FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds

Daehyeok Kim[1], Tianlong Yu[1], Hongqiang Harry Liu[3], Yibo Zhu[2], Jitu Padhye[2], Shachar Raindel[2], Chuanxiong Guo[4], Vyas Sekar[1], Srinivasan Seshan[1]

[1]Carnegie Mellon University, [2]Microsoft, [3]Alibaba, [4]Bytedance

Contents

概要

概要

- コンテナを利用した大規模クラウドアプリケーション
  - high resource efficiency
  - light weight isolation
- その一方で data-intensive なアプリケーション（deeplearning, data analytics）は高性能のネットワーク（RDMA）を要求する
- コンテナ化によるメリットと RDMA によるメリットをいいとこ取りしたい！ $\rightarrow$ **FreeFlow**

目的・背景

背景

- コンテナ化のコアメリットは効率的で柔軟なアプリケーションマネジメントの提供である。
- containerized cloud ではネットワークについて以下の要件をコンテナが満たしていてほしい
  - *Isolation*：container はそれぞれの namespace を持っていてほしい
  - *Portability*：virtual network (overlay network) を利用してコンテナ間通信できてほしい
  - *Controllability*：control / data plane policy をコントロールできてほしい
- RDMA の完全仮想化は性能を殺してしまうし、RDMA 自体は control / data plane の policy 変更が難しい（HW に寄るため）
  - 実際のところ、TensorFlow や Spark などは専用ベアメタル + RDMA などの形態で運用しパフォーマンスを担保するケースがある。
  - これは提供側にとってもユーザ側にとっても辛い

目的

- 目的：container 環境でも RDMA が使えて、ベアメタル + RDMA くらいの性能が出るようにする！
- *Isolation*, *Portability*, *Controllability* に加えて、*high performance* も達成する！
- container から利用できる RDMA のその他の手法に関してまとめたものが下図

Property	Native	SR-IOV [21]	HyV [39]	SoftRoCE [36]
Isolation	✗	✓	✓	✓
Portability	✗	✗	✓	✓
Controllability	✗	✗	✗	✓
Performance	✓	✓	✓	✗

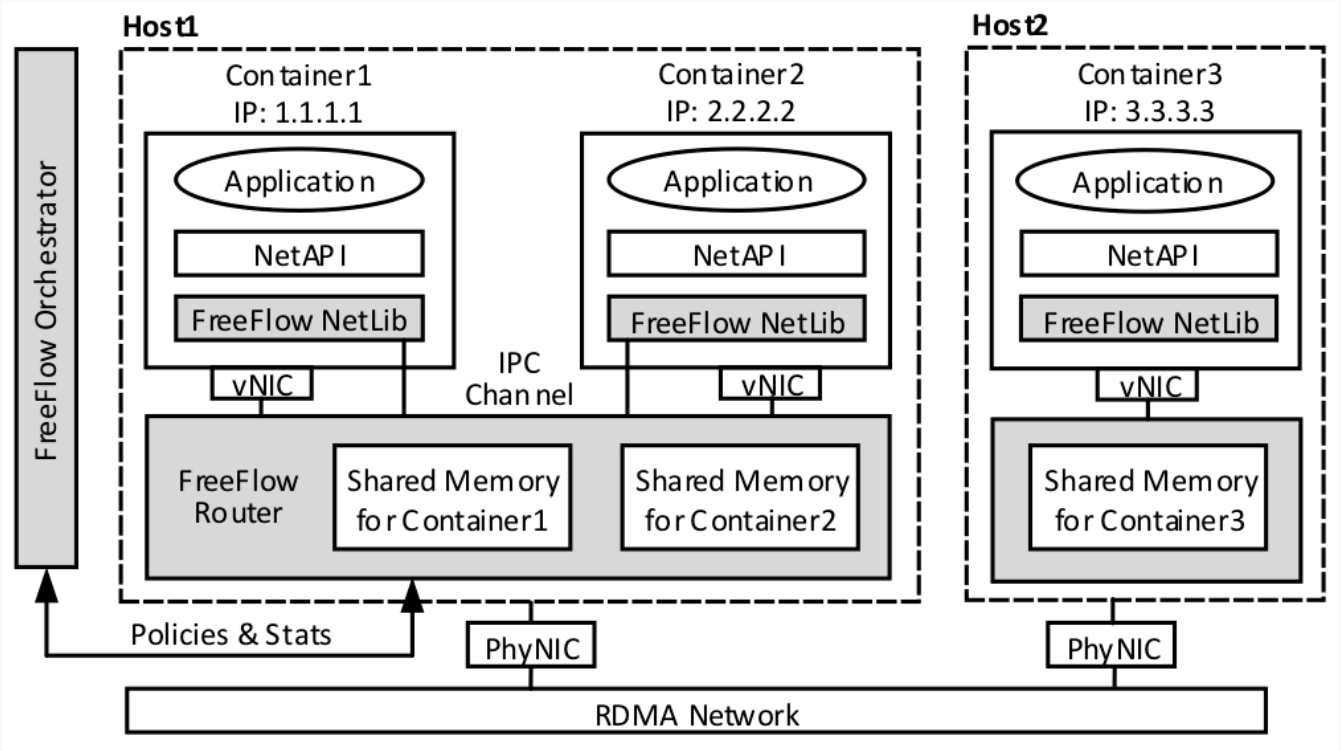
FreeFlow

FreeFlow の概要

- RDMA では API を利用してアプリケーションが直接 HW NIC にコマンドを投げる
- FreeFlow ではこのアプリケーションと HW NIC の間に割って入る。
  - FreeFlow Router (以降, FFR) は同一ホストの別コンテナとして起動しておく。
  - FFR で virtual network の提供 + control / data plane policy の適用。
  - FFR は各 contianer の memory 領域を share していて、App → NIC も NIC → App もここのメモリを橋渡しにする。
  - 同じ物理メモリを利用している（これが shadow memory?）ため zero-copy (FFR → App)

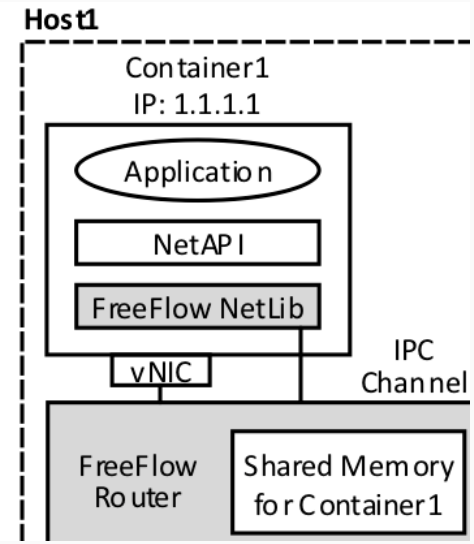
Architecture

- グレーの部分が FreeFlow で修正した部分



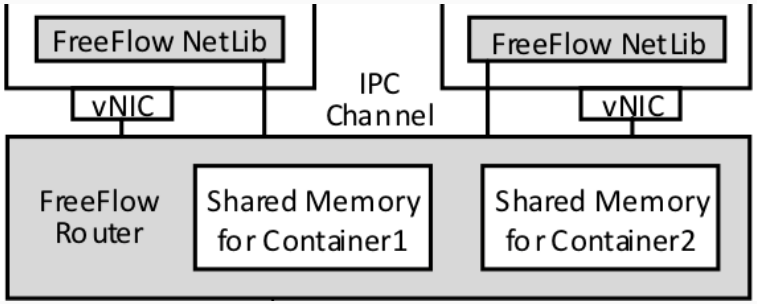
FreeFlow Library (FFL)

- App コンテナ内部で App に
  FreeFlow transparent を提供する
- ただし、App 内部からは通常の RDMA
  Verbs library を利用しているようにしか見えない
- App はコードの修正なし
  （もしくはほんの少しの修正）に利用できる
- FFR とやり取りする



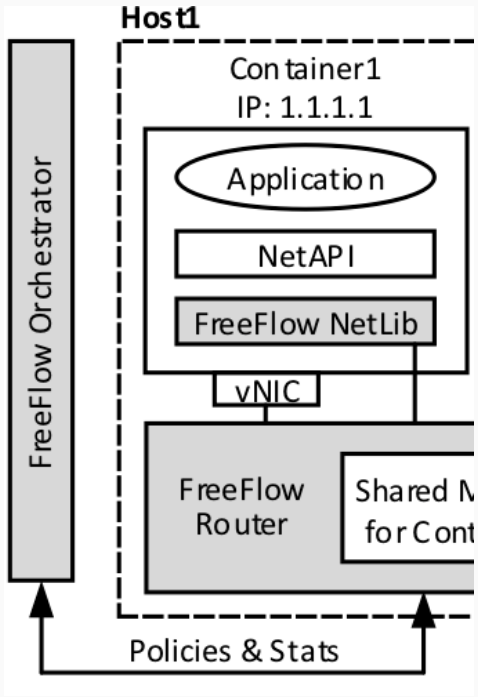
FreeFlow Router (FFR)

- 各ホストでシングルインスタンス（コンテナ）として起動し、そのホストの各コンテナに virtual network の提供を行う
- FFR との channel をコントロールすることで、data plane のリソースポリシー（QoS など）を実装する
- message-level event のハンドリングになる。
- FFO と連携し、IP アドレスなどのタスク処理も行う



FreeFlow Orchestrator (FFO)

- コントロールプレーンを司る
  （らしいが論文を通してあまり説明が無い。。。）
- コンテナ通信のコントロールや
  クラスタのリアルタイムモニタリングなど。
- メモリマップの保持もここで行っている



Transparent Support for RDMA operation

Transparent Support for RDMA operation

- これどこまでかこうかね...

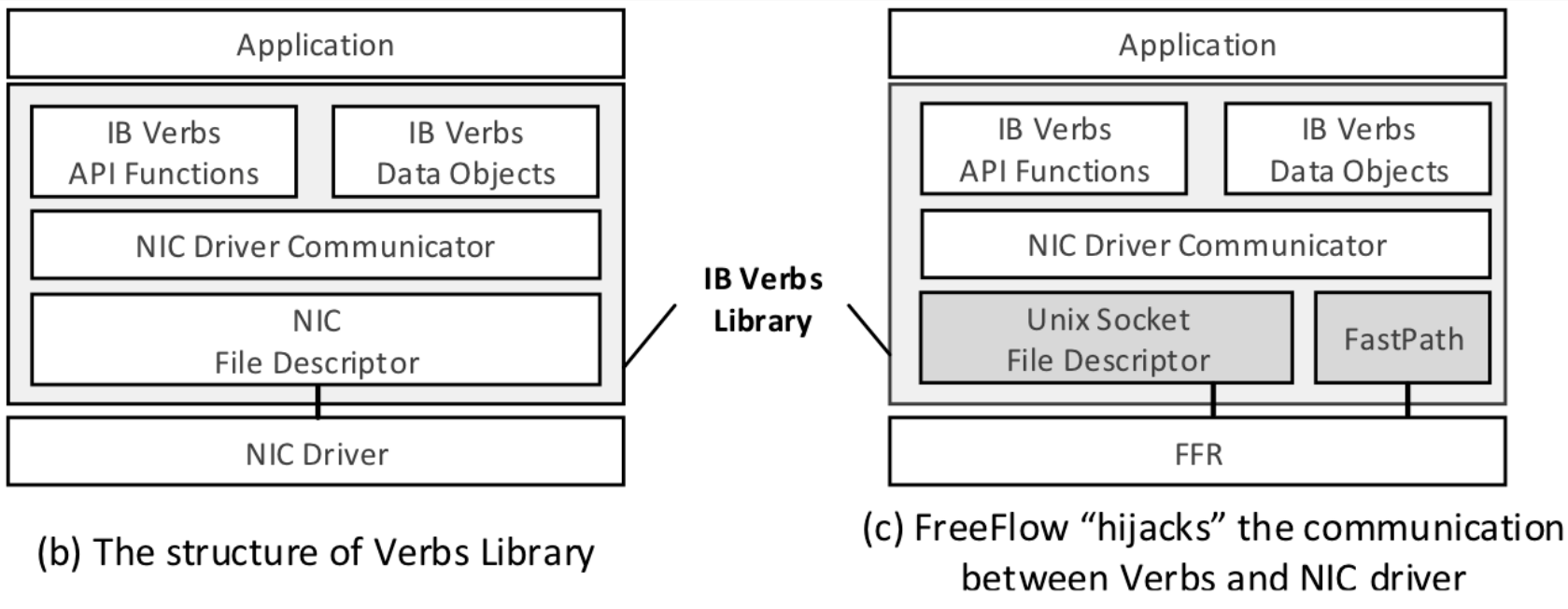
Communication channel between FFL and FFR

Communication Channel between FFL and FFR

- FFL と FFR の間の Communication Channel について。
- FFR 側では App が RDMA に対して行ったリクエスト（API コール）を模倣する必要がある。
- 2 つの考え方がある。
  - Unix file descriptor を利用する
  - Fastpath を利用する

Verbs forwarding via File Descriptor

- API コール自体を RPC のように模倣するのではなく、NIC fd に対して行われる要求を FFR 側で模倣する。
- コンテナ内部の NIC fd を Unix domain socket に差し替える。もう片方の終端は FFR。
- FFR はコンテナ内の仮想 Queue への操作を物理 NIC の実際の Queue への同じ操作に mapping する。逆の通信も同様の動きをする。



(b) The structure of Verbs Library

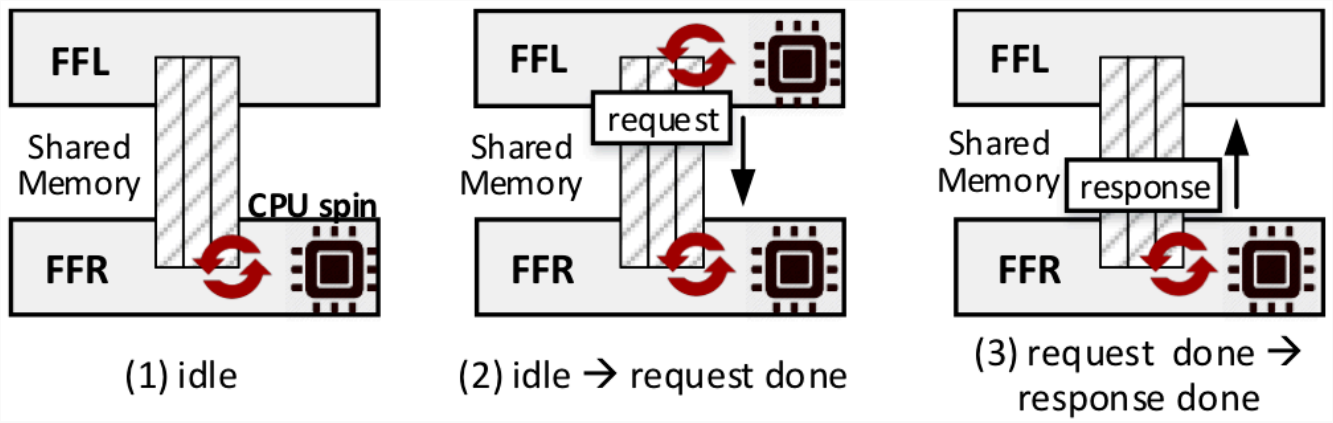
(c) FreeFlow “hijacks” the communication between Verbs and NIC driver

File Discriptor 方式の欠点

- レイテンシが大きい。ラウンドトリップタイムを計測したらかんたんに 5μs を超える。
- レイテンシセンシティブなアプリケーションでは致命的。。。
- ただし、CPU はほとんど使わない $\rightarrow$ ボトルネックの解消に CPU リソースを少しだけ拝借する設計 **Fastpath** を考えた

Fastpath between FFL and FFR

- FFL と FFR でメモリを共有する方式。
  1. まず FFR が CPU を回し、FFL からのリクエスト、すなわちメモリなにか書かれないか確認する。
  2. リクエストを受け取ったら FFR は直ちに処理する、その際 FFL も CPU を回しレスポンスを確認する。
  3. レスポンスが帰ってきたら直ちに処理し、FFL 側は CPU を手放す。



Fastpath 方式の欠点と対策のための実装

- CPU overhead がある。
  - 1FFR（つまり 1 ホスト）で 1CPU のみ利用する。複数コンテナの着信確認は 1CPU で行うようにデザインした。
  - データパス上にあり、且つ non-blocking な関数のときにのみ利用される（？）ため、FFL 側での CPU 利用時間は極めて小さい。
- FFO はホスト上にレイテンシセンシティブなアプリケーションが無いことを知っている場合（これはコンテナイメージの実行で判断？）、Fastpath と CPU の利用を止めることができる。

Discussion

CPU overhead

- 今回は performance ためにあえて許容した。
- CPU がやるべき仕事を hardware にオフロードできれば解決は可能かもね。
- future work の一つ

Security

- FFR で各コンテナのメモリ状態を取りまとめているから、同一ホストだし IPC 領域スキャンすれば確認できるのでは？
  - FFR は個々の QP に共有メモリバッファを作成する、つまりコンテナ内部に存在する共有メモリが、コンテナの仮想メモリアドレスにマッピングされるため問題ない。
- memory key（？）の問題．これは FreeFlow というより raw RDMA における one-sided operation の問題であり、FreeFlow で別段悪化するなどはない。

Work with external legacy peer

- FFR はリモートの peer が FFR 使ってるか普通の RDMA かなどの判別をするわけではないため、RDMA で普通に通信可能。

Container migration

- offline migration（shutdown → move → reboot）可能。
- IP アドレスは変わらないので、reboot したら RDMA connection を再確立させることができる。
- live migration はできない。

VM host

- 今回の prototype では基本的にベアメタル上にコンテナを起動させているが、physical NIC にアクセスできればいいので、SR-IOV などを利用している VM 上なら可能。

Congestion control

- RDMA NIC に輻輳制御メカニズムは入っており、FreeFlow はそれに準拠する。

評価

構成

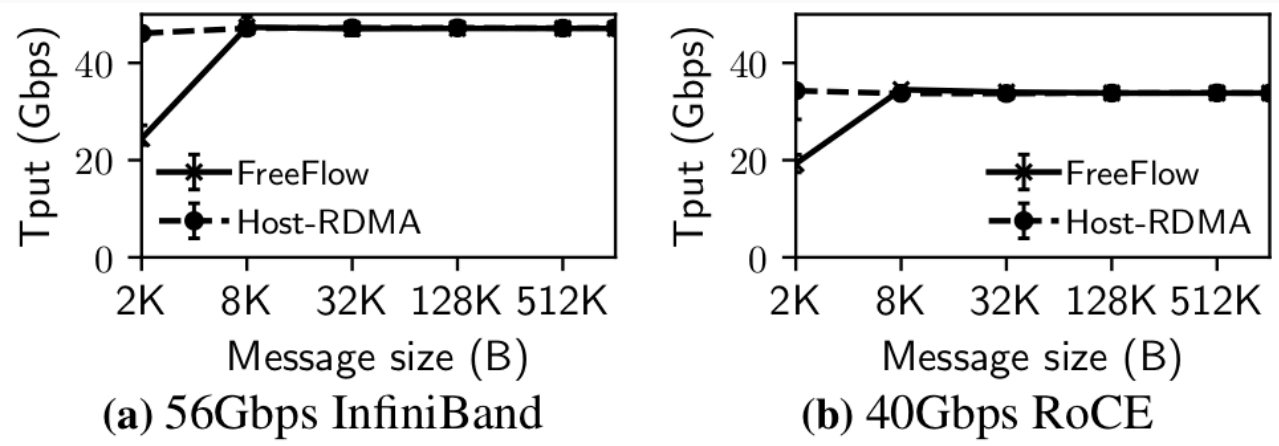
1. Infiniband
   - CPU : Intel Xeon E5-2620 2.10GHz 8-core × 2
   - RAM : 64GB
   - NIC : 56Gbps Mellanox FDR CX3
   - OS : Ubuntu 14.04 (3.13.0-129-generic)
2. RoCE
   - CPU : Intel Xeon E5-2609 2.40GHz 4-core
   - RAM : 64GB
   - NIC : 40Gbps Mellanox CX3 NIC
   - OS : Ubuntu 14.04 (4.4.0-31-generic)
3. その他
   - Container Runtime : Docker (v1.13.0)
   - Virtual Network : Weave (v1.8.0)
   - OvS kernel module : enabled

Microbenchmark: Throughput and Latency

- ib_send_lat, ib_send_bw で two-sided operation (SEND) の latency, bandwidth の測定
- ib_write_lat, ib_write_bw で one-sided operation (WRITE) の latency, bandwidth の測定
- これらのツールは FreeFlow 上でコードの変更なしに実行できた
- inter-host と intra-host を区別しないため（？）、inter-host performance value を計測した。（なんかあとで Latency 計測のときに Throughput と同じようにとか言いながら ToR 越しに接続しているのでなんともわからん。/ あー interhost 文しか計測してないぞという話か？）

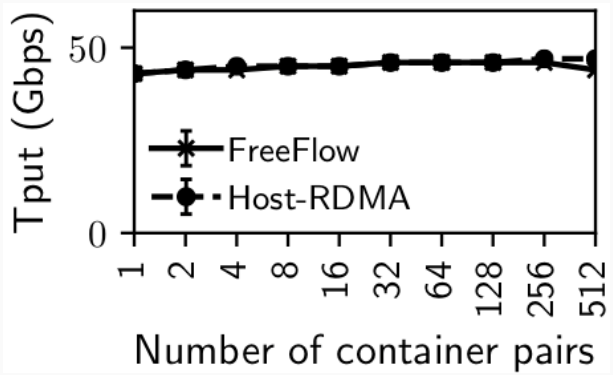
Microbenchmark: Throughput

- 2 種類の testbed で RDMA SEND/WRITE のスループットを測定。
- 1GB のデータを 2KB〜1MB の単位で分割して転送している。
- Host-RDMA はベアメタル + RDMA の性能
- message size $\geq$ 8KB の場合、Infiniband/RoCE ともにベアメタルの性能と同じになる。
- (WRITE のデータは READ よりわずかに良いらしいが論文からは省略されている。)



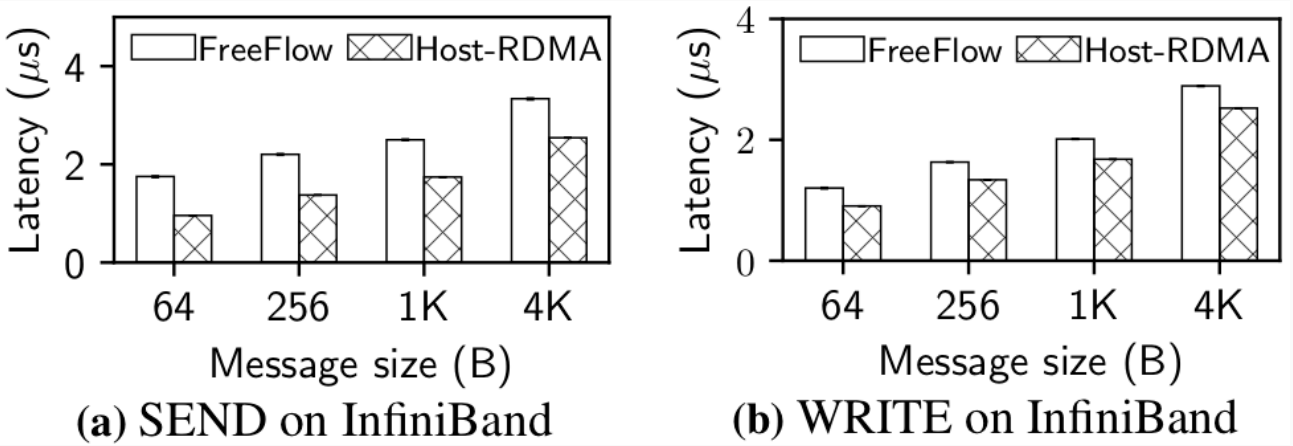
Microbenchmark: Throughput

- message size $\leq$ 8KB の場合について、同時稼働コンテナ数（flow 数）を 512 にスケールアップしたところブラフがベアメタル + RDMA とほぼ同じになった。
- このときの帯域幅は各フローで均等に分散していることも Jain's fairness index により確認した。
- 細かすぎて latency に食われる分、Throughput が飽和していなかっただけのよう。（実際 2CPU を FRR に利用する方法でもうまく行ったらしい）



Microbenchmark: Latency

- 64B, 256B, 1KB, 4KB のメッセージを送信し、レイテンシを測定した。
- one-sided である WRITE のほうが、two-sided である SEND よりレイテンシが少なく、FreeFlow とベアメタルの間のギャップも小さい。
- とはいえ SEND の場合であっても余剰 Latency は 1.5μs 程度。
- これは FFL と FFR の間の IPC によるレイテンシで、WRITE の場合は trigger が 1 回だけど、SEND の場合は 2 回なのが効いているから。



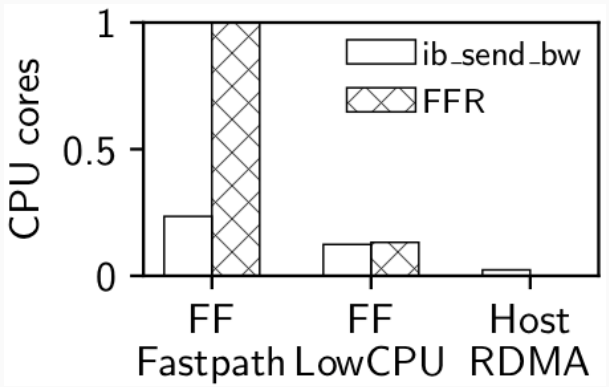
Microbenchmark: Latency

- ちなみに、1.5mus ってどんれくらい？
- ネットワークの one hop：0.55mus
- TCP stack：least 10mus
- TCP virtual network latency：40mus 以上（彼らのテストでは）
- → FreeFlow は container virtual network を有効にした状態で RDMA の latency advantage を保存できている！

Microbenchmark: CPU overhead

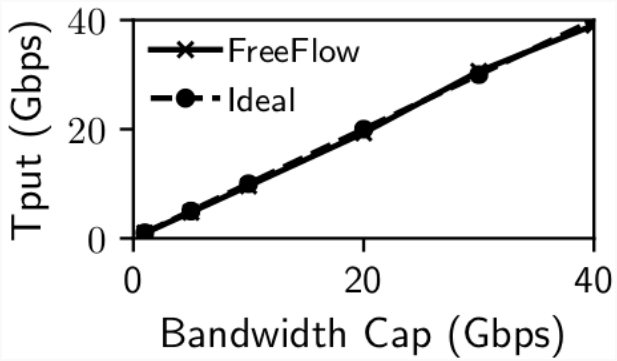
- Fastpath enable の場合とそうでない場合は実際どのくらいレイテンシに差が出るか。
  - Fastpath だと 2.4μs だが、disable すると 17.0μs まで上昇する。
- Throughput 測定時（このときどのパターンでも帯域はフルに使っており、message size は 1MB）の CPU Utilization はやはり LowCPU（disable Fastpath）のほうが断然良い。
- ワークロードによって使い分けるのがベスト

Host RDMA	Fastpath	LowCPU
1.8μs	2.4μs	17.0μs



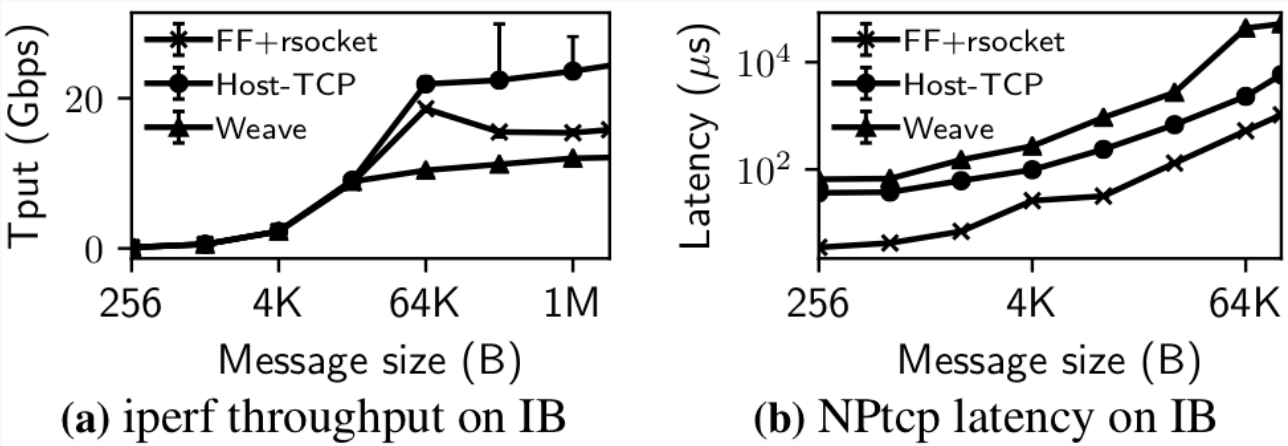
Microbenchmark: Rate limiter and Performance Isolation

- rate limiter の性能を検証した
- Infiniband の testbed で、異なるホストのコンテナ間で単一のフローを開始した。
- flow rate に制限をかけ、帯域幅を 1Gbps〜40GBps まで変更して測定した。
- 図から分かる通り理想通りにリミットがかかっている。これを 6 の CPU overhead で実現している。
- これはコンテナペア間で分離できており、それぞれに制限をかけても問題なく動いた。



Microbenchmark: TCP socket over RDMA

- 本説とは少しずれる気がするが、socket-based application についても利益をもたらすよという話。
- 通常の TCP/IP virtual network に比べて、rsocket を利用して FreeFlow で RDMA すると早い。
- iperf で TCP throughput を、NPtcp を利用して TCP latency をそれぞれ計測した。(これらも変更の必要はなかった)
- Weave と比べると常に早く、Throughput でホストに勝てないのは socket と verbs の変換部分の overhead。



Real-world application

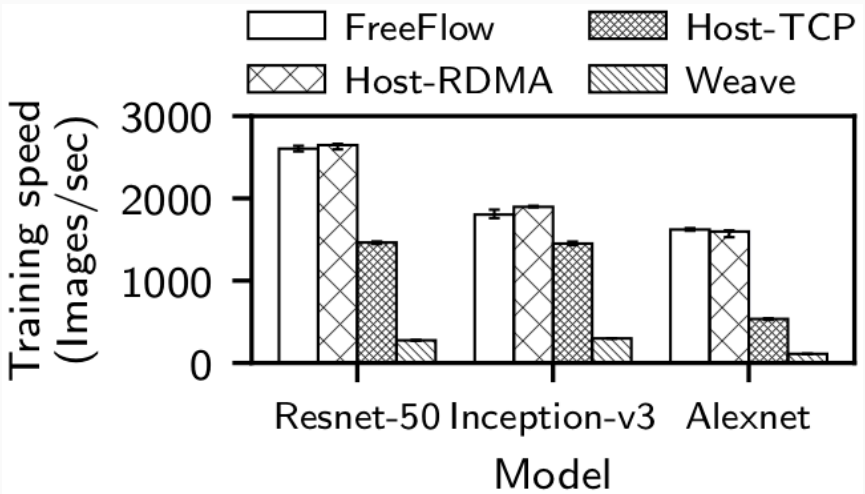
- Tensorflow と Spark を利用して機械学習とデータアナリティクスの性能を確認した。
- 比較対象は Host-RDMA（これに近づくほど理想的）、Host-TCP, Weave の三種類。
- これらはすべて Infiniband 構成で実行した。

Tensorflow

- 3 台のサーバ上で RDMA 対応の Tensorflow を実行。（変更は一行）
- 各サーバには NVIDIA GTX 1080Ti × 3 がそれぞれ搭載されており、一基が master かつ parameter サーバ, それ以外が workor
- 2 種類の training workload を実行する
  - Convolutional Neural Network： 画像認識用途
  - Recurrent Neural Network： 音声認識用途

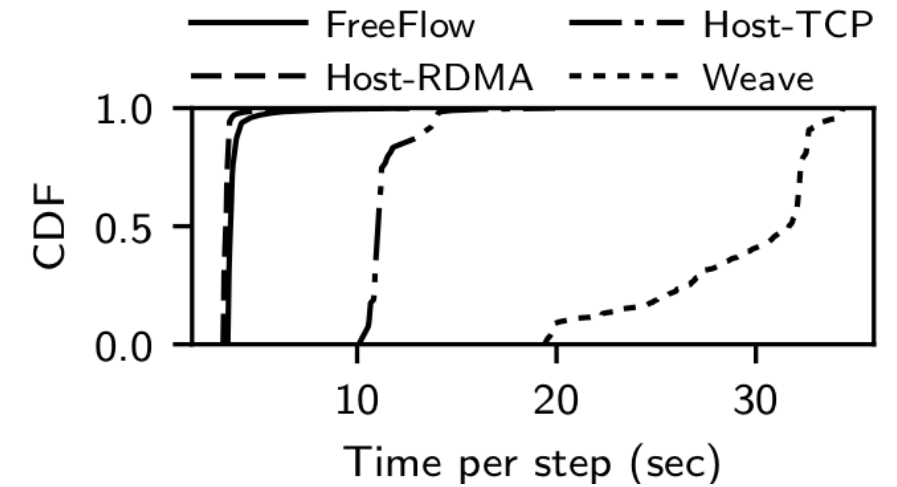
Tensorflow: CNN

- 3 種類のモデル（ResNet-50, Inception-v3, AlexNet）を training data として synthetic ImageNet data を利用。
- 分散トレーニングにおいてネットワークパフォーマンスがボトルネックになる。
  1. Host-RDMA と Host-TCP の比較をすると RDMA のほうが 1.8〜3 倍程度良い。
  2. FreeFlow と Weave を比較するとより大きい差がでている。(Alexnet に関しては 14.6 倍ほど RDMA のほうが良い)
- FreeFlow の性能は Host-RDMA に切迫している！ (Alexnet は少し高いがこれはノイズだろうとのこと)



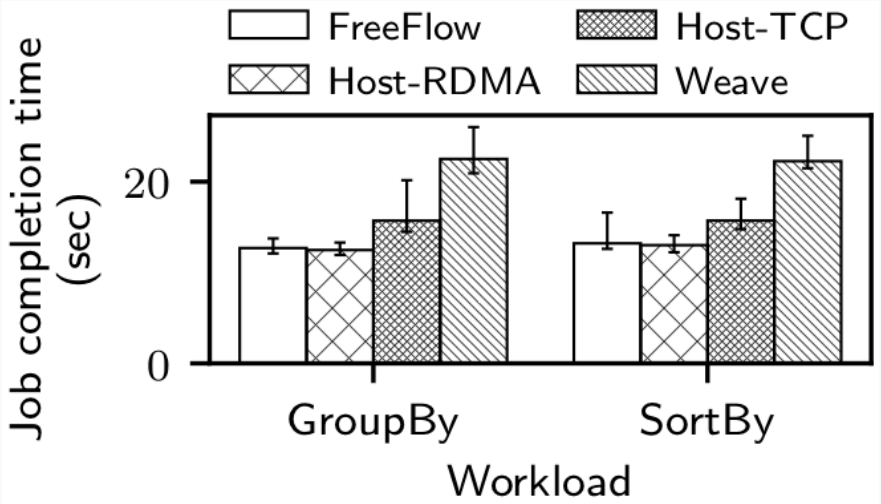
Tensorflow: RNN

- エンコーダ／デコーダとか隠れ層とかの設定の話があったがちょっと良くわからなかった。。。
- 各訓練ステップに費やされた時間を CDF にしたグラフ
- CNN の場合と同様に、FreeFlow は Host-RDMA に切迫しており、Weave に比べて 8.7 倍くらいの性能が出ている



Spark

- 2 つのサーバーで Spark を実行。片方には master, slave の container, もう片方には slave container で動作。
- Spark に同梱されている、*GroupBy* と *SortBy* のベンチマークで計測した。
- 結果的に、Tensorflow と同様の結果になり、ネットワーク性能がアプリケーション性能に大きく影響しているのがわかる。
- FreeFlow は Host-RDMA に切迫しており、Weave に比べて 1.8 倍の性能がでる。



結論

結論

- FreeFlow により、コンテナ環境に対して、*Isolation*、*Portability*、*Controllability*、*Performance* を満たす RDMA 環境を提供することができた。
- 肝心の Performance に関してもベアメタル RDMA に匹敵するくらいの性能が出る。
- Github はこちら