システム外部からのOSメモリの書き換えによるシステム障害からの復旧

木村　健人[1]　光来　健一[1]

概要：近年の大規模かつ複雑なシステムにおいてシステム障害を完全に回避するのは困難である．したがって，障害発生時には迅速かつ正確に障害を検知して復旧を行うことが重要である．システム障害からの復旧を行う際には，管理者がシステムにリモートログインして作業を行ったり，対象システム内のプロセスやカーネルを用いて動作する復旧システムが自動復旧を行ったりする．しかし，これらの手法はシステム障害の影響を受けやすいため，復旧が行えない場合にはハードウェアリセットが必要となり，システムのデータや状態が失われる可能性がある．本稿では，障害の影響を受けにくいGPU上の復旧システムがOSを間接的に制御することでシステム障害からの復旧を行うGPUfasを提案する．GPUfasでは，GPUからメインメモリ上のOSデータを書き換え，OS自身の機能を用いて復旧を試みる．例えば，プロセスにシグナルを疑似送信することにより，障害の原因となったプロセスを強制終了させることができる．我々はLinuxとCUDA，LLVMを用いてGPUfasを実装し，GPUからメインメモリ上のOSデータを透過的に書き換えられるようにした．VM内のシステムについても，VMイントロスペクションを拡張して障害からの復旧を行えるようにした．さらに，カーネル内に組み込んだ復旧支援機構をGPUやVM外部から呼び出せるようにした．そして，GPUfasを用いてプロセスレベル障害からの復旧の有効性と性能について調べる実験を行った．

1. はじめに

システム障害の原因として，ソフトウェアの不具合，性能・容量不足，設定・操作ミスなどが挙げられる．ソフトウェアの不具合や性能・容量不足はシステム開発者が十分に留意するべき点ではあるが，近年の大規模かつ複雑なシステムにおいてはこれらを回避するのは難しい．実際に，AWSでは2020年にサーバ内で動作するスレッド数がOSの限界値を超えるという障害が発生し，何千ものオンラインサービスに影響を与えた[1]．仮に，完璧なシステムを構築できたとしてもシステム管理者の操作ミスなどによりシステム障害を引き起こす場合もある．東京証券取引所の株式売買システムにおいて部品故障時にバックアップ機への自動切り替えを行う設定がオフになっていたため，2020年に終日取引を行えなくなる障害が発生した[2]．システム障害が発生するとサービスの利用者や提供者は大きな損失を被るため，迅速に障害を検知して復旧を行うことが重要になる．

システムに障害が発生したことを検知すると，管理者がシステムにリモートログインして復旧作業を行うことが多い．しかし，リモートアクセスはネットワーク障害やシステムのリソース不足などの影響を受けやすい．そこで，復旧システムをあらかじめ対象システム内で動作させておき自動復旧を行う手法も用いられている．例えば，復旧システムをプロセスとして実行したり[3]，カーネル内に組み込んだり[4][5]する方法があるが，対象システム内で動作するため障害の影響をまだ受けやすい．これらの方法で復旧できない場合にはハードウェアリセットによって復旧するしかないが，システムのデータや状態が失われてしまう可能性がある．

本稿では，可能な限りハードウェアリセットを回避するために，対象システムに搭載されているGPU上で復旧システムを動作させ，OSの挙動を間接的に変更することで障害からの復旧を行うGPUfasを提案する．GPUfasはGPUからメインメモリ上のOSデータを書き換え，OS自身の機能を用いて障害の原因を自動的に取り除く．障害からの復旧の一例として，GPUから障害の原因となっているプロセスに疑似的にシグナルを送信することで，プロセスを強制終了させることができる．GPUはシステム障害の影響を受けにくいため，GPUfasは復旧システムが正常に動作する可能性を高くすることができる．

我々はGPUfasをLinuxとCUDA，LLVMを用いて実

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

装した．Linux のメモリ管理機構を拡張して CUDA のマップトメモリ機構を用いることで，GPU からメインメモリ全体を書き換えられるようにした．VM 内のシステムの障害からの復旧も行えるようにするために，KVMonitor[6] を拡張して GPUfas-VM を実装した．また，GPU からメインメモリ上の OS データを容易に参照可能にする LLView[7] を拡張して，プロセスへのシグナル疑似送信とプロセスの疑似スケジューリングを実装した．さらに，カーネル内に復旧支援機構を実装し，GPU や VM 外部から呼び出せるようにした．実験により，GPUfas がプロセスレベル障害からの復旧できることを確認し，復旧にかかる時間の測定を行った．

以下，2 章ではシステム障害から復旧するために用いられている手法について述べる．3 章では，GPU からメインメモリ上の OS データを書き換えることでシステム障害からの復旧を行う GPUfas を提案する．4 章では GPUfas の実装について述べ，5 章で GPUfas の有効性を確かめるために行った実験について述べる．6 章では関連研究について述べ，7 章で本稿をまとめる．

2. システム障害からの復旧

システム障害を検知した後には，迅速に障害からの復旧を行う必要がある．一般的には，管理者が障害の発生したホストに SSH などを用いてリモートログインを行い，障害からの復旧を試みることが多い．この手法の利点は管理者が障害の状況を確認し，復旧の手法を選択できる点である．しかし，この手法には障害の影響を大きく受けるという欠点がある．例えば，システムのネットワーク機能が停止するとリモートログインすることができなくなる．システムのメモリが不足するとスラッシングが発生し，リモートログインやログイン後のリモートアクセスに時間がかかる．

管理者がリモートログインせずに復旧を行う手段として，システム内部で自動復旧システムを動作させておく手法がある．この手法では，障害の影響を受けやすいリモートアクセスを用いないため，より迅速な復旧が見込める．この内部復旧の例として，復旧システムをプロセスとして動作させる手法が用いられている [3]．システム内部で動作するプロセスが定期的に動作してシステムの状態をチェックし，障害を検知した際には復旧を行う．この手法の利点は，新しい復旧方法を容易に適用することができることである．しかし，プロセスから行える復旧作業は限られており，復旧システムがまだ障害の影響を大きく受けるという欠点がある．

OS カーネル内に復旧システムを組み込むことも提案されている [4][5]．この手法では一般に，タイマ割り込みを用いて定期的に復旧システムを実行し，障害が検知されたら復旧を行う．この手法の利点は，プロセスとして実行する復旧システムに比べて障害の影響を受けにくく，様々な復旧手法を実装することができることである．しかし，復旧システムをカーネルモジュールとして組み込む場合には利用できる変数や関数が制限されるため，実装できる復旧手法が限られてしまう．カーネルのすべての機能を利用するにはカーネル本体に復旧システムを組み込む必要があるため，新しい復旧手法の適用が難しい．また，障害によってタイマ割り込みが機能しなくなると復旧システムを実行することができない．

これらの手法を用いて復旧が行えない場合には，ハードウェアリセットによりシステムを再起動することによって復旧を行うことができる．リモートホストからハードウェアリセットを行うために，IPMI やリモート電源管理装置などが用いられる．IPMI はサーバの多くに搭載されており，ネットワーク経由でサーバに搭載された管理用 IC チップにアクセスすることでリセットを行う．リモート電源管理装置はコマンドメールや VPN などを用いてネットワーク経由でサーバの電源を遮断することでリセットを行う．また，ウォッチドッグタイマを用いることで一定時間，応答がない場合にシステムを自動的にリセットをすることも可能である．システムが仮想マシン（VM）内で動いている場合には，ホスト OS と通信して VM を遠隔操作することでリセットすることができる．

しかし，ハードウェアリセットは強力な復旧手法である一方で，システムの状態やデータが失われる危険性が高い．障害の発生したシステムの状態が失われると，システム障害の原因を特定することが困難になる．そのため，再度同じシステム障害が発生したとしてもリセットを行うしかなく，システムの信頼性を低下させる．また，リセット時にメモリ上に存在し保存されていなかったアプリケーションのデータや，ファイルキャッシュに書き込まれただけでディスクに書き戻されていなかったデータも失われる．ファイルシステムへのアクセス中にリセットを行うと不整合が発生してファイルが失われ，復旧できなくなる可能性もある．システムの状態やデータの消失を防ぐために様々な機構が提案されてきた [8][9][10] が，必ずしもすべてのデータの復旧ができるわけでなく，データを修復できたとしてもコストや時間がかかる．

3. システム外部からの障害復旧

3.1 障害モデル

本稿ではプロセスレベルの障害を復旧の対象とする．例えば，プロセスがメモリを使い過ぎることによってシステム全体のメモリが不足したり，プロセスが OS の限界値まで作成されることによって復旧処理を実行できなかったりすることを想定する．そのため，障害発生時にはカーネルは復旧を行える程度には正常に動作していることを仮定する．カーネルレベルでのリソース不足やデッドロックによる無限待機についても復旧を行える可能性はあるが，本稿

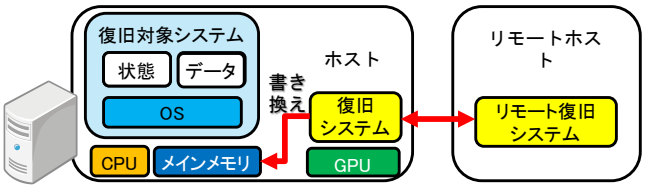


図 1 GPUfas のシステム構成

では対象外とする．クラッシュによるカーネルの異常停止はハードウェアリセット以外では復旧が困難であるため，本研究では復旧の対象としない．

3.2 GPUfas

本稿では，GPU 上で動作させる復旧システムが OS データを変更することでシステム障害から復旧を行う GPUfas を提案する．GPUfas のシステム構成を**図 1** に示す．GPUfas は障害発生前に起動させ，障害を検知した後に復旧システムがメモリ上の OS データを書き換え，OS の機構を用いて障害から復旧を試みる．そのため，カーネル内に組み込んだ復旧システムに近く，様々な復旧手法を実装することができる．復旧システムが GPU 上で動くアプリケーションとして動作するため，新しい復旧手法の適用も容易である．GPU は障害の発生したシステムが動作する CPU やメインメモリから物理的に隔離されているため，GPU 上で動作する復旧システムは障害の影響を受けにくい．また，GPU は専用の演算コアやメモリを持ってるため，復旧対象システムのリソース不足による影響も受けない．

GPUfas は GPUSentinel[7] を用いて障害の検知を行う．監視対象システムに搭載された GPU 上で検知システムを動作させ，メインメモリ上の OS データを監視する．システム障害を検知すると検知システムは OS の詳細情報を取得し，障害の原因となったプロセスを特定する．自動的に復旧を行う際には，GPUfas は特定されたプロセス情報を基に，GPU 内で判断して復旧を行う．GRASS[11] を併用することで OS を介さずに GPUDirect RDMA を用いてリモートホストと通信を行い，対話的に復旧手法を選択することもできる．その場合にはリモートホストに検知結果を送り，管理者の判断を仰ぐことができる．さらには，リモートホストの AI の判断を仰ぐこともできる．

GPUfas による復旧手法の例として，障害を引き起こした異常プロセスへのシグナルの疑似送信が挙げられる．例えば，大量にメモリを使用しているプロセスを終了させることでメモリを解放させ，メモリ不足によってスラッシングが起きないようにすることができる．また，CPU を使い続けているプロセスを一時停止することで CPU 負荷を下げ，システムの性能を回復させることができる．OS 上ではシグナルはシステムコールを用いて送信され，カーネル内ではカーネル関数を呼び出すことによって送信される．しかし，GPU から直接システムコールやカーネル関数を

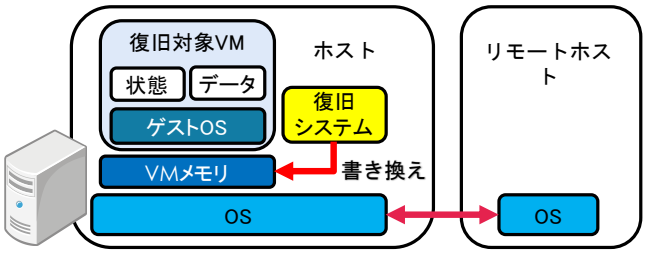


図 2 GPUfas-VM のシステム構成

呼び出すことはできないため，GPU は直接シグナルを送信することができない．そこで，GPUfas はメインメモリ上のカーネルメモリ内にあるプロセス情報を書き換えることでシグナルが送られた状態に変更し，プロセスへのシグナル送信を疑似的に実現する．

また，GPUfas はプロセスの疑似的なスケジューリングを行うこともできる．例えば，疑似送信されたシグナルを即座に処理させることで，高速に復旧を行うことができる．シグナルは送信先のプロセスがスケジュールされるまで処理されないため，シグナルの疑似送信を行うだけでは停止中のプロセスを制御することはできない．GPUfas はプロセススケジューラの情報を書き換えることで，プロセススケジューリングの変更を疑似的に実現する．

このように GPUfas は疑似的に OS の機能を実現することで復旧を行うが，メモリ書き換えだけでは実現が難しい機能もある．例えば，ロックの獲得は CPU のアトミック操作命令が必要となり，GPU には実現できない．GPU は CPU にプロセッサ間割り込みを送ることもできない．また，GPU にも実装できるが，複雑すぎて実装が難しい OS の機能もある．実装できても，メインメモリへのアクセスが多くなると復旧性能が低下してしまう．そのため，GPUfas は必要に応じてカーネル内に組み込んだ復旧支援機構との連携を行う．GPU 上の復旧システムはメインメモリを介してカーネル内の復旧支援機構と通信を行い，カーネル内で処理を実行する．ただし，復旧支援機構は障害の影響を受けやすいため，耐障害性や実装のしやすさ，性能などのトレードオフを考慮する必要がある．

GPUfas を用いたとしても仮復旧しか行えない場合がある．例えば，プロセスを終了させることにより，システムが障害発生前のサービスを提供できなくなる場合がある．この場合，管理者がリモートログインできる状態まで復旧できるのであればログインしてデータを保存し，それからシステムの再起動を行うことでデータの消失を防ぐことができる．仮復旧すら行えない場合には，GRASS を用いて OS のネットワーク機能に頼らずにリモートホストにメモリデータを送信してからハードウェアリセットを行うことができる．リモートホストで受信したメモリデータの解析を行うことで，リセットによってデータが消失したとしても復元することができる．

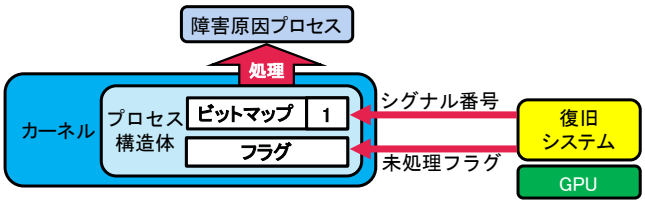


図 3 GPUfas のシグナル疑似送信

3.3 GPUfas-VM

GPUfas-VM は復旧対象システムが VM 内で動作している場合に GPUfas と同様に障害からの復旧を行う．GPUfas-VM のシステム構成を**図 2** に示す．VM 内のシステム障害は基本的に VM 外部に影響を与えないため，GPUfas-VM は GPU を用いずに復旧システムをホスト OS のプロセスとして動作させる．VM 外部で動作する復旧システムは，正常なホスト OS のネットワーク機能を利用してリモートホストと通信しながら復旧を行うことができる．GPUfas-VM は VM のメモリを解析してゲスト OS の情報を取得する VM イントロスペクション [6] を拡張して，VM 外部からゲスト OS のデータを書き換える．

4. 実装

我々は Linux 4.18.0 に修正を加え，CUDA 10.0.130 と LLVM 8.0 を用いて GPUfas を実装した．また，KVMonitor 用に修正した QEMU-KVM 2.11.2 と LLVM 8.0 を用いて GPUfas-VM を実装した．

4.1 シグナルの疑似送信

一般的に OS のシグナル機構を用いる際には，kill コマンドなどが発行するシステムコールを用いてプロセスにシグナルを送信する．プロセスは受信したシグナルに応じてあらかじめ登録しておいた処理またはデフォルトの処理を行ったり，シグナルを無視したりする．システムコール内でシグナルを送信する際には，対象プロセスが持つシグナルビットマップにシグナル番号を記録し，未処理のシグナルがあることを示すためのフラグを対象プロセスにセットする．その後，対象プロセスがスケジュールされてカーネルモードからユーザモードに遷移する際にカーネルがこのフラグをチェックし，未処理のシグナルがあればシグナルの処理を行う．

GPUfas はシステム外部からプロセスへのシグナルの疑似送信を行うために，カーネルがシグナル送信に用いているデータ構造を直接，書き換える．シグナル疑似送信の様子を**図 3** に示す．まず，対象プロセスの task_struct 構造体の中の sigpending 構造体が持つシグナルビットマップ（sigset_t）を見つけ，送信するシグナルの番号に対応するビットを 1 にセットする．次に，対象プロセスの thread_info 構造体を見つけ，TIF_SIGPENDING フラグをセットする．

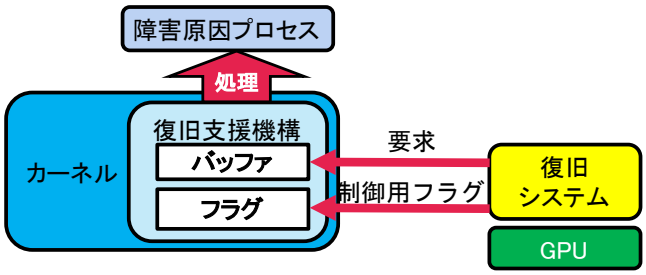


図 4 復旧支援機構への要求書き込み

その後はシステムコールを用いた場合と同様に，カーネルがシグナルの処理を行う．例えば，STOP シグナルの場合はプロセスを一時停止させ，KILL シグナルの場合はプロセスを強制終了させる．

4.2 疑似的なスケジューリング

カーネル内でシグナルの送信処理を行った後，シグナルができるだけ早く処理されるように，カーネルは対象プロセスのスケジューリングを行う．シグナルは受信するプロセスがカーネルモードからユーザモードに遷移する際にのみ処理されるため，プロセスがスリープして待機状態なっていると処理されないためである．プロセスをスケジュールするために，まずそのプロセスをスケジューラの実行キューに追加し，プロセスを実行可能状態に変更する．その後，カーネル内のプロセススケジューラが対象プロセスを実行状態にするとシグナルが処理される．

GPUfas はシステム外部からプロセスの疑似的なスケジューリングを行うために，Linux で一般的に用いられているスケジューラである CFS が用いている赤黒木を直接，書き換える．まず，対象プロセスの task_struct 構造体の中にある sched_entity 構造体を CFS の cfs_rq 構造体の中にある赤黒木に追加する．この時に sched_entity 構造体が持つ仮想実行時間をキーとして適切な位置を探索する．次に，プロセスの状態を TASK_RUNNING に変更する．その後，カーネル内の CFS は仮想実行時間の短いプロセスから順にスケジューリングを行う．現在のところ，疑似スケジューリングは GPUfas-VM にのみ実装できている．

4.3 復旧支援機構

シグナル送信先のプロセスが他の CPU で実行中の場合には，カーネルはプロセスの実行を一旦，中断してカーネルモードに遷移させる．そして，そのプロセスが次にユーザモードに遷移するタイミングでシグナルの処理を行う．GPUfas はプロセスの実行を中断させるために，復旧支援機構に要求を送信し，そのプロセスが実行されている CPU に対して再スケジューリングのためのプロセッサ間割り込み（IPI）を送信する．

復旧支援機構への要求書き込みの様子を**図 4** に示す．カーネル内の復旧支援機構はメインメモリ上に要求を受

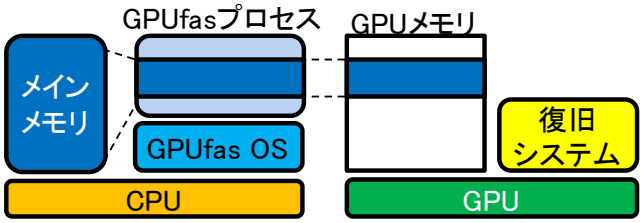


図 5 GPU へのメインメモリのマッピング

信するためのバッファを用意し，GPUfas はそのバッファに要求を書き込むことによって要求を送信する．このバッファには同時に複数の要求を書き込むことができ，GPUfas の書き込み中に復旧支援機構がアクセスしないようにフラグを用いて排他制御を行う．復旧支援機構はタイマ割り込みによって 1 ミリ秒ごとに呼び出され，要求が送信されていれば要求に合わせた処理を実行する．

4.4 GPU からのメインメモリ書き換え

GPUfas では，GPU からメインメモリを書き換えられるようにするために，CUDA が提供するマップトメモリ機能を用いる．マップトメモリは GPU 上のプログラムから参照可能にするために，メインメモリを GPU メモリアドレス空間にマッピングする機能である．障害前にマッピングを行うことによって，障害発生後でもメインメモリを参照することが可能になる．CUDA ではメインメモリを直接 GPU メモリにマッピングすることは不可能であるため，プロセスのアドレス空間を介して GPU のアドレス空間にマッピングする．しかし，そのままメインメモリ全体をマッピングすると全メモリが使用中になるため，システムの空きメモリがなくなってしまう．

この問題を回避するために，GPUfas では GPUSentinel で提案されているメモリ管理機構を Linux 4.18 に移植した．移植したメモリ管理機構の様子を**図 5** に示す．Linux カーネルが/dev/pmem という特殊なデバイスファイルを用意し，GPUfas はこのデバイスファイルを GPUfas プロセスに読み書き可でマッピングする．/dev/pmem は mmap システムコールでマッピングされる時に，メモリページの参照カウンタを増やさないようにしてメモリが使用中にならないようにする．また，CUDA にはメインメモリ全体のサイズより少し小さいサイズしか指定できないという制限があるため，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．

メインメモリ上のデータ構造の書き換えを行う際には，**図 6** のように GPU が自動的にメインメモリと GPU メモリ間で DMA 転送を行う．GPUfas が OS データを書き換える際にはまず，復旧対象システムのページテーブルを用いて仮想アドレスを物理アドレスに変換し，それをさらに GPU アドレスに変換する．変換先の GPU アドレスにアクセスすると，GPU が自動的にメインメモリから GPU メモリに DMA 転送を行う．GPUfas が DMA 転送された OS データを書き換えると，GPU が書き換えたデータを自動的にメインメモリに DMA 転送することで書き換えが反映される．

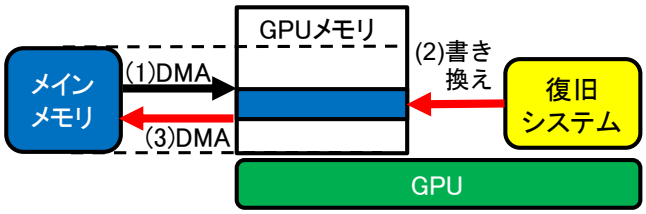


図 6 GPU からメインメモリ上の OS データの書き換え

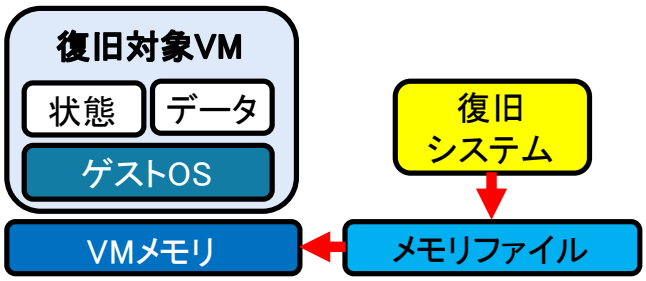


図 7 VM のメモリ上の OS データの書き換え

GPUfas では GPU がメインメモリ全体にアクセスできるようになるため，攻撃者がこの権限を悪用できてしまうと危険である．そのため，GPUfas OS はマッピングしたメインメモリに復旧システムのみがアクセスできるように制限している．GPUfas ではどのプロセスにもメインメモリ全体をマッピングできるが，GPUfas OS はそのプロセスのページテーブルを変更することにより，マッピングしたメモリへのアクセスを禁止する．攻撃者はそのプロセスを乗っ取ってもメインメモリにアクセスすることはできない．また，GPUfas では起動時に実行された復旧システムが GPU を占有するため，攻撃者が自身の GPU カーネルを実行するには復旧システムを終了させる必要がある．GPUfas OS は起動後 1 回だけプロセスへのメインメモリのマッピングを許可するため，攻撃者の GPU カーネルにメインメモリをマッピングするのは難しい．攻撃者は GPU 上で動作している復旧システムを乗っ取ることができればメインメモリにアクセスできるが，GPU カーネルを乗っ取るのは容易ではないと考えられる．

4.5 VM のメモリ書き換え

GPUfas-VM では，VM の外側から VM のメモリを書き換えられるようにするために，KVMonitor[6] の拡張を行った．KVMonitor はホスト OS 上に作成したメモリファイルを QEMU-KVM のメモリアドレス空間にマッピングし，VM のメモリとして利用する．**図 7** のように，このメモリファイルを別のプログラムからアクセスすることで VM

表 1 復旧対象ホスト

CPU	Intel Core i7-9700
メモリ	DDR4-2666 16GB
スワップ領域	7GB
GPU	NVIDIA GeForce GTX 960
OS	Linux 4.18.0
ソフトウェア	NVIDIA GPU driver 430.40 CUDA 10.0.130 LLVM 8.0

表 2 復旧対象 VM

仮想 CPU	3
メモリ	2GB
スワップ領域	4GB
ゲスト OS	Linux 4.18.0
ソフトウェア	QEMU-KVM 2.11.2

表 3 用いた復旧手法の組み合わせ

	GPUfas/ GPUfas-VM		GPUfas-VM のみ	
シグナル疑似送信	○	○	○	○
疑似スケジューリング			○	○
A:カーネル内スケジューリング		○		
B:カーネル内 CPU モード遷移		○		○

イントロスペクションを実現する．従来の KVMonitor は VM のメモリの読み込みのみに対応していたが，VM のメモリ書き換えを実現するためにメモリファイルを読み書き可でマッピングするようにした．

VM の外側で動作する復旧システムが OS データの書き換えを行う際には，まず，ゲスト OS の仮想アドレスを VM 内の物理アドレスに変換する．この変換の際には，QEMU-KVM と通信して CR3 レジスタの値を取得し，ゲスト OS が使っているページテーブルを探索する．次に，VM 内の物理アドレスを復旧システムのプロセスの仮想アドレスに変換してデータを書き込む．

4.6 LLView の拡張

GPUfas において OS データの書き換えを行うにはその度にアドレス変換が必要となる．復旧システムの開発を容易にするために，GPUfas では LLView[7] を拡張することで OS データのアドレス変換を透過的に行う．LLView は OS のソースコードを用いて GPU で動作するプログラムを記述すると，メインメモリ上の OS データを参照するプログラムに変換するツールである．拡張 LLView では LLVM を用いてプログラムのコンパイルを行う際に，中間表現の中でメモリにデータを書き込む際に用いられる store 命令の直前にアドレス変換関数の呼び出しを挿入する．

5. 実験

GPUfas の有用性を確かめるための実験を行った．復旧対象ホストには**表 1** のマシンを用い，VM 内のシステムを復旧する実験ではこのホスト上で**表 2** の VM を動作させた．復旧システムがシグナル疑似送信のみを用いる場合と，さらに復旧支援機構を用いてスケジューリングおよび CPU モード遷移も行った場合について実験を行った．GPUfas-VM については，シグナル疑似送信に加えて疑似スケジューリングも行った場合，および復旧支援機構を用いてさらに CPU モード遷移も行った場合について実験を行った．**表 3** に実験で用いた機構の組み合わせを示す．比較として，復旧システムをプロセスとして動作させた場合（復旧プロセス）と，カーネル内で動作させた場合についても実験を行った．復旧プロセスは kill システムコールを用いて対象プロセスに KILL シグナルを送信した．カーネル内復旧システムはカーネル関数を呼び出して対象プロセスに KILL シグナルを送信した．

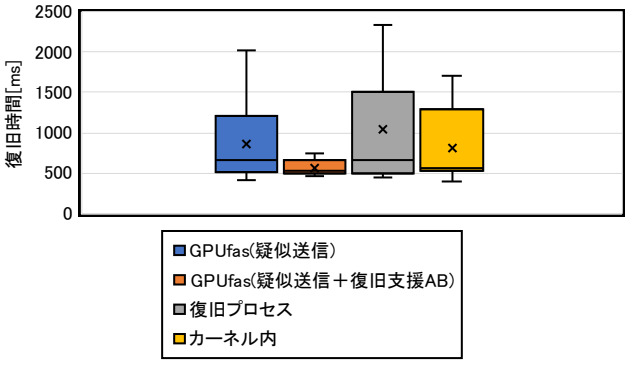


図 8 メモリ不足からの復旧時間

5.1 メモリ不足による障害からの復旧

メモリが不足するとシステムの性能が著しく低下したりログインが円滑に行えなくなったりするため，システム障害となる．そこで，システムのメモリが不足していることを検知し，障害から復旧する実験を行った．メモリ不足による障害を発生させるために，復旧対象ホストのメモリを 19GB 確保するプロセスを 1 つ実行し，スワップを発生させるために常に確保したメモリへの書き込みを行った．その状態で SSH を用いてリモートログインを行うとスラッシングが発生し，ログインに正常時の約 20 倍の時間がかかった．

復旧システムはシステムのメモリ使用量が 80%を超えた時に障害が発生したとみなし，KILL シグナルを送信して対象プロセスを強制終了した．障害発生から復旧完了までにかかった時間を 10 回測定した結果を**図 8** に示す．この結果から，GPUfas は復旧プロセスやカーネル内復旧システムよりも迅速かつばらつきを抑えて復旧が行えることが分かった．しかし，シグナル疑似送信だけでは復旧時間のばらつきが大きく，平均値や中央値も大きくなった．これは，シグナルを疑似送信しただけでは処理されないため，

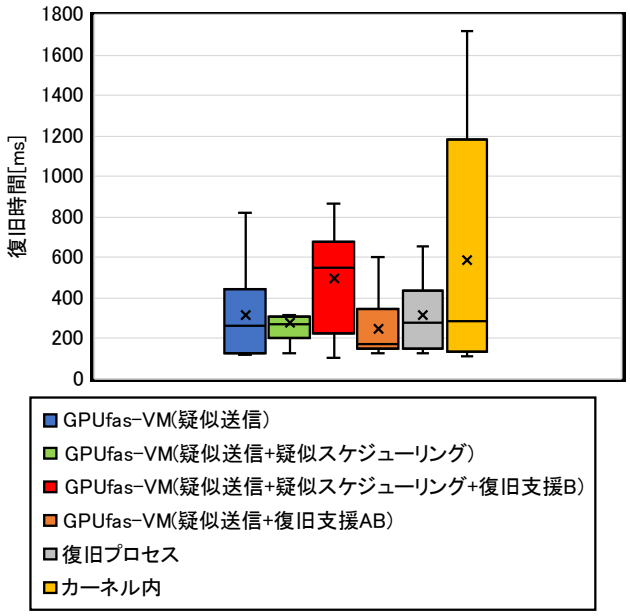


図 9　VM 内のメモリ不足時の復旧時間

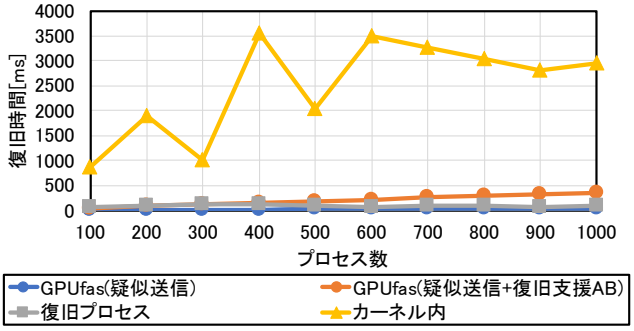
実際にシグナルが処理されたタイミングがばらついたためと推測できる．復旧支援機構を併用することにより，シグナル疑似送信のみを用いた場合よりも復旧時間のばらつきが小さくなり，平均値も 34%小さくなった．復旧プロセスやカーネル内復旧システムにおいて復旧時間が長くなったのは，システムのメモリが不足したことによって復旧システムが大きな影響を受けたためと考えられる．

VM でもメモリを 5GB 確保して同様の実験を行った．その状態でリモートログインを行うと，正常時の約 15 倍の時間がかかった．障害発生から復旧完了までにかかった時間を 10 回計測した結果を**図 9** に示す．この結果から，GPUfas-VM は疑似スケジューリングを行うことでシグナル疑似送信のみよりもばらつきを抑えて復旧できることが分かった．一方，復旧支援機構を併用すると復旧時間のばらつきが大きくなった．このことから復旧支援機構はリソース不足の影響を大きく受けるため，復旧支援機構の利用は最小限にするべきだということが分かった．

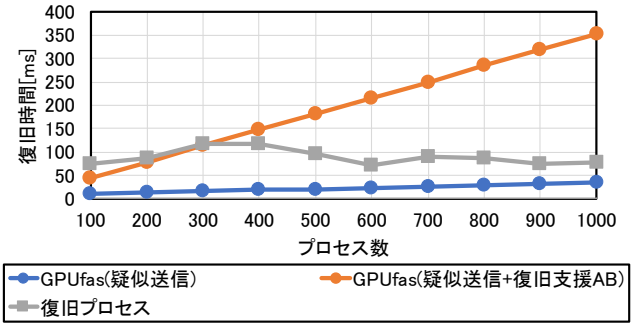
5.2　大量のプロセスへのシグナル疑似送信

GPUfas を用いて大量のプロセスに KILL シグナルを疑似送信してプロセスを強制終了させる性能を調べる実験を行った．この実験では，一定時間スリープして起床する処理を繰り返すプロセスを指定した数だけ実行した．そして，指定した名前と一致するプロセスすべてにシグナルを送信してからすべてのプロセスが終了するまでにかかる時間を計測した．

図 10 は 100〜1000 個のプロセスが 10ms のスリープを繰り返し，10ms に 1 回の頻度で起床することで CPU モードを遷移させた場合の結果である．この場合には，シグナル疑似送信のみを行った場合が最も迅速にシグナルの処



(a) すべての復旧システム



(b) カーネル内復旧システム以外

図 10　10ms ごとに動くプロセスへのシグナル送信

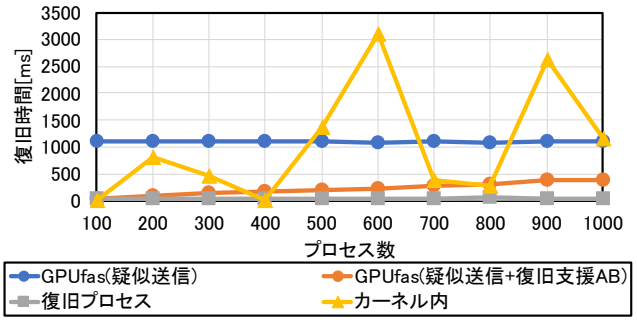
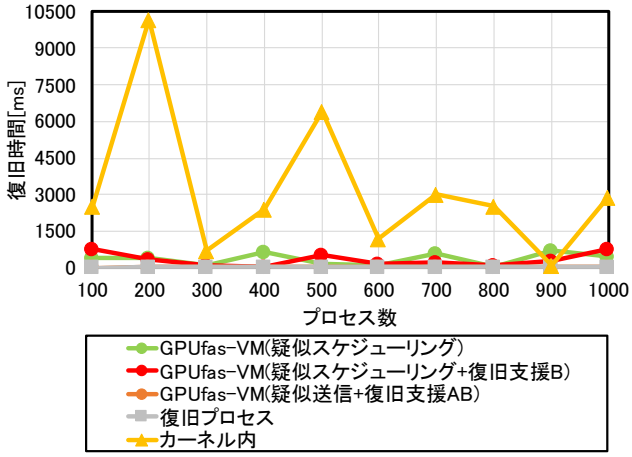


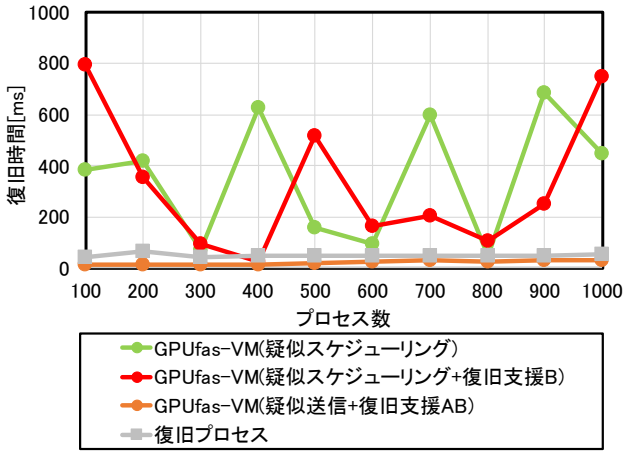
図 11　1 秒ごとに動くプロセスへのシグナル送信

理を行えたことが分かる．これは，復旧支援機構に要求を送ってプロセスをスケジュールする前にプロセスが自発的に起床してシグナルが処理されることが多かったためだと考えられる．プロセス数が 300 までは復旧支援機構を用いる GPUfas のほうが復旧プロセスより高速であったが，プロセス数が増えると復旧プロセスのほうが高速になった．復旧支援機構を用いた場合，プロセス数が増えれば復旧時間も増えていることから，復旧支援機構に要求を送るオーバヘッドが大きいと推測できる．一方，カーネル内復旧システムは復旧に常に長い時間がかかった．この原因は現在調査中である．

図 11 はすべてのプロセスが 1 秒のスリープを繰り返し，1 秒に 1 回だけ起床することで CPU モードを遷移させた場合の結果である．この場合，シグナル疑似送信のみでは GPUfas による復旧には常に 1 秒以上かかることが分かった．これはスリープしているプロセスが自発的に起床する

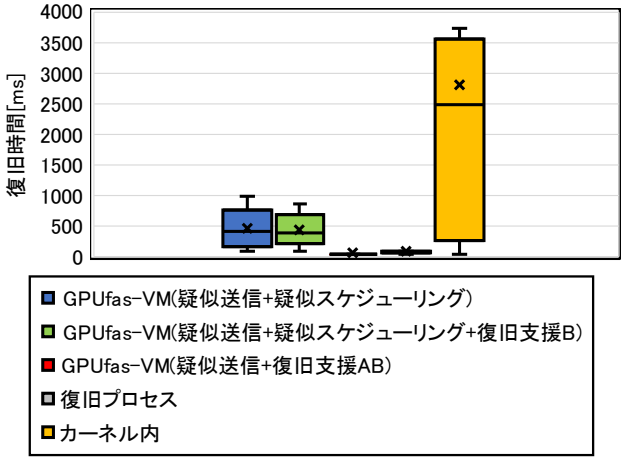


(a) すべての復旧システム

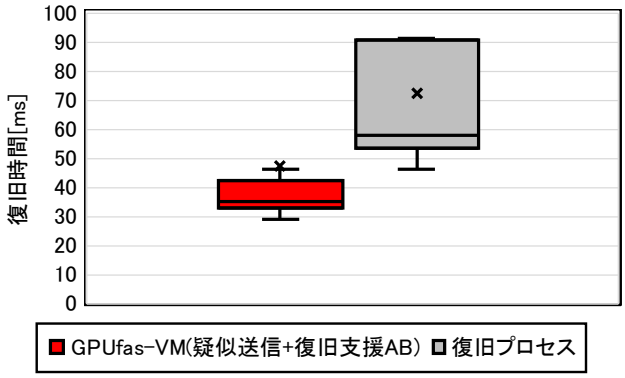


(b) カーネル内復旧システム以外

図 12 VM 上で長時間スリープするプロセスへのシグナル送信



(a) すべての復旧システム



(b) GPUfas の最速の場合と復旧プロセス

図 13 VM 上の 1000 個のプロセスへのシグナル送信

までシグナルが処理されなかったためである．復旧支援機構を用いた場合には，プロセスをスケジュールすることでシグナルをより短い時間で処理することができ，大幅に復旧時間を短縮することができた．復旧プロセスを用いた場合のほうが復旧時間が短くなったが，障害が発生して復旧プロセスがその影響を受けると復旧時間は長くなると考えられる．一方，カーネル内復旧システムは復旧時間のばらつきが非常に大きかった．この原因は現在調査中である．

5.3 疑似スケジューリングの性能

疑似スケジューリングは GPUfas-VM にしか実装できていないため，GPUfas-VM を用いて疑似スケジューリングの性能を調べる実験を行った．この実験では，指定した数のプロセスを長時間スリープさせることにより，自発的な CPU モードの遷移が起こらないようにした．そして，5.2 節と同様にしてプロセスに KILL シグナルを送信して復旧時間を測定した．**図 12** は，100〜1000 個のプロセスに対する復旧時間である．また，**図 13** は動作させるプロセス数を 1000 に固定して 10 回計測した結果である．GPUfas-VM においてシグナル疑似送信のみを用いた場合は，プロセスが起床するまでシグナルが処理されず，復旧時間が非常に長くなったため省略した．

これらの結果から，対象プロセスが長時間のスリープを行っていても疑似スケジューリングを行うことによって復旧時間を短くできたことが分かる．しかし，疑似スケジューリングを行った場合，復旧支援機構に要求を送ってカーネル内でスケジューリングを行った場合と比べて大幅な性能の低下および復旧時間のばらつきの増大が見られた．これは，現在の疑似スケジューリングの実装がまだ不完全であることが原因だと考えられる．復旧支援機構にプロセスのスケジューリングを行わせた場合には，復旧プロセスを用いる場合よりも迅速かつ小さなばらつきで復旧が行えている．そのため，疑似スケジューリングを用いる場合もさらに復旧時間を短縮できることが期待できる．

6. 関連研究

Linux カーネルは障害からの復旧を行うために oops や OOM Killer と呼ばれる機能を提供している．oops はカーネル内でのエラーを検知した時にエラーの原因となったプロセスを強制終了し，システムの実行を継続する．しかし，

発生したエラーによってはカーネルの状態を正常に戻せるとは限らない．OOM Killer はメモリが不足してシステムが停止する恐れがある時に，メモリを多く消費しているプロセスを強制終了する．OOM Killer は深刻なメモリ不足になるまで実行されず，プロセスが使っているメモリやスワップ領域以外の要素は考慮されない．GPUfas はより柔軟に終了させるプロセスを選択することができる．

SHFH [4] は，様々なシステムハングを検知して障害からの復旧を行う．3 つの復旧手法が用いられており，障害の原因と考えられるプロセスやスレッドの強制終了または一時停止，ストールした CPU への NMI の送信，システムの再起動を行う．SHFH は主にカーネル内で障害検知と復旧を行うため障害の影響を受けやすく，GPU 上で復旧システムを動作させる GPUfas より信頼性が低い．

Backdoors[12] は，リモートホストから RDMA を用いて OS データを書き換えることでシステム障害からの復旧を行う．復旧の例として，GPUfas と同様に，プロセスに KILL シグナルを疑似送信する手法が用いられている．しかし，RDMA でプロセステーブルにアクセスできるように復旧対象の OS を改変する必要がある．それに対して，GPUfas では既存の OS を用いることができる．また，Backdoors はリモートホストから直接 OS メモリへのアクセスを許可する必要があるため，新たな脆弱性になり得る．一方で，GPUfas では GPU があらかじめ決められた復旧のみを行うためより安全である．GPUfas を GRASS[7] と組み合わせることでリモートホストからの復旧が可能になるが，この場合でも GPU があらかじめ決められた復旧のいずれかを行うだけであるため，Backdoors のような危険性はない．

Exterior[6] は VM 内のシステムが攻撃を受けた時に VM 外部から回復を行うことを可能にする．Exterior は対象 VM と同一のカーネルを動かす別の VM を用意し，その VM 内で実行したコマンドによるメモリ更新を対象 VM に反映する．この機構を用いて kill コマンドを実行してプロセスを強制終了させたり，rmmod コマンドを実行してカーネルモジュールをアンロードしたりすることができる．この手法は VM 内で動作するシステムのみに利用できるが，GPUfas は VM を用いていないシステムにも利用できる．

Otherworld [10] は，カーネル内で障害が発生した時にカーネルをマイクロリブートする．通常の再起動とは異なり，マイクロリブートは実行中のアプリケーションの状態を破壊せずに再起動を行う．再起動後に，障害発生時に実行していたアプリケーションのメモリ空間，開いていたファイル，およびその他のリソースの状態を復元する．GPUfas を用いてシステムの復旧が行えない場合に，再起動の影響を最小限に抑えるために用いることができる．

VM を用いたフェーズベース・リブート [13] はカーネル障害からの復旧時間を短縮する．フェーズベース・リブートではブートシーケンスをフェーズに分割し，起動時の状態を 3 つのフェーズに分けて保存しておく．システム復旧時には最適なフェーズの状態に戻すことにより，再起動にかかる時間を最小限に抑える．ただし，障害発生時に保存されていなかったシステムの状態は失われる．

7. まとめ

本稿では，システム障害の影響を受けにくい GPU 上で復旧システムを動作させ，OS データを変更することでシステム障害からの復旧を行う GPUfas を提案した．GPUfas では，障害を検知した後に復旧システムがメインメモリ上の OS データを書き換え，OS の機構を用いて障害からの復旧を試みる．GPUfas による復旧手法の例として，障害を引き起こしたプロセスへのシグナルの疑似送信やプロセスの疑似的なスケジューリングが挙げられる．メモリ書き換えだけでは実現が難しい機能については，GPUfas はカーネル内に組み込んだ復旧支援機構との連携を行う．Linux のメモリ管理機構の拡張と CUDA のマップトメモリ機構を用いて GPUfas を実装し，KVMonitor を拡張することで GPUfas-VM を実装した．実験により，GPUfas や GPUfas-VM はプロセスレベルの障害から短時間で復旧できることが分かった．

今後の課題は，疑似スケジューリングの安定性を向上させることである．そのためには，復旧支援機構と連携してスケジューラのロックを獲得したりすることが必要である．また，現在は GPUfas-VM にしか実装できていないため，GPUfas にも実装する予定である．さらに，NMI を利用することにより，割り込み禁止を伴うデッドロックなどのカーネルレベルの障害からも復旧支援機構と連携して復旧できるようにすることが挙げられる．

謝辞 本研究の一部は，JST，CREST，JPMJCR21M4 の支援を受けたものである．

参考文献

[1] AWS: Summary of the Amazon Kinesis Event in the Northern Virginia (US-EAST-1) Region, https://aws.amazon.com/jp/message/11201/.
[2] 株式会社日本取引所グループ：システム障害に係る独立社外取締役による調査委員会の報告書，https://www.jpx.co.jp/corporate/news/news-releases/0020/20201130-03.html.
[3] Linux-HA Japan JAPAN: Pacemaker の概要，http://linux-ha.osdn.jp/wp/.
[4] Y. Zhu and Y. Li and J. Xue and T. Tan and J.Shi and Y. Shen and C. Ma: What is System Hang and How to Handle it, *IEEE 23rd International Symposium on Software Reliability Engineering*, pp. 131–150 (2012).
[5] J. Leners and H. Wu and W. Hung nad M. Aguilera and M. Walfish: Detecting failures in distributed systems with the Falcon spy network, *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 279–294 (2011).

[6] K. Kourai and K. Nakamura: Efficient VM Introspection in KVM and Performance Comparison with Xen, *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pp. 18–21 (2014).
[7] Y. Ozaki and S. Kanamoto and H. Yamamoto and K. Kourai: Detecting System Failures with GPUs and LLVM, *APSys 2019*, pp. 47–53 (2019).
[8] P. Chen and W. Ng and S. Chandra and C. Aycock and G. Rajamani and D. Lowell: The Rio File Cache: Surviving Operating System Crashes, *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 74–83 (1996).
[9] F. David and J. Carlyle and H. Campbell: Exploring Recovery from Operating System Lockups, *ATC 2007*, pp. 351–356 (2007).
[10] A. Depoutovitch and M. Stumm: Otherworld - Giving Applications a Chance to Survive OS Kernel Crashes, *EuroSys 2010*, pp. 181–194 (2010).
[11] 金本颯将, 光来健一：GPUDirect RDMA を用いた高信頼な障害検知機構, *ComSys 2019* (2019).
[12] A. Bohra and I. Neamtiu and P. Gallard and F. Sultan and L. Iftode: Remote Repair of OS State Using Backdoors, *International Conference on Autonomic Computing*, pp. 256–263 (2004).
[13] K. Yamakita and H. Yamada and K. Kono: Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery, *Proc. Int. Conf. Dependable Systems and Networks*, pp. 168–180 (2011).