Vol.2020-OS-150 No.15
2020/7/31

テイル・レイテンシ削減のためのハードウェアIRQハンドラにおけるパケット処理

菊地 隆文[1]　名取 廣[1]　河野 健二[1]

概要：現代の情報サービスは，分散システムが基盤となっている．分散システムにおいて，大規模な障害が発生すると，サービスの停止につながる．したがって，分散システムの信頼性を向上させることは重要な課題である．分散システムにおける障害の発生要因の 1 つに，パケット処理のテイル・レイテンシがある．例えば，ハートビートの遅延は，リソースモニタリングの遅延や障害の誤検知を引き起こす．このようなパケットの遅延は，ソフトウェアが要因となり発生することが報告されている．高速なパケット処理を可能とする技術として，DPDK や XDP が挙げられるが，他の割込みによる遅延や既存のシステムへの統合などの課題が残る．本論文では，オペレーティングシステム内において，パケット処理のテイル・レイテンシを削減するシステムを提案する．ハードウェア割込みハンドラ内に，安全にユーザーコードをロードし，パケット処理を可能とすることで，テイル・レイテンシを引き起こす要因を回避する．本システムの有効性を示すため，I/O 負荷がある状態において，パケットのエコープログラムを用いてラウンドトリップタイムを測定した．その結果，DPDK や XDP といった既存の技術と比較して，パケット処理のテイル・レイテンシを削減できることを確認した．

1. はじめに

現代の情報サービスは，分散システムを利用して構築されている．分散システムでは，1 つのリクエストに対して，複数のノードを用いて並列に処理を行う．それによって，1 台のマシンだけで処理を行う状況と比較して，処理時間を大きく削減することが可能となる．その結果，ユーザのリクエストに対して，ミリ秒スケールで応答するサービスが普及している．しかし，分散システムにおいて，大規模な障害が発生すると，サービスのクオリティの低下や停止を引き起こす．したがって，分散システムの信頼性を向上させることは，重要な課題であると言える．

分散システムにおける障害の発生要因の 1 つとして，パケット処理のテイル・レイテンシがある．テイル・レイテンシとは，通常のレイテンシと比較して，稀に発生する非常に大きいレイテンシのことを言う．近年の分散システムでは，その規模が大きくなった結果，頻繁にテイル・レイテンシが発生することが指摘されている [11]．

パケット処理のテイル・レイテンシが，分散システムにおける障害の発生要因となる例として，ハートビートの遅延を説明する．分散システムでは，ノード間でパケットを定期的にやりとりすることで，ノードの状態を確認している．このハートビートが，定められた時間内に応答がなかった場合，ノードが停止していると判断して障害を報告する．ハートビートの処理が遅延してしまうと，ノードが正常に動作しているにもかかわらず，障害が報告されてしまうことがある．HDFS では，ガベージコレクションが要因となり，ネームノードのハートビート処理が遅延することで障害を誤検知するという指摘 [6] や，従来より軽量なハートビートの提案 [3] がなされている．

パケット処理が遅延する要因は，ソフトウェアにあることが報告されている [17]．ソフトウェアにおける遅延は，カーネルからアプリケーションまで様々な要因がある．例えば，ディスク I/O，プロトコル処理，ガベージコレクション，アプリケーション処理などが挙げられる．このようなパケット処理が遅延する要因は，1 つに特定することができず，アプリケーション独自の対策では問題を根本的に解決することは難しい．そこで，システムソフトウェアにおいてパケット処理の遅延を削減する技術が必要である．

パケットを高速に処理するアプローチには，(1) カーネルバイパス と (2) カーネル内でのパケット処理 がある．(1) カーネルバイパス のアプローチの代表として DPDK[2] が挙げられる．DPDK は図 1(a) のように，DPDK アプリケーションが直接 NIC とインタラクションを行い，パケット処理を実行する．しかし，DPDK はカーネルをバイパ

[1] 慶應義塾大学
Keio Uniersity

スするため，既存のシステムへの統合が難しく，マシンのリソースを占有してしまう．(2) カーネル内でのパケット処理 の代表として XDP[4] が挙げられる．XDP は図 1(b) のように，パケット処理のプログラムをソフトウェア割込みハンドラ内で実行する．XDP では，カーネルと統合されたパケット処理を可能とするが，カーネルのリソースがビジー状態である際に，テイル・レイテンシが発生してしまう．これらのアプローチを採用した技術は，パケット処理を劇的に高速化することに成功しているが，テイル・レイテンシの発生や既存のシステムへの統合に対して課題を残している．したがって，パケット処理のテイル・レイテンシ削減を行うための新しい仕組みが必要である．

本論文では，カーネル内においてパケット処理のテイル・レイテンシを削減するシステムを提案する．具体的には図 1(c) のように，ハートウェア割込みハンドラ内に安全にユーザーコードをロードし，パケット処理を可能にする．ソフトウェアスタックの最下層であるハードウェア割込みハンドラ内で，パケット処理を行うことで，他のプロセスや割込みに起因する遅延を回避する．本システムでは，ユーザに記述されたパケット処理のプログラムを検証し，安全にカーネル内で実行されることを保証する．そして，そのプログラムはパケットのドロップ，送信，ペイロードの記録などの処理を実行することができる．

本システムの有効性を確認するために，Linux カーネルに実装を行い，DPDK や XDP といった既存のパケット処理技術との比較を行った．パケットのエコープログラムを記述し，ラウンドトリップタイムを測定することで，パケット処理のテイル・レイテンシを確認した．その結果，I/O 負荷がある状態で，パケット処理のテイル・レイテンシ削減に成功した．具体的には，99.9 perceintile において，DPDK と比較して 67.6%，XDP と比較して 74.9% の削減を行った．

本論文は次のように構成されている．2 章では，既存のパケット処理技術を紹介し，その問題点を説明する．3 章では，本システムの概要とその実装について説明する．4 章では，評価を取り，本システムがパケット処理のテイル・レイテンシ削減に有効であることを示す．5 章では，関連研究を紹介する．6 章では，本論文のまとめを記述する．

2. 既存のパケット処理技術の問題点

本章では，カーネルにおける従来のパケット処理と，パケットを高速に処理するアプローチについて説明する．そして，テイル・レイテンシについての評価を取ることで，既存のパケット処理技術の問題点を明らかにする．パケットを高速に処理するアプローチには，(1) カーネルバイパス と (2) カーネル内でのパケット処理 がある．カーネルバイパスのアプローチとして DPDK を，カーネル内でのパケット処理のアプローチとして XDP を説明する．

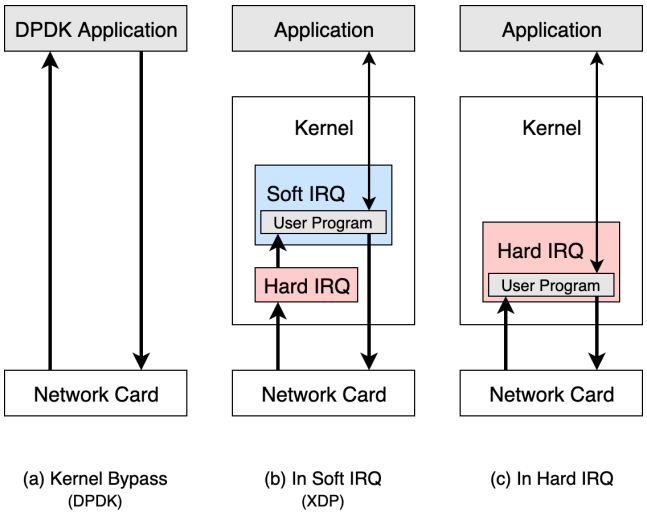

図 1 パケット処理のアプローチ

2.1 Linux Socket

従来のパケット処理では，カーネルが NIC とのインタラクションを制御し，アプリケーションがプロトコル処理を終えたパケットをシステムコールを用いて読み出す．まず，NIC にパケットが到着すると，NIC は DMA を用いてメモリにパケットを転送し，割込みを発生させる．カーネルでは，ハードウェア割込みハンドラが実行されることで，パケットの受信を検知して，ソフトウェア割込みをスケジュールする．その後，ソフトウェア割込みハンドラにおいて，ポーリングによるパケットの受信処理を行う．この際に，カーネルはリングバッファからパケットを読み出し，プロトコル処理などを実行する．そして，アプリケーションはシステムコールによって，ソケットからパケットを読み出すことでパケットを受信する．従来のパケット処理は，ソケット API として提供され，使用されている．しかし，近年の高速化したネットワーク（i.e. 10G, 40G, 100G）では，カーネルにおける TCP/IP やソケットなどのオーバーヘッドが非常に大きいことが指摘されている [16]．

2.2 DPDK

DPDK とはカーネルをバイパスし，パケットの高速処理を可能にする技術である．DPDK の特徴は，ポーリングによるユーザーレベルの送受信キューの操作である．DPDK アプリケーションは，NIC と直接インタラクションを行い，パケットの送受信をコントロールする．ポーリングを行い，キューを監視することで，パケット受信に高速に対応することが可能になる．そのため，従来の割込みを用いたパケット処理と比較して，スループットとレイテンシの双方について，非常に高い性能を発揮する．トレードオフとして，CPU，メモリ，NIC のポートなどのハードウェアリソースを占有する．例えば，ポーリングが CPU を無駄に消費している問題が指摘されている [12]．さらに，

DPDK アプリケーションはユーザー空間で動作するため，カーネル空間での処理の影響を受けることによって，テイル・レイテンシが発生する可能性がある．また，カーネルをバイパスするために，既存のシステムへの統合が難しい．DPDK は，従来のパケット処理と比べて高い性能を発揮するが，テイル・レイテンシ発生の可能性、リソース占有，既存システムとの統合などが問題となる．

2.3 XDP

XDP とはカーネルのソフトウェア割込みハンドラ内でパケット処理を可能にする技術である．ユーザは，パケット処理のプログラムを記述し，カーネルによって検証されたバイナリをロードすることができる．パケット処理のプログラムは，パケットのヘッダからペイロードまで参照可能であり，パケットのドロップ，送信，フォワーディングなどの処理を実行できる．マップを用いることで，ユーザー空間のプログラムと情報の共有も可能である．XDP では，ソフトウェア割込みハンドラにおいて，リングバッファからパケットを読み出した直後にパケット処理が可能である．受信したパケットに対して，sk_buff の割り当てや，プロトコル処理のオーバーヘッドを回避することができるため，非常に高速である．しかし，ディスク I/O などでカーネルがビジー状態である際には，パケットを受信するためのソフトウェア割込みハンドラのスケジューリングが遅れることがある．そのため，テイル・レイテンシが発生する可能性がある．

2.4 既存技術におけるテイル・レイテンシの調査

これまでの説明では，それぞれのシステムにおいてテイル・レイテンシが発生する可能性を指摘してきた．そこで，Linux socket、DPDK、XDP を対象として，エコーアプリケーションを用いて，パケットのラウンドトリップタイムを測定することで，実際にテイル・レイテンシが発生することを示す．

2.4.1 実験方法

まず，測定方法について説明を行う．エコーアプリケーションを用いて，パケットを往復させることで，ラウンドトリップタイムを測定する．クラインアントアプリケーションは，64 バイトの UDP パケットを 10,000 packets/sec のレートで 10 秒間送信を行い，受信したパケットからラウンドトリップタイムを計算する．クライアントの送受信処理がラウンドトリップタイムへ与える影響を最小化するために，クライアントアプリケーションは DPDK を使用している．サーバーアプリケーションは，受信したパケットの送信元と送信先のアドレスをスワップし，返信を行う．サーバ側では，RSS [7] がラウンドトリップタイムに与える影響を最小化するために，アプリケーションを 1 つのコアに固定し，そのコアに NIC からの割込みを集約させている．

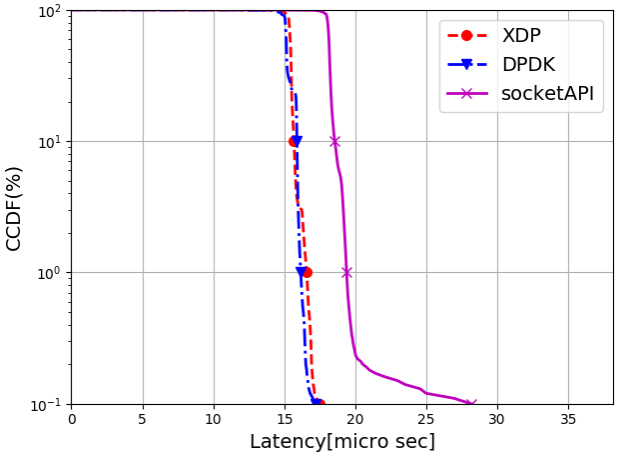

図 2 パケット往復時間の CCDF （負荷なし）

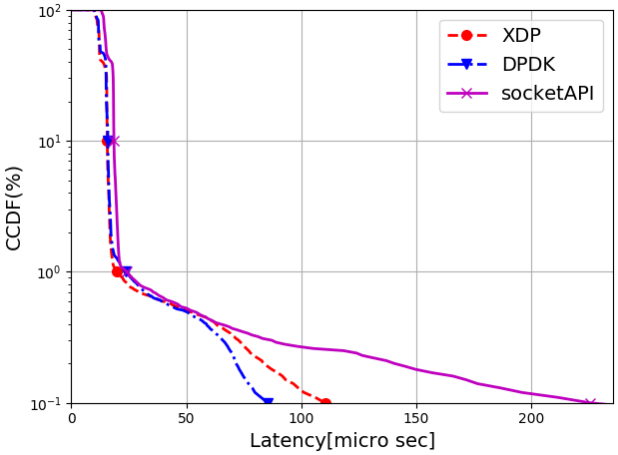

図 3 パケット往復時間の CCDF （負荷あり）

表 1 サーバーアプリケーション動作時の CPU 使用率（負荷なし）

System	CPU usage[%]
socket API	3.54
XDP	2.02
DPDK	100.00

次に，実験環境について説明する．サーバーマシンは，Intel(R)Xeon(R)E-2126G CPU@3.30GHz 6 コア，メモリ 16GB を使用した．オペレーティングシステムは，Ubuntu 18.04.2 LTS （Linux kernel 4.15.0）が動作している．クライアントマシンは，Intel(R)Xeon(R)CPU X5650@2.67GHz 6 コア，メモリ 4GB を使用した．どちらのマシンも，Intel Corporation Ethernet Controller X710 for 10GbE SFP+ を装備し，NIC を直結させて実験を行った．この際，CPU の governor を performance に設定し，CPU が最大周波数で動作するようにしている．また，DPDK のバージョンは 19.05 を使用している．

2.4.2 結果

図 2 は，システムに負荷がない状態でのパケットのラウンドトリップタイムの CCDF である．DPDK と XDP は，ほぼ同じレイテンシの分布であることがわかる．平均のラウンドトリップタイムが，DPDK と XDP が 15us であるのに対して，socketAPI が 18us であり，従来のカーネルによるパケット処理のオーバーヘッドが大きいことがわかる．このオーバーヘッドは，カーネルのネットワークスタックや，カーネル空間とユーザー空間のコンテキストスイッチなどが原因である．99.9 percentile では，socketAPI は DPDK や XDP と比較して，60%〜64% ほど大きい値である．

図 3 は，システムに I/O 負荷がある状態でのパケットのラウンドトリップタイムの CCDF である．I/O 負荷として，stress-ng [8] を用いて，テンポラリファイルへの書き込みと削除を行うことで，ディスク I/O に起因する割込みを発生させた．システムに I/O 負荷がある状態では，全てのシステムにおいてテイル・レイテンシが増加していることがわかる．99.9 percentile では，システムに I/O 負荷がない状態と比較して，socketAPI は 8.0 倍，DPDK は 5.0 倍，XDP は 6.3 倍にラウンドトリップタイムが増加している．100 percentile では，DPDK と XDP はそれぞれ 185us と 223us で，マイクロスケールのレイテンシであるのに対して，socketAPI は 15.287ms で，ミリスケールまでレイテンシが増加している．

また，表 1 は負荷がない状態で，サーバーアプリケーションが動作したときの CPU 使用率である．測定に使用したクライアントアプリケーションは，低負荷であるため，socketAPI と XDP の CPU 使用率は低いものである．それに対して，DPDK はポーリングを行うために，ネットワークの負荷にかかわらず，CPU 使用率が 100% である．この結果は 2.2 節で説明したリソース占有の問題を明らかにしている．

以上の調査を通して，DPDK や XDP はパケット処理を高速化することに成功しているが，テイル・レイテンシの問題に対処できていないことが示された．これは，割込み処理のスケジューリングなどが要因となるリソースの競合に，既存の技術が対応できていないためである．

3. 提案と実装

3.1 概要

本論文では，カーネル内においてパケット処理のテイル・レイテンシを削減するシステムを提案する．2 章で説明したテイル・レイテンシを引き起こす要因を避けるために，ハートウェア割込みハンドラ内に安全にユーザーコードをロードし，パケット処理を可能にする．ハードウェア割込みハンドラは，ソフトウェアスタックにおいてパケットの受信を検知する最も早いコンテキストである．そして，他のプロセスによって処理を中断されることがない．したがって，ハードウェア割込みハンドラにおけるパケット処理は，ソフトウェアで発生しうる遅延の要因を避けて実行される．本システムでは XDP と同様に，パケット処理のプログラムをカーネルによって検証し，ロードを行う．パケット処理のプログラムは，パケットのドロップ，送信，ペイロードの記録などの処理を実行することができる．

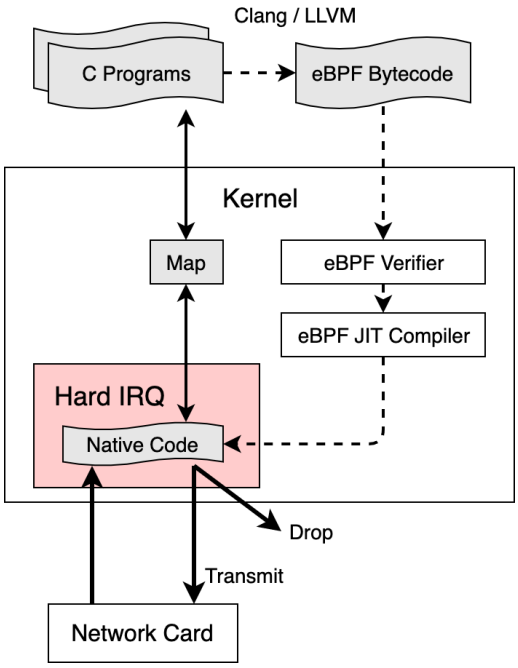

図 4 システムアーキテクチャ

図 4 は本システムのアーキテクチャである．本システムは主に (1) ハードウェア割込みハンドラ内でのパケット処理 と (2) 安全なユーザープログラムの実行 の 2 つに特徴付けられる．本章では，それぞれの概要と実装について説明を行う．そして，ユーザープログラムのロードと実行の流れについて説明を行う．

3.2 ハードウェア割込みハンドラ内でのパケット処理

本システムでは，ハードウェア割込みハンドラにおいて，パケットの取得，送信，ドロップなどの機能を提供する必要がある．2.1 節で説明したように，ハードウェア割込みハンドラはソフトウェア割込みのスケジュールを行い，即座にその処理を終了する．そのため，ハードウェア割込みハンドラには，リングバッファ内のパケットを取得する機能はない．本システムでは，ソフトウェア割込みハンドラで実行されるパケットの受信処理をハードウェア割込みハンドラに移植した．そのコードには，ソフトウェア割込みハンドラ内で実行されることを前提とした処理（e.g. sk_buff の割り当て，プロトコル処理）がある．そこで，ハードウェア割込みハンドラ内での実行が不可能な処理を削除し，最低限の受信処理のみを残すことで実行を可能とした．また，

本システムではハードウェア割込みハンドラ内で，パケットの送信とドロップを実行する．これらの処理を実装するために，XDP においてパケットの送信とドロップに利用される関数を移植した．それによって，図 4 に示されているハードウェア割込みハンドラ内でのパケットの取得，送信，ドロップを実現した．

現在の制約として，通常のネットワークスタックを利用したパケット受信が実行不可能である．この理由は，ハードウェア割込みハンドラ内で受信処理を行うために，sk_buff の割り当てやプロトコル処理を削除したためである．しかし，この制約は解消することが可能である．キューを実装し，ハードウェア割込みハンドラからソフトウェア割込みハンドラにパケットを渡し，ソフトウェア割込みハンドラで通常の受信処理を実行することができる．

3.3 安全なユーザープログラムの実行

ユーザープログラムをハードウェア割込みハンドラ内で実行するためには，カーネルの動作に影響を与えないように，様々な制約が必要となる．例えば，ユーザープログラムは，カーネルによって許可された領域のみへのメモリアクセスや，実行時間の保証（e.g. スリープやロックを使用しない，命令数の制限）などの特性をみたす必要がある．このような特性を保証するために，本システムでは extended Berkely Packet Filter（eBPF）[5] を用いる．eBPF は主に 2 つの理由から，本システムに採用された．1 つ目の理由は，eBPF が eBPF バイトコードの検証機構と JIT コンパイラを備えているからである．eBPF の検証機構は，カーネル内実行に必要な特性を保証し，JIT コンパイラは eBPF バイトコードの効率的な実行を可能にする．2 つ目の理由は，LLVM コンパイラのバックエンドが eBPF バイトコードをサポートしているからである．Clang などのフロントエンドを利用することで，高級言語でパケット処理のコードを記述し，eBPF バイトコードにコンパイルすることが可能である．

本システムの実装にあたって，複数の変更を eBPF に行った．まず，eBPF に新しいタイプを追加し，ハードウェア割込みハンドラに明示的にプログラムをロードできるようにした．次に，eBPF の検証機構が既存の eBPF マップ使用を禁止するように変更を加えた．eBPF マップは，ユーザー空間のアプリケーションとインタラクションを可能にするが，ソフトウェア割込みハンドラ内での実行を前提に実装されている．したがって，eBPF マップをハードウェア割込みハンドラで実行すると，データの一貫性を保証することができない．ハードウェア割込みハンドラ内で安全に eBPF プログラムを実行するためには，eBPF マップの使用を禁止する必要がある．最後に，通常の eBPF マップの代わりに，ハードウェア割込みハンドラで実行可能な新しいマップを追加した．このマップは，ハードウェア割込みハンドラ内からユーザー空間のアプリケーションに対して，データを見せることを可能にする．マップの一貫性を保証するために，single-producer/single-consumer なリングバッファを用いて実装を行った．

3.4 ユーザープログラムのロードと実行

図 4 を用いて，ユーザープログラムのロードと実行について説明する．まず，ユーザは C 言語でパケット処理のプログラムを記述する．そのプログラムは，Clang/LLVM によって eBPF バイトコードにコンパイルされ，eBPF Verifier によって安全性を検証される．安全性が保証されたものは，eBPF JIT Compiler によってバイナリにコンパイルされて，ハードウェア割込みハンドラにロードされる．ロードされたユーザープログラムは，パケットの送信，ドロップ，マップへの書き込みといった処理を実行する．以上のように，本システムではハードウェア割込みハンドラ内でのパケット処理が行える．

3.5 実装

本システムの実装を Linux カーネル 4.18.8 に行った．実装は主に Intel 40 Gigabit Linux driver（i40e driver）と eBPF に対して行われた．また，本システムの動作には XDP を動作させることが必須である．この理由は 3.2 節で説明したように，実装に XDP の関数を利用しているからである．

4. 評価

本システムがパケット処理のテイル・レイテンシ削減に有効であることを示すために評価を行う．実験環境は 2.4 節で説明した環境と同じである．ただし，本システムは Linux カーネル 4.18.8 に実装しているため，本システムの評価の際にはカーネルを入れ替えている．評価を行うために 2 つの測定を行った．1 つ目は 2.4 節で使用したエコーアプリケーションによるラウンドトリップタイムの測定である．これは，本システムのテイル・レイテンシ削減に対する効果を確認するものである．2 つ目は マップ操作を実行するエコーアプリケーションによるラウンドトリップタイムの測定である．これは，本システムで実装したマップがテイル・レイテンシを発生させないことを確認するものである．

4.1 エコーアプリケーションの評価

まず，システムに負荷がない状態でベースラインとなるラウンドトリップタイムを測定した．その結果，本システムは XDP と同等のレイテンシの分布であった．この理由は，システムに負荷がない状態では，ハードウェア割込みハンドラの実行直後に，ソフトウェア割込みハンドラが実行されるためである．そのため，どちらの割込みのコンテ

キストでパケットを処理しても性能は変わらない．

次に，システムに I/O 負荷がある状態で測定を行った．その結果は図 5 に示されている．システムに負荷がある状態では，DPDK と XDP はテイル・レイテンシが増加しているの対して，本システムはテイル・レイテンシを削減していることがわかる．本システムの 99.9 perceintile は 27.7us であり，DPDK と比較して 67.6%，XDP と比較して 74.9% の削減に成功している．また，本システムの 100 percentile は 57.2us であり，他システムの 99.9 percentile より小さい値である．

4.2 マップ操作を含めたエコーアプリケーションの評価

マップを使用したエコーアプリケーションによるテイル・レイテンシを測定した．アプリケーションは，パケットを受信するとペイロードの値をマップに書き込み，パケットの返信を行う．マップに書き込まれたデータは，ユーザー空間のアプリケーションから読むことができる．比較のために，本システムと eBPF マップを使用した XDP を測定した．XDP アプリケーションでは配列タイプの eBPF マップを使用している．

システムに負荷がない状態では，本システムと XDP は同等の分布であった．本システムで実装したマップは eBPF マップと同等のオーバーヘッドであると言える．システムに I/O 負荷をかけた状態の結果は図 6 に示されている．本システムはマップを使用した場合でもテイル・レイテンシを削減していることがわかる．本システムの 99.9 percentile は 29.8us であり，XDP と比較して 69.4% の削減に成功している．また，100 percentile では本システムが 37.8us，XDP が 213.6us であり，本システムが安定したレイテンシでマップを使用できることを示している．

以上の評価から，本システムはハードウェア割込みハンドラ内でパケット処理を行うことで，テイル・レイテンシの削減に成功していることがわかる．また，テイル・レイテンシを発生させることなく，マップを用いたユーザー空間へのデータ転送が可能であることが示された．

5. 関連研究

5.1 テイル・レイテンシ

テイル・レイテンシは，サービスの応答性能に影響を与える要因や，システムの障害発生の要因となっている．MittOS[13] は，システムのリソースがビジー状態である場合に，ストレージからのデータ読み出しが遅延することで，NoSQL のリクエスト応答にテイル・レイテンシが発生していることを指摘している．MittOS では，OS がリソースを監視することで，SLO に対応した I/O インターフェースを提供している．SafeTimer[17] は，ハートビートのようにタイムアウトを用いたプロトコルにおいて，テイル・レイテンシが要因となり，false positive が発生することを指摘している．SafeTimer では，ソフトウェアにおいてハートビートが遅延しても，false positive を発生させない仕組みを提案している．本論文では，ネットワーク I/O で発生するテイル・レイテンシを削減するため，タイムアウトを用いたプロトコルへの応用が可能である．

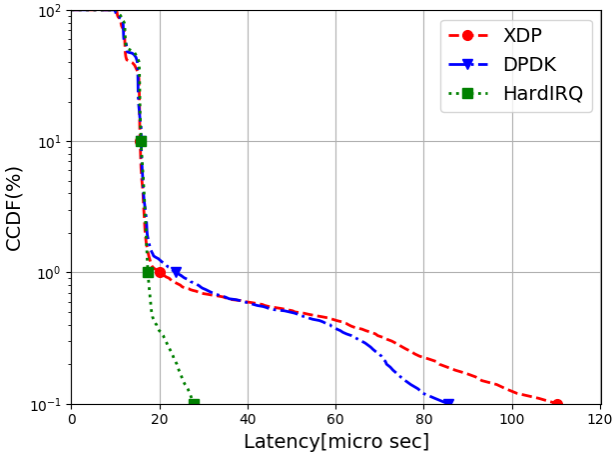

図 5 本システムを含めたパケット往復時間の CCDF （負荷あり）

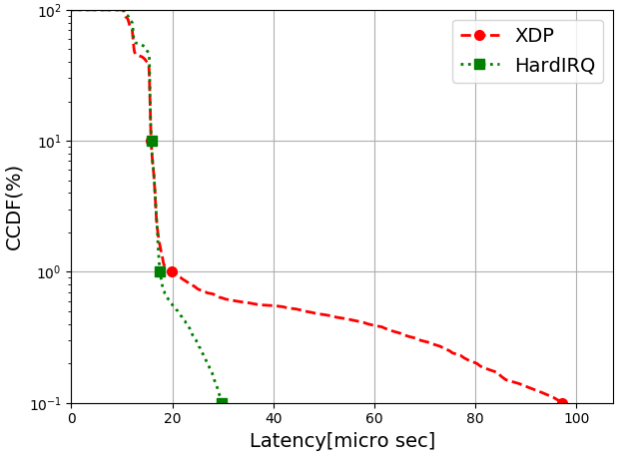

図 6 マップ使用時の本システムと XDP のパケット往復時間の CCDF （負荷あり）

5.2 高速なパケット処理技術

高速なパケット処理を行う技術を利用した研究は盛んに行われている．DPDK は，ユーザーレベルネットワークスタック [15]，[16] や，ネットワークデータプレーンなシステム [18] の基盤として利用されている．XDP は，Linux カーネルと互換のあるパケット処理技術として，ソフトウェアルータ，DDoS 攻撃へのフィルタ，ロードバランサなどの多様なユースケースに対応する [14]．Smart NIC と呼ばれるプログラマブル NIC は，汎用的な処理を NIC にオフロードすることを可能にした．そして，Smart NIC を利用したネットワークアプリケーションの高速化 [20] や，

smart NIC をアクセラレータとしたシステムアーキテクチャ [21] などの研究が行われている．本論文のシステムでは，パケット処理のプログラムをカーネルにロードしているが，同様に smart NIC にオフロードするシステムに拡張することが可能である．

5.3 安全なプログラム実行環境

eBPF はカーネル内における安全なプログラム実行を可能にする．eBPF はパケットフィルタリングのための提案され，Linux カーネルで利用されてきた．現在では，サンドボックス機能を提供する seccomp や，カーネルイベントのトレース [1] など幅広い用途で使用されている．さらに，XDP におけるユーザーコードの実行環境にも eBPF が使用されている．このような実績から eBPF は多くの研究に利用されている．ExtFUSE[10] では，FUSE（Filesystem in Userspace）の拡張として，カーネルにリクエストハンドラをオフロードするための機構として利用される．Hyperupcalls[9] では，ゲスト VM とハイパーバイザのセマンティックギャップを解消するために，ゲスト VM からハイパーバイザに対して，リクエストをロードするための機構として利用される．本論文でも，安全なプログラム実行のために eBPF を利用している．その他のアプローチには，NetBricks[19] のように Rust などの安全な言語によるパケット処理実行環境の構築が挙げられる．

6. まとめ

分散システムにおける障害の発生要因の 1 つとして，パケット処理のテイル・レイテンシがある．この遅延は，ソフトウェアにおいて発生しており，カーネルからアプリケーションまで様々な要因がある．既存のパケットを高速に処理するアプローチでは，パケット処理の高速化に成功しているが，テイル・レイテンシの削減については課題を残している．本論文では，カーネル内においてパケット処理のテイル・レイテンシを削減するシステムを提案した．具体的には，ハートウェア割込みハンドラ内に安全にユーザーコードをロードし，パケット処理を可能にした．本システムにおいて，パケットのエコープログラムを記述し，ラウンドトリップタイムを測定することで，パケット処理のテイル・レイテンシを確認した．その結果，I/O 負荷がある状態で DPDK や XDP といった既存の技術と比較して，パケット処理のテイル・レイテンシを削減できることを確認した．

謝辞 本研究は JST，CREST，JPMJCR19F3 の支援を受けたものである．

参考文献

[1] : BPF Compiler Collection, https://www.iovisor.org/technology/bcc.
[2] : Data Plane Development Kit, https://www.dpdk.org.
[3] : DataNode Lifeline Protocol: an alternative protocol for reporting DataNode liveness, https://issues.apache.org/jira/browse/HDFS-9239.
[4] : eXpress Data Path, https://www.iovisor.org/technology/xdp.
[5] : extended Berkeley Packet Filter, https://www.iovisor.org/technology/ebpf.
[6] : Prevent NN's unrecoverable death spiral after full GC, https://issues.apache.org/jira/browse/HDFS-9107.
[7] : Scaling in the Linux Networking Stack, https://www.kernel.org/doc/Documentation/networking/scaling.txt.
[8] : stress-ng, https://kernel.ubuntu.com/~cking/stress-ng.
[9] Amit, N. and Wei, M.: The Design and Implementation of Hyperupcalls, *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, USENIX Association, pp. 97–112 (online), available from ⟨https://www.usenix.org/conference/atc18/presentation/amit⟩ (2018).
[10] Bijlani, A. and Ramachandran, U.: Extension Framework for File Systems in User space, *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, USENIX Association, pp. 121–134 (online), available from ⟨https://www.usenix.org/conference/atc19/presentation/bijlani⟩ (2019).
[11] Dean, J. and Barroso, L. A.: The Tail at Scale, *Commun. ACM*, Vol. 56, No. 2, p. 74–80 (online), DOI: 10.1145/2408776.2408794 (2013).
[12] Golestani, H., Mirhosseini, A. and Wenisch, T. F.: Software Data Planes: You Can't Always Spin to Win, *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, New York, NY, USA, Association for Computing Machinery, p. 337–350 (online), DOI: 10.1145/3357223.3362737 (2019).
[13] Hao, M., Li, H., Tong, M. H., Pakha, C., Suminto, R. O., Stuardo, C. A., Chien, A. A. and Gunawi, H. S.: MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface, *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, Association for Computing Machinery, p. 168–183 (online), DOI: 10.1145/3132747.3132774 (2017).
[14] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D. and Miller, D.: The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel, *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, New York, NY, USA, Association for Computing Machinery, p. 54–66 (online), DOI: 10.1145/3281411.3281443 (2018).
[15] Jeong, E., Wood, S., Jamshed, M., Jeong, H., Ihm, S., Han, D. and Park, K.: mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems, *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, USENIX Association, pp. 489–502 (online), available from ⟨https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong⟩ (2014).

[16] Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A. and Anderson, T.: TAS: TCP Acceleration as an OS Service, *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3302424.3303985 (2019).

[17] Ma, S. and Wang, Y.: Accurate Timeout Detection Despite Arbitrary Processing Delays, *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, USENIX Association, pp. 467–480 (online), available from ⟨`https://www.usenix.org/conference/atc18/presentation/ma-sixiang`⟩ (2018).

[18] Ousterhout, A., Fried, J., Behrens, J., Belay, A. and Balakrishnan, H.: Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads, *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, USENIX Association, pp. 361–378 (online), available from ⟨`https://www.usenix.org/conference/nsdi19/presentation/ousterhout`⟩ (2019).

[19] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S. and Shenker, S.: NetBricks: Taking the V out of NFV, *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, USENIX Association, pp. 203–216 (online), available from ⟨`https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda`⟩ (2016).

[20] Phothilimthana, P. M., Liu, M., Kaufmann, A., Peter, S., Bodik, R. and Anderson, T.: Floem: A Programming System for NIC-Accelerated Network Applications, *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, USENIX Association, pp. 663–679 (online), available from ⟨`https://www.usenix.org/conference/osdi18/presentation/phothilimthana`⟩ (2018).

[21] Tork, M., Maudlej, L. and Silberstein, M.: Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, New York, NY, USA, Association for Computing Machinery, p. 117–131 (online), DOI: 10.1145/3373376.3378528 (2020).