# 支持 Unikernel 的流式计算引擎:Hummer

李 冰[1),2)] 张志斌[1),2)] 钟巧灵[1),2)] 程学旗[1)]

[1)](中国科学院计算技术研究所中国科学院网络数据科学与技术重点实验室 北京 100190)

[2)](中国科学院大学计算机与控制学院 北京 100049)

**摘 要** 社会计算中,社会公共安全、企业商务智能和舆情计算等众多领域均对实时计算的性能提出了越来越高的要求.流式计算引擎作为大数据计算研究领域的研究热点之一,致力于提供高吞吐量和低延迟的实时计算能力.流式处理任务对处理延迟非常敏感,数据价值随着处理时长的增长而快速递减.传统流式计算引擎设计中,操作系统、JVM 等占用大量计算资源,如何提升计算资源利用率成为目前亟待解决的问题.为此,本文提出了一种基于 C++ 语言实现的支持 Unikernel 的高性能实时数据分析计算引擎 Hummer.首先,通过引入 Unikernel 机制,Hummer 可绕过传统操作系统,直接运行于裸机或虚拟化层,减少传统操作系统无关组件带来的性能开销,支持分布式环境下的快速部署与启动,为高性能大数据计算引擎设计提出新的思路.其次,通过使用 Unikernel 对计算引擎进行封装,解决了 C++ 应用需本地化编译难以在集群中部署的问题.最后,系统使用灵活的网络通信方案,支持异构网络部署及网络资源隔离.实验表明,Hummer 端到端处理延迟低于 30 ms,较 Flink 系统低 2 倍,较 Spark Streaming 低 15.8 倍,且吞吐量达到 Flink 的 2 倍.使用 Unikernel 封装的 Hummer 系统镜像仅为 100 MB,启动时间约为 2 s.

**关键词** 大数据;数据流;分布式计算;流处理系统;微内核操作系统

**中图法分类号** TP311 **DOI 号** 10.11897/SP.J.1016.2019.01755

## Hummer: A Stream Computing Engine with Unikernel Support

LI Bing[1),2)] ZHANG Zhi-Bin[1),2)] ZHONG Qiao-Ling[1),2)] CHENG Xue-Qi[1)]

[1)](*CAS Key Laboratory of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)

[2)](*School of Computer and Control Engineering, University of Chinese Academy of Sciences, Beijing* 100049)

**Abstract** In social computing, it is well-known that the real-time computing plays an important role in social public security, business intelligence and public opinion monitoring. Therefore, in order to provide high throughput and low latency capabilities, the stream computing engine has sprung up recently as a research hotspot in big data computing area. Generally, most stream processing tasks is very sensitive to latency, and the data value decreases rapidly as the processing time increases. In the traditional streaming computing engine design, the operating system, JVM, etc. occupy a large amount of computing resources and suffer from JVM overheads such as pointer chasing and transparent memory management. Lacking their inability to exploit modern CPUs efficiently and not being able to utilize the entire network bandwidth of modern high-speed networks. How to improve the utilization of computing resources has become an urgent problem to be solved. Therefore, we propose a high-performance real-time stream computing engine, referred to as Hummer, by utilizing C++ programing language and Unikernel. It is known that the traditional operating systems, like CentOS, are designed as a general-purpose system and

收稿日期:2018-04-27;在线出版日期:2018-12-28.本课题得到中国科学院战略先导科技专项(A 类)(XDA19020400)资助.**李 冰**,博士研究生,主要研究方向为流计算系统、存储系统、分布式系统.E-mail: libingqgzy@163.com.**张志斌**,博士,副研究员,主要研究方向为网络流处理、网络测量.**钟巧灵**,博士研究生,主要研究方向为深度学习、深度学习系统、分布式系统.**程学旗**,博士,教授,博士生导师,主要研究领域为信息检索、社会计算、分布式计算.

contain a large number of services to support various applications and hardware configurations. However, many of them are useless, such as sound card or printer driver, and generally results in a huge system size and unnecessary computation overhead. Besides, the hypervisor has to simulate clock interrupts so that traditional operating systems can work properly, which causes that most computing resources are consumed by the operating system when there is no workload. To reduce the unnecessary computing overhead caused by useless services, we consider utilizing the Unikernel to make Hummer bypasses the operating system and run directly on hypervisor or bare-metal environment. Particularly, Hummer also supports quick deployment and startup in a cluster. To the author's best knowledge, we are the first to apply Unikernel to the design of big data stream computing engines. Secondly, since localized compilation and third-party library dependencies make C++ applications difficult to deployed in a cluster, we can utilize Unikernel to solve these problems by packaging the application as an image and eliminating the divergence of machine using hypervisor. Thirdly, we designed a flexible task-oriented network communication solution to decouple the network communication component from TaskManager as normal task. This brings many benefits, such as heterogeneous network support and network source isolation. In most situations, batch processing pays more attention to the throughput, and it almost occupies all the bandwidth of network IO, thus significantly affects the latency sensitive stream processing. Most existing solutions are not optimized for this situation. Nevertheless, we can solve this problem by isolating batch and stream networks using our flexible task-oriented network configuration. Our experiments show that the end-to-end record processing latencies of Hummer is less than 30 ms, which is also 1.7x and 15.8x lower than that of Flink and Spark Streaming, respectively. Moreover, the achievable throughput of Hummer is around 2x faster than that of Flink. The Hummer image using Unikernel is only around 100 MB, and the boot time is about 2 s.

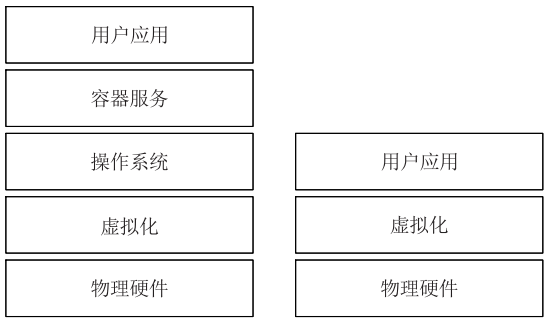
**Keywords** big data; data stream; distributed computation; stream processing system; Unikernel system

## 1 引 言

在现代科学技术快速发展的背景下,社会网络信息呈现爆炸性、指数式增长,传统社会计算正由关联规则挖掘等结构化分析转换为更高层次的宏观知识挖掘,数据量快速增长和多源数据联合分析对大数据计算引擎提出了新的要求.社会计算中,社会公共安全、企业商务智能和舆情计算等领域均体现出实时计算的重要性.如何解决社会计算场景下的海量数据实时计算问题已成为目前急迫之需.

传统数据中心中,为了保证环境隔离与管理方便,应用一般独立部署,存在集群资源利用率低、部署和扩展复杂、资源无法动态调整、无法快速响应业务等问题.因此,越来越多的企业通过云计算平台,将传统数据中心运行模式由独立部署升级为混合部署,使用虚拟化技术将计算资源整合,解决传统数据中心中的上述问题.此外,虚拟化技术还可用于众核处理器,通过将处理器模拟为多个单核处理器,提高众核处理器利用率.

目前主流集群应用部署模式如图 1(a)所示,在物理硬件层使用虚拟化技术以消除硬件差异,复用计算资源并支持动态扩容.虚拟化层之上安装操作系统,并于操作系统之上安装容器服务,以提供良好




(a) 主流集群部署模式 (b) 绕过操作系统部署模式

图 1 集群应用部署模式

的运维环境.此运行模式中,所有用户应用以容器形式运行于容器服务中.操作系统的功能只是为容器服务提供资源调度等必要支撑,但由于虚拟化技术已包含资源管理等功能,操作系统层变得冗余.因此,为减少传统操作系统无关组件带来的性能开销和消除操作系统与虚拟化层资源调度的冲突,图 1(b)将用户应用直接在虚拟化层运行,形成绕过操作系统的部署模式.该模式适用于对延迟敏感的流式处理系统.

目前,Spark[1-2]、Storm[3]、Apache Flink[4]等主流计算引擎均不支持绕过操作系统部署模式,依托传统操作系统造成了不必要的性能开销[5],影响计算引擎性能.此外,由于 C++等本地化编译语言可移植性较差,难以解决用户作业在集群中批量部署的问题,大多数计算引擎选择使用 JVM 技术实现,其内存访问及垃圾回收机制带来了无关的性能开销.

为此,本文使用 C++语言实现了一个支持 Unikernel[6]的流式大数据计算引擎 Hummer.系统通过使用 Dataflow 模型,提供面向流式处理场景的低延迟消息处理服务;同时,系统将批式处理当作特殊的流处理,以实现高吞吐量的批式消息处理.本文所做的贡献主要有:

(1)提出了第一个支持 Unikernel 的流式大数据计算引擎 Hummer.系统同时支持流式处理和批式处理,支持分布式环境下的快速部署与启动.Hummer 可绕过传统操作系统,直接部署于虚拟化层或裸机,减少了传统操作系统无关组件带来的性能开销,解决了传统操作系统调度和 Hypervisor 调度冲突问题,同时也解决了 C++应用需本地化编译,难以在集群中批量部署的问题.

(2)提出全新的网络通信方案,将网络负载当作普通任务负载对待,简化 TaskManager 设计,以适配 Unikernel 单一地址空间及轻量化设计.同时,系统支持精确到每个任务的网络配置,支持异构网络部署及网络资源物理隔离.

(3)本文对比展示了流计算引擎在 Unikernel、Docker 以及 CentOS 系统下的性能,分析了容器技术及虚拟化技术在流计算引擎应用的优劣,为高性能大数据计算引擎提出新的设计思路.

本文第 2 节介绍主流的流计算引擎模型以及 Unikernel 基本概念;在第 3 节介绍本系统的设计与实现;在第 4 节介绍系统的实验设计及实验结果,并给出分析;最后,在第 5 节总结本文所述工作,并对未来研究做出展望.

## 2　背景及相关工作

### 2.1　大数据计算引擎

MapReduce[7]作为第一代计算引擎打开了大数据分析的篇章,通过将算法拆分为 Map 和 Reduce 阶段,实现分布式并行计算以及容错等功能.但同时,MapReduce 也有表达能力差、中间结果存储在 HDFS 中、磁盘读写开销大以及迭代任务支持性较差等缺点.Tez[8]等为代表的第二代计算引擎使用灵活的 DAG(Directed Acyclic Graph,有向无环图)取代 MapReduce 模型.Spark 与 Flink 作为第三代计算引擎代表,在使用 DAG 基础上增加了实时计算功能[2].第三代计算引擎使用的计算模型分为 BSP(Bulk Synchronous Parallel)模型和 Dataflow 模型两类:

(1)BSP 模型.MapReduce[7]、Dryad[9-10]、Spark 和 FlumeJava[11]等系统使用 BSP[12]模型.如图 2 所示,模型将计算划分为多个阶段,阶段内部由并行的本地运算组成,阶段之间通过 barrier 同步.计算阶段的引入简化了系统容错及资源调度,系统调度器在阶段之间进行快照保存或重新调度等操作即可.由于计算单元随着阶段的转换而改变,因此可减少数据在集群节点间的移动,所以使用 BSP 模型的系统吞吐量往往较高,且具有较好的容错和扩展性能.但使用 BSP 模型处理流式数据时,需要将流式数据按照时间片 $T$ 划分为 micro-bath 计算,其数据处理延迟的下界为 $T$,且由于每个 micro-batch 任务之间需要与 master 通信并等待新一轮调度,$T$ 值不能设置太小,否则会显著增加系统调度开销[13].因此,使用 BSP 模型的系统往往导致数据处理延迟较高.

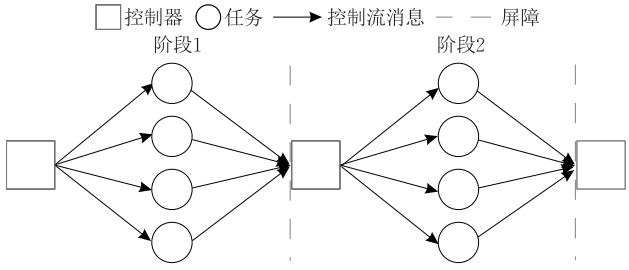



图 2　BSP 模型图

(2)Dataflow 模型.Dataflow 模型[14]最早应用于数据库系统[15-17]中.近年来,逐渐被 Naiad[18]、Flink[19]、StreamScope[20]等一些追求低延迟的流式计算引擎所使用.如图 3 所示,Dataflow 模型中,计算不再按照阶段划分,而是转换为由 operator 节点

组成的 DAG.operator 是系统中的基本计算单元，在系统启动时被创建，随后生存于系统整个生命周期. 模型中数据以 record 形式进入系统，并在 operator 间流动. 与 BSP 模型不同的是，record 间不需要与 master 通信或等待新的调度，同时也没有任何 barrier，因此 Dataflow 模型的处理延迟非常低. 但是由于模型没有计算阶段划分，系统难以选择合适的时机进行快照保存或重新调度等操作，因此使用 Dataflow 模型的系统通常需要利用分布式全局一致性快照算法[21]来进行容错.

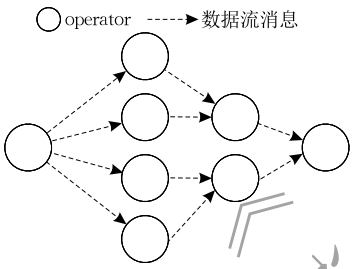



图 3 Dataflow 模型图

### 2.2 微内核操作系统

传统操作系统的设计理念是为了支持众多的应用软件和硬件配置，众多的硬件和功能支持使操作系统体积变得庞大，造成了不小的计算资源开销. 另外，为了使传统操作系统可以正常工作，Hypervisor 必须模拟时钟中断，这使得操作系统在没有负载时也会消耗计算资源，造成资源浪费[22].

微内核操作系统(Unikernel)是专用的、单地址空间的、使用 Library OS 构建出来的镜像. 系统将硬件驱动当作支持库，只编译用户应用需要的驱动，进程间资源调度和硬件适配全部由底层 Hypervisor 负责，实现轻量级操作系统内核. 其具有应用体积小、启动速度快、高安全性、高性能等特性，且由于 Unikernel 计算资源完全交由 Hypervisor 管理，解决了传统操作系统的资源管理和 Hypervisor 资源管理冲突造成性能下降的问题. 较传统操作系统，Unikernel 在虚拟化环境下拥有更好的性能.

## 3 Hummer 系统介绍

本节详细介绍 Hummer 的内部实现，从系统架构开始，介绍了一个 WordCount 程序如何从用户代码转换为作业并在集群中执行. 本系统使用 C++ 语言实现，代码约为 3 万行.

### 3.1 系统架构设计

由于主从架构相比无主架构，具有高吞吐、低延迟等特性，且本系统中主节点不会成为性能瓶颈. 因此本系统采用主从架构设计，其架构图如图 4 所示，包含 Client、JobManager 和 TaskManager 三个部分.

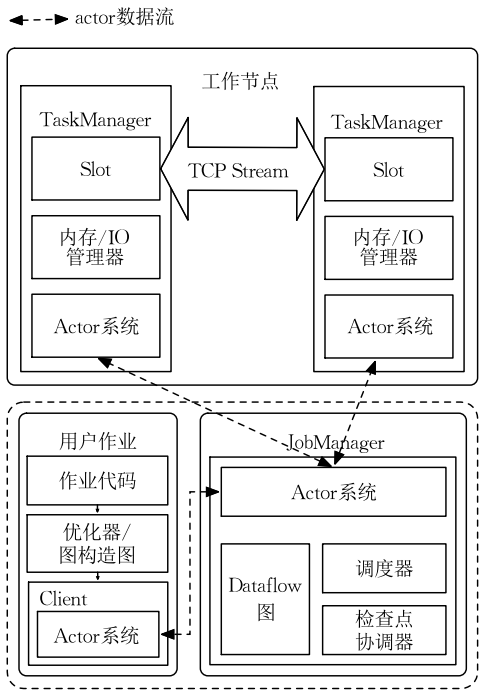



图 4 系统架构图

用户客户端 Client 是用户与系统交互的桥梁，负责将用户代码转换为 DAG 并对其进行优化，随后客户端将 DAG 和 UDF(User-Defined Function)序列化并提交至 JobManager 等待执行，同时，客户端还支持查看任务进度和取消任务等操作.

用户使用系统时，首先需启动 JobManager 实例，JobManager 从用户客户端 Client 接受作业，并负责作业的管理与调度. TaskManager 负责计算节点上计算资源及作业任务管理，可从 JobManager 接受任务，并放于空闲 Slot 执行，Slot 是系统计算资源管理的基本单位.

Hummer 采用两种不同方案实现控制流通信和数据流通信. 控制流通信用于 JobManager 和 TaskManager 之间传递控制信息. 可支持 JobManager 单节点上万并发连接. 数据流通信用于在 TaskManager 之间传递用户作业中的数据流信息，通过 TCP Stream 实现，并利用零拷贝、CPU CacheLine 优化等技术提升性能.

由于 Unikernel 应用难以调试. 除支持 Unikernel 模式外，Hummer 还支持传统的单机及分布式环境，便于用户对作业代码及框架进行调试.

### 3.2 编程模型介绍

与 Flink 等主流流计算系统相似，Hummer 使用 DAG 表示用户作业，DAG 中的节点表示用户作

业中的数据处理任务，边表示数据流动，使用 DAG 有两大优势：(1) DAG 允许任务有多个输入输出，简化了诸如 Join 等经典数据操作的实现；(2) DAG 的边明确表达了数据的流通路径，有利于系统对任务调度进行优化.

考虑到系统对数据处理实时性的要求，Hummer 使用 Dataflow 模型. 系统将流式数据抽象为 DataStream，支持从内存、文件、socket、kafka 等输入源或其他 DataStream 的输出创建. 并通过 map、flatMap、groupBy、reduce 等 Transform 操作进行转换，生成新的 DataStream. 系统自动将 Transform 操作分散部署在多个节点并行计算，部署和调度过程对用户完全透明.

本文以 WordCount 为例，介绍如何从用户代码转换为作业并在集群中执行. 过程 1 为 WordCount 用户作业代码. 代码 2～4 行通过 JobManager 获取 StreamExecutionEnvironment 对象，对象中记录了用户作业和集群配置等信息，默认情况系统会自动检测计算节点硬件环境进行配置. 随后，代码 6 行从文件 source.txt 创建 DataStream，7～10 行依次在 DataStream 上做 flatMap、groupBy、count 和 print 操作，来对 DataStream 进行转换. 转换操作均采用抽象接口设计，用户可通过继承相应接口实现 UDF 操作. 最后，代码 11～12 行将作业进行命名，并提交给集群执行. 流式处理中，execute 方法返回作业提交结果，批式处理则阻塞客户端程序直到作业执行完毕，并返回作业执行结果.

**过程 1.**　WordCount 作业.

```
1.  //get execution environment from JobManager
2.  shared_ptr<StreamExecutionEnvironment>env
3.      =StreamExecutionEnvironment
4.          ::getExecutionEnvironment(JobManager);
5.  //create dataStream from source
6.  auto dataStream=env->fromFile("source.txt");
7.  dataStream
8.      ->FlatMap(new SplitStringFunction())
9.      ->groupByCount()
10.     ->print<std::tuple<string,int>>();
11. auto &result=
12.     env->execute("Distributed world count.");
```

### 3.3　分布式运行说明

本小节以 WordCount 作业为例，详细介绍了作业从代码至分布式运行的整个流程. 用户作业输入系统后进行一些列转换，由作业源代码转换为最终的 ExecutionGraph，并交由 TaskManager 执行，作业转换过程如图 5 所示.

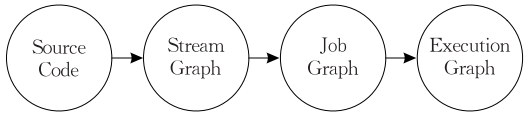



图 5　用户作业转换图

#### 3.3.1　DAG 逻辑视图生成

用户代码提交至客户端后首先被转换为 StreamGraph. StreamGraph 是未经优化的作业 DAG，其逻辑结构与用户代码相对应. StreamGraph 由 StreamNode 和 StreamEdge 组成. StreamNode 是计算节点，其中包含 SourceOperator、OneInputOperator 和 TwoInputOperator 三种 operator. SourceOperator 负责表示 DAG 中没有输入，只有输出的节点，一般表示用户程序中的数据源输入；OneInputOperator 表示 DAG 中只有一个输入的节点，可用于表示用户代码中一个输入对应一个输出的 Map，FlatMap 等操作，也可表示只有一个输入但没有输出的 SinkOperator 等特殊节点，SinkOperator 是 DAG 中的尾节点，用于表示用户代码中的运算结果输出逻辑；TwoInputOperator 表示 DAG 中有两个输入的节点，多个输入可由 OneInputOperator 和 TwoInputOperator 组合表示. StreamEdge 表示 DAG 中的边，记录了数据在系统中的流动规则.

图 6 进一步展示了 3.2 节中作业 WordCount 生成的 StreamGraph，图 6 中每个 StreamNode 节点代表了用户代码中对数据流的一次操作，StreamEdge

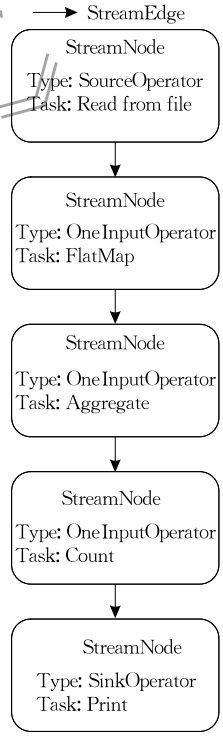



图 6　WordCount StreamGraph

边表示数据流动方向.

3.3.2 DAG 逻辑视图优化

客户端生成 StreamGraph 后通过对其优化,生成 JobGraph,并将其提交至 JobManager. JobGraph 是 StreamGraph 经优化后的任务逻辑视图,由 JobVertex 和 JobEdge 组成. JobVertex 包含 StreamGraph 中的一个或多个 StreamNode 计算任务,JobEdge 表示 JobVertex 间的数据流动.

系统支持使用 OP(Operator)融合机制优化 StreamGraph. OP 融合机制使用如图 7 的规则将关联度较高的 operator 合并. 融合有两大优势:(1)确保关联度较高的 operator 不会跨节点分布,避免序列化和网络传输开销;(2)避免同节点下 operator 通信带来的内存拷贝开销和多线程资源竞争.

图 8 为 3.3.1 节中 WordCount 的 StreamGraph

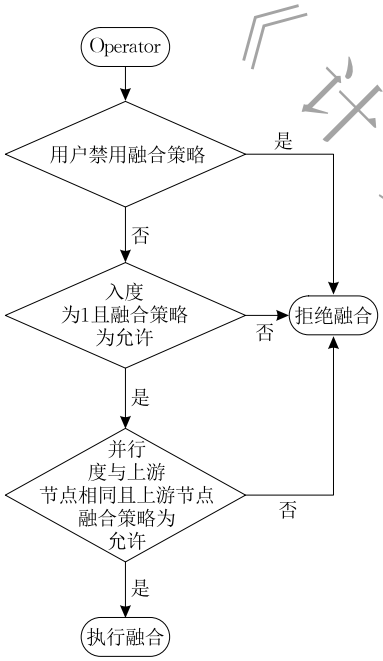



图 7 OP 融合规则

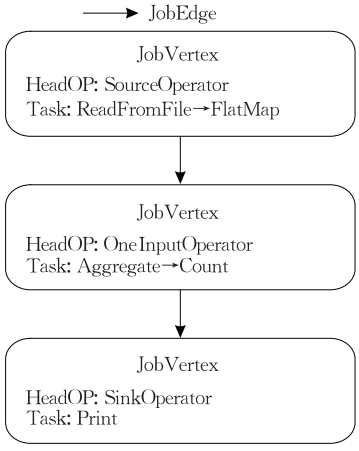



图 8 WordCount JobGraph

生成的 JobGraph. 图中的 JobVertex 节点代表优化后的数据处理任务,已将部分 StreamNode 合并. JobEdge 表示数据流动方向.

3.3.3 DAG 物理视图生成

JobManager 收到客户端提交的 JobGraph 后,JobManager 将其转换为 ExecutionGraph. 与抽象的 JobGraph 不同,ExecutionGraph 包含了作业执行与调度所需的全部信息,由 StreamTask 和 StreamEdge 组成. StreamTask 由 JobGraph 中的 JobVertex 结合调度方案生成,包含了作业任务信息与作业调度信息,与物理执行环境中的任务一一对应,是 TaskManager 中任务执行的基本单位. StreamEdge 包含 StreamTask 之间的连接信息. 图 9 为由 3.3.2 节中 WordCount 的 JobGraph 生成的 ExecutionGraph.

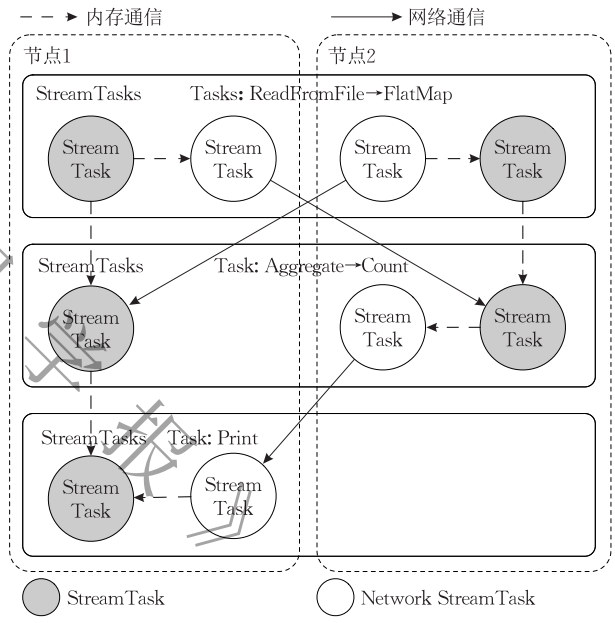



图 9 WordCount ExecutionGraph

JobManager 收到 JobGraph 后,需要对 JobGraph 中的每个节点确定运行并行度. 分配过程中参照用户配置的作业总并行度、节点并行度以及节点最大和最小并行度进行分配. 如未指定节点并行度,则根据剩余计算资源平均分配,并保证节点并行度在用户配置允许范围且总并行度不超过用户作业配置.

随后 JobManager 根据并行度设置生成调度方案. 由于 JobGraph 生成时已使用 OP 融合机制将关联度较高的 operator 合并,系统使用 Round-robin 策略生成任务调度策略,并尽量保证 JobGraph 中同一节点的不同副本分配至不同 TaskManager.

作业调度方案生成后,结合 JobGraph 生成 StreamTask,并对相关联节点生成 StreamEdge 连接

信息. StreamEdge 分为 MemoryEdge 与 NetworkEdge，分别用于连接同节点和跨界点的 StreamTask.

最后，系统将 NetworkEdge 转换为 MemoryEdge 和一个负责网络连接的 StreamTask. 系统可根据用户网络配置，对不同作业任务节点网络连接分配不同的计算资源，实现灵活的网络配置支持. 同时，通过将不同任务封装为统一的 StreamTask，可消除不同任务间的差异，简化 TaskManager 的设计. 随后 JobManager 根据 StreamTask 中的调度信息将 StreamTask 序列化并发送至对应 TaskManager.

#### 3.3.4　任务执行

TaskManager 收到任务后对其反序列化并初始化 StreamTask 对象，并对相关联的 StreamTask 使用高性能无锁队列建立连接；跨节点网络连接组件由于已被封装为 StreamTask，在 TaskManager 看来与其余任务无异，故无需关注. 随后，TaskManager 将封装好的 StreamTask 分配至空闲 Slot 执行，Slot 是系统资源分配的基本单位，一个 Slot 代表占用一个 CPU 内核. 最后，Slot 通过 StreamTask 的 invoke() 方法启动任务，并通过 cancel() 和 stop() 等接口对任务行管理，这种轻量级设计可以使 TaskManager 占用更少的资源，更好的适配 Unikernel 等单一地址空间场景.

由于 Unikernel 应用难以调试，为了便于用户对作业代码及框架进行调试，除支持 Unikernel 部署模式外，系统还支持单机以及传统的分布式模式运行. 单机环境下，JobManager 和 TaskManager 之间通过函数调用取代 actor 机制进行通信，Task 之间使用共享内存等方式进行通信，用户作业代码和计算引擎共用同一进程，进一步简化用户代码调试难度.

### 3.4　网络通信设计说明

系统网络通信分控制流通信和数据流通信两部分. 控制流通信用于在 JobManager 和 TaskManager 间传递控制信息，其特点为 JobManager 单节点高并发连接，但传输数据量少. 数据流连接用户任务间的数据流传输，并发度低，仅在需要传输数据的任务间建立连接，但传输数据量大，对延迟敏感. 基于以上原因，本系统将控制流网络通信和数据流网络通信独立设计：

(1) 控制流通信. 控制流通信使用基于协程机制的 Actor[23] 模型实现. 模型中最基本的运算单元是 actor，每个 actor 可完成一组独立的功能，actor 之间通过异步消息传递调用. 使用 Actor 机制可以将系统 JobManager 和 TaskManager 解偶，并使 JobManager 单节点支持上万并发连接.

(2) 数据流通信. 主流流计算引擎中，TaskManager 统一负责网络通信传输，这种方式可以简化系统设计实现流程，但同时也降低了系统网络配置的灵活性，单一的网络拓扑结构无法应对复杂的业务场景. Hummer 使用全新的网络设计，TaskManager 不再管理数据流网络通信，而将网络通信当作普通任务负载对待，用户可针对每个任务单独配置网络通信方案. 网络通信支持独占一个或多个 Slot，也可使用 OP 融合机制与任务共享 Slot. 与主流方案相比，本方案将网络负载与 TaskManager 解耦，简化 TaskManager 设计以适配 Unikernel 场景，此外，本方案还有以下优势：

① 灵活的网络配置. 可根据任务优先级和任务类型针对任务单独配置网络模块. 网络 IO 需求较低的任务使用 OP 融合的形式与任务负载共享 Slot，降低内存拷贝等开销；网络 IO 需求较高的任务可单独占用一个或多个 Slot 来满足其较高的网络性能需求.

② 异构网络支持. 针对不同的任务连接不同的硬件网络环境，实现异构网络接入支持.

③ 网络资源隔离. 在传统批式处理流程中，Shuffle 阶段对网络带宽使用率较高，通常情况下会占用几乎全部网络资源，严重影响到对延迟敏感的流式处理系统，目前工业界常常将流式处理和批式处理分别部署于两个物理集群，以减少批式处理对流式处理的影响. Hummer 通过在同一集群中对批式处理和流式处理分别使用不同的网络接口，做到物理网络隔离，减少批式处理中网络带宽占用对流式处理的影响，在提高集群资源利用率的同时降低维护成本.

### 3.5　容错及扩展说明

与 BSP 模型不同，Dataflow 模型由于没有计算阶段划分，难以在同一时刻对系统记录全局快照，容错较为困难，本系统使用分布式全局一致性快照算法[24] 实现容错，支持 Exactly-once 语义. 系统每隔固定时间间隔记录全局快照并备份输入源位置，出错时将所有计算节点恢复至最近快照并回退输入源至相应位置.

系统要求数据源可缓存数据并可从任意时刻回放. 如图 10 所示，系统每隔固定的时间向数据源中注入 Barrier，并使之随数据记录在 DAG 中流动，当 Barrier 流动到计算节点时，计算节点会对当前的状态记录快照，当 Barrier 流动到 Sink 节点时，全局快照记录完成，整个快照记录过程中，数据流无需暂

停,快照记录代价较小.但当系统出错时,需要将全部计算节点恢复至最近快照状态,并回退输入流至相应位置,数据恢复代价较大.

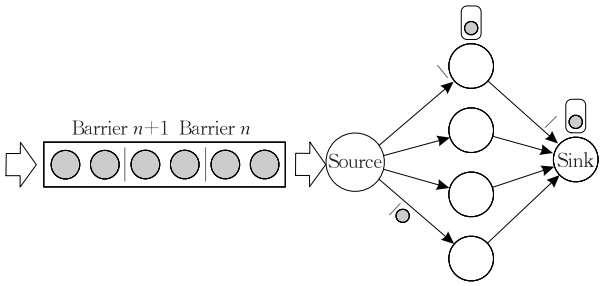



图 10 分布式全局一致性快照算法

## 4 实 验

### 4.1 实验环境

实验集群由四台服务器组成,其中一台机器作为 JobManager 和 TaskManager 共享使用,三台机器 TaskManager 独享使用,服务器 CPU 配置为 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40 GHz x2、内存 128 GB、10 Gbps 网卡.操作系统为 CentOS 7.3、内核为 3.10.0、GCC 编译器版本 6.2.1、QEMU 版本 2.0.0.

实验数据集使用小说《Game of Thrones》多次拼接生成,文本大小 16 GB,共 155 800 000 行,包含单词 2 900 520 000 个,实验从批式处理和流式处理两个方面比较 Hummer 与 Flink、Spark 的性能,并对比虚拟化环境下使用 Unikernel 封装的 Hummer 与 CentOS 系统封装的 Flink 性能差距,实验 Spark 版本为 spark-2.2.0-bin-hadoop2.7,Flink 版本为 1.4.2-hadoop28-scala_2.11.

### 4.2 批式处理

批处理实验任务流程如图 11 所示,实验使用小说文本文件做数据源,系统读入数据后依次进行字符串分割、GroupBy 和 Sum 操作,最后将计算结果写入磁盘保存.实验从 CPU、内存、吞吐量三方面比较单机环境和四节点集群环境下 Hummer 与 Spark、Spark Streaming 和 Flink 在 CentOS 系统下的性能.实验中所有系统均使用最大系统资源配置,且关闭容错功能.


图 11 批式处理实验任务流程图

系统在单机环境下的吞吐量如表 1 所示,Spark 由于使用 BSP 模型,其吞吐量对比 Dataflow 模型的 Flink 与 Hummer 具有先天优势.实验表明,Hummer 吞吐量约为同样使用 Dataflow 模型的 Flink 系统的两倍,与使用 BSP 模型的 Spark 和 Spark Streaming 相差无几.由图 12 系统 CPU 使用率对比图可以看出,Flink 和 Spark 几乎占满了 CPU 资源,而 Hummer 的 CPU 使用率约为 80%,如能提升 IO 性能,系统整体性能会有进一步提升.图 13 为系统内存使用率对比图,可以看出,Hummer 做为针对虚拟化环境优化的轻量级系统,内存利用率明显低于 Spark 和 Flink.在 4 节点集群环境下,Hummer 依然保持较好的性能,由于硬件资源 I/O 限制,四个系统均未达到线性扩展,但由表 2 可以看出,Hummer 在四节点分布式环境下依然具有良好的性能.

**表 1 单机环境性能对比表**

| 系统名称 | 任务耗时/s | 吞吐量/(MB/s) |
|---|---|---|
| Spark | 96 | 170.66 |
| Spark Streaming | 101 | 162.21 |
| Flink | 236 | 69.42 |
| Hummer | 115 | 142.47 |

**表 2 四节点集群环境性能对比表**

| 系统名称 | 任务耗时/s | 吞吐量/(MB/s) |
|---|---|---|
| Spark | 38 | 431.15 |
| Spark Streaming | 65 | 252.06 |
| Flink | 80 | 204.80 |
| Hummer | 41 | 399.61 |

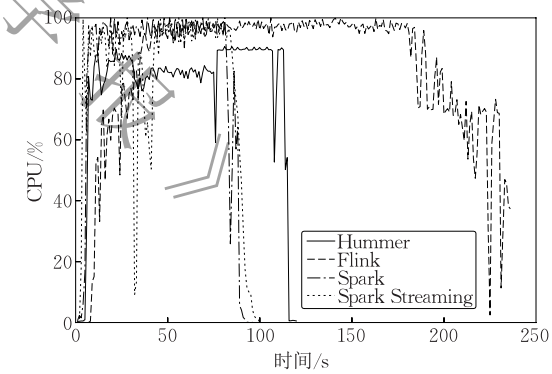



图 12 系统 CPU 利用率对比图

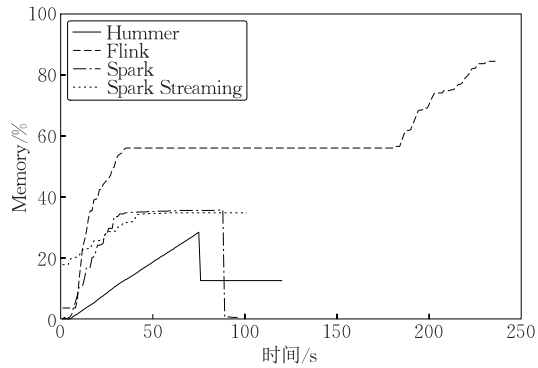



图 13 系统内存利用率对比图

### 4.3　流式处理

流式处理实验任务流程如图 14 所示，实验数据源使用 ZeroMQ 以 40 W 消息/s 的速率发送本文数据，并在发送时记录发送时间，系统收到消息后依次进行字符串分割和正则表达式过滤，最后计算处理延迟. 实验中所有系统均使用最大系统资源配置，Spark Streaming 的 micro-batch 大小设置为 1 s. 实验比较分布式环境下 Hummer 与 Spark Streaming 和 Flink 系统的处理延迟.

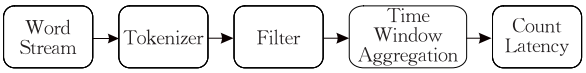



图 14　延迟测试任务流程图

分布式环境下延迟测量误差主要来源于时间不统一，为了在分布式环境下尽可能准确的测量延迟，系统使用 NTP(Network Time Protocol)机制同步服务器时间，但由于 KVM 虚拟机存在时间漂移问题，传统 NTP 机制无法做到细粒度时间同步，本文自行设计实现了一个轻量级 SNTP(Simple Network Time Protocol)客户端，通过 UDP 协议获取 NTP 服务器时间. NTP 请求过程如图 15 所示，SNTP 客户端在 $T1$ 时刻向 NTP 服务端发送请求，随后 NTP 服务端于 $T2$ 时刻收到请求并于 $T3$ 时刻作出响应，最后 NTP 客户端于 $T4$ 时刻收到响应. 由此可知，NTP 请求的时间误差 $\Delta$ 为

$$\Delta=(T4-T1)-(T3-T2).$$

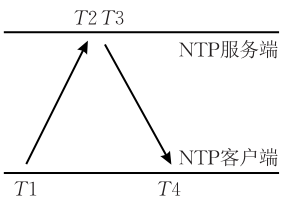



图 15　*NTP* 请求过程图

在本例中，可忽略 NTP 服务器响应时间($T3-T2$)，即 NTP 请求误差为

$$\Delta=(T4-T1).$$

本实验环境中，连续进行 1000 次实验，$\Delta$ 的平均值为 0.316 ms，远小于毫秒级的系统处理延迟，故本实验忽略 NTP 误差对延迟测量带来的影响.

本对比实验中，设置数据源发送速率 4W/s. 由于 Spark Streaming 使用基于时间窗口的 micro-batch 机制实现流式处理，消息需要积攒够时间窗口大小才能进行处理，故延迟较高. 各系统平均延迟如表 3 所示，本实验中 Hummer 比同样使用 Dataflow 模型的 Flink 延迟低 1.7 倍，比 Spark Streaming 低 15.8 倍. 系统延迟 CDF(累积分布函数)图如图 16 所示，可以看出，Hummer 在约 95%的情况下延迟低于 Flink，在 100%的情况下延迟远低于 Spark Streaming.

**表 3　分布式系统平均处理延迟对比表**

| 系统名称 | 平均延迟/ms |
|---|---|
| Spark Streaming | 333.46 |
| Flink | 35.86 |
| Hummer | 21.13 |

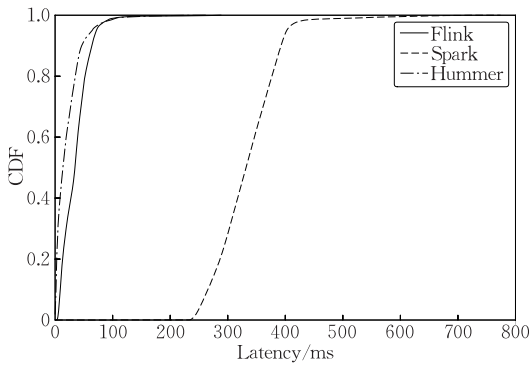



图 16　Flink、Hummer 和 Spark Streaming 处理延迟 CDF 图

### 4.4　Unikernel 模式

Unikernel 实验使用 CentOS 系统将 Spark 和 Flink 封装为虚拟机镜像，并与使用 Unikernel 封装的 Hummer 进行对比. 实验网络拓扑如图 17 所示，使用 QEMU-KVM 虚拟机，通过 Linux Tun 接口利用网络桥接技术将虚拟机和外部网络连接. KVM 经过性能调优，使用最大资源配置. 此外，还添加了目前流行的 Docker 容器环境进行性能对比测试.

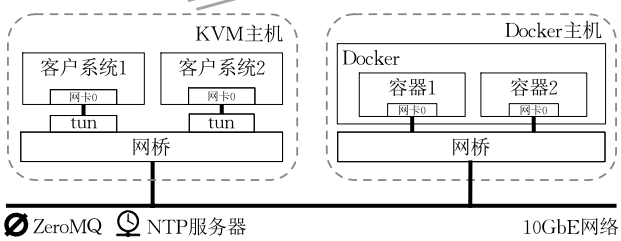



图 17　虚拟化环境实验网络拓扑图

实验的数据源及应用负载与流式处理实验相同，Spark Streaming 设置 micro-batch 窗口大小为 1 s. Docker 实验使用 17.12.1-ce 版本的 Docker 系统，Flink 使用 Flink：1.4.2-hadoop28-scala_2.11 官方镜像，Spark 镜像基于 java 官方镜像(java:8-jdk)构建，Spark 版本为 spark-2.2.0-bin-hadoop2.7.

Docker 通过使用 Namespace 和 CGroups 机制实现系统环境和计算资源的隔离，无需虚拟化硬件，应用直接使用宿主机器硬件资源，因此计算性能几

乎与宿主机一致，而 KVM 由于使用 hypervisor 虚拟硬件资源，且客户机操作系统同样需要消耗计算资源，导致性能损耗较大. 研究表明[25]，Docker 在 CPU 和内存随机读写方面性能几乎与宿主操作系统一致，而 KVM 的 CPU 利用率仅为宿主机的 50%，内存随机读写由于比 Docker 多一层虚拟物理地址转换，较 Docker 约有 10%的性能损失. 虚拟化环境延迟 CDF 如图 18 所示，Hummer 在 100%情况下延迟优于其他系统. 系统平均延迟、镜像体积及启动时间如表 4 所示，其中，镜像体积为包含用户作业代码的体积，启动时间包括系统启动以及用户作业部署. 实验表明，无论对比 Docker 或 KVM 环境，Hummer 在平均延迟、镜像体积以及启动时间上全部占优. 且如果将 Hummer 绕过操作系统部署在 ESXI 等虚拟化环境，并利用硬件直通技术直连网卡，性能会有进一步提升.

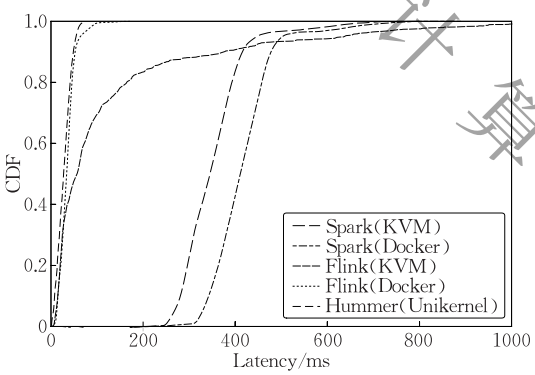



图 18 虚拟化环境延迟 CDF 图

**表 4 虚拟化环境系统性能对比表**

| 系统名称 | 平均延迟/ms | 镜像体积 | 启动时间/s |
|---|---|---|---|
| Spark Streaming(KVM) | 351.90 | 5 GB | ~60 |
| Spark Streaming(Docker) | 416.76 | 1.2 GB | ~6 |
| Flink (KVM) | 129.83 | 5 GB | ~60 |
| Flink (Docker) | 35.57 | 800 MB | ~6 |
| Hummer (Unikernel) | 28.42 | 100 MB | ~2 |

Docker 虽然具有良好的计算性能，但其资源隔离能力较差，CGroups 机制只能限制最大计算资源，而无法限制资源不被其他应用使用. 另外，由于 Docker 共用操作系统内核，导致其安全性较差. Hummer 在高性能保障前提下解决了 Docker 的弱资源隔离问题，相对 Docker 和 KVM 环境，具有镜像体积小和启动时间快等优势. 本实验中，包含用户代码的 Hummer 系统镜像文件仅为 100 MB，启动时间约为 2 s.

## 5 结 论

本文介绍了一种基于 C++ 实现的支持 Unikernel 的流式大数据计算引擎 Hummer. 系统使用 Dataflow 模型，同时支持流式和批式数据处理. 吞吐量达到同样使用 Dataflow 模型的 Flink 两倍，与使用 BSP 模型的批式处理引擎 Spark 基本持平. 同时，系统延迟较 Flink 低 1.7 倍，较 Spark Streaming 低 15.8 倍. 此外，系统支持精确到单个任务的网络配置，可解决异构网络部署及网络资源隔离等问题. 最后，系统支持绕过传统操作系统，直接部署于虚拟化环境，为流计算引擎在虚拟化环境使用提出了一种新的思路.

未来工作包括以下三点：(1) API 完善. 系统时间窗口机制支持较少，目前仅支持基于处理时间的固定时间窗口，其他类型支持后续有待完善；(2) Unikernel 支持. 目前 Unikernel 网卡驱动移植尚未完成，未来希望该系统可以支持 VMware ESXI 等虚拟化平台并通过网卡硬件直通技术加速；(3) 容错及动态负载均衡支持. 目前系统使用通用的分布式全局一致性快照算法进行容错，没有针对虚拟化环境作特别优化，未来希望系统可针对 Unikernel 单进程及虚拟化平台特性重新设计高效的容错方案，以支持流式处理场景下的动态弹性伸缩及负载均衡.

**致 谢** 在此，作者向对本文的工作给予支持和建议的同行表示感谢！

## 参 考 文 献

[1] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing//Proceedings of the USENIX Conference on Networked Systems Design and Implementation. San Jose, USA, 2012: 2-2

[2] Zaharia M, Xin R S, Wendell P, et al. Apache spark: A unified engine for big data processing. Communications of the ACM, 2016, 59(11): 56-65

[3] Toshniwal A, Taneja S, Shukla A, et al. Storm@Twitter//Proceedings of the ACM SIGMOD International Conference on Management of Data. Snowbird, Utah, USA, 2014: 147-156

[4] Carbone P, Katsifodimos A, Ewen S, et al. Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4): 28-38

[5] Bratterud A, Walla A A, Haugerud H, et al. IncludeOS: A minimal, resource efficient Unikernel for cloud services// Proceedings of the International Conference on Cloud Computing Technology and Science. Vancouver, Canada, 2015: 250-257

[6] Madhavapeddy A, Scott D J. Unikernels: The rise of the virtual library operating system. Communications of the ACM, 2014, 57(1): 61-69

[7] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107-113

[8] Saha B, Shah H, Seth S, et al. Apache Tez: A unifying framework for modeling and building data processing applications//Proceedings of the ACM SIGMOD International Conference on Management of Data. Melbourne, Australia, 2015: 1357-1369

[9] Isard M, Budiu M, Yu Y, et al. Dryad: Distributed data-parallel programs from sequential building blocks. ACM SIGOPS Operating Systems Review, 2007, 41(3): 59-72

[10] Yu Y, Isard M, Fetterly D, et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language//Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. San Diego, USA, 2008: 1-14

[11] Chambers C, Raniwala A, Perry F, et al. FlumeJava: Easy, efficient data-parallel pipelines//Proceedings of the ACM SIGPLAN Conference on Programming Language Design & Implementation. Toronto, Canada, 2010: 363-375

[12] Valiant L G. A bridging model for parallel computation. Communications of the ACM, 1990, 33(8): 103-111

[13] Chintapalli S, Peng B J, Poulosky P, et al. Benchmarking streaming computation engines: Storm, Flink and spark streaming//Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops. Chicago, USA, 2016: 1789-1792

[14] Johnston W M, Hanna J R P, Millar R J. Advances in dataflow programming languages. ACM Computing Surveys, 2004, 36(1): 1-34

[15] Graefe G. Encapsulation of parallelism in the volcano query processing system. ACM SIGMOD Record, 1990, 19(2): 102-111

[16] Chandrasekaran S, Cooper O, Deshpande A, et al. TelegraphCQ: Continuous dataflow processing//Proceedings of the ACM SIGMOD International Conference on Management of Data. San Diego, USA, 2003: 668-668

[17] Abadi D J, Carney D, Cherniack M, et al. Aurora: A new model and architecture for data stream management. The VLDB Journal, 2003, 12(2): 120-139

[18] Murray D G, Mcsherry F, Isaacs R, et al. Naiad: A timely dataflow system//Proceedings of the 24th ACM Symposium on Operating Systems Principles. New York, USA, 2013: 439-455

[19] Schelter S, Ewen S, Tzoumas K, et al. All roads lead to Rome: Optimistic recovery for distributed iterative data processing//Proceedings of the 22nd ACM International Conference on Information and Knowledge Management. San Francisco, USA, 2013: 1919-1928

[20] Lin W, Fan H, Qian Z, et al. StreamScope: Continuous reliable distributed processing of big data streams//Proceedings of the USENIX Conference on Networked Systems Design and Implementation. Boston, USA, 2016: 439-453

[21] Chandy K M. Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems, 1985, 3(1): 63-75

[22] Bratterud A. Maximizing hypervisor scalability using minimal virtual machines//Proceedings of the IEEE International Conference on Cloud Computing Technology and Science. Bristol, UK, 2013: 218-223

[23] Hewitt C, Bishop P, Steiger R. A universal modular ACTOR formalism for artificial intelligence//Proceedings of the International Joint Conference on Artificial Intelligence. California, USA, 1973: 235-245

[24] Lightweight asynchronous snapshots for distributed dataflows. http://arxiv.org/abs/1506.08603, 2015, 6, 29

[25] Felter W, Ferreira A, Rajamony R, et al. An updated performance comparison of virtual machines and Linux containers//Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software. San Jose, USA, 2007: 171-172

**LI Bing**, Ph. D. candidate. His research interests include stream computing system, storage system and distributed system.

**ZHANG Zhi-Bin**, Ph. D., associate professor. His research interests include network flow processing and network measurement.

**ZHONG Qiao-Ling**, Ph. D. candidate. His research interests include deep learning, deep learning system and distributed system.

**CHENG Xue-Qi**, Ph. D., professor, Ph. D. supervisor. His research interests include information retrieval, social computing, and distributed computing.

**Background**

Computing engine is a new research hotspot in big data research area. It aims to provide powerful computing support for data mining. The development of data mining research is inseparable from computing engine. Recently, the big data industry has made great progress in computing engine technology. For example, the first-generation open source big data processing technology, called Hadoop, can handle super-large-scale data collection. Besides, the second-generation big data processing engine, called Spark, makes better use of memory and further accelerates the performance of big data processing. The value of data is not only reflected in the spatial dimension, but also extends in time dimension. Although Spark has good data processing throughput, the latency is too high for real-time computing because of the BSP model. Particularly, Flink using the Dataflow model can make up for this shortcoming. Further, in order to integrate computing resources and support multi-core or many-core processors, virtualization has been widely used in data center, but existing stream computing engines have not optimized for virtualization.

The contributions of this paper are summarized as follows. First of all, this paper presents the proposed stream computing engine, the so-called Hummer, which supports Unikernel and directly runs on hypervisor or bare-metal environment by bypassing operating system. It eliminates the unnecessary computation overhead in traditional operating systems and reduces the scheduling conflict caused by traditional operating systems and hypervisor. It solves the problem that C++ applications need to compiled locally and difficult to deploy in the cluster. To the author's best knowledge, we are the first to apply Unikernel to the design of big data stream computing engines. Secondly, to adapt the architecture of the Unikernel. We propose a flexible task-oriented network communication solution to decouple the network communication component from TaskManager as normal task. This brings many benefits, such as heterogeneous network support and network source isolation. Thirdly, this paper evaluates the performance of stream computing system using Unikernel, Docker and CentOS, and discuss the advantages and disadvantages of the stream computing system when using container and virtualization. We propose a new design idea for high performance big data computing engine.