GPUDirect RDMA を用いた高信頼な障害検知機構

金本 颯将[1]　光来 健一[1]

概要：従来，様々な障害検知手法が提案されてきたが，高信頼・高性能・汎用性の 3 つを満たすのは難しかった．そこで，信頼性の高いホワイトボックス監視として，監視対象ホストに搭載された GPU を用いて障害検知を行う GPUSentinel が提案されている．GPUSentinel では，GPU 上で動作する監視システムがメインメモリ上の OS データを解析し，VRAM にグラフィックデータを直接書き込むことで障害情報を通知する．しかし，画面に出力できる情報量には限界があるため，詳細な障害情報を通知するには OS の通信機能を利用する必要がある．そのため，OS に障害が発生すると障害情報を通知することができなくなる可能性がある．本稿では，OS を介さずに GPU と直接ネットワーク通信を行い，詳細な障害情報をリモートホストに通知することができるシステム *GRASS* を提案する．GRASS では，リモートホストが GPUDirect RDMA を用いて GPU メモリに直接アクセスすることにより，監視対象ホストの CPU を用いずに通信を行う．リモートホスト上のリモート監視システムと GPU 上の OS 監視システムは GPU メモリに対してポーリングを行うことで同期をとる．我々は CUDA や Verbs API を用いて GRASS を実装し，様々な障害を検知できる OS 監視システムを開発した．実験により，障害発生時にリモートホストにおいて監視対象ホストの障害情報を取得できることを確認した．

1. はじめに

近年，システムが複雑化しているのに伴い，システム障害を避けるのが難しくなってきている．システム障害が発生すると提供しているサービスが停止し，サービス提供者は大きな損失を被る．例えば，2018 年の Amazon Prime Day では 1 時間の障害により 7,200 万ドルが失われたと言われている [1]．このようなシステム障害はできるだけ迅速かつ正確に検知する必要があり，従来，ブラックボックス監視やホワイトボックス監視による様々な検知手法が用いられてきた．しかし，ブラックボックス監視では監視対象システムの内部情報にアクセスすることができないために詳細な情報が得られず，ホワイトボックス監視では監視システムが障害の影響を大きく受けてしまうという問題があった．

そこで, 信頼性の高いホワイトボックス監視として, GPU を用いて障害検知を行う GPUSentinel [2] が提案されている．GPUSentinel は CPU から物理的に隔離された GPU 上で監視システムを実行し，メインメモリ上の OS データを解析することで障害検知を行う．障害情報は VRAM にグラフィックデータを直接書き込むことでシステム管理者に通知する．しかし，画面に出力できる情報量には限界があるため，詳細な障害情報を通知したり，インタラクティブに情報を取得したりするのは難しい．そのため，OS の通信機能を利用する必要があるが，OS が異常停止すると障害情報を通知することができなくなる可能性がある．

本稿では，OS を介さずに GPU と直接ネットワーク通信を行い，障害に関する詳細な情報をリモートホストに通知することができるシステム *GRASS* を提案する．GRASS は GPUDirect RDMA [3] を用いることで，リモートホスト上で動作するリモート監視システムと，監視対象ホスト内の GPU 上で動作する OS 監視システムとの間での直接通信を可能にする．リモート監視システムが RDMA Write を用いて要求を GPU メモリに書き込むと，OS 監視システムはポーリングを行うことでそれを受け取る．OS 監視システムが検知情報を GPU メモリに書き込むと，リモート監視システムは RDMA Read を用いてポーリングを行うことでそれを取得する．CPU やメインメモリを介さずに通信を行うため，GPU と NIC が正常に動作していれば通信が可能であり，障害情報の通知がシステム障害の影響を受けにくい．

我々は CUDA [4] や Verbs API を用いて GRASS を実装した．GPU 上の OS 監視システムは GPUSentinel を用いて動作させ，メインメモリ上の OS データに基づいて障害検知を行うようにした．SHFH [5] で提案されているメトリクスを取得する OS 監視システムを開発し，障害情報をリモート監視システムに通知できるようにした．実験の結果，監視対象ホストにおいて OS が異常停止しても，リモート監視システムと OS 監視システムとの間で通信が行えることを確認した．また，カーネルレベルおよびプロセ

[1] 九州工業大学
Kyushu Institute of Technology

スレベルで様々な障害を発生させ，リモート監視システムにおいて短時間で障害検知を行うことができることを確認した．プロセスレベルの障害の場合には原因となったプロセスを特定できることも確認した．

以下，2 章では従来の障害検知手法および GPUSentinel について述べる．3 章では監視対象ホストの OS 等を介さずに直接 GPU と通信を行い，詳細な障害情報を取得するシステム GRASS を提案する．4 章で GRASS の実装について述べ，5 章で GRASS の有効性を確かめるための実験について述べる．6 章で関連研究について述べ，7 章で本稿をまとめる．

2. 障害検知

2.1 従来の障害検知手法

システム障害による損失を減らすには，障害から早期に復旧することが重要となる．そのためには，システム障害の発生後にできるだけ早く障害を検知することが必要である．可能であれば，サービスが完全に停止する前にシステム障害の兆候を検知することが望ましい．また，システム障害をできるだけ正確に検知することも必要である．システム管理者が障害の種類や原因を特定できなければ，次回のシステム障害を防ぐことができないためである．誤検知を起こすと，実際には発生していないシステム障害の原因を調べるためにサービスを停止させなければならなくなる可能性もある．

一般に，システム障害を検知する際には，ブラックボックス監視またはホワイトボックス監視のいずれかの手法が用いられる．ブラックボックス監視では，外部の監視システムを用いて監視対象ホストの OS やサービスに定期的にハートビートを送ることで OS やサービスの状態を監視する．そのため，システム障害の影響を受けることなく対象のシステムを監視することが可能であり，応答がなければシステム障害が発生していると判断することができる．一方で，外部の監視システムは監視対象システムの内部情報にアクセスすることができないという制限がある．そのため，システム管理者は何らかの障害が発生したことまでは把握できるが，監視対象システムの詳細な情報が得られず，多くの場合，障害の原因までは特定することができない．

ホワイトボックス監視では，監視対象システムの内部で動作する監視システムを用いることでシステムの状態を監視する．そのため，システム内部の情報にアクセスすることができ，詳細な情報に基づいて障害検知を行うことが可能である．加えて，障害が発生した際に詳細な情報を取得することで，システム管理者は障害が発生した原因を容易に特定することができる．一方で，ホワイトボックス監視では監視システムが対象システムの内部にあるため，システム障害の影響を大きく受けてしまうという問題がある．例として，OS が停止した場合，その上で動作する監視システムも停止してしまい，システムの監視や外部への通知が行えなくなる．

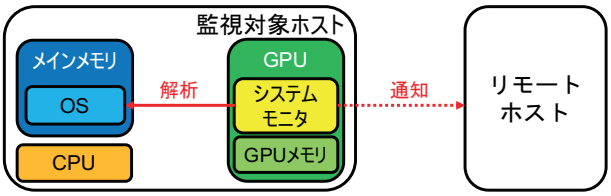

図 1: GPUSentinel のシステム構成

2.2 GPUSentinel

そこで，高信頼なホワイトボックス監視を可能にする GPUSentinel [2] が提案されている．GPUSentinel のシステム構成を図 1 に示す．GPUSentinel では，監視対象ホストに搭載された GPU 上で監視システムが GPU を占有して自律的に動作する．GPU 上の監視システムはメインメモリ上の OS データを解析することで，正確に障害検知を行うことができる．GPUSentinel はメインメモリに格納されたデータから特定できる，ソフトウェアに関する障害を検知可能である．例えば，メモリなどのリソースの不足によるシステム障害やスピンロックによるデッドロックなどを検知することができる．

GPU を用いることで，高信頼・高性能・汎用性の 3 つを満たすことが可能である．GPU は OS が動作する CPU やメインメモリから物理的に隔離されており，監視対象システムのソフトウェア障害による影響を受けにくいため，障害検知の信頼性は高い．また，GPU は多数の演算コアを有しており，それらを用いることで複数の監視システムを並列に動作させることができる．加えて，それぞれの監視システムで実行される検知処理を並列化することで高速化が可能である．その上，GPU は多くの計算機に標準的に搭載されており，GPUSentinel のために専用で用いる GPU を用意することも容易である．

GPUSentinel では，検知した障害をシステム管理者に通知するために，GPU 上の監視システムがメインメモリ上の VRAM 領域にグラフィックデータを直接書き込むことで画面に文字や画像を出力する．画面に出力された内容は，IPMI [6] のリモートコンソール機能や KVM スイッチを用いることで，ネットワーク経由で取得することができる．しかし，メインメモリ上に VRAM 領域を確保する GPU が必要となる上，画面に出力できる情報量には限界がある．そのため，詳細な検知情報をシステム管理者に通知するには OS の通信機能を利用する必要がある．この場合，従来の障害検知手法と同様に，OS の異常停止により外部に検知情報を通知することができなくなる可能性がある．信頼性を向上させるためには，OS に依存しない通信を行う必要があるが，GPU はネットワーク通信を行う機能を持っていない．

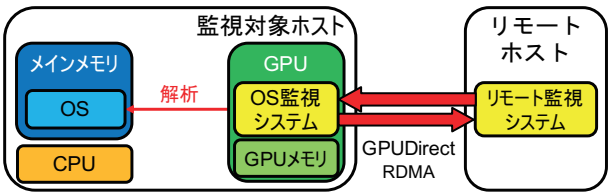

図 2: GRASS のシステム構成

3. GRASS

本稿では，OS を介さずに GPU と直接ネットワーク通信を行うことができるシステム *GRASS* を提案する．GRASS のシステム構成を図 2 に示す．GRASS は GPUDirect RDMA [3] を用いることで，リモートホスト上で動作するリモート監視システムと，監視対象ホスト内の GPU 上で動作する OS 監視システムの間での直接通信を可能にする．GPUDirect RDMA はリモートホストから CPU を介さずに GPU メモリに直接アクセスするためのハードウェア機構である．GPUDirect 機構を用いて GPU メモリを物理メモリアドレス空間にマッピングし，NIC の RDMA 機構を用いることでマッピングした GPU メモリに対してリモートホストから直接アクセスすることができる．

GRASS を用いてリモートホストと監視対象ホストが GPU メモリを介した通信を行うことにより，障害の種類や原因に関する詳細な情報をリモートホストに通知することができる．また，リモートホスト上でインタラクティブに監視対象システムの状態を調べることもできる．GPU 上の OS 監視システムが障害情報を GPU メモリに格納すると，リモート監視システムは GPU メモリから直接，その情報を取得する．そのため，監視対象ホストにおいて OS に障害が発生した場合でも GPU，NIC および OS 監視システムが正常に動作していれば，リモートホストから障害情報を取得することができる．また，リモート監視システムから OS 監視システムに対してハートビートを送信することで，GPU，NIC および OS 監視システムが正常に動作していることを確認することも可能である．

リモート監視システムが障害検知の要求を行う際には，RDMA Write を実行して監視対象ホストの GPU メモリに要求を書き込む．それまでの間，OS 監視システムは要求の書き込みをポーリングによりチェックし続ける．ポーリングを行うのは，RDMA Write の完了を GPU に通知するハードウェア機構が存在しないためである．要求を受信した OS 監視システムは GPU メモリに障害情報を格納する．一方，リモート監視システムは障害情報の取得を行うために，GPU メモリに対して繰り返し RDMA Read を実行することでポーリングを行う．GPU メモリに障害情報が格納されたことを確認すると，ポーリングを終了して障害情報を取得する．

GRASS では，OS 監視システムが GPUSentinel を用いてメインメモリ上の OS データを解析して障害検知を行い，障害の原因まで特定してリモート監視システムに通知する．その際に，**インライン検知**および**バックグラウンド検知**の 2 種類を用いることができる．インライン検知では，OS 監視システムはリモート監視システムからの要求を受信した際に障害検知処理を実行する．要求された検知処理のみを行うため，GPU やネットワークへの負荷を最小限に抑えることができるが，検知に時間がかかる場合には応答時間が長くなる．一方，バックグラウンド検知では，OS 監視システムが定期的に障害検知処理を実行し，検知結果を GPU メモリに格納しておく．リモート監視システムからの要求を受信した際には格納しておいた検知結果を返すだけで済むため応答性に優れているが，GPU や監視対象システムへの負荷は大きくなる．

GPU ですべての障害検知処理を行う手法以外に，GPU，リモートホスト，ネットワークの負荷を考慮して，OS 監視システムとリモート監視システムの間で障害検知処理の分担割合を柔軟に調整することも可能である．1 つ目の例は，OS 監視システムが解析した OS データをリモート監視システムに送信し，リモート監視システムにおいて障害検知および原因の特定を行う手法である．2 つ目の例は，OS 監視システムは要求されたメモリ領域のデータを取得・送信するだけで，リモート監視システムにおいて OS データの解析を行う手法である．3 つ目の例は，OS 監視システムが OS カーネルのメモリ全体をリモート管理システムに一括送信し，リモート管理システムにおいて OS データの解析を行う手法である．障害検知と原因の調査とで異なる手法を用いることも可能である．

4. 実装

我々は CUDA 8.0 [4]，Verbs API，RDMA コネクションマネージャを用いて GRASS を実装した．GPUDirect RDMA を利用できるようにするために，Mellanox OFED 4.1 および nvidia-peer-memory 1.0.7 を用いた．また，監視対象ホストの OS として GPUSentinel 向けに修正した Linux 4.4.64 を動作させ，OS 監視システムのコンパイルには GPUSentinel で用いられている LLVM 5.0.0 を用いた．

4.1 GPUDirect RDMA を用いた通信

リモートホストから監視対象ホストに対して RDMA 通信を行えるようにするために，それぞれのホストで Verbs API を用いて保護領域を作成する．そして，保護領域の中に RDMA 通信の端点となるキューの組を作成する．保護領域を用いることで，外部からの RDMA アクセスの範囲を制限することができる．また，送受信の完了を知るために完了キューを作成する．その後で，監視対象ホストからリモートホストに対してコネクションを確立する．

監視対象ホストではリモートホストとの RDMA 接続を

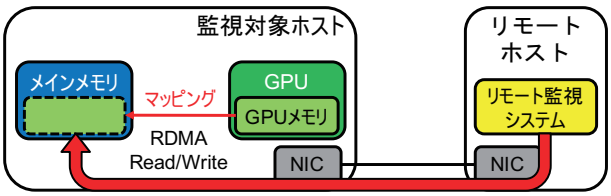

図 3: GPUDirect RDMA 機構

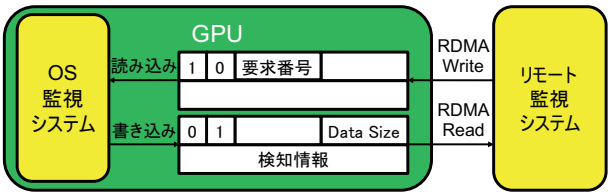

図 4: GPU との通信の流れ

確立した後，図 3 のように GPU メモリ上に GPUDirect RDMA に使用するバッファを確保し，Verbs API を用いてメモリ領域を作成する．そのバッファのアドレスを RDMA Send を用いてリモートホストに送信する．GPU メモリ上のバッファはプロセスの統合仮想アドレス空間にマッピングされており，指定した仮想アドレスは NVIDIA ドライバによって物理アドレスに変換される．同時に，ホストチャネルアダプタによって生成されるリモートキーも送信する．リモートキーは RDMA Read/Write の際に使われ，値が一致しない場合にはアクセスが拒否される．

RDMA Write を用いることで，リモートホストは監視対象ホストの GPU メモリにデータを書き込む．リモートホストは書き込むデータが格納されたメインメモリのアドレスとサイズおよび，書き込み先の GPU メモリのアドレスを指定する．一方，RDMA Read を用いることで，リモートホストは監視対象ホストの GPU メモリからデータを読み込む．リモートホストは読み込むデータが格納された GPU メモリのアドレスとサイズおよび，読み込み先のメインメモリのアドレスを指定する．リモートホストは完了キューを用いることで RDMA Read/Write の完了通知を受け取る．

4.2 障害情報の要求・取得

GPU メモリ上に確保した RDMA 用バッファをヘッダ領域とデータ領域に分割する．ヘッダ領域は書き込み完了フラグ，読み込み許可フラグ，要求番号，データサイズから構成される．書き込み完了フラグと読み込み許可フラグは RDMA 通信を行うホスト間で同期をとるために用いられる．要求番号には OS 監視システムで行う障害検知処理に割り振られた番号を格納し，データサイズにはデータ領域に書き込まれた障害情報のサイズを格納する．

GPU とリモートホストの通信の流れを図 4 に示す．OS 監視システムに要求を送信する際に，リモート監視システムは RDMA Write を用いて要求番号を書き込む．その後で，再度 RDMA Write を用いて書き込み完了フラグをセットし，要求の送信完了を OS 監視システムに通知する．このとき，OS 監視システムは GPU の演算コアを 1 つ専有し，ポーリングを用いて書き込み完了フラグがセットされるまでチェックし続ける．GPU は多数の演算コアを持つため，その 1 つを専有しても検知性能への影響はほとんどないと考えられる．OS 監視システムは書き込み完了フラグがセットされたことを検知すると，要求番号を取得し，書き込み完了フラグをクリアする．

受信した要求番号に従って，OS 監視システムはインライン検知を行って検知結果をデータ領域に格納するか，直近のバックグラウンド検知の結果をデータ領域に格納する．障害検知情報のデータサイズを格納した後に読み込み許可フラグをセットすることで，検知情報の送信準備の完了をリモート監視システムに通知する．それまでの間，リモート監視システムはポーリングを用いて読み込み許可フラグに対して繰り返し RDMA Read を行う．これは GPU からリモートホストに対して RDMA Write を実行することができないためである．なお，一定の間隔をあけてポーリングを行うことで，リモートホストの CPU 負荷を下げることができる．読み込み許可フラグがセットされたことを検知すると，リモート監視システムは RDMA Read でデータ領域から障害情報を取得し，再度 RDMA Write を行うことで読み込み許可フラグをクリアする．

4.3 OS 監視システムに対する障害の検知

リモート監視システムから OS 監視システムに対して定期的にハートビートを送信することで，GPU，NIC および OS 監視システムが正常に動作していることを確認する．ハートビートの手順は通常の障害情報の要求・取得とほぼ同じであるが，OS 監視システムはハートビートのための要求番号を受信するとすぐに読み込み許可フラグをセットする．読み込み許可フラグがセットされたことをリモート監視システムが検知すると，OS 監視システムの正常動作の確認を完了する．

OS 監視システムからの応答がない場合，OS 監視システム自体に障害が発生した場合と GPU や NIC に障害が発生した場合の 2 種類の障害が考えられる．前者の場合には，リモート監視システムは RDMA Write を用いて書き込み完了フラグをセットするものの，OS 監視システムはそれを検出して読み込み許可フラグをセットすることができない．そのため，リモート監視システムは読み込み許可フラグがセットされるのをタイムアウトするまで待って，OS 監視システムの障害を検知することができる．後者の場合には，GPU や NIC に異常が発生しているために，リモート監視システムは書き込み完了フラグをセットすることができない．この場合，リモート監視システムは RDMA エラーを受け取ることになり，GPU や NIC の障害を検知す

表 1: SHFH におけるシステム障害の分類

障害	内容		
F1	無限ループ	割り込み禁止	
F2		割り込み許可	プリエンプション許可
F3			プリエンプション禁止
F4	無期限待機	リソースが解放されない	
F5		リソースが	ロックを保持してスリープ
F6		ゆっくり解放	異常なリソース消費

表 2: SHFH に基づく障害検知

障害	CPU			プロセス			メモリ		I/O
	sys	usr	iowait	run	blk	cs	pswpout	memfree	util
F1	○	○				○			
F2	○	○							
F3	○	○				○			
F4			○		○				
F5	○	○				○			
F6							○	○	

ることができる．

4.4 OS に対する障害検知

GPUSentinel で実装された 3 種類の障害検知に加え，SHFH [5] で提案されているメトリクスに基づいた障害検知も行えるようにした．SHFH では，システムを停止させる障害は表 1 に示す 6 種類に分類されている．無限ループか無期限待機かによって大きく 2 つに分類され，無限ループでは割り込みやプリエンプションの許可・禁止によって 3 種類，無期限待機ではリソースの解放状況によって 3 種類に分類される．例えば，GPUSentinel でも実装されているスピンロックによるデッドロックは F1 に分類される．

SHFH では，分類した 6 種類のシステム障害を検知するために 9 つのメトリクスを提案している．CPU 関連のメトリクスとして，カーネル（sys），プロセス（usr），入出力待ち（iowait）で消費された CPU の使用率が用いられる．メモリ関連のメトリクスとして，空きメモリ量（memfree）およびスワップアウト回数（pswpout）が用いられる．ディスク関連のメトリクスとして，ディスクの入出力処理に使われた CPU 時間（util）が用いられる．スケジューリング関連のメトリクスとして，実行可能状態（run）および待ち状態（blk）のプロセス数とコンテキストスイッチ回数（cs）が用いられる．

これらのメトリクスを表 2 のように組み合わせて障害検知を行う．CPU に関するメトリクスの閾値として，sys は全 CPU が 95%以上，usr は全 CPU が 1%以下，iowait は GRASS のプロセスが使う 1 つの CPU を除いて 90%以上とした．プロセスに関するメトリクスの閾値として，blk は 32 以上，cs は 1 秒あたり 350 回以下に設定した．メモリに関するメトリクスの閾値として，pswpout は 1 秒あたり 3000 回以下，memfree は空きメモリが 256MB 以下とした．現在の実装では run と util は利用していない．

4.5 OS 監視システム

GPUSentinel を用いて Linux のカーネルメモリからこれらのメトリクスの値を取得して障害検知を行う OS 監視システムを実装した．CPU のシステム時間，ユーザ時間，入出力待ち時間は CPU ごとに Linux の kcpustat_cpu マクロを用いて kernel_cpustat 構造体から取得する．GPU 上で clock64 レジスタを読み出すことによって 1 秒間ビジーウェイトして CPU 時間の増分を取得し，それぞれの CPU 使用率を算出する．空きメモリ量は si_meminfo 関数を用いて取得し，スワップアウト回数は各 CPU における vm_event_state 構造体の PSWPOUT イベントの回数を取得して 1 秒間の増分を算出する．実行可能状態および待ち状態のプロセス数は nr_running 関数および nr_iowait 関数を用いて取得し，コンテキストスイッチ回数は nr_context_switches 関数を用いて取得して 1 秒間の増分を算出する．なお，ディスクの入出力処理に使われた CPU 時間に関しては実装方法を検討中である．

これらの関数や構造体は GPUSentinel が提供している LLView フレームワークを用いて実装した．LLView は Linux のカーネルモジュールのように記述した GPU プログラムがメインメモリにアクセスしながら動作することを可能にする．そのためには，OS データの仮想アドレスを物理アドレスに変換し，変換した物理アドレスをさらに GPU アドレスに変換する必要がある．LLView は GPU プログラムを LLVM [7] を用いてコンパイルし，生成された中間表現を変換することで透過的なアドレス変換を実現する．メモリからデータを読み込む際に実行される load 命令の直前にアドレス変換を行うコードを挿入することで，変換されたアドレスに対して load 命令を実行させる．また，中間表現で使われている OS の大域変数は対応する仮想アドレスに置換する．

GPU からメインメモリの内容を参照できるようにするために，GPUSentinel が提供しているメモリ管理機構を用いた．GPUSentinel の Linux カーネルが提供する /dev/pmem を一旦，プロセスアドレス空間にマップし，それを CUDA のマップトメモリ機構を利用して GPU アドレス空間にマップする．これはメインメモリを直接，GPU アドレス空間にマッピングすることができないためである．/dev/pmem を用いることで，メインメモリ全体を GPU アドレス空間にマッピングする際にメインメモリ全体がピン留めされて使用中になってしまうのを防ぐことができる．OS に障害が発生する前にメインメモリを GPU アドレス空間にマッピングしておくことで，障害が発生した後も GPU 上の OS 監視システムはメインメモリの情報を参照することができる．また，メインメモリ全体のサイズのメモリをマップトメモリで利用できないという CUDA の制限を回避するために，sysinfo システムコールをフックし，メインメモリのサイズとして少し大きな値を返すようにした．

表 3: 監視対象ホスト

OS	Linux 4.4.64
CPU	Intel Xeon E5-1603 v4
メモリ	DDR4-2400 8GB
GPU	NVIDIA Quadro M4000
NIC	Mellanox MCX455A-ECAT ConnectX-4 VPI
ソフトウェア	NVIDIA Graphics Driver 375.66 CUDA 8.0, Mellanox OFED 4.1 nvidia-peer-memory 1.0.7

表 4: リモートホスト

OS	Linux 4.10.0
CPU	Intel Xeon E3-1270 v3
メモリ	DDR3-1600 8GB
NIC	Mellanox MCX455A-ECAT ConnectX-4 VPI
ソフトウェア	Mellanox OFED 4.1

5. 実験

GRASS の有用性を確かめるための実験を行った．監視対象ホストおよびリモートホストにはそれぞれ表 3 および表 4 のマシンを用いた．これらのホストは 100 ギガビットイーサネットで接続した．

5.1 OS 障害発生時の GRASS の動作確認

監視対象ホストの OS が異常停止した際に，OS 監視システムとリモート監視システムとの間で正常に通信が行えることを確認する実験を行った．そのために，監視対象ホストで/proc/sysrq-trigger に”c”を書き込み，カーネルパニックを発生させた．実験の結果，監視対象ホストにおいてカーネルパニックが発生しても，OS 監視システムとリモート監視システムとの間で通信が行えることを確認した．

5.2 ハートビートの性能

リモート監視システムから GPU，NIC および OS 監視システムの異常を検知するために送信するハートビートにかかる時間を計測した．比較として，リモートホストから監視対象ホストへの ping にかかる時間の計測も行った．ハートビートと ping をそれぞれ 1 秒間隔で 100 回繰り返し，応答時間の平均および標準偏差を算出した．

計測結果を図 5 に示す．ハートビートでは，リモート監視システムが要求を書き込み，書き込み完了フラグをセットするまでにかかる時間が 48 μs であった．その後，GPU 上の OS 監視システムによる読み込み許可フラグのセットを確認できるまでの時間が 27 μs であった．ping では応答が返ってくるまでの時間は 95 μs であった．標準偏差はハートビートで 9 μs，ping で 24 μs であった．実験結果から，GRASS では十分に短い時間で安定したハートビートを行うことができることが確認できた．

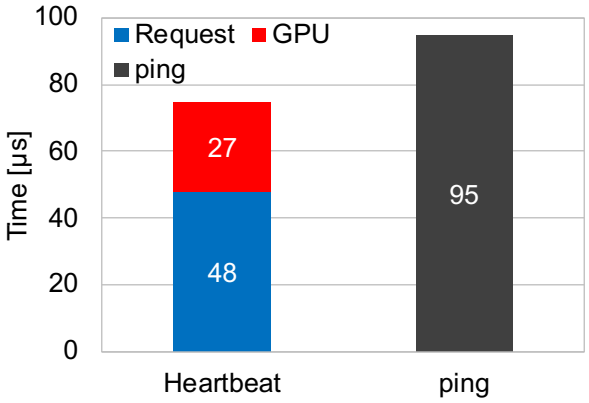

図 5: ハートビートの性能

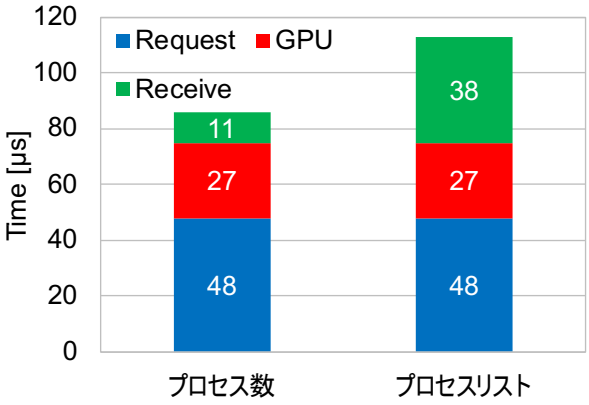

図 6: プロセス情報の取得性能

5.3 プロセス情報の取得性能

監視対象ホストのプロセス情報を取得するための要求を送信し，情報を受信するのにかかる時間を計測した．プロセス数のみを取得する要求およびプロセスリストを取得する要求の 2 種類を用いた．それぞれの要求を 1 秒間隔で 100 回繰り返し，取得時間の平均および標準偏差を算出した．監視対象ホストのプロセス数は 155 であった．

計測結果を図 6 に示す．どちらの要求もリモート監視システムが書き込み完了フラグをセットするまでにかかる時間および GPU 上の OS 監視システムによる読み込み許可フラグのセットを確認できるまでの時間は同じであった．バッファに書き込まれた情報を受信するのにかかる時間はプロセスリストのほうが 27 μs 長かった．通信時間の標準偏差はいずれも 12 μs であった．実験結果から，障害情報として取得する情報が増えると RDMA Read にかかる時間が長くなることが分かった．

5.4 プロセスレベルの障害の検知

監視対象ホストで軽度の障害を発生させるプログラムを実行してからリモート監視システムで検知できるまでの時間を計測した．OS 監視システムには GPUSentinel で実装された CPU 高負荷の検知とメモリリークの検知を用いた．CPU 高負荷の検知プログラムはすべての CPU の使用率

```
[Command: cpuhighload
 Status = CPU_HIGH_LOAD(1)
 pid:2776 comm:yes usage:100.00
 pid:2775 comm:yes usage:100.00
 pid:2774 comm:yes usage:100.00
 pid:2773 comm:yes usage:100.00
```

(a) CPU 高負荷

```
[Command: memoryleak
 Status = MEMORY_LEAK(2)
 totalram  : 8063996 [KiB]
 totalswap : 8302588 [KiB]
 freeram   : 101868 [KiB]
 freeswap  : 1703588 [KiB]
 pid:3014 comm:mem_leak
```

(b) メモリリーク

図 7: 障害検知情報の出力例

が 90%以上の状態が 5 秒継続すると障害と判定する．メモリリークの検知プログラムは空きメモリ量と空きスワップ量がどちらも 30%を下回った時に障害と判定する．この実験では，CPU と同じ数の yes コマンドを実行することで CPU を高負荷にし，メモリを確保し続けるプログラムを実行することでメモリリークを発生させた．リモート監視システムは 1 秒ごとに要求を OS 監視システムに送り，障害のインライン検知を行った．それぞれの障害の検知を 10 回ずつ行い，検知時間の平均および標準偏差を算出した．

障害検知時にリモート監視システムが出力した障害情報を図 7 に示す．CPU 高負荷の検知時には，原因となったプロセスの ID と名前および CPU 使用率がリモート監視システムに通知された．メモリリークの検知時には，メモリに関する情報と最もメモリを多く消費しているプロセスの ID と名前が通知された．計測結果から，CPU 高負荷の検知にかかる時間は平均 6.6 秒，標準偏差は 0.3 秒となり，メモリリークの検知にかかる時間は平均 68.6 秒，標準偏差は 0.7 秒となった．メモリリークの検知に時間がかかったのは，スワップ領域を 70%消費するのに時間がかかったためである．

5.5 カーネルレベルの障害の検知

監視対象ホストの Linux カーネルに障害を挿入してからリモート監視システムで検知できるまでの時間を計測した．SHFH において分類された 6 種類の障害を発生させるために作成したカーネルモジュールを表 5 に示す．F4 については，カーネルモジュールのロード後に作成されたキャラクタデバイスに対して多数のプロセスを用いて読み込みを行うプログラムを実行した．リモート監視システムは 1 秒ごとに要求を OS 監視システムに送り，同時に 6 種類の障害に対するインライン検知を行った．それぞれの障害の検知を 10 回ずつ行い，検知時間の平均および標準偏差を算出した．

計測結果を図 8 に示す．実験結果から，いずれの障害についてもリモート監視システムは 2.2 秒程度で検知できることが分かった．一方，他の障害と比べて F4 の検知時間のばらつきが少し大きいことも分かった．これは入出力待ちではなくアイドル状態となっている CPU があったことが原因であるが，その原因は現在調査中である．

表 5: 障害を発生させるカーネルモジュール

障害	内容
F1	全 CPU でスピンロックによるデッドロック
F2	全 CPU で無限ループ
F3	全 CPU で無限ループ（プリエンプション禁止）
F4	読み込み時にブロックするキャラクタデバイスを提供
F5	スピンロック後にスリープ，全 CPU がスピンロックを待機
F6	大量のメモリを確保

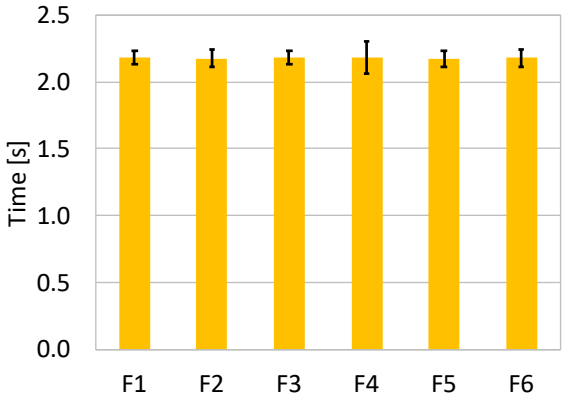

図 8: SHFH に基づく障害の検知性能

6. 関連研究

GPUnet [8] は GPU プログラムに対してソケットと高水準ネットワーク API を提供する．同じマシンか異なるマシンかに関わらず，GPU 内スレッドが他の GPU または CPU のスレッドと通信することができる．リモートホストが GPU プログラムにデータを送信する際には，GPUDirect RDMA を用いて GPU メモリに対して直接データを書き込む．しかし，GPU プログラムがリモートホストにデータを送信する際には，CPU 経由で RDMA Write を実行するため GPU だけでは通信を行うことができない．そのため，OS に障害が発生するとリモートホストに障害情報を通知することができなくなる．

GPU だけで通信を行えるようにしたシステムも提案されている [9]．このシステムは Infiniband の設定を CPU 上で行い，Infiniband のリソースを GPU 上にコピーする．さらに，MMIO アドレスを GPU アドレス空間にマッピングすることで，GPU プログラムが自律的にリモートホストにデータを送信することができる．しかし，GPU ドライバ，ネットワークドライバ，ライブラリに変更を加える必要がある．GRASS では GPU プログラムからデータの送信を開始することはできないが，これらのシステムソフ

トウェアへの改変は必要ない．

セキュリティの向上を目的としてハードウェアを用いたシステムが数多く提案されており，そのいくつかは信頼性の高い障害検知にも適用可能であると考えられる．Copilot [10] は PCI カード上の ARM 評価ボードを用いて監視対象システム外部から OS カーネルの整合性を監視する．PCI カードは DMA を使ってカーネルメモリのデータを取得し，そのハッシュ値を計算してリモートホストに送信する．GPU と同様に，PCI カードはシステム障害の影響を受けにくい．一方，GPU と異なり，PCI カード自身がリモートホストと通信を行う機能を持つため，リモートホストとの通信は容易に行える．しかし，専用の PCI カードが必要となるため汎用性の面で課題がある．

Intel 製 CPU のシステム管理モード（SMM）を用いて安全に監視を行うシステムも提案されている．HyperGuard [11] は SMM でハイパーバイザの整合性チェックを行う．通常モードでの実行ではアクセスできないメモリ上に SMM 用のプログラムが置かれるため，監視プログラムが障害の影響を受けにくい．しかし，SMM での実行は低速であり，実行中は監視対象システム全体が停止するという問題がある．また，監視対象システムと同じ CPU を用いるため，障害の影響を受ける可能性がある．SMM で実行できるプログラムは BIOS の一部であるため，様々な監視プログラムを実行させるのも容易ではない．一方，HyperCheck [12] は SMM でネットワークドライバのみを実行し，メモリ全体のデータをリモートホストに転送して監視を行う．このシステムでは様々な監視プログラムを動かすのが容易であるが，NIC に合わせて様々なデバイスドライバを実装する必要がある．

HyperSentry [13] は IPMI [6] を用いて監視対象ホストと通信し，SMM を利用してハイパーバイザの監視を行う．SMM での実行時に割り込みを禁止し，ハイパーバイザ内に送り込んだ監視プログラムを安全に実行する．IPMI 用の NIC を用いるため，通信が監視対象システムの障害の影響を受けない．しかし，監視プログラムは CPU の通常モードで実行されるため，監視対象システムの障害の影響を受ける可能性がある．

ソフトウェアのみで障害検知を行うシステムも提案されている．本稿で参考にした SHFH [5] は，リアルタイム・プロセスとカーネルモジュールを用いて実装される．通常の状態ではリアルタイム・プロセスがシステムの監視を軽量に行い，システム障害の兆候を検出するとカーネルモジュールがシステムの状態をより詳細に検査する．カーネルレベル障害検知機構 [14] は検知精度を向上させるために，定期的にカーネル内で障害検知処理を行う．これらのシステムはカーネルタイマに依存しているため，タイマ割り込みが禁止の状態で障害が発生すると，障害を検知することができない．Falcon [15] は OS，ハイパーバイザ，ネットワークスイッチ等の様々な階層で監視プログラムを実行し，下位層から上位層の障害を検出する．障害検知にはハートビートを用いるため，詳細な障害情報を取得することはできない．

監視対象システムが仮想マシン（VM）で動作している場合，VM イントロスペクション [16] と呼ばれる手法を用いて VM の外から OS データを取得することができる．この手法を障害検知に用いることで，監視システムがシステム障害の影響を受けにくくすることができる．しかし，VM を用いないシステムに適用したり，ハイパーバイザの障害を検知したりすることはできない．GRASS は仮想化されていないシステムに対しても信頼性の高い障害検知を行うことを可能にする．

7. まとめ

本稿では，OS を介さずに GPU と直接ネットワーク通信を行い，障害に関する詳細な情報をリモートホストに通知することができるシステム GRASS を提案した．GRASS は GPUDirect RDMA を用いることで，リモートホスト上で動作するリモート監視システムと，監視対象ホスト内の GPU 上で動作する OS 監視システムの間での直接通信を可能にする．リモート監視ホストは RDMA Read/Write を用いて GPU メモリを読み書きし，ポーリングを用いて OS 監視システムとの同期をとることで障害情報の要求・取得を行う．CUDA や Verbs API を用いて GRASS を実装し，GPUSentinel を用いてメインメモリ上の OS データを解析することで様々な障害を検知することができる OS 監視システムを開発した．実験結果から，Linux カーネルに障害を挿入してもリモート監視システムに障害情報が通知されることが確認できた．

今後の課題は，SHFH で提案されているメトリクスの内，ディスクの入出力処理に使われた CPU 時間も取得できるようにし，より精度の高い障害検知を行えるようにすることである．また，OS 監視システムが障害検知を行う手法だけでなく，リモート監視システムがデータを取得して障害検知を行う手法も実装する予定である．その際には，データの転送量が増大するため，並列転送による転送性能の向上も検討している．加えて，より幅広い種類の障害が発生した際にも監視対象ホストとの通信を維持し，情報を取得することができるかどうかを確認する必要がある．また，監視対象ホストで障害が発生した際にメインメモリ上の OS データを書き換えることで障害からの部分的な復旧を行うことができないかも検討している．

謝辞　本研究成果の一部は，国立研究開発法人情報通信研究機構の委託研究により得られたものです．

参考文献

[1] Digital Commerce 360. The Potential Cost of Amazon's Prime Day Miss? $72 Million. https://www.digitalcommerce360.com/2018/07/17/the-potential-cost-of-amazons-prime-day-miss-72-million/, 2018.

[2] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai. Detecting system failures with gpus and llvm. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 47–53, 2019.

[3] NVIDIA Corporation. Developing a Linux Kernel Module Using RDMA for GPUDirect. Technical Report TB-06712-001 v10.1, NVIDIA, 2019.

[4] NVIDIA Corporation. CUDA Toolkit Documentation v8.0. https://docs.nvidia.com/cuda/archive/8.0/.

[5] Y. Zhu, Y. Li, J. Xue, T. Tan, J. Shi, Y. Shen, and C. Ma. What is System Hang and How to Handle it. In *Proceedings of International Symposium on Software Reliability Engineering*, pages 141–150, 2012.

[6] Intel, Hewlett-Packard, NEC, and Dell. Intelligent Platform Management Specification Second Generation v2.0, 2004.

[7] The LLVM Foundation. The LLVM Compiler Infrastructure. https://llvm.org/.

[8] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 201–216, 2014.

[9] L. Oden, H. Fröning, and F. Pfreundt. Infiniband-Verbs on GPU: A Case Study of Controlling an Infiniband Network Device from the GPU. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 976–983, 2014.

[10] N. Petroni, Jr., T. Fraser, J. Molina, and W. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of USENIX Security Symposium*, 2004.

[11] J. Rutkowska and R. Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions. Black Hat USA, 2008.

[12] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: A Hardware-assisted Integrity Monitor. In *Proceedings of International Symposium Recent Advances in Intrusion Detection*, pages 158–177, 2010.

[13] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 38–49, 2010.

[14] 岩間響子, 毛利公一, 齋藤彰一. 多様な障害へ対応したカーネルレベル障害検知機能の提案と実装. In **情報処理学会研究報告**, volume 2016-OS-136, 2016.

[15] J. Leners, H. Wu, W. Hung, M. Aguilera, and M. Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 279–294, 2011.

[16] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 191–206, 2003.