Vol.2019-OS-145 No.12
2019/3/1

GPUとLLVMを用いたOSレベルでの障害検知機構

尾崎 雄一[1]　山本 裕明[1]　光来 健一[1]

概要：情報システムに障害が発生すると大きな損失となるため，システム障害はできるだけ早く検知して復旧を行う必要がある．システム障害の検知を行うには，監視対象システムの外部から監視を行う方法と内部から監視を行う方法が挙げられる．しかし，外部から監視を行う場合はネットワーク障害などで監視を継続できなくなることがあり，内部から検知を行う場合はシステム障害の影響を受けて障害検知ができなくなる可能性がある．そこで，本稿ではシステム内部で監視でき，システム障害の影響を受けにくいGPU を用いて障害検知を行う GPUSentinel を提案する．GPUSentinel では，GPU 上の検知プログラムがメインメモリを参照することによってシステムの状態を取得する．OS のソースコードを最大限に利用して検知プログラムを記述可能にするために，LLVM を用いてプログラム変換を行う．Linux，GPU ドライバ，CUDA を用いて GPUSentinel を実装し，意図的に発生させた障害が検知できることを確認した．

1. はじめに

近年，情報システムは機能の多様化，大規模システムを開発する技術の発達等により複雑化してきている．その結果，システム障害の発生も増加傾向にある．大きなシステム障害だけでもここ 8 年間で倍増している [1]．システム障害が発生するとサービスが停止してしまい，サービス提供者やシステム管理者，サービスの内容によってはユーザにも損失を発生させてしまう．そこで，損失を減らすためにシステム障害をできるだけ早く，正確に検知する必要がある．

従来の障害検知手法として，監視対象システムの外部から監視を行う手法と内部で監視を行う手法がある．システムの外部から監視する場合，ホストやサービスに対する死活監視を行って障害を検知するのが一般的である．監視対象が仮想マシン（VM）の場合には，VM の外側で仮想ハードウェアの状態を監視することもできる．この手法にはシステム障害時でも監視が行えるという利点があるが，詳細なシステム情報の取得ができないという欠点がある．一方，システムの内部で監視する場合，OS 上で検知プログラムを動作させたり OS に検知プログラムを組み込んだりすることになる．この手法は，システム内部の詳細な情報を用いて精度の高い障害検知を行うことができるが，監視対象システム内で検知プログラムを動作させているため障害の影響を受けやすいという欠点がある．

そこで，本稿では監視対象ホストの GPU 上で検知プログラムを動作させて障害検知を行うシステム GPUSentinel を提案する．GPU は監視対象システムを動作させている CPU やメインメモリから独立してプログラムを実行することができるため障害の影響を受けにくい．また，監視対象ホスト内部で動作するため，障害検知に OS レベルの詳細な情報を利用可能である．GPUSentinel では，障害検知のために GPU 上の検知プログラムがメインメモリを参照し，OS のデータを解析する．OS のソースコードを最大限に利用して検知プログラムを記述可能にするために，LLVM を用いてプログラム変換を行う．システムに関する様々な情報が障害検知には必要となるが，GPU の多数のコアを用いることで障害検知の並列化が可能である．

我々は GPUSentinel を CUDA を用いて実装した．メインメモリを GPU から参照できるようにするために，CUDA のマップトメモリ機能を用いた．メインメモリ全体を GPU メモリにマップできるようにするために，Linux カーネルと NVIDIA の GPU ドライバに修正を加えた．また，検知プログラムが OS データにアクセスする際にその仮想アドレスを GPU アドレスに透過的に変換するために，LLVM を用いる LLView フレームワークを開発した．GPUSentinel を用いて CPU の高負荷状態やメモリ枯渇，デッドロックを検知するプログラムを作成し，意図的に発生させた障害を検知できることを確認した．

以下，2 章で従来の障害検知手法の問題点について述べる．3 章で GPUSentinel を提案し，4 章でその実装について述べる．5 章で GPUSentinel を用いて動作する検知プログラムの性能について調べた実験について述べる．6 章で関連研究に触れ，7 章で本稿をまとめる．

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

図 1: 従来の監視手法

2. 障害検知

近年，情報システムはクラウドコンピューティングなどの多様な技術を取り入れることで多機能化を実現しており，システム構造が複雑化している．その結果，システム障害の発生件数も年々増加してきている [1]．システム障害が発生するとサービス提供者やシステム管理者，ユーザに大きな損失が生じてしまう．オンラインバンキングを提供するシステムを例に挙げると，システムに障害が発生した場合，サービス提供者は障害から復旧するまで利益を得られなくなる．また，システム管理者は障害の原因を特定してシステムを復旧しなければならない．システムのユーザにも取引ができないことによる損失が発生する可能性がある．

このような損失を軽減するためには，障害から素早く復旧しサービス提供を再開することが重要である．そのためには障害をできるだけ早く検知する必要がある．可能であれば，障害によってサービスが提供できなくなる前に予兆を検知するのが望ましい．また，障害をできるだけ正確に検知することも必要である．障害を誤検知してしまうとシステム管理者が発生していない障害の原因を調査しなければならず，サービスを停止させなければならなくなる場合もあるためである．その結果，サービス提供者やユーザの損失につながってしまう．

従来のシステムの監視方法として，図 1 のように監視対象システムの外部から監視する方法と内部で監視する方法がある．監視対象システムの外部からネットワーク経由で監視する例として，監視対象ホストの死活監視を行う方法や監視対象システムが提供しているネットワークサービスに定期的に接続する手法があげられる．後者の手法ではサービス単位で正常に動作しているかどうかを確認できる．これらの手法の利点は，監視対象システムの内部に依存することなくブラックボックスとして障害検知ができることである．しかし，監視対象システムの内部状態はわからないため，障害発生時に詳細な情報を取得するのが難しい．そのため，応答がない時にどのような障害が発生しているかがわからないという点が欠点である．

監視対象システムが VM 内で動作している場合は，VM の外側でより詳細な情報を取得することが可能である．検知システムは仮想化システムから VM の仮想ハードウェアの状態を取得することができ，その情報を障害検知に利用することができる．例えば，VM の CPU 使用率，ディスクアクセス量，ネットワークパケット数などを取得することで異常を検知しやすくなる．しかし，それ以上の詳細な情報は得られず，これらの情報を利用できるのは VM を使用している場合に限られる．

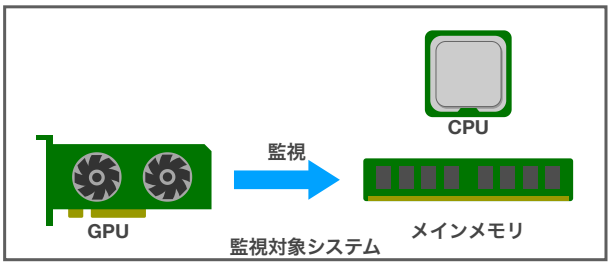

図 2: GPUSentinel のシステム構成

一方，監視対象システムの内部で監視する場合，検知プログラムを OS に組み込んで動作させたり，OS 上で動作させて OS から情報を取得したりする．そのため，検知プログラムがシステム内部の詳細な情報を利用することができ，より精度の高い障害検知を行うことができる．しかし，監視対象システムの OS に障害が発生した場合に検知システムが正常に機能しなくなったり，機能を停止したりする可能性が高い．例えば，システムのメモリが足りなくなった場合には検知プログラムが強制終了させられる場合がある．また，監視対象システムと同じホストで検知プログラムを実行するため，システム全体の性能低下を引き起こす可能性もある．

3. GPUSentinel

本稿では監視対象ホストに搭載された GPU 上で検知プログラムを動作させてシステムを監視し，障害を検知するシステム GPUSentinel を提案する．システムの概要図を図 2 に示す．GPUSentinel では，検知プログラムは障害が発生する前のシステム起動時に開始され，GPU を占有して自律的に動作する．監視対象システムが GPU を使う場合でも，別途，GPU を用意することで GPUSentinel が利用可能である．GPU からシステムを監視するために，検知プログラムはシステムのメインメモリ上の OS データを解析する．これにより，OS レベルの詳細な情報を利用して精度の高い障害検知を行うことが可能となる．検知プログラムによって障害と判断された場合は管理者に障害発生を通知する．

GPU は監視対象システムが動作している CPU やメインメモリから独立して動作するため，システム障害が発生しても GPU 上の検知プログラムは動作し続ける可能性が高い．GPU でシステムの監視を行うことにより，メモリや PCI Express の帯域に影響を与える可能性はあるものの，CPU 性能には影響を与えないようにすることができる．また，GPU は多数の演算コアを有しているため，同時に複数の監視処理を行うことが可能である．この特徴により，さまざまな障害の予兆を並列して調べることができ

る．それぞれの監視処理についても複数のコアを用いて並列化することで高速化することができる．

GPU 上の検知プログラムはメインメモリに格納されているデータから見つけられる障害を検知可能である．GPUSentinel で検知可能な障害の例として，システムリソースの枯渇による障害が挙げられる．メモリを大量に使用するプロセスがあったり，メモリリークを起こしたりしている場合にはメモリを確保できなくなる．また，システムが飢餓状態に陥ったことによる障害も検知可能である．スピンロックによってすべての CPU がデッドロックした場合には OS が動作を停止する．このような障害を検知するために，GPUSentinel の検知プログラムは OS 内の空きメモリ容量や使われた CPU 時間を調べることができる．そして，これらの値が閾値を下回ったり超えたりした場合に障害として検知することができる．ただし，GPU はメインメモリ以外にはアクセスできないため，その他のハードウェア障害の検知は難しい．

GPUSentinel では，検知プログラムの開発を支援するために，LLVM を用いたプログラム変換を行う．GPU 上の検知プログラムがメインメモリ上の OS データを解析する際には，OS の仮想アドレスから GPU アドレスにアドレス変換を行う必要がある．しかし，検知プログラムにアドレス変換の処理を記述するのは煩雑である．そこで，LLVM を用いて検知プログラムをコンパイルし，中間表現を変換することで透過的にアドレス変換を行う．これにより，OS のソースコードを最大限に利用して，検知プログラムをカーネルモジュールのように記述することができる．

GPUSentinel は検知した障害を管理者に通知する機構を提供する．障害発生時には OS が停止している可能性があるため，OS の機能を用いる通知機構は利用できる保証がない．そこで，GPUSentinel は GPU からメインメモリ上の VRAM 領域に値を書き込むことで画面に文字や画像を出力する．IPMI のリモートコンソール機能や KVM スイッチを利用することでリモートホストで通知を受け取ることができる．受信した画面の画像処理を行うことで通知内容の読み取りを自動化することもできる．なお，GPUDirect RDMA をサポートした GPU と RDMA 対応ネットワークが利用可能であれば，GRASS[2] を用いて通知を管理者に送ることができる．GRASS では GPU とリモートホストが直接通信を行うため，画面には表示できない詳細な情報を管理者に通知可能である．

4. 実装

我々は Linux 4.4 と NVIDIA ドライバ 375.66 に修正を加え，CUDA 8.0 と LLVM 5.0 を用いて GPUSentinel を実装した．

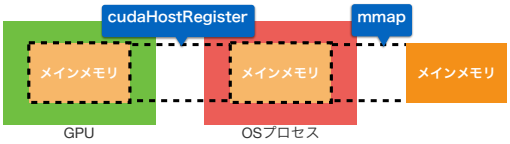

図 3: マップトメモリを用いたメインメモリのマッピング

4.1 メインメモリのマッピング

GPUSentinel では CUDA が提供するマップトメモリ機能を用いて GPU 側から自律的にメインメモリにアクセスできるようにする．マップトメモリはプロセスのメモリページを GPU のアドレス空間にマップして，GPU 上のプログラム（GPU カーネル）から参照可能にする機能である．マップした領域に GPU カーネルがアクセスすると透過的に DMA 転送が行われる．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プしておくことにより，障害によりシステムが機能しなくなっても GPU カーネルはメインメモリを参照することができる．

マップトメモリ機能を利用するために，GPUSentinel はまず，図 3 のようにメインメモリをプロセスのアドレス空間にマップする．これは CUDA ではプロセスのメモリページしか GPU アドレス空間にマップできないためである．マップしたメインメモリを GPU アドレス空間にマップする際には，ページアウトされないように CUDA によってすべてのメモリページがピン留めされる．そのため，単純にメインメモリ全体をプロセスにマップするとシステムの空きメモリがなくなる．プロセスにマップしたメモリページがすべてロックされて使用中になるためである．

そこで，GPUSentinel では Linux カーネルのメモリ管理に修正を加え，/dev/pmem という特殊なデバイスファイルを提供するようにした．このデバイスファイルは mmap システムコールでマップされる時に，メモリページの参照カウンタを増やさないようにすることで，メモリが使用中にならないようにする．GPUSentinel では，プロセス自身がマップされたメモリを使用するわけではないので，メモリページの参照カウンタが 0 でも問題にはならない．また，このようにしてマップされたメモリページをピン留めする際にもロックを行わないようにする．

このようにしてマップした後で，ピン留めされたメモリページに対してピン留めを正常に解除し，アンマップできるようにするために，Linux カーネルだけでなく NVIDIA ドライバにも修正を加えた．通常，ピン留めを解除する際にはメモリページがアンロックされるが，/dev/pmem をマップした場合にはアンロックしないようにする．また，アンマップする際には参照カウンタが減らされるため，同様に参照カウンタを変更しないようにする．NVIDIA ドライバの大部分はソースコードが非公開であるが，メモリをピン留めする部分は公開されているため，その部分だけを修正した．

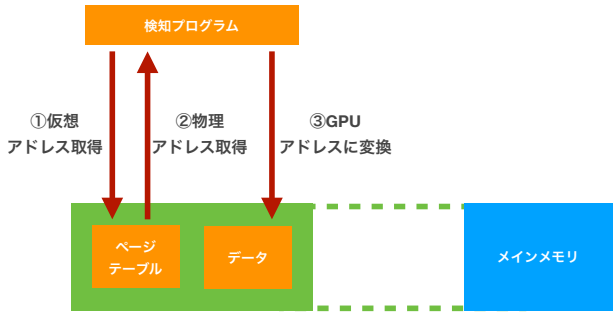

図 4: アドレス変換の流れ

CUDA のマップトメモリ機能の制限を回避するために，GPUSentinel は sysinfo システムコールの偽装を行う．CUDA ではメインメモリ全体のサイズより少し小さなサイズのメモリしか GPU アドレス空間にマップすることができない．これはすべてのメモリページをピン留めした結果，システムが動作しなくなるのを防ぐためだと考えられるが，GPUSentinel は実際にはメモリページのピン留めを行わないため不要な制限である．そこで，CUDA プログラムを起動する時にだけ LD_PRELOAD を用いて 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い値を返すことでメインメモリ全体をマップ可能にした．

4.2 自動アドレス変換

GPU 上で動作する検知プログラムはメインメモリから OS のデータを取得する際にアドレスを変換する必要がある．検知プログラムは仮想アドレスを用いて OS のデータにアクセスするが，仮想アドレスのままでは GPU アドレス空間にマップされたメインメモリにはアクセスできないため，GPU アドレスに変換してからアクセスを行う．そのために，図 4 のように，まずメインメモリ上にある OS 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 GPU アドレスに変換することになる．このアドレス変換を検知プログラムがデータを取得するたびに記述するのは冗長かつ煩雑である．

そこで，このアドレス変換を透過的に行えるようにするフレームワークである LLView を開発した．LLView は検知プログラムを LLVM 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自動アドレス変換を実現する．LLVM の中間表現では，メモリからデータを読み込む箇所で `load` 命令が使用される．この `load` 命令の直前でアドレス変換を行う関数を呼び出し，変換されたアドレスを用いて `load` 命令を実行するように命令を置き換える．アドレス変換を行う関数は引数のアドレスが OS カーネルのアドレスではない場合には渡されたアドレスをそのまま返す．これにより，マップされたメインメモリにアクセスする時以外は GPU のローカルメモリにアクセスすることができる．

```
#include <linux/sched.h>

void follow_proc_list()
{
  struct task_struct *p;
  p= &init_task;

  do {
    printf("pid: %d\n", p->pid);
    printf("comm: %s\n", kstr(&p->comm));
    p = list_entry(p->tasks.next,
                   struct task_struct, tasks);
  } while (p != &init_task);
}
```

図 5: プロセスリストをたどる検知プログラム

LLView は中間表現の変換に Pass を用いる．Pass は中間表現に対して最適化を施すための LLVM の仕組みである．LLView は中間表現中に `load` 命令を見つけると，メモリからの読み込みを行おうとしている変数とその型を取得する．そして，それらの情報を用いてアドレス変換を行う `g_map` 関数を呼び出す命令を生成し，`load` 命令の直前に挿入する．また，検知プログラムが OS カーネル内のグローバル変数にアクセスできるようにするために，LLView は中間表現におけるカーネル変数をカーネル内で用いられている仮想アドレスに置換する．Linux の場合，カーネル変数と仮想アドレスの対応は System.map ファイルから取得できる．

4.3 検知プログラムの開発

GPUSentinel では，検知プログラムは Linux カーネルのヘッダファイルを用いてカーネルモジュールのように記述する．図 5 はプロセスリストをたどって情報を取得する検知プログラムの例である．このコードはカーネル変数の `init_task` を開始点として循環リストをたどりながら，プロセス ID とプロセス名を表示している．ここで，プロセス名は `kstr` 関数を用いて明示的にアドレス変換を行っている．これは，文字列が文字列型のポインタであるため，LLView では自動アドレス変換が行われないためである．

検知プログラムは GPU 上で動作する CUDA のデバイスコードとして C 言語で記述される．通常，CUDA プログラムはデバイスコードと CPU 上で動作するホストコードの両方を C++で記述する．しかし，Linux のソースコードは C 言語で記述されているため，それを利用して記述したデバイスコードを C++でそのままコンパイルするのは難しい．例えば，C++の予約語と C 言語の変数名が衝突したり，C 言語では不要な型キャストが必要とされたり，void 型のポインタ演算が行えなかったりする．

LLView では，デバイスコードを C 言語としてコンパイルできるように clang に修正を加えた．clang ではプログ

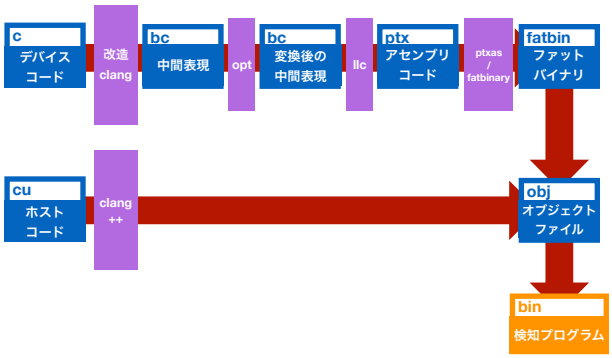

図 6: LLView を用いたコンパイルの流れ

ラムの種類ごとにコンパイルに用いる規格が定義されているため，CUDA プログラムが C90 と GCC 拡張を用いてコンパイルされるようにした．LLView を用いて CUDA プログラムをコンパイルする流れは図 6 のようになる．まず，デバイスコードを修正を加えた clang でコンパイルし，中間表現が格納されたビットコードを生成する．このビットコードに対して opt で Pass を適用してプログラム変換を行う．続いて，llc を用いて変換後のビットコードからアセンブリコードを生成し，CUDA の ptxas および fatbinary を用いてファットバイナリと呼ばれる埋め込み可能なバイナリを生成する．最後に，ホストコードを標準の clang++ でコンパイルしてオブジェクトファイルを生成し，それにファットバイナリを埋め込んで CUDA バイナリを生成する．

CUDA がデバイスコードに対して提供している変数や関数について，LLView は C 言語のラッパー関数を提供する．これは，LLView ではデバイスコードを C 言語としてコンパイルするため，C++で実装されている CUDA の変数や関数をそのまま利用することができないためである．例えば，GPU カーネルで一意なスレッド ID を得るには，ブロック内のスレッド数を返す `blockDim` 変数とブロック内のスレッドインデックスを返す `threadIdx` 変数が必要である．そこで，これらを基にスレッド ID を返す C 言語の `get_thread_id` 関数を用意した．また，排他的にメモリアクセスをする際に用いられる `__threadfence` 関数については `__gs_threadfence` 関数を用意した．

4.4 検知プログラムの例

GPUSentinel を用いてシステムリソースの枯渇およびデッドロックを検知する検知プログラムを作成した．障害を検知するための閾値はすべて実行環境ごとにチューニングが必要なパラメータである．

4.4.1 CPU の高負荷状態の検知

この検知プログラムは以下の計算式を用いて 1 秒ごとに各 CPU の使用率を計測する．

$$cpurate = \frac{user + sys + nice}{jiffies} \quad (1)$$

$user$ は前回の測定以降にプロセスのユーザ空間での実行に用いられた CPU 時間，sys は前回の測定以降にカーネル空間での実行に用いられた CPU 時間，$nice$ は前回の測定以降に優先度を上げたプロセスの実行に用いられた CPU 時間，$jiffies$ は前回の測定から経過した時間である．それぞれの CPU 時間は CPU ごとに Linux の `kcpustat_cpu` マクロを利用して取得し，$jiffies$ の値は Linux カーネル内の `jiffies` 変数から取得する．

すべての CPU の使用率が 90%を超える状態が 5 秒以上継続した場合，異常と判断してより詳細なプロセスごとの CPU 使用率を計測する．その計算式には

$$process\_cpurate = \frac{\sum_{i=0}^{n} user_i + \sum_{i=0}^{n} sys_i}{jiffies} \quad (2)$$

を用いる．n は総プロセス数，$user_i$ はプロセス i が前回の測定以降にユーザ空間で消費した CPU 時間，sys_i はプロセス i が前回の測定以降にカーネル空間で消費した CPU 時間である．$user_i$，sys_i はそれぞれ `task_struct` 構造体を用いて Linux カーネルの `task_cputime_adjusted` 関数と同じ処理を行うことで取得する．測定したプロセスの使用率を高い順にソートし，障害発生をその原因とともに管理者に通知する．

4.4.2 メモリの枯渇の検知

この検知プログラムはメインメモリの空き容量とスワップ領域の空き容量を取得し，それぞれの容量全体に対する割合を算出する．それぞれの値は Linux カーネルの `si_meminfo` 関数と `si_swapinfo` 関数と同じ処理を行うことで取得する．メインメモリとスワップの両方の空き容量が 30%未満になると異常と判断し，プロセスごとの OOM スコアを算出する．OOM スコアは Linux カーネルの OOM Killer で利用される値であり，スコアが高いプロセスが強制終了させられる．OOM スコアを計算するために，Linux カーネルの `oom_badness` 関数の処理を一部変更して用い，プロセスが使っているメインメモリ，ページテーブル，スワップ領域に基づいて OOM スコアを計算する．検知プログラムは OOM スコアが最も高いプロセスを障害の原因として管理者に通知する．

4.4.3 デッドロックの検知

4.4.1 節と同様に，この検知プログラムは 1 秒ごとに各 CPU の $user$ と sys の値を取得し，$jiffies$ の値に対するそれぞれの割合を算出する．同時に，すべての CPU におけるコンテキストスイッチの回数の合計を Linux カーネルの `nr_context_switches` 関数と同じ処理を行って取得する．この検知プログラムはデッドロックによる 2 種類の異常を検知する．一つは，一部の CPU のデッドロックであり，デッドロックが発生した CPU を OS が使えなくなることにより，システム全体の動作が遅くなる．この異常は 1 つ以上の CPU で sys の割合が 95%より大きくかつ，$user$ の割合が 4%より小さい状態が 5 秒以上継続した場合に検

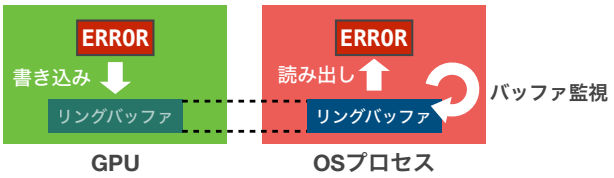

図 7: リアルタイム出力機構

知される．

もう一つは，すべての CPU のデッドロックであり，すべての CPU が使用できなくなることにより，システムが応答しなくなる．この異常はすべての CPU で *sys* の割合が 95%より大きくかつ *user* の割合が 4%より小さいかまたは，コンテキストスイッチが 1 秒間でまったく発生しない場合に検知される．これらの閾値は文献 [4][5] を参考にした．これらのいずれかの異常が検知されると，障害発生を管理者に通知する．

4.5 障害の通知

GPUSentinel は GPU 上の検知プログラムが検知した障害を外部に通知するためのリアルタイム出力機構を提供する．GPU カーネルには `printf` 関数が提供されているが，GPU カーネルが終了するまでは出力されないため，障害を即座に通知するのには利用できない．リアルタイム出力機構は図 7 のようにリングバッファを用いて OS 上のホストプロセスにデータを送信する．まず，ホストプロセスでリングバッファを確保し，これをマップトメモリ機能を用いて GPU のメモリにマップする．リングバッファのどこまでデータが書き込まれているかを管理するためのポインタ変数も同様に確保して GPU にマップする．GPU カーネルがデータを出力する時にはまず，リングバッファに出力するデータを書き込み，ポインタを書き込んだデータサイズ分だけ進める．ホストプロセスはポインタ変数を定期的に監視し，変化を検知するとその分だけリングバッファからデータを読み取る．このデータをリモートホストに転送することで障害発生を通知することができる．

障害が発生しても OS が動作している間はリアルタイム出力機構とネットワーク通信を用いることができるが，OS が停止した場合にはこの通知方法を用いることはできない．そこで，GPUSentinel では，GPU カーネルがメインメモリ上の VRAM 領域に画像データを書き込むことで，OS の機能を利用することなく画面に画像や文字を出力する．障害を通知する画像は検知プログラムの起動時にビットマップ形式のデータを GPU メモリに転送しておく．一方，文字用の画像データとしては，Linux カーネル内の `font_vga_8x16` 変数に格納されているフォントデータを利用する．画面出力は IPMI や KVM スイッチなどのハードウェアを用いることにより，リモートホストからでも確認することができる．ただし，この通知機構を用いるには，

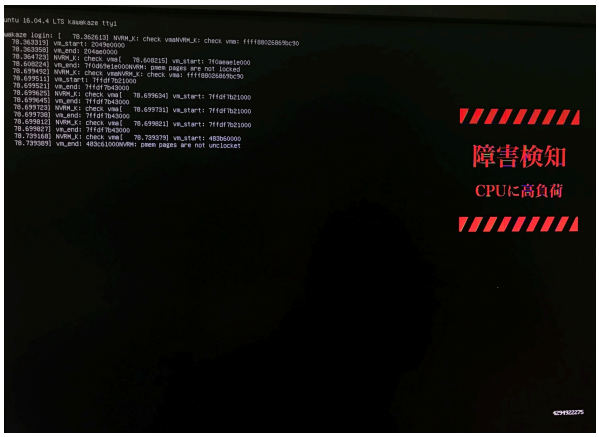

図 8: 障害検知を通知する画面

メインメモリ上に VRAM を配置する CPU 内臓 GPU などが必要である．

5. 実験

GPUSentinel を用いて，GPU 上の検知プログラムがシステムに発生した障害を検知できるかどうかを確認する実験を行った．比較として，既存の監視システムである Zabbix を用いて同様の実験を行った．また，GPUSentinel がシステムに与えるオーバヘッドおよびシステムから受けるオーバヘッドを測定する実験も行った．実験には，Intel Core i7-7700（4 コア，8 スレッド）の CPU，8GB の DDR4-2400 メモリ，GeForce GTX 960 の GPU，HD Graphics 630 の CPU 内臓 GPU を搭載したマシンを用いた．このマシンでは Linux 4.4.67，NVIDIA ドライバ 375.66，CUDA 8.0.61 を動作させた．実験を行う際には，リアルタイム出力機構を用いてログをファイルに出力させ，障害検知時には通知画像と Linux カーネルから取得した jiffies 変数の値を画面に表示するようにした．

5.1 CPU の高負荷状態の検知

GPUSentinel が CPU の高負荷状態を検知できることを確かめるために，CPU に負荷をかける処理を並列に実行するプログラムを実行した．このプログラムはまず 4 つのプロセスを生成して負荷をかけ，処理が終了するのを待つ．その後，jiffies 値を proc ファイルシステムから取得・表示してから，8 つのプロセスを用いて負荷をかけた．負荷をかけたまましばらく経つと，GPUSentinel によって画面に図 8 のような障害通知画像と jiffies 変数の値が表示された．

その間に GPUSentinel が計測した CPU 使用率の推移を図 9 に示す． 1 秒の時点で負荷をかけるプログラムの実行を開始し，17 秒の時点で 2 回目の負荷をかけ始めた．プログラムが表示した jiffies 値と高負荷検知時に表示された jiffies 変数の値から，2 回目の負荷をかけ始めてから 4.7 秒で高負荷を検知できたことがわかる．ただし，CPU の高負荷状態が検知されても正常な処理である可能性もあるた

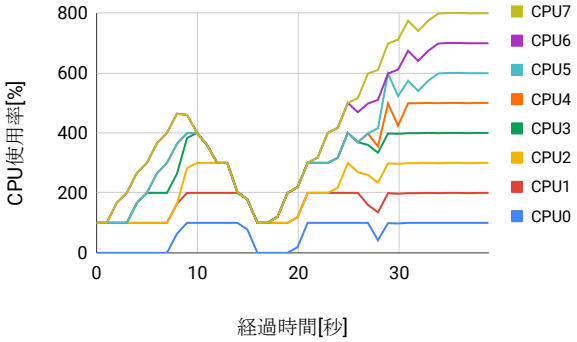

図 9: 各 CPU の使用率の推移

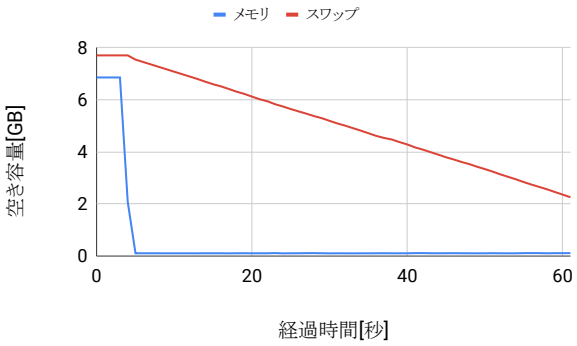

図 10: メモリとスワップの空き容量の推移

め，障害かどうかは管理者が総合的に判断する必要がある．

5.2 メモリ枯渇の検知

GPUSentinel がメインメモリの枯渇を検知できることを確かめるために，メモリを大量に使用するプログラムを実行した．このプログラムは jiffies 値を表示してから 1KB ずつメモリを確保して書き込みを行い，合計で 20GB のメモリを確保した．このプログラムを起動してしばらく経つと OS の応答に遅延が見られるようになり，さらに時間が経つと画面に障害通知画像と jiffies 変数の値が表示された．

GPUSentinel が計測した空きメモリ容量と空きスワップ容量の推移を図 10 に示す．3 秒の時点で負荷をかけるプログラムの実行を開始した．この実験ではリアルタイム出力機構を用いて 1 秒ごとにメモリとスワップの空き容量をログとして出力していたが，負荷をかけ始めて 1 秒後にログ出力が一時停止した．これは GPUSentinel のホストプロセスの動作が極端に遅くなり，ログ出力ができなくなったためと考えられる．この時に障害が発生したと考えると，その時の jiffies 値と障害検知時に表示された jiffies 変数の値から，GPUSentinel はメモリの枯渇を 57 秒で検知できたことが分かる．

5.3 デッドロックの検知

スピンロックを用いて CPU をデッドロックさせるカーネルモジュールを作成し，デッドロックさせる CPU の数を変化させて実験を行った．このカーネルモジュールは 2 つのスレッドがそれぞれスピンロックを獲得した後，指定した数のスレッドがそれらのスピンロックの解放を待つ．デッドロックに陥る直前に jiffies 変数の値を出力させた．

5.3.1 一部の CPU をデッドロックさせた場合

一部の CPU がデッドロックにしたことを検知できるかどうかを確かめるために，2 つの CPU をデッドロックさせるカーネルモジュールを実行した．その後しばらく経つと，画面に障害通知画像と jiffies 変数の値が表示された．デッドロック直前に表示された jiffies 値と障害検知時に表示された jiffies 変数の値から，GPUSentinel はデッドロックを 5.1 秒で検知できたことがわかる．

5.3.2 OS をハングアップさせた場合

OS をハングアップさせる障害を検知できることを確かめるために，すべての CPU をデッドロックさせるカーネルモジュールを実行した．実行開始直後に，表示させていたスクリーンセーバーが停止し，端末も反応しなくなり OS がハングアップ状態に陥った．その後，約 1 秒で障害通知画像が表示された．ハングアップ状態に陥った後は画面に表示されるコンテキストスイッチ数が変化していないかった．

デッドロック直前に表示された jiffies 変数の値と障害検知時に表示された jiffies 変数の値から，GPUSentinel は OS が停止する障害を 1.7 秒で検知できたことがわかる．その十数秒後に Watchdog タイマによりソフトロックアップ・メッセージが画面に出力された．このことから，GPUSentinel は Watchdog タイマより早く検知できたことがわかる．

5.4 Zabbix を用いた障害検知との比較

既存システムを用いて 5.1 節〜5.3 節の障害検知が可能かどうかを調べるために，Zabbix 3.0.25[3] を用いてリモートホストからの監視を行った．Zabbix サーバにデフォルトで用意されている Template OS Linux のテンプレートを用いて監視対象ホストを登録し，CPU 使用率，空きメモリ量，空きスワップ領域の更新間隔を 1 秒に変更した．

5.1 節のプログラムを実行したところ，CPU 負荷のグラフからロードアベレージが上昇したこと，および，CPU 使用率のグラフから負荷をかけた約 2 分間で最大で 60% の CPU 使用率になったことが確認できた．このように，CPU が高負荷な状態でも Zabbix サーバに正常に情報を送信できていた．しかし，Zabbix では CPU 使用率を最小でも 1 分間の平均で算出するため，高負荷状態が 2 分間程度続いてもそれを検知することはできなかった．

5.2 節のプログラムを実行したところ，実行開始から 14 秒後にスワップ領域が少なくなっているという警告がダッシュボードに表示された．Zabbix サーバが取得した空きメモリ容量と空きスワップ容量を図 11 に示す．これは負荷をかけ始める 10 秒前からのデータであり，95 秒の時点

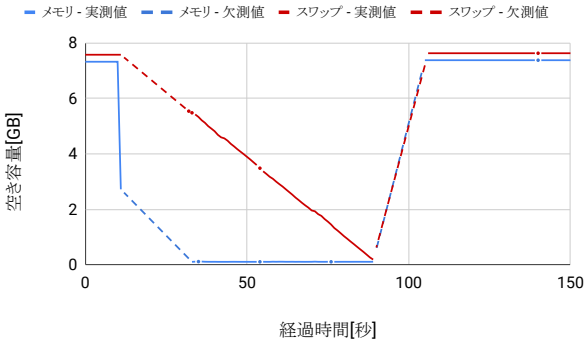

図 11: Zabbix から確認された空きメモリ量のグラフ

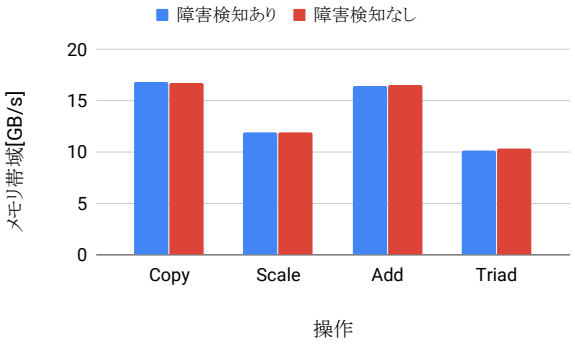

図 12: GPUSentinel によるメモリ帯域への影響

で OOM Killer により負荷をかけたプログラムが終了させられた．12 秒の時点からの 19 秒間と 90 秒の時点からの 14 秒間，その他 1 秒間ずつ 5 箇所ではデータが受信できなかった．これは，メモリの圧迫によりページングが多発してシステムの性能が低下したためと考えられる．このことから，Zabbix ではメモリの枯渇を正確に検知するのは難しいことが分かった．

2 つの CPU をデッドロックさせるカーネルモジュールを実行したところ，CPU 使用率のグラフにおいてシステム時間の割合が 20%まで上昇して安定した．CPU ごとのシステム時間とユーザ時間の割合を送信するように設定することで，2 つの CPU のデッドロックを検知できると考えられる．

すべての CPU をデッドロックさせるカーネルモジュールを実行したところ，全グラフの更新が停止した．この状態のままにしておくと実行から 5 分後に監視対象ホストにアクセスできないというアラートがダッシュボードに表示された．監視対象ホストで何らかの障害が発生していることは確認できたが，内部でどのような障害が発生しているかは分からなかった．このことから，Zabbix では OS のハングアップの原因がデッドロックであることを特定できないことが分かった．

5.5 オーバヘッドの測定

監視対象システム上でベンチマークを実行し，GPUSentinel がシステムに与える影響および，システムが GPUSentinel に与える影響を調べた．メモリについては STREAM ベンチマークを用い，CPU については UnixBench ベンチマークの Dhrystone を用いた．この実験では，4.4 節の検知プログラムに加え，1 スレッドあたり 4KB をメインメモリの特定のアドレスから GPU メモリにコピーする GPU カーネルを用いた．

5.5.1 STREAM

検知プログラムを動作させている時にメインメモリにどの程度の負荷がかかっているかを調べるために，検知プログラムを動作させている時とさせていない時で STREAM

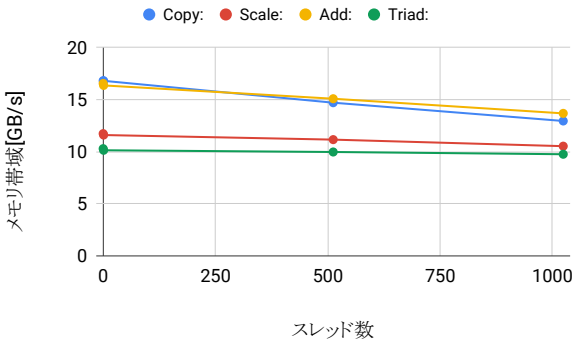

図 13: スレッド数を変化させた時のメモリ帯域への影響

を 10 回ずつ実行した．4 種類の操作を行った際のメモリ帯域を図 12 に示す．この結果より，検知プログラムのオーバヘッドはほぼないことを確認できた．用いた検知プログラムは 3 スレッドだけを使って障害検知を行っており，DMA で転送されるメインメモリのデータが少なかったためオーバヘッドが小さくなったと考えられる．

そこで，メモリコピープログラムのスレッド数を増やしながら STREAM を実行してオーバヘッドを測定した．その時の帯域の変化を図 13 に示す．スレッド数を増やすとメモリ帯域が減少していき，GPUSentinel のオーバヘッドが増大していくことが分かる．1024 スレッドの時に Copy 処理で 33%，Scale 処理で 10%，Add 処理で 17%，Triad 処理で 5%のオーバヘッドが見られた．実際の検知プログラムではメインメモリからデータを取得してさらに解析処理を行う場合が多いため，メインメモリへのオーバヘッドは一般的にこの結果より小さいと考えられる．

逆に，システムがメインメモリの帯域を圧迫している時の GPUSentinel への影響を調べるために，メモリコピープログラムのスレッド数を増やしながらその実行時間を測定した．その結果を図 14 に示す．この結果から，スレッド数が増大するほど GPUSentinel はメモリ帯域の圧迫の影響を受けるようになることが分かる．1 スレッドの時にはほぼオーバヘッドがなかったが，1024 スレッドになると 20%のオーバヘッドになった．

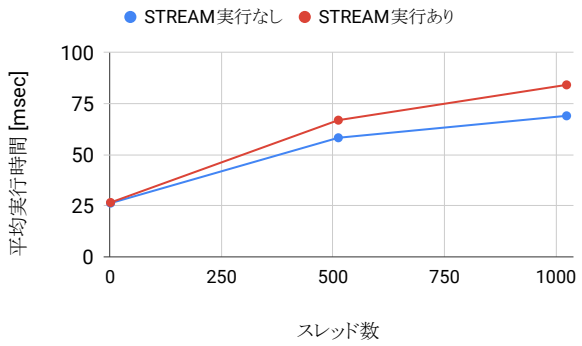

図 14: メモリ帯域の圧迫による GPUSentinel への影響

5.5.2 UnixBench

まず，GPUSentinel の検知プログラムを動作させている時とさせていない時で Dhrystone を 10 回ずつ実行した．その結果により，検知プログラムによる CPU へのオーバヘッドは 0.5%であった．これはホストプロセスがリアルタイム出力機構のバッファを 1 秒ごとにポーリングするオーバヘッドと考えられる．次に，GPU カーネルのスレッド数を 1024 まで増やしながらメモリコピープログラムを動作させて Dhrystone を実行した．その結果よりスレッド数を 1024 にしても CPU にほぼ影響がないことを確認した．

6. 関連研究

最小限の性能指標だけを用いてシステム障害を検知して復旧を行う SHFH[4] が提案されている．SHFH はシステム障害の原因を無限ループ，デッドロック，リソース枯渇などの 6 種類に分類し，障害検知に必要な CPU，プロセス，メモリ，ディスク I/O に関する性能指標を見つけている．SHFH はリアルタイム・ユーザプロセスとカーネルモジュールとして実装され，通常時はプロセスを用いた軽量な検知処理を行い，異常検知時にカーネル内で詳細な情報収集を行う．そのため，OS が動作しなくなる障害が発生すると検知を行うことができなくなる．

OS 内で障害を検知するカーネルレベル障害検知機構 [5] が提案されている．障害検知をカーネル空間で動作させることにより，オーバヘッドを軽減しつつ多様な障害検知を可能にしている．この機構はカーネルタイマを使って定期的に障害検知処理を行い，カーネルデータから CPU 使用率，メモリ使用率，プロセス情報を取得する．しかし，OS が正常に動作しなくなると障害検知が行えなくなる．

カーネル間メモリ監視による障害検知機構 [6] では，Orthros を用いて 1 台の計算機上で 2 つの OS を動作させ，監視対象となる ActiveOS のメモリ上に存在するカーネルパラメータをもう一方の BackupOS から監視する．そのために，2 つの OS 間に共有メモリを用意し，ActiveOS が書き込んだ監視対象のパラメータを BackupOS が読み取る．ActiveOS に障害が発生しても BackupOS で障害を検知できるが，障害の種類によっては BackupOS も機能しなくなる．また，監視対象 OS を修正する必要もある．

これまでに，攻撃の影響を受けずにシステムを安全に監視するための様々な機構が研究されてきた．それらの機構を用いることで，システム障害の影響を受けにくい障害検知システムを構築することも可能である．Copilot[7] は専用の PCI カードを用いて DMA でカーネルメモリの内容を取得し，リモートホストで監視を行う．SPE Observer[8] は Cell/B.E. プロセッサを用いて，OS が動作する PPE から隔離された SPE 上で監視システムを実行し，DMA でカーネルメモリを取得する．しかし，いずれも専用ハードウェアや普及していないプロセッサを用いるため，一般的な利用は難しい．

Intel の汎用 CPU を用いて，システムから隔離された環境で監視システムを動作させることもできる．HyperGuard[9] は CPU のシステムマネジメントモード（SMM）を用いて，メインメモリから隔離された SMRAM 上で監視システムを実行する．HyperCheck[10] は SMM で NIC のドライバを動作させ，メインメモリの内容をリモートホストに転送して監視を行う．HyperSentry[11] は SMM を用いることでハイパーバイザ内で安全に監視システムを実行する．一方，Flicker[12] は Intel TXT を用いて安全に監視システムを実行する．しかし，SMM は低速であり，監視中はシステムを停止させる必要がある．TXT でも監視システムを実行する CPU コア以外を停止させる必要がある．

VM を用いたシステムの場合には，VM イントロスペクション [13] を用いることで，VM の外側からシステムの詳細な情報を取得することができる．これにより，障害に強く，検知能力の高い障害検知システムを構築することができる．しかし，VM を用いない場合には利用することができず，ハイパーバイザやホスト OS の障害検知には利用できない．GPUSentinel は VM イントロスペクションの技術を GPU に適用したものである．

7. まとめ

本稿では，監視対象ホストの GPU 上で検知プログラムを動作させて障害検知を行うシステム GPUSentinel を提案した．GPUSentinel では，GPU からメインメモリ上の OS データを監視することにより障害を検知する．Linux カーネルおよび GPU ドライバに変更を加えることで，マップトメモリ機能を用いてメインメモリ全体をロックすることなく GPU にマップできるようにした．また，検知プログラムの作成を容易にするために，透過的にアドレス変換を行うことで OS のソースコードを最大限に利用可能にするフレームワーク LLView を開発した．GPUSentinel を用いて OS が停止する障害を含む 4 種類の障害を検知する検知プログラムを作成し，それぞれの障害を検知できることを確認した．また，作成した検知プログラムのオーバヘッド

はほぼないことを確認した．

今後の課題は，実装している検知プログラムではシステム構成によっては誤検知する可能性があるため，より多くの情報を基に総合的に障害を検知できるようにすることである．また，検知できる障害の種類を増やすことでGPUSentinel のさらなる有効性を示すことも必要である．将来的には，OS のデータを書き換えることで障害からの復旧を行えるようにすることも検討している．

参考文献

[1] 松田晃一，目黒達生：情報システムの障害状況 2017 年前半データ，SEC journal, Vol. 13, No2，pp. 52–58, 2017.
[2] 金本颯将, 光来健一：GPUDirect RDMA を用いたリモートホストの異常検知手法，情報処理学会研究報告，Vol.2018-OS-144，2018.
[3] Zabbix LLC: Zabbix :: The Enterprise-Class Open Source Network Monitoring Solution, https://www.zabbix.com/.
[4] 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 Proc. Int. Symp. Software Reliability Engineering, pp.141–150, 2012.
[5] 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，情報処理学会研究報告，Vol.2016-OS-136，No.7，pp. 1–9，2016.
[6] 松下 馨, 岩間 響子, 瀧本 栄二, 毛利 公一, 齋藤 彰一：多重 OS 実行環境に置けるカーネル間メモリ監視による障害検知機構の実装，情報処理学会研究報告，Vol.2017-OS-139，No.2，pp. 1–8，2017.
[7] N. Petroni, Jr., T. Fraser, J. Molina, and W. Arbaugh: *Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor,* in Proc. Conf. USENIX Security Symp., 2004.
[8] K. Kourai and T. Nagata: *A Secure Framework for Monitoring Operating Systems Using SPEs in Cell/B.E.,* In Proc. Pacific Rim Int. Symp. Dependable Computing, pp.41–50, 2012.
[9] J. Rutkowska, R. Wojtczuk, and A. Tereshkin: *Xen 0wning Trilogy,* Black Hat USA, 2008.
[10] J. Wang, A. Stavrou, and A. Ghosh: *HyperCheck: A Hardware-Assisted Integrity Monitor,* in Proc. Int. Symp. Recent Advances in Intrusion Detection, pp. 158-–177, 2010.
[11] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky: *HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity,* in Proc. Conf. Computer and Communications Security, pp. 38-–49, 2010.
[12] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki: *Flicker: An Execution Infrastructure for TCB Minimization,* in Proc. European Conf. Computer Systems, pp. 315—328, 2008.
[13] T. Garfinkel and M. Rosenblum, *A Virtual Machine Introspection Based Architecture for Intrusion Detection,* in Proc. Network and Distributed Systems Security Symp., pp. 191-–206, 2003.