Vol.2018-SE-198 No.26
2018/3/10

サンプリングに基づく分散悪性競合のオンライン検出

片平遥香[†1,a)]　荒堀喜貴[1,†1,b)]　権藤克彦[1,†1,c)]

概要：分散システム固有のバグとして分散並行バグ（DCbug）の存在が明らかになっている [2]．DCbug の検出手法として，マルチスレッド競合検出手法の適用はスレッド数が極度に多い分散システムではスケールしない．計算コストを減らすサンプリング手法を用いても，全体のごく一部である DCbug を検出するのは難しい．DCbug を検出する先駆者的手法として DCatch[1] が提案されているが，小規模な実行トレースをオフラインで検査するにとどまっている．
本論文では，大規模分散システムに適用可能なオンライン DCbug 検出手法ポケットレーサを提案する．ポケットレーサは， データサンプリング，分散メタデータ簡約と呼ぶ二種類の最適化手法に基づく．データサンプリングは，システムの異常動作につながるフィールドアクセスがノード間通信に依存するかを識別することで DCbug の温床となるアクセスを重点的に調べるサンプリング法である．分散メタデータ簡約は分散システムに特化した *VectorClock* のバージョン管理方法ノードバージョン配列を導入し，*VectorClock* 配列全体へのコストの高い処理を削減することで，大規模分散システムにスケールする競合解析を実現する．実験では，代表的な四種類の分散データシステムに特徴的な通信パターンを捉えた合成ベンチマークに対してポケットレーサを適用するシミュレータを行った．実験結果としてポケットレーサは，既存のメインスレッドプログラムへの競合検出器に比べて同等以下の性能を維持し，DCbug の検出率は 4 倍以上となった．

キーワード：分散システム，並行バグ，データ競合，動的競合検出，オンライン解析，サンプリング，依存性解析

1. はじめに

分散並行バグ (Distributed Concurrency bug - DCbug) は大規模システムの中の計算/通信タイミングに依存して非決定的にごく稀に発生し，再現は容易ではない [2]．この性質によって，DCbug は深刻なバグであるにもかかわらずデバックや修正が難しくなっている．

この問題に対し，大規模分散システムに対応可能な世界初の動的競合検出器として DCatch[1] が考案された．DCatch は *Happens-Before* モデル [3] を分散システムに適応させ，オフラインで競合検査を行う．オフライン解析の問題点として，バグの早期発見が難しく，長期間運用することで起きるバグはトレースサイズが肥大化し解析できないことが挙げられる．これらの問題を解決するため，本研究は分散並行バグのオンライン解析手法ポケットレーサを提案する．

マルチスレッドプログラムにおけるオンライン競合解析手法は数多く提案されている．Pacer[6] は検査するフィールドアクセスをサンプリングすることで $O(n)$ の操作を削減し，時間/空間オーバーヘッドの削減を実現したが，サンプリング率と競合検出率が比例するため 1 度の実行で全体の数%の競合しか検出できない．ポケットレーサは Pacer のサンプリングのアイデアを踏襲しつつ，分散システムに特化したサンプリングを行うことで大規模な分散システムに対応可能なオンライン分散競合解析を目指す．

本研究での成果は次の通りである．

- マルチスレッド競合検出サンプリング手法を適用した世界初の DCbug オンライン検出手法の提案
- ノード間通信を考慮し，DCbug の温床となるアクセスを重点的に調べる追加のサンプリング手法データサンプリングの導入
- 分散システムに特化した，計算コスト削減のための *VectorClock* のバージョン管理方法であるノードバージョン配列の導入

評価として分散システムのシミュレータを作成し Pacer

[1] 情報処理学会
IPSJ, Chiyoda, Tokyo 101–0062, Japan
[†1] 現在，東京工業大学
Presently with Tokyo Institute of Technology
[a)] hkatahira@sde.cs.titech.ac.jp
[b)] arahori@cs.titech.ac.jp
[c)] gondow@cs.titech.ac.jp

とポケットレーサの精度と性能を比較する実験を行なった．実験ではポケットレーサは Pacer の 4 倍以上の DCbug を報告し，同等以下の時間/空間オーバーヘッドを維持した．

2. Background, Motivation, Challange

この章では，従来の分散/並行システム動的競合検出手法と，オンライン分散競合検出器の課題について述べる．

2.1 研究背景 DCbug と DCatch

DCatch は，大規模分散システムに対応可能な世界初の動的分散競合検出器である．この節では，DCbug と DCatch の概要を説明する．

2.1.1 DCbug

ノード間通信によって非同期的に発生するアクセスが引き起こす競合によるバグや，競合がノード外に伝搬し障害命令に至るバグを Distributed Concurrency bug(DCbug) と呼ぶ．DCbug の根本原因は Local Concurrency bug(LCbug) と同じ単一ノード内の競合であるため，深く研究されている LCbug 検出手法である *Happens-Before* モデルを使用できる．一方で *Happens-Before* 関係を構築する命令にはノード間通信も含まれ，また競合が発生したノードと致命的なバグが生じるノードが異なるなど，分散システム固有の特徴もある．

2.1.2 DCatch

DCatch は Lamport の *Happens-Before* モデル [3] を分散システムに拡張し，検出される競合から LCbug や無害なバグを取り除くことで DCbug を検出する世界初の動的分散競合検出器である．並行システムにおける *Happens-Before* モデルに加えて，RPC 関数，Socket 通信の送受信，分散同期システムを介した状態変化通知などに *Happens-Before* 関係を定義している．

この *Happens-Before* モデルで検出される競合のうち，多くがノード間通信のタイミングに依存しない LCbug やシステムの動作に影響しない無害な競合である．DCatch は検査対象フィールドを静的解析で絞り込む事前の静的解析や，報告された競合から無関係なものを刈り取る Static Pruning を行う．

2.1.3 オフライン解析と問題点

DCatch は実行トレースを用いたオフライン解析で *Happens-Before* 関係の計算と競合検査を行う．それゆえ長時間実行するプログラムではトレースサイズの肥大化により解析がスケールしない．さらに DCatch の静的解析によるフィールドの選別や Static Pruning は正確ではなく，1 度も検出できない DCbug のケースも考えられる．

2.2 DCbug 検出のオンライン化と問題点

Happens-Before 関係を用いた LCbug オンライン競合検出器は深く研究されている．DCbug の根本原因は LCbug と同じであり，オンライン分散並行バグ検出にこれらの技術を活用することが可能である．

Algorithm 1 Lock Acquire ー Thread t acquire lock m

$VC_t \leftarrow VC_t \sqcup VC_m$

Algorithm 2 Lock Release ー Thread t release lock m

$VC_m \leftarrow VC_t$
$inc_t(VC_t)$

Algorithm 3 Thread Fork ー Thread t fork thread u

$VC_u \leftarrow VC_t \sqcup VC_u$
$inc_t(VC_t)$

Algorithm 4 Thread Join ー Thread t join thread u

$VC_t \leftarrow VC_t \sqcup VC_u$
$inc_u(VC_u)$

Algorithm 5 Sampling copy VC ー $VC_m \leftarrow VC_t$

```
if not sampling then
  shareFlag_t ← true
  VC_m ←_shallow VC_t
else
  VC_m ←_deep VC_t
end if
vepoch_m ← VV_t[t]
```

Algorithm 6 Sampling increment ー inc_t

```
if sampling then
  if shareFlag_t == true then
    VC_t ←_deep VC_t
    shareFlag_t ← false
  end if
  VC_t[t] ← VC_t[t] + 1
  VV_t[t] ← VV_t[t] + 1
end if
```

2.2.1 FastTrack

FastTrack[5] は *VectorClock*[4] を用いた動的競合検出器である．FastTrack における *Happens-Before* 関係の計算はアルゴリズム 1〜4 の通りである．各スレッド，フィールド，排他制御/同期処理に使用されるオブジェクトは *VectorClock* メタデータを持つ．

また，FastTrack は *VectorClock* の簡約表現として epoch を導入し，フィールド用メタデータの時間/空間オーバーヘッドを削減している．

2.2.2 Pacer

Pacer[6] は，FastTrack に基づく競合検査の時空間オーバーヘッドをサンプリングにより削減する．検査するアクセスをサンプリングし，非サンプリング期間の競合検査や計算コストの高い処理を削減することで時間/空間オーバーヘッドを削減している．このサンプリング方式はサンプリング率競合検出率がおおよそ一致し，オーバーヘッドもサンプリング率に比例する．

Pacer の *VectorClock* のコピー，インクリメント，join 演算はアルゴリズム 5〜7 で再定義される．非サンプリング 期間に *VectorClock* が増加しないことで冗長な join が発生し，これを検出することで時間/空間オーバーヘッド

Algorithm 7 Sampling join — $VC_t \leftarrow VC_t \sqcup VC_m$

```
Let v@u = vepoch_m // VC_m = VC_u, VV_u[u] = v
if v@u ≠ null then
  if VC_m ⋢ VC_t then
    if shareFlag_t = true then
      VC_t ← VC_t
      shareFlag_t ← false
    end if
    ∀i.VC_t[i] ← max(VC_t[i], VC_m)
    VV_t[t] ← VV_t[t] + 1
  end if
  VV_t[u] ← v
end if
```

を削減している．

Pacer の競合検査は，サンプリング期間は FastTrack と同様である．非サンプリング期間は各フィールドアクセスに対し一度のみ競合検査を行い，以降はメタデータを破棄し，検査も行わない．この簡約によって時間/空間オーバーヘッドを削減している．

FastTrack のサンプリング法は他にも研究されているが，全ての競合を報告する可能性を持ち，計算/空間オーバーヘッドを大幅に削減した Pacer は分散システムへの拡張にあったって有用なサンプリング方式である．

2.3 Challange

本研究では前述したオフライン解析の問題点に対処すべく，精度/性能面を考慮した DCbug のオンライン解析に挑戦する．ただし，従来の LCbug を対象とするオンライン解析を単純に適用するだけでは正確かつ効率的に検査することはできない．本節ではこの問題点を論じる．

2.3.1 分散システムのオンライン解析

マルチスレッドプログラムを対象とする従来のオンライン競合検出手法を分散システムに単純適用した場合に生じる問題点は，以下の通りである．

検出漏れ DCbug はノード間通信など分散システムに固有の処理に起因するため，それらの処理を重点的に検査しない単純なサンプリング方式では DCbug の検出漏れが多発する．

誤検出 DCbug と LCbug は根本原因がノード内の競合であるが，ノード間通信を経由して他ノードの障害に影響を及ぼす競合のみが DCbug である．このため，他ノードへの影響を考慮しない従来のオンライン競合検出では誤検出が多発する．

オーバーヘッド 分散システムでは多数のノード上の多数のスレッドがノード間通信を行い，全体の処理を進める．従来のマルチスレッドオンライン解析手法の分散システムへの単純適用では，システム全体のスレッドを管理する *VectorClock* の要素数が膨大になり，競合解析の空間オーバーヘッドが大きい．またノード間通信に伴う多数の *VectorClock* の join 操作が競合解析の時間オーバーヘッドを増大させる．

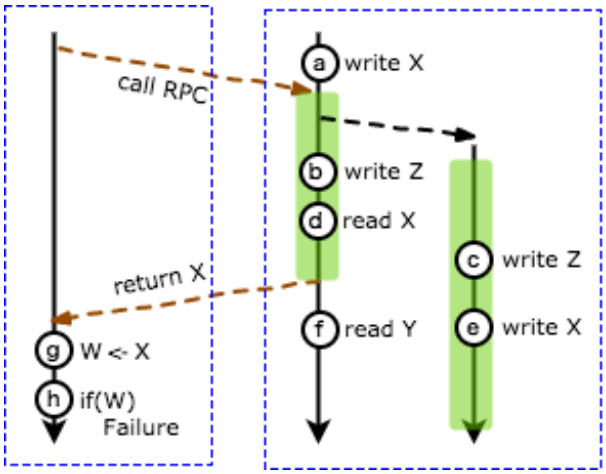

図 1 ノード間通信関連スレッドの伝搬

ポケットレーサは，バグの発生から報告までの遅延がなく，長時間の実行にも耐えうるオンライン分散競合検出器である．時間/空間オーバーヘッドが少なく，かつ DCbug を効率的に検出するオンラインサンプリング手法に挑戦する．

3. ポケットレーサ

ポケットレーサの主なアイデアは次の 3 点である．

- データサンプリング
- ポケットアノテーション
- 分散メタデータ簡約

この章では，3 つを順に紹介する．

3.1 データサンプリング

本節では DCbug に関連するフィールドアクセスを重点的に検査し，かつオーバーヘッドが少ない手法を提案する．

3.1.1 RPC-related Field

DCbug は，ノード間通信によって発生するアクセスの競合によって起こる．そこで RPC 関数内のフィールドアクセスや，ソケット通信によって起動されるスレッド内のアクセスを重点的に調べる必要がある．**図 1** の変数 X，Z はいずれも RPC 関数の呼び出しによってアクセスが発生する．このとき変数 X，Z を RPC-related Field と呼び，区別する．

RPC-related Field の特定は動的に行われ，DCatch の StaticPruning より正確である．DCatch は RPC 関数とその callee を追跡するが，簡単のため 2 段階以上離れた callee 内のアクセスを追跡しない．

RPC-related Field の特定はとても単純で，フィールドアクセスがあった場合にそのアクセスがノード間通信処理スレッド内で起きているか調べるだけで良い．ノード間通信処理スレッドとは，ノード間通信を受信し，関連した処

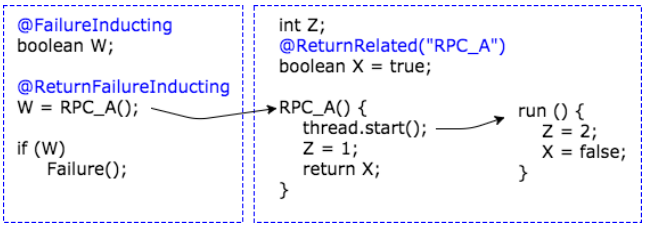

図 2 図 1 へのアノテーション例

理を行なっている間のスレッドを指す．図 1 の緑部分がノード間通信処理スレッドであり，イベント b，c，d，e がアクセスする変数が RPC-related Field である．

3.1.2 Sampling

PocketRacer が重点的に競合検査を行うフィールドは，以下の条件を満たす．

(1) RPC-related Field
(2) DCbug を引き起こす障害命令に到達するフィールド
(3) 複数スレッドから共有アクセスされるフィールド

(2) のフィールドの特定方法は複数考えられる．その中で本論文ではアノテーションを用いた特定方法を提案する (3.2 節)．(3) は単にアクセスされたスレッド ID を記録することで容易に実現可能である．図 1 ではイベント b，e のアクセスがこれに当てはまる．

ポケットレーサは非サンプリング期間であっても，上記 3 つの条件全てを満たすフィールドへのアクセスをサンプリング率の 10 倍の確率で検査する．競合検査を行う場合，直前で自スレッドの時刻を 1 進める必要がある．

3.2 ポケットアノテーション

DCbug を引き起こす障害命令に到達するフィールド (Failure-Inducting Field) を特定する手法としてポケットアノテーションを導入する．Failure-Inducting Field は大きく次の三種類に分けられる．

(1) 障害[*1]につながる条件分岐で使用されるフィールド
(2) Failure-Inducting Field の値に依存するフィールド
(3) RPC 関数の返り値が Failure-Inducting Field に代入される場合，RPC 関数の返り値に依存するフィールド

ポケットレーサではユーザが専用のアノテーションを使用して Failure-Inducting Field を指定する．アノテーションの一覧を**表 1** に示す．図 1 のコードに対してアノテーションを付加した例を**図 2** に示す．

ポケットアノテーションを用いたデータサンプリングは図 1 の例で次のように実行される．イベント a による X へのアクセスが起きたとき，X は RPC 関数内で実行されていないため，競合検査を行わない．変数 Z へのアクセスは，ポケットアノテーションによるアノテーションが付加されていないため競合検査を行わない．イベント d，e は RPC 関数内で X へのアクセスが発生している．X は返り値に影響するフィールドであり，現在の関数呼び出しの返り値が障害に到達しうることが ReturnFailureInducting アノテーションによって判明している．このためイベント d，e のアクセスは高確率で検査される．

このように，ポケットアノテーションを用いて DCbug の疑いが高いフィールドを重点的に検査することができる．

3.3 分散メタデータ簡約

時間/空間オーバーヘッド削減のため，本研究では分散システムに特化したメタデータ簡約を提案する．read/write アクセスに関するメタデータの簡約と，ノードバージョン配列を用いたノード間通信による冗長な join の検出によって Pacer の単純な適用に比べて時間/空間オーバーヘッドを削減する．

3.3.1 Variable メタデータ

FastTrack や Pacer の単純な分散拡張では，read アクセスの時刻を記録する Variable メタデータとしての *VectorClock* の要素数は全ノードの総スレッド数となる．しかし実際には，DCbug の根本原因は LCbug と同じであるため，ノード内のアクセス履歴の比較のみで競合を検出することができる．したがって Variable メタデータの *VectorClock* サイズはノード内のスレッド数しか必要ない．

プログラムの処理の大部分が read/write アクセスであり，Variable メタデータの削減によって *VectorClock* の比較演算コストが大幅に低下する．空間オーバーヘッドについても，メタデータの大部分を占める Variable メタデータの要素数削減によって大幅に低下する．

3.3.2 ノードバージョン配列

VectorClock 同士の join はスレッドの fork やノード間通信によって引き起こされるが，*VectorClock* サイズが肥大化する分散システムにおいてこのオーバーヘッドは無視できない．Pacer にはスレッドが同期操作をする際の *VectorClock* の copy，join のオーバーヘッドを減らすアイデア *VersionVector* があったが，分散システムではノード間通信による冗長な join が多く発生し，これらを検出することができない．ここではスレッド *VectorClock* 同士の演算によるオーバーヘッドを抑えるアイデアとしてノードバージョンとノードバージョン配列を導入し，ノードバージョン配列要素数を総ノード数に抑えつつ，ノード間通信に起因する冗長な join を削減する方法を提案する．

ノードバージョンは各スレッドの *VectorClock* に割り当てられるノード内で固有のバージョンである．*VectorClock* が更新されるたびにノードバージョンも増加し，値の異なる 2 つの *VectorClock* は必ず異なるバージョンを持つ．ノードバージョン配列は，ノード間通信によって他ノードから受信し join を行なった *VectorClock* の ノードバー

[*1] ここでいう障害とは，System.exit() 等によるシステムの異常停止/終了，プログラマが記述したエラー処理，catch できない例外のスロー，無限ループによるハングを指す [9].

表 1　アノテーション一覧

アノテーション	役割	
FailureInducting	フィールド宣言時	フィールドが障害命令に到達することを示す
ReturnFailureInducting	関数呼び出しの直前	関数呼び出しの返り値が障害命令に到達することを示す
ReturnRelated(String[] RpcName)	フィールド宣言時	指定した名前の RPC 関数の返り値に到達することを示す

Algorithm 8 Increment $VV_{t@n}[t] \leftarrow VV_{t@n}[t] + 1$

$VV_{t@n}[t] \leftarrow VV_{t@n}[t] + 1$
$globalNV_n \leftarrow globalNV_n + 1$
$localNV_{t@n} \leftarrow globalNV_n$

ジョンを 記録する配列である．他ノードから通信を受けると，受信した *VectorClock* のノードバージョンとノードバージョン配列に記録されたノードバージョンを比較し，join の必要があるか判定を行う．ノードバージョン配列を用いて，**図 3** の b'，d'，e' において冗長な join を検出することができる．

ここで，ノード n 上のスレッド t の *VectorClock* を $VC_{t@n}$ と表す．ノードバージョンの計算には，各スレッドのノードバージョンの管理に使用する $localNV_{t@n}$ とノード内で共有する変数 $globalNV_n$ を使用する．$globalNV_n$ はノードバージョンの値の衝突を避けるために使用される．ノードバージョンの更新方法の詳細をアルゴリズム 8 に示す．

また，ノード n 上のスレッド t のノードバージョン配列を $NVV_{t@n}$ と表す．$NVV_{t@n}[u]$ はノード n がノード u から受信したノードバージョンとスレッド ID を記録する bit 配列のペアであり，$< version, tbit[] >$ と表す．ノードバージョン配列に基づく冗長な join の検知と回避の具体的な仕組みはアルゴリズム 9 の通りであり，以下の通り単純である．

まず，ノード u から n への初回のノード間通信時に送信スレッド $s@u$ と受信スレッド $t@n$ の *VectorClock* 間で join を実行すると同時に，受信スレッド t@n のメタデータ $NVV_{t@n}[u].version$ に $localNV_{s@u}$ を記録し，メタデータ $NVV_{t@n}[u].tbit[s]$ に 1 を記録する．これにより，$VC_{s@u}$ の最新バージョンと join した事実が受信スレッド $NVV_{t@n}$ に記録される．次に，同じようにノード u から n へノード間通信が起きた時に，ポケットレーサは $localNV_{s@u}$ を $NVV_{t@n}$ に記録された $s@u$ のノードバージョンと比較する．この比較により，ポケットレーサは過去に送受信スレッド $s@u$ と $t@n$ 間で *VectorClock* が join 済みであることを検知し冗長な join を回避する．

この要領で，ポケットレーサは以降同じバージョンでのノード間通信に伴う join は全て回避する．図 3 はノードバージョン配列の更新と冗長な join の検出例を表し，イベント b'，d'，e' において join を削減している．

このようにして，PACER では検出できなかったノード間/スレッド間通信の冗長な join を， ポケットレーサでは実現している．

Algorithm 9 Join — $VC_{t@n} \leftarrow VC_{t@n} \sqcup VC_{u@m}$

Let $v@bits[] = NVV_{t@n}[n_1]$
if not $v \geq localNV_{u@m}$ **or** $bits[m] =$ **false then**
  if $VC_{u@m} \not\sqsubseteq VC_{t@n}$ **then**
    if $\text{shareFlag}_t =$ **true then**
      $VC_{t@n} \leftarrow_{deep} VC_{t@n}$
      $\text{shareFlag}_{t@n} \leftarrow$ **false**
    end if
    $\forall i. VC_{t@n}[i] \leftarrow max(VC_{t@n}[i], VC_{u@m})$
    $VV_{t@n}[t] \leftarrow VV_{t@n}[t] + 1$
  end if
  if $v \leq localNV_{u@m}$ **then**
    Clear bits[]
    $v \leftarrow localNV_{u@m}$
  end if
  $bits[m] \leftarrow$ **true**
  if $n = u$ **then**
    $VV_{t@n}[u] \leftarrow VV_{u@m}[u]$
  end if
end if

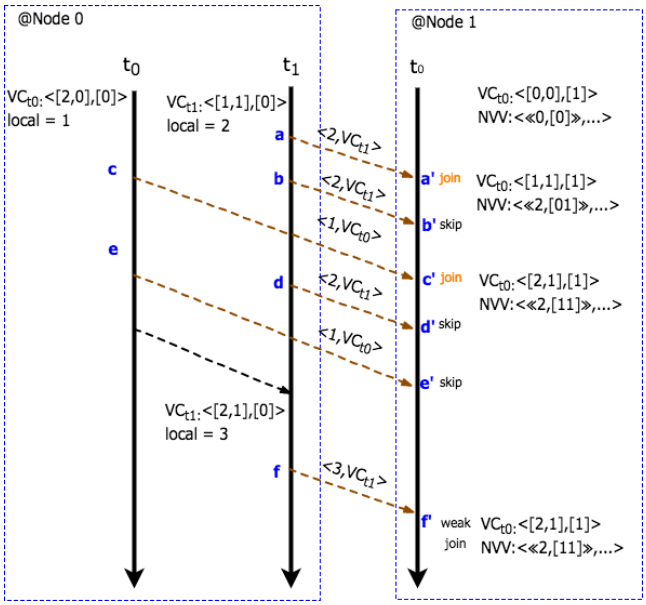

図 3　冗長な join の検出

4. 実装

4.1 メタデータ

分散システムの *VectorClock* メタデータは，プロセス ID と配列の HashMap で実装している．これを *NodeClock* と呼ぶ．配列はプロセス ID に対応するノード内のスレッドの *VectorClock* となっている．各変数，スレッド，Lock 用オブジェクトはそれぞれ *VectorClock*，*NodeClock* メタデータやデータサンプリング用フラグ，ノードバージョン配列

メタデータを持つ．加えて RPC 通信や Socket 通信によって送信されるオブジェクトには，メタデータを送信するためのフィールドを追加する．

4.2 計装

Javassist を用いてバイトコードに対して計装を行なった．フィールドアクセス，スレッドの fork/join，ロックの獲得/解放操作，ノード間通信の送信/受信に対して，該当メタデータへの処理を追加した．また，RPC 関数の名称は判明しているものとし，RPC 関数の呼び出しや Socket 通信の直前には送信オブジェクト内に *NodeClock* 等の情報を格納する処理も追加した．

4.3 サンプリング

サンプリングフラグの切り替えはノード内でフィールドアクセスが 10 回発生するたびに，サンプリング率 r の確率でサンプリング期間に突入する．

5. 実験

大規模分散システムの特徴を捉えたベンチマークプログラム群（以降シミュレータと呼ぶ）を作成し，これらにポケットレーサを適用することで DCbug の検査精度・効率を計測する実験を行った．この実験を通して，ポケットレーサは Pacer によるサンプリングと比較して同等以下の時間/空間オーバーヘッドで DCbug の誤検出を増やすことなく 4 倍以上の検出率を達成することを確認した．

比較対象として Pacer に DCatch の *Happens-Before* モデルを適用したツールを実装した．ポケットレーサ，シミュレータ，比較対象ツールの実装言語は全て Java で，MacOS 上に V Mを介して CentOS を 10 台設置し，各 OS 内が通信を行う．実行環境は MacBook Pro（15-inch，2016），macOS Sierra 10.12.6，プロセッサ 2.9GHz Intel Core i7，メモリ 16GB，ストレージ 2TB，VMWare Fusion Pro 10.0.1，CentOS Linux release 7.4.1708 である．

5.1 シミュレータ

シミュレータは大規模分散システムに特徴的なイベントをモデル化している．イベントの発生はランダム化し，その割合は DCatch に従った．シミュレータに含まれるイベントとその発生割合は**表 2** に示す．ロック獲得対象オブジェクト，RPC/Socket 通信の相手もランダムに決定する．現実の大規模分散システムの遅延 [10] を考慮し，ノード間通信に 1ms の遅延，更に Socket によるデータ送信に追加で 250ms の遅延を設定した．

5.1.1 DCbug トリガ

DCbug 検出精度の計測のために，シミュレータに TaxDC[2] を元にした 10 種類の DCbug トリガ，障害に到達しない Fake DCbug トリガ，ノード間通信に依存しない LCbug トリガを仕込んだ．

表 2 シミュレータのイベントと発生確率

イベント	Access	Thread	Event	Lock	RPC Socket
発生率	88.5%	1.8%	0.5%	7.1%	2.0%

5.1.2 パラメータ

シミュレータの動作を決定するパラメータとそのデフォルト値は，ノード数 $n = 100$，初期スレッド数 $t = 10$，1 スレッドあたりの実行イベント数 $e = 200$，サンプリング率 $r = 3\%$ である．スレッド数は実行中に増加し，1 ノードあたりの総スレッド数は初期スレッドの約 10 倍となる．このパラメータに則り，メタデータの各 *VectorClock* の初期サイズをノード数 100× スレッド数 10 = 1000 とした．

5.2 DCbug 検出精度

ポケットレーサの DCbug 検査精度として誤検出率，DCbug 検出数増加率，検出漏れ率を評価した．

5.2.1 誤検出率と DCbug 検出数増加率

図 4 は，ポケットレーサ，Pacer を 10 回実行した平均のノード総競合報告数を表す．LCbug，Fake DCbug の報告数は Pacer とポケットレーサで差がない．したがって，ポケットレーサのデータサンプリングによって誤検出が起こっていないことがわかる．また DCbug の検出数は Pacer と比較して 4 倍以上に増加しており，ポケットレーサのデータサンプリングによってサンプリング率を遥かに上回る DCbug を正しく検出できることを確認した．

5.2.2 検出漏れ

図 5，**図 6** は，シミュレータを 10 回実行し，各 DCbug トリガを 100 ノード合計でそれぞれ 3 回及び 1 回以上検出された回数を表す．全ての DCbug トリガーにおいてポケットレーサは Pacer の検出率を上回っている．ポケットレーサは極端に検出率の低い DCbug トリガーが存在せず，全ての DCbug トリガーを 1 度の実行で 3 回以上検出している．一方で Pacer は極端に検出率の低い競合が 4 つ存在し，ポケットレーサはそのような稀な競合も高い確率で検出している．

以上より，ポケットレーサではポケットアノテーションとデータサンプリング による DCbug の重点検査により DCbug の検出数のみが増加し各 DCbug をより高い確率で検出できることを確認した．

5.3 分散システムの規模とパフォーマンス

スレッド数を変化させ，ポケットレーサの性能を調査した．ポケットレーサはデータサンプリングによって *VectorClock* の値が増加し，Pacer に比べてコストの高い操作が増えるが，Variable メタデータの削減やノードバージョン配列による join 回数削減によって，Pacer をわずかに

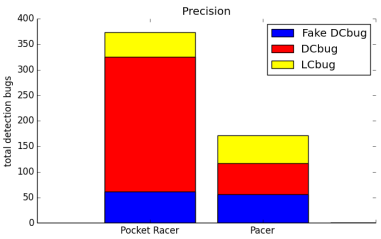

図 4　ツールと総競合報告数

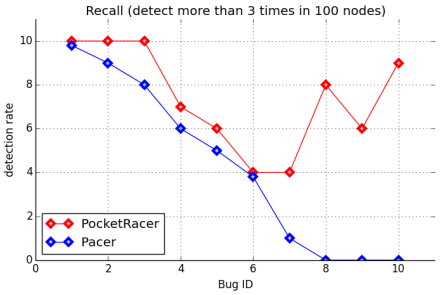

図 5　100 ノード全体で 3 回以上検出された割合

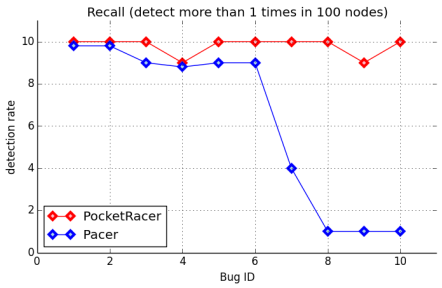

図 6　100 ノード全体で 1 回以上検出された割合

下回る時間オーバーヘッドを実現した．

5.3.1　時間オーバーヘッド

図 7 は横軸が総スレッド数，縦軸が実行時間オーバーヘッドの増加率を表す．ノード数を変化させることで総スレッド数を増加させている．各ノード数で 5 回計測を行い，全ノードの平均値を使用している．

総スレッド数が大きい場合はポケットレーサの方がわずかにオーバーヘッドが小さい．これはノードバージョン配列による冗長な join の削減による効果だと考えられる．**表 3** はポケットレーサと PACER の 1 ノードあたりの join 回数を計測したものである．スレッド数に応じて join 回数も増加するが，ポケットレーサはノードバージョン配列によって join 回数が PACER の半分程度に削減された．総スレッド数が増えるほど join のコストは高いため，スレッド数が増えるほどポケットレーサは PACER に比べてコストが低い．本論文では 100 ノード以上での実験を行うことができなかったが，総ノード数が増加するほどポケットレーサが優位になると予測する．

5.4　空間オーバーヘッド

図 8 は横軸が総スレッド数，横軸が実行空間オーバーヘッドの増加率を表す．総スレッド数の増加に伴い *VectorClock* サイズが増加し，空間オーバーヘッドも増加する．ポケットレーサは PACER に比べてデータサンプリングとノードバージョン配列に用いるメタデータが追加されているが，ノードバージョン配列による join の削減が非サンプリング期間の *VectorClock* データ共有操作を増やし，結果として空間オーバーヘッドがやや低下している．時間オーバヘッドと同様，空間オーバヘッドについても，総ノード数の増加に応じてポケットレーサが有利になると予測する．

以上より，ポケットレーサは PACER に比べて 4 倍以上の DCbug を報告し，時間/空間オーバーヘッドは同等以下に低下した結果となった．

6.　関連研究

6.1　LCbug 検出器

並行システムに対する競合検査の研究は盛んに行われ，その多くが *LockSet*[8] や *VectorClock*[4] を利用している．*LockSet* はオーバーヘッドが少ないが，誤検出を多く報告する．*VectorClock* は誤検出が起きない一方，計算コストが高い．

VectorClock を用いた動的競合検出のオーバーヘッドをサンプリングによって削減するアイデアも多く研究されている．LiteRace[7] は低いサンプリング率で高い検出率を達成するが，全ての競合を同じ確率で検出することはできず，検出率の極端に少ない競合も存在する．同期操作は FASTTRACK と同様に $O(n)$ の計算オーバーヘッドが発生する．PACER は LiteRace の問題点を解決するサンプリング法である．

6.2　DCbug 検出器

DCbug への対処として，モデルチェック，検証，そして DCatch の動的検査の 3 つのアプローチが存在するが，分散システムにスケールする決定的な手法はまだない．

DCatch は初の動的 DCbug 検出器である．分散システムで *Happens-Before* 関係を導出するためのモデルを構築し，トレースから *Happens-Before* 関係を表す有効グラフを構築している．誤検出を減らすため，DCatch はその後静的検査による競合報告の刈り取りや再実行も行っている．正確に DCbug を報告することができるが，トレース解析を行うため長時間実行することで発生する DCbug を検出するオーバーヘッドが高い．

7.　結論

分散システム固有のバグである DCbug は，従来のマルチスレッド動的競合検出手法での検出が難しい．DCatch は DCbug 動的検出手法として初の試みだが，実行時間の長い分散システムに対してスケールしない．本研究では DCbug を重点的に検査するデータサンプリングと，分散システムに特化したオーバーヘッド削減手法 Distributed-Metadata Reduction を導入したオンライン DCbug 検出手法を設計，実装した．シミュレータを用いた実験では，PACER によるサンプリングと同等以下の計算コストで DCbug 検出率は 4 倍以上となった．今後の課題として，*VectorClock* サイズを削減する *ChainClock* の分散システムへの導入やデータサンプリングの効率化による時間/空間オーバーヘッドの

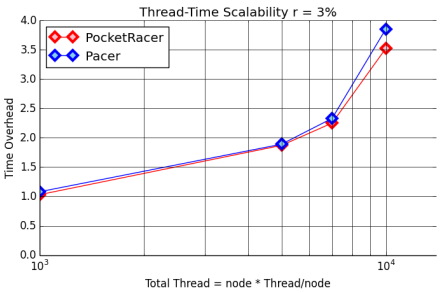

図 7 総スレッド数と時間オーバーヘッド

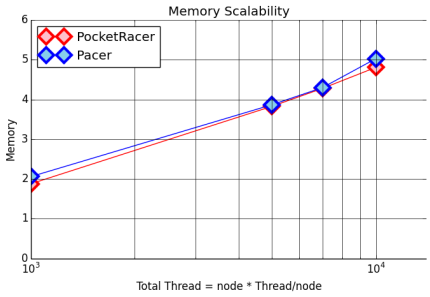

図 8 総スレッド数と空間オーバーヘッド

表 3 1 ノードあたりの初期スレッド数と join 回数

	ポケットレーサ	Pacer
$t = 5$	104	190
$t = 10$	233	445
$t = 20$	533	1020
$t = 50$	1392	2561

削減，ポケットアノテーション の半自動化が挙げられる．また，データセンタやスーパーコンピュータで実際に稼働する分散システムへの適用実験や，データ競合に起因するDCbug 以外の分散システムに固有のバグ検出への応用を目指したい．

ポケットレーサはオフライン解析に精度で劣らない，オンライン DCbug 検出器への初めての挑戦である．

謝辞 本研究は JSPS 科研費 16K00093「並行ソフトウェアの正確かつ高速な実行時検査」の助成を受けたものです．

参考文献

[1] Liu, H., Li, G., J. F. Lukman, et al. : *DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems.* Proc. ASPLOS'17, pp. 677-691 (2017).
[2] Leesatapornwongsa, T. ,J. F. Lukman, Lu, S., et al. : *TaxDC: A Taxonomy of Non- Deterministic Concurrency Bugs in Datacenter Distributed Systems.* Proc. ASPLOS'16, pp. 517-530 (2016).
[3] Lamport, L : *Time,clocks,and the ordering of events in a distributed system.* Proc. Communications of the ACM, 21(7):558-565 (1978).
[4] Mattern, F. : *Virtual Time and Global States of Distributed Systems.* Proc. Workshop on Parallel and Distributed Algorithms (1988).
[5] Flanagan, C. , and Stephen N. Freund. : *FastTrack: Efficient and Precise Dynamic Race Detection.* Proc. PLDI'09, pp.121-133 (2009).
[6] Michael D. Bond, Katherine E. Coons and Kathryn S. McKinley : *PACER: proportional detection of data races.* Proc. PLDI'10, pp. 255-268 (2010).
[7] Marino, D., Musuvathi, M. and Narayanasamy,S. : *LiteRace: effective sampling for lightweight data-race detection.* Proc. PLDI'09, pp.134-143 (2009).
[8] Savage, S., Burrows, M., Nelson, G. et al. : *Eraser: a dynamic data race detector for multi-threaded programs.* Proc. SOSP'97, pp 27-37 (1997).
[9] Zhang, W., Lim, J., Olichandran, R. et al. : *ConSeq: Detecting Concurrency Bugs through Sequential Errors.* Proc. ASPLOS'11, pp 251-264 (2011).
[10] Jiang, D., Ooi, B. C., Shi, L., et al.: *The performance of MapReduce: an in-depth study.* Proc. Proceedings of the VLDB Endowment Volume 3 Issue 1-2, September 2010, pp 472-483(2010).