Vol.2018-HPC-163 No.8
2018/3/1

# メモリアクセスパターン依存故障の注入のためのQEMUベース故障注入器

小林 佑矢[1,2,a)]　實本 英之[1,b)]　野村 哲弘[1,c)]　松岡 聡[1,2,d)]

**概要**：近年，大規模計算機システムでの故障の増加が問題になっている．様々な耐故障技術が開発されたが，オーバーヘッドと耐故障性の間にトレードオフがあり，適切な手法の選択が難しい．耐故障技術の評価には故障注入が用いられるが，既存の故障注入器ではハードウェア特有の故障を再現し，アプリケーションレベルの詳細な解析ができない．
我々は，ハードウェアに特有の故障を再現し，アプリケーションレベルの解析を容易に行える故障注入プラットフォームの提供を目的とする．本研究では，故障の主な発生源の一つである DRAM を対象にして，メモリアクセスパターン依存故障をメモリ I/O フックにより注入した上で，耐故障性テスト対象のプロセスのメモリマップ情報を取得できる QEMU ベース故障注入器 MH-QEMU を作成した．評価では，MH-QEMU の故障注入機能使用時には，テスト対象アプリケーションの実行時間が最良の場合でも 77 倍になることを確認した．中でも，メモリインテンシブであったり，ノード間通信が少ないアプリケーションほどオーバーヘッドが大きいことを確認した．また，MH-QEMU で NAS Parallel Benchmark の CG カーネルの耐故障性評価を行い，Row-Hammer 発生時に Silent Data Corruption につながりやすいデータ領域を特定した．

**キーワード**：耐故障, フォールトトレランス, 故障注入, QEMU, Silent Data Corruption, Row-Hammer

## 1. はじめに

近年，大規模計算機システムでは，部品の高密度化やシステムの大規模化などにより，故障の増加が問題になっている [5]．故障の増加に伴い，SDC（Silent Data Corruption）の増加も懸念されている．SDC とは，アプリケーションに故障が発生した際に，異常終了やハングアップなどの明らかな障害が発生せず，誤った計算結果のみが出力される問題である．SDC 特有の問題として，特別な技術がなければ検出できない事がある．こうした中で，以前よりも耐故障技術の重要性が増している．

これまで開発された耐故障技術にはチェックポイントリスタートやリプリケーション，アルゴリズムベース耐故障技術，ハードウェアベース耐故障技術など様々な種類がある．しかし，それぞれオーバーヘッドや耐故障強度，保護対象などが異なるため，実システムにおいて最適な耐故障技術を選択することが難しい．各隊故障技術を評価するためには，アプリケーションなどに意図的に故障を再現する故障注入が用いられる．しかし，既存の故障注入器では，Row-Hammer などのハードウェアに特有の故障を注入し，アプリケーションレベルの解析を行うことができない．

我々は，ハードウェアに特有の故障を再現し，アプリケーションレベルの解析を容易に行える故障注入プラットフォームの提供を目的とする．本研究では，故障の主な発生源の一つである DRAM を対象にして，メモリアクセスパターン依存故障を注入した上で，耐故障性テスト対象のプロセスの情報を取得できる VM ベース故障注入器 MH-QEMU を作成した．MH-QEMU は VM エミュレータである QEMU に対し故障注入機能として，メモリアクセスパターン依存故障を注入できる MH 機能と，ランダムビットフリップを注入できる MFM 機能を追加して実装した．また，ゲストマシン上の耐故障性テスト対象のプロセスの情報を取得するための ADM 機能も実装した．さらに，MH-QEMU　の故障注入機能による性能オーバー

[1] 東京工業大学
Tokyo Institute of Technology
[2] 産総研・東工大 実社会ビッグデータ活用 オープンイノベーションラボラトリ
AIST-TokyoTech Real World Big-Data Computation Open Innovation Laboratory (RWBC- OIL), National Institute of Advanced Industrial Science and Technology (AIST)
[a)] kobayashi.y.bk@m.titech.ac.jp
[b)] jitumoto@gsic.titech.ac.jp
[c)] nomura.a.ac@m.titech.ac.jp
[d)] matsu@is.titech.ac.jp

ヘッドを測定したところ，MH では最良の場合でもゲストマシン上のアプリケーションの実行時間が 77 倍になった．MH によるオーバーヘッドはアプリケーションごとにことなり，メモリインテンシブでノード間通信の少ないアプリケーションほど性能低下が大きくなった．一方，MFM によるオーバーヘッドは最悪の場合でも 10% となり，最良の場合では 0% で，故障注入テストでは無視できるほど小さいことを確認した．加えて，MH-QEMU を用いて NPB（NAS Parallel Benchmark）の CG カーネルの耐故障性評価を行い，故障発生時に SDC につながりやすいデータ領域を特定した．

## 2. DRAM 故障パターン

この節では，本研究が対象としている DRAM における故障パターンについて説明する．故障の種類が異なると計算結果への影響が異なるため [1]，耐故障性評価では様々な故障を再現することが望ましい．

### 2.1 DRAM の構成

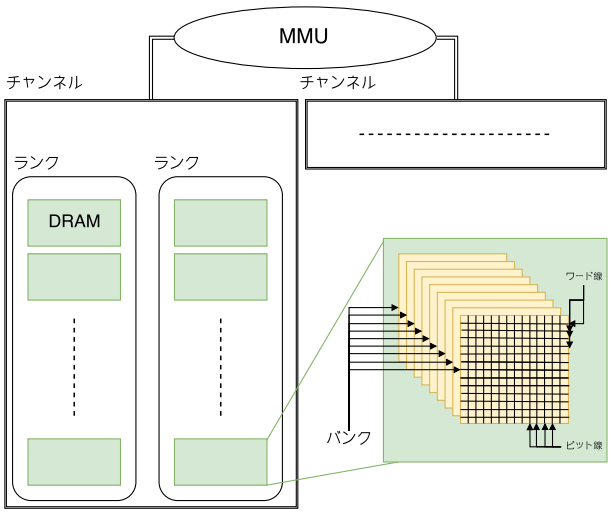



**図 1:** DRAM

ここでは，本研究の対象ハードウェアである DRAM の故障を説明するために必要な，DRAM の構造を述べる（図 1）．DRAM チップは DIMM 上にまとめられ，DIMM 単位でシステムに搭載される．DRAM 上のビットは，対応するチャンネル・ランク・バンク・行・列を指定してアクセスする．これらのインデックスは階層を成している．チャンネルはメモリコントローラが一度に制御できる DIMM の集合であり， ランクは同じロックステップで動作する DRAM チップの集合である．よって， チャンネルとランクを指定することで DRAM チップを指定できる．バンクは DRAM チップ内に複数個存在するメモリセル列それぞれに対応する．一つのメモリセル列ではメモリセルが格子状に並んでいる．縦の線をビット線，横の線をワード線と呼び，それらの交点にメモリセルがある．行・列のインデックスはこのビット線・ワード線を指定する．よって，バンク・行・列のインデックスを指定することで， DRAM チップ内でアクセスするメモリセルを指定できる．

メモリにアクセスする際には，対応するビット線とワード線が印加され対象のビット値が読み書きされる．

### 2.2 故障の種類

ここでは実際にこれまで実システムやハードウェア上で観測されている故障を紹介する．ここで紹介する故障を MH-QEMU では注入できるようにする．

#### 2.2.1 ソフトエラー

ソフトエラーは宇宙線や $\alpha$ 線などを原因として，無作為なビットが反転する一時的故障である．ソフトエラーは実システムでも最も多く観測される故障である [16]．故障注入では一般的に採用され，発生時間も発生箇所も無作為に選択されることが多い．

#### 2.2.2 メモリアクセスパターン依存故障

ハードウェア構造に基づき，メモリアクセスに依存する故障パターンが存在する．その一例として Row-Hammer と呼ばれる，DRAM に特有で， 一本のワード線が短時間に何度もアクセスされたとき， 隣接するワード線上のビットが消失する一時的故障がある．DRAM の高密度化によりメモリセル同士が互いに干渉しやすくなり，この問題が近年になって報告された．Kim ら [10] によれば，Row-Hammer の発生には，同じワード線への短時間アクセスが非連続で，ワード線の電圧が変動しなくてはならない．また，メモリセルごとにビット消失の確率が異なり，一度ビットが消失したメモリセルでは再発の可能性が高い．

#### 2.2.3 Functional Memory Fault

故障モデルの一つに, Functional Memory Fault Model [4] がある．これは，メモリの各部品ごとに役割を設け，その役割の誤りを故障としてモデル化する．Functional Memory Fault Model では，1) 部品の論理値がある値に固定され，値の変更ができなくなる永続的故障である縮退故障，2) ある部品の値の遷移により，他の部品の値も変わる故障であるカップリング故障，3) ある部品の値が別の部品の値により変わる故障であるブリッジ故障，などが定義されている．

## 3. 関連研究

ここでは既存の故障注入器を紹介する．故障注入器には，ソフトウェア故障注入器，ハードウェア故障注入器がある．

### 3.1 ソフトウェア故障注入器

#### 3.1.1 コード改変型故障注入器

アセンブリやソースコード，LLVM-IR に専用コードを挿入して故障注入するツールがある [11].

LLFI [17] は LLVM-IR に故障注入用コードを挿入して故障を再現する．ソースコードを変更せずに故障を再現でき，ソースコードレベルの詳細な情報が取得できる利点がある．LLFI の対象の故障は，CPU 上で実行される各命令の結果への故障である．

コード改変型故障注入器はソースコードの情報を取得でき，詳細な解析を可能にする．しかし，ハードウェアの構成を認識できず，ハードウェア特有の故障やマシンの動作に依存する故障を再現できない．

#### 3.1.2 シミュレータベース故障注入器

仮想マシンエミュレータである QEMU や gem5 に機能を追加して作成した故障注入器もある [6, 15] .

D-Cloud [8] は QEMU をベースとした故障注入器 FaultVM をクラウド管理ソフトウェアである Encalyptus [14] により管理することで，容易に大規模並列な故障注入を可能にしている．さらに，FaultVM には独自デバイスの追加が可能になっている．FaultVM では故障注入のために，I/O フック方式と仮想化ハードウェアデバイスの状態変更方式をとっている．ターゲットデバイスは，ハードディスクコントローラ，ネットワークコントローラとメモリである．

シミュレータベース故障注入器の利点としては，一つのホストマシンの上で，様々なハードウェアやアーキテクチャについて故障注入テストが行える事がある．しかし，プログラムに関する情報の取得が難しく，保護すべきデータの選択などの際に必要な詳細な解析ができない．

### 3.2 ハードウェア故障注入器

ハードウェア故障注入器は実ハードウェア上で故障を引き起こす．一時的を再現するためには，重イオン線照射 [7] や中性子砲 [13]，電磁場 [9] を用いる故障注入器がある．これらの故障注入器はハードウェアに接触することなく一時的故障を発生される．一方で，故障注入用のハードウェアによる故障注入器も開発されている [2, 12]．これらの故障注入器は論理回路に直接作用し，縮退故障やブリッジ故障など様々な故障を注入できる．

ハードウェア故障注入器の利点として，故障注入が高速であり，故障の粒度が比較的細かいことある．しかし，プログラムの情報を得られず，アプリケーションレベルの解析には適さない．

## 4. MH-QEMU の設計

節 3 で述べたように，既存の研究では Row-Hammer のようなハードウェアに依存する故障を注入した上で，アプリケーションレベルの詳細な解析をし，破壊されたデータ領域を特定する故障注入器がなかった（表 1）．そのため，現実に発生する故障に対して，保護すべきデータ領域や採用すべき耐故障技術の故障注入による選択が困難だった．

表 1: 関連研究の比較

| | ハードウェア特有故障 | プロセス情報 |
|---|---|---|
| LLFI | | レ |
| D-Cloud | レ | |
| Hardware Fault Injector | レ | |

本研究ではこの問題の解決のために，ハードウェアに依存する故障を注入でき，故障の解析を容易に行える故障注入プラットフォームの確立を目標にしている．そのためにまず，故障が主に起こる部品の一つであるメインメモリを対象にして，メモリアクセスパターン依存故障を注入でき，破壊されたデータ領域を特定するための故障注入器 MH-QEMU を開発する．

MH-QEMU は QEMU に追加の機能を実装した，VM（仮想マシン）ベースの故障注入器である．QEMU を採用することで，ハードウェアの構造を認識でき，ハードウェアに依存する故障注入を行える．さらに，実ハードウェアと異なり，テスト環境の管理・変更が容易になる．

MH-QEMU に実装する機能は，一つが VM のメインメモリに故障を注入する機能であり，もう一つは故障発生箇所のデータ領域を特定する機能（ADM）である．

### 4.1 故障注入機能

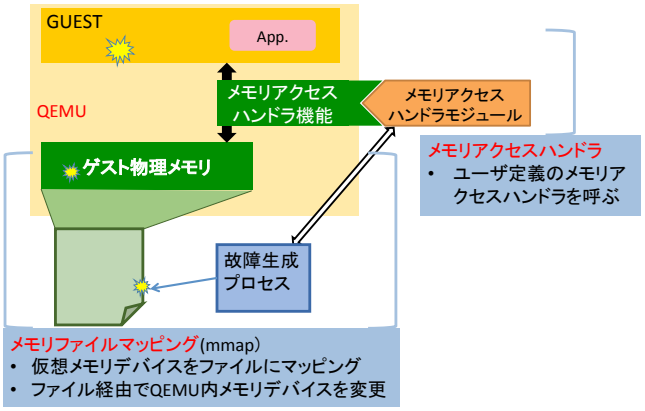



図 2: MH-QEMU の故障注入機能

MH-QEMU が対象とする故障は，節 2.2 で紹介した，メインメモリにおける一時的故障，永続的故障，メモリアクセスパターン依存故障である．MH-QEMU では，QEMU が管理しているゲスト物理メモリ（ゲストマシンの物理メモリ）への操作を制御し，故障注入を行う．各故障を注入するためには，次の要件を満たす必要がある．

**一時的データ変更** 一時的故障の再現には，アプリケーション実行中にデータをビット単位で変更でき，かつ

データの上書きがその変更を打ち消す必要がある．

**永続的データ変更**　永続的故障の再現には，アプリケーション実行中に，データをビット単位で変更でき，かつデータの上書きによるデータ変更が不可能である必要がある．

**メモリアクセスパターンに応じたデータ変更**　メモリアクセスパターン依存故障の再現には，メモリアクセスパターンを収集でき，メモリアクセスパターンに応じて一時的・永続的データ変更ができる必要がある．

これらを実現するために，「メモリファイルマッピング（MFM)」と「メモリアクセスハンドラ（MH)」を QEMU に追加する（図 2)．

#### 4.1.1 メモリファイルマッピング（MFM）

MFM は「一時的データ変更」のための機能である．MFM を使用すると，VM の物理メモリがホスト OS のファイルにマップされる．これにより，ファイルを通して VM の物理メモリの内容を読み書きすることができる．また，MFM 自体に故障を注入する能力はないため，故障注入の際には，ファイルの内容を変更するためのプログラムなどを用意する必要がある．しかしこれにより，ユーザがメモリ破壊パターンを柔軟に決定できる利点がある．

#### 4.1.2 メモリアクセスハンドラ（MH)

MH は「永続的データ変更」「メモリアクセスパターンに応じて一時的・永続的データ変更」のための機能である．MH はゲストマシンからメインメモリへのアクセスがある度に，ユーザ定義のメモリアクセスハンドラを呼び出す．メモリアクセスハンドラでは，アクセスされるゲスト論理・物理アドレスなどのメモリアクセス情報を取得できる．さらに，アクセスされる値やアドレスの変更などもできる．メモリアクセスハンドラの定義は MH-QEMU にはなく，ユーザが共有ライブラリオブジェクトとして MH-QEMU に渡す必要がある．これにより，ユーザがメモリアクセスパターンに応じたデータ破壊パターンを柔軟に決定できる．MH により，特定のアドレスへのアクセスの際に読まれる値を変更することで「永続的データ変更」を再現できる．更に，メモリアクセスハンドラによりメモリアクセスパターンを収集し，ゲスト物理メモリの内容を変更することで「アクセスパターンに応じたデータ変更」も再現できる．

### 4.2 故障が注入された箇所のデータ領域を特定する機能（ADM）

ここでは，ADM 機能の設計を説明する．故障発生箇所のデータ領域の特定に際し，故障が注入されたゲスト物理アドレスが既知であることを前提とする．なお，ADM を利用するためには，ゲストマシンのメモリ管理構造体が既知である必要がある．

このとき，次の手順でゲスト物理アドレスを対応するデータオブジェクトに変換し，破壊されたデータオブジェクトを特定できる．

```
#vaddr−range,paddr−range
0x400000−0x401000,0x129c8000−0x129c9000
0x401000−0x402000,0x12a7c000−0x12a7d000
```

**(a)** プロセスのメモリページマッピング情報

```
#vaddr−range,offset,file
0x1681000−0x16a2000,0,[heap]
0x7f984fa6b000−0x7f984fc23000,0,/usr/lib64/libc−2.17.so
0x7f985056f000−0x7f9850570000,0,
0x7ffe3d2d3000−0x7ffe3d356000,0,[stack]
```

**(b)** ファイルマッピング情報

**図 3:** ADM により出力されるファイル例

(1) プロセスのメモリページマッピング情報を探索し，ゲスト物理アドレスをゲスト論理アドレスに変換する．
(2) プロセスのファイルマッピング情報から，先程取得したゲスト論理アドレスに対応するメモリ領域を特定する．
(3) メモリ領域が実行ファイルや共有ライブラリに対応するならば，領域内オフセットから対応するデータをファイルから取得する．メモリ領域がスタックやヒープならば，メモリダンプをし，バックトレースなどで解析をおこなえばいい．

このためには，「プロセスのメモリページマッピング情報」「プロセスのファイルマッピング情報」「プロセスが使用しているファイルの内部構造の情報」が必要になる．ADM ではゲストメモリをホストマシン上にある QEMU 内部から直接操作し「プロセスのメモリページマッピング情報」「プロセスのファイルマッピング情報」を取得する．「プロセスのメモリページマッピング情報」としては図 3a のように，ゲスト物理アドレスとゲスト論理アドレスの対応が取得できる．「プロセスのファイルマッピング情報」としては図 3b のように，ファイルがマップされているゲスト論理アドレスの範囲とファイル内オフセット，ファイルパスが取得できる．この内容は，ゲストマシン上の `/proc/<PID>/maps` の内容と同じである．

## 5. 実装

ここでは，MH-QEMU の各機能の実装を説明する．MH-QEMU の基となる VM エミュレータ QEMU のバージョンは 2.3.1 である．

### 5.1 メモリファイルマッピング（MFM）

MFM では `mmap()` を用いてホストマシン上のファイルを QEMU のメモリ空間にマップし，その領域をゲスト物理メモリとして設定する（図 4)．これによりファイルを通してゲスト物理メモリの内容を操作できる．

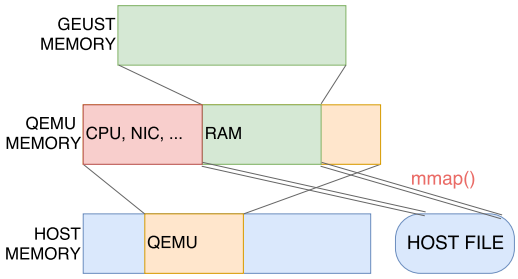



図 4: MFM におけるメモリマッピング

MFM を使用するためには，VM 起動時にコマンドラインオプション -mem-path でマップされるファイルを指定する．

### 5.2 メモリアクセスハンドラ（MH）

QEMU の TCG を拡張し MH を実装した．TCG は QEMU の CPU エミュレーション機構の一つであり，ゲストコードを独自の中間コードを介してホストコードに変換する．MH-QEMU では中間コードのロード命令とストア命令をホストコードへ変換する部分を拡張し，メモリアクセス時にメモリアクセスハンドラ関数を呼び出すホストコードを生成するように拡張した．

MH を使用するには，コマンドライン引数か QMP/HMP でメモリアクセスハンドラモード，メモリアクセスハンドラ関数をエクスポートする共有ライブラリへのパス，メモリアクセスハンドラ関数名を指定する必要がある．なお，QMP/HMP は QEMU で VM の稼働中に VM の操作をするためのインターフェースである．これにより，コマンドライン引数により VM 起動時から，QMP/HMP により VM 稼働中に，メモリアクセスハンドラ関数の登録や MH の無効化ができる．

### 5.3 ADM

ここでは，ADM の実装について説明する．ADM は Linux の構造体の探索を模倣し，必要な情報を取得する．そのため，まず Linux カーネル内でタスクのメタデータから「ページマッピング情報」「ファイルマッピング情報」を取得する方法を確認する．

#### 5.3.1 Linux Internals

ここでは Linux 3.10.105 の x86_64 バージョンを例にして，Linux 内でタスクのメタデータから対象の情報を取得する手順を説明する．

まず，Linux のタスクのメタデータは struct task_struct 構造体で表現される．struct task_struct を起点としてメンバ変数を辿ることで，タスクの親子関係やメモリ管理情報などが取得できる．

##### 5.3.1.1 特定のプロセスの struct task_struct の取得

Linux プロセスの struct task_struct は互いにつながり，循環リストを構成している（図 5）．そのため，あるプロセスの struct task_struct が取得できているならば，任意のプロセスの struct task_struct が取得できる．ある struct task_struct から次のプロセスの struct task_struct へのアドレスを取得するには，次の手順を踏めば良い．

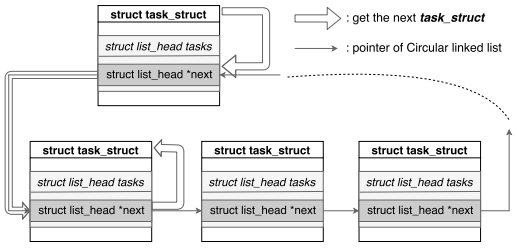
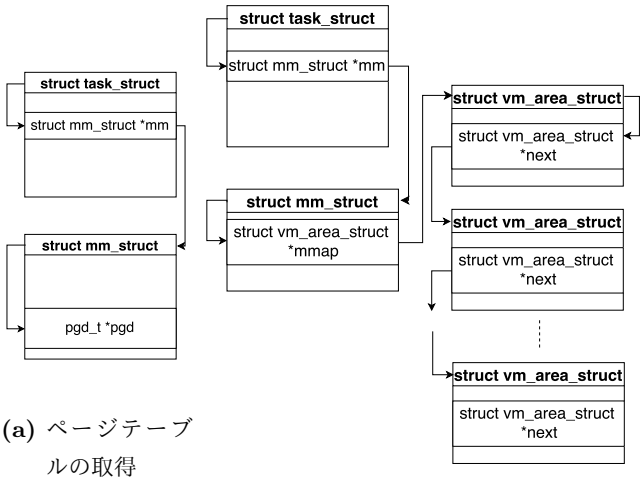



図 5: プロセスの struct task_struct の循環リスト

(1) struct task_struct のメンバ変数 struct list_head tasks のメンバ変数 struct list_head *next から次のプロセスの struct task_struct の，同様のメンバ変数 struct list_head *next のアドレスを取得する．

(2) 取得したアドレスからメンバ変数 struct list_head *next の struct task_struct 内のオフセットサイズを引くことで，該当の struct task_struct のアドレスを取得する．




(a) ページテーブルの取得

(b) ファイルマッピング情報の取得

図 6: Linux Internals

##### 5.3.1.2 ページマッピング情報

ページマッピング情報を取得するためには，対象プロセスの struct task_struct のメンバ変数 struct mm_struct *mm のメンバ変数 pgd_t *pgd が指すページテーブルのアドレスを取得し，ページテーブルを走査すればよい．

##### 5.3.1.3 ファイルマッピング情報

struct task_struct のメンバ変数 struct mm_struct *mm のメンバ変数 struct vm_area_struct *mmap は，そのプロセスにおけるメモリ領域一つ一つのメタデータの

線形リストを指している．struct vm_area_struct において線形リストの次の要素を指すメンバ変数は struct vm_area_struct *next であり，これを辿ることでそのプロセスの全てのメモリ領域のメタデータにアクセスできる．この手順で取得したメモリ領域のメタデータより，その領域に対応するファイルのパスやマップされているオフセットなどのファイルマッピング情報を取得できる．

#### 5.3.2 ADM による Linux カーネル内のデータの探索方法

節 5.3.1 で説明した Linux カーネルの動作を ADM が模倣するためには，次のことが必要である．

**VM のカーネルの仮想アドレスから物理アドレスへの変換** ゲストマシンのカーネル空間探索中に，ポインタ変数よりゲスト論理アドレスを取得した際に，該当データへアクセスするには，ゲスト論理アドレスをゲスト物理アドレスに変換する必要がある．これは ADM は MH-QEMU の一部なので，ADM のメモリアクセスにはホスト論理アドレスかゲスト物理アドレスが必要なためである．

**メンバ変数のオフセットの取得** 構造体内のメンバ変数にアクセスするに当たり，そのメンバ変数の構造体内のオフセットを特定する必要がある．

**メンバ変数のサイズの取得** メモリから内容を取得するに当たり，データサイズが既知である必要がある．

**一つのプロセスの struct task_struct へ予めアクセス可能** カーネルの動作の模倣に当たり，少なくとも 1 つのプロセスの struct task_struct のゲスト物理アドレスが特定されている必要がある．少なくとも一つのプロセスの struct task_struct にもアクセスできなくては，循環リストにすらアクセスできず，任意のプロセスの struct task_struct を取得できないため，ページマッピング情報やファイルマッピング情報も取得できない．

これらに対して，次のように対処した．

**VM のカーネルの仮想アドレスから物理アドレスへの変換** カーネル空間のページテーブルの物理アドレスを，Linux カーネルバイナリより取得する．再配置が行われなければ，ADM が VM のカーネル空間における仮想アドレスと物理アドレスの変換を行える．

**メンバ変数のオフセットの取得** Linux カーネルバイナリより取得する（後述）．

**メンバ変数のサイズの取得** これは，Linux ソースコードからメンバ変数の型を確認し，同じ型を ADM のソースコードにハードコードすることで対応した．

**一つのプロセスの struct task_struct へ予めアクセス可能** Linux のアイドルタスクの struct task_struct の物理アドレスを Linux カーネルバイナリより取得する．再配置が行われなければ，ADM はゲストマシン上の任意のプロセスの struct task_struct を取得できる．

Linux カーネルバイナリよりシンボルの物理アドレスを取得するためには，まず，バイナリのシンボルテーブルよりシンボルの論理アドレスを取得する．取得した論理アドレスをもとに，対象のシンボルを内包するセグメントを特定する．その後，対象シンボルのセグメント内オフセットと，セグメントの物理アドレスを取得し，足し上げることで，シンボルの物理アドレスを取得できる．

```
set logging file kernel_offset.csv
set logging on

macro define offset(t, f) &((t *) 0)->f)
printf "struct␣task_struct,tasks,0x%x\n", offset(struct
    task_struct, tasks)

quit
```

**(a)** GDB スクリプト

```
struct task_struct,tasks,0x430
```

**(b)** 図 7a の出力ファイル

**図 7:** GDB によるメンバ変数オフセット取得

また，ある構造体内のメンバ変数のオフセットは，GDB で Linux カーネルを読み込んだ上で，図 7a のコマンド offset により取得できる．

ユーザが ADM を利用するには，まず，これらの情報を VM 起動時に設定ファイルとして MH-QEMU にコマンドライン引数で渡す必要がある．その後は，QMP/HMP によりコマンドを発行するか，メモリアクセスハンドラ関数内から ADM 用 API を呼び出すことで，

## 6. 評価

ここでは，MFM や MH のオーバーヘッドを評価し，さらに，NPB（NAS Parallel Benchmarks） [3] の CG カーネルの耐故障性評価を MH-QEMU を用いて行う．

### 6.1 マシン・アプリケーション設定

ここではすべての評価で共通の設定を述べる．評価の際には 1 台のホストマシン上に MH-QEMU を用いて VM を 8 台稼働させ，その上でアプリケーションを実行する．VM はホストマシン上の TUN/TAP ネットワークを用いて相互に接続されている．

ホストマシンでは仮想化支援機能 VT-x が有効で 6 コア 1 2 スレッドもつ Intel X5650 CPU 2 つが 2.67GHz で動作する．メモリは ECC つき DDR4 SDRAM が 4 6 GB 分搭載されている．OS は CentOS 7.1 であり，カーネル

バージョンは 3.10.0 である．

VM の CPU は 1 コアの x86_64 CPU であり，メモリは 512 MB 分ある．OS は Scientific Linux 7.4 であり，カーネルバージョンは 3.10.0 である．

今回の評価には NPB のバージョン 3.3 の MPI 実装の計算カーネルを MPICH 3.2 と共に使用する．実行時には 1 ノード当たり 1 プロセスが割り当てられる．

### 6.2 MH-QEMU の速度評価

ここでは，MH-QEMU の故障注入機能の速度を，VM 上で実行した NPB の実行時間を測定し評価する．故障注入器のオーバーヘッドが大きいと，故障注入テストが長くなり実行できない可能性もあるため，故障注入器が高速なことは重要である．

#### 6.2.1 評価設定

表 2: MH-QEMU の設定

| MH-QEMU の設定名 | KVM | MFM | MH |
|---|---|---|---|
| **KVM**（Baseline） | レ | | |
| **MFM** | レ | レ | |
| **CLEAN** | | | |
| **MH** | | | レ |

MH-QEMU の速度評価には，表 2 の各設定の MH-QEMU による VM 上でアプリケーションを実行し，実行時間を測定する．基準は，VM を利用する際に標準的である **KVM** とする．MFM では，マップされるファイルを `tmpfs` などの高速なメモリベースファイルシステム上に置く．また **CLEAN** の KVM の無効化は，MH を有効にする際には KVM を使用できないため，KVM の無効化によるオーバーヘッドとその上での MH によるオーバーヘッドを測定するためである．なお，MH のメモリアクセスハンドラ関数は，何もしない空の関数である．

#### 6.2.2 評価結果

図 8a は **KVM** で正規化した，**CLEAN · MH** の実行時間を表している．この図が示すように，KVM を無効にした際には，アプリケーション間で性能低下が異なる．最良の場合の IS では実行時間が 24 倍になったが，最悪の場合である EP では実行時間が 127 倍になった．EP の実行時間が他よりも増加した理由としては，EP がノード間通信をしないためである．これは，I/O デバイスのエミュレーションが KVM の有効無効に関わらず QEMU の機構で行われ KVM 無効化による影響がないため，ノード間通信が多いほど他のオーバーヘッドが隠蔽される一方で，ノード間通信が少ないアプリケーションでは全体でのオーバーヘッドが大きくなるため．また，MG のオーバーヘッドも比較的大きくなっている．これは QEMU の TCG で，メモリアクセス命令が他の命令と異なり，メモリ管理ユニットも模倣するためにオーバーヘッドが大きくなり，メモリインテンシブな MG ではその影響が顕著に現れたためである．また，MH を有効にした際には，実行時間が最大で 266 倍となった．

また，MH の有効化によるオーバーヘッドを確認するために，**CLEAN** の実行時間で正規化した **MH** の実行時間を図 8b に示す．最悪の場合である CG では実行時間が 3.4 倍になり，最良の場合である EP では実行時間が 1.7 倍になった．特にオーバーヘッドが大きくなっている IS と CG には，メモリアクセスがイレギュラーであるという点が共通している．ただし，このオーバーヘッドは，KVM を無効化したオーバーヘッドと比較すると小さいため，メモリアクセスパターンよりも実行時間内の I/O やメモリアクセスの割合がより重要になることが確認できる．

図 8c は，**KVM** で正規化された **MFM** の実行時間を表す．MFM によるオーバーヘッドが MH よりも比較的小さいことがわかる．最も大きいオーバーヘッドでも 10% にとどまっている．

### 6.3 アプリケーションのデータ毎の耐故障性評価

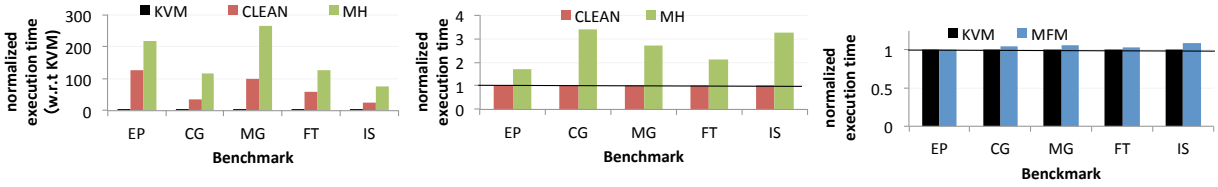
ここでは，実際に MH-QEMU を用いて，アプリケーションの耐故障性評価を行う．

対象とするアプリケーションは NPB の CG カーネルである．CG カーネルは逆冪乗法で正定値対称疎行列の絶対値最大固有値を求めるカーネルであり，逆冪乗法の各反復において共役勾配法を用いてる．逆冪乗法および共役勾配法が反復法であるため，計算の途中結果がビットフリップにより破壊されても，最終的には CG カーネルの計算結果が収束する可能性がある．ただし，NPB はベンチマークであるため，計算結果が十分に収束しても，規定回数の反復を終えるまで実行を続ける．今回はこれを変更し，逆冪乗法では計算結果の真値との誤差が NPB が定める値以下になったときに計算を終了するようにした．また，共役勾配法では残差が $10^{-20}$ 以下になったときに計算を終了するようにした．この値は，故障が発生しないときの逆冪乗法の反復回数が変化しないような閾値である．

注入する故障は Row-Hammer であり，アプリケーション実行中に全ノードを通して一度のみ故障が起きるように設定した．これは，MH-QEMU の MH を用いて次のように再現した．

(0) VM を起動し，テスト対象アプリケーションを実行する．

(1) アプリケーションの開始からランダムな時間が経過したら，全ノードに対し Row-Hammer 用のメモリアクセスハンドラを登録する．なお，このメモリアクセスハンドラが有効にされる時間は，時刻 0（アプリケーションの開始時刻）を下限とし，あらかじめ計測しておいたアプリケーションを KVM を利用しない VM 上




**(a)** KVM 無効化，MH によるオーバーヘッド　**(b)** KVM 無効から MH 有効化よるオーバーヘッド　**(c)** MFM によるオーバーヘッド

**図 8:** MH-QEMU におけるオーバーヘッド

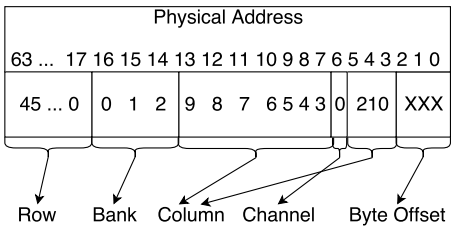



**図 9:** 物理アドレスとメモリシステムでのインデックスの変換ルール

で動かした際の実行時間を上限として，時間方向に一様ランダムに選択した．

(2) メモリアクセスハンドラにより，次のように故障が注入される．

(a) メモリアクセスがある度に，アクセスされるゲスト物理メモリを，対応するメモリ上のチャンネル・バンク・メモリ行に変換し，そのメモリ行にこれまでアクセスされた回数を記録する．なお，物理メモリから，対応するチャンネルやバンクに変換するに当たり，82955X-MCH のルールに従った（図 9）

(b) もし，そのメモリ行への総アクセス回数がある閾値（本評価では 1000 回とする）を超えていた場合は，ある確率（本評価では 0.0000000005 とする）で故障を発生させるかを決定する．これは，アクセス頻度の少ないメモリ行における Row-Hammer を発生させず，かつ，ある程度アクセスのあるメモリ行の内で Row-Hammer が発生するメモリ行をランダムに選択する操作である．故障発生をさせない場合には，メモリアクセスハンドラ関数を終了し，2a に戻る．故障を発生させる場合には，次のことをメモリアクセスハンドラ関数内で行う．

(i) ノード間排他制御を行う．もしもロックを全ノード中で初めて取得したのでないならば，故障を注入せずに 2(b)iv に進む．

(ii) ADM を利用し，VM 上のテスト対象のアプリケーションのプロセスのメモリマッピング情報を出力する．

(iii) アクセスされたメモリ行の隣のメモリ行上のセルをある確率（今回は 50%）で選択し，論理値を 0 にする．

(iv) MH を無効化する．以降は故障が注入されない．

以上の手順で 2443 回の故障注入を行う．

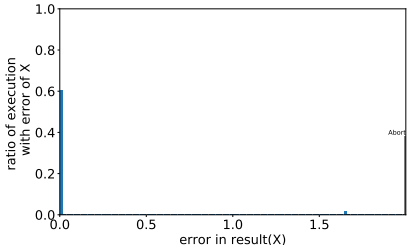



**図 10:** CG の計算結果の相対誤差の割合ヒストグラム（1%幅）

故障注入を行ったところ，CG の計算結果と真値との相対誤差の割合は図 10 のようになった．図はヒストグラムであり，各階級の幅はすべて 1% となっている．図中の右にある灰色の棒は異常終了の割合を示し，ハングアップや異常終了に加え，異常の検出が容易な NaN（Not a Number）が出力された場合も含む．エラーが 5% を超える計算結果を出力した場合を SDC とすると，SDC 発生時におけるエラーは約 160% に集中している．この原因としては次のことが考えられる．

- 全ノードの内一つのノードのみでデータ破壊が起きたため，プロセス間で同期されるデータの一貫性が破壊され，予期しない出力をした．
- CG カーネルにおける入力行列が故障により変化し，問題自体が変更され，新たな解へと収束した．
- CG カーネルが局所解へと収束した．しかし，逆冪乗法は局所解を持たないため，これは棄却される．

故障が注入されたデータ領域と障害の種類の関係を図 11 に示す．‘MPI’ は mpich のランタイムに使われるファイルがマップされた領域を示し，‘SYSTEM UTIL’ は `ld` などの，mpich や NPB 以外のファイルがマップされた領域を指す．また，‘ANONYMOUS’ は.bss セクションに対応しない無名メモリ領域を示す．障害の種類で Benign は故障

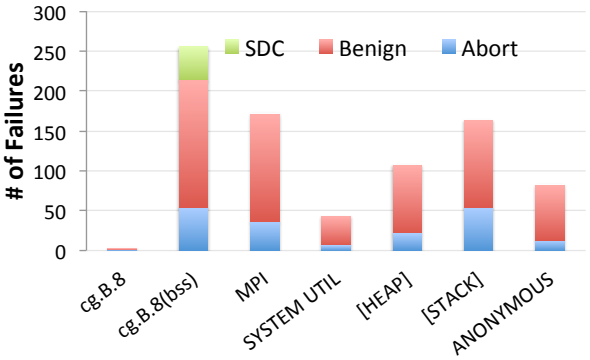



図 11: 破壊データ毎に生じた障害数

発生時でも正常な計算結果を返すことを示す．Abort は異常終了やハングアップ，NaN の出力を示す． `cg.B.8(bss)` には CG カーネルの入力行列や，途中の計算結果を格納する変数などある．表を見ると，cg.B.8(bss) に故障が注入された場合にのみ，SDC が発生している．そのため，さらなる詳細な解析は必要ではあるが，cg.B.8 の.bss セクションを中心にデータを保護することが妥当である可能性がある．また，`[heap]` や `[stack]` に故障が発生しても，SDC が発生していない．`[stack]` においては，ビットフリップが一度に複数発生する事により，関数のリターンアドレスが破壊され，異常終了へとつながる確率が高いことが考えられる．また，CG カーネルでは大きなデータの確保を `[heap]` では行わないため，アプリケーションの動作に `[heap]` における故障が影響する可能性が低かったことがあげられる．

## 7. まとめ

本研究では，故障注入器 MH-QEMU を作成した．MH-QEMU には故障注入機能として，メモリアクセスパターン依存故障を注入するための MH 機能と，ランダムビットフリップを注入するための MFM 機能を実装した．また，ゲストマシン上のテスト対象プロセスの情報を取得するための ADM 機能も実装した．これにより，従来行われていなかった，Row-Hammer による破壊箇所とその影響を関連付けられるようになった．評価では，MH-QEMU のオーバーヘッドを測定し，MH では実行時間が最良の場合でも 77 倍で最悪の場合では 260 倍になることを確認した．アプリケーションの特定との関連としては，I/O が多いアプリケーションやメモリインテンシブでない，もしくはメモリアクセスが非定型でないアプリケーションほど性能低下が低いことを確認した．また，MFM では KVM を利用できることもあり，オーバーヘッドが最良の場合で 0%，最悪の場合で 10% になり，比較的小さくなった．加えて，MH-QEMU を実際に利用して，NPB の CG カーネルに対する耐故障性評価をし，SDC につながりやすいデータ領域を特定できることを確認した．

今後は更なる耐故障性評価と MH-QEMU の拡張が必要になる．具体的には，耐故障性評価では，DRAM 以外の様々なメモリにおける故障の影響評価や実アプリケーションでの耐故障性評価，特定のメモリ領域だけを保護した際のアプリケーション全体での耐故障性評価がある．また，MH-QEMU では，故障注入を並列かつ容易に行うためのツールの開発や，I/O デバイスや CPU といったデバイスへの故障注入のサポートなどが必要である．

## 参考文献

[1] Fatimah Adamu-Fika and Arshad Jhumka. An investigation of the impact of double bit-flip error variants on program execution. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 799–813. Springer, 2015.

[2] Jean Arlat, Yves Crouzet, and J-C Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, pages 348–355. IEEE, 1989.

[3] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[4] Michael Bushnell and Vishwani Agrawal. *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*, volume 17. Springer Science & Business Media, 2004.

[5] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[6] Qiang Guan, Nathan BeBardeleben, Panruo Wu, Stephan Eidenbenz, Sean Blanchard, Laura Monroe, Elisabeth Baseman, and Li Tan. Design, use and evaluation of p-fsefi: A parallel soft error fault injection framework for emulating soft errors in parallel applications. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, pages 9–17. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

[7] Ulf Gunneflo, Johan Karlsson, and Jan Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, pages 340–347. IEEE, 1989.

[8] Toshihiro Hanawa, Hitoshi Koizumi, Takayuki Banzai, Mitsuhisa Sato, Shin'ichi Miura, Tadatoshi Ishii, and Hidehisa Takamizawa. Customizing virtual machine with fault injector by integrating with specc device model for a software testing environment D-cloud. *Proceedings - 16th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2010*, pages 47–54, 2010.

[9] Johan Karlsson, Peter Folkesson, Jean Arlat, Yves Crouzet, Günther Leber, and Johannes Reisinger. Application of three physical fault injection techniques to the experimental assessment of the mars architecture. *Dependable Computing and Fault Tolerant Systems*, 10:267–288, 1998.

[10] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[11] Dong Li, Jeffrey S Vetter, and Weikuan Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 57. IEEE Computer Society Press, 2012.

[12] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. Rifle: A general purpose pin-level fault injector. In *European Dependable Computing Conference*, pages 197–216. Springer, 1994.

[13] Sarah E Michalak, Andrew J DuBois, Curtis B Storlie, Heather M Quinn, William N Rust, David H DuBois, David G Modl, Andrea Manuzzato, and Sean P Blanchard. Assessment of the impact of cosmic-ray-induced neutrons on hardware in the roadrunner supercomputer. *IEEE Transactions on Device and Materials Reliability*, 12(2):445–454, 2012.

[14] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131. IEEE Computer Society, 2009.

[15] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D Antonopoulos, and Nikolaos Bellas. Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 622–629. IEEE, 2014.

[16] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *ACM SIGPLAN Notices*, volume 50, pages 297–310. ACM, 2015.

[17] Anna Thomas and Karthik Pattabiraman. Llfi: An intermediate code level fault injector for soft computing applications. In *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.