/s。此外，我们的集群提供基于 Lenovo DSS-G、用于 IBM Spectrum Scale [19] 的通用并行文件系统，总带宽为 200 GiB/s。在实验中，我们最多使用高性能计算系统的 128 个节点，即 6144 个核心。另一方面，基于 64 位 ARMv8-A Neoverse-N1 [24] 架构的 AWS Graviton2 处理器包含 32 个核心，每个核心的标称频率为 2.50 GHz，主存总容量为 64 GiB。对于 Graviton2 处理器，我们将实验限制在单个节点上。

4.2 高性能计算基准测试

在使用 *MPIWasm* 进行实验时，我们采用英特尔 MPI 基准测试 [55], 、NASA 高级超级计算部（NAS）并行基准测试套件（NPB）中的两项基准测试 [31], 、IOR 基准测试 [60], 以及高性能计算梯度（HPCG）基准测试 [73]。

英特尔 MPI 基准测试用于测量一系列消息大小下的点对点和全局通信操作的 MPI 性能。我们使用这些基准测试，因为它们能够表征集群的性能，并反映所用 MPI 实现的效率。NPB 基准测试套件包含一组旨在评估高性能计算集群整体性能的基准测试。由于目前将 Fortran 编译为 Wasm 的支持尚处于早期阶段（§5），因此只使用了该套件中以纯 C语言编写的整数排序（IS）和数据传输（DT）

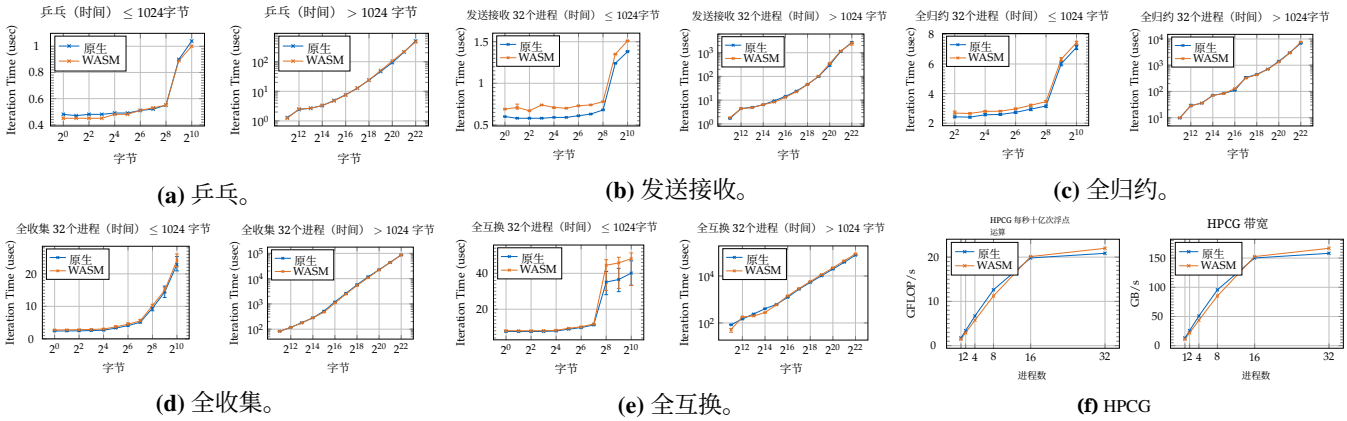


(a) 乒乓。 **(b)** 发送接收。 **(c)** 广播。

(d) 全归约。 **(e)** 全收集。 **(f)** 全互换。

(g) 归约。 **(h)** Gather。 **(i)** 散布。

图 3. 在我们的高性能计算系统上，MPIWasm 的英特尔 MPI 基准测试与其原生执行性能对比。



(a) 乒乓。 **(b)** 发送接收。 **(c)** 全归约。

(d) 全收集。 **(e)** 全互换。 **(f)** HPCG

图4. 在 AWS Graviton2 处理器上，MPIWasm 与原生执行方式在部分英特尔 MPI 基准测试和 HPCG 中的性能对比。

基准测试，因为它们均以纯 C语言编写。IS基准测试通过分桶方式在所有参与进程间并行排序整数，而 DT 基准测试则通过在节点拓扑结构中发送数据，测试高性能计算集群的通信能力以及 64位浮点运算性能。DT 基准测试使用黑洞（bh）、白洞（wh）和乱序（sh）这几种拓扑结构。在实验中，IS 和 DT 基准测试分别使用 C类和B类。IOR

基准测试衡量了 MPI 进程可用的文件系统I/O性能。它支持多个后端，这些后端利用不同的 API 执行系统I/O。在我们使用 *MPIWasm*进行的实验中，我们采用 POSIX API后端，因为 POSIX 文件系统 API 已包含在 WASI 规范中（§2.3，§3.2）。HPCG 基准测试旨在通过使用共轭梯度法求解线性方程组，评估高性能计算系统在实际应用中的性能。在我们的实验中，我们使用 HPCG 默认提供的问题规模

。请注意，在我们的实验中，IntelMPI 和 NAS 并行基准测试分别使用 2019 Update 6 和 3.3.1 版本。

4.3 实验设置

在所有实验中，我们均在纯 MPI 配置下运行基准测试，不使用 OpenMP 进行共享内存并行化，因为 MPIWasm 目前尚不支持此功能。我们使用 OpenMPI-4.0 作为 MPI 库，因为它在我们的高性能计算系统上可用，并且可以轻松安装在 AWS Graviton2节点上。在我们的高性能计算系统（§4.1）上编译原生应用程序时，我们使用 clang-11编译器；在 AWS Graviton2 节点上则使用 gcc7.1编译器。在这两种情况下，应用程序均使用 -O3 优化标志进行编译。将不同的基准测试编译为 Wasm 时，我们使用 clang-11编译器以及定制的 WASI-SDK，并在两种测试系统上均使用 -O3 -msimd128 标志（§3.2）。-msimd128 标志可启用 Wasm 中 SIMD 指令的生成。我们仅在本地系统上将应用程序编译为 Wasm 一次，然后直接通过 MPIWasm 在不同的测试系统上运行。我们在本地系统上为不同平台构建 MPIWasm，即 `x86_64` 和 aarch64，并使用 OpenMPI 为 rsmpi 生成绑定（§3.7）。随后，我们按照清单 4 所示，直接在测试系统上运行编译为 Wasm 的应用程序。每个 MPI进程对应一个嵌入器实例，并拥有自己的 Wasm 模块。原生应用程序则直接使用 mpirun 运行。

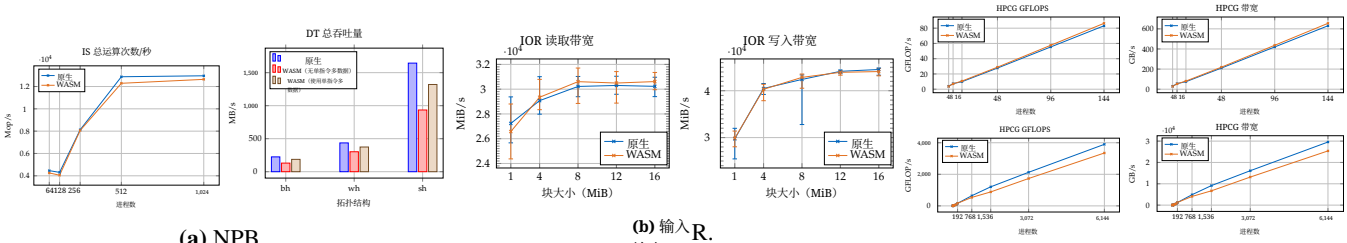
4.4 比较 Wasm二进制大小

表2 展示了不同应用程序绝对二进制大小的对比。通过向`-static` 编译器`clang-11` （§4.3）提供该标志，并使用所需库的静态版本（如`libmpi.a`、`libopen-rte.a`和`libz.a`）链接不同的应用程序，我们生成了二进制文件的静态版本。为此，我们对不同应用程序使用的`Make` [50] 和`CMake` [39]文件进行了必要的修改（§4.2）。尽管基于栈的指令集和紧凑二进制格式使 Wasm 有望为相同应用程序生成比原生动态链接二进制文件更小的二进制文件， [52], 但我们使用的五个应用程序中有三个，编译为 WebAssembly 后的二进制大小反而大于对应的原生动态链接二进制文件。虽然 Wasm 在逐函数表示上可以更精简，但实际上，原生动态链接二进制文件能够依赖系统中已有的常用库，从而抵消这一优势。例如，原生二进制文件可以动态链接`glibc`，而 Wasm 二进制文件必须静态包含`wasi-libc` （§2.3）中的函数。不过，与容器相比，Wasm 二进制文件要小得多，因此更适合在高性能计算环境中进行应用程序分发。此外，Wasm 二进制文件的平均大小比.小 139 不同应用程序的静态链接二进制文件。这是因为链接器（即 lld）在静态链接过程中，会将应用程序所用不同库中的所有库例程复制到二进制文件中。

4.5 MPIWasm 基准测试

图 3 和图 4 展示了不同英特尔 MPI 基准测试在原生执行与通过 *MPIWasm* 分别在我们的高性能计算系统和 AWS Graviton2 处理器上执行时的迭代时间。使用 *MPIWasm* 执行时，迭代时间不包括将 Wasm 模块编译为本机机器码所需的时间（§3.3）。为避免重复，我们省略了 Graviton2 处理器的部分结果。图中的误差线表示英特尔 MPI 基准测试报告的迭代时间最小值和最大值，图中的点则表示基准测试报告的平均时间。

对于乒乓基准测试，在所有消息大小下，使用 MPIWasm 后，`x86_64` 系统的几何平均（GM）平均性能下降为 0.05 倍，而 aarch64 系统的几何平均加速比为 1.01 倍（图 3a、4a）。我们通过将基准测试报告的原生执行指标 t_avg_us 除以使用 MPIWasm 执行时报告的数值，再对所得数值计算几何平均数，得出该值。计算性能下降时，我们将所得几何平均数减去 1。我们观察到，在高性能计算系统和 Graviton2 处理器上原生执行乒乓基准测试时，最大带宽分别为 12.80 GiB/s 和 10.98 GiB/s。另一方面，使用 MPIWasm 时，我们在这两个系统上观察到的最大带宽分别为 13.44 GiB/s 和 10.61 GiB/s。对于 SendRecv 基准测试，在所有消息大小下，使用 MPIWasm 时，`x86_64` 和 aarch64 系统的几何平均性能下降分别为 0.06 倍和 0.07 倍（图 3b、4b）。基准测试的原生版本在这两个系统上观察到的最大带宽分别为 7.24 GiB/s 和 11.01 GiB/s，而使用 MPIWasm 时观察到的最大带宽分别为 7.50 GiB/s 和 10.83 GiB/s。对于集合通信的 Broadcast 例程，如图 3c 所示，在 128 个节点上，所有消息大小下使用 MPIWasm 的平均几何平均性能下降为 0.13 倍。如图 3d 和 4c 所示，对于集合通信的 AllReduce 例程，在所有消息大小下使用 MPIWasm 的平均几何平均性能下降分别为 0.06 倍和 0.10 倍。对于 AllGather，使用 MPIWasm 时，在所有消息大小下，高性能计算系统和 Graviton2 处理器的平均几何平均性能下降分别为 0.06 倍和 0.09 倍（图 3e、4d）。类似地，对于 Alltoall 集合通信例程，如图 3f 和 4e 所示，在所有消息大小下，两个系统使用 MPIWasm 的平均几何平均性能下降均为 0.10 倍。对于高性能计算系统的 16 个节点，如图 3g、3h 和 3i 所示，例程 Reduce、Gather 和 Scatter 在不同消息大小下的平均几何平均性能下降分别为 0.12 倍、0.14 倍和 0.05 倍。另一方面，对于 128 个节点，



(a) NPB. **(b)** 输入输出R. **(c)** HPCG。

图 5. 在我们的高性能计算系统上，*MPIWasm* 与原生执行的标准化高性能计算基准测试性能对比。

我们观察到这三个例程的平均几何平均性能下降分别为 0。05 倍、0。10 倍和 0。08 倍。使用编译为 Wasm 的英特尔 MPI 基准测试来测试*MPIWasm*的结果表明，在 Wasmer [87]中调用宿主函数的机制以及 *MPI-Wasm*中实现的转换层都不会给 MPI 通信带来显著开销（§3.5、§3.6、§3.7）。我们在 §*4.6*中进一步讨论MPIWasm 的转换开销。总体而言，我们的结果表明，在 *x*86_64 和 aarch64架构上，MPI- Wasm 在不同 MPI 例程上的性能接近原生水平。

在我们的高性能计算系统上，*MPIWasm*运行IS 和DT 基准测试的性能如图 5a所示。对于 IS基准测试，使用 *MPIWasm*时，所有进程的平均性能为每秒 8260 兆次操作，而原生执行的平均值为每秒 8546 兆次操作。对于采用不同拓扑结构（§4.2）的 *DT* 基准测试，与原生执行相比，使用*MPIWasm* 会降低吞吐量。DT基准测试会执行大量成对比较操作，这些操作能够从使用单指令多数据指令进行的向量化中显著受益。为展示单指令多数据对 DT基准测试的影响，我们通过禁用和启用单指令多数据指令的生成，将其编译为 Wasm。启用单指令多数据的 Wasm 版本DT 基准测试的吞吐量比未启用单指令多数据的 Wasm 版本高 1.36 倍（图5a）。与原生版本的DT 基准测试相比，性能差异可归因于 Wasm 规范仅支持 128 位单指令多数据指令， [52] 而现代英特尔处理器支持 512 位单指令多数据指令 [38, 57, 76] （§3.3）。Wasm 支持更宽位数的单指令多数据是其路线图上的一个重要里程碑，但不在本工作的讨论范围内（§5）。

图5b 展示了 IOR基准测试中所有 MPI 进程可用的总读写带宽。图中的数据点表示基准测试报告的平均带宽，误差线则表示在块大小相同的情况下，基准测试所有迭代中观测到的最大和最小带宽。在我们的高性能计算系统中使用四个节点时，采用 4.1 节所述配置运行 IOR 时，可达到的带宽上限为 400 GBit/s (≈ 47684 MiB/s)。与基准测试的原生执行相比，MPIWasm 在所有块大小下的平均读带宽（29411 MiB/s）和写带宽（40206 MiB/s）相近。对 MPIWasm 的文件系统I/O性能进行测试表明，Wasmer 为实现文件系统隔离而采用的用户空间权限处理和虚拟目录树（§3.4），在通过 POSIX 文件系统 API 执行输入输出时，不会对可达到的带宽产生显著影响。对于 HPCG 基准测试，在最多 192 个 MPI 进程的范围内，使用 MPIWasm 执行时的性能与其在高性能计算系统和 Graviton2 处理器上的原生执行相近（图 5c 和图 4f）。随着进程数增加，HPCG 基准测试的原生执行性能超过了 MPIWasm 执行，如图 5c 所示。对于 6144 个 MPI 进程，使用 MPIWasm 执行时，每秒十亿次浮点运算降低了 14%。这可归因于 HPCG 基准测试需要极其频繁的通信 [63]。HPCG 会反复使用 AllReduce 例程，对所有 MPI 进程上的一个 DOUBLE 类型变量进行归约，以完成向量点积运算。随着进程数增加，调用全归约例程的次数也随之增加。例如，使用 768 个进程执行 HPCG 时，调用全归约的次数是使用 192 个进程执行时的四倍。因此，MPIWasm 中反复进行的数据类型转换会增加调用集合通信例程的成本，导致性能下降（§4.6）。

4.6 分析数据类型转换开销

为衡量*MPIWasm*中的数据类型转换开销，我们实现了一个自定义的乒乓应用程序，在两个进程之间发送和接收不同大小的消息，并遍历不同的 MPI 数据类型，即BYTE、CHAR、INT、FLOAT、DOUBLE和LONG。随后，我们对 MPIWasm中的*Send*例程进行插桩，以确定不同数据类型转换的延迟。最后，我们在高性能计算系统上运行该应用程序。图6 展示了*MPIWasm*中不同数据类型和消息大小对应的转换开销。我们观测到平均开销为 85.44ns，

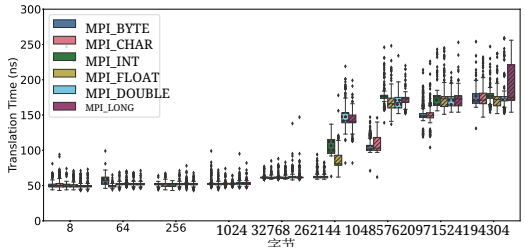


图 6. 比较*MPIWasm*中的数据类型转换开销。

84.72ns、99.78ns、96.32ns、103.35ns 和 104.79ns；对于所有消息大小，这些数值分别对应 MPI 数据类型 BYTE、CHAR、INT、FLOAT、DOUBLE和LONG。当消息大小超过 256KB 时，我们观察到转换开销有所增加。这可能是因为从Env 结构获取读锁的延迟增加；该结构负责维护 *MPIWasm*（§3.7）中用于转换的全局状态。

5 展望未来：Wasmand 高性能计算

在本节中，我们将重点介绍并探讨 Wasm 社区提议对当前 Wasm 规范 [52] 所作的不同扩展。这些扩展可在嵌入器中实现，以提升高性能计算应用的性能和可移植性。

Wasm 模块的受控线程管理。Wasm 线程提案 [16] 为将 Wasm 用于多线程算法奠定了基础。该提案使 Wasm 模块能够定义共享内存，从而告知嵌入器该模块预期由多个线程访问内存。为确保对共享内存的多线程访问安全，该提案还定义了 Wasm 原子指令，可用于在模块的函数中实现锁和原子数据结构。为了让高性能计算应用能够利用线程提案新增的功能，嵌入器需要添加一个 API，使 Wasm 模块能够自行创建额外线程。在嵌入器中实现 POSIX 线程 [15] 和 OpenMP [41] API，将使现有高性能计算应用中的线程代码能够兼容。

Wasm 扩展 SIMD。当前的 Wasm SIMD 提案 [52] 仅规定了 128 位 SIMD 指令，而现代 CPU 支持更高位宽的 SIMD，例如针对 x86_64 的 AVX-512 指令集扩展，其中规定了 512 位 SIMD 指令。为此，Wasm 灵活向量提案 [13] 旨在支持宽度大于 128 位的 SIMD 指令。此外，Wasm 宽松 SIMD 指令 [21]旨在使利用定义不够明确的硬件 SIMD 指令成为可能，即这些指令的舍入行为与 Wasm 规范不同。在嵌入器中实现这些提案，将使包含可向量化代码的高性能计算应用编译而成的 Wasm 模块能够更充分地利用 SIMD 指令可在现代 CPU 架构中使用。

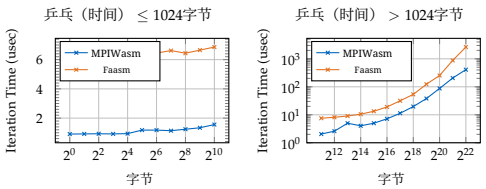


图 7. 比较 *MPIWasm* 和 Faasm 的性能 [78]。

Wasm PGAS。分区全局地址空间（PGAS）是一种用于并行分布式内存应用的编程模型，它引入了一个跨越多个进程本地内存的内存地址空间。进程可以使用来自这个全局地址空间的内存地址，读取和写入其他进程的内存。由于 Wasm 已定义导入和定义内存的概念，因此可以扩展嵌入器，为 Wasm 模块提供非本地内存。为了支持这一用例，需要实现 Wasm 多内存提案 [14] needsto，该提案允许 Wasm 模块定义或导入多个内存。

Wasm 模块动态链接。目前已有一些工作致力于为 Wasm 模块之间的动态链接建立 ABI [12], ，但尚未将其标准化。支持动态链接将大幅减小更复杂应用程序的 Wasm 二进制文件体积，因为它们不再需要静态链接 wasi-libc 的部分内容。对于高性能计算，这将使 MPIWasm 能够提供基础线性代数子程序等常用库。将动态链接与为 Wasm 模块提供存储库（如 WAPM）的工作相结合 [88]，或可实现 Wasm 应用程序的自动依赖管理。

将 Fortran 应用程序编译为 Wasm 目前，将基于 Fortran 的应用程序编译为 Wasm 的支持还处于非常初期的阶段，目前已知的尝试只有一种，该尝试基于 Dragon-egg[17]。不过，Memory64 提案的实现 [20] 应能使现有的 Fortran LLVM 编译器（如 F18）可用于轻松地将基于 Fortran 的应用程序编译为 Wasm [18]。

Wasm 与加速器 模块执行提示提案 [23] 指出了 Wasm 规范需要进行哪些修改，才能支持在 GPU 等硬件加速器上执行 Wasm 模块。在嵌入器中实现该提案，将能够兼容现有的基于 GPU 的高性能计算应用程序。

6 相关工作

高性能计算应用的打包与分发解决方案。近期，出现了若干面向高性能计算的工具，例如 Charliecloud [72] 和 Singularity [58]，可通过容器化分发高性能计算应用。相比之下，我们利用通用二进制指令格式 Wasm 来打包和分发高性能计算应用。此外，

容器化需要针对不同平台构建高性能计算应用容器，而高性能计算应用只需编译为 Wasm 一次，即可借助支持 Wasm 的嵌入器在任何平台上执行。

MPI 与 WebAssembly。据我们所知，*Faasm* [78] 是唯一支持执行编译为 Wasm 的 MPI 应用程序的计算平台。它基于一个名为`Faabric`的 gRPC 分布式消息传递库，并包含 Wasm 运行时、工作负载调度器和分布式状态存储。要在 Faasm 上运行应用程序，必须先将其编译为 Wasm 并上传到共享函数存储。之后，可通过 HTTP 请求等事件调用该应用程序。为支持 MPI 应用程序，Faasm 仅在其消息传递库和自有工作负载调度器之上实现了 MPI-1 规范的一个子集。此外，它也不支持英特尔 MPI 基准测试（§4.1）所需的用户定义通信器。相比之下，我们采用相反的方式，*MPIWasm* 构建于现有原生 MPI 库之上，使 Wasm 模块能够高效调用其中的函数。图7 展示了*MPIWasm* 与 Faasm 在乒乓基准测试（§4.2）中的性能对比。使用*MPIWasm*，我们在所有消息大小下相较于 Faasm 实现了 4.28 倍的几何平均加速比。

7 结论与未来工作

本文迈出了将 WebAssembly 引入高性能计算生态系统的第一步，并介绍了 *MPIWasm*：一种 Wasm 嵌入器，可在不同处理器架构上高性能地执行编译为 Wasm 的 MPI 应用程序。未来，我们计划扩展 *MPIWasm*，使其支持高性能计算系统中的图形处理器等加速器 [23]。

8 致谢

我们感谢指导人 Milind Chabbi 协助准备本文的最终版本。此外，我们感谢匿名审稿人提出的深刻见解和宝贵反馈，这些意见显著提升了本文的质量。本研究获得德国联邦教育与研究部（BMBF）资助，资助项目属于 Software Campus 计划。

参考文献

[1] [无日期]。 *AWS Graviton 2* 处理器。 https://aws.amazon.com/ec2/graviton/ [2] [无日期]。 *Blake-3* 哈希函数。 https://github.com/BLAKE3-team/BLAKE3[3] [无日期]。 *Cranelift* 编译器。 https://github.com/bytecodealliance/wasmtime/tree/main/cranelift[4] [无日期]。 *IntelOmni-PathFabric*。 https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-fabric-software-components.html [5] [无日期]。 英特尔 *Performance Scaled Messaging 2*。 https://github.com/cornelisnetworks/opa-psm2/blob/master/README

[6] [n.d.]. MPI：消息传递接口标准。https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf [7] [n.d.]. Rust 的 MPI 绑定。 https://github.com/rsmpi/rsmpi [8] [n.d.]. MPICH：高性能、可移植的 MPI。https://www.mpich. org/ [9] [n.d.]. MVAPICH：基于 InfiniBand、Omni-Path、以太网/iWARP 和 RoCE 的 MPI。 http://mvapich.cse.ohio-state.edu/ [10] [n.d.]. OpenMPI：开源高性能计算。 https: //www.open-mpi.org/ [11] [n.d.]. Top500 榜单。 https://www.top500.org/lists/top500/2022/06/ [12] [n.d.]. WebAssembly 动态链接。https://github.com/WebAssembly/tool-conventions/blob/master/DynamicLinking.md [13] 2020. WebAssembly 灵活向量提案。https://github.com/WebAssembly/flexible-vectors/blob/master/proposals/flexible-vectors/Overview.md[14] 2020. WebAssembly 多内存提案。 https://github.com/WebAssembly/multi-memory/blob/master/proposals/multi-memory/Overview.md [15] 2020. POSIX 线程。 https://man7.org/linux/man-pages/man7/ pthreads.7.html [16] 2020. WebAssembly 线程提案。https://github.com/WebAssembly/threads/blob/master/proposals/threads/Overview.md [17] 2021. 将 Fortran 应用程序编译为 Wasm。https://github.com/StarGate01/Full-Stack-Fortran[18] 2021. Fortran 与 WebAssembly。 https://chrz.de/2020/04/21/fortran- in-the-browser/[19] 2021. Lenovo DSS-G。https://tinyurl.com/ytmjjauy [20] 2021. WebAssembly Memory64 提案。https: // github.com/WebAssembly/memory64/blob/master/proposals/memory64/Overview.md [21] 2021. WebAssembly 宽松 SIMD 提案。 https://github.com/WebAssembly/relaxed-simd/blob/main/proposals/relaxed-simd/Overview.md[22] 2021. WASI Libc。 https://github.com/WebAssembly/wasi-libc [23] 2021. Wasm 与 GPU。 https://github.com/WebAssembly/design/ issues/1448[24] 2022. ARM Neoverse-N1 架构。https://www.arm.com/products/silicon-ip-cpu/neoverse/neoverse-n1 [25] 2022 . Cranelift-IR 与 LLVM-IR 的比较。https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/docs/compare-llvm.md[26] 2022. GNU C 库（glibc）。 https://www.gnu.org/software/libc/ [27] 2022. libloading 库。 https://docs.rs/libloading/latest/ libloading/ [28] 2022. WASI SDK。 https://github.com/WebAssembly/wasi-sdk [29] 14 拥抱容器技术的科技公司。 [n.d.]. https: //learn.g2.com/container-technology. 访问日期：09/29/2021。[30] Subil Abraham、Arnab K Paul、Redwan Ibne Seraj Khan 和 Ali R Butt。2020。关于在高性能计算环境中使用容器。载于《2020 IEEE 第 13 届国际云计算大会（CLOUD）》。IEEE，284–293。 https://doi.org/10.1109/ CLOUD49709.2020.00048[31] D. H. Bailey、E. Barszcz、J. T. Barton、D. S. Browning、R. L. Carter、L. Dagum、R. A. Fatoohi、P. O. Frederickson、T. A. Lasinski、R. S. Schreiber、H. D. Simon、V. Venkatakrishnan 和 S. K. Weeratunga。1991。NAS 并行基准测试——摘要和初步结果。载于《1991 年 ACM/IEEE 超级计算大会论文集》（美国新墨西哥州阿尔伯克基）（Supercomputing '91）。ACM，美国纽约州纽约，158–165。https://doi.org/10.1145/ 125826.125925[32] JF Bastien、Thomas Lively 和 Heejin Ahn。2019。LLVM WebAssembly 后端。https://github.com/llvm/llvm-project/blob/main/llvm/lib/Target/WebAssembly/README.txt

[33] Lucas Benedicic、Felipe A Cruz、Alberto Madonna 和 Kean Mariotti。2019。Sarus：面向高性能计算系统的高度可扩展 Docker 容器。载于《国际高性能计算大会》。Springer，46–60。https://doi.org/10.1007/978-3-030-34356-9_5[34]David E Bernholdt、Swen Boehm、George Bosilca、Manjunath Gorentla Venkata、Ryan E Grant、Thomas Naughton、Howard P Pritchard、Martin Schulz 和 Geoffroy R Vallee。2017。美国百亿亿次级计算项目中的 MPI 使用情况调查。《并发与计算：实践与经验》（2017）。https: //doi.org/ 10 1002 4851 . /cpe. Blazor。n.d. 。[35] [ ]. https://dotnet.microsoft.com/apps/aspnet/web-apps/ blazor. 访问日期：09/27/2021。[36] Brian Austin 等。 [n.d.]. NERSC-10 工作负载分析。https://portal.nersc.gov/project/m888/nersc10/workload/N10_ Workload_Analysis.latest.pdf. 访问日期：09/27/2021。[37] Bytecode Alliance。 [n.d.]. Wasmtime——用于 WebAssembly 的独立运行时。https://github.com/bytecodealliance/wasmtime [38] Mohak Chadha、Anshul Jindal 和 Michael Gerndt。2021。计算密集型 FaaS 函数的架构特定性能优化。载于《2021 IEEE 第 14 届国际云计算大会（CLOUD）》。478 483 10 1109 53861 2021 00062 – 。[39] [ ] 08 12 2022 https://doi.org/ 。/CLOUD 。 。CMake。n.d. 。https://cmake.org/. 访问日期：/ / 。[40] 将 Rust 编译为 WebAssembly。 [n.d.]. https://developer.mozilla.org/ en-US/docs/WebAssembly/Rust_to_ wasm. 访问日期：09/27/2021。[41] Leonardo Dagum 和 Rameshm Enon。1998。OpenMP：共享内存编程的行业标准 API。《计算科学与工程》，IEEE 5, 1（1998），46–55。[42] Frank Denis。 [n.d.]. WebAssembly 中的内存管理：面向 C 和 Rust 程序员的指南。https://www.fastly.com/blog/webassembly-memory-management-guide-for-c-rust-programmers [43] Docker。 [n.d.]. https://www.docker.com/. 访问日期：07/07/2022。[44] Docker Buildx。 [n.d.]. https://docs.docker.com/buildx/working-with- buildx/. 访问日期：07/07/2022。[45] Emscripten。 [n.d.]. https://emscripten.org/. 访问日期：09/27/2021。[46] Wasmer 中的 FileSystemCache。 [n.d.]. https://docs.rs/wasmer-cache/3.0. 2/wasmer_ cache/struct.FileSystemCache.html. 访问日期：12/11/2022。[47] Phani Kishore Gadepalli、Sean McBride、Gregor Peach、Ludmila Cherkasova 和 Gabriel Parmer。2020。Sledge：面向无服务器架构、轻量级的边缘端 Wasm 运行时。载于《第 21 届国际中间件大会论文集》（荷兰代尔夫特）（Middleware '20）。计算机协会，美国纽约州纽约，265 279 10 1145 3423211 3425680 – 。https://doi.org/ 。/ 。[48] Holger Gantikow、Steffen Walter 和 Christoph Reich。2020。使用 Podman 为高性能计算提供无根容器。载于《国际高性能计算大会》。Springer，343–354。https://doi.org/10.1007/ 978-3-030-59851-8 23_[49] Lisa Gerhardt、Wahid Bhimji、Shane Canon、Markus Fasel、Doug Jacobsen、Mustafa Mustafa、Jeff Porter 和 Vakho Tsulaia。2017。Shifter：面向高性能计算的容器。载于《物理学杂志：会议系列》，第 898 卷。IOP 出版社，082021。https://doi.org/10.1088/1742-6596/898/8/082021[50] GNU Make。 [n.d.]. https://www.gnu.org/software/make/. 访问日期：08/12/2022。[51] Robbert Gurdeep Singh 和 Christophe Scholliers。2019。WARDuino：用于微控制器编程的动态 WebAssembly 虚拟机。载于《第 16 届 ACM SIGPLAN 托管编程语言与运行时国际大会论文集》（希腊雅典）（MPLR 2019）。计算机协会，美国纽约州纽约，27 36 10 1145 3357390 3361029 – 。[52] https://doi.org/ 。/ 。Andreas Haas、Andreas Rossberg、Derek L. Schuff、Ben L. Titzer、Michael Holman、Dan Gohman、Luke Wagner、Alon Zakai 和 JF Bastien。2017。借助 WebAssembly 提升 Web 速度。《SIGPLAN 通知》52, 6（2017 年 6 月），185–200。https://doi.org/10.1145/ 3140587.3062363

[53] Adam Hall 和 Umakishore Ramachandran。2019年。《边缘端无服务器函数的执行模型》。载于《物联网设计与实现国际会议论文集》（加拿大魁北克蒙特利尔）（IoTDI '19）。计算机协会，纽约，纽约州，美国，225–236。https://doi.org/10.1145/ 3302505.3310084[54] Torsten Hoefler 和 Roberto Belli。2015年。《并行计算系统的科学基准测试：报告性能结果时告知大众的十二种方法》。载于《高性能计算、网络、存储与分析国际会议论文集》。1 12 10 1145 2807591 2807644 –。https://doi.org/。/。[55] 2018 英特尔公司。。《介绍英特尔® MPI 基准测试》。https://software.intel.com/content/www/us/en/develop/articles/intel-mpi-benchmarks.html [56] Abhinav Jangda、Bobby Powers、Emery D. Berger 和 Arjun Guha。2019年。《并没有那么快：WebAssembly 与本机代码的性能分析》。载于《2019年USENIX年度技术会议》（USENIX ATC 19）。USENIX 协会，伦顿，华盛顿州，107–120。https://www.usenix.org/conference/atc19/presentation/jangda [57] Michael Kiener、Mohak Chadha 和 Michael Gerndt。2021年。《揭开无服务器计算中函数内并行的神秘面纱》。载于《第七届无服务器计算国际研讨会论文集》（WoSC7）2021（线上活动，加拿大）（WoSC '21）。计算机协会，纽约，纽约州，美国，42–49。https://doi.org/ 10 1145 3493651 3493672 。/。Gregory M Kurtzer、[58] Vanessa Sochat 和 Michael W Bauer。2017年。《Singularity：用于计算迁移的科学容器》。PLOS ONE 12，5（2017），0177459 10 1371（），e。https://doi.org/。/journal.pone. 0177459 [59] 2004 C. Lattner 和 V. Adve。。《LLVM：用于终身程序分析与转换的编译框架》。载于《代码生成与优化国际研讨会，2004年。CGO 2004》。75–86。https: //doi.org/ 10 1109 2004 1281665 。/CGO.。劳伦斯[60] 利弗莫尔国家实验室。2021年。HPC IO 基准测试仓库。https://github.com/hpc/ior [61] Linux。 [不详。]。Fakeroot。https://wiki.debian.org/FakeRoot。访问日期：2021年9月27日。[62] Linux 内核对杂项二进制格式的支持。 [不详。]。https:// www.kernel.org/doc/html/latest/admin-guide/binfmt-misc.html。访问日期：2022年7月7日。[63] Vladimir Marjanovi´c、José Gracia 和 Colin W. Glass。2015年。《HPCG 基准测试的性能建模》。载于《高性能计算系统：性能建模、基准测试与模拟》，Stephen A. Jarvis、Steven A. Wright 和 Simon D. Hammond（编）。施普林格国际出版公司，尚，172–192。[64] Mellanox。 [不详。]。InfiniBand。https://wiki.archlinux.org/title/ InfiniBand[65] 消息传递接口论坛。2009年。《MPI：消息传递接口标准 2.2 版》。https://www.mpi-forum.org/docs/mpi- 2.2/mpi22-report.pdf [66] 消息传递接口论坛。2015年。《MPI：消息传递接口标准 3.1 版》。https://www.mpi-forum.org/docs/mpi- 3.1/mpi31-report.pdf [67] Mozilla。 [不详。]。asm.js。http://asmjs.org/ [68] Mozilla。2021年。《了解 WebAssembly 文本格式》。https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format [69] 开放容器倡议（OCI）。 [不详。]。https://opencontainers.org/。访问日期：2022年7月7日。[70] Podman。 [不详。]。无根模式下的文件系统注意事项。https: // tinyurl.com/2p8efntt。访问日期：2021年9月27日。[71] Reid Priedhorsky、R. Shane Canon、Timothy Randles 和 Andrew J. Younge。2021年。《构建 HPC 容器时的权限最小化》。载于《高性能计算、网络、存储与分析国际会议论文集》（圣路易斯，密苏里州）

（SC '21）。计算机协会，纽约，纽约州，美国，文章
32, 14 10 1145 3458817 3476187 页。
[72] 2017
https://doi.org/。/。Reid Priedhorsky 和 Tim Randles。。《Charliecloud：用于用户定义软件栈的 HPC 非特权容器》。载于《高性能计算、网络、存储与分析国际会议论文集》。
1 10 10 1145 3126908 3126925 –。
[73] 2013
https://doi.org/。/。S. Report、M. Heroux、J. Dongarra 和 P. Luszczek。。HPCG技术规范。[74] Oleksandr Rudyy、Marta Garcia-Gasulla、Filippo Mantovani、Alfonso Santiago、Raül Sirvent 和 Mariano Vázquez。2019年。《HPC 容器：生产环境生物模拟中的可扩展性与可移植性研究》。载于《2019年IEEE国际并行与分布式处理研讨会》（IPDPS）。IEEE，567–577。https://doi.org/10.1109/IPDPS.2019. 00066[75] Amit Ruhela、Stephen Lien Harrell、Richard Todd Evans、Gregory J Zynda、John Fonner、Matt Vaughn、Tommy Minyard 和 John Cazes。2021年。《在拍级规模上，刻画 CPU 和图形处理器架构中容器化高性能计算应用的性能》。载于《国际高性能计算大会》。Springer，411–430。https://doi.org/10.1007/978-3-030-78713-4 22_[76] Robert Schöne、Thomas Ilsche、Mario Bielert、Andreas Gocht 和 Daniel Hackenberg。2019年。《英特尔 Skylake-SP 处理器的能效特性及其对性能的影响》。载于《2019年国际高性能计算与模拟会议》（HPCS）。IEEE，399–406。https://doi.org/10.1109/HPCS48598.2019. 9188239[77] Thomas Schwinge。2019年。GCC邮件列表：回复"还可以实现更多兼容性吗？"。https://gcc.gnu.org/legacy-ml/gcc/2019-11/msg00095. html[78] Simon Shillaker 和 Peter Pietzuch。2020年。《Faasm：用于高效有状态无服务器计算的轻量级隔离》。载于《2020年USENIX年度技术会议》（USENIX ATC 20）。USENIX 协会，419–433。https://www.usenix.org/conference/atc20/presentation/shillaker [79] Volodymyr Shymanskyy 和 Steven Massey。 [不详。]。Wasm3。https: // github.com/wasm 3 3 /wasm Singularity。不详。。无根模式下的文件系统注意事项。
[80] [ ]
https://apptainer.org/admin-docs/master/installation.html。访问日期：2021年9月27日。[81] Solomon Hykes。 [不详。]。Wasm+WASI：Linux 容器的替代方案。https://twitter.com/solomonstre/status/1111004913222324225? lang= en。访问日期：2021年9月27日。[82] Michal Sudwoj。2020年。高性能计算环境中的 Rust 编程语言。学士论文。苏黎世联邦理工学院。https: // www.research-collection.ethz.ch/handle/20.500.11850/474922[83] 内核开发社区。 [不详。]。Seccomp BPF（带过滤器的安全计算）。https://www.kernel.org/doc/html/v4.16/ userspace-api/seccomp_ filter.html [84] Alfred Torrez、Timothy Randles 和 Reid Priedhorsky。2019年。《高性能计算容器运行时几乎没有或完全没有性能影响》。载于《2019年IEEE/ACM高性能计算隔离环境容器与新型编排范式国际研讨会》（CANOPIE-HPC）。
37 42 10 1109 49598 2019 00010 –。
[85]
https://doi.org/。/CANOPIE-HPC。。Robert Wahbe、Steven Lucco、Thomas E. Anderson 和 Susan L. Graham。1993年。《高效的基于软件的故障隔离》。SIGOPS 操作系统评论 27，5（1993年12月），203–216。https://doi.org/10.1145/173668. 168635[86] Wasmer公司。 [不详。]。Wasmer单遍编译器。https://docs.rs/wasmer-compiler-singlepass/latest/wasmer_compiler_singlepass/ [87] Wasmer公司。 [不详。]。通用WebAssembly运行时。https://wasmer. io/[88] Wasmer公司。 [不详。]。WAPM 是 WebAssembly包管理器。https://wapm.io [89] WebAssembly社区组。2021年。WebAssembly 系统接口。https://github.com/WebAssembly/WASI
[90] Miguel G Xavier、Marcelo V Neves、Fabio D Rossi、Tiago C Ferreto、Timoteo Lange 和 Cesar AF De Rose。2013年。面向高性能计算环境的基于容器的虚拟化性能评估。载于 *2013*年第二十一届*Euromicro*并行、分布式和网络处理国际会议。IEEE，233–240。https://doi.org/
10 1109 2013 41 。/PDP.。Yutian Yan、Tengfei Tu、
[91]
Lijian Zhao、Yuchen Zhou 和 Weihang Wang。2021年。了解 WebAssembly 应用程序的性能。载于 第*21*届*ACM*互联网测量会议论文集 (线上活动) *(IMC '21)*。计算机协会，纽约，美国纽约州，
533 549 10 1145 3487552 3487827 – 。
[92] 2003
https://doi.org/ 。/ 。Andy B Yoo、Morris A Jette 和 Mark Grondona。 。Slurm：简单的 Linux 资源管理实用程序。载于 并行处理作业调度策略研讨会。Springer，44–60。

A 制品附录

A.1 描述

*MPIWasm*是一个用于基于 MPI 的高性能计算应用的嵌入器，基于 `Wasmer` [87]。它可高性能地执行编译为 WebAssembly（Wasm）的这些应用，并有两个用途：

1. 实现接近原生应用性能的表现，即应用程序不使用 Wasm、直接在主机上执行时的性能。2. 支持将基于 MPI 的高性能计算应用作为 Wasm 二进制文件分发。

我们的制品包含*MPIWasm*的源代码、用于将基于 MPI 的 C语言++ 应用编译为 Wasm 的工具链、本文所用不同标准化高性能计算基准测试的 Wasm 二进制文件、适用于不同操作系统的预构建嵌入器版本，以及用于解析实验数据和生成图表的脚本。该制品可通过以下地址获取：

https://doi.org/10.5281/zenodo.7468121
或
https://github.com/kky-fury/MPIWasm

A.2 入门指南

为测试我们的 Wasm 嵌入器执行编译为 WebAssembly 的 MPI 应用程序，我们提供了一个预构建的 Docker 镜像，适用于 linux/amd64 平台，并包含所有必需的依赖项。

```
1  sudo docker run  -it kkyfury/ppoppae:v2 /bin/bash
2  #执行编译为 Wasm 的 HPCG 基准测试
3  mpirun  --allow -run -as -root  -np 4./target/release/嵌入器\
4      examples/xhpcg.wasm
5  #运行编译为 Wasm 的 IntelMPI 基准测试
6  mpirun  --allow -run -as -root  -np 4./target/release/嵌入器\
7      examples/imb.wasm
```

清单 5。MPIWasm 入门指南。

为此，用户可以按照清单 5 中描述的步骤进行操作。之后，*MPIWasm* 应能成功运行`HPCG` 和 `IntelMPI` 基准测试。我们在提供的制品中给出了这两项基准测试的示例输出。用户可以增加或减少运行基准测试的进程数（`-np`）。不过，根据

容器运行所在的系统，用户可能需要向 mpirun 提供 -oversubscribe 标志。

A.3 使用*MPIWasm*运行实验

本节介绍如何使用我们的嵌入器运行实验，以获得与本文中的图表相似的图表。

A.3.1 运行小规模实验。 我们提供了一个端到端脚本，可在 Docker 容器中运行小规模实验。该脚本：

1. 分别运行 HPCG、IS 和 IntelMPI 基准测试，包括原生执行以及使用*MPIWasm*执行的情况。 2. 解析所得结果并生成相关图表。

```
sudo docker run -it kkyfury/ppoppae:v2 /bin/bash
 cd run_experiments
    ./runme.sh
```

清单 6. 使用 *MPIWasm* 运行小规模实验。

运行实验时，用户可以按照清单 6 中所述的步骤操作。该脚本大约需要 10-15 分钟才能执行完毕。完成后，可以在 *run_experiments/experiment_data* 文件夹中查看生成的数据。生成的图表位于 *run_experiments/Plots* 文件夹中。我们的制品中提供了不同基准测试的示例图表。不过，在容器内执行基准测试时，应用程序原生执行与 *MPIWasm* 之间的性能差异约为 8-12%。

A.3.2 在高性能计算系统上运行大规模实验。 要使用我们的嵌入器运行大规模实验，用户需要执行以下操作：

1. 根据具体的架构、操作系统以及系统上的 MPI 库，为你的高性能计算系统构建一个嵌入器版本。MPIWasm 目前支持 OpenMPI 库，对 MPICH 和 MVAPICH 的支持有限。我们在制品中提供了针对不同操作系统和架构构建 *MPIWasm* 的示例。

2. 在高性能计算系统上使用构建好的嵌入器执行 MPI 应用程序。可以通过向高性能计算系统上的 RJMS 软件（例如 SLURM [92]）提交作业来完成。我们在制品中提供了适用于我们所用高性能计算系统（即使用 SLURM 的 SuperMUC-NG）的示例作业脚本。

3. 执行应用程序后，用户可以使用我们在制品中提供的不同解析器来解析基准测试数据。随后，可以使用制品中提供的绘图辅助工具将结果可视化。

A.4 将 C/C++应用程序编译为 Wasm

我们设置了一个 Docker 容器，其中包含将符合 MPI-2.2 标准的不同 MPI 应用程序编译为 Wasm 所需的依赖项。该制品还包含 HPCG、IntelMPI 和 IS 基准测试作为示例。

```
sudo docker run -it kkyfury/wasitoolchain:v1 /bin/bash
#编译 HPCG
cd /work/example/hpcg-基准测试
./wasi-cmake.sh
cd cmake-build-wasi
make
```

列表 7。将应用程序编译为 Wasm。

列表 7 描述了用户将 HPCG 基准测试编译为 Wasm 时可以遵循的步骤。有关将其他基准测试编译为 Wasm 的步骤，请参阅制品中提供的基础 Readme.md文件。本文使用的所有编译为 Wasm 的不同应用程序也都包含在制品中。

A.5 使用*MPIWasm*

有关详细的使用说明，请参阅制品中提供的基础 Readme.md 文件。

A.6 修改*MPIWasm*

要修改我们的嵌入器，建议使用制品中提供的 docker-compose 文件。该 docker-compose 文件会将包含嵌入器源代码的卷挂载到容器内。因此，对其源代码所做的任何更改都会反映在容器中。我们目前支持以下操作系统：

1. CentOS-8.2 2.
Opensuse-15-1 3.
Ubuntu-20-04 4.
MacOS-monterey

```
cd wasi-mpi-rs2 docker compose run centos-8-23 cargo build --release4 #构建过程完成后，可以在 5 #/target/release/ 文件夹中看到构建好的嵌入器。
```

清单 8.构建 *MPIWasm*。

清单 8 说明了修改后为 *MPIWasm* 构建 CentOS-8.2 的步骤。其他操作系统下构建嵌入器的说明见制品内基础版本的 Readme.md 文件。构建过程完成后，可以使用docker-cp 命令将嵌入器复制到用户的本地文件系统，如清单 9所示。

```
docker cp <容器-标识符>:/s/target/release/嵌入器\
        <目标-路径-用户-文件系统>
```

清单 9。复制*MPIWasm*。

我们在制品中提供了适用于不同受支持操作系统的基础镜像 Docker 文件。这些示例 Docker 文件可以轻松扩展，以支持其他 Linux 发行版。

A.7 支持 aarch64

我们的嵌入器也支持在 linux/arm64 平台上执行。我们在制品中提供了适用于 arm64 的嵌入器预构建版本，涵盖不同的受支持操作系统。

A.7.1 构建 aarch64 镜像。如果用户在 x86_64 系统上构建 Docker 镜像，则需要使用 docker-buildx。请注意，在这种情况下，构建镜像可能需要大约 12 小时。

```
1 sudo docker buildx c r e a t e  --name mybuilder  --use  -- boot s t ra p 2 cd wasi -消息传递接口 - rs / .
g i t l a b / ci / 镜像 / 3 sudo docker buildx build  --push  -f ubuntu -20 -04. Dockerfile \
```

```
4               --平台 linux /arm64 \
5               -t kkyfury / ubuntumodifiedbase : v1 。
6   cd  . . / . . / . . /
7   sudo docker buildx build  --push  -f Dockerfile \
8               --platform linux /arm64 \
9               -t kkyfury / 嵌入器arm : v1 。
```

清单 10。为 x86_64 上的 aarch64 构建 *MPIWasm* 。

清单 10介绍了使用 *MPI-Wasm* 和 docker-buildx 为 ARM 系统构建所需的步骤docker-buildx。用户应根据自己的 Docker 仓库账户更改 Docker 镜像标签，即将 kkyfury 替换为自己的仓库用户名。另一方面，如果用户使用的是 aardch64系统，请遵循 §A.6中介绍的说明。