クラウドシステムの非決定的性能バグ検査器

和田　智優†　　荒堀　喜貴†　　権藤　克彦†

† 東京工業大学　〒 152–8550 東京都東京都目黒区大岡山 2–12–1
E-mail: †wadatti@sde.cs.titech.ac.jp, ††{arahori,gondow}@cs.titech.ac.jp

あらまし　クラウドシステムには，特定のジョブの実行タイミングによってシステム全体のパフォーマンスが低下する PCbug（Performance Cascading Bug）と呼ばれる非決定的性能バグが存在する．これを自動的に検知する従来手法として PCatch が存在するが，検知漏れや誤検知を引き起こす．これに対して我々は，提案手法としてファジングを用いた PCbug の網羅的検査および記号実行を用いたパス・インターリーブの実行可能性解析を検討する．また，PCatch における誤検知や検知漏れが Apache Hadoop 等の現実の検体においてどのような原因でどの程度発生するかの調査を目指す．

キーワード　クラウドシステム，分散並行処理，パフォーマンスバグ

1　はじめに

クラウドシステムには，特定のジョブの実行タイミングによって処理の遅延が伝搬し，クラウドシステム全体のパフォーマンスが低下する PCbug（Performance Cascading Bug）と呼ばれる非決定的性能バグが存在する．これを自動的に検知する従来手法として PCatch [1] が存在するが，検知漏れや誤検知を引き起こす．これに対して我々は，提案手法としてファジングを用いた PCbug の網羅的検査および記号実行 [8] を用いたパス・インターリーブの実行可能性解析を検討する．また，PCatch における誤検知や検知漏れが Apache Hadoop [7] 等の現実の検体においてどのような原因でどの程度発生するかの調査を目指す．

2　PCbug: Performance Cascading Bug

PCbug とは，分散並行処理において，あるスレッドで発生した処理の遅延が他のスレッドの処理に伝搬し，最終的にシステム全体のパフォーマンスが低下する非決定的バグのことである．遅延の伝搬は Happens-Before（HB）関係によって表現できる．ここで，HB 関係とは，プログラム実行中に出現する 2 つのイベント間の実行順序制約である．例えば，イベント A とイベント B が存在するとき，A Happens-Before B とは A が必ず B より前に実行されるという関係である．

2.1　例　　題

図 1 は，実際の分散並行システムである Hadoop MapReduce で観測された PCbug の概略図である．あるクライアントが，JobTracker ノードに対して，ジョブの割当を行うと，それを受けて JobTracker ノードの Thread1-1 は TaskTracker ノードに対してタスクの割当を行う．それ受けて，TaskTracker ノードでは Thread2-1 がクリティカルセクションに入ろうとしてロック L1 を獲得しファイルダウンロードを実行する．もしこのダウンロードされるファイルサイズが巨大であった場合，Thread2-1 上で実行されるループは時間のかかるループとなる（図 1 の Non-Scalable Loop）．このループの処理中に，同じノード内の他のスレッド（Thread2-2）が，ロック L2 を獲得してからロック L1 を獲得しようとする場合がある．このとき，Thread2-2 は Thread2-1 による L1 の解放を待つ．同様に，Thread2-3 でもロック L3 を獲得してから Thread2-2 が既に獲得しているロック L2 を獲得しようとするが既に獲得されているため，待ちが発生する．このように Non-Scalable Loop を起点とする複数スレッドによるロック獲得待ちが，最終的に Heartbeat 処理を行っているスレッド Thread2-4 に伝搬することで，本来のタイミング（本来の Heartbeat）に比べ Heartbeat 処理に遅延が生じ（遅れた Heartbeat），JobTracker ノードでは TaskTracker ノードが停止していると認識し（Node2 dead），TaskTracker ノードで実行されているタスク全てを別のノードで実行しようとする．その結果，クラウドシステム全体のパフォーマンスが低下する．

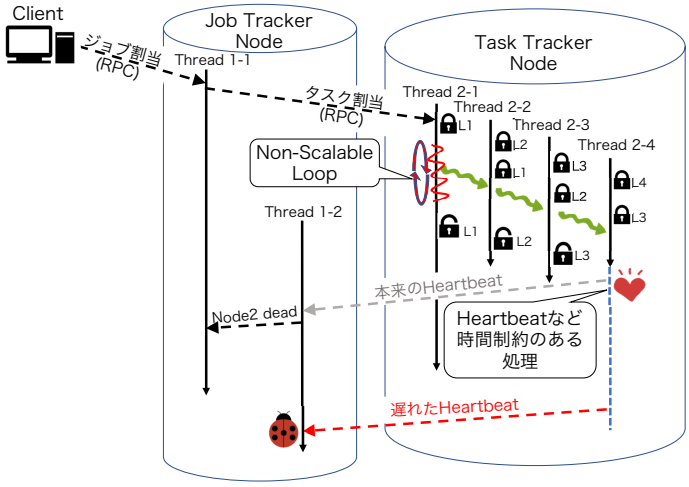

図 1　PCbug 概略図

2.2　PCbug の特徴

PCbug は次の 3 つの特徴を持っており，それらが PCbug の

発見と修正を難しくしている．

- ワークロードの規模に依存
- 非決定的に発生
- Non-Scalable Loop から Sink へ遅延が伝搬

2.2.1 ワークロードの規模に依存

上記の例のように PCbug の原因は，ファイルダウンロードによるループのような「外部入力によってループのアッパーバウンドが決定するループ処理」にある．このようなループを Non-Scalable Loop と呼ぶ．Non-Scalable Loop はワークロードに比例して処理時間が大きくなるため，大規模ワークロードにおいてのみ PCbug が発生する．

2.2.2 非決定的に発生

同一の大規模ワークロードをクラウドシステムに与えたとしても，必ずしも PCbug が発生するとは限らない．なぜなら，PCbug の要因となる複数スレッド間でのロック獲得待ちの連鎖は特定のスレッドインターリーブによって非決定的に発生するためである．

2.2.3 Non-Scalable Loop から Sink へ遅延が伝搬

PCbug は，Non-Scalable Loop で発生した処理の遅延が HB 関係を介して Sink に伝搬することで発生する．ここで，Sink とはクラウドシステムにおいて Heartbeat などの重要な処理に対応するイベントである．遅延が伝搬し Sink の実行が遅れた場合に，クラウドシステム全体のパフォーマンスが低下する事態に陥る．

3 従来の PCbug 検知法とその問題点

3.1 解析手法概要

PCbug を自動検知するための従来手法として PCatch がある．PCatch は Java で記述された分散並行プログラムを対象する．まず，PCatch は小規模ワークロードを与えて対象プログラムを実行して得たトレースから複数スレッド間におけるロック獲得待ちの連鎖を捕捉する HB グラフを作成する．ここで，対象プログラムに与える小規模ワークロードは，クラウドシステムの開発者が手動で準備する．次に，対象プログラムの静的ソースコード解析を行うことで，Non-Scalable Loop を特定する．最後に HB グラフを解析することで，Non-Scalable Loop から Sink に至る HB 関係の連鎖を特定し PCbug を検知する．

3.1.1 計　　装

PCatch は Java バイトコード変換フレームワークである Javassist [10] を用いて検査対象プログラムに対し計装を行っている．計装とは，検査対象プログラムに対してロギングコードを挿入することを意味する．PCatch のロギングは PCbug 検知に関連するイベントの実行トレースを出力する．

3.1.2 ループ解析

ループ解析では，まず，対象コード全体を静的に解析し，Non-Scalable Loop の候補となるループを列挙する．次に列挙された各ループの終了条件とループインデックスに注目することで，Non-Scalable Loop か否かを判別する．代表的な Non-Scalable Loop は大きく 3 つのパターンに分類される．

- ループのインデックスがワークロード規模依存
- ループバウンドがワークロード規模依存
- ループボディで時間のかかる入出力操作を実行

PCatch では，Java の静的解析フレームワークである WALA [11] を用いてループ解析を行う．

3.1.3 HB 解 析

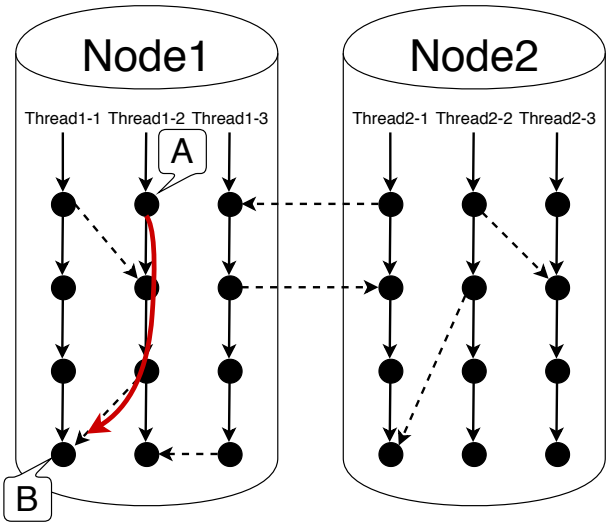

図 2　HB グラフ例

計装を行った対象プログラムの実行で得たトレースから PCatch は，複数スレッド間のロック待ち連鎖を捉える HB グラフを作成する．図 2 は，HB グラフの例である．図の実線矢印は各スレッド内で発生するイベント間の HB 関係を表している．破線矢印はスレッド間通信イベント，もしくはノード間通信イベントに対応する HB 関係を表している．また，グラフ上の黒点は 3.2 節に挙げるイベントを表している．例えば，図 2 の HB グラフでは，イベント A からイベント B へのパス（赤色の矢印）が存在する．このとき，イベント A とイベント B の間に A Happens Before B という関係が成立している．PCatch では，HB グラフ上に，Non-Scalable Loop から Sink へのパスがあるか否かをグラフ到達可能性判定アルゴリズム [12] を用いて解析する．Non-Scalable Loop から Sink へのパスがあった場合に，PCatch は PCbug の原因となった Non-Scalable Loop と，影響を受ける Sink の箇所，複数スレッド間のロック獲得待ちの連鎖を出力する．

3.2 HB 関係を構成するイベント

PCatch の計装によるロギングは，以下に示す HB 関連のイベントの実行トレースを出力する．

a） ノード間スレッド間通信イベント

- Remote Procedure Call（RPC）
- 非同期ソケット通信
- カスタム分散同期

b） ノード内スレッド間通信イベント

- スレッドの Fork/Join
- イベントキューに基づく非同期並行処理
- ロック/アンロック

c） ノード内単一スレッド内イベント

- メモリアクセス

PCatch では，これらのイベント間で成り立つ HB 関係を表現する HB グラフを作成する．

3.3 PCatch の問題点

PCatch は検知漏れと誤検知を引き起こす．

3.3.1 PCatch における検知漏れ

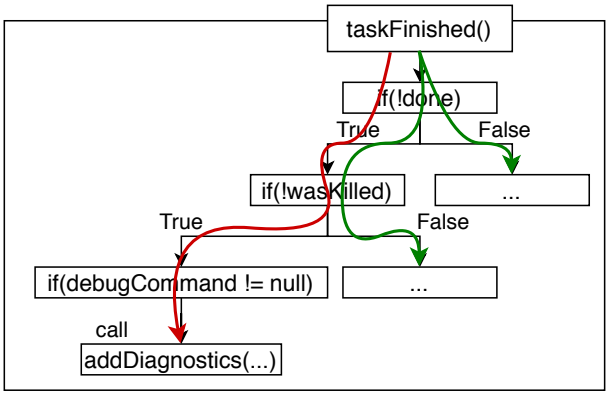

図 3　PCatch 検知漏れ例

```
1 public void addDiagnostics(String file, ...){
2     RandomAccessFile rafile = null;
3     rafile = new RandomAccessFile(file, "r");
4     ...
5     // Non-Scalable Loop
6     while((line = rafile.readLine()) != null){
7     ...
8   }
9 }
```

図 4　addDiagnostics メソッド

検知漏れの原因として，PCbug の潜む実行パスを十分に網羅できない点が挙げられる．PCatch では，与えられたテスト入力（小規模ワークロード）で対象プログラムが通った実行パス上の PCbug しか検知できない．例えば，図 3 の場合，taskFinished メソッドが赤色の実行パスで実行された場合にのみ addDiagnostics メソッドが呼び出され，このメソッド内で Non-Scalable Loop（図 4）が実行される．図 4 の 6 行目の while ループは，rafile のサイズが大きい場合にループの処理時間が増大するため，Non-Scalable Loop である．したがって，赤色の実行パスを通った場合 PCbug の検知が可能である．それ以外の緑色の実行パスでは，Non-Scalable Loop が実行されないため PCbug は検知されない．このように，入力によって実行パスが異なるため，PCatch はテスト入力による対象プログラムの実行で通過できなかったパス上の PCbug を検知できない（検知漏れ）．

```
1 File[] blockFiles = dir.listFiles();
2 int DataSize = blockFiles.length;
3
4 if(DataSize > THRESHOLD)
5   Thread.yield(); // 他のスレッドに実行を移譲
6
7 for (i=0; i < DataSize; i++){ // Non-Scalable Loop
8   ...
9 }
```

図 5　PCatch 誤検知例

3.3.2 PCatch における誤検知

また，誤検知の原因として，小規模ワークロードと大規模ワークロードでの実行パスの違いが挙げられる．図 5 のコードに対し，このコードの実行スレッドは小規模ワークロードでは他のスレッドに実行を移譲することなく Non-Scalable Loop を実行するが，大規模ワークロードでは，先に他のスレッドに実行を移譲することで Non-Scalable Loop の処理を後に回す．このようにワークロードの規模によって各スレッドの実行パスおよび実行順序が異なるため，検査時の小規模ワークロードで PCatch が PCbug を検知した場合でも，実際には大規模ワークロードで PCbug が起きないという誤検知が起きる．

4 提 案 手 法

提案手法の概要を図 6 に示す．提案手法では，PCatch と大きく 2 つの点で異なる．

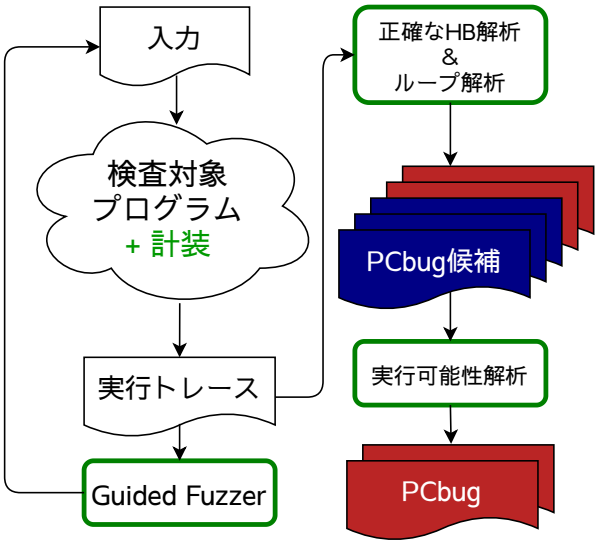

図 6　提案手法概要図

4.1 ファジングによる PCbug の網羅的検知

提案手法では，既存手法の問題点である検知漏れの低減のために，ファジングを用いて網羅的に PCbug 候補を検知する．ファジングでは，実行トレースを用いて Non-Scalable Loop を実行する入力を自動的に生成する．実行トレースには，前回までの実行で通過した対象コードのブロックカバレッジの情

報などが含まれている．従来手法では，図 3 のように，入力値によっては，Non-Scalable Loop を実行できず，PCbug を見逃す可能性がある．例えば，図 7 のように外部からの入力 A を与えてプログラムを実行した場合には，HB グラフ①が得られる．このとき，HB 解析を行っても Non-Scalable Loop から Sink（Heartbeat）へのパス（緑色の矢印）は存在しないため，PCbug は検知されない．しかし，提案手法のファジングでは，入力 A を与えて得た実行トレースから新たに入力 B を生成する．入力 B を与えて同じプログラムを実行した場合，HB グラフ②が得られ，同様に HB 解析を行うと Non-Scalable Loop から Sink（Heartbeat）へのパス（赤色の矢印）が見つかり PCbug を検知できる．さらに，入力 B から生成した入力 C を与えてプログラムを実行した場合には，HB グラフ③が得られる．この HB グラフ上では入力 A や入力 B では実行されなかった新たな Non-Scalable Loop（Non-Scalable Loop②）を実行しており，そこから Sink（Heartbeat）へのパス（赤色の矢印）が検知され，結果として新たな PCbug を検知できる．

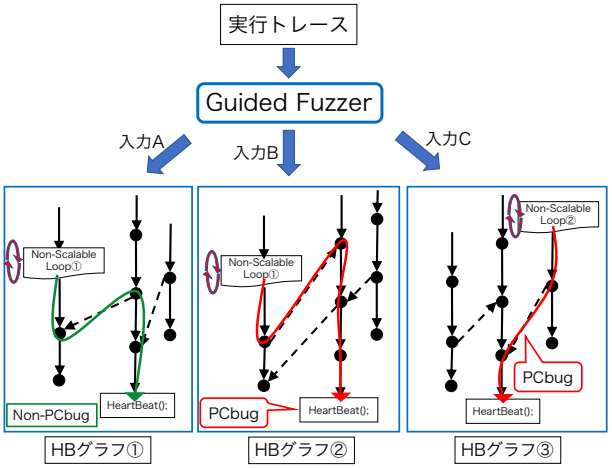

図 7　提案手法による PCbug 検知漏れの抑制

4.2　記号実行によるスレッド間パス・インターリーブの実行可能性解析

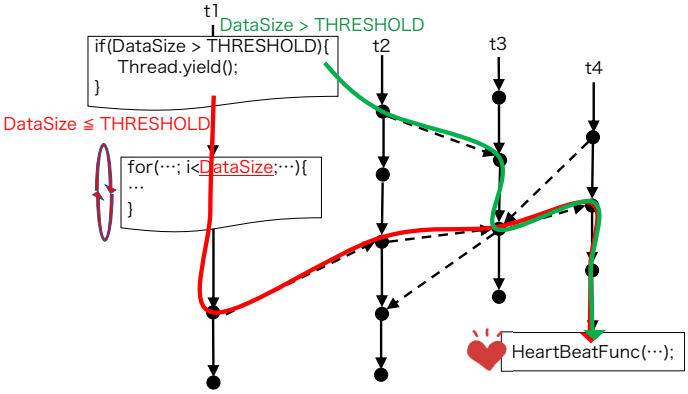

図 8　提案手法による PCbug 誤検知の抑制

提案手法では，既存手法の問題点である誤検知の低減のために，HB 解析で得た PCbug 候補の実行可能性を記号実行で解析する．例えば図 5 のコードを対象に，提案手法の HB 解析を行うと，PCbug 候補として図 8 に示す赤色のパス・インターリーブ上の PCbug 候補が検知される．ただし，この PCbug 候補は大規模ワークロード（`DataSize` > `THRESHOLD`）では実際には実行されないパス・インターリーブである（PCbug の誤検知）．この誤検知を防ぐために提案手法では，赤色のパス・インターリーブに沿ってこれが本当に実行可能であるかを問う制約式を収集する．具体的には，赤色のパス・インターリーブが実行可能となる制約式は，`DataSize` ≦ `THRESHOLD` である．しかし，大規模ワークロードで成立する制約は，`DataSize` > `THRESHOLD` であるため，大規模ワークロードでこのコードを実行した場合は，赤色のパス・インターリーブは，実行されず緑色のパス・インターリーブが実行される．その結果，Non-Scalable Loop から Sink（Heartbeat）に至るパス・インターリーブは大規模ワークロードでは実行されないことがわかり，PCbug 候補は実際には PCbug ではないという検査結果を得る．このようにして，提案手法では，HB 解析で得た PCbug 候補が真に実行可能であるかを記号実行で収集した制約を解くことで判定し，PCbug の誤検知を抑制する．

5　関 連 研 究

表 1　関連研究と提案手法の比較

	Target Processing	Target Bugs	Input Gen.	Precision	Recall	Scalability
PerfFuzz	Sequential	PerfBug	✓	✓	✓	X
COZ	Concurrent	Causal Profile	X	✓	X	✓
The Mystery Machine	Distributed Concurrent	Causal Relationships	✓	✓	-	✓
DCatch	Distributed Concurrent	DCbug	X	✓	X	X
ScaleCheck	Distributed Concurrent	Scalability Bug	X	✓	-	✓
PCatch	Distributed Concurrent	PCbug	X	X	X	X
Our Approach	Distributed Concurrent	PCbug	X	✓	✓	X

表 1 は関連研究と我々の提案手法の目標を定性的に比較した表である．本節では，この表に基づき，既存のパフォーマンスバグ解析や分散並行バグ解析と我々の提案手法の目的や効果の違いを説明する．

5.1　パフォーマンスバグ解析

PerfFuzz [4] では，逐次プログラムにおけるパフォーマンスバグの解析を行う．遺伝的アルゴリズムとミューテーションファジングを組み合わせることで，ホットスポットを実行する入力を生成する．

ScaleCheck [3] は，大規模分散システム特有のバグであるスケーラビリティバグを静的解析によって見つけ出す．ScaleCheck は大規模クラスタを 1 台のコンピュータ上で仮想的に動作させることでスケーラビリティ向上のボトルネックを特定する．

5.2 分散並行バグ解析

DCatch [2] は，HB 解析を用いて分散並行システムにおける分散並行バグの自動検知を行う．分散並行バグとは，単一ノード内で発生した共有資源アクセスの競合による共有資源の不正な状態が，他のノードに影響しシステム全体に障害が発生するバグのことである．

5.3 因果関係解析

COZ [5] はマルチスレッドプログラムにおいて，他のスレッドの処理に遅延を挿入することで，相対的に特定のスレッドを高速化させ，そのスレッドを実際に最適化した場合にプログラム全体が，どの程度性能向上するかを予測する．

The Mystery Machine [6] は，クラウドシステムのサーバからクライアントまで end-to-end での因果関係解析を行う．これによって，異なるコンポーネント間の因果関係を明らかになり，クラウドサービスのパフォーマンスチューニングを効率的に行えるようになる．

5.4 PCbug 検知

以上の関連研究はいずれも，我々が対象とする PCbug の検知を行うものではない．これに対し PCatch は PCbug の検知に特化した手法である．しかし，前述の通り，PCatch は PCbug の検知漏れと誤検知を引き起こす．これに対し，我々の提案手法は検知漏れと誤検知の少ない PCbug 検知を目標とする．

6 ま と め

本論文では，クラウドシステムにおける PCbug のより網羅的かつ正確な検知手法を検討した．従来手法の問題点として，PCbug の検知漏れと誤検知がある．検知漏れは，開発者が手動で用意したテスト入力で PCbug の潜む実行パスを網羅できないことに起因する．誤検知は，テスト時に開発者が与えた小規模ワークロードで PCbug の実行パスが観測されても，現実の大規模ワークロードではそのようなパスが実行不可能であることに起因する．我々は，これらの問題点を解決するために，ファジングを用いた PCbug の網羅的検知と，記号実行による実行パス・インターリーブの実行可能性解析を提案した．

文　献

[1] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Dongsheng Li, and Xicheng Lu, “PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems,” In Proceedings of EuroSys Conference, 2018.

[2] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian, “DCatch: Automatically detecting distributed concurrency bugs in cloud systems,” In Proceedings of ASPLOS, 2017.

[3] Caser A Stuardo, Tanakorn Leesatapornwongsa, Riza O. Suminto, Huan Ke, Jeffery F. Lukman, Wei-Chiu Chuang, Shan Lu, Haryadi S. Gunawi, “ScaleCheck: A Single-Machine Approach for Discovering Scalability bugs in Large Distributed Systems” In Proceedings of FAST Conference, 2019.

[4] Caroline Lemieux, Rohan Padhye, Koushik Sen, Dawn Song, “PerfFuzz: Automatically Generating Pathological Inputs,” In Proceedings of ISSTA Conference, 2018.

[5] Charlie Curtsinger, Emery D. Berger, “COZ: Finding Code that Counts with Causal Profiling” In Proceedings of SOSP, 2015.

[6] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch, “The mystery machine: End-to-end performance analysis of large-scale internet services” In Proceedings of OSDI, 2014.

[7] Apache Hadoop, `https://hadoop.apache.org/`

[8] Robert Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, Irene Finocchi, “A Survey of Symbolic Execution Techniques,” Computing Survey, 2018.

[9] Leslie Lamport, “Time, clocks, and the ordering of events in a distributed system,” Commun. ACM, 1978.

[10] Javassist by jboss-javassist, `https://www.javassist.org/`

[11] Main Page - WalaWiki, `http://wala.sourceforge.net/wiki/index.php/Main_Page`

[12] H. V. Jagadish, “A Compression Technique to Materialize Transitive Closure,” TODS, 1988.