การจัดตารางหน่วยประมวลผลกลางและการจัดการความกว้างช่องสัญญาณสำหรับงานคำนวนแบบอาสาสมัคร

นายกรกฤต สีมาคุปต์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2558
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

Nicer CPU Scheduling and Bandwidth Management for Volunteer Computing

Mr. Korakit Seemakhupt

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering Program in Computer Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2015

Thesis Title	Nicer CPU Scheduling and Bandwidth Management for Volunteer Computing
By	Mr. Korakit Seemakhupt
Field of Study	Computer Engineering
Thesis Advisor	Assistant Professor Krerk Piromsopa, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

....Dean of the Faculty of Engineering

(Associate Professor Supot Teachavorasinskun, Ph.D.)

THESIS COMMITTEE

....Chairman

(Assistant Professor Natawut Nupairoj, Ph.D.)

....Thesis Advisor

(Assistant Professor Krerk Piromsopa, Ph.D.)

....Examiner

(Assistant Professor Veera Muangsin, Ph.D.)

....External Examiner

(Pongtawat Chippimolchai, Ph.D.)

กรกฤต สีมาคุปต์ : การจัดตารางหน่วยประมวลผลกลางและการจัดการความกว้างช่องสัญญาณสำหรับงานคำนวนแบบอาสาสมัคร (Nicer CPU Scheduling and Bandwidth Management for Volunteer Computing) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: ผศ. ดร. เกริก ภิรมย์โสภา, 35 หน้า.

ในงานวิจัยนี้ เราเสนอวิธีการลดปัญหาการเสียประสิทธิภาพของระบบคอมพิวเตอร์ขณะทำงานประมวลผลอาสาสมัคร ในส่วนแรก เราแก้ปัญหาการแบ่งเวลาบนระบบที่ใช้การจัดตารางแบบ CFS เราใช้หลักการการจองเวลาแบบปรับตัวได้ในการรักษาเวลาทำงานของหน่วยประมวลผลกลางสำหรับงานหลัก เราแก้ปัญหาของการแบ่งกั้นระหว่างผู้ใช้งานด้วยการใช้สถิติรวมทั้งระบบ วิธีของเราสามารถรักษาไว้ซึ่งประสิทธิภาพของงานหลัก และสามารถดึงเอาเวลาของหน่วยประมวลผลกลางที่สูญเสียไปคืนมาได้มากกว่าวิธีการจองเวลาแบบคงตัว ในส่วนที่สอง เราศึกษาโปรโตคอล HTTP/2 โดยโปรโตคอล HTTP/1.1 ที่ใช้อยู่ก่อนหน้า แก้ปัญหาการใช้ความกว้างช่องสัญญาณด้วยการเปิดหลายๆการต่อเชื่อมพร้อมๆกัน แต่นั่นสามารถทำให้เกินการแย่งชิงความกว้างช่องสัญญาณได้ หากช่องสัญญาณมีจำนวนจำกัด ในงานนี้เราศึกษาการใช้งาน Multiplexed streams ในโปรโตคอล HTTP/2 จากผลการทดลองของเรา Multiplexed stream สามารถถูกใช้ทดแทนการเปิดหลายๆการเชื่อมต่อได้ และยังไม่แย่งความกว้างช่องสัญญาณกับการเชื่อมต่ออื่นๆ

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต	....
สาขาวิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อ อ.ที่ปรึกษาหลัก	....
ปีการศึกษา	2558		

5770106421 : MAJOR COMPUTER ENGINEERING
KEYWORDS:

KORAKIT SEEMAKHUPT: Nicer CPU Scheduling and Bandwidth Management for Volunteer Computing. ADVISOR: ASST. PROF. KRERK PIROMSOPA, Ph.D., 35 pp.

We proposed methods dealing with performance reduction problem when running volunteer computing application. First, we deal with CPU time allocation problem on CFS-based system. We use adaptive reservation dealing with maintaining CPU time for foreground process. We solved problem of user boundary using system's global statistics. Our method can maintain foreground application's performance and can reclaim more CPU time compared to that of static allocation method. In the second part, we study new HTTP/2. The previous HTTP/1.1 solves problem of unutilized bandwidth by opening multiple connection. However, this can cause network bandwidth contention when bandwidth is limited. In this work, we focus our study on the uses of multiplexed streams. Our result shows that the multiplexed streams can replace multiple seperate connections and is more network friendly to other applications.

Department:	Computer Engineering	Student's Signature	....
Field of Study:	Computer Engineering	Advisor's Signature	....
Academic Year:	2015		

ACKNOWLEDGEMENTS

This work is supported by Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University.

I would also like to thank my thesis advisor, Dr. Krerk Piromsopa and thesis committee, Dr. Natawut Nupairoj, Dr. Veera Muangsin and Dr. Pongtawat Chippimolchai for their input for improving of my work.

CONTENTS

Page

THAI ABSTRACT iv

ENGLISH ABSTRACT v

ACKNOWLEDGEMENTS vi

CONTENTS vii

List of Figures ix

List of Tables x

1. Introduction 1

1.1 Objectives 1

1.2 Scope 2

1.3 Organization of the Dissertation 2

2. Nicer Processing 3

2.1 Background & Related works 3

2.1.1 The Completely Fair Scheduler(CFS) 3

2.1.2 Analysis and Problems of the CFS 4

2.2 Our solution 8

2.2.1 Design concept 8

2.2.2 How to detect Starvation 9

2.3 Implementation and Experiments 11

2.3.1 Implementation 11

2.3.2 The experiments 11

2.3.2.1 Experiment 1 12

2.3.2.2 Experiment 2 16

Page

2.4 Section 2 Conclusion 17

3. Nicer Protocol 18

3.1 Background and Related works 18

3.1.1 The Hypertext Transfer Protocol Version 2(HTTP/2) 18

3.1.2 Previous works regarding HTTP/2 performance 20

3.1.3 Previous works regarding SPDY protocol performance 21

3.1.4 Analysis 21

3.2 Experiment I : Stream-Connection equivalence 22

3.2.1 Experiment I Test setup 22

3.2.2 Experiment I Test result 23

3.3 Experiment II : HTTP/2, HTTP/1.1 co-running 27

3.3.1 Experiment II Test setup 27

3.3.2 Experiment II Test result 28

3.4 Section 3 Conclusion 30

4. Conclusion 31

REFERENCES 32

VITA 35

List of Figures

Figure 1: Volunteer Computing Architecture 1

Figure 2: Example of CPU-time allocation of SCHED_NORMAL 5

Figure 3: Architectural Design 8

Figure 4: Starvation Detection Algorithm 9

Figure 5: Starvation Detection Flowchart 10

Figure 6: Experiment I test1, LINPACK Benchmark 13

Figure 7: Experiment I test2, Apache Bench 14

Figure 8: Experiment I test3, H.264 Video Playback 15

Figure 9: Experiment 2, LINPACK background with H.264 running as a foreground 16

Figure 10: HTTP/1.1 Head of line blocking 19

Figure 11: HTTP/2 Multiplexed Streams 20

Figure 12: Experiment I test setup 22

Figure 13: Experiment I Payload transfer rate, 32 bytes payload 23

Figure 14: Experiment I Payload transfer rate, 1KiB payload 24

Figure 15: Experiment I payload transfer rate, 32 bytes payload, 1% loss 25

Figure 16: Experiment I payload transfer rate, 1KiB payload, 1% loss 26

Figure 17: Experiment II test setup 27

Figure 18: Experiment II Ratio of HTTP/2 and HTTP/1 bandwidth, 32 connections 28

Figure 19: Experiment II Ratio of HTTP/2 and HTTP/1 bandwidth, 32 inflight requests 29

List of Tables

Table 1: Comparision of Current Solutions 7

Table 2: Summary of Experiment I tests... 12

1. Introduction

Volunteer computing uses computers volunteered by general public to do distributed scientific computing[1]. This allows research projects to access large pool of computing power without investing in expensive computing infrastructure and operating cost. But, one drawback of joining a volunteer computing project is the decrease of performance in participating machines particularly processor performance. However, other resources such as memory and network bandwidth also play important roles in system performance. Our goal here is to avoid starvation of resources caused by volunteer computing application. In another word, we want to make volunteer computing application "nicer" to other processes in the system. However, as different resource type has different characteristics and constrain, thus requires different method preventing starvation. Figure 1 shows architecture of volunteer computing application.

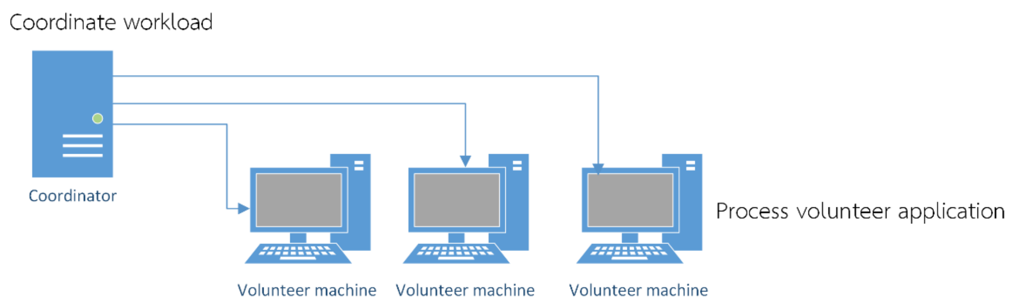

Figure 1: Volunteer Computing Architecture

1.1 Objectives

1. The system should be able to mitigate foreground task's performance loss.
2. The system should be able to work without direct knowledge of an application or without an elevated privilege.
3. The system could be implemented without modification of an application and minimal modification to the operating system.

1.2 Scope

1. The system only limits resources usage of the background processes, does not directly change scheduling policy of the operating system.
2. In this project we only consider applications, both with constant and variable resources usage
3. In this project we only consider starvation caused by background processes and we assume that application's resources requirements don't exceed system capacity.

1.3 Organization of the Dissertation

This dissertation is organized as follows. In section 2, “Nicer Processing”, we developed a CPU allocation technique that can prevent CPU starvation from Background process usage. In section 3, “Nicer Protocol”, we investigated an upcoming transport layer protocol, the HTTP/2 and its effect on network contention due to multiple connection. We also include preliminary study on effects of memory performance from Background process in the appendix. We concluded our work in Section 4. We also include some technical background in the APPENDIX.

2. Nicer Processing

In this section, we developed a CPU allocation technique to solve the problem of running volunteer application on a system with the Completely Fair Scheduler(CFS). Our allocation technique is based on adaptive reservation but works without requiring administrative privilege and can work across user boundaries. From the experiment, our method can maintain performance of foreground workload and can reclaim more CPU time compared to static allocation method.

2.1 Background & Related works

2.1.1 The Completely Fair Scheduler(CFS)

Prior to the Linux kernel version 2.6.24, scheduler contains two scheduling classes, SCHED_RT and SCHED_NORMAL [2]. Tasks in SCHED_RT always run before tasks in another class. The rest belongs to SCHED_NORMAL class. Scheduler allocates CPU time according to nice value of each task. After kernel v2.6.24, the completely fair scheduler (CFS) was merged into the mainline kernel, replacing the existing SCHED_NORMAL scheduling class. The CFS is in a class of scheduler called Fair Share Scheduler[2]. The Fair Share Scheduler allocates CPU time proportional to number of share. The new design delivered weight-based scheduling of CPU bandwidth, enabling arbitrary partitioning. This allowed support for group scheduling to be added and managed using cgroups[3]. These are some features supported by the CFS.

- Group Scheduling: In early version, the CFS only implements fairness between processes[4]. In order to deal with multi process resource hogging task, processes are grouped according to TTY [5] (Automatic process grouping) or Session ID [6] in later version.
- Control Groups: Control groups allow resources to be allocated among user-defined groups of tasks. Control groups are also hierarchical. For example, CPU

time is allocated according to ratio of share in each level in a top-down fashion[7].

In depth details about the CFS can be found in [8].

2.1.2 Analysis and Problems of the CFS

Prior to the Linux kernel version 2.6.24, the nice value of a task was used as one of the tunable parameter in CPU time allocation. The nice value is then used to calculate the priority of the task. The priority was compared to one belonging to all other processes in the system and was used to determine CPU time for each processes. Non-privileged user can only monotonically drop his/her nice value to give up CPU time to another user.

Since v2.6.24, the concept of Fair Share Scheduling was introduced with the CFS, nice value is only used as a scheduling parameter in the same group level or in the same TTY. This is done in order to enforce fairness among users and “improve interactivity”. The problem is, while in default policy, non-privileged user cannot decrease his/her nice value or increase shares, user also cannot change his/her shares relative to other users. The user can only change share between process groups under his/her own group. For example, in *Figure 2*, User B changes process group B1 to the minimum share of 1. Since CPU-time allocation is top-down, process group B1 receive CPU-time as much as process group A1 and A2 combined. Thus, increasing in the number of processes or process groups of a user does not translate into more CPU-time allocation.

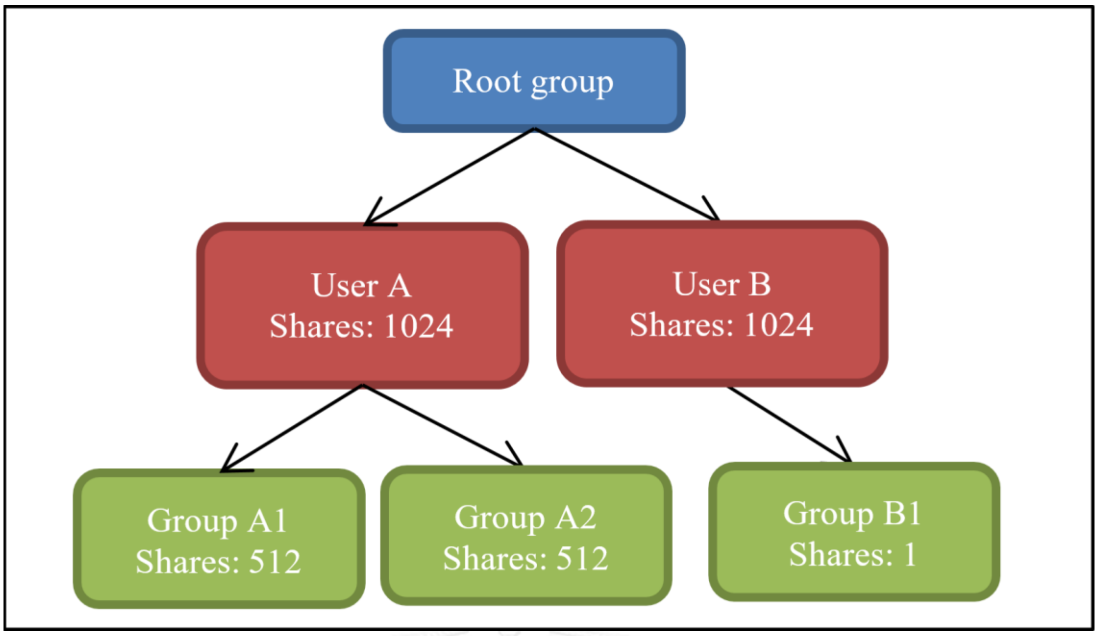

Figure 2: Example of CPU-time allocation of SCHED_NORMAL

2.1.3 Current Solutions

There are many solutions to alleviate the problem of background volunteering computing processes spending too much CPU time.

1. Do nothing

   All users have same amount of CPU share, since a user cannot set his/her share. On the other hand, CPU time available to a user can be as low as 1/number of users.

2. Suspend background processes

   The controller of background (volunteering computing) processes monitors CPU time used by other applications. If other processes use CPU time more than a predetermined threshold, background processes will be suspended. This approach, while provides minimal performance impact to the foreground processes, leaves all remaining CPU time unused. This is the default approach used by the BOINC Client on Linux based operating system[9].

3. CPU Time Reservation

   By reserving a CPU time for maximum foreground processes (with timing constrain) utilization, service quality could be guaranteed at a cost of wasted CPU time. This technique has long been studied in [10] since system's scheduler was primitive. In the CFS scheduler, the execution cap could be controlled through cgroups interface.

4. Adaptive Reservation

   By using service quality of foreground process as an input of a controller, CPU time allocation of foreground processes could be controlled to optimize service quality while minimizing wasted CPU time. The implementation of Adaptive Reservation in Linux was purposed in[11]. However, if foreground and background applications are from different users, this method breaches user policy of isolation.

Table 1 shows the comparison of current solutions.

Table 1: Comparision of Current Solutions

Approach	Service quality	CPU Fraction of foreground process	Wasted CPU time	Breach user policy?
Do nothing	No guarantee	1/Number of scheduling group in the same level	No	No
Suspend background process	Best	1	Very High	No
Static execution cap	Very good, if correctly predicted	Predetermined	High	No
Adaptive Reservation	Acceptable	Changed in runtime	Low	Yes
Purposed Solution	Acceptable	Changed in runtime	Low	No

2.2 Our solution

2.2.1 Design concept

Our solution is based on an idea of feedback adaptive reservation. To ease explaining, Figure 3 shows architectural design of our solution. Since default user privilege policy does not allow feedback signal to be sent to another user, we use global monitoring tools to get amount of CPU time spend idling to detect CPU time starvation in another user's process (foreground process). If a process is in starvation, it should use more available CPU time. If background process, however, could deplete all CPU time available, there will be no CPU time left for starvation detection. Our solution is to limit a background process so that there will always be CPU time left similar to PI controller. Eventually, other processes will consume reserved CPU time until meeting its requirement. (Unless the system is not overloaded by other processes.) In other words, trying to maintain idle CPU time serves 2 purposes: 1. background processes are "nicer" to other processes 2. absence of idle CPU time indicates CPU starvation.

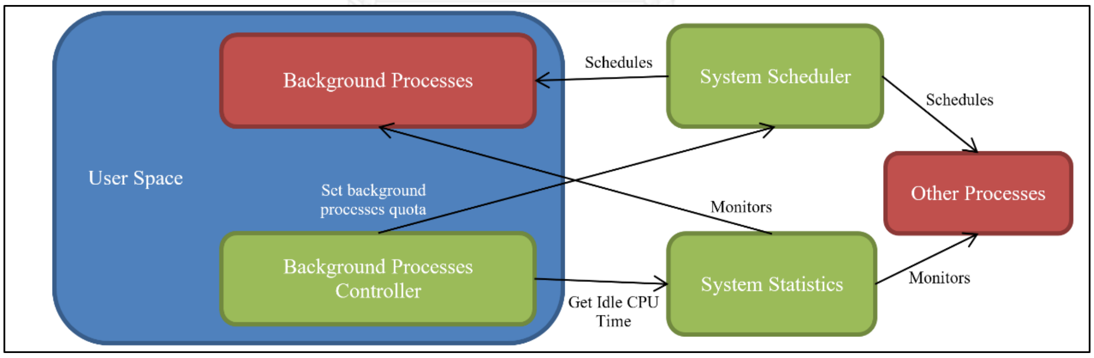

Figure 3: Architectural Design

2.2.2 How to detect Starvation

Figure 4 shows the Starvation Detection Algorithm.

```
initialize reserveBandMultiplier to minimumReserveBandMultiplier
initialize overloadCount  to  0
set backgroundQuotaFraction to 0.0
forever do
        get idleCPUFraction
        if idleCPUFraction = 0 then
                increment overloadCount
                double reserveBandMultiplier but no more than
maxReserveBandMultiplier
        else
                reset overloadCount to 0
                decrement reserveBandMultiplier but no less than
minimumReserveBandMultiplier
        end if
        if overloadCount > overloadThreshold then
                reset overloadCount to 0
                reset backgroundQuotaFraction to 0.0;
                set reserveBandMultiplier to maxReserveBandMultiplier
        else
                set backgroundQuotaFraction to backgroundQuotaFraction +
(idleCPUFraction - (reserveBandMultiplier*minimumReserveBand))
        end if
        wait until next scheduling round
end
```

Figure 4: Starvation Detection Algorithm

Our Starvation Detection Algorithm can be separated into 3 parts. First part tries to maintain small CPU idle time(reserveBand) to detect and prevent starvation. Second part manages with fluctuating workloads by detecting overloading and multiplicatively increase reserve band. However, if workload is constant, reserveBand decay linearly to the minimum level. The last part deals with instantaneous workload. When detecting multiple consecutive overloading, background process CPU time is reduced to the minimum. *Figure 5* shows a Flowchart of second and last part of the algorithm.

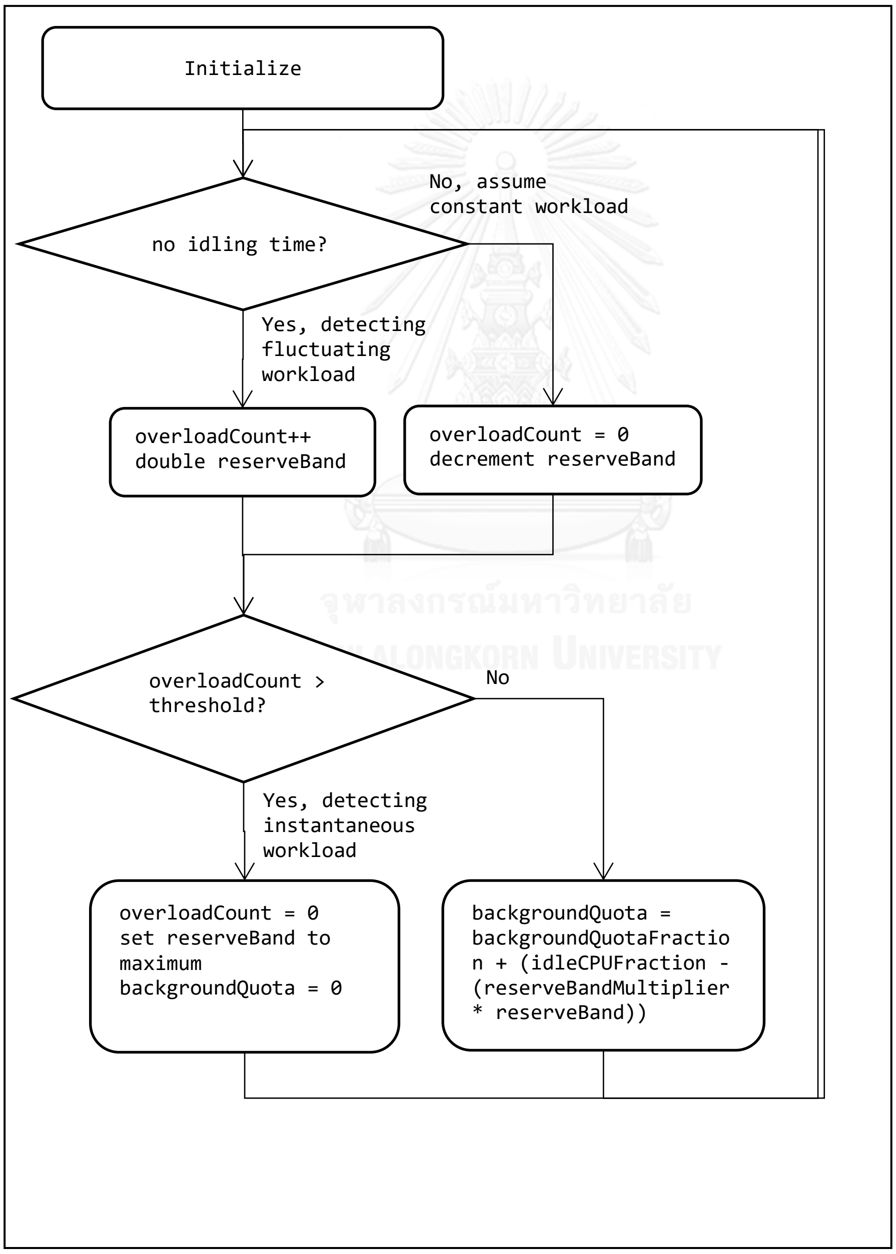
Initialize
no idling time?
No, assume constant workload
Yes, detecting fluctuating workload
overloadCount++ double reserveBand
overloadCount = 0 decrement reserveBand
overloadCount > threshold?
No
Yes, detecting instantaneous workload
overloadCount = 0 set reserveBand to maximum backgroundQuota = 0
backgroundQuota = backgroundQuotaFraction + (idleCPUFraction - (reserveBandMultiplier * reserveBand))

Figure 5: Starvation Detection Flowchart

2.3 Implementation and Experiments

2.3.1 Implementation

We implemented our algorithm on Linux based operating systems. Our implementation does not require any modification to kernel or administration privilege except for enabling cgroups file-system interface. The idleCPUFraction was extracted from /proc/stat. The backgroundQuotaFraction was set through file-system interface of cgroups. We conducted experiments on Ubuntu Desktop 14.04 with cgroup installed.

2.3.2 The experiments

We evaluated our implementation by using two sets of experiments. The goal of the first experiment is to evaluate performance loss of foreground workload. The second experiment measured the performance of background workload in order to evaluate allocation efficiency of our allocation method against that of static allocation method.

Parameters used in Starvation Detection Algorithm in this experiment are

- minimumReserveBandMultiplier = 1
- maxReserveBandMultiplier = 16
- overloadThreshold = 4

We acquired parameters by hand-tuning and can be different according to resources in the system.

2.3.2.1 Experiment 1

In this experiment, we measured the performance of foreground processes while running CPU hogging processes as a background. The background workload could fully utilize both cores of the processor. Foreground processes and background processes were grouped with the same type. Background processes group is controlled by our allocation technique. We compared the result against a run without background processes CPU allocation control (w/o control) and a run without background processes running (no). In this experiment, we use multiple instances of “md5sum /dev/zero” as background workload. *Table 2* shows summary of experiment 1 tests.

Table 2: Summary of Experiment I tests

Test Case	Workload Description	Foreground Workload
1	Compute-heavy	LINPACK Benchmark
2	Latency-sensitive	Apache Bench
3	Soft real-time	H.264 Video Playback

Test 1: LINPACK Benchmark

LINPACK is a library for solving linear algebra. LINPACK benchmark is used to measure floating point performance of the computer. This test represents compute-heavy workloads without timing constrain. Since foreground workload is easily predictable, we can see in Figure 6 that background workload has no significant impact on foreground process.

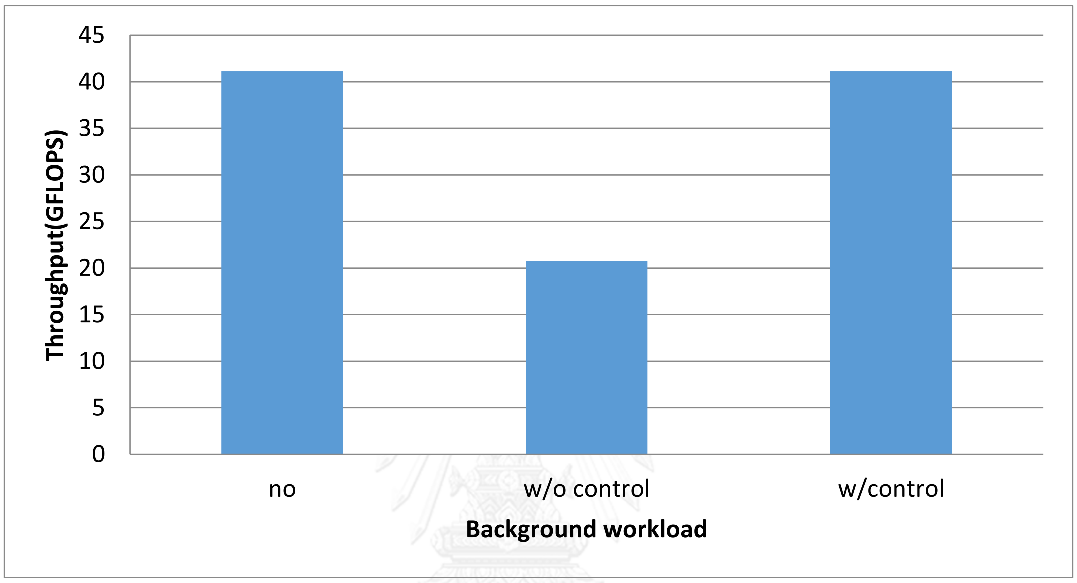

Figure 6: Experiment I test1, LINPACK Benchmark

Test 2: Apache Bench

Apache is a widely used web server. The Apache Bench measures how many requests per second a given system can sustain when carrying out 1,000,000 requests with 100 requests being carried out concurrently [10]. While intensity of foreground workload is not high, it is highly sensitive to service latency. In Figure 7 we can see that even if background workload is controlled with our allocation technique, the performance of web server in this test could not be maintained. However, if we change parameter “minimumReserveBandMultiplier” to 8 (minBand>=8), performance degradation becomes acceptable. This is due to limited parallelism of 100 concurrent requests.

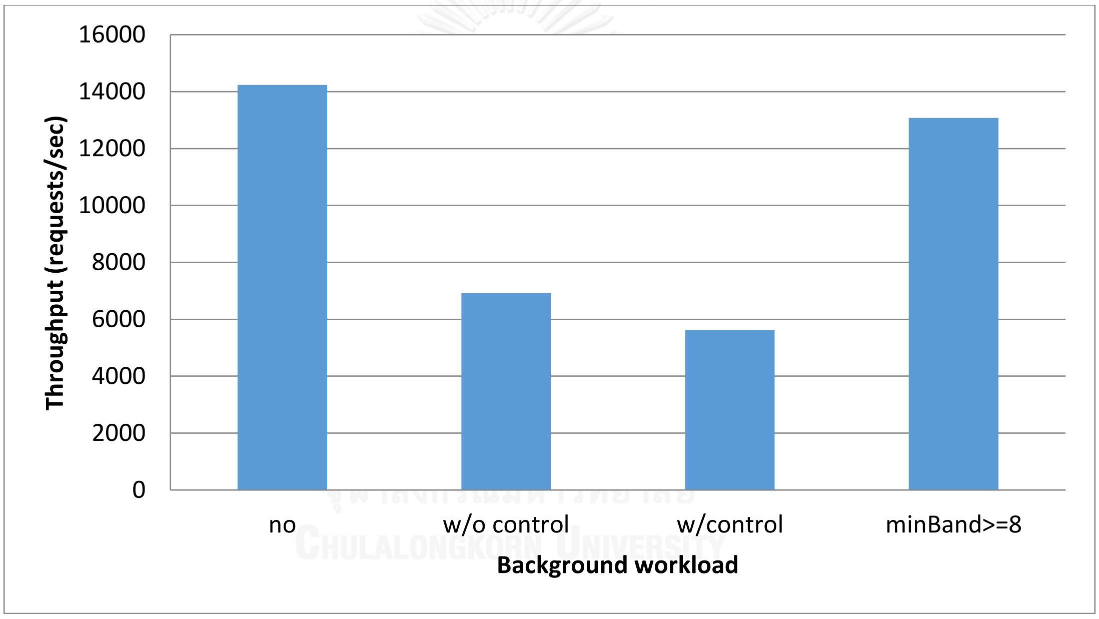

Figure 7: Experiment I test2, Apache Bench

Test 3: H.264 video playback

In this test, we use VLC player 2.1.4 to play a variable bit-rate H.264 video of 32 second length with average bit-rate of 70Mbit/s. Build-in FFmpeg decoder is used and adaptive decoding (Hurry-up option) is disabled. This test represents continuous soft real-time application with variable processor requirement. Since this foreground processes in this test cannot use half of total share in our test environment, we disable one of the processor core to decrease processing power available to the system. In Figure 8, we can see that, with our allocation technique, performance penalty of running CPU hogging processes is minimal.

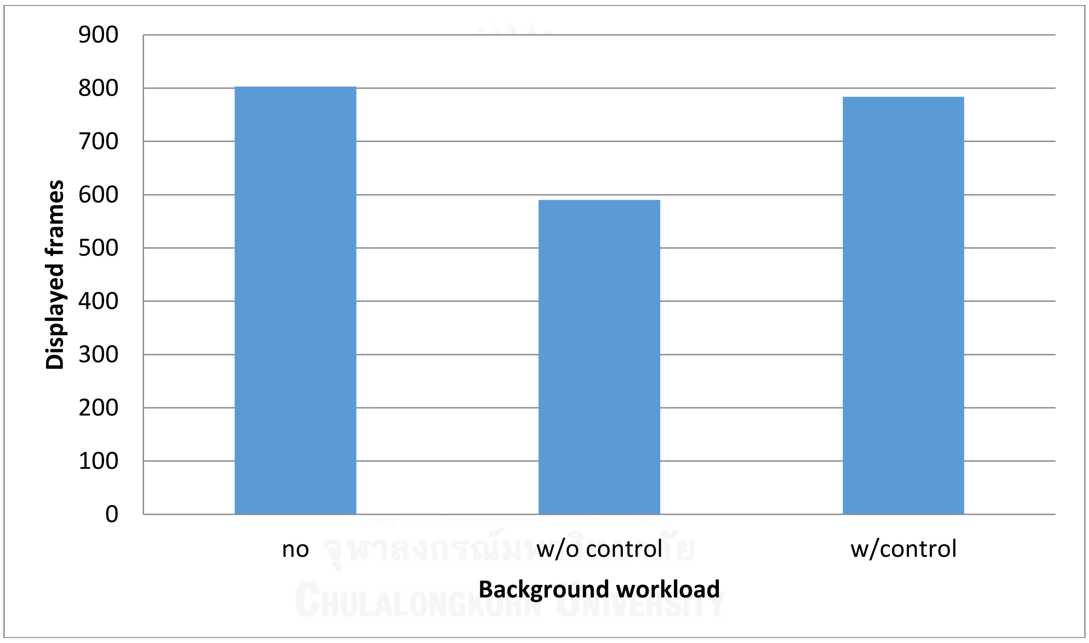

Figure 8: Experiment I test3, H.264 Video Playback

2.3.2.2 Experiment 2

In this experiment, we used VLC player to play H.264 video with average bit-rate of 50 Mbit/s as a foreground workload and LINPACK benchmark as a background workload. The background workload was controlled using worst case static allocation method (static) and our allocation method (dynamic). We measured performance of background workload to compare the performance loss of each allocation method. The comparison is shown in *Figure 9*. We can see that our allocation can reclaim more performance compared to the static allocation method. Also, suspending method used by BOINC client in Linux-based system does not background workload of foreground workload reach certain threshold.

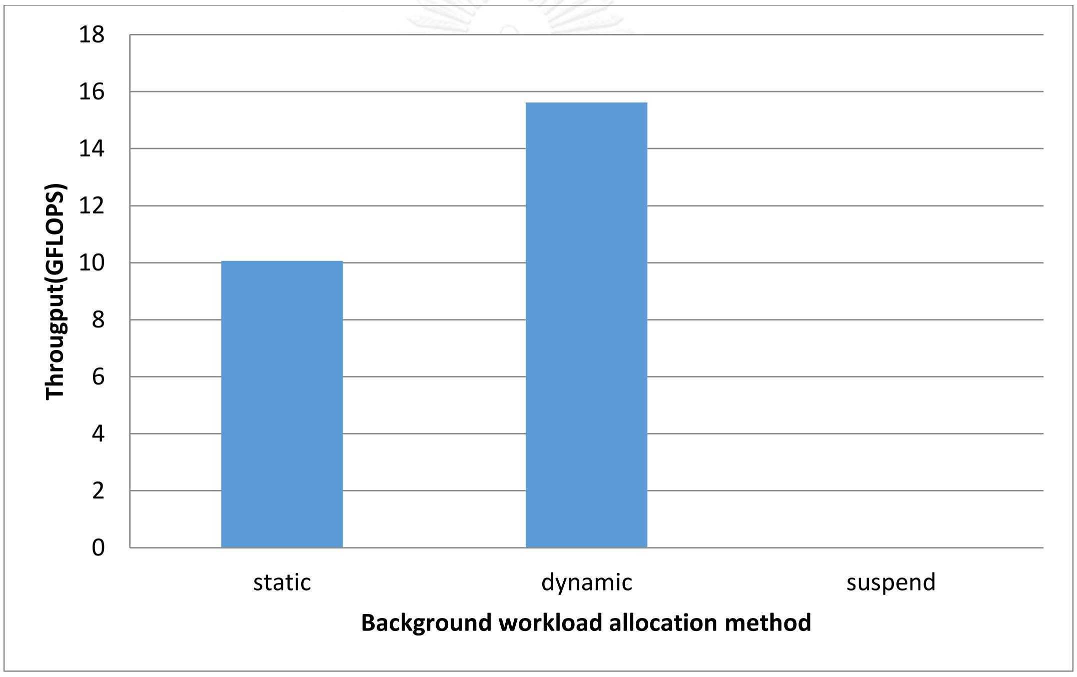

Figure 9: Experiment 2, LINPACK background with H.264 running as a foreground

2.4 Section 2 Conclusion

Our CPU allocation technique for volunteer computing application based on Adaptive Reservation can solve the problem of performance degradation when running a volunteer application as a background process on a system with Completely Fair Scheduler. We solved the problem of feedback signal across user boundary by replacing it with the idle CPU statistics from system monitoring tools. The idle CPU statistics is used for our Starvation Detection Algorithm to both keep track of CPU usage requirement and to control background processes to be nice to other processes in the system. The implementation was done on a Linux-based system with CFS. The impact was evaluated with several types of foreground applications. With compute-heavy workload without timing constrain as a foreground workload, performance degradation from background workload is negligible in our allocation technique. In low intensity, latency sensitive web server test, tuning of parameters is necessary to maintain good performance. While in video playback test, performance drop is acceptable. In another experiment, we found that our allocation method improves background workload CPU idle time. It can reclaim 50 percent CPU time without the loss of foreground performance.

3. Nicer Protocol

As bandwidth of a link continues to increase, link latency is constrained by physical limits. So called “latency-bandwidth product” becomes too large to be easily utilized. This is especially true in current mobile networking technology such as LTE where link speed could exceed 1 Gbit/s but with Round-Trip-Time as high as 30 ms[12]. Also, the size of content being delivered is growing. In February 2016, average size of a web page is around 2200 kB and is delivered over the average of 100 requests [13]. Current application-layer protocol, HTTP/1.1, is not able to benefit from this situation. Therefore, the HTTP/2 protocol was created. From our survey, HTTP/2’s benefits over its predecessor is studied in several literatures but is far from covering all aspects. In this section, we investigated factors that impact HTTP/2’s performance compared to that of the HTTP/1.1. We focused our study on the effects of the multiplexed stream.

3.1 Background and Related works

3.1.1 The Hypertext Transfer Protocol Version 2(HTTP/2)

The HTTP/2 was proposed to mitigate problems of the previous HTTP/1.1 and its underlying transport protocols. In the original HTTP/1.0, each connection can serve only one request. Persistence connection was later added but not included in HTTP/1.0 standard. Persistence connection allows reuse of connection by multiple requests, therefore reducing overhead of creating new TCP connection, but still limited to single simultaneous request. In HTTP/1.1, Pipelining was added and Persistence connection became standard. Pipelining allows multiple inflight requests over single connection. However, requests Pipelining in HTTP/1.1 needs to be served in order. This creates head-of-line blocking problem as illustrated in *Figure 10* and *Figure 11*. Therefore, it was not widely implemented by popular web browsers [14, 15]. Common workaround is to interleave requests over multiple TCP connections simultaneously. However, creating multiple connections have several drawbacks including TCP slow start and may pollute other traffics on the network. Many application-layer protocols were proposed as a supplemental or an alternative to the HTTP/1.1. One of them is Google's SPDY [16] protocol which was later used as a basis for HTTP/2. In May 2015, Internet Engineering Task Force(IETF) proposed RFC7540 as a standard that defines HTTP/2. The HTTP/2 solves these issues by 1) allowing multiplexing of multiple requests and response messages over single connection, 2) using compression for HTTP header fields to save header space, 3) changing from textual to binary message framing to simplify parsing and 4) server push to hide request latency [17].

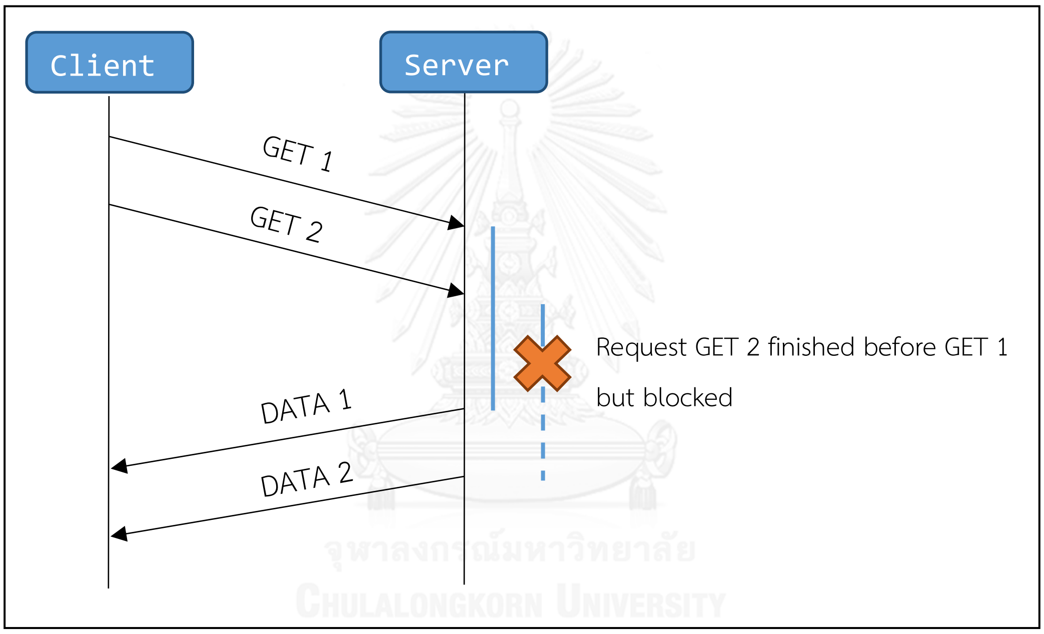

Figure 10: HTTP/1.1 Head of line blocking

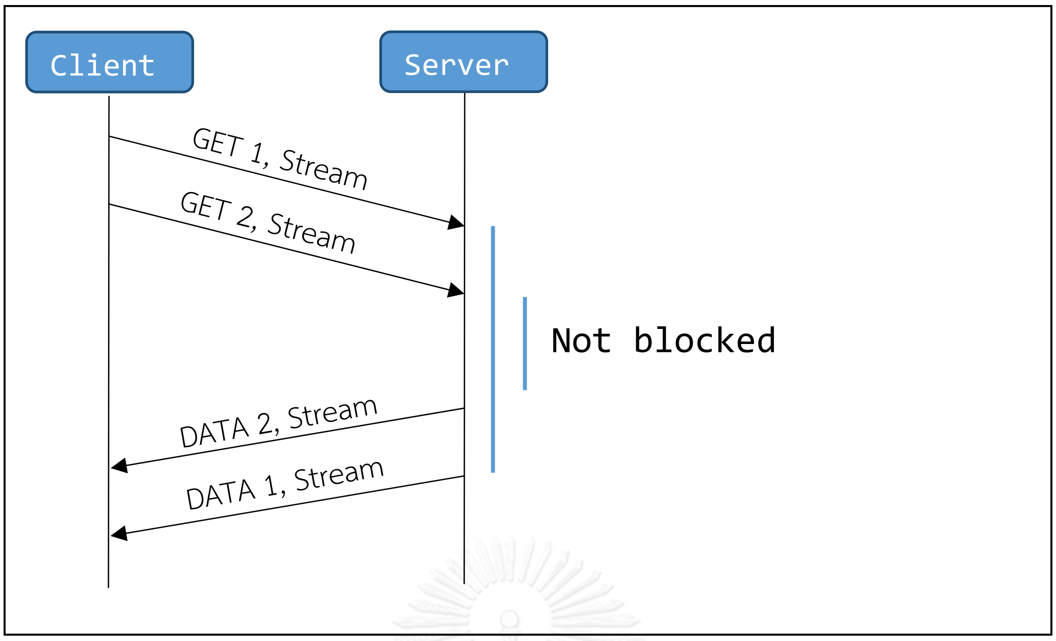

Figure 11: HTTP/2 Multiplexed Streams

3.1.2 Previous works regarding HTTP/2 performance

There are several works related to HTTP/2's performance as follows:

- Is HTTP/2 Really Faster Than HTTP/1.1?[18]

  This work studied the impact of number of requests, link latency and packet loss on Page Load Time. It concludes that HTTP/2 has significantly faster page load time for websites with great number of requests. However, it is less advatageous with the presence of packet loss compared to the HTTP/1.1.

- To HTTP/2, or Not To HTTP/2, That Is The Question[19]

  This work studied the implementation of HTTP/2 in several websites and also Page Load Time under various network condition. It concludes that HTTP/1.1 suffers more penalty in Page Load Time than HTTP/2 under high latency or with packet loss. Also, HTTP/2's penalty can be further reduced by domain sharding.

- The upcoming new standard HTTP/2 and its impact on multi-domain websites[20]

  This work compares the Page Load Time of several websites with both HTTP/1.1 and HTTP/2 on wired and wireless network. The study concludes that HTTP/2 may cause slowdown on site that contains small number of resouces in some cases.

3.1.3 Previous works regarding SPDY protocol performance

In addition to HTTP/2, there exist works related to SPDY's performance, which is a proprietary protocol designed by Google to solve similar issues. We include works regarding SPDY because it was better studied. The SPDY protocol also has multiplexed stream which is our interest here.

- Can SPDY Really Make the Web Faster?[21]
  The work studied the effects of latency, packet loss, shading and multiplexed stream on Time on Wire(measured on the network interface) in both real sites and controlled environment. They found that penalty and benefits of SPDY's multiplexed streams can vary from site to site. For example, Twitter site with less resources count gains less reduction in time of wait with SPDY compared to that of HTTPS in YouTube site with high resources count.
- How Speedy is SPDY?[22]
  This work studied the effects of link bandwidth, latency, number and size of object in the webpage on Page Load Time and comparison to the HTTP/1.1. They conclude that the benefits of SPDY multiplexed stream is limited by dependencies in webpages.

3.1.4 Analysis

From our survey, one thing in common from those studies is the use of sampled web page as a test payload and the use of Page Load Time as performance metric. However, we want to study sustain performance of the HTTP/2 that is important in some applications such as streaming and effects of latency, packet loss compared to that of the HTTP/1.1. We also want to study HTTP/2 performance under highly congested network.

3.2 Experiment I : Stream-Connection equivalence

In this experiment, we measure payload transfer rate while varying number of inflight requests and payload size. We want to prove that HTTP/2.0's stream is equivalent to multiple separate connection when the network is not under bandwidth limited condition. We also want to see whether using multiplexed stream helps to mitigate effect of lossy connection.

3.2.1 Experiment I Test setup

In this test, we setup 2 machines, one as a server, another as a client. The machines were connected together through an Ethernet link with latency of 5ms and bandwidth of 100Mbit/s. We used Apache 2.4.18[23] as a web server, h2load [24] as an HTTP/2 client and ApacheBench as HTTP/1.1 client. *Figure 12* shows test setup of this experiment. We enabled keep-alive in HTTP/1.1 but not pipelining because it not was not widely implemented in popular browsers. We varied number of inflight requests from 10 to 100. In HTTP/2 case, we fixed number of streams in each connection to 1 and 10. We used the number of streams multiplied by the number of connection as a number of inflight requests. The payload size we used in this experiment is 32 B and 1 KB. Round trip time between server and client was 5 milliseconds. We tested with and without packet loss rate of 1%. We compared Payload Transfer Rate from each type of connection. We have verified that both Web Server and Client are not CPU-bounded in this experiment.

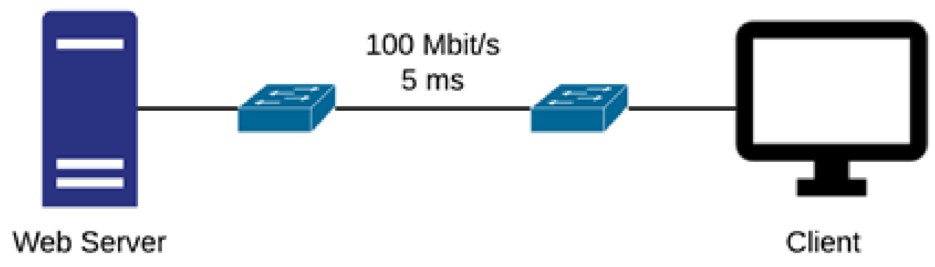

Figure 12: Experiment I test setup

3.2.2 Experiment I Test result

Without packet loss, we can see from *Figure 13* that with the same number of inflight request, HTTP/2's and HTTP/1.1's payload transfer rates are roughly equal for both transfer size when there is less than 50 inflight requests. However, due to increased overhead of HTTP/2 protocol, there is small slowdown with more connection compared to the HTTP/1.1.

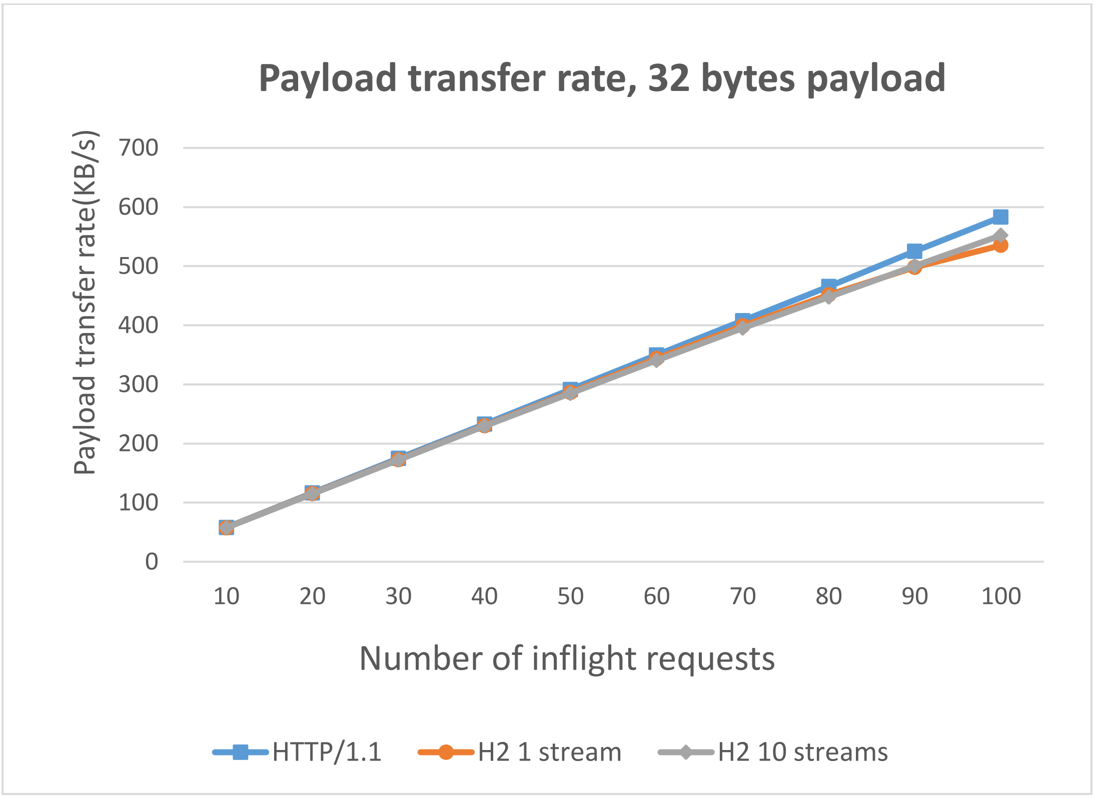

Figure 13: Experiment I Payload transfer rate, 32 bytes payload

However, as shown in *Figure 14*, HTTP/2's peak payload transfer rate is higher in 1KiB transfer size because our bandwidth limit is hit and there is less header space overhead from compressed header.

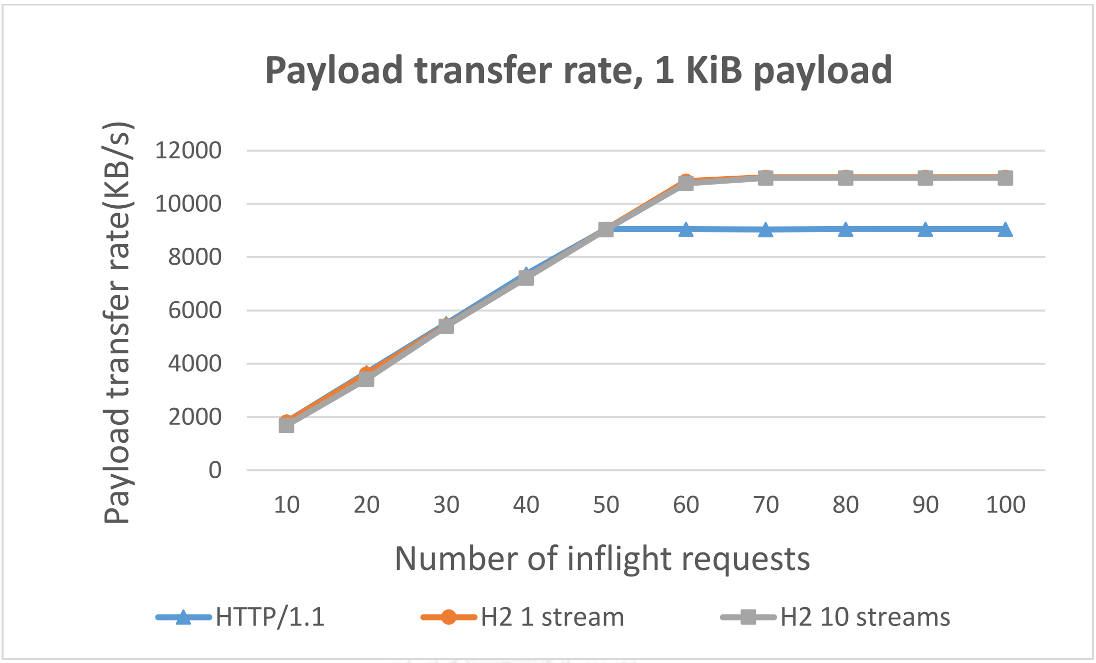

Figure 14: Experiment I Payload transfer rate, 1KiB payload

With packet loss, in 32 bytes payload case, throughput of 10-stream connection grows significantly faster than HTTP/1.1 and HTTP/2 with single stream. This is due to faster TCP retransmission. 1-stream HTTP/2 connections also suffer more penalty from loss than HTTP/1.1 as shown in *Figure 15*. But from our experiment, the improvement is significant even with only 2 multiplexed streams.

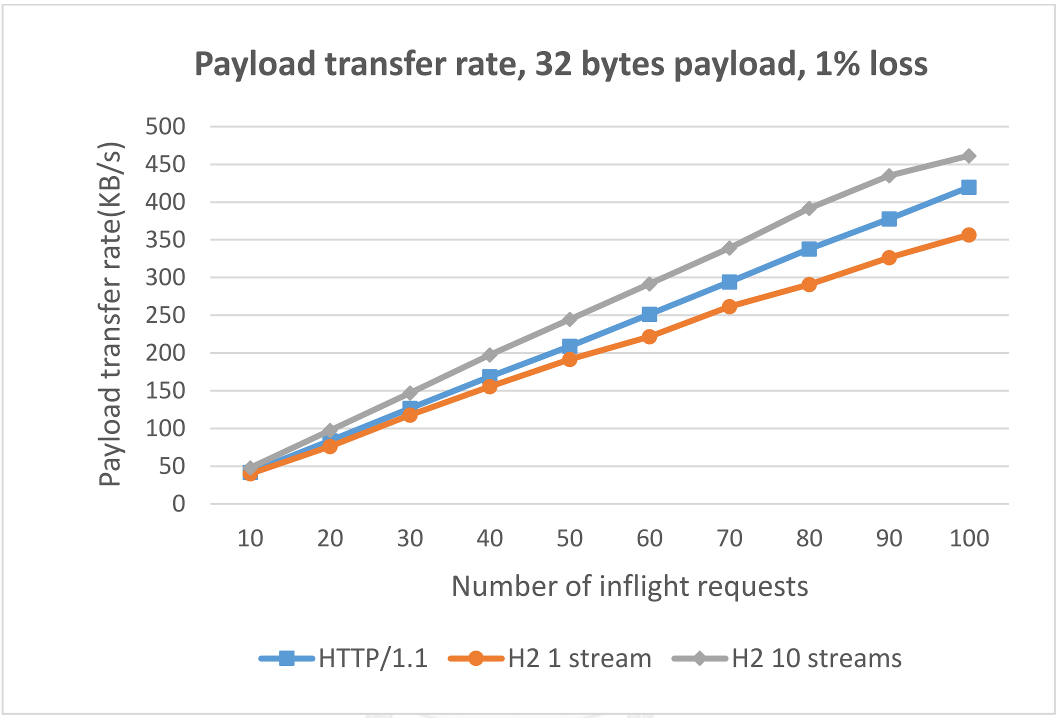

Figure 15: Experiment I payload transfer rate, 32 bytes payload, 1% loss

The result in 1 KiB payload is similar but HTTP/2 with single stream achieves higher peak transfer rate than HTTP/1.1 due to the header compression. The result was shown in *Figure 16*.

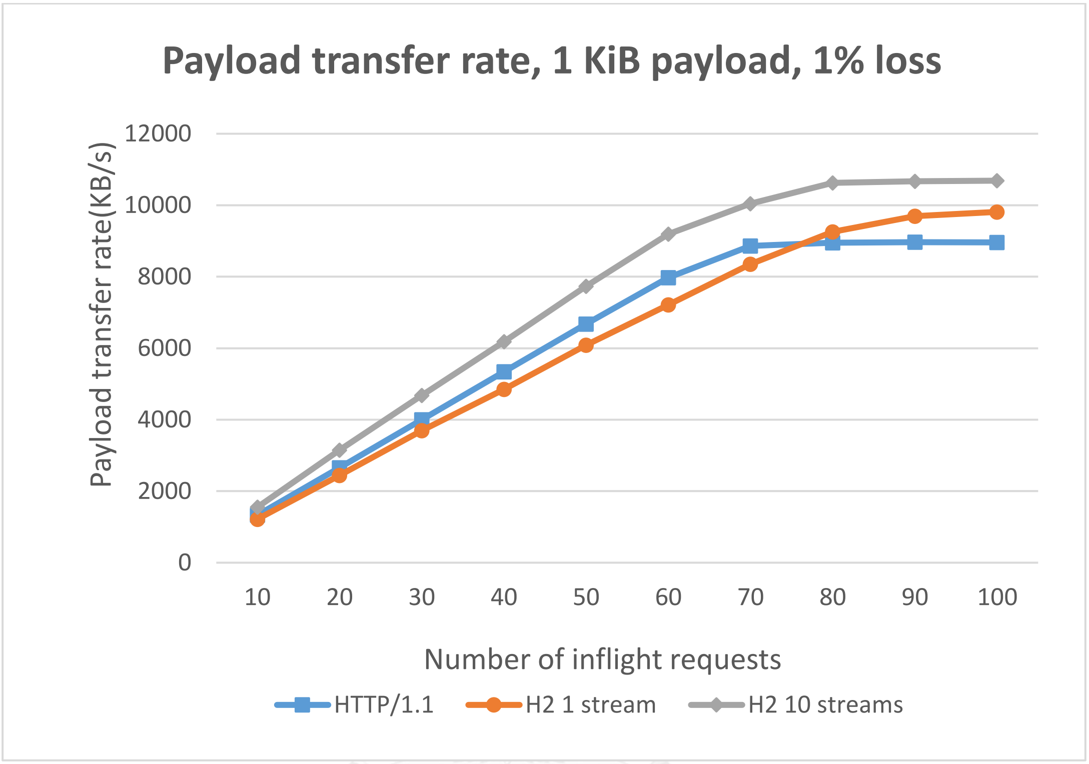

Figure 16: Experiment I payload transfer rate, 1KiB payload, 1% loss

3.3 Experiment II : HTTP/2, HTTP/1.1 co-running

In this experiment, we want to compare HTTP/2 performance in highly congested network to that of the HTTP/1.1. We want to know whether STREAM-CONNECTION EQUAVALENCE holds true in highly congested network or not.

3.3.1 Experiment II Test setup

In this test, we setup 2 client machines connected to a server machine through a bottleneck link of 100 Mbit/s. Link latency between client machine and server machine was 5ms. To measure payload bandwidth under congestion, we assigned one of the client machine as a stress machine and another as a measured machine. The stress machine loaded the server with HTTP requests and we ran the test on measured machine as shown in *Figure 17*. However, it is important to note that we had verified that in this test, server's CPU is not fully utilized, therefore there is no a performance bottleneck. As in previous experiment, we used Apache 2.4.18 as a web server on server machine, ApacheBench as HTTP/1.1 client and h2load as HTTP/2 client. For each test, we ran HTTP/2 load on measured machine alongside HTTP/1.1 on stress machine and vice versa. We varied payload size from 1KiB to 128KiB. Then we switched stress machine to HTTP/2 and measure machine to HTTP/1.1 such that we always measure the same machine to eliminate systematic error. We then compare the ratio of payload transfer rate between HTTP/1.1 and HTTP/2 both of the same number of connections and the same number of inflight requests.

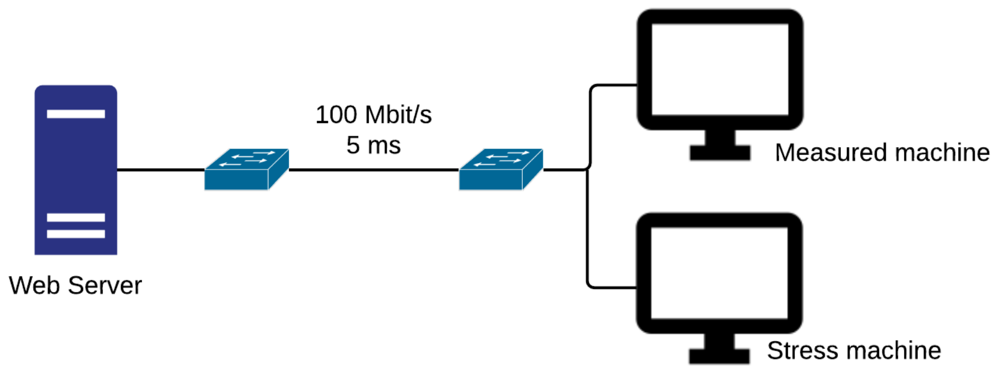

Figure 17: Experiment II test setup

3.3.2 Experiment II Test result

From *Figure 18*, we can see that, with equal number of connection, ratio between HTTP/2 and HTTP/1.1 pay load throughput starts from ration between number of inflight requests and converges toward 1 in larger payload size.

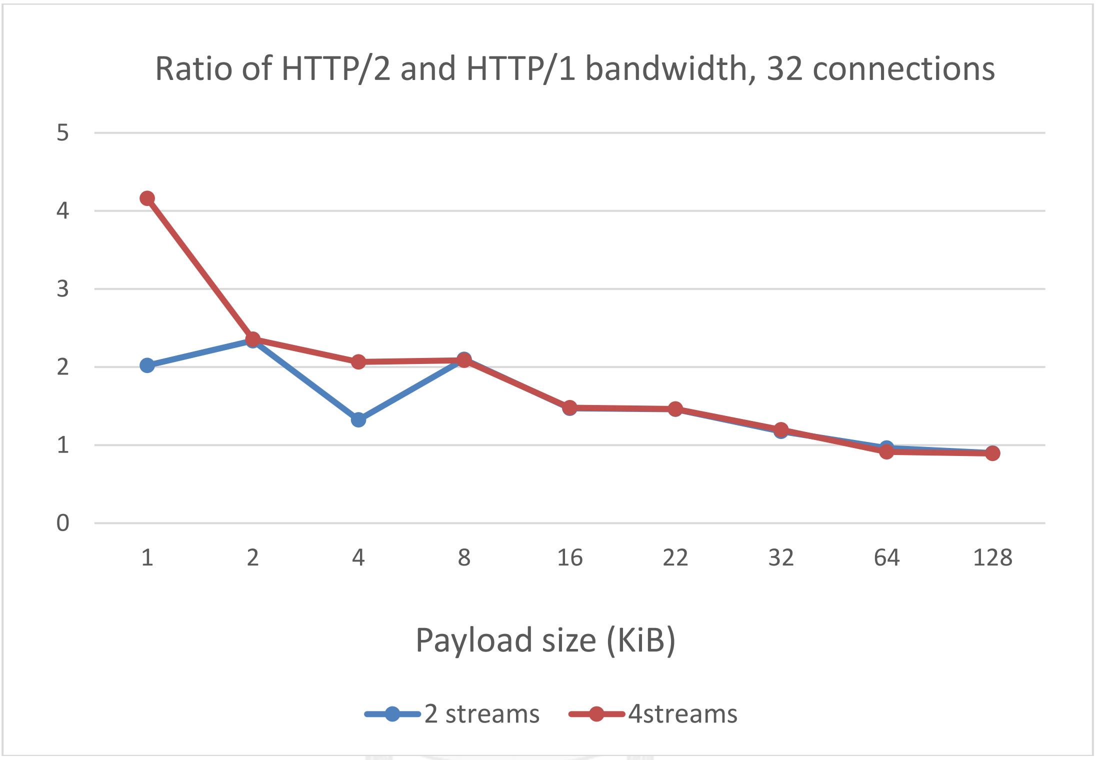

Figure 18: Experiment II Ratio of HTTP/2 and HTTP/1 bandwidth, 32 connections

In *Figure 19*, we can see that ratio between HTTP/2 and HTTP/1.1 converges to the ratio of number of HTTP/2 connections and HTTP/1.1 connections. In other words, ratio of payload throughput is proportional to number of inflight requests in latency limited case and converges toward number of connection as payload size increases in bandwidth limited case.

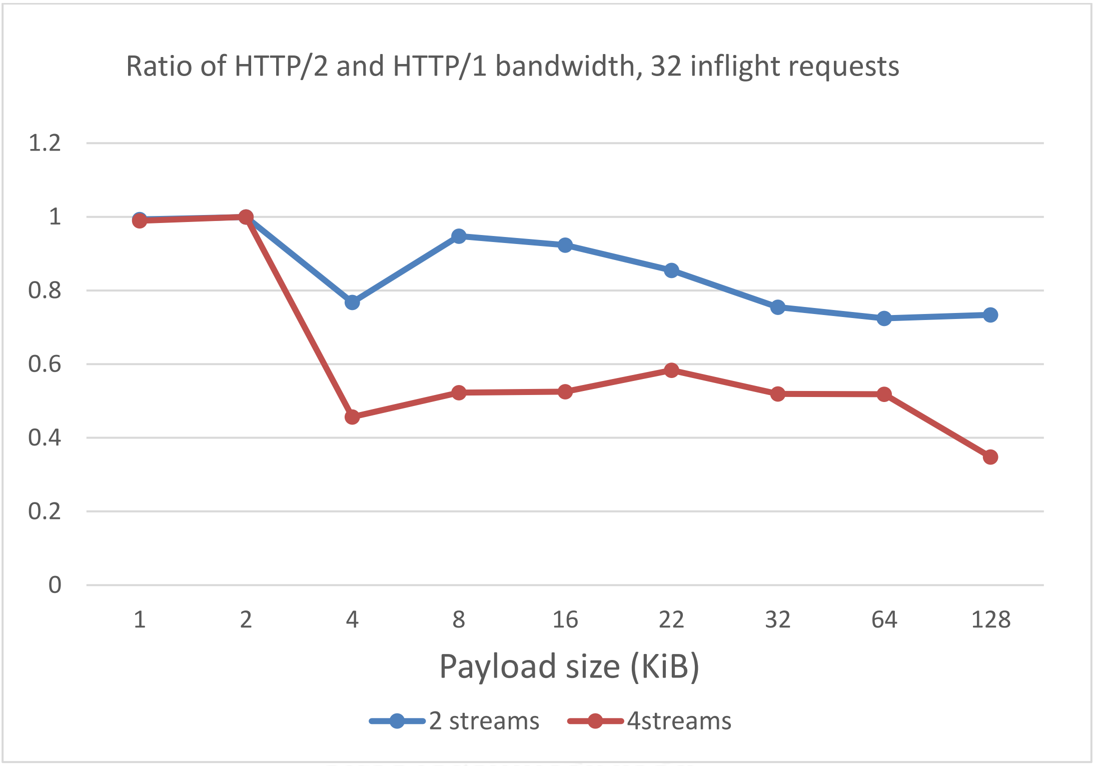

Figure 19: Experiment II Ratio of HTTP/2 and HTTP/1 bandwidth, 32 inflight requests

3.4 Section 3 Conclusion

As shown from our experiments, the HTTP/2 multiplexed streams can be effectively used to replace multiple HTTP/1.1 connection, increasing service capacity on the server machine and improve bandwidth utilization. However, in highly congested network, from web users' perspective, there is a penalty from using multiplexed stream especially in case of large payload size as transport layer protocol tries to balance bandwidth between each connection. On the other hand, from ISP and service providers' perspective, using multiplexed streams is more friendly to other connections comparing to multiple connections. This also helps preventing bandwidth starvation. Therefore, web users may want to use multiplexed stream only if they have excess bandwidth or they don't want to compete for bandwidth such as running a background job such as volunteer computing with networking requirements. However, service providers can greatly benefit from multiplexed stream because overhead from multiple connections can be significant. For volunteer computing application, using multiplexed streams can serve 2 purposes: 1. It can fully utilize bandwidth without creating contention with other application. 2. It reduces connection overhead on the coordinator machine.

4. Conclusion

In this work, we solved the problem of performance loss when running volunteer computing application. In the “Nicer Processing” section, we used idea of adaptive reservation to solve problem of share between users in CFS based system. We solved problem of violation of user boundary by using global statistics. Our method also does not require any modification to the operating system. In section “Nicer Protocol”, we studied the HTTP/2 multiplexed stream. We showed the equivalence between multiplexed stream and multiple separate connections. We also showed that the HTTP/2 multiplexed stream is more network friendly compared to multiple HTTP/1.1 connections. The implementation of HTTP/2 only require modification of web server, web page and web browser.

Thus, we have demonstrated 2 different approaches dealing with 2 different resources important to volunteer computing applications.

In “Nicer Processing", the of our work is to allow a person to continuously donate their processing power without interfering with their foreground usage. The principal in this work can also be adapt to other resource such as GPU. For example, processing time allocation for asynchronous physics simulation and graphic processing on the Graphic Processing Unit. However, one of the point that can be improved is the parameters in Starvation Detection Algorithm. In this work, we hand-tuned the parameters. In order to tune parameters to suite wide range of application, an automated tuning is required.

In “Nicer Protocol”, our intention in this study is to uncover another aspect of HTTP/2 performance characteristics and its potential in volunteer computing application. However, our study is far from being comprehensive. Though the HTTP/2 standard was finalized, its implementation has yet to be solidified. Therefore, further studies are required.

REFERENCES

[1] D. P. Anderson and G. Fedak, "The computational and storage potential of volunteer computing," in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, 2006, pp. 73-80.

[2] J. Kay and P. Lauder, "A fair share scheduler," *Communications of the ACM,* vol. 31, pp. 44-55, 1988.

[3] P. Turner, B. B. Rao, and N. Rao, "CPU bandwidth control for CFS," 2010.

[4] J. Corbet. (2007). *CFS group scheduling*. Available: https://lwn.net/Articles/240474/

[5] J. Corbet. (2007). *TTY-based group scheduling*. Available: https://lwn.net/Articles/415740/

[6] J. Corbet. (2007). *Group scheduling and alternatives*. Available: https://lwn.net/Articles/418884/

[7] M. Prpic, R. Landmann, and D. Silas, "Red Hat Enterprise Linux 6 Resource Management Guide," *Managing system resources on Red Hat Enterprise Linux,* vol. 6, pp. 1-40, 2013.

[8] C. S. Pabla, "Completely fair scheduler," *Linux Journal,* vol. 2009, p. 4, 2009.

[9] (2015). *BOINC preferences*. Available: http://boinc.berkeley.edu/wiki/preferences

[10]C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Multimedia Computing and Systems, 1994., Proceedings of the International Conference on*, 1994, pp. 90-99.

[11]T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni, "Adaptive reservations in a Linux environment," in *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, 2004, pp. 238-245.

[12]M. Laner, P. Svoboda, P. Romirer-Maierhofer, N. Nikaein, F. Ricciato, and M. Rupp, "A comparison between one-way delays in operating HSPA and LTE networks," in *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt), 2012 10th International Symposium on*, 2012, pp. 286-292.
[13](2016, 10 March). *HTTP archive*. Available: http://httparchive.org
[14]*HTTP Pipelining - The Chromium Projects*. Available: https://www.chromium.org/developers/design-documents/network-stack/http-pipelining
[15](2016). *Bug report: Enable HTTP pipelining by default*. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=264354
[16]*SPDY: An experimental protocol for a faster web*. Available: https://www.chromium.org/spdy/spdy-whitepaper
[17]M. Belshe, M. Thomson, and R. Peon. (2015). *Hypertext transfer protocol version 2 (http/2)*. Available: http://tools.ietf.org/html/rfc7540
[18]H. de Saxcé, I. Oprescu, and Y. Chen, "Is HTTP/2 really faster than HTTP/1.1?," in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2015, pp. 293-299.
[19]M. Varvello, K. Schomp, D. Naylor, J. Blackburn, A. Finamore, and K. Papagiannaki, "To HTTP/2, or not to HTTP/2, that is the question," *arXiv preprint arXiv:1507.06562,* 2015.
[20]H. Kim, J. Lee, I. Park, H. Kim, D.-H. Yi, and T. Hur, "The upcoming new standard HTTP/2 and its impact on multi-domain websites," in *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*, 2015, pp. 530-533.
[21]Y. Elkhatib, G. Tyson, and M. Welzl, "Can SPDY really make the web faster?," in *Networking Conference, 2014 IFIP*, 2014, pp. 1-9.
[22]X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "How speedy is SPDY?," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 387-399.
[23]*The Apache HTTP Server Project*. Available: https://httpd.apache.org
[24]T. Tsujikawa. *Nghttp2: HTTP/2 C Library*. Available: https://nghttp2.org

APPENDIX

Reservation-Based Scheduling(RBS)

RBS allows user to specify processor requirement and controls allocation of processor time to the process in each period. It guarantees availability of the processor to the process by restricting processor admission and utilization of other processes.

Adaptive Reservation (Feedback Scheduling)

RBS scheme that selects RBS's parameter based on previous scheduling error.

$$e = d - t$$

where e is scheduling error, d is scheduling deadline and t is task's soft deadline. Adaptive Reservation tries to drive e towards 0.

PID Controller

The PID algorithm is describe by

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right)$$

where y is the measured process variable, r the reference variable, u is the control signal and e is the control error. The control signal is a sum of three terms: proportion term, integral term and derivative term. Since CPU time measuring range is limited, we only consider first and second term.

VITA

Korakit Seemakhupt was born in Bangkok, Thailand on April 1992. He received Bachelor of Engineering in Computer Engineering from Chulalongkorn University, Thailand in 2013.