

# Probabilistic Calling Context

Michael D. Bond, Kathryn S. McKinley

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

---

2017/10/10

# 概要

---

背景:オブジェクト指向言語は手続き型言語よりもメソッド/関数間の制御の遷移が多量&複雑

- オブジェクト指向言語を分析するのにCalling Contextは有用、しかし計算は高価

目的:Calling Contextを安価に計算できる手法(PCC)を提案

提案手法:PCC(Probabilistic Calling Context)

- CCを区別するためのインデックス計算
  - インデックス値の分散(衝突回避)
  - インデックス計算の軽量化

実験結果:

- インデックス(PCC値)の衝突はほとんどなし
- JVMへの実行オーバーヘッドは平均+3%
- 空間オーバーヘッドは1word/1context(既存研究と比較して安価)

# 研究背景

## 1-1: 開発者の現状

### アプリケーションの理解、デバッグ、テスト、最適化が困難

- アプリ規模が巨大化、複雑化 → 目的達成のために、あちこちからモジュールを組み立てがち
- 使っている言語がJava, C#など → 細かいモジュールの集合のため、コンテキスト表現がinter-procedural制御フローになりがち
  - Inter-procedural: ここでは複数の関数にまたがって制御フローが解析されるという意味(⇒intra-procedural)
- →アプリケーション全体の理解が困難
- →十分なテストができない
- →静的解析の意義が薄い

→何かいい方法はないものか...



# 研究背景

## 1-2:Calling Context(CC)とは

### プログラムメソッドのシーケンス

- =コールサイトの連なりのこと
  - →コールスタックに積まれている
- →本論文では
  - アクティブコールサイト
  - &プログラムロケーション
  - の組み合わせ
- コールサイト=
  - メソッド & 行番号
  - 次のメソッドの呼ばれる位置
- つまりCC=コールスタックの中身
- 動的に取ることで、特定のポイントでのコールスタックの情報を得られる

アクティブコールサイト

```
at com.mckoi.database.jdbcserver.JDBCDatabaseInterface.  
execQuery():213
```

```
at com.mckoi.database.jdbc.MConnection.  
executeQuery():348
```

```
at com.mckoi.database.jdbc.MStatement.  
executeQuery():110
```

```
at com.mckoi.database.jdbc.MStatement.  
executeQuery():127
```

```
at Test.main():48
```

上のCCは、

- mainメソッドの48行目でexecuteQuery()が呼ばれて
- executeQuery()127行目でexecuteQuery()が呼ばれて
- ...
- 今現在execQuery()の213行目

プログラムロケーション

# 研究背景

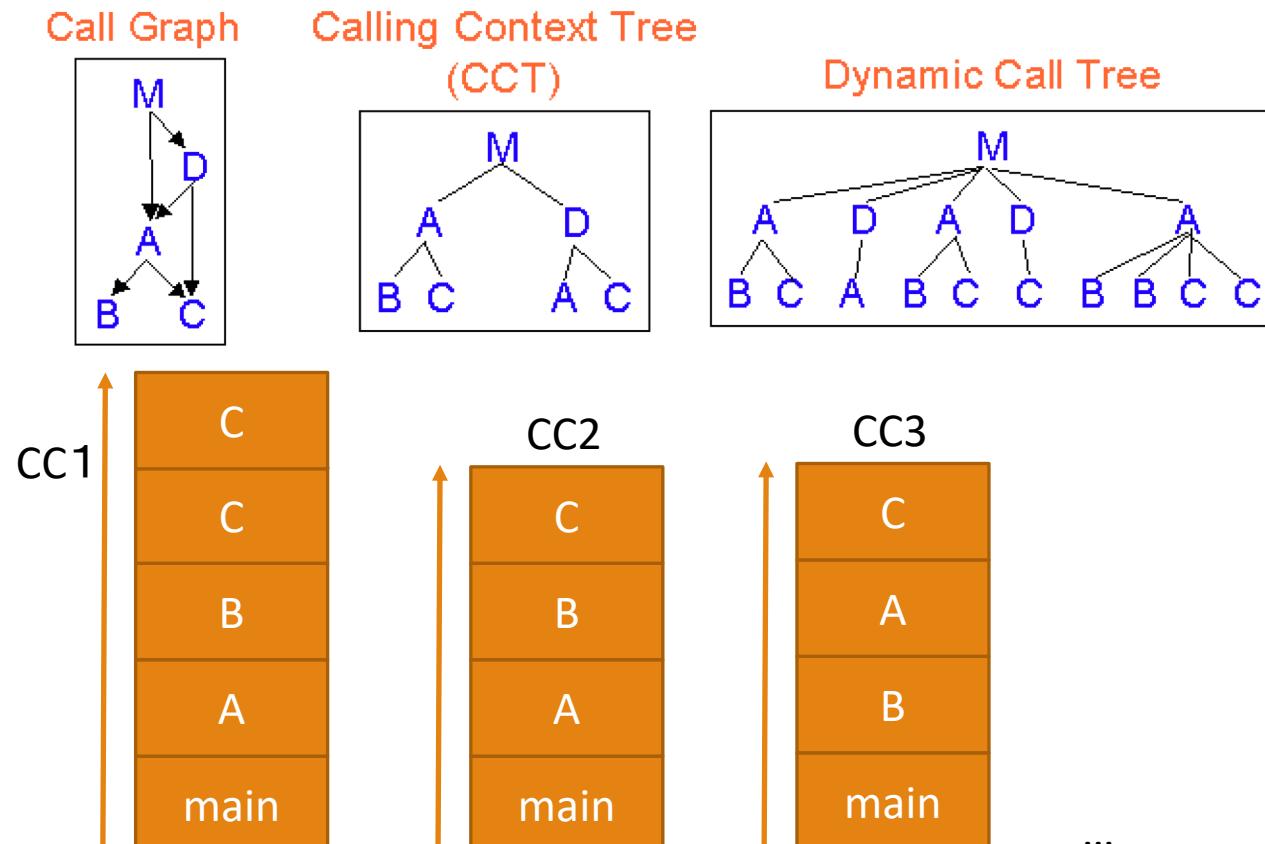
## 1-2:Calling Context(CC)とは

### CCの粒度

- コールグラフ < CCT < DCT
- 粗 → 細
- コールグラフ:
  - パスを考慮しない
- CCT(Calling Context Tree):
  - パスと重複を考慮
  - →関連研究
- DCT(Dynamic Call Tree):
  - 実行の重複を考慮しない

### Cプログラムでは安価に計算することが難しい

- Javaプログラムはさらに難しい
- GCC(大きめのCプログラム): 57,777個の個別CC
- Javaベンチマークでは1,000,000以上の個別CCが存在
  - 10個中5個が100,000以上



# 研究背景

## 1-3: 既存研究

### Stack-walking(SW)

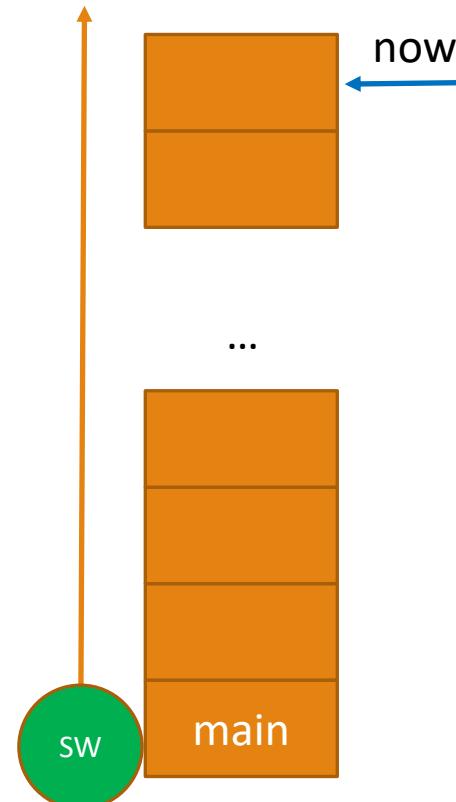
- 実際にコールスタックを調べてみる手法
- シンプルな手法だがコストが高い

### Calling Context Tree(CCT)

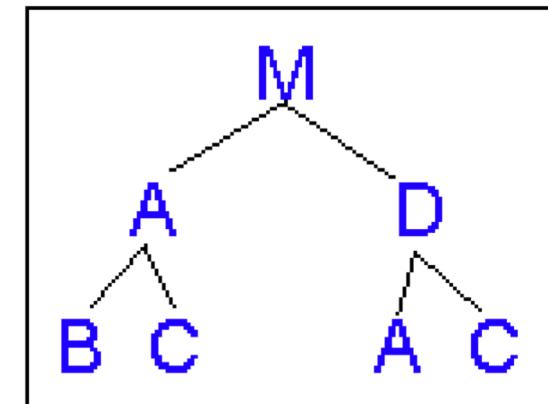
- 動的に木構造を構築する手法
  - 各ノードにCCの情報が入る
- コストが高い(+100%~+300%)

### Sampling

- 頻繁な実行箇所をサンプリング
  - コスト削減
- セキュリティ違反のテスト、
- デバッグ、検査には向き



### Calling Context Tree (CCT)



上のCCTから得られるCC

- (行番号を考慮しなければ)
- M, MA, MD,
- MAB, MAC, MDA, MDC

# 研究背景

## 1-4 : Probabilistic Calling Context(PCC)

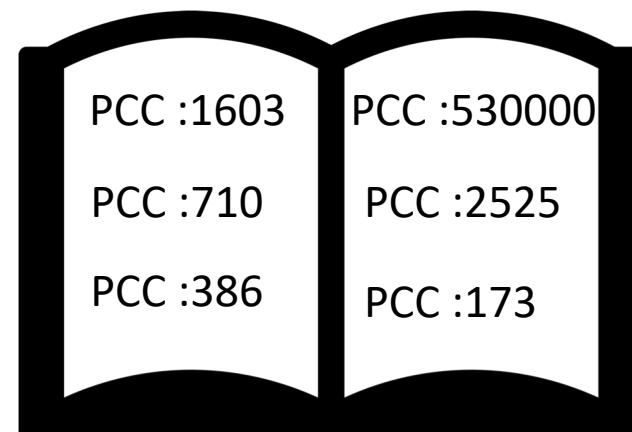
PCC値(PCC Value)を各CCについて計算

- PCC値が異なる = それぞれ別のCC
- CCの区別にのみ利用されるので、PCC値はCCの中身を表しているわけではない
  - →中身を知りたい場合には、Stack-walkingなどが別途必要
- →PCC値の計算結果が分散するようにランダム要素を用いる
  - Probabilistic: 確率的
  - PCC値が近い≠対応するCCの構造が近い

利用するにあたり、二つのフェーズが存在

- トレーニングフェーズ
  - プログラムの挙動(PCC値)を収集する
- プロダクションフェーズ
  - トレーニングでの挙動と、実際の挙動をPCC値で比較

つまり、  
個別のCCを表すインデックスのみを管理する

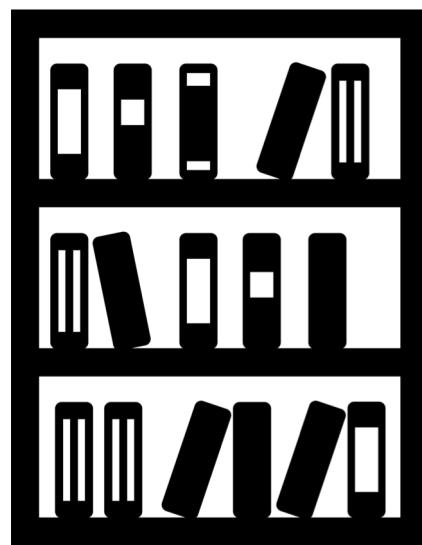


# モチベーション

## 2-1:

CCはオブジェクト指向プログラムの解析に重要になっているが**高価**

- 大事なことなので...
- →効率化して安価に求めたい
  - →ただしサンプリングのように**カバレッジを損なう**のは避けたい



既存手法



PCC



題名しか載ってないので  
中身は調べないと分からぬ

# モチベーション: 2-2:PCCの利用先(CCの利用先)

## 残存テスト(Residual test)

- テスト時には実行されなかったパスを識別すること
- 残存テストを実行するためには、すでに実行したパスを把握しなければならない
  - →既知のPCC値でなければ新しいパスの実行をしている

## 異常ベースバグ検知(Anomaly-based Bug Detection)

- エラーに対して、関係する異常な挙動を捉えて識別する
- CCは計算コストの高さから利用されてこなかった
  - →既知のPCC値でなければ異常な挙動だと判断できる

## 異常ベース侵入検知(Anomaly-based Intrusion Detection)

- これまで観察されなかった挙動=攻撃と判定する
- より高度な模倣も見破れる
  - 模倣:攻撃者が内部の挙動を真似てセキュリティの監視を潜り抜けようとしている
    - →模倣してもPCC値の違いから見抜ける

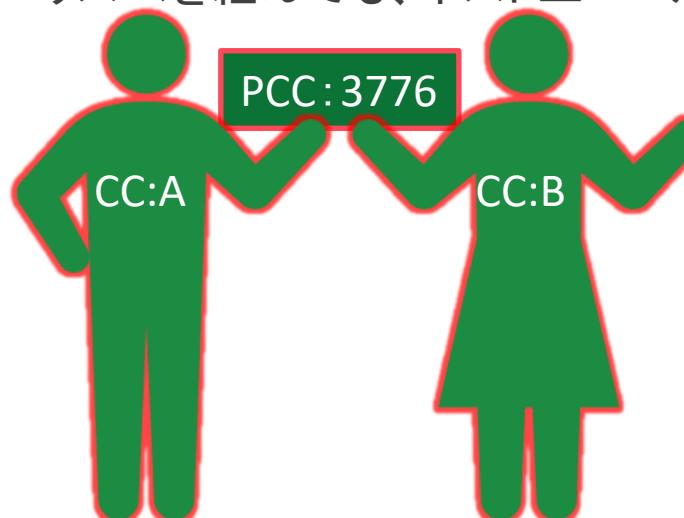
実験結果5-4を鑑みると、  
判断は困難な可能性

# 提案手法：

## 3-1:Probabilistic Approach

### 衝突(Confliction)

- 異常な挙動のCCと、普通の挙動のCCと同じPCC値として計算してしまう可能性
  - PCC値が同じ = 同じCCと判断される
  - 前述の例で言えば同じ題名の本が書かれるようなこと
- →がしかし、その割合は十分低くなる
  - 32bitの場合で、10,000,000件の異なるCCがあっても衝突率は0.1%以下
- 仮に攻撃者がメタPCCでアルゴリズムを組んでも、ホスト上コールサイトをランダム化すれば攻撃不可
  - 事実上不可能ということ
  - ただし未検証



# 提案手法

## 3-1 : Probabilistic Approach

### 衝突率の概算

- $E[conflicts] := n - m + m \left( \frac{m-1}{m} \right)^n$
- n: ビット数(32 or 64)
- m: 値の取り得る範囲(10,000,000など。上限 $2^n$ )

多くのプログラムは1,000万個未満の個別CC

- 衝突の確率は低い(32bitの場合)
- 仮により巨大なプログラムでも、64bitで十分
- 100億個の個別CCでも衝突率は限りなく0に近い

Random values	Expected conflicts	
	32-bit values	64-bit values
1,000	0 (0.0%)	0 (0.0%)
10,000	0 (0.0%)	0 (0.0%)
100,000	1 (0.0%)	0 (0.0%)
1,000,000	116 (0.0%)	0 (0.0%)
10,000,000	11,632 (0.1%)	0 (0.0%)
100,000,000	1,155,170 (1.2%)	0 (0.0%)
1,000,000,000	107,882,641 (10.8%)	0 (0.0%)
10,000,000,000	6,123,623,065 (61.2%)	3 (0.0%)

**Table 1.** Expected conflicts for various populations of random numbers using 32-bit and 64-bit values.

# 提案手法

## 3-2:PCC値の計算

PCC値の計算に求められること

- 衝突が起きないように、ランダムに分散する
- 決定的である。すなわち、同一のCCは常に同じ値が計算される
- 現在のPCC値から次のPCC値を計算することは効率的である

PCC値の計算式

- $f(V, cs) = 3 \times V + cs$ 
  - 基本式
  - ただし計算結果はmodulo  $2^{32}$
- $V_{callee} = f(V_{caller}, cs_{callee})$ 
  - 伝搬式( $V$ の初期値は $0[1]$ )
  - $V_{caller}$ :呼び出す前のPCC値
  - $V_{callee}$ :呼び出された後のPCC値
  - $cs_{callee}$ :コールサイトID[1]。コンパイル時に静的に定まる(分散要素)

```
method() {
    int temp = V;           // ADDED: load PCC value
    ...
    V = f(temp, cs_1);     // ADDED: compute new value
    cs_1: calleeA(...);   // call site 1
    ...
    V = f(temp, cs_2);     // ADDED: compute new value
    cs_2: calleeB(...);   // call site 2
    ...
}
```

[1]: Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses[PLDI'10]

Michael D. Bond et al.

# 提案手法

## 3-2: PCC値の計算 非可換性と効率的構成性

---

### 非可換性 (Non-commutativity)

- 一般的に等号で結ばれることがないこと
- CCでは  $A \rightarrow B \rightarrow C$  と  $C \rightarrow A \rightarrow B$  を別のコンテキストと分類

### 効率的構成性 (Efficient composability)

- 実装をいかに効率的にするか
  - →本手法ではPCCの演算をいかに単純化できるか
- PCC計算に必要な定数...

◦ →CS

### 非可換性

- 一般的に
  - $(3 \times cs_A) + cs_B \neq (3 \times cs_B) + cs_A$
  - なので  $A \rightarrow B$  と  $B \rightarrow A$  を比較すると
  - $f(f(V, cs_A), cs_B) = 9 \times V + (3 \times cs_A) + cs_B$
  - $\neq f(f(V, cs_B), cs_A) = 9 \times V + (3 \times cs_B) + cs_A$
- また、和積を組み合わせることで分散する

### 効率性

- $(3 \times cs_B) + cs_A$  はコンパイル定数として計算可能
  - →効率的
  - ただしコンパイルのコストは上がる
  - →実験結果5-3

# 提案手法

## 3-4 : PCC値の参照

---

プログラムロケーションのPCC値を得るには、現在の行番号を $V$ に適用するだけ

- メソッド内の特定ポイントで $f(V, cs)$ を計算

PCC値を参照するために目的ロケーションにPCC値の要求を計装しておく必要がある

- →コンパイル時にPCC値参照要求([PCCクエリ](#))を計装

前述のトレーニングフェーズとプロダクションフェーズでは

- トレーニング:PCC値を参照、格納
- プロダクション:PCC値を参照、比較

```
method() {  
    ...  
    cs: query(f(V, cs));           // ADDED: query PCC value  
    statement_of_interest; // application code  
    ...  
}
```

# 実装

## 4-1: 構築環境

### 構築環境

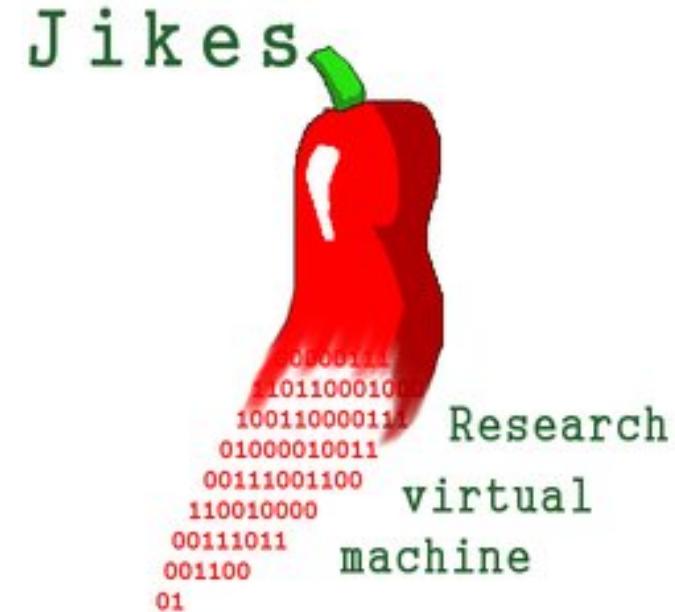
- Jikes RVM 2.4.6
    - 高性能Java-in-JVM
    - 二つのコンパイラを保有
      - 非最適化コンパイラ
      - 最適化コンパイラ
- } 両方修正 & PCCを計装

### V(PCC値)は大域変数

- マルチスレッドの場合は各スレッドごとにVを保有
  - メソッドの初めに値をローカルコピー(退避)
    - →次のCCをfで計算 & Vを更新
    - →Return時にローカルコピーをVに戻す
  - →例外処理などのときにVの値を正確に維持するため

### PCC値の格納、検索用にグローバルハッシュテーブルを使用

- $2^k$ 個の32ビットスロットを用いて実装
  - 下位kビットでインデックス化(ハッシング)



# 実験評価

---

1 : 実験環境、評価方法  
その他制限等

2 : PCCの精度評価  
典型的なクライアントに対して  
目的:PCCが

3 : PCCのパフォーマンス評価  
Stack-walkingとの比較

4 : Calling Context Profilingの有用性評価  
Call Site Profilingとの比較

# 実験評価

## 5-1: 実験環境

### 精度比較用にCCTを実装

- 既存の実装よりも空間、実行オーバーヘッド減少
  - 収集対象をノードに限定
  - 簡単化のためにコンパイラの最適化を無効化

### Jikes RVMの最適化コンパイラについて

- タイマーベースなので**非決定的**
- 測定のためにリプレイコンパイルを使用**
  - 各メソッドの最適化レベル
  - DCGプロファイル
  - エッジプロファイル
- アドバイスファイルを元に最適化を再現、実行

アドバイスファイルとして出力

### ベンチマーク

- 対象: DaCapo (ver. 2006-10), SPEC JVM'98**
  - 入力は大規模前提 (eclipseを除く)
  - xalanはリプレイコンパイルが正しく実行されないので除外
    - パフォーマンス関連のみ除外
  - eclipseはメモリが不足するので中規模の入力
- 固定ヒープサイズの3倍を確保
- 世代別マーカクスイープGCを使用
- 各ベンチマークは五回以上計測

### プラットフォーム

- Linux 2.6.12
- 3.6 GHz Pentium 4
  - L1, L2 キャッシュ (ラインサイズ 64byte)
  - L1 データキャッシュ (8-way set-associative: 16byte)
  - L1 命令トレースキャッシュ (12Kμops)
  - L2 オンチップキャッシュ (8-way set-associative: 2Mbyte)
- メインメモリ: 2GB

# 実験評価

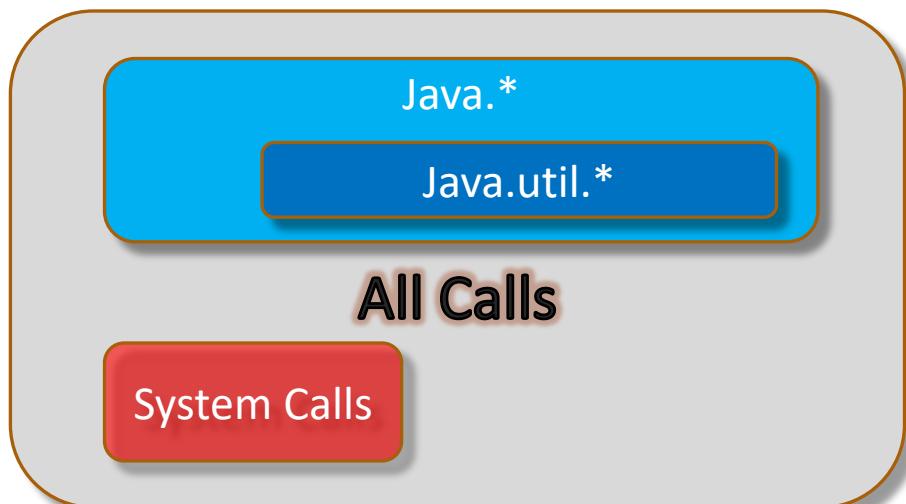
## 5-2:PCCの精度評価

目的:

- PCCで正しくCCを区別できるているか計測
- 衝突が起きていないか計測

典型的なクライアントについて

- 現実的な利用方法として典型的な3種類+1種類
- **PCCクエリ**: ここではCCを計算する動作



### ***System Calls***

- システムコールの前にPCCクエリを追加
- 想定利用方法:異常ベース侵入検知など

### ***Java Utility Calls***

- java.util.\*メソッドのコール時にPCCクエリを追加
- 想定利用方法:残存テスト

### ***Java API Calls***

- java.\*メソッドのコール時にPCCクエリを追加
  - Java.util.\*のスーパーセット
- 想定利用方法:残存テスト

### ***All Calls***

- 全てのコールサイトでPCCクエリを追加
- PCCが及ぼすオーバーヘッドの上限を測定するため

# 実験評価

## 5-2:PCCの精度評価

### SystemCalls

Dynamic:

PCCクエリの数

Distinct:

区別できるPCC値の数

Conf.:

衝突数(※)

※CCTで得られたCCの数との差と思われる

Program	System calls		
	Dynamic	Distinct	Conf.
antlr	211,490	1,567	0
bloat	12	10	0
chart	63	62	0
eclipse	14,110	197	0
fop	18	17	0
hsqldb	12	12	0
jython	5,929	4,289	0
luindex	2,615	14	0
lusearch	141	11	0
pmd	1,045	25	0
xalan	137,895	59	0
DaCapo geo	843	60	
pseudojbb	507,326	145	0
compress	7	5	0
jess	50	6	0
raytrace	7	5	0
db	7	5	0
javac	7	5	0
mpegaudio	7	5	0
mtrt	7	5	0
jack	7	5	0
SPEC geo	30	7	
Geomean	188	23	

7つのベンチマークで  
1000以上のシステムコールが  
呼ばれている

2つのベンチマークで  
1000以上の異なるコンテキストを実行

# 実験評価

## 5-2:PCCの精度評価

JavaUtilityCalls ~ Java API calls

Dynamic:

PCCクエリの数

Distinct:

区別できるPCC値の数

Conf.:

衝突数(※)

※CCTで得られたCCの数との差と思われる

Program	Java utility calls			Java API calls		
	Dynamic	Distinct	Conf.	Dynamic	Distinct	Conf.
antlr	698,810	8,010	0	24,422,013	128,627	3
bloat	1,030,955,346	143,587	3	1,159,281,573	600,947	40
chart	43,345,653	44,502	0	258,891,525	202,603	4
eclipse	3,958,510	54,175	0	132,507,343	226,020	5
fop	5,737,083	25,528	0	9,918,275	37,710	0
hsqldb	90,324	267	0	81,161,541	16,050	0
jython	76,150,625	131,992	2	543,845,772	628,048	48
luindex	5,437,548	1,024	0	39,733,214	102,556	0
lusearch	23,183,861	176	0	113,511,311	905	0
pmd	372,159,946	442,845	24	537,017,118	847,108	79
xalan	744,311,518	6,896	0	2,105,838,670	17,905	0
DaCapo geo	19,667,815	12,689		163,072,787	85,963	
pseudojbb	18,944,200	475	0	30,340,974	3,410	0
compress	1,018	682	0	8,138	1,081	0
jess	4,851,299	2,061	0	16,487,052	5,240	0
raytrace	1,078	684	0	5,331,338	3,383	0
db	65,911,710	767	0	90,130,132	1,439	0
javac	6,499,455	55,994	0	24,677,625	255,334	4
mpegaudio	874	682	0	7,575,084	1,668	0
mtrt	880	682	0	5,573,455	3,366	0
jack	14,987,342	14,718	0	21,771,285	29,461	0
SPEC geo	199,386	1,724		7,074,200	5,410	
Geomean	2,491,316	5,168		39,734,213	24,764	

Java utility calls, java API callsは動的コール数が多い

→コール数の割にオーバーヘッドは少ない(実験結果5-3)

衝突の数がほとんど0なのはGOOD

# 実験評価

## 5-2: PCCの精度評価 All Calls

Dynamic: PCCクエリの数

Distinct: 区別可能PCC値の数

Conf.: 衝突数

Ave:

  コールチェインの平均深度

Max:

  コールチェインの最大深度

Program	All contexts			Call depth	
	Dynamic	Distinct	Conf.	Avg	Max
antlr	490,363,211	1,006,578	118	21.5	164
bloat	6,276,446,059	1,980,205	453	30.6	167
chart	908,459,469	845,432	91	16.6	29
eclipse	1,266,810,504	4,815,901	2,652	15.0	102
fop	44,200,446	174,955	2	22.4	49
hsqldb	877,680,667	110,795	1	19.3	36
jython	5,326,949,158	3,859,545	1,738	58.3	223
luindex	740,053,104	374,201	12	19.4	34
lusearch	1,439,034,336	6,039	0	15.2	24
pmd	2,726,876,957	8,043,096	7,653	28.9	416
xalan	10,083,858,546	163,205	6	19.4	63
DaCapo geo	1,321,327,982	562,992		22.3	78
pseudojbb	186,015,473	19,709	0	7.1	25
compress	451,867,672	1,518	0	13.6	17
jess	198,606,454	18,021	0	43.1	83
raytrace	557,951,542	21,047	0	6.7	18
db	91,794,359	2,118	0	13.0	18
javac	135,968,813	2,202,223	544	29.5	122
mpegaudio	218,003,466	7,576	0	21.9	26
mtrt	564,072,400	21,040	0	6.7	18
jack	35,879,204	82,514	1	22.3	49
SPEC geo	200,039,740	20,695		14.8	32
Geomean	565,012,654	127,324		18.6	52

区別可能なCC: 100万越えが6つ、10万越えが5つ

# 実験評価

## 5-3:PCCのパフォーマンス:実行オーバーヘッド

Base: PCC計装なし

PCC: PCC計装単体

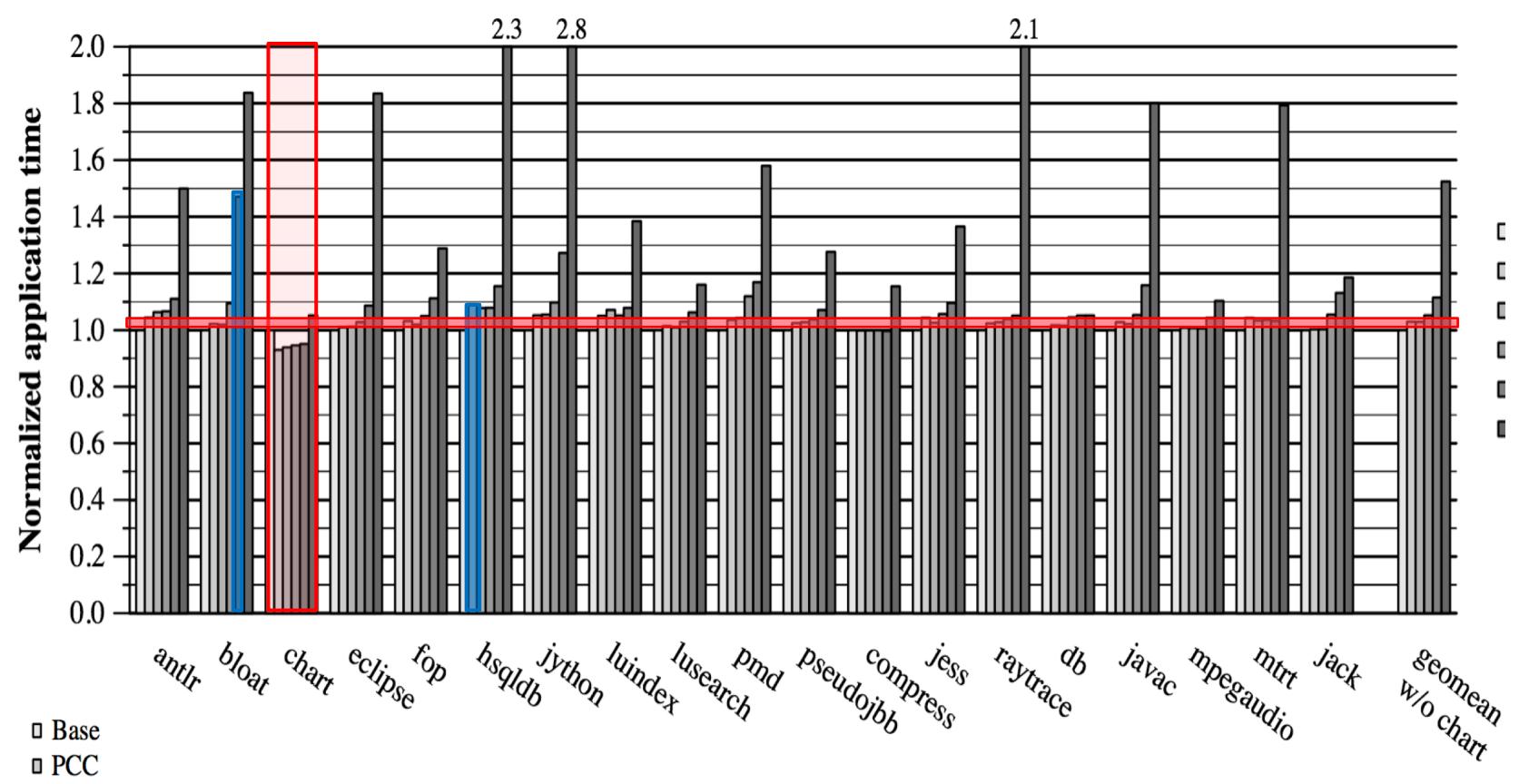
PCC + \*:

PCC計装 + 要求クライアント

Baseを基準として正規化

→Base=1

要求クライアントは5-2準拠



目的: PCCがCCをどれほど安価に区別ができるか評価

- chartの実行時間が減っているのはトレースキャッシュ等が変化したことによる例外
- PCC単体は平均+3%のオーバーヘッド
  - 最大値は+9%(hsqldb)
  - util(平均+2%)とAPI(平均+9%)はPCCクエリの数に対しては随分低コスト(実験5-2)
  - 最大値オーバーヘッドは+47%(bloat @API calls)

# 実験評価

## 5-3: PCCのパフォーマンス:コンパイルオーバーヘッド

Base: PCC計装なし

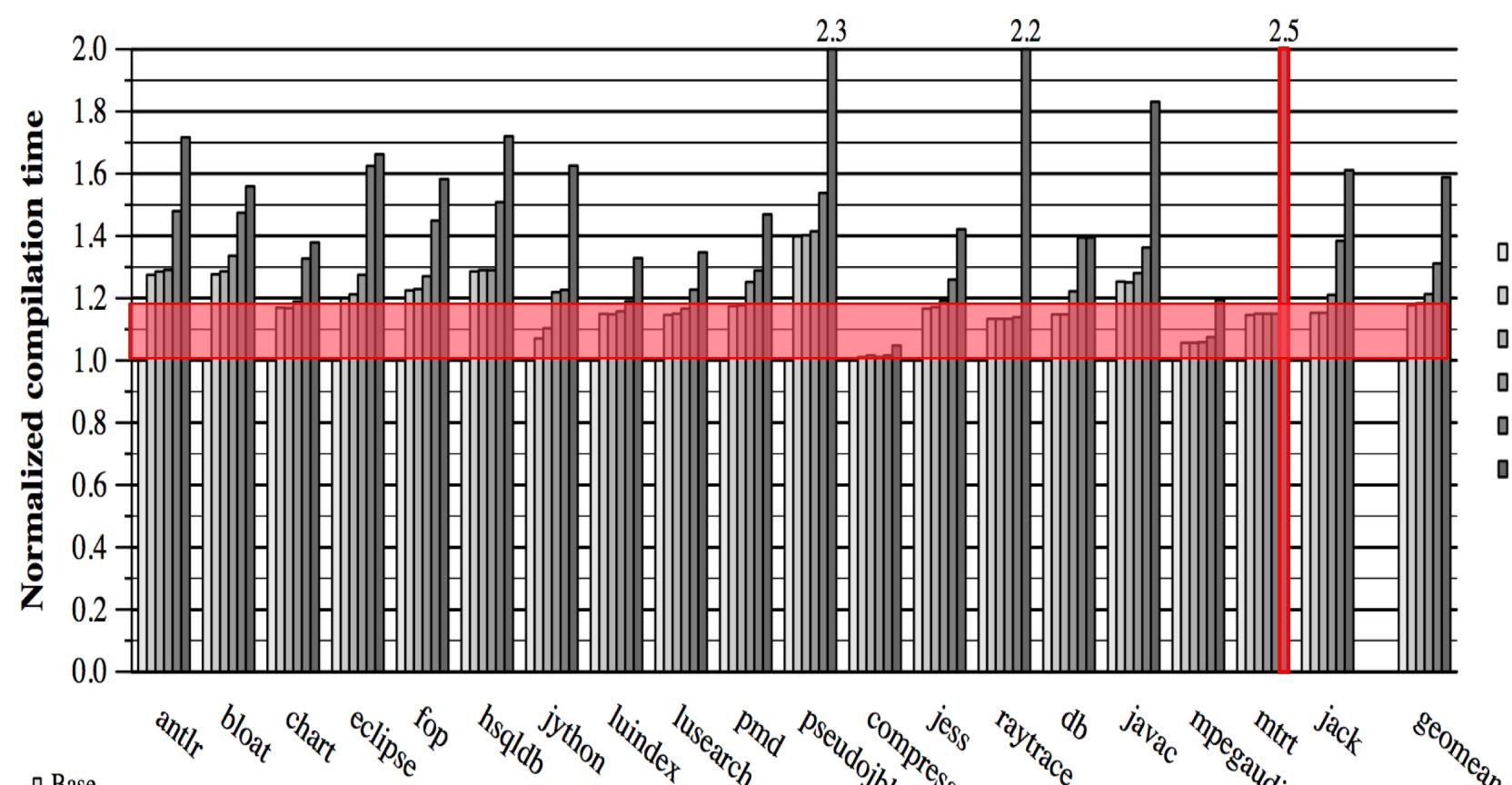
PCC: PCC計装単体

PCC + \*:

PCC計装 + 要求クライアント

Baseが基準として正規化

要求クライアントは5-2準拠



- Base
- PCC
- PCC + system calls
- PCC + util calls
- PCC + API calls
- PCC + all calls

- PCCは性質上コンパイル時間が増大
  - 全体の実行時間を考えるとごく一部とみなせる
- PCC単体は平均+18%
- *system, util, API*はPCC単体+0%～+31%
- All callsの最大は+150%(mtrt)
  - インライン化処理をしなければ削減可能

# 評価実験

## 5-3:PCCのパフォーマンス 空間オーバーヘッド

空間オーバーヘッドは主に3つ

- PCC値の格納
  - PCC値の個数に比例
    - →1word/1context
  - PCC値の格納場所によってはローカリティなくなり遅くなる可能性はある
    - →巨大な構造体にPCC値を格納する場合など
- PCC値用のハッシュテーブル
  - 典型的クライアントの実験(5-2)では $2^{20}$ スロット = 4MBのテーブルを用意
    - 実際のクライアントでは必要に応じて拡張
- 追加コード
  - コンパイルで追加付与されるコード
    - 平均値は以下の通り

目的:PCCによって発生する空間オーバーヘッドの軽量さを評価

PCC	PCC+SystemCalls	PCC + Util Calls	PCC + API Calls	PCC + All Calls
18%	0%	2%	6%	14%

追加コードでのスペースオーバーヘッド

# 実験評価

## 5-3:PCCのパフォーマンス

### vs Stack-walking(※)

SW: Stack-Walking

Base: SW計装なし

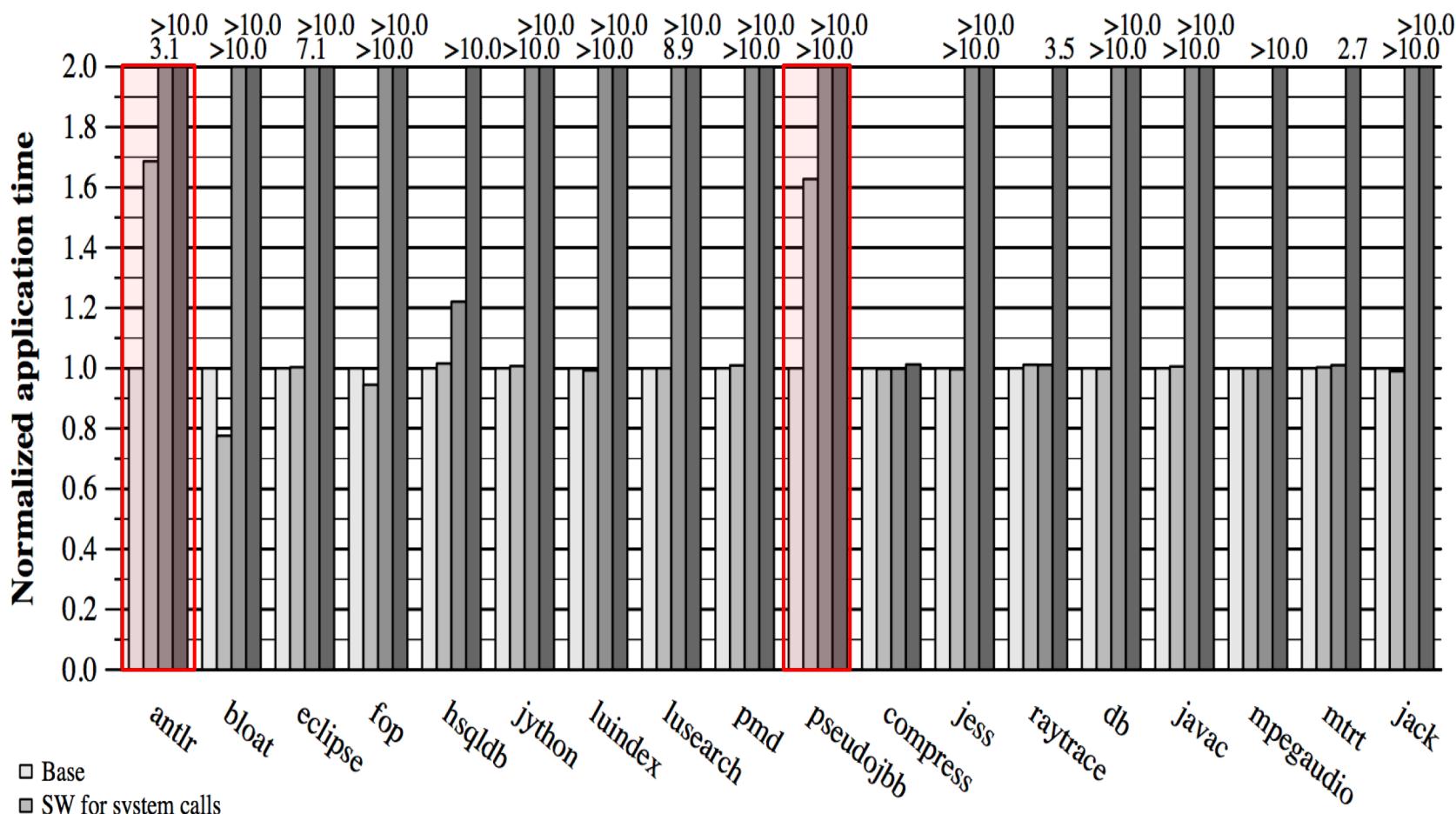
SW for \*:

SW計装+要求クライアント

Baseが基準として正規化

要求クライアントは5-2準拠

※最適化は省略



- PCCクエリと同じタイミングでSWを実行させる
- All Callsはオーバーヘッドが大きくなりすぎるので割愛
- システムコールは無視できるオーバーヘッド
  - ただし、antlr, pseudojbbは200,000以上のシステムコールが呼ばれるので高価
  - 攻撃対象になりやすいアプリではシステムコールが頻繁に呼ばれる可能性
- システムコールを除きPCCよりも高価

# 実験評価

## 5-4: New CCの判定 規模による変化

目的: 入力規模による個別CCの区別精度の評価

Dyn.:

Dynamic(5-2)の増加レート

New Distinct:

Large-Medium “( $\frac{\text{Large} - \text{Medium}}{\text{Large}}$ )”

Conf.: 衝突数

要求クライアントは5-2準拠

Program	Relative increase of large input compared to medium input					
	Java utility calls			Java API contexts		
Dyn.	New distinct	Conf.	Dyn.	New distinct	Conf.	
antlr	2.5x	0 (0.0%)	0	2.5x	8 (0.0%)	0
bloat	11.8x	74,536 (51.9%)	3	10.2x	320,864 (53.4%)	33
chart	2.3x	31,419 (70.6%)	0	2.5x	139,599 (68.9%)	4
eclipse*	4.2x	15,114 (27.9%)	0	5.8x	121,939 (54.0%)	4
fop	1.0x	0 (0.0%)	0	1.0x	0 (0.0%)	0
hsqldb	6.0x	0 (0.0%)	0	2.5x	13 (0.1%)	0
jython	7.5x	12,705 (9.6%)	0	7.3x	59,202 (9.4%)	5
luindex	1.0x	0 (0.0%)	0	1.0x	7,398 (7.2%)	0
lusearch	2.0x	0 (0.0%)	0	2.0x	0 (0.0%)	0
pmd	4.4x	368,862 (83.3%)	24	4.4x	711,223 (84.0%)	79
xalan	10.0x	0 (0.0%)	0	10.0x	15 (0.1%)	0
Dacapo geo	3.5x			3.3x		
compress	1.1x	0 (0.0%)	0	2.2x	0 (0.0%)	0
jess	61.5x	530 (25.7%)	0	56.7x	1,827 (34.9%)	0
raytrace	1.0x	0 (0.0%)	0	11.0x	25 (0.7%)	0
db	71.6x	25 (3.3%)	0	61.4x	72 (5.0%)	0
javac	39.9x	36,419 (65.0%)	0	36.9x	163,916 (64.2%)	6
mpegaudio	1.0x	0 (0.0%)	0	10.9x	32 (1.9%)	0
mtrt	1.0x	0 (0.0%)	0	7.7x	25 (0.7%)	0
jack	8.5x	0 (0.0%)	0	8.5x	0 (0.0%)	0
SPEC geo	6.0x			14.7x		
Geomean	4.4x			6.2x		

- 中規模入力→大規模入力を行い取得CCの差を比較
- 予想: 十分なテストを行ったのなら新しいCCは多くはない
  - 予想に反して多くの新しいCCが検知された
  - →PCCが正しく区別しているということの証左
  - 新しいCCを生成するベンチマークとほとんど生成しないベンチマークの差が顕著

# 実験評価

## 5-4: New CCの判定

### CC Profiling vs Call Site Profiling

目的: コールサイトプロファイリングと比較して、CCプロファイリングから得られる情報量の評価

Large input:

API Callsでの大規模入力

Contextsは5-2と同値

Large-medium diff:

中大規模の差

Contextsは5-4規模による変化と同値

Context w/new call sites:

新しいコールサイトから得られるCC数

Program	Large input		Large-medium diff		Contexts w/ new call sites
	Contexts	Call sites	Contexts	Call sites	
antlr	128,627	4,184	8	0	0
bloat	600,947	3,306	320,864	82	1,002
chart	202,603	2,335	139,599	379	9,112
eclipse*	226,020	9,611	121,939	1,240	46,206
fop	37,710	2,225	0	0	0
hsqldb	16,050	947	13	0	0
jython	628,048	1,830	59,202	1	1
luindex	102,556	654	7,398	0	0
lusearch	905	507	0	0	0
pmd	847,108	1,890	711,223	48	388
xalan	17,905	1,530	15	2	2
Dacapo geo	85,963	1,897			
pseudojbb	3,410	846	17	0	0
compress	1,081	1,017	0	0	0
jess	5,240	1,363	1,827	22	22
raytrace	3,383	1,215	25	2	5
db	1,439	1,105	72	4	4
javac	255,334	1,610	163,916	9	201
mpegaudio	1,668	1,072	32	1	4
mrtt	3,366	1,190	25	2	5
jack	29,461	2,173	0	0	0
SPEC geo	5,410	1,242			
Geomean	24,764	1,568			

- 数千のコールサイトが数十万のCCを生成
- CCは増えるのにコールサイトは増えない場合もある
- 最後の列は新しいコールサイトから得られるCCの数
  - 新しいコールサイトが新しいCCに必ずしも対応しているわけではない

# 関連研究：

---

## Waling-stack

- プログラムスタックを調べ、CCT内の対応するCC情報を得る手法
- 低頻度でもない限り高価すぎる

## CCT

- CCTの各ノードがCCに対応し、動的CCから現在のCCT内の位置を把握し続ける手法
- やはり高価

## サンプリング

- ホットコンテキスト情報から最適化する手法
- 低オーバーヘッドだが、セキュリティ関連の解析には向き

## DCG (Dynamic Call Graph) プロファイリング

- コールエッジをプロファイリングして、最適化の情報を得る手法
- CCプロファイリングよりも情報量が減る

## パスプロファイリング

- CFG内の実行可能パスの個々にそれぞれ数値を計算する手法
- この発想をコールグラフに適応すると指数関数的に計算量が増大する

# まとめ(概要再掲)

---

背景:オブジェクト指向言語は手続き型言語よりもメソッド/関数間の制御の遷移が多量&複雑

- オブジェクト指向言語を分析するのにCalling Contextは有用、しかし計算は高価

目的:Calling Contextを安価に計算できる手法(PCC)を提案

提案手法:PCC(Probabilistic Calling Context)

- CCを区別するためのインデックス計算
  - インデックス値の分散(衝突回避)
  - インデックス計算の軽量化

実験結果:

- インデックス(PCC値)の衝突はほとんどなし
- JVMへの実行オーバーヘッドは平均+3%
  - 基本部分のみ。運用方法で更に実行オーバーヘッドが生じる場合も
- 空間オーバーヘッドは1word/1context(既存研究と比較して安価)
  - + $\alpha$

# 余談: 提案手法

## 3-2: PCC値の計算

### 伝搬係数3について

係数2ではいかんのか？

- 単純なシフト演算になるやないか

→シフト演算の観点から過去の伝搬情報が失われる可能性がある(直訳)

- →具体的には、再帰関数のとき都合が悪い
  - →CCでは再帰の深さでも別個のものに区別したい
- $f(V, cs) = 2 \times V + cs$ だと...
- $f^{32}(V, cs) = (2^{32} \times V + (2^{32} - 1) \times cs) \text{ modulo } 2^{32}$ 
  - $= 2^{32} - cs$
  - $f^{33}(V, cs) = (2 \times 2^{32} \times V + (2 \times 2^{32} - 1) \times cs) \text{ modulo } 2^{32}$ 
    - $= 2^{32} - cs = f^{32}$

[koizumiyuutanoMa  
CS:710 V:2317  
0 V:2317  
1 V:5344  
2 V:11398  
3 V:23506  
4 V:47722  
5 V:30618  
6 V:61946  
7 V:59066  
8 V:53306  
9 V:41786  
10 V:18746  
11 V:38202  
12 V:11578  
13 V:23866  
14 V:48442  
15 V:32058  
16 V:64826  
17 V:64826  
18 V:64826  
19 V:64826]

[koizumiyuutanoMa  
CS:710 V:2317  
0 V:2317  
1 V:7661  
2 V:23693  
3 V:6253  
4 V:19469  
5 V:59117  
6 V:46989  
7 V:10605  
8 V:32525  
9 V:32749  
10 V:33421  
11 V:35437  
12 V:41485  
13 V:59629  
14 V:48525  
15 V:15213  
16 V:46349  
17 V:8685  
18 V:26765  
19 V:15469

2<sup>16</sup>での再帰伝搬(V, csは適当な値)  
左はf(V, cs)=2\*V+cs, 右は3\*V+cs

# 余談: 提案手法

## 3-2: PCC値の計算 実例

- $V_{main} = f(0, 48) = 48$
- $V_A = f(V_{main}, 127) = 48 * 3 + 127 = 271$
- $V_B = f(V_A, 110) = 271 * 3 + 110 = 923$
- $V_C = f(V_B, 348) = 923 * 3 + 348 = 3117$
- $V_{execQuery} = f(V_C, 213) = 3117 * 3 + 213 = 9564$
  
- 仮に、AとBの順番が逆の場合
- $V_{main} = f(0, 48) = 48$
- $V'_B = f(V_{main}, 110) = 48 * 3 + 110 = 254$
- $V'_A = f(V'_B, 127) = 254 * 3 + 127 = 889$
- $V'_C = f(V'_A, 348) = 889 * 3 + 348 = 3015$
- $V'_{execQuery} = f(V'_C, 213) = 3015 * 3 + 213 = 9258$

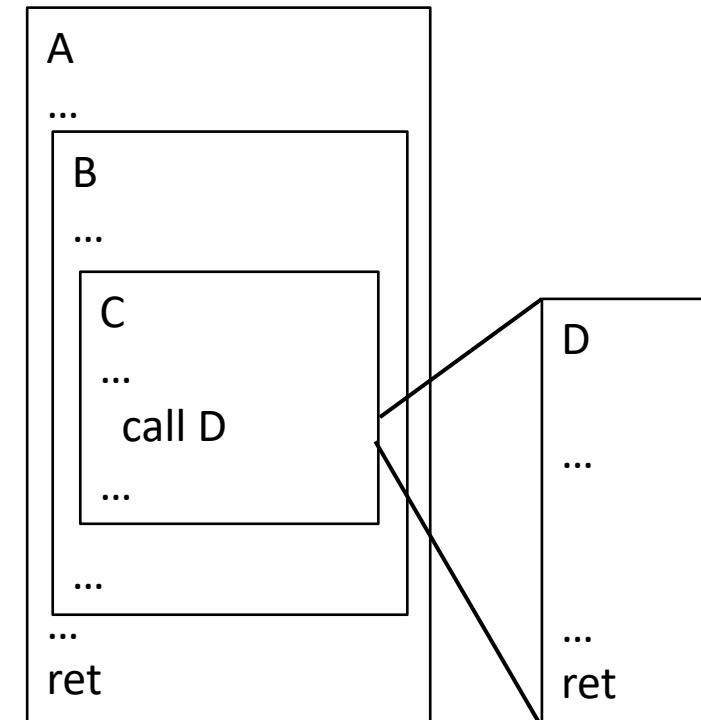
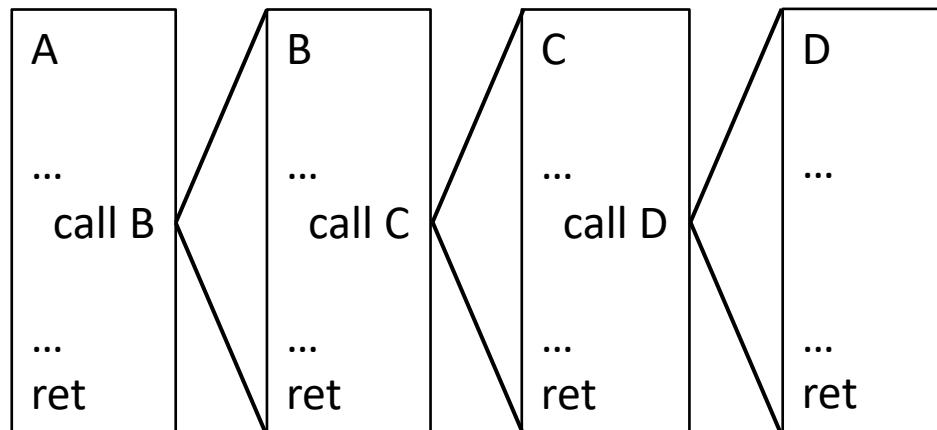
```
at com.mckoi.database.jdbcserver.JDBCDatabaseInterface.  
    execQuery():213  
at com.mckoi.database.jdbc.MConnection.  
    executeQuery():348 C  
at com.mckoi.database.jdbc.MStatement.  
    executeQuery():110 B  
at com.mckoi.database.jdbc.MStatement.  
    executeQuery():127 A  
at Test.main():48
```

# 余談：実装

## 4-1：構築環境 インライン化

昨今の高機能なコンパイラは隙あらばINLINE化してくる

- →INLINE化にも対応することでより効率性を高める
- ex) A→B→C→Dというコール順で、B,CがA内にINLINE化されたら
  - $f(V_A, cs_B), f(V_B, cs_C), f(V_C, cs_D)$ を計算するのは避け、 $f(V_A, cs_{D'})$ を計算したい
- コンパイル時にINLINE化される複数のコールサイトを統合、計算
  - →静的に導出が可能



# 余談：実装

## 4-1：コールサイトの省略

Java.\*メソッドやVMメソッド

- →開発者によっては検査対象外
- →興味がない場合は単純化を施す

```
at result.Value.equals():164
at java.util.LinkedList.indexOf():406
at java.util.LinkedList.contains():176
at option.BenchOption.getFormalName():80
at task.ManyTask.main():46
```



```
at result.Value.equals():164
at option.BenchOption.getFormalName():80
at task.ManyTask.main():46
```

コンテナクラスのjava.utilを介するシーケンス

```
at dacapo.TestHarness.<clinit>():57
at com.ibm.JikesRVM.classloader.VM_Class.
    initialize():1689
at com.ibm.JikesRVM.VM_Runtime.
    initializeClassForDynamicLink():545
at com.ibm.JikesRVM.classloader.
    VM_TableBasedDynamicLinker.resolveMember():65
at com.ibm.JikesRVM.classloader.
    VM_TableBasedDynamicLinker.resolveMember():54
at Harness.main():5
```



```
at dacapo.TestHarness.<clinit>():57
at Harness.main():5
```

VMコールを介するシーケンス

# 参考文献

---

## Probabilistic Calling Context [OOPSLA'07]

- Michael D. Bond, Kathryn S. McKinley
- OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

## Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses [PLDI'10]

- Michael D. Bond , Graham Z. Baker , Samuel Z. Guyer
- PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

## バイナリ変換による透過的なループ構造解析とコード実行時の区間実行時間の計測 [IPSJ'13]

- 佐藤 幸紀
- IPSJ SIG Technical Report , Vol.2013-HPC-140 No.3

# 画像出典

---

<http://pictogram2.com>

- 各ピクトグラム

<https://www.silhouette-illust.com>

- 各ピクトグラム

[http://pages.cs.wisc.edu/~larus/Talks/path\\_talk/tsld028.htm](http://pages.cs.wisc.edu/~larus/Talks/path_talk/tsld028.htm)

- コールグラフ、CCT、DCTイメージ

[http://researcher.watson.ibm.com/researcher/view\\_page.php?id=7235](http://researcher.watson.ibm.com/researcher/view_page.php?id=7235)

- Jikes RVMイメージ

# 单語集

---

## 基本

- CC: Calling Context
- Call Site
- Call Stack

## 関連研究

- SW: stack-walking
- CCT: Calling Context Tree
  - Call Graph
  - Dynamic Call Tree
- Sampling
- Dynamic Call Graph(DCG) Profiling
- Path Profiling
- Call Site Profiling

## CC利用先

- Residual test
- Anomaly-based bug detection
- Anomaly-based intrusion detection

## 性質

- Interprocedural
- Intraprocedural
- Non-commutativity

## その他

- Jikes RVM
- Replay compile
- Inlining