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**
- FreeFlow によりベアメタル RDMA と同程度のネットワークパフォーマンスを達成しながらコンテナの恩恵を受けることができる。

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 を利用してどのホストでも固定の virtual ip で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
  - 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

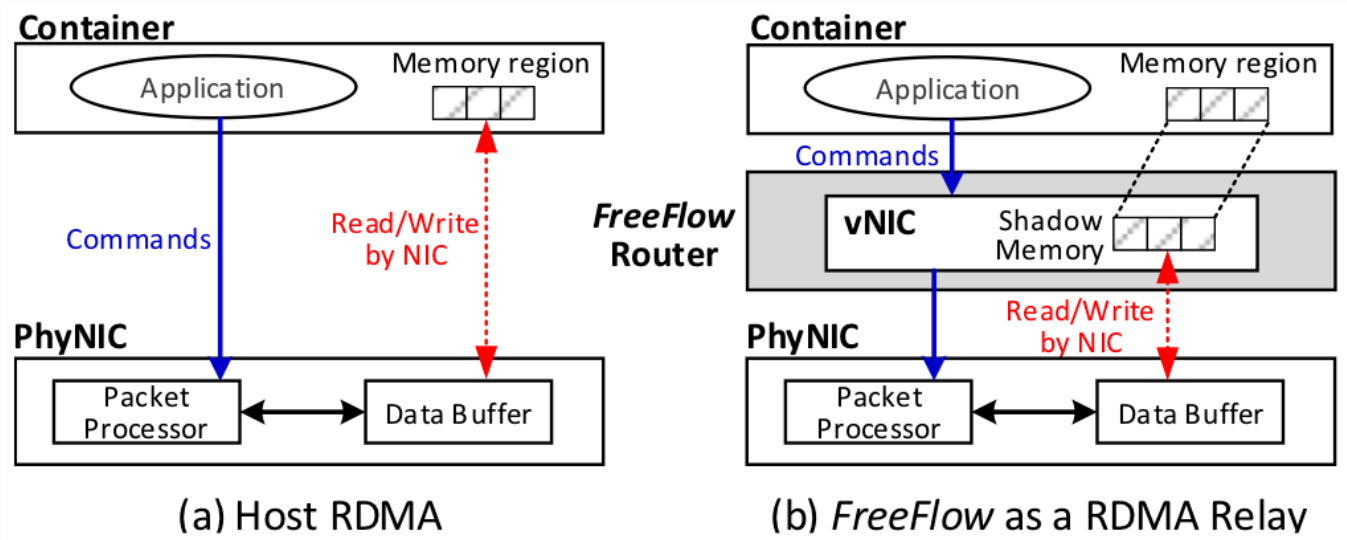
- 目的：container 環境でも RDMA が使えて、ベアメタル + RDMA くらいの性能が出るようにする！
- *Isolation*, *Portability*, *Controllability* に加えて、*high performance* も達成する！
- container から利用できる RDMA のその他の手法に関してまとめたものが下図

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

FreeFlow

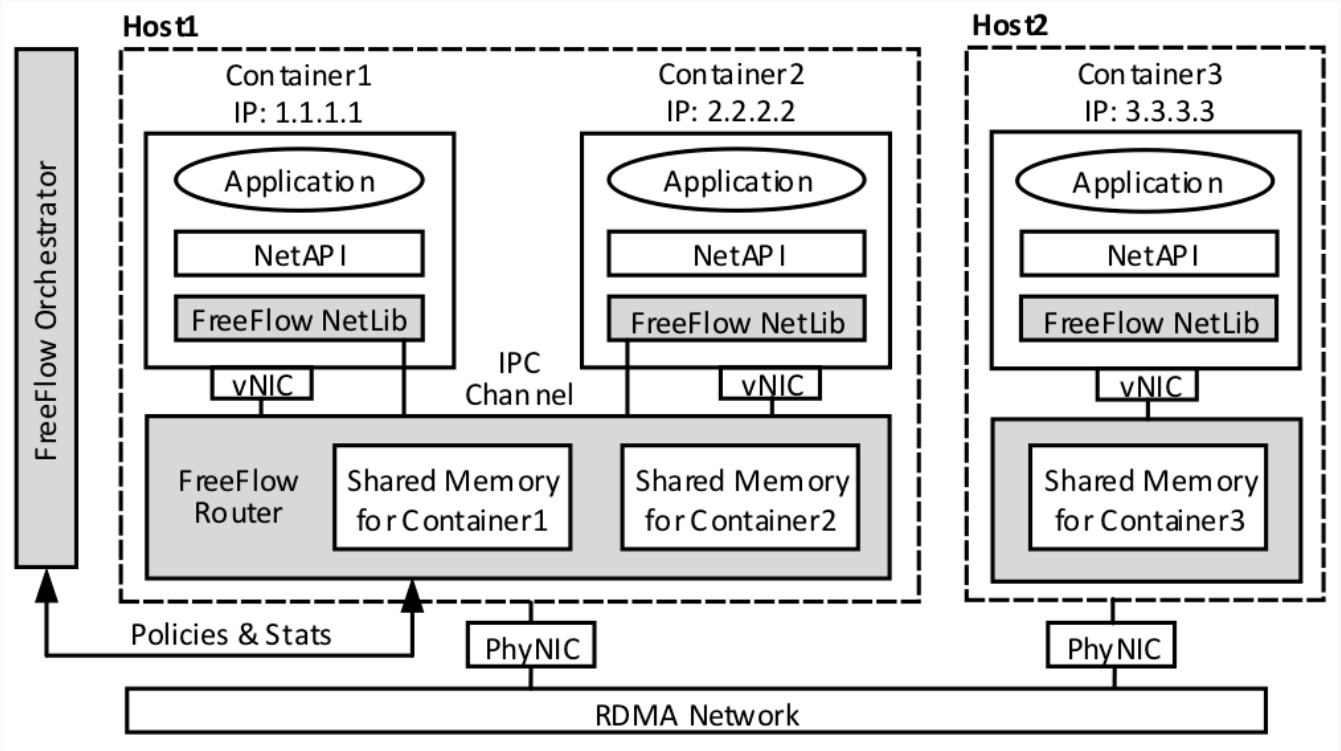
FreeFlow の概要

- RDMA では API を利用してアプリケーションから 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。
  - 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。



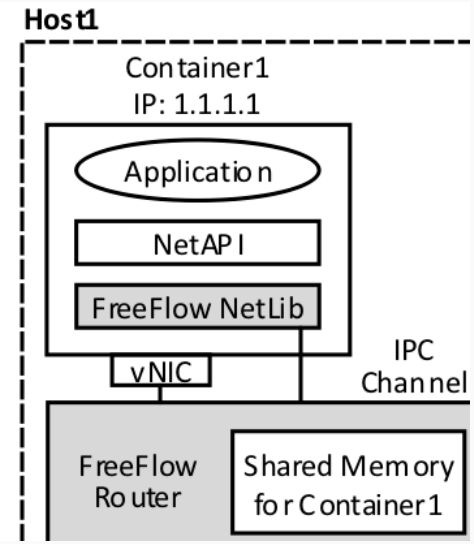
Architecture

- グレーの部分が FreeFlow で修正した部分



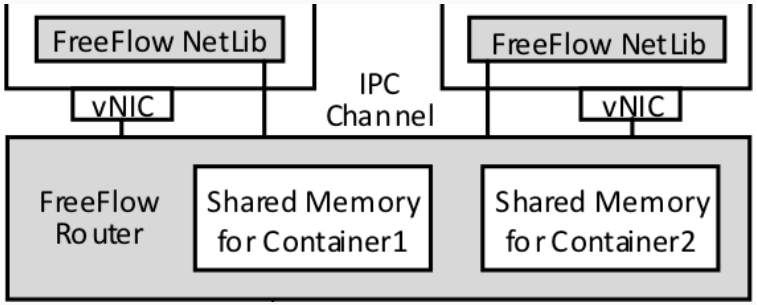
FreeFlow Network Library (FFL)

- 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



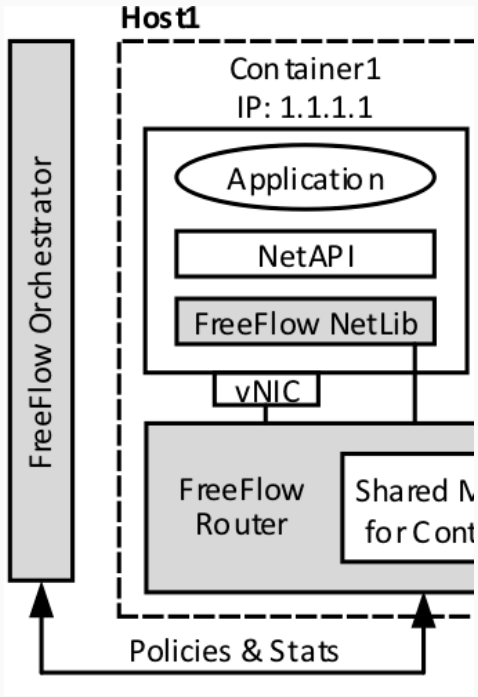
FreeFlow Router (FFR)

- 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
- FFL 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
- 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
  リアルタイムモニタリングなど。



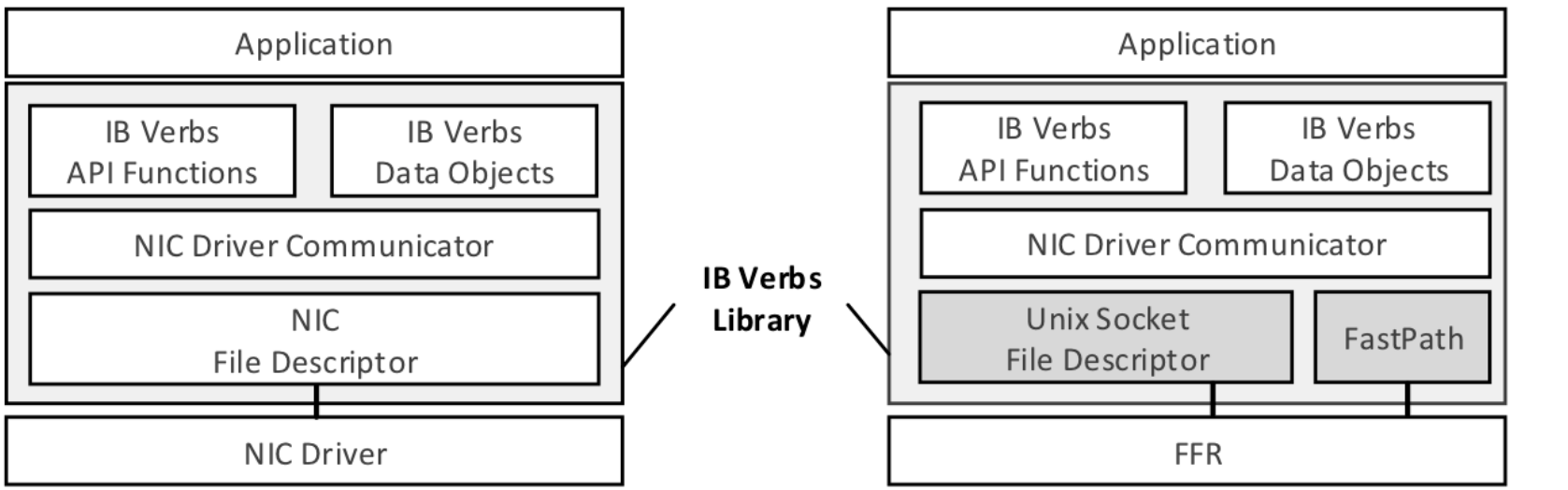
Communication channel between FFL and FFR

Communication Channel between FFL and FFR

- FFL と FFR の間の Communication Channel について。
- FFL 側で App が行った RDMA リクエスト（API コール）を、FFR 側で模倣する必要がある。
- 2 つの方式を考えた。
  - File Descriptor を利用する。(Unix domain socket による IPC)
  - Fastpath を利用する。(Shared memory による IPC)
- 以降特に断りがない場合、FreeFlow は Fastpath による方式だと思っていただければ。

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。(そのため App 側の変更は少ない)
- コンテナ内部の NIC fd を Unix domain socket に差し替える。もう片方の終端を FFR にすることで FFL の内容を FFR に伝える。
- Unix Domain Socket ベースの IPC



(b) The structure of Verbs Library

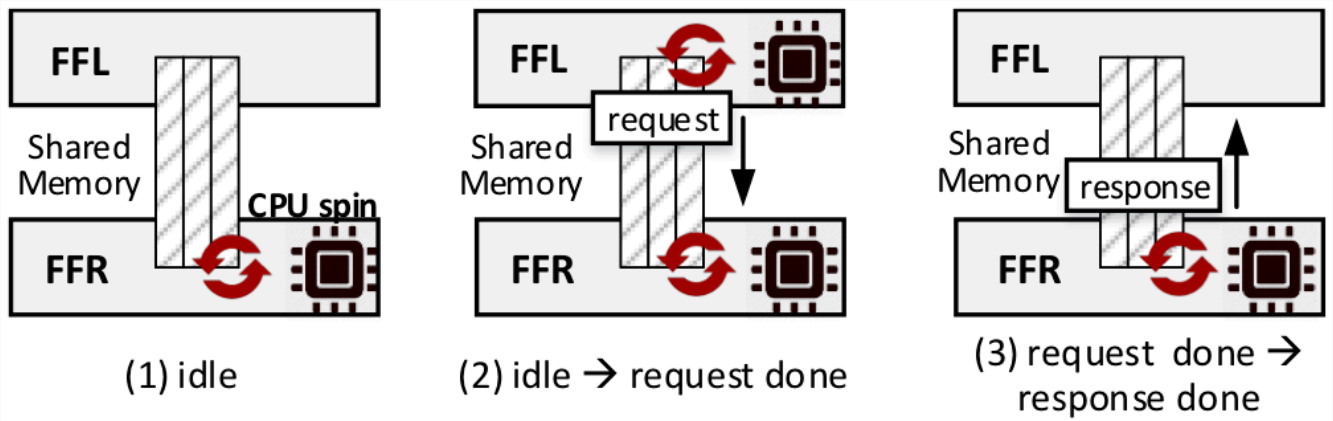
(c) FreeFlow "hijacks" the communication between Verbs and NIC driver

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らしい。
- 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。
- shared memory による IPC



Fastpath 方式の欠点と対策のための実装

- CPU を消費する。（メモリチェックが必要）
  - 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。
  - non-blocking な関数のときにのみ利用されるなどの制限を設けたため、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、Fastpath と CPU の利用を止めることができる。

評価

構成

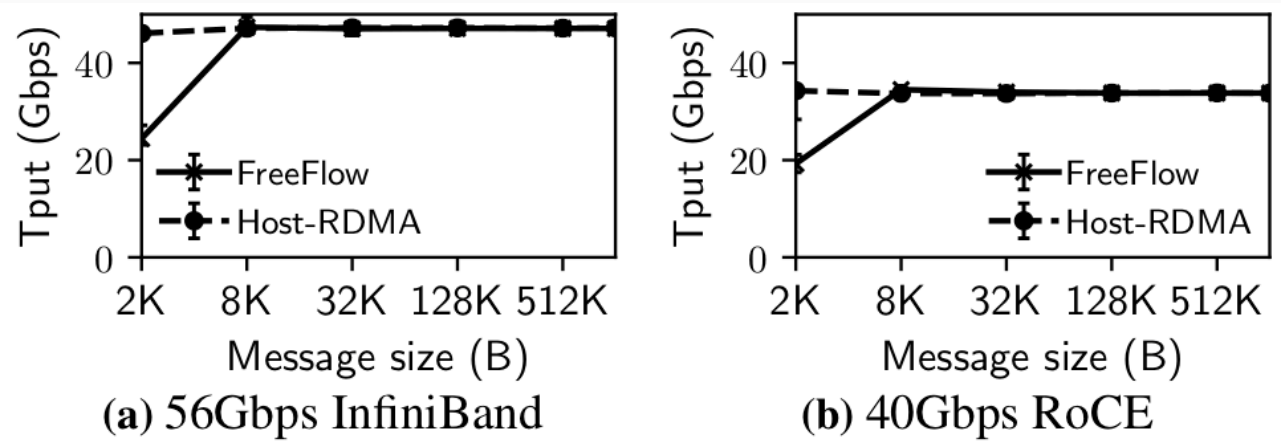
1. Infiniband
   - CPU : Intel Xeon E5-2620 2.10GHz 8-core × 2
   - RAM : 64GB
   - NIC : 56Gbps Mellanox FDR CX3
   - OS : Ubuntu 14.04 (3.13.0-129-generic)
2. RoCE
   - CPU : Intel Xeon E5-2609 2.40GHz 4-core
   - RAM : 64GB
   - NIC : 40Gbps Mellanox CX3 NIC
   - OS : Ubuntu 14.04 (4.4.0-31-generic)

Microbenchmark: Throughput and Latency

- SEND/WRITE の latency, bandwidth を測定
- ベンチマーク測定用ツールは FreeFlow 上でコードの変更なしに実行できた

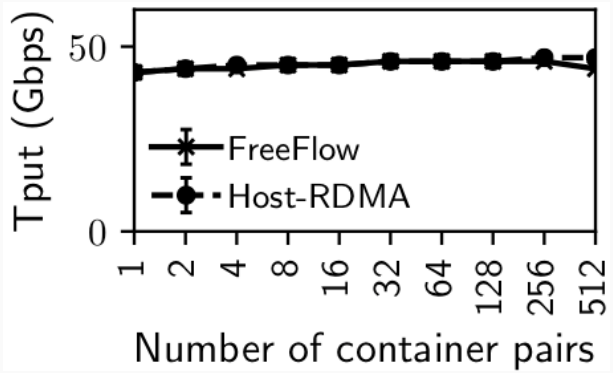
Microbenchmark: Throughput

- 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。)



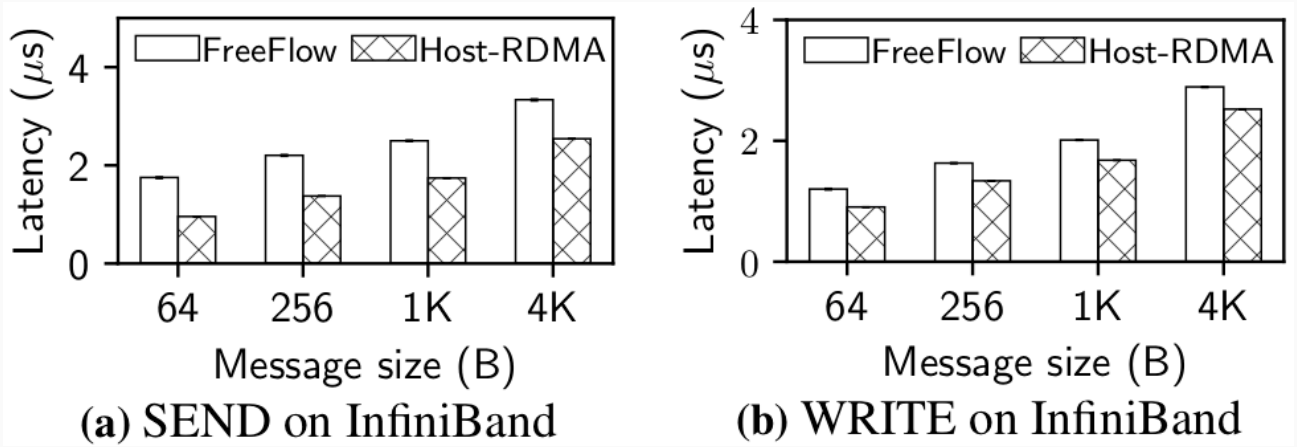
Microbenchmark: Throughput

- 同時稼働コンテナ数（flow 数）を 512 にスケールアップした場合においてもグラフ。
- 増加させてもほぼ一定の性能を維持している。（コンテナのスケールにも強い）
- 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確認した。
- （このグラフの Host-RDMA は一体なんなのかはよくわからない。。。）



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。
- WRITE のほうが、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。
- これは FFL と FFR の間の IPC の違いによるレイテンシの差。
- とはいえ SEND の場合であっても余剰レイテンシは 1.5μs 程度。



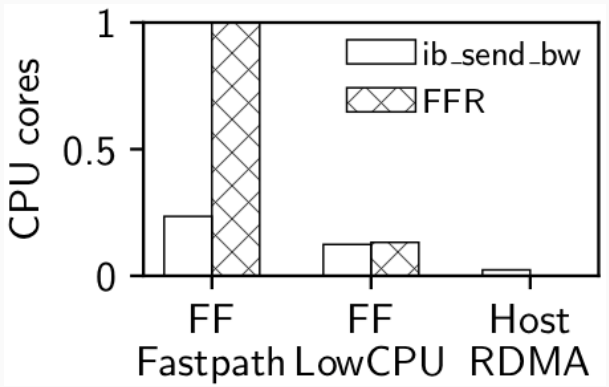
Microbenchmark: Latency

- ちなみに、1.5μs ってどれくらい？
- ネットワークの one hop：0.55μs
- TCP stack：least 10μs
- TCP virtual network latency：40μs 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

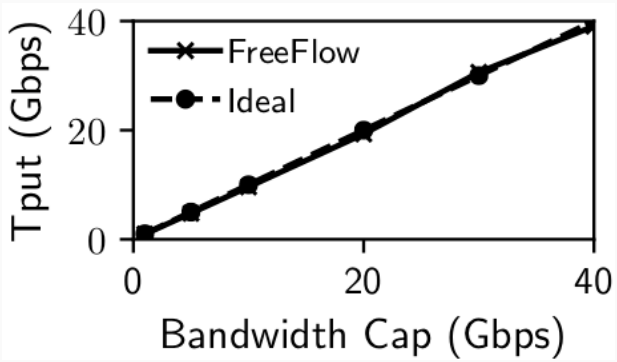
- Fastpath enable の場合とそうでない場合は実際どのくらいレイテンシに差が出るか。
  - Fastpath だと 2.4μs だが、そうでない場合は 17.0μs まで上昇する。
- Throughput 測定時（このときどのパターンでも帯域はフルに使っており、message size は 1MB）の CPU Utilization はやはり LowCPU（disable Fastpath）のほうが断然良い。
- ワークロードによって使い分けるのがベスト

Host RDMA	Fastpath	LowCPU
1.8μs	2.4μs	17.0μs



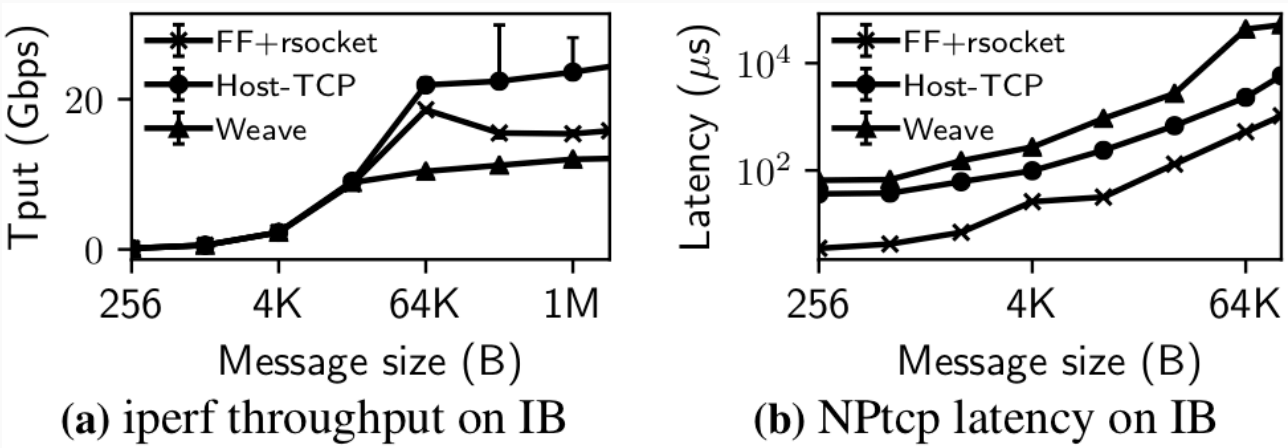
Microbenchmark: Rate limiter and Performance Isolation

- 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。
- 複数コンテナペア間でそれぞれに制限をかけても想定通り動いた。



Microbenchmark: TCP socket over RDMA

- 本説とは少しずれる気がするが、socket-based application（rsocket）についても利益をもたらすよという話。
- 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

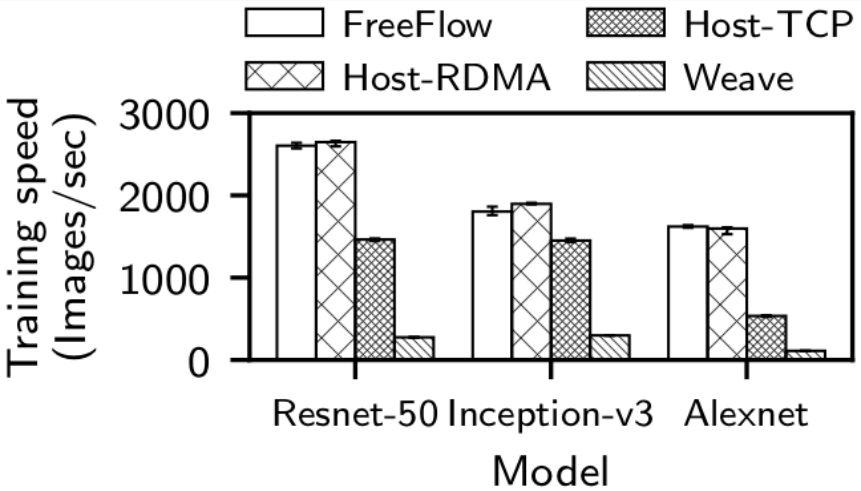
- Tensorflow と Spark を利用して機械学習とデータアナリティクスの性能を確認した。
- 比較対象は Host-RDMA（これに近づくほど理想的）、Host-TCP, Weave の三種類。
- これらはすべて Infiniband 構成で実行した。

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 worker サーバ。
- 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

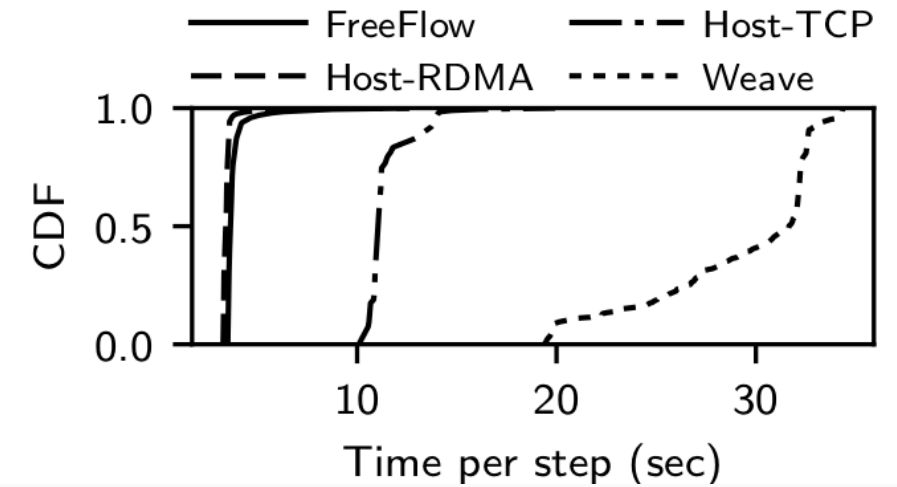
Tensorflow: CNN

- 3 種類のモデル（ResNet-50, Inception-v3, AlexNet）を training data として 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)



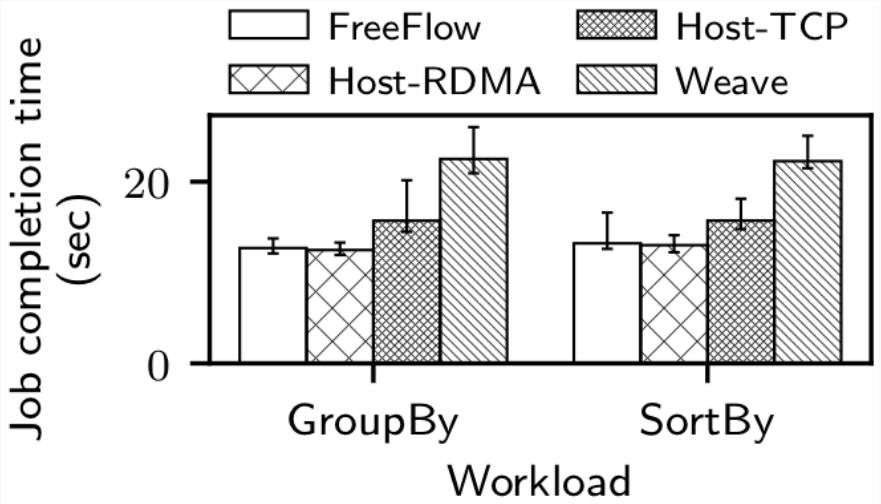
Tensorflow: RNN

- 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。



Discussion

CPU overhead / Congestion Control

- 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
- 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。

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（？）。説明がないのでわからなかったが、RDMA の問題なので FreeFlow で別段悪化はしない。

Container migration

- offline migration（shutdown → move → reboot）可能。
- FFR ／ FFO により 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 / Work with external legacy peer

- VM 上でコンテナを動作させる場合
  - 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、SR-IOV などを利用している VM 上なら可能。
- その他 RDMA ピアとの通信
  - 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。

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