Vol.2020-OS-150 No.6
2020/7/30

GPUからの疑似的なシグナル送信による プロセスレベル障害からの復旧

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回避するのは難しい．そのため，システムに障害が発生した場合には迅速に障害を検知して復旧を行うことが重要である．障害からの復旧を行うにはシステムにログインして作業を行う必要があるが，システムが応答しなければハードウェアリセットによって復旧するしかない．しかし，強制的にシステムのリセットを行うとデータが失われ，復旧に時間やコストがかかる可能性がある．本稿では，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障害の原因を取り除く．復旧の一例として，GPU からプロセスに疑似的なシグナルを送信することによりプロセスレベルの障害からの復旧を可能にする．我々は Linux と CUDA 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 内のシステムの障害については，VM イントロスペクションを拡張して OS データを書き換えられるようにした．GPUfas を用いてプロセスレベルの障害からの復旧についての実験を行った．

1. はじめに

システム障害の原因として，(1) ソフトウェアの不具合，(2) 性能・容量不足，(3) 設定・操作ミス，(4) 不慮の事故が考えられる [1]．原因 (1)(2) はシステム開発者が十分に留意して回避するべきであるが，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困難である．仮に，原因 (1)(2) を持たないシステムが構築できたとしても，原因 (3) のようにシステム利用者が操作ミスなどをしてシステム障害を引き起こす場合もある．熟練の利用者が使用したとしても，原因 (4) のようにハードウェアの故障や停電などは回避できないこともある．以上のことから，システム障害を事前に防ぐことは非常に難しい．

システム障害が発生するとシステム上で動作しているサービスの品質が低下したり，完全に停止したりするため，サービスの利用者や提供者は大きな損失を被る．そのため，迅速に障害を検知して復旧を行うことが重要である．システムの復旧を行うためにはシステムにログインして作業を行う必要があるが，システムが応答しない場合にはハードウェアリセットを行うしかない．しかし，強制的にシステムのリセットを行うとデータの損失などが起こる可能性があり，障害発生前の正常な状態に戻すために時間とコストがかかる．

本稿では，できる限りシステムのリセットを回避するために，GPU 上で動作する復旧システムが 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 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，プロセスレベル障害からの復旧を行う．例えば，プロセスを一時停止させることで CPU 負荷を低下させたり，プロセスを終了させることでメモリ不足を解消したりすることができる．

我々は GPUfas を Linux や CUDA，LLVM を用いて実装した．GPU から直接メインメモリ全体を書き換えられるようにするために，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た．GPU がメインメモリ上の OS データを書き換える際には煩雑なアドレス変換が必要になるため，LLView[2] を拡張して復旧システムのプログラムに対してアドレス変換が透過的に行われるようにした．また，VM 内のシステム障害については GPU を用いず，VM イントロスペクションを拡張して OS データを書き換えて復旧するように KVMonitor[3] に機能を追加した．実験により，GPUfas がプロセスレベルの障害からの復旧を行えることを確認し，復旧にかかる時間の測定

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

ⓒ 2020 Information Processing Society of Japan

を行った．

以下，2 章ではシステム障害からの従来の復旧手法について述べる．3 章では，本稿で提案するシステムである，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 について述べる．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

システム障害には，システムのリソース不足による障害やソフトウェアの不具合による障害などがある．リソース不足による障害の例として，CPU が過負荷になってシステム性能が低下したり，メモリが不足してシステムの実行が継続できなくなったりすることが挙げられる．ディスクやネットワークの帯域が不足して入出力性能が大幅に低下する場合もある．また，ソフトウェアの不具合による障害としては，カーネル内でデッドロックが発生して無限待機させられることによって，システムが停止することがある．この他にも，カーネルがクラッシュして異常停止する場合もある．

このようなシステム障害が発生した場合には，迅速かつ正確に障害を検知してシステムの復旧を行う必要がある．そのために管理者はまず，SSH やシリアルコンソールなどを用いて障害の発生したホストにリモートログインし，障害の原因の除去を試みる．しかし，システムのリソースが不足している場合には，リモートログインに長い時間がかかったり，ログイン後の作業が円滑に行えなかったりすることが多い．例えば，メモリ不足によりスラッシングが発生すると，処理が著しく遅くなる．ディスクやネットワークの帯域が圧迫されている場合もそれらを用いる処理に大きな影響が出る．また，カーネルの不具合によるメモリ不足やデッドロック，クラッシュなどが発生した場合にはシステムが応答しなくなり，リモートログインすることができない．

このような場合にはハードウェアリセットを行うことによって，システムを再起動する必要がある．例えば，IPMI を用いることによ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ことができる．リモート電源管理装置を用いてサーバの電源を遮断することでリセットを行うこともできる．また，ハードウェア・ウォッチドッグタイマを用いることで，一定期間内にシステムからの信号がない場合にシステムをリセットすることもできる．

しかし，システムのリセットを行うとデータが失われる恐れがあるため，できるだけ避けるべきである．リセット時にメモリ上に置かれていて，ディスクに保存されていなかったアプリケーションのデータは失われる．メモリ上の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いデータも失われる．ファイルシステムの不整合が発生してファイルが失われたり，復旧できなくなったりする場合もある．このような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 [4][5][6] 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．

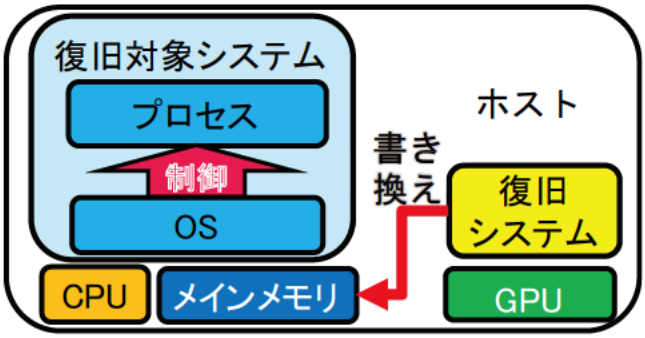

図 1 GPUfas のシステム構成

本稿ではシステム障害からの復旧の第一歩として，プロセスレベルのリソース不足による障害を対象とする．そのため，障害発生時にはカーネル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リセット以外では復旧が困難であるため，本研究の対象としない．

3. GPUfas

本稿では，GPU 上で動作する復旧システムが OS の挙動を間接的に変更することでシステム障害からの復旧を行う GPUfas を提案する．GPUfas では，障害発生時に GPU からメモリ上の 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取り除く．GPUfas のシステム構成を**図 1** に示す．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，復旧対象システムの障害の影響を受けにくい．GPU は専用の演算コアやメモリを持っているため，復旧対象システムのリソース不足の影響を受けない．また，システム起動時に GPU を占有して GPUfas を実行することにより，システム障害発生後に復旧システムが実行できないという事態を避ける．

GPUfas による復旧の例として，プロセスへのシグナルの疑似送信が挙げられる．OS はプロセスにシグナルを送ることによってプロセスを制御することができるが，GPU から直接プロセスにシグナル送信を行うことはできない．そこで，GPUfas がメインメモリ上にあるプロセス情報を書き換えてシグナルが送られた状態に変更することで，プロセスへのシグナル送信を疑似的に実現する．疑似送信されたシグナルは OS によって処理され，プロセスに配送さ

ⓒ 2020 Information Processing Society of Japan

れる．例えば，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，システムが応答可能な状態にすることができる．また，大量のメモリを確保しているプロセスを終了することでメモリを解放させ，メモリ不足によるスラッシングを起こさないようにすることができる．

GPUfas によるその他の復旧手法としては，プロセスのリソースを制限したり，スケジューリングを変更したりすることが可能である．プロセスが利用できる CPU 時間，ディスク帯域，ネットワーク帯域などを制限することで，システムの CPU 負荷を下げたり，ディスクやネットワーク帯域の圧迫を解消したりすることができる．また，CPU 負荷が高いプロセスの優先度を下げることで CPU 負荷を下げることができる．このような OS の挙動の変更によって復旧が完了する場合もあるが，そうでない場合には GPUfas が一時的な復旧を行ってリモートログインが可能な状態にした後で管理者が復旧を行う．

GPUfas はシステム障害を検知して障害の原因を特定した後で，障害の原因を取り除いてシステムの復旧を行う．障害検知は先行研究の GPUSentinel[2] を用いて行う．GPUSentinel は，監視対象システムに搭載された GPU 上で OS 監視システムを動作させてメインメモリ上の OS データを監視し，システム障害を検知する．システム障害を検知すると OS 監視システムは 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．GPUfas は特定されたプロセスを間接的に制御することにより障害の原因を自動的に取り除く．さらに，GRASS[7] を用いることで，障害情報を受け取ったリモートの管理者がインタラクティブに復旧を行うことも可能である．GRASS は GPUDirect RDMA を用いて障害検知結果および障害の原因を GPU からリモートホストに直接通知するシステムである．

仮想マシン（VM）内のシステム障害に対しても，VM 内のゲスト OS を間接的に制御することによって復旧を行う．この場合には，GPU は用いずに VM イントロペクション [3] を拡張して VM 外から OS データの書き換えを行う．VM イントロスペクションは本来，VM のメモリにアクセスして VM 内の情報を取得する技術であるが，GPUfas ではさらに VM 内の情報の書き換えも行う．このように，VM 内のシステムの復旧には GPU は不要であるが，ハイパーバイザや管理 VM の障害に対しては GPU を用いて復旧を行う必要がある．

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，VM 内のシステム障害からの復旧は KVMonitor 用に修正した QEMU-KVM 2.11.2 を用いて実装した．

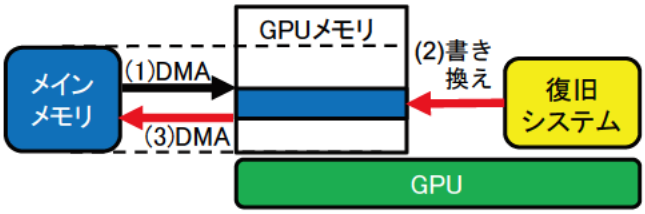

図 2 GPU からメインメモリ上の OS データの書き換え

4.1 シグナルの疑似送信

Linux では kill システムコールを用いてプロセス ID を指定して標準シグナルを送信すると，対象プロセスにシグナルが配送される．カーネル内では対象プロセスが持つシグナルビットマップの対応するビットの値を 1 にすることで，どのシグナルが送信されたかを記録する．次に，未処理のシグナルがあることを示すフラグを対象プロセスにセットする．そして，対象プロセスを実行状態にしてカーネルモードに遷移させる．その後，対象プロセスがカーネルモードからユーザモードに戻る際に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を配送する．

GPUfas では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 GPU から書き換える．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．疑似送信するシグナルとしては SIGSTOP と SIGKILL を想定している．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．現在のところ，対象プロセスを実行状態にしてカーネルモードに遷移させる処理は行っていない．これらを可能にするには，プロセスのスケジューリングや割り込みを実現する必要がある．

4.2 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．マップトメモリはメインメモリを GPU メモリアドレス空間にマッピングして，GPU 上のプログラムから参照可能にする機能である．CUDA ではメインメモリを直接 GPU のアドレス空間にマッピングできないため，メインメモリを一旦，プロセスのアドレス空間に読み書き可でマッピングしてから GPU のアドレス空間にマッピングする．障害が発生する前にメインメモリをマッピングしておくことにより，障害発生時でもマップトメモリ機構を利用することができる．しかし，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り，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

ⓒ 2020 Information Processing Society of Japan

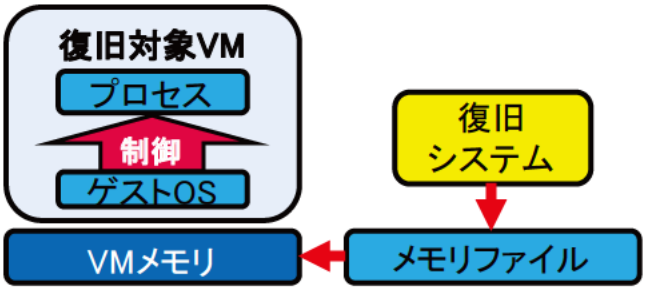

図 3 VM のメモリ上の OS データの書き換え

た．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プロセスにマッピングする．/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．

OS データの書き換えを行う時には，まず，OS の仮想アドレスを物理アドレスに変換し，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，**図 2** のように GPU が OS データを自動的にメインメモリから GPU メモリに DMA 転送する．GPU 上の復旧システムが DMA 転送された OS データを書き換えると，GPU が自動的に書き換えたデータをメインメモリに DMA 転送し，メインメモリに反映する．

4.3 VM のメモリ書き換え

KVM において 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[3] を拡張した．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．別のプログラムがメモリファイルをそのメモリアドレス空間にマッピングしてアクセスすることで，VM イントロスペクションを実現する．従来の KVMonitor は VM のメモリ読み込みのみに対応していたが，メモリファイルを読み書き可で復旧システムのメモリアドレス空間にマッピングすることにより，**図 3** のように VM のメモリ書き換えを実現する．

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メモリアドレスに変換してアクセスする．

```
%3 = bitcast i64* %arrayidx to i8*
%4 = call i8* @g_map(i8* %3)
%5 = bitcast i8* %4 to i64*
store i64 512, i64* %5
```

図 4 拡張 LLView を適用した後の中間表現

4.4 LLView の拡張

GPUfas において OS データの書き換えを行うにはアドレス変換が必要となり，復旧システムの開発が煩雑となる．そのため，LLView[2] を拡張して OS データのアドレス変換を透過的に行う．LLView は LLVM を用いてプログラムのコンパイルを行い，中間表現にアドレス変換を挿入することで自動アドレス変換を実現する．LLView はメモリからデータを読み込む際に用いられる load 命令を中間表現から探し，その直前にアドレス変換関数の呼び出し命令を挿入する．そして，変換後のアドレスを用いるように load 命令を置換する．また，OS の大域変数のシンボルを対応する仮想アドレスに置換することにより，自動アドレス変換を行ってメインメモリにアクセスするようにする．

GPUfas の拡張 LLView では，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 命令の直前にもアドレス変換関数の呼び出しを挿入する．**図 4** にプロセスのシグナルビットマップを書き換える際のアドレス変換の例を示す．アドレス変換を行う g_map 関数は void ポインタを引数にとるため，1 行目で 64 ビットの配列のアドレスを 8 ビットのポインタにキャストしている．2 行目で g_map 関数を呼び出し，3 行目で g_map 関数の返り値を再び 64 ビットのポインタにキャストしている．そして，4 行目で変換後の GPU アドレスを用いて store 命令を実行している．この拡張 LLView は GPU 上で動作する復旧システムだけでなく，KVMonitor を用いて VM の外側で動作する復旧システムにも対応している．

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．

5.1 リソース不足による障害からの復旧

CPU やメモリが不足するとシステムの性能が著しく低下するため，システムのリソースが不足していることを検知して復旧できることを確認する実験を行った．GPUfas による復旧との比較として，SSH でリモートログインして pkill コマンドを用いて復旧を行った．メモリ不足による障害を発生させるために，16GB のメモリを確保するプロセスを 1 つ実行した．その後で SSH を用いてリモートログインすると，このプロセスのためにスラッシングが発生し

ⓒ 2020 Information Processing Society of Japan

表 1　復旧対象ホスト

OS	Linux 4.18.0
CPU	Intel Core i7-8700
メモリ	DDR4-2667 16GB
スワップメモリ	5GB
GPU	NVIDIA GeForce GTX 960
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

ゲスト OS	Linux 4.15.0
割当メモリ量	2048MiB
CPU 数	1 個
割当ディスク量	50GiB
ソフトウェア	QEMU-KVM 2.11.2

て正常時の約 25 倍の時間がかかった．メモリを確保した後はプロセスが定期的にスケジュールされるように 1 秒間のスリープを繰り返した．この障害からの復旧にかかった時間を**図 5** に示す．

GPUfas を用いた場合，システムのメモリ不足を検知して障害の原因となったプロセスを特定した後，624ms で KILL シグナルを疑似送信してそのプロセスを強制終了させることができた．これによりシステムのメモリ不足が解消され，障害からの復旧を行うことができた．一方，pkill コマンドを用いた場合，SSH ログイン後のコマンド実行の際にもスラッシングが発生し，障害の原因となったプロセスを強制終了させるのに 3 秒以上の時間がかかった．このことから，GPUfas のほうが 5 倍高速に復旧を行えることが分かった．

CPU 過負荷による障害を発生させるために，yes コマンドを CPU コアの数だけ実行した．これらのプロセスによって Unixbench の fstime テストにおいて約 40 %処理性能が低下した．この障害からの復旧にかかった時間を**図 6** に示す．GPUfas を用いた場合，すべての CPU の使用率が高いことを検知した後，yes コマンドを実行しているプロセスに KILL シグナルを疑似送信して強制終了させるのに 27ms かかった．一方，pkill コマンドを用いた場合，13ms でプロセスを強制終了させることができた．このことから，yes コマンドによる CPU 過負荷は pkill コマンドの実行に大きな影響を与えていないことが分かる．また，GPUfas ではプロセスリストから対象プロセスを探すのに時間がかかっている可能性がある．

5.2　大量のプロセスへのシグナル疑似送信

GPUfas による大量のプロセスへのシグナル疑似送信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実行して無限ループさせた後，GPUfas がプロセスリストをたどって名前が一致するプロセスに KILL ま

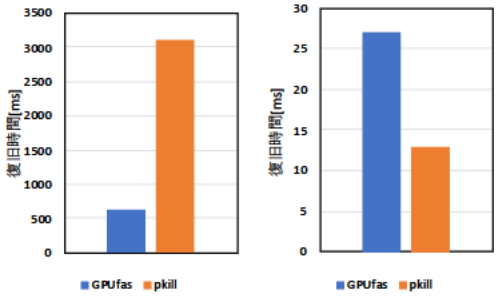

図 5　メモリ不足からの復旧時間　　**図 6**　CPU 過負荷からの復旧時間

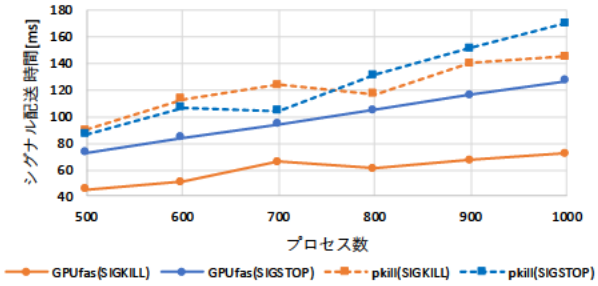

図 7　10ms ごとに動くプロセスへのシグナル配送時間

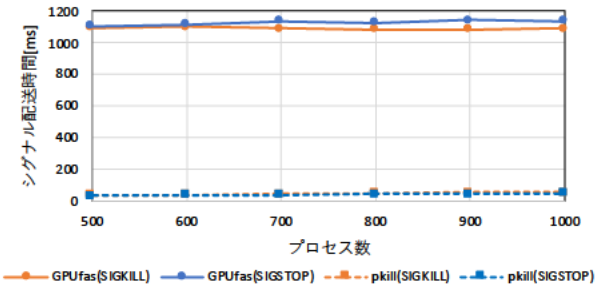

図 8　1 秒ごとに動くプロセスへのシグナル配送時間

たは STOP シグナルを疑似送信した．KILL シグナルを送信した場合はすべてのプロセスが終了するまでの時間を測定し，STOP シグナルを送信した場合にはすべてのプロセスが停止状態になるまでの時間を測定した．比較として，pkill コマンドを使ってこれらのシグナルを送信し，すべてのプロセスが終了または一時停止するまでの時間を測定した．

無限ループして行う処理が異なる 2 種類のプロセスについて，様々な数のプロセスへのシグナル配送が完了するまでにかかった時間を**図 7** と**図 8** に示す．図 7 はプロセスが 10ms のスリープを繰り返し，10ms ごとにカーネルモードからユーザモードへ遷移させた場合の結果である．この場合には，GPSfas の方が pkill コマンドよりも高速にシグナル配送を行えたことが分かる．KILL シグナルの場合は平均 61ms，STOP シグナルの場合は平均 25ms 高速となった．また，KILL シグナルの方が STOP シグナルより高速に配送できることが分かった．

ⓒ 2020 Information Processing Society of Japan

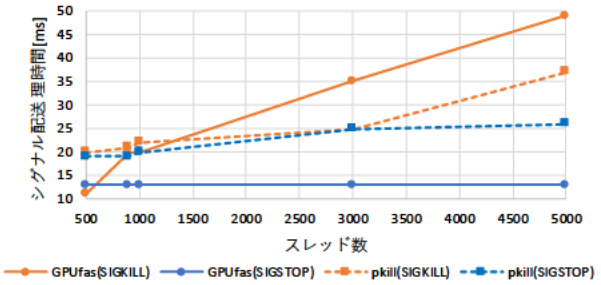

図 9 大量のスレッドを持つプロセスへのシグナル配送時間

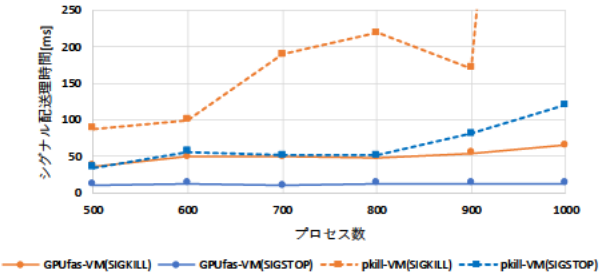

図 10 VM 内で 10ms ごとに動くプロセスへのシグナル配送時間

一方，図 8 はプロセスが 1 秒のスリープを繰り返した場合の結果である．この場合には，GPUfas がシグナルの疑似送信を行っても，プロセスが最大 1 秒間スケジュールされないため，GPUfas のほうがシグナル配送にかかる時間が大幅に長くなった．高速化のためには，シグナルを疑似送信したプロセスを即座にスケジュールさせるようにする必要がある．

5.3 大量のスレッドをもつプロセスへのシグナル疑似送信

GPUfas による大量のスレッドを持つプロセスへのシグナル疑似送信の性能を調べる実験を行った．この実験では，1 つのプロセスが指定した数のスレッドを実行して無限ループさせた後，GPUfas がプロセスリストをたどって対象プロセスに KILL または STOP シグナルを疑似送信した．KILL シグナルを送信するとそのプロセスの持つすべてのスレッドが終了し，STOP シグナルを送信するとすべてのスレッドが一時停止する．比較のために，pkill コマンドを用いて同様のシグナルを送信した．

様々な数のスレッドを持つプロセスへのシグナル配送が完了するまでの時間を**図 9** に示す．この実験では，各スレッドは 10ms のスリープを繰り返すようにした．STOP シグナルの配送については GPUfas の方が高速であることが分かった．一方，KILL シグナルの配送はプロセスの持つスレッド数が増えると pkill コマンドよりも時間がかかった．図 7 とは異なり，スレッド数が多い場合には STOP シグナルの配送のほうが KILL シグナルの配送よりも高速であった．これはスレッドの停止が終了よりも高速であるためと考えられる．

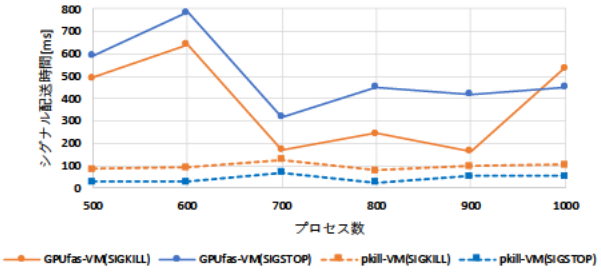

図 11 VM 内で 1 秒ごとに動くプロセスへのシグナル配送時間

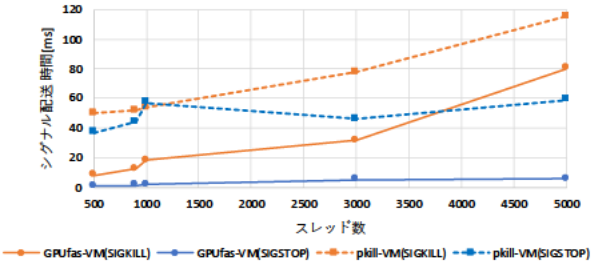

図 12 VM 内の大量のスレッドを持つプロセスへのシグナル配送時間

5.4 VM 内のプロセスへのシグナル疑似送信

VM 内のゲスト OS 上で動作している大量のプロセスに対して，5.2 節と同様にシグナルを疑似送信する性能を調べた．シグナルの配送時間を**図 10** と**図 11** に示す．プロセスが 10ms のスリープを繰り返した場合，VM を用いない場合と同様に GPUfas-VM のほうが pkill コマンドよりも高速にシグナルを配送することができた．GPU 上の復旧システムより VM の外側の CPU 上で動く復旧システムのほうが高速であるため，VM を用いない場合よりもシグナル配送時間が短かったと考えられる．また，VM を用いない場合と異なり，STOP シグナルの配送のほうが KILL シグナルの配送よりも高速であった．一方，プロセスが 1 秒のスリープを繰り返した場合，VM を用いない場合と同様に GPUfas-VM は pkill コマンドよりも遅くなったが，VM を用いない場合よりは高速であった．しかし，プロセス数による配送時間のばらつきが大きくなった．これは VM に仮想 CPU を 1 つだけ割り当てたことが原因となっている可能性がある．

5.3 節と同様に大量のスレッドを持つプロセスにシグナルを疑似送信した場合の配送時間を**図 12** に示す．図 9 と同様に，GPUfas-VM によるシグナル配送時間は他の場合よりも短くなった．VM を用いない場合よりも高速であり，STOP シグナルの配送のほうがより高速であった．一方，VM 内で pkill コマンドを実行すると VM を用いない場合より大幅に実行時間がかかった．これは仮想化によるプログラムの実行オーバヘッドが大きいためと考えられる．

ⓒ 2020 Information Processing Society of Japan

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 [8] 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[9] 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，セキュリティホールになり得る．一方で，GPUfas では GPU から復旧を行うためよりセキュアである．GPUfas を GRASS[2] 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を行うため，Backdoors のような危険性はない．

Exterior[3] は VM 内のシステムが攻撃を受けた時に VM の外側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．kill コマンドを用いてプロセスを強制終了させたり，rmmod コマンドを用い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．

Otherworld [6] 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 [10] 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，GPU 上でシステム障害を検知した後で 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 を提案した．GPUfas では，GPU 上で動作する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 自身の機能を用いて障害の原因を取り除く．復旧の一例として，プロセスにシグナルを疑似的に送信することで，プロセスレベルの CPU 過負荷やメモリ不足などの障害からの復旧を可能にする．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，開発者がアドレス変換を意識せずに復旧システムを開発できるように LLView を拡張した．実験により，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，GPU からプロセスのリソース使用量を制限したりスケジューリングを変更したりすることでも，プロセスレベルの障害からの復旧を行えるようにすることである．スケジューリングを変更できるようにすることでスケジューリング待ちプロセスに即座にシグナルを配送できるようになるだけでなく，カーネルスレッドの実行を制御することでカーネルレベルの障害からの復旧を行うことも可能になる．また，OS データを書き換える際にカーネル内のロックを獲得できるようにすることで競合を防ぐことも必要である．

参考文献

[1] 大和田尚孝：システムはなぜダウンするのか，日経 BP (2009).
[2] Ozaki, Y., Kanamoto, S., Yamamoto, H. and Kourai, K.: Detecting System Failures with GPUs and LLVM, *APSys 2019*, pp. 47–53 (2019).
[3] Kourai, K. and Nakamura, K.: Efficient VM Introspection in KVM and Performance Comparison with Xen, *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pp. 18–21 (2014).
[4] Chen, P., Ng, W., Chandra, S., Aycock, C., Rajamani, G. and Lowell, D.: The Rio File Cache: Surviving Operating System Crashes, *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 74–83 (1996).
[5] David, F., Carlyle, J. and Campbell, H.: Exploring Recovery from Operating System Lockups, *ATC 2007*, pp. 351–356 (2007).

ⓒ 2020 Information Processing Society of Japan

[6] Depoutovitch, A. and Stumm, M.: Otherworld - Giving Applications a Chance to Survive OS Kernel Crashes, *EuroSys 2010*, pp. 181–194 (2010).
[7] 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).
[8] Zhu, Y., Li, Y., Xue, J., Tan, T., J.Shi, Shen, Y. and Ma, C.: What is System Hang and How to Handle it, *IEEE 23rd International Symposium on Software Reliability Engineering*, pp. 131–150 (2012).
[9] Bohra, A., Neamtiu, I., Gallard, P., Sultan, F. and Iftode, L.: Remote Repair of OS State Using Backdoors, *International Conference on Autonomic Computing*, pp. 256–263 (2004).
[10] Yamakita, K., Yamada, H. and Kono, K.: Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery, *Proc. Int. Conf. Dependable Systems and Networks*, pp. 168–180 (2011).

ⓒ 2020 Information Processing Society of Japan