计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2021.20190723
58(1):164-177,2021

基于在网计算加速的拜占庭容错算法

杨 帆[1,2] 张 鹏[1,2] 王 展[1] 元国军[1] 安学军[1]
[1](中国科学院计算技术研究所 北京 100190)
[2](中国科学院大学 北京 100049)
(yangfan@ncic.ac.cn)

Accelerating Byzantine Fault Tolerance with In-Network Computing

Yang Fan[1,2], Zhang Peng[1,2], Wang Zhan[1], Yuan Guojun[1], and An Xuejun[1]
[1](*Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)
[2](*University of Chinese Academy of Sciences, Beijing* 100049)

Abstract Byzantine fault tolerance algorithm is one kind of fault-tolerant algorithms which can tolerate various software errors and system vulnerabilities. It is of vital importance to the reliability of cloud computing. Compared with other fault-tolerant algorithms, such as proof-of-work (PoW), Byzantine fault tolerance algorithm is much more stable, however, its poor performance cannot meet the demand of cloud computing which requires high throughput and low latency. In-network computing is a data-centric architecture that uses the network to perform some calculations. Using in-network computing, data can be processed as it moves, thereby improving system performance. To solve the performance problem of Byzantine fault tolerant system, in this paper, we propose a Byzantine fault tolerance algorithm optimization strategy with in-network computing, which offloads some of the computational tasks to the network interface card (NIC). The processor and NIC form a multi-stage pipeline which helps us improve the system throughput. Simply using in-network computing can not meet the performance goals in all scenarios, hence we utilize multi-threading technology to scale the system. We evaluate our method on real testbed, and the experimental results show that, compared with the default Byzantine fault tolerant system, we can obtain 46% improvement in overall throughput and 65% decrease in latency. The results have proved our solution to be available and effective.

Key words distributed system; Byzantine fault tolerant algorithms; in-network computing; accelerator; high performance computing

摘 要 拜占庭容错算法是一类能够容忍各种形式的软件错误和安全漏洞的容错算法,对云计算的可靠性保障有着重要意义.与其他容错算法相比,拜占庭容错算法稳定性更高,但是其性能表现低下,不能满足当前系统对高吞吐、低延时的需求.在网计算是一种以数据为中心的体系结构,它用网络承担部分

收稿日期:2019-10-11;**修回日期**:2020-04-14
基金项目:国家重点研发计划项目(2018YFB0204400,2016YFB0200205);国家自然科学基金青年基金项目(61702484);中国科学院战略性先导科技专项(B类)项目(XDB24050100)
This work was supported by the National Key Research and Development Program of China (2018YFB0204400, 2016YFB0200205), the National Natural Science Foundation of China for Young Scientists (61702484), and the Strategic Priority Research Program of the Chinese Academy of Sciences (class B) (XDB24050100).
通信作者:王展(wangzhan@ncic.ac.cn)

计算功能,使数据在流动过程中获得处理,从而提高系统性能.为解决拜占庭容错系统的问题,提出了一种基于在网计算的拜占庭容忍共识算法优化方案,将算法的一部分处理任务卸载到网卡上执行,利用网卡和处理器形成的多级流水线提升系统吞吐量.由于仅使用在网计算的方案在特定场景下效果不佳,因此,使用多线程方法来提升优化方案的可扩展性.同时,对算法进行了详细的系统评测,实验结果表明:相对于普通的拜占庭容错系统,使用在网计算与多线程结合的优化方案能够获得46%的吞吐率提升以及65%的延迟下降,证明了基于在网计算的拜占庭容忍共识算法优化方案的可行性与有效性.

关键词　分布式系统;拜占庭容错算法;在网计算;加速器;高性能计算

中图法分类号　TP391

随着计算机网络和分布式应用的不断发展,网络中充斥着越来越多的恶意攻击,应用的复杂度越来越高,稳定性受到严重威胁.为了给分布式应用提供可靠性保障,人们提出了拜占庭容错技术,该技术能够容忍包括人为失误、软件故障和安全漏洞等各种形式的错误和攻击[1],因此被广泛应用于区块链、数据库等对系统稳定性、安全性有着高要求的分布式系统中.

实用拜占庭容错算法(practical Byzantine fault tolerance, PBFT)[2]是最具代表性的拜占庭容错技术之一.该技术以多副本形式保证对错误的容忍,使用3阶段的共识过程来保证各副本能够以相同顺序执行相同的客户端请求,系统在受到部分节点的恶意干扰时也不会出错.与工作量证明(proof-of-work, PoW)[3]、权益证明(proof-of-stake, PoS)[4]等其他拜占庭容错算法相比,该算法具有高吞吐、低延迟的优点;而与非拜占庭容错算法(non-Byzantine fault tolerant, NBFT)相比,其稳定性较高,但是性能却较为低下.因此,在实际使用中,人们在选择使用不同的算法时往往需要在稳定性和高性能之间做出取舍.

表1给出了各种共识算法的优劣对比.为了兼顾系统的稳定性和高性能,人们提出了各项针对实用拜占庭容错算法的优化技术.我们将其分为3个方面:1)优化算法设计.尽可能简化算法的一致性协议,使得算法在没有恶意节点干扰的情况下提供高性能服务.2)降低算法开销.算法执行过程中包含大量的密码学相关的计算,降低这些计算的开销有助于提升整体性能.3)提高算法并行度.充分利用多核处理器的优势,使用多线程技术提升执行效率.

Table 1　Comparison of Different Consensus Algorithms

表1　不同共识算法性能对比

共识算法	性能	稳定性
PoW	低	高
PBFT	中	高
NBFT	高	低

现有研究工作主要着力于上述3种方案的某一方面,而缺少三者的协同优化设计.本文尝试从在网计算的角度提出一种新的实用拜占庭容错算法的优化方案,同时兼顾降低算法开销和提升算法并行度.该方案主要使用在网计算的方式将PBFT算法中共识过程中的部分计算任务卸载到网卡上去完成.一方面网卡强大的流式处理能力有助于降低延迟开销;另一方面网卡和处理器组成的多级处理流水线能够提高算法的吞吐量.同时,我们使用多线程方法进一步提高算法并行度,扩展了优化方案的适用场景.实验表明基于在网计算的拜占庭容错算法最多能够获得46%的吞吐量提升以及65%的延迟下降.

1　背景介绍

1.1　实用拜占庭容错算法

实用拜占庭容错算法广泛应用于大规模分布式系统中,为系统安全性和稳定性提供保障.该算法能够容忍不超过节点总数1/3的恶意节点;换言之,当系统节点总数为n,恶意节点数为f时,只要满足$n\geqslant 3f+1$,系统就能够对外界表现出正常的工作状态.在后续论述中我们将沿用这些参数定义.

使用该算法的系统中包含2类节点:主节点和备份节点.其中主节点负责为客户端请求选择序号,并向各备份节点广播请求.各节点按序执行客户端请求.节点通过运行3种基本协议进行协同工作,包括一致性协议(agreement)、检查点协议(checkpoint)以及视图更换协议(view change).其中一致性协议负责节点间的共识,检查点协议负责生成节点的稳定状态,视图更换协议负责主节点的切换.一致性协议是算法的主体执行部分,占据了绝大部分的时间开销,因此我们将重点介绍一致性协议,而对后两者

略去不表.

一致性协议运行在主节点为非恶意节点的情况下,该协议通过 Pre-Prepare,Prepare,Commit 这 3 阶段完成备份节点之间的共识,使得每个节点能够按相同顺序收到相同的请求,并执行相同的操作,从而保证副本之间的一致性.在一致性协议中,客户端与节点之间通过 Request 和 Reply 操作完成交互:客户端向主节点提交 Request,并收集节点返回的 Reply.当客户端收到 $f+1$ 条相同的 Reply 后,便表示 Request 已经在系统中成功提交并执行.整个算法流程分 5 个阶段执行:

1) 客户端向主节点发送 Request;

2) 主节点在收到了客户端的 Request 后进入 Pre-Prepare 阶段,将 Request 广播给主节点及其他备份节点;

3) 备份节点收到 Request 后,进入 Prepare 阶段,将 Request 广播给主节点及其他备份节点;

4) 主节点及备份节点收到广播消息后,首先进行消息验证以保证消息的完整性与不可否认性,然后保存消息以作为节点当前状态的凭证,最后将其与 Pre-Prepare 阶段的 Request 进行匹配,以判断本节点与其他节点是否可以就请求序号和请求内容达成一致,当节点收到 $2f$ 条匹配的消息后,进入 Commit 阶段,将 Request 广播给其他节点;

5) 所有节点重复 Prepare 阶段的消息接收与验证,并在收到 $2f$ 条匹配的消息后按照序号所规定的顺序依次执行各请求,在请求执行完成后向客户端返回 Reply.

图 1 给出了一致性协议的详细执行过程,其中节点 1 是主节点,其余节点是备份节点.分析可知,在一致性协议的执行过程中,一条客户端请求的成功执行至少需要进行 $6f^2+6f$ 次节点间的通信,且每次通信过程都包含消息验证、消息保存等操作,因此降低这些操作的开销对提升算法的性能有着重要意义.

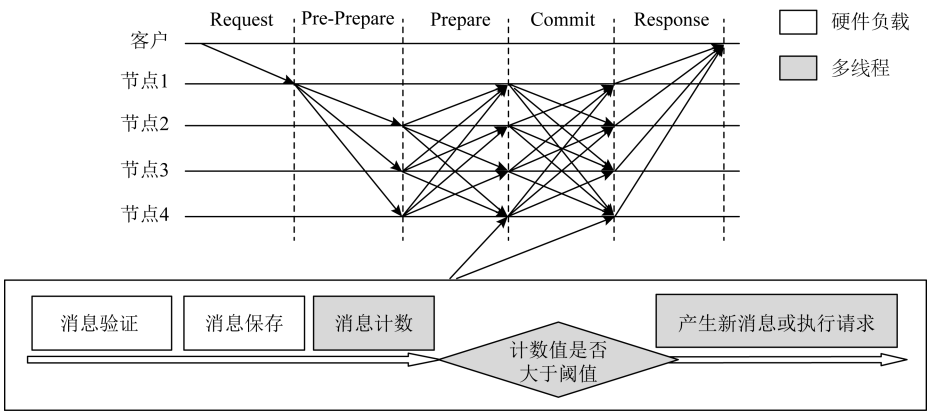


Fig. 1 Consensus protocol in PBFT

图 1 实用拜占庭容错算法的一致性协议

1.2 在网计算

在网计算是一种以数据为中心的计算模式[5],其核心思想是将一部分流式计算任务卸载到网络设备上执行,以达到提升计算效率、减少网络流量的目的.当前,在网计算已经成为优化分布式应用的重要手段之一.例如 Dang 等人[6-7]提出的 NetPaxos 使用可编程交换机对非拜占庭容错算法 Paxos 进行优化,István 等人[8]提出了基于可编程网卡的原子广播优化.

通过分析我们发现,拜占庭容错算法与在网计算有着极佳的适配性.一方面,消息验证与消息保存操作都可以在网卡上以专用硬件结构实现,其处理效率高于通用处理器;另一方面,网卡和处理器构成了多级处理流水线,能够提升算法执行的并行度.

2 设计与分析

本节着重介绍系统的整体设计方案以及设计中的考量与取舍.如引言所述,对 PBFT 的优化主要有 3 个方面:减少算法执行开销,提高并行度,以及简化算法设计.

2.1 系统整体设计

本文设计并实现了一种软硬件协同的实用拜占庭容错算法优化方案.算法的一致性协议中,特别是 Prepare 和 Commit 阶段,节点对消息的处理可以分为 4 个阶段:消息验证、消息保存、消息计数、请求执行(或生成新消息).不同的执行阶段被分配到系统中不同的组件上执行,这样的设计方案带来了 2 个

方面的显著优势:

1) 算法的密码学验证与消息保存被卸载到专用硬件网卡上执行,大大降低了处理时间开销;

2) 多个组件协同完成整个处理过程,形成了多级流水线,提升了系统的吞吐率.

图 2 和图 3 分别给出了软硬件流水线示意和系统的软硬件架构图.硬件部分的核心组件是消息保存模块及消息验证模块.其中,消息验证模块从以太网模块中获取网络数据,并使用密码学方法对消息进行安全验证,将通过验证的消息传递给消息保存模块;消息保存模块使用散列表将以键值形式组织的消息存储到主存中供拜占庭容错算法使用.软件方面,基于实用拜占庭容错算法,本文借助多线程方法,由不同线程负责客户端请求的共识和已共识请求的执行.

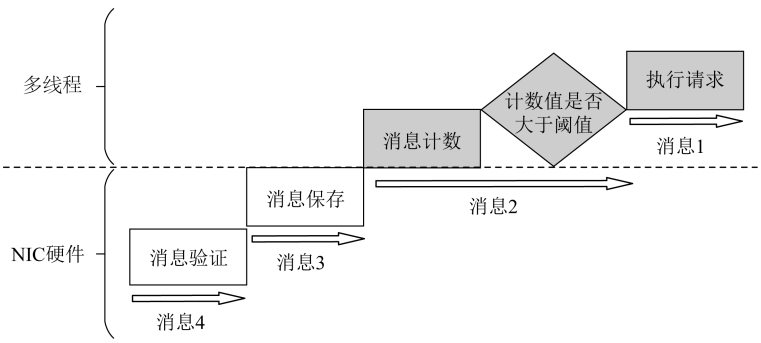


Fig. 2　Multi-stage pipeline of CPU and NIC

图 2　CPU 与网卡组成的多级流水线

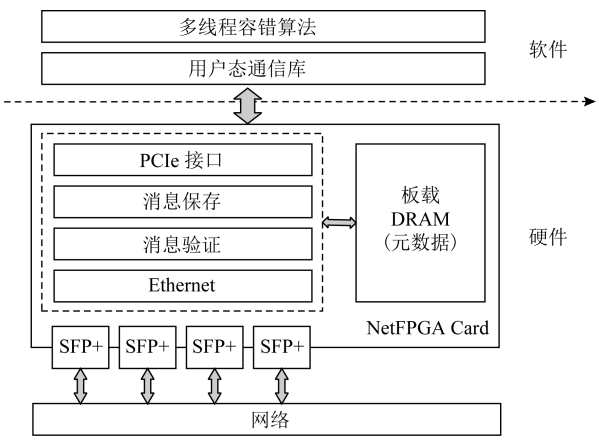


Fig. 3　Byzantine fault tolerant system architecture based on in-network computing

图 3　基于在网计算的拜占庭容错系统架构

2.2　网卡硬件加速的优势

与仅使用多线程进行优化的方式相比,使用专用硬件执行算法能够带来显著的性能提升,这种收益来自于 2 个方面:1)与通用处理器相比,专用硬件针对算法特征做了定制化设计,避免了处理器中繁琐的取指译码等流程,执行效率更高.例如,使用定制化的电路可以在几个时钟周期内完成密码学验证、散列等复杂的计算.2)多个线程主要通过共享内存的方式进行数据交互,频繁的通信过程会带来较大的内存读写开销.而硬件模块之间可以通过硬件队列和信号进行数据传递,时间开销较低.

为进一步说明这一点,本文分别使用 FPGA 和通用处理器进行 32 B 数据的 SHA-256 运算,其中硬件电路的频率为 200 MHz,处理器使用 Intel Xeon CPU E3.图 4 给出了 2 种计算模式的延迟对比.可以看到,使用硬件加速方案相对于通用处理器减少了 95%的延迟开销.

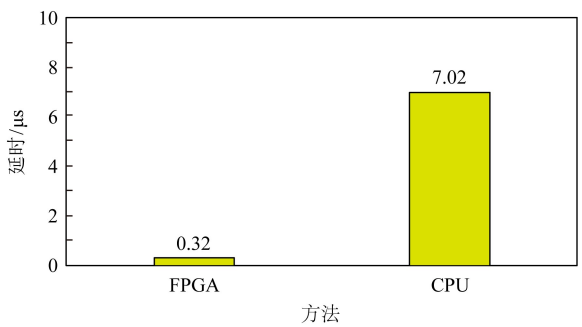


Fig. 4　Latency of different methods of SHA-256

图 4　SHA-256 在不同实现方法下的延时对比

使用加速器能够在一定程度上减少计算开销,但目前的加速器[9-10]大部分采用的是主从模式,这种模式存在较大的数据拷贝开销.以图 5 为例,网络数据首先被搬移到主存,然后处理器通知加速卡获取待处理数据.当加速卡计算完成后,主处理器读回处理结果,整个过程包含了多次数据搬移.而可编程网卡天然位于网络数据路径上,当数据从网卡流经内存就已经完成了处理,不会带来额外的数据拷贝

开销.因此,在网卡上对算法流程进行卸载是较好的选择.表 2 给出了我们对网卡硬件加速与多线程和普通加速卡的优劣对比.

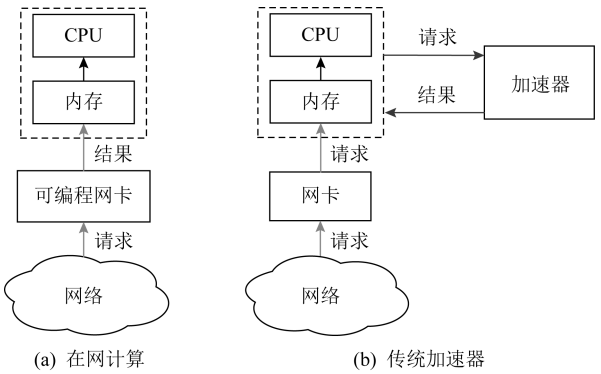


Fig. 5 Comparison of in-network computing and other accelerators

图 5 在网计算和传统加速器对比

Table 2 Comparison of Different Optimization Strategies

表 2 不同优化方式开销对比

优化方式	线程间通信成本	内存拷贝成本
多线程	高	高
传统加速器	低	高
在网计算	低	低

2.3 多线程的必要性

使用网卡硬件进行计算卸载需要满足一个先决条件,即数据包的处理应当能够达到线速,否则有可能造成网络丢包.为满足这一限制条件,我们抽象了 2 个特征对不同的计算任务进行筛选:1)没有循环计算;2)没有数据依赖.具备这 2 种特征的计算任务被称为流式计算.

基于上述分析,我们提取了算法中各个执行阶段的计算特征,如表 3 所示:

Table 3 Characteristics of Different Tasks

表 3 不同任务的计算特征

任务	循环	数据依赖	数据流计算
消息验证	否	否	是
消息保存	否	否	是
消息计数	是	是	否
请求执行	是	是	否

通过分析我们发现,算法中的消息验证与消息保存的核心是散列计算,是一种典型的流式计算,因此,这 2 部分的计算被卸载到网卡上执行.而算法中的消息计数,客户端请求的执行涉及到内存访问及循环操作,更适合在通用处理器上以更灵活的多线程方式实现.

2.4 简化算法的一致性协议

简化算法的一致性协议是拜占庭容错算法的一种重要的优化手段,许多工作使用这种方式获得了显著的优化效果,如 Zyzzyva[11],fastBFT[12],SBFT[13].

在简化算法一致性协议方面,通常是引入 1 个或多个收集节点,由该节点收集其他节点消息并广播统计情况,将多对多的消息传递转化为多对 1 和 1 对多消息传递,以降低系统的消息复杂度.Zyzzyva,fastBFT,SBFT 等都不同程度上使用了这一手段,Zyzzyva 选择使用客户端作为收集节点,而 fastBFT 和 SBFT 则从备份节点中选拔收集节点.

使用收集节点虽然能够简化算法一致性协议,但也会带来一些负面影响.首先,它会让算法的其他部分变得复杂,如 Zyzzyva 虽然在理想情况下可以将一致性协议的消息复杂度由 $O(n^2)$ 降低至 $O(n)$,但付出的代价是需要一个更为复杂的视图更换协议,而且这种更为复杂的视图更换协议有可能存在被攻击的漏洞[14].其次,使用收集节点会为算法引入更为复杂的密码学操作,如 fastBFT 和 SBFT 使用多签名的方式来保证收集节点所广播消息的可信度.

简化算法的一致性协议虽然能够降低系统消息复杂度,但也有其自身的局限性,因此我们并没有采用这种优化方式.

3 实 现

3.1 定制化消息格式

为了将计算任务嵌入到网络数据流的处理过程中,我们需要对消息进行标识,网卡通过识别消息中的特殊标志位后进行相应的处理.因此,我们对消息格式进行了定制化设计.

如图 6 所示,消息被划分为 3 个部分:消息头(header)、消息键(key)以及消息值(value).消息头主要提供消息的长度信息,以便网卡对消息负载进行界定和处理;消息内容以键值对的形式进行组织,承载应用要传递的信息.消息键承载了消息的特征信息,包括消息类型、源节点序号、请求序号、视图号,用于消息保存中的散列值计算.消息值则包含了客户端请求摘要、消息验证码向量.消息验证码向量可以保证消息在传播过程中的完整性与不可否认性,它由一组面向不同接收节点的消息验证码组成.

头	键的长度		值的长度	
键	消息类型	视图	请求序号	源节点
值	摘要			
	消息验证码1			
	⋮			
	消息验证码n			

Fig. 6　Customized message format

图 6　定制化消息格式

3.2　基于网卡的消息验证模块

消息验证模块被卸载到网卡硬件上执行,它提供了基于 SHA-256 安全散列算法的消息验证功能.如图 7 所示,该模块通过消息预处理、验证码提取、安全散列值计算、验证码对比等操作完成消息的验证.其中消息预处理模块提取出消息的长度信息、源节点信息以及消息验证向量,将其传递给验证码提取模块使用,同时,将消息的负载传递给安全散列计算模块进行密码学运算;安全散列计算模块实现了 SHA-256 安全散列算法,该算法根据消息和对应密钥计算目标验证码;验证码提取模块则从消息所携带的验证码向量中提取出与本节点对应的验证码.最后验证码对比模块将消息携带的验证码与安全散列计算得到的目标验证码对比,决定是否向后续模块传递该信息.其中安全散列计算的处理过程较为复杂,这里对其设计方案进行详细说明.

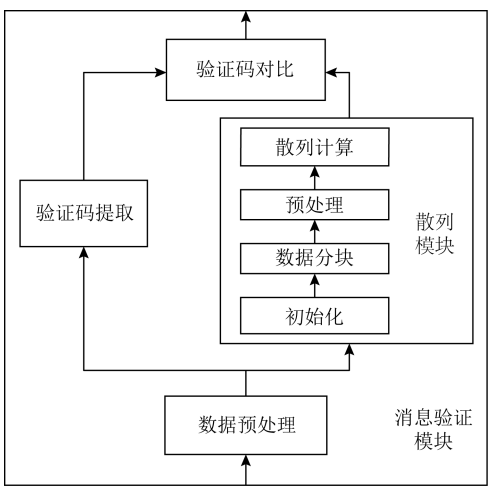


Fig. 7　Message verification module

图 7　消息验证模块

1) 初始化

这一步包含散列值初始化和密钥初始化.在 SHA-256 中,散列值是一段 256 b 的二进制串,它的初始值按如下规则计算得到:使用二进制形式计算自然数中前 8 个质数(2,3,5,7,11,13,17,19)的平方根,分别取每个计算结果小数部分的前 32 b.密钥是一段长度为 2 048 b 的二进制串,被划分为 64 个字(1 个字是 4 B),密钥由用户指定.表 4 给出了初始散列值.

Table 4　Initial Hash Values

表 4　初始散列值

初始散列序号	质数	初始散列值
0	2	0x6a09e667
1	3	0xbb67ae85
2	5	0x3c6ef372
3	7	0xa54ff53a
4	11	0x510e527f
5	13	0x9b05688c
6	17	0x1f83d9ab
7	19	0x5be0cd19

2) 数据分块

SHA-256 是一种增量式的散列算法,即一边输入数据一边更新散列值.为了实现增量计算,算法对输入数据分块,以块为单位进行散列计算,每个块的长度是 512 b.同时,算法规定要在输入数据的尾部进行填充操作.填充内容主要有数据长度和比特.数据长度填充指将输入数据的长度转换为 64 b 二进制数,填写在输入数据最后一个块的最后 64 b 上.比特填充则是用附加的位将输入数据长度扩充至 512 b 的整数倍,保证输入数据能够被划分为整数个块.而且,算法规定比特填充的长度不能为 0,至少 1 b,填充的第 1 个位为 1,其余位为 0.比特填充的位置在输入数据尾部长度填充之前.图 8 给出了数据分块示意图.

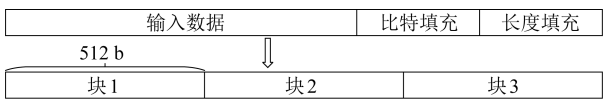


Fig. 8　Message filling and data partitioning

图 8　消息填充和数据分块

3) 预处理

当数据分块完成后,所得结果并不能直接被散列计算模块使用.需要先将一个数据块构造成 64 个 4 B 大小的字.构造规则为:

① 前 16 个字由该块平均划分得到;

② 其余字由迭代得到,其中 W_t 指代第 t 个字,$\ggg$指循环右移,$\gg$指逻辑右移:

$$W_t=\delta_1(W_{t-2})+W_{t-7}+\delta_0(W_{t-15})+W_{t-16},\quad(1)$$

$$\delta_0(x)=(x\ggg 7)\oplus(x\ggg 18)\oplus(x\gg 3),\quad(2)$$

$$\delta_1(x)=(x\ggg 17)\oplus(x\ggg 19)\oplus(x\gg 10).\quad(3)$$

4) 散列计算

散列计算是一个循环迭代的过程,一轮循环包括 64 步,每步输入预处理生成的 1 个字以及密钥中对应的 1 个字,每步都更新 1 次散列值,1 轮循环恰好处理输入的 1 个块.一步散列计算的过程依照式(4)~(10)进行:

$$f_0(x,y,z)=(x\wedge y)\oplus(y\wedge z)\oplus(x\wedge z),\quad(4)$$

$$f_1(x,y,z)=(x\wedge y)\oplus(\neg x\wedge z),\quad(5)$$

$$\Sigma_0(x)=(x\gg 2)\oplus(x\gg 13)\oplus(x\gg 22),\quad(6)$$

$$\Sigma_1(x)=(x\gg 6)\oplus(x\gg 11)\oplus(x\gg 25),\quad(7)$$

$$\{B_{i+1},C_{i+1},D_{i+1},F_{i+1},G_{i+1},H_{i+1}\}=\{A_i,B_i,C_i,E_i,F_i,G_i\},\quad(8)$$

$$E_{i+1}=D_i+\Sigma_1(E_i)+f_1(E_i,F_i,G_i)+H_i+K_i+W_i,\quad(9)$$

$$A_{i+1}=\Sigma_0(A_i)+f_0(A_i,B_i,C_i)+\Sigma_1(E_i)+f_1(E_i,F_i,G_i)+H_i+K_i+W_i,\quad(10)$$

其中,$A\sim H$ 是 8 个字,它们共同组成散列值,W_i 代表输入的第 i 个字,K_i 代表密钥的第 i 个字.

图 9 是计算过程示意图.从图 9 可以看到,散列计算过程中,新散列值可以分为 3 个不同部分:1)B,C,D,F,G,H,它们仅仅由旧散列值的对应字右移得到;2)E,它由旧散列值的 D,E,F,G,H 以及输入字和密钥经过与、非、异或、加法计算得到;3)A,它由旧散列值中除 D 外所有字以及输入字和密钥经过与、非、异或、加法计算得到.实际上,算法通过对 A 和 E 的计算及以字为单位的右移,让输入数据信息能够影响散列值每一位,从而使最终得到的散列值能够代表输入数据.

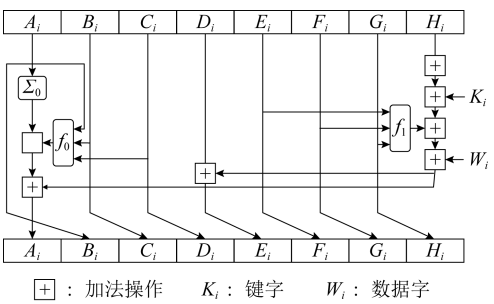


Fig. 9 SHA-256 Hash calculation

图 9 SHA-256 散列计算

3.3 基于网卡的消息保存

如 2.1 节所述,消息保存功能也被卸载到网卡上执行.如图 10 所示,消息保存模块由散列计算、冲突管理、数据上传 3 个子模块组成.节点收到的消息通过散列表的方式进行保存,我们在板载 DRAM 中保存散列表的元数据,在主存中保存实际数据.散列计算模块负责提取消息键并计算散列值,散列值决定了消息所占用的散列表项;冲突管理模块负责解决散列冲突问题;数据上传模块负责将消息封装为 PCIe 事务包并上传.接下来阐述 3 个模块的详细设计.

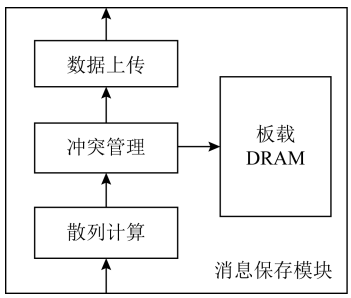


Fig. 10 Message store module

图 10 消息保存模块

1) 散列计算子模块

散列计算子模块根据消息中的键计算出消息的散列值.该模块实现了非密码学相关散列算法 Murmur 来进行散列表索引的计算,与密码学相关算法 SHA-256 相比,该算法计算过程更加简洁,延迟更低.同时,Murmur 已经被广泛应用于主流数据库应用中,具有较高的安全性和稳定性.

Murmur 的主要计算过程分为 3 部分:数据体处理、数据尾处理和散列值处理.数据体指输入数据中可以被划分为整块的部分,块长度为 128 b,数据体处理指将输入数据以块为单位划分并进行散列计算.数据尾是指输入数据结尾不够一整块的部分,数据尾处理指以数据尾为输入时进行的散列计算.散列值处理,数据尾计算得到的散列值并不是最终的输出散列值,还需要经过散列值处理才能输出.

步骤 1. 数据体处理.输入数据被切分成 128 b 的块,每一块分别进行常量乘、循环左移、异或、常量加等操作.具体计算过程:

$$t=H_i\oplus(((C_i\times c_1)\lll r_1)\times c_2),\quad(11)$$

$$H_{i+1}=(t\lll r_2)\times m+n,\quad(12)$$

其中,C_i 为输入块,$\lll$ 代表循环左移操作,H_i 代表当前散列值,H_{i+1} 代表更新后的散列值,c_1,c_2,r_1,r_2,m,n 均为常量.

步骤 2. 消息尾处理.当输入数据结尾部分不足以构成一个完整的块时,需要使用专门的消息尾

处理操作进行散列值计算.消息尾有其特定的计算方式,首先调整消息尾端序,若本身是大端序则调整为小端序,反之亦然;然后进行常量乘、循环左移、异或操作:

$$H_o=H_i\oplus((S(T)\times c\lll r_1)\times c_2),\quad(13)$$

其中,T 代表数据尾,S 代表端序调整函数,H_i 代表当前散列值,H_o 代表计算所得散列值.

步骤 3. 散列值处理.Murmur 算法规定,当所有数据输入完毕,计算所得到的散列值不能直接输出,而是通过一系列异或、移位、常量乘运算才能获得最终的散列值,计算过程为

$$f(x,y)=x\oplus(x\gg r),\quad(14)$$

$$H_o=f(f(f(H_i\oplus L,r_3)\times c_3,r_2)\times c_4,r_3),\quad(15)$$

其中 L 代表输入消息的长度.

2) 冲突管理子模块

散列冲突指的是多个输入在经过散列计算后得到同一个散列值的情况.在散列表中,发生散列冲突时,多个输入会被映射到同一个散列表项中,导致数据的丢失.因此,本文设计了一种多槽位散列冲突解决方案.如图 11(a)所示,每一个散列表项包含多个槽位,发生散列冲突的消息放置在同一个散列表项的不同槽位中.同时,如图 11(b)所示,该模块在网卡的板载 DRAM 中维护一个位图,以记录散列表项中槽位的使用情况.

	散列桶0	散列桶1	散列桶2	散列桶3	散列桶4	散列桶5
槽位0	KV0_0	KV1_0	KV2_0	KV3_0	KV4_0	KV5_0
槽位1	KV0_1		KV2_1	KV3_1		KV5_1
槽位2	KV0_2			KV3_2		
槽位3						

(a) 散列表

	散列桶0	散列桶1	散列桶2	散列桶3	散列桶4	散列桶5
槽位0	1	1	1	1	1	1
槽位1	1	0	1	1	0	1
槽位2	1	0	0	1	0	0
槽位3	0	0	0	0	0	0

(b) 位图

Fig. 11　Multi-slot Hash table and bitmap

图 11　多槽位散列表及位图

图 12 给出了冲突管理模块的架构图,该模块包含 3 个子模块:位图读取模块、位图管理模块以及调度模块.位图读取模块根据散列值生成位图访问请求,该请求读取位图中对应项槽位的使用情况.位图管理模块主要负责接收位图返回的信息,根据槽位的使用情况为消息分配槽位,并将新的槽位使用情况写回位图中.由于位图读取模块和位图管理模块均需要向板载 DRAM 发送请求,所以需要调度模块负责消息的调度,避免发生竞争.

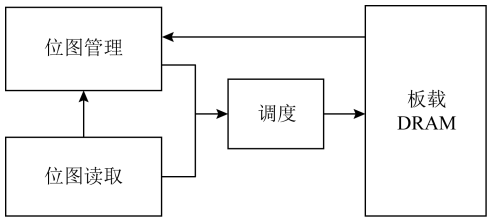


Fig. 12　Collision management module

图 12　冲突管理模块

3) 数据上传模块

数据上传模块根据消息的散列值和槽位号计算消息在散列表中的偏移,进而计算消息在内存中的地址,然后将消息封装为 PCIe 事务包传递给网卡的 PCIe 接口.表 5 给出了各项参数的含义.需要说明的是这里的内存地址指的是物理内存地址,我们配置操作系统预留充足的物理内存空间供散列表使用,应用程序通过内存重映射方式访问该内存区域.

待保存消息对应内存地址的计算方法为

$$A=H_A+S_W\times(H\times S_N+S_{ID}),\quad(16)$$

其中各参数的定义如表 5 所示:

Table 5　Parameters of Memory Address

表 5　内存地址计算的参数定义

参数	定义
A	消息内存地址
S_W	槽位宽度
H	消息散列
S_N	每个散列桶的槽位数
S_{ID}	消息槽位 ID
H_A	散列表基准地址

3.4　多线程拜占庭容错

图 13 是算法一致性协议的流程图.我们将算法的一致性协议划分为共识和执行客户端请求 2 个过程,分别由图 13(a)(b)表示的线程负责.其中虚线部分为卸载到网卡上的消息验证与消息保存操作.如图 13 所示,图 13(a)是共识线程中部分流程图,该线程监听等待网络消息,当网络消息到达后,判断网络消息类型,执行相关处理流程.Request 消息是客户端发送的请求,只有主节点会收到该请求,当主节点收到 Request 请求后,使用消息验证码验证消息来源是否可靠,并为其选取一个请求号,请求号按主节点

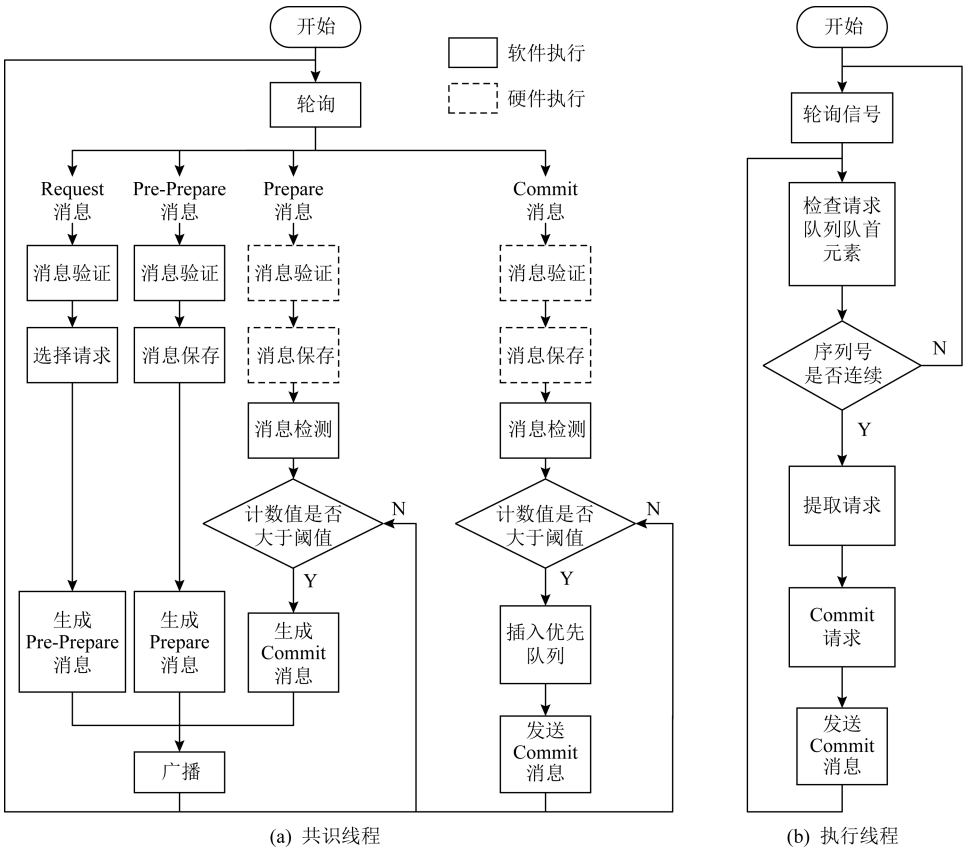


Fig. 13 Flow chart of consensus protocol

图 13 一致性协议流程图

收到客户端请求的顺序递增选取,然后根据客户端请求生成 Pre-Prepare 消息并广播.节点收到 Pre-Prepare 消息,先根据消息验证码对消息来源进行验证,保存 PrePrepare 消息及其包含的客户端请求,生成 Prepare 请求并广播.节点收到 Prepare 消息,根据消息验证码验证消息来源,保存消息及其验证码向量作为凭证,为剔除恶意节点的干扰检查收到的 Prepare 消息与同请求号的 Pre-Prepare 消息内容是否相符,对通过检查的消息计数,当计数值达到阈值时,生成 Commit 消息.收到 Commit 消息的处理流程与收到 Prepare 的流程类似,区别在于消息计数值达到阈值时共识完成,提交本次所共识客户端请求;这里的提交是指,将客户端请求插入优先队列,并发送执行信号给请求执行线程.需要说明的是图 13(a)中虚线部分在本文中卸载到网卡上以硬件形式完成,而非以软件形式实现.

图 13(b)是请求执行线程流程图,该线程并没有采用无间断轮询优先队列的方式,而是在没有请求待执行的时候,让线程阻塞在相关信号的等待上,节约了系统计算资源.该线程收到请求执行信号后,读优先队列头部请求的序号,判断该请求序号与最近执行的最后 1 条请求序号是否连续,执行该判断的目的是保证节点严格按照已共识的请求号顺序执行客户端请求.若连续,则从优先队列取出并执行该请求,并根据执行结果向客户端发送 Reply 响应;然后,按照上述描述继续读取并执行请求,直到队头请求与最近执行请求的序号不连续为止.

4 系统评测

4.1 实验环境

本文使用 2 台物理机作为测试平台,物理机配置如表 6 所示.我们通过配置内核启动参数 memmap 预留 4 GB 内存作为保存消息的散列表使用.散列表中每个散列表项静态配置 4 个槽位,每个槽位宽度为 256 b.网卡硬件使用 NetFPGA-SUME 开发板进行消息验证与消息保存的卸载,通过 8 通道 PCIe Gen3 接口与主板连接,提供 SFP+网络接口,带宽为 10 Gbps,同时,我们使用 Intel X710-DA4 网卡作为对照实验.

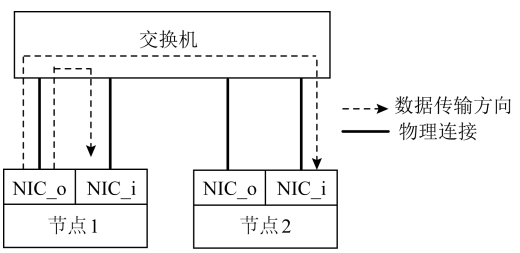
Table 6　System Configurations

表 6　系统配置

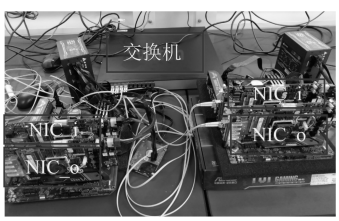
硬件	配置
CPU	Intel® Xeon® CPU E3-1220 v5@3.00 GHz
内存/GB	32
OS	CentOS 6.9(Kernel Version 2.6.32)
NIC 1	NetFPGA-SUME
NIC 2	Intel X710-DA4

NetFPGA 网卡的网络通信部分借助于 cHPP 控制器[15-16]实现,cHPP 控制器提供了一整套节点间通过直接网络互联通信的解决方案,包括硬件网卡、设备驱动和软件通信库.

为了在 2 个物理节点上真实展现多节点执行拜占庭共识算法时的通信场景,每个节点上使用 2 张网卡.如图 14(a)所示,1 张网卡发送消息,1 张网卡接收消息,接收消息的网卡负责消息验证与消息保存的卸载.2 个物理节点共使用 4 张网卡,网卡之间通过以太网交换机进行数据转发.图 14(b)展示了本文搭建的原型系统.



(a) 节点互联



(b) 原型系统

Fig. 14　Interconnection of different nodes

图 14　节点间互联方式

4.2　评测方法

测评主要针对多种不同场景对系统进行压力测试,以观察各种优化手段的效果.客户端和服务器端以同步方式工作,客户端向主节点发送请求并等待节点响应,当收到 $f+1$ 条符合要求的响应后认为系统已经完成当前请求的提交,继续发送下一条请求.为了给系统提供足够的压力,需要使用多个客户端同时发起请求.

评测主要关注系统吞吐和延时,以每秒钟系统所完成的共识次数来代表系统吞吐,以客户端发出请求到收到 $f+1$ 条响应之间的间隔为系统延时.所有测试结果均是多次测量并去掉最大值、最小值,取剩余测试结果的平均值得到.

由于本文同时使用了在网计算和多线程的优化方法,因此,为了评估每种优化手段对性能提升的影响,测评中共包含 4 种配置拜占庭容忍共识算法实现,如表 7 所示.首先,根据是否使用在网计算方法将实验配置划分为 2 类:使用在网计算方法的实验配置中由我们的在网计算平台 NetFPGA 完成消息验证与消息保存的卸载;不使用在网计算方法的实验配置则采用具有相同带宽的 Intel X710 网卡.同时,根据是否使用多线程方法将实验配置划分为 2 类:1)使用单线程串行执行算法共识过程和请求执行过程;2)使用多线程分别负责算法共识过程和请求执行过程.

Table 7　Experiment Configuration

表 7　实验配置

实验组别	在网计算	多线程或单线程
1	不使能	单线程
2	不使能	多线程
3	使能	单线程
4	使能	多线程

为简述起见,下文的描述以及图表中,以 INC 代表在网计算,N-INC 代表非在网计算,ST 代表单线程,MT 代表多线程.

4.3　评测结果分析

4.3.1　不同请求压力下的系统性能表现

本节对比了不同请求压力下系统的吞吐和延时变化.客户端以同步方式工作,单客户端不足以让服务端满负荷运转,所以使用多客户端同时发送请求的方式,客户端数量代表了请求压力大小.这里共启动 2 个服务节点,然后逐步增加客户端的数量,服务端对该请求提供简单的计数器服务.

图 15 是 2 种配置下系统吞吐随客户端数量增长的对比.首先,2 种配置下吞吐变化趋势相同,都是随着客户端数量的增长,吞吐先快速上升,后慢速上升,最后到达一个稳定值.当客户端数量过少时,不能给系统提供足够的压力;随着客户端数量增加,系统压力逐渐增大,吞吐随之升高;当客户端数量为 12 个时,系统满负荷运转,系统吞吐到达一个稳定值.其次,综合使用在网计算与多线程的配置能获得

更好的吞吐性能,与非在网计算加单线程配置相比,最大可获得46%的吞吐提升.吞吐性能的提升得益于系统并行度的提升.如4.2节所提到的,NetFPGA卸载消息验证与消息保存操作,能够在网卡和CPU上形成一个消息处理的多级流水线;而且多线程方式执行共识过程和请求执行过程,进一步提升了系统的并行度.

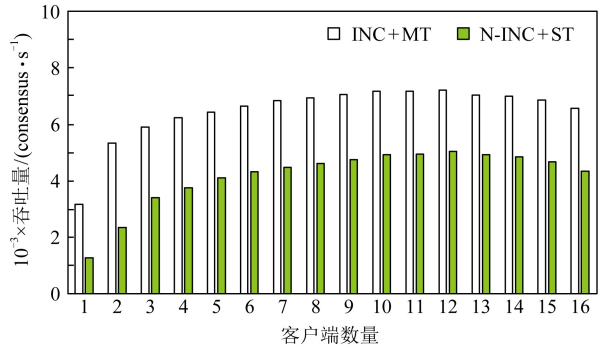


Fig. 15 System throughput

图15 系统吞吐量

当客户端数量继续增加,系统吞吐量出现下降趋势,这是因为实验中使用的物理机1具有16个核,主节点程序的4个线程(轮询客户端请求、轮询共识请求、共识线程以及请求执行线程)分别绑定到4个核上,每个客户端程序也绑定到一个特定的核上.因此当客户端数量超过12个时,就会出现处理器资源的竞争,从而使得单位时间内完成的共识请求数量减少,同时,单次共识请求的延时也随之上升.但是,与非在网计算加单线程的配置相比,在网计算加多线程的优化方式仍然具有较高的吞吐提升.

图16是2种配置下系统延时随客户端数量增长的对比.与吞吐量的变化趋势类似,延时先慢速增长,后快速提升.比较两者最小延时,可以看到,在请求压力较大时,系统延时下降达到了65%.系统延时

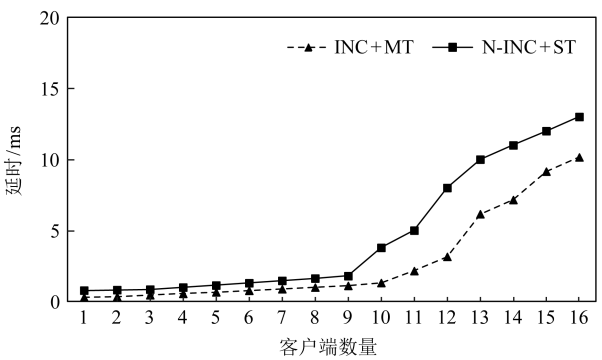


Fig. 16 Latency of request

图16 请求延时

的降低有2方面原因:1)正如2.2节所介绍的,基于在网计算的消息验证与消息保存的卸载会降低系统延时;2)在NetFPGA上借助cHPP通信库实现网络通信,用户程序可以借助其通信库直接访问网卡驱动,绕过了操作系统内核,减少了软件调用层次和内存拷贝,从而进一步降低了系统延时.

4.3.2 在网计算与多线程的特点分析

本节主要对比了不同应用负载下多线程和硬件卸载的优化效果.实验配置是2个服务节点、多客户端同时发起请求,客户端数量为12个,服务端对请求提供简单的计数服务.本实验引入了不同的应用负载,用服务端收到一个已共识请求后对计数器进行的循环自增次数代表应用负载.

图17展示了不同应用负载下在网计算和多线程优化的效果.从图17中能够看到,随应用负载的增加,系统的吞吐性能是逐渐降低的.而且,在应用负载较小时,在网计算方法在优化中起主导作用,随着应用负载增大,多线程优化的效果逐渐凸显,当增大到10万次循环自增时,多线程方法在优化中占据主导作用.从图17中我们能得到2点结论:1)综合使用多线程和在网计算优化方法的效果要好于单独使用一种优化方法;2)多线程优化和在网计算优化是一种相辅相成的优化方式,在网计算优化在应用负载较小时有较好的优化效果,多线程优化在应用负载较大时有较好的表现.

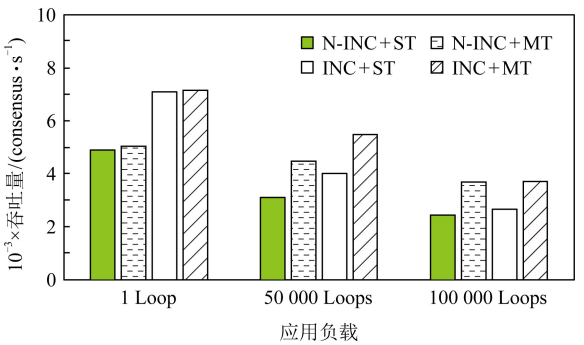


Fig. 17 System throughput of different applications

图17 不同应用负载下的系统吞吐量

与单独使用一种优化方式相比,综合优化能够在网卡与CPU上建立一条更深的多级流水线,获得更好的系统并行度.另一方面,当应用负载较小时,执行客户端请求消耗的时间较小,单线程与多线程的吞吐差别不大,这时使用多线程优化收益较低;而随着应用负载增大,请求执行时间增加,使用多线程优化收益逐渐凸显. 图18展示不同应用负载下

的系统最小延时.从图 18 中能够看到,系统延时随应用负载的增大而增大;在网计算方法能够降低系统延时,但是多线程方法会在一定程度上提高系统延时.

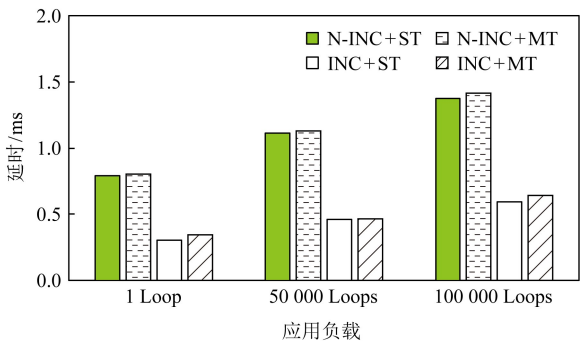


Fig. 18　System delay of different applications

图 18　不同应用负载下的系统延时

在网计算方法能够降低系统延时的现象可以从 2 方面解释:1)消息验证与消息保存的卸载会在一定程度上降低系统延时;2)非在网计算配置下系统使用 X710 网卡,而在网计算配置下系统借助 NetFPGA 上实现的 cHPP 控制器进行网络通信,该配置下用户使用通信库直接访问网卡驱动,绕过了系统内核,也有利于降低系统延迟.

多线程方式会在一定程度上提高系统延时的现象主要原因是:为了使用多线程优化,需要通过队列进行线程间的消息传递,而且在完成共识后还需要共识线程发送事件信号唤醒请求执行线程,线程间的交互和线程从挂起到唤醒的时间损耗为系统延时带来了负面影响.不过从实验结果中可以看到,多线程方式对系统延时的影响并不大,仅有 10～30 μs 的额外时间开销.

4.3.3　优化对算法可扩展性的影响

从 2.1 节的分析中我们知道,实用拜占庭容忍共识算法的消息复杂度为 $O(n^2)$,n 为节点数目,系统可扩展性不佳.本节通过观察在不同节点规模下的优化效果,探讨本文所述优化方法对算法可扩展性的影响.实验配置是采用 2 个物理节点,运行 2～7 个服务节点进程,多客户端同时发起请求,服务端对请求提供简单的计数服务.

图 19 和图 20 展示了不同服务节点规模下综合使用在网计算优化和多线程优化的效果.图 19 是不同系统规模下的吞吐性能表现,图 20 是不同系统规模下的延时表现.首先,随着服务节点数量的增加,系统吞吐逐渐下降,系统延时逐渐上升.因为随着服务节点数量的增加,完成一次共识所需通信次数也随之增加,每个节点需要处理更多消息.其次,使用本文所述的优化方法能够在不同系统规模下获得较好的性能表现,缓解系统规模扩大带来的性能下降问题.本文所述优化方法能够在一定程度上提高系统的可扩展性.

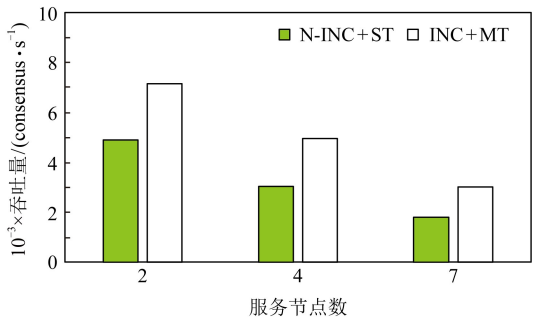


Fig. 19　System throughput of different scales

图 19　不同规模下系统吞吐量

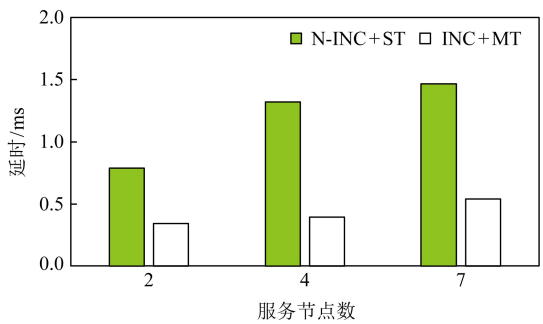


Fig. 20　System delay of different scales

图 20　不同规模下系统延时

5　总结与展望

拜占庭容忍共识算法为云计算和高性能计算提供了一种高可靠的容错方案,但是其通信需求高、消息验证需求高、消息保存需求高的“三高”特点限制了它的性能表现.本文提出一种基于在网计算的拜占庭容忍共识算法优化方案,尝试解决其性能低下的问题.该解决方案将算法中的部分执行流程卸载到网卡硬件上执行,并通过多线程优化方法提高算法并行度.测试结果证明本文所述优化方法不仅可以提高算法吞吐、降低延时,还可以增强算法的可扩展性.

未来我们将考虑对现有设计方案进行优化,主要方向有 2 点:

1) 硬件广播.当前算法实现中使用了应用层广播而非硬件广播,在网络中加入硬件广播的支持,可能会获得进一步的性能提升.

2) 更多操作的硬件卸载.可以尝试卸载更多的处理流程,如发送端的消息验证码的生成等.实际上,最终可以把共识过程作为下层协议与上层的请求执行过程分离,整个共识过程实现在 NetFPGA 或专用芯片上,通用处理器只负责业务逻辑的执行.

参 考 文 献

[1] Fan Jie, Yi Letian, Shu Jiwu. Research on the technologies of Byzantine system [J]. Journal of Software, 2013, 24(6): 1346-1360 (in Chinese)
(范捷, 易乐天, 舒继武. 拜占庭系统技术研究综述[J]. 软件学报, 2013, 24(6): 1346-1360)

[2] Castro M, Liskov B. Practical Byzantine fault tolerance [C] // Proc of the 3rd USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 1999: 173-186

[3] Gervais A, Karame G O, Wüst K, et al. On the security and performance of proof of work blockchains [C] //Proc of the 23rd ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2016: 3-16

[4] Vasin P. Blackcoin's proof-of-stake protocol v2 [OL]. [2019-09-17]. https://blackcoin. co/blackcoin-pos-protocol-v2-whitepaper. pdf

[5] Graham R L, Bureddy D, Lui P, et al. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction [C] //Proc of the 1st Int Workshop on Communication Optimizations in HPC. Piscataway, NJ: IEEE, 2016: 1-10

[6] Dang H T, Sciascia D, Canini M, et al. Netpaxos: Consensus at network speed [C] //Proc of the 1st ACM SIGCOMM Symp on Software Defined Networking Research. New York: ACM, 2015: 1-7

[7] Dang H T, Canini M, Pedone F, et al. Paxos made switch-y [J]. ACM SIGCOMM Computer Communication Review, 2016, 46(2): 18-24

[8] István Z, Sidler D, Alonso G, et al. Consensus in a box: Inexpensive coordination in hardware [C] //Proc of the 13th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2016: 425-438

[9] Michalakes J, Vachharajani M. GPU acceleration of numerical weather prediction [J]. Parallel Processing Letters, 2008, 18(4): 531-548

[10] Choi Y, Cong J, Fang Zhenman, et al. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms [C] //Proc of the 53rd Annual Design Automation Conf. New York: ACM, 2016: 109-109

[11] Kotla R, Alvisi L, Dahlin M, et al. Zyzzyva: Speculative Byzantine fault tolerance [J]. ACM SIGOPS Operating Systems Review, 2007, 41(6): 45-58

[12] Liu Jian, Li Wenting, Karame G, et al. Scalable Byzantine consensus via hardware-assisted secret sharing [J]. IEEE Transactions on Computers, 2018, 68(1): 139-151

[13] Gueta G G, Abraham I, Grossman S, et al. SBFT: A scalable and decentralized trust infrastructure [C] //Proc of the 49th Annual IEEE/IFIP Int Conf on Dependable Systems and Networks (DSN). Piscataway, NJ: IEEE, 2019: 568-580

[14] Abraham I, Gueta G, Malkhi D, et al. Revisiting fast practical Byzantine fault tolerance [J]. arXiv preprint, arXiv: 1712.01367, 2017

[15] Wang Kai, Chen Fei, Cao Zheng, et al. HPP Controller: A system controller dedicated for message passing [C] //Proc of the 7th Int Conf on Parallel and Distributed Computing, Applications and Technologies. Piscataway, NJ: IEEE, 2010: 261-266

[16] Chen Fei, Cao Zheng, Wang Kai, et al. HPP controller: A system controller for high performance computing [J]. Frontiers of Computer Science in China, 2010, 4(4): 456-465

Yang Fan, born in 1992. PhD, engineer at the Institute of Computing Technology. His main research interests include in-network computing and high performance storage system.

杨 帆,1992 年生.博士,中国科学院计算技术研究所工程师.主要研究方向为在网计算和高性能存储系统.

Zhang Peng, born in 1993. Master at the Institute of Computing Technology. His main research interests include consensus algorithms and high performance networking.

张 鹏,1993 年生.中国科学院计算技术研究所硕士.主要研究方向为共识算法和高性能网络.

Wang Zhan, born in 1986. PhD, associate professor. Member of CCF. His main research interests include high performance computing, the interconnection network of high performance computer, and system virtualization.

王 展,1986 年生.博士,副研究员,CCF 会员.主要研究方向为高性能计算、高性能计算机互连网络和系统虚拟化等.

Yuan Guojun, born in 1983. PhD, associate professor, master supervisor. Member of CCF. His main research interests include computer architecture, high performance computing, optical network, distributed deep learning and system interconnection.

元国军,1983 年生.博士,副研究员,硕士生导师,CCF 会员.主要研究方向为计算机体系结构、高性能计算、光网络、分布式深度学习、系统互连等.

An Xuejun, born in 1966. PhD, professor, PhD supervisor. Member of CCF. His main research interests include HPC architecture, HPC network, graph computing accelerator, SoC and system interconnection.

安学军,1966 年生.博士,正高级工程师,博士生导师,CCF 会员.主要研究方向为高性能计算机体系结构、高性能计算网络、图计算加速器、片上系统、系统互连等.