

Leak Pruning[ASPLOS'09]

Michael D. Bond, Kathryn S. McKinley
ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

研究概要: 1-1

背景: 十分なテストしてもプログラムにリークが混入する可能性は消えない

- ガベージコレクションでは将来アクセスされない潜在的リークオブジェクトを排除できない

論文の目的:

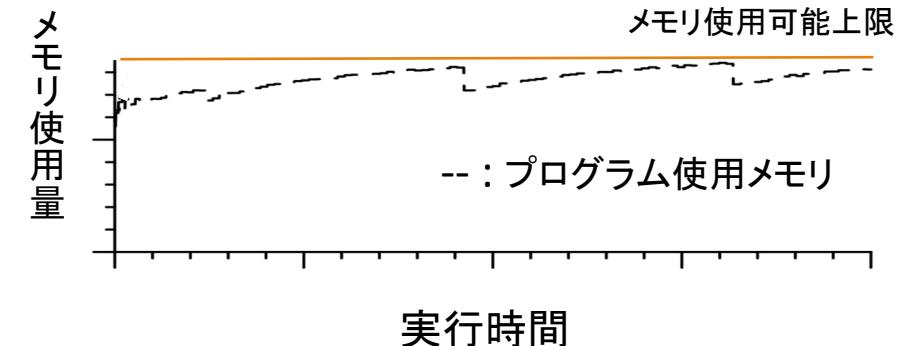
- 適用することでプログラム実行時間を延長する手法の提案
- 手法に用いられた新たなリーク予測アルゴリズムの正確性の実証

提案手法: **Leak Pruning** (pruning: 剪定)

- Stalenessとデータ構造を評価する新たな予測アルゴリズム
 - Staleness: オブジェクトの古さ、経過GC回数で概算
 - データ構造: 型情報と解放可能サイズ
- リーク判定を受けたオブジェクト群を実際に解放してメモリの枯渇を回避

実験結果:

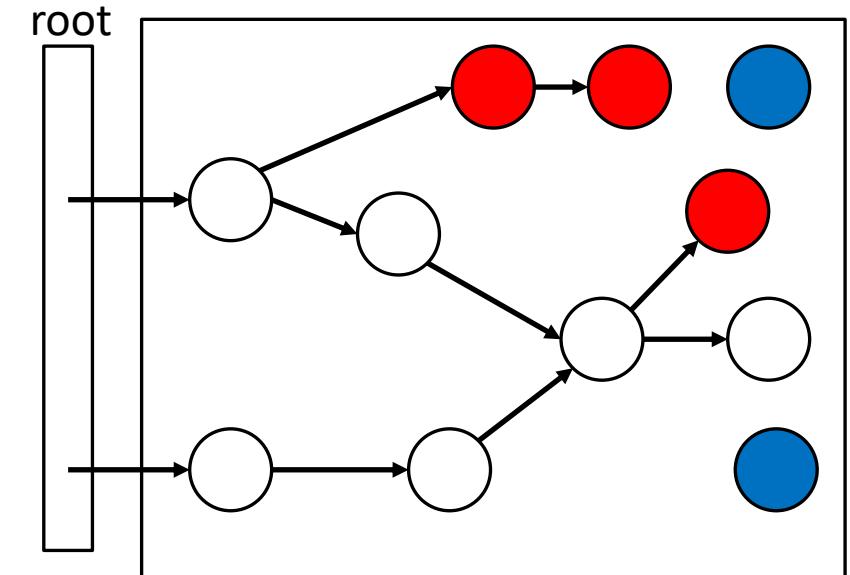
- ベンチマークの3/10で24時間以上の実行、5/10で1.6倍～81倍実行時間が延長
 - 2/10では効果を認められず
- 実行オーバーヘッドは平均+5% (Pentium4), +3% (Core 2)



背景: 2-1 メモリリーク

管理言語(Java、C#など)はGCを使っている

- GCは明確なリークオブジェクトを回収することはできる
 - 明確なリーク: 到達不可能なオブジェクト
 - 今回扱うGCは基本的にMark&Sweep GCとする
- **が、潜在的リークオブジェクトを正確に捉えることは不可能**
 - 潜在的リーク: 到達可能だが将来使うことがないオブジェクト
 - メモリリークはプログラムのパフォーマンス低下、クラッシュを誘発
 - リークを誘引するのはどれも非決定的な要素のため特定が難しい

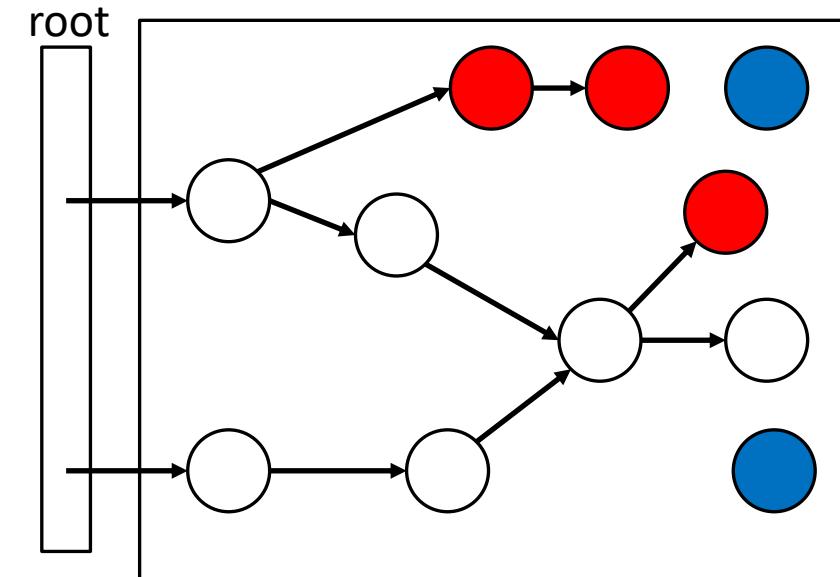


- : GCで回収可能(到達不可)
- : GCで回収不可
(到達可だが将来アクセスなし)

背景: 2-1 メモリリーク

管理言語(Java、C#など)はGCを使っている

- GCは明確なリークオブジェクトを回収することはできる
 - 明確なリーク: 到達不可能なオブジェクト
 - 今回扱うGCは基本的にMark&Sweep GCとする
- **が、潜在的リークオブジェクトを正確に捉えることは不可能**
 - 潜在的リーク: 到達可能だが将来使うことがないオブジェクト
 - メモリリークはプログラムのパフォーマンス低下、クラッシュを誘発
 - リークを誘引するのはどれも非決定的な要素のため特定が難しい



: GCで回収可能(到達不可)

: GCで回収不可
(到達可だが将来アクセスなし)

こいつを見つけていいんじゃあ

背景: 2-1 メモリリーク

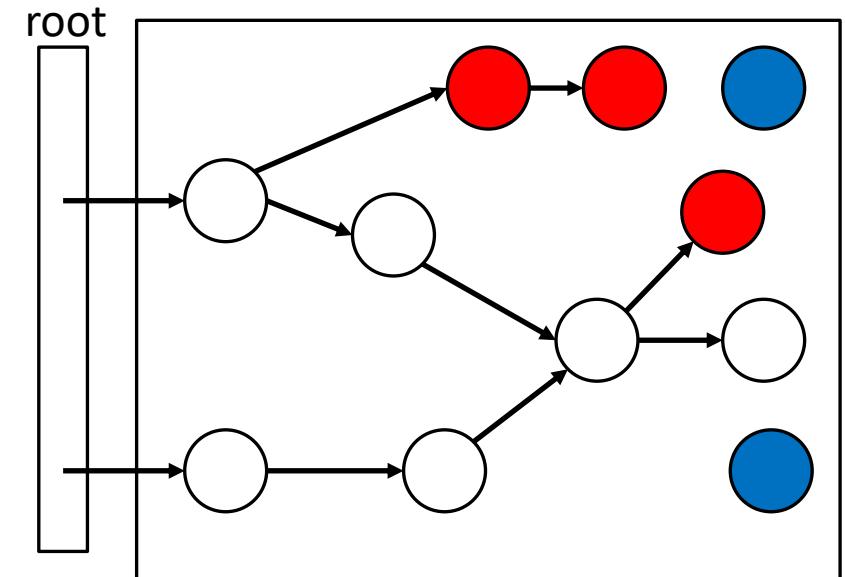
管理言語(Java、C#など)はGCを使っている

- GCは明確なリークオブジェクトを回収することはできる
 - 明確なリーク: 到達不可能なオブジェクト
 - 今回扱うGCは基本的にMark&Sweep GCとする
- が、潜在的リークオブジェクトを正確に捉えることは不可能
 - 潜在的リーク: 到達可能だが将来使うことがないオブジェクト
 - メモリリークはプログラムのパフォーマンス低下、クラッシュを誘発
 - リークを誘引するのはどれも非決定的な要素のため特定が難しい

既存研究が用いてきたStalenessのみの判定では不正確

- (´・ω・`)!
 - → Stalenessとデータ構造の両方を評価することでより正確に判定する

こいつを見つければいいんじゃあ



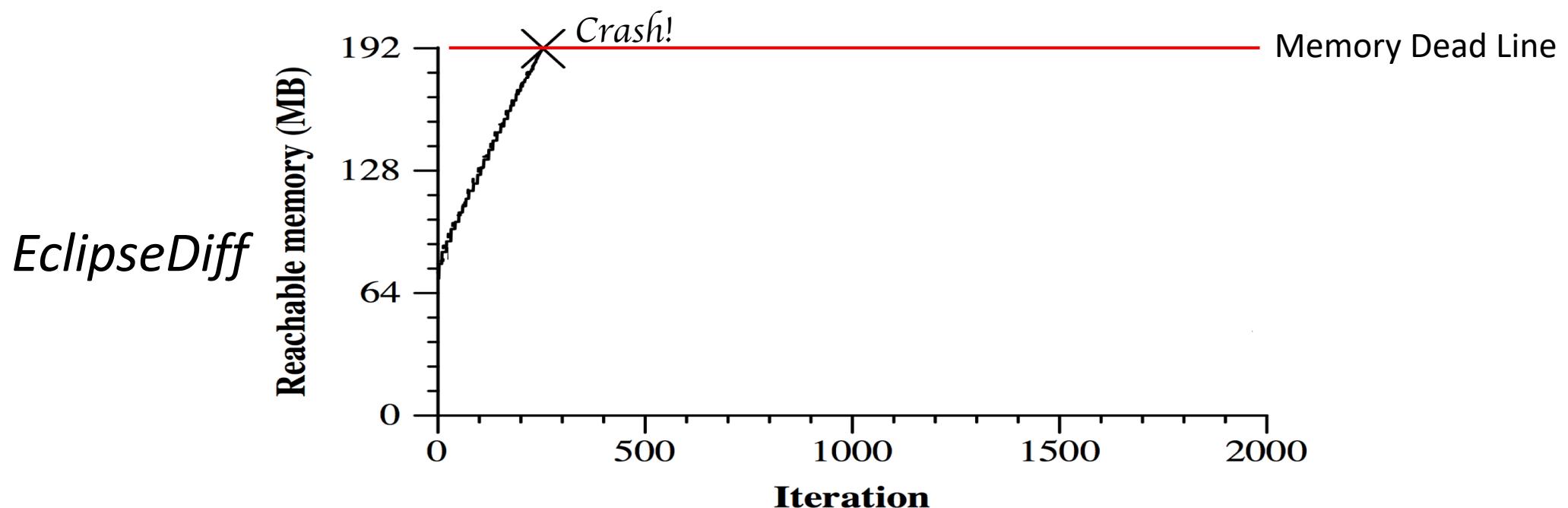
赤い丸: GCで回収不可
(到達可だが将来アクセスなし)

背景: 2-2:モチベーション

リーク予測のための新規アルゴリズムの検証

適用することでプログラムの実行時間を延長する手法の提案

-

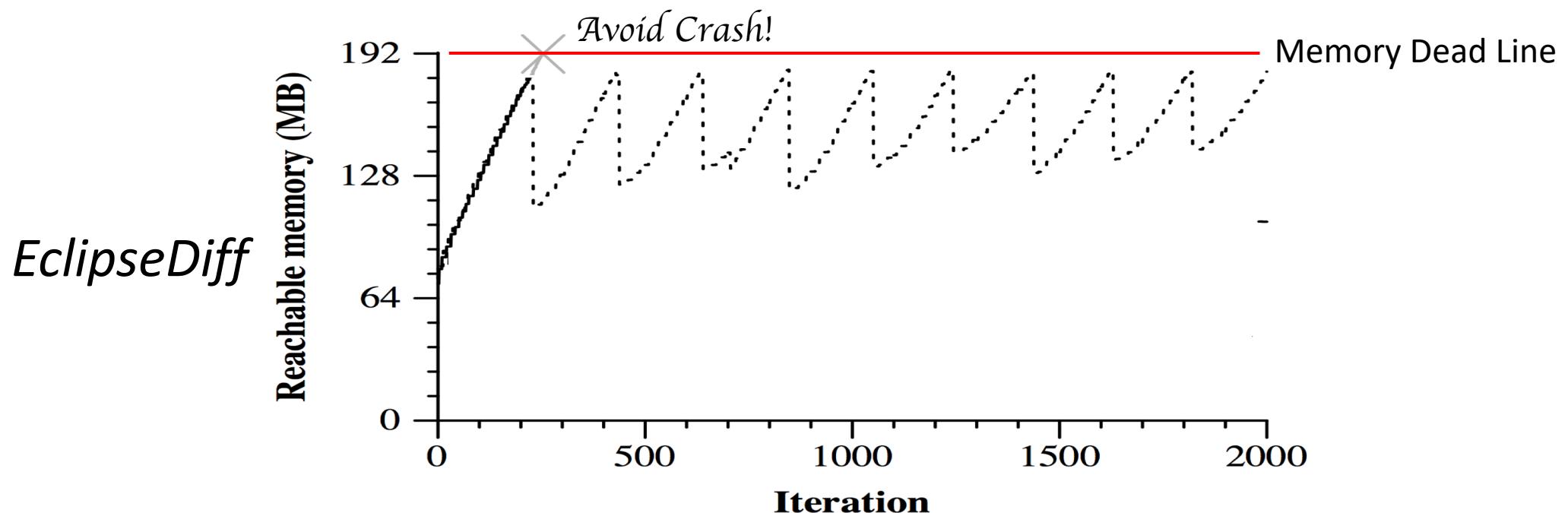


背景: 2-2:モチベーション

リーク予測のための新規アルゴリズムの検証

適用することでプログラムの実行時間を延長する手法の提案

- 究極的には、リークを引き起こして停止してしまうプログラムを無期限に実行



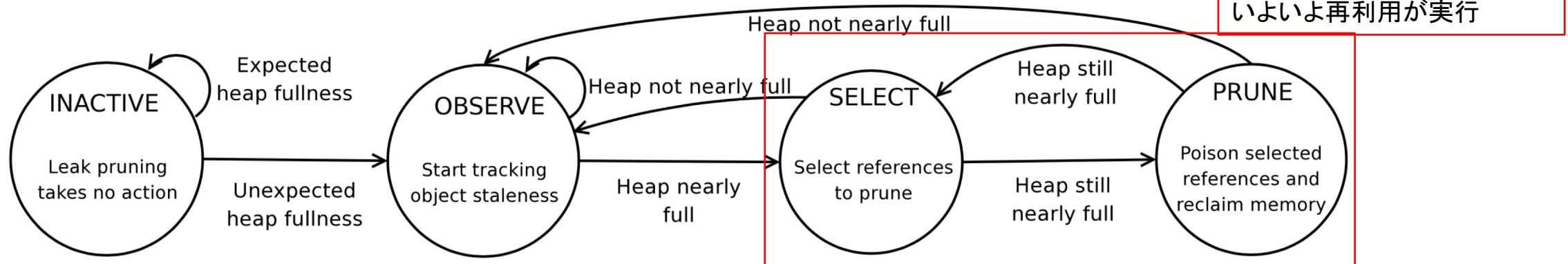
提案手法: 3-1 リークプルーニングの状態遷移

プルーニングの状態は以下4つ

- INACTIVE: ヒープ余裕有。何もしない。ヒープ使用率が閾値Aを超えるとOBSERVEへ
 - デフォルト閾値A: 50%
- OBSERVE: オブジェクトの監視開始。ヒープ使用率が閾値Bを超えるとSELECTへ
 - デフォルト閾値B: 90%
- SELECT: 剪定対象をOBSERVEで得た情報から選定
- PRUNE: SELECTで選択した参照を毒付けして再利用

ヒープ使用率は
フルGC後の値を参照
&
フルGC後に毎回状態遷移
(同じ状態になることもある)

連続するフルGCで
ヒープ使用率が満杯に近いと
いよいよ再利用が実行



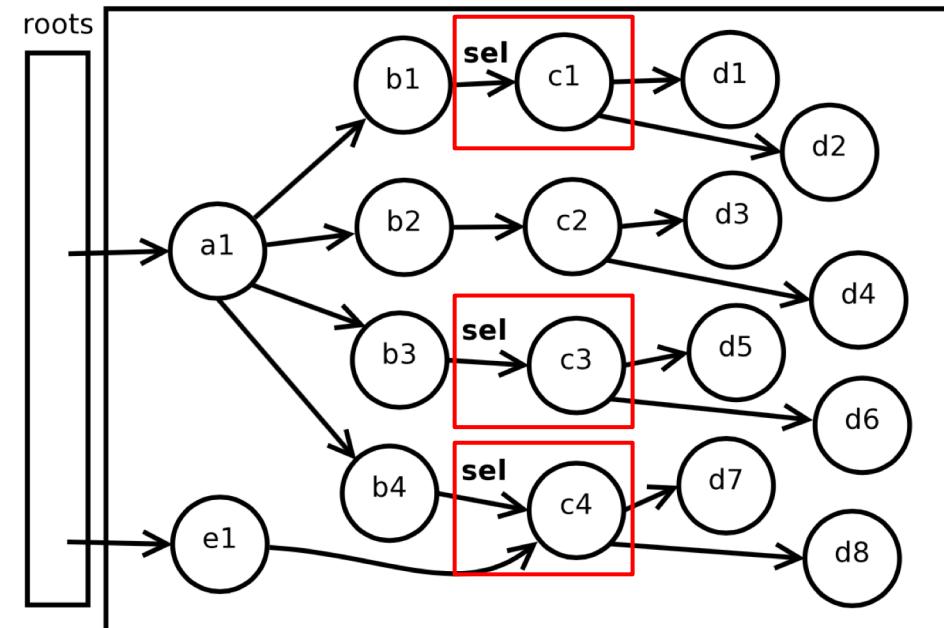
提案手法: 3-2 SELECT → PRUNE 剪定

OBSERVE状態で付与した情報を元に剪定対象を決定(図内のselエッジ)

- 対象の決定(4-3)
 - オブジェクトの古さとデータ構造を評価

剪定ではselエッジを切り(参照できないようにし)、
POISONING(3-3)

- エッジを切るとマークスイープGCは辿れない
 - →辿れない=到達不可なのでGCはその領域を解放する
 - →再利用
 - →別のルートから辿れるのであれば解放されない



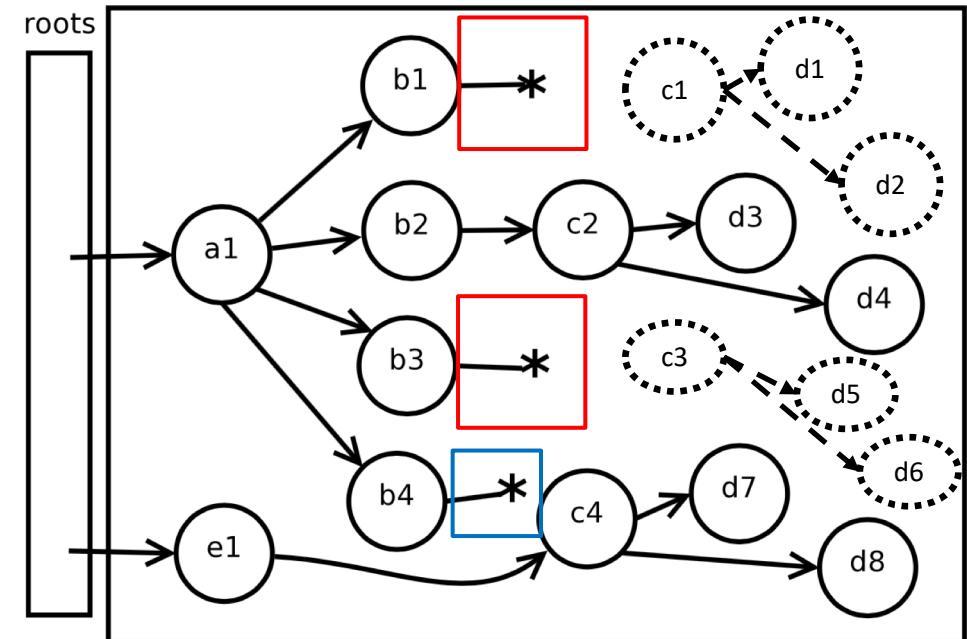
提案手法: 3-2 SELECT → PRUNE 剪定

OBSERVE状態で付与した情報を元に剪定対象を決定(図内のselエッジ)

- 対象の決定(4-1③)
 - オブジェクトの古さとデータ構造を評価

剪定ではselエッジを切り(参照できないようにし)、
POISONING(3-3)

- エッジを切るとマークスイープGCは辿れない
 - →辿れない=到達不可なのでGCはその領域を解放する
 - →再利用
 - →別のルートから辿れるのであれば解放されない



提案手法: 3-3 POISONING

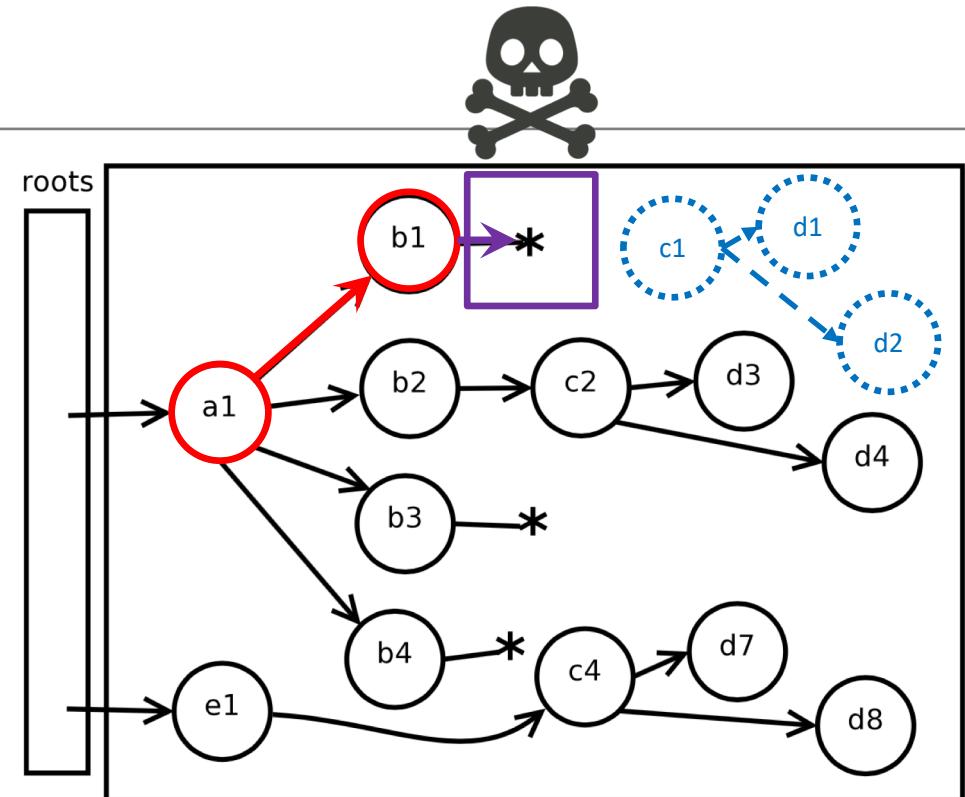
剪定で切ったエッジは通らないという予測

予測が外れて、再アクセスが起きた場合に備えて、
POISONINGを剪定場所に施す

- 実装(4-2)

POISONING箇所に再アクセスが起きた場合

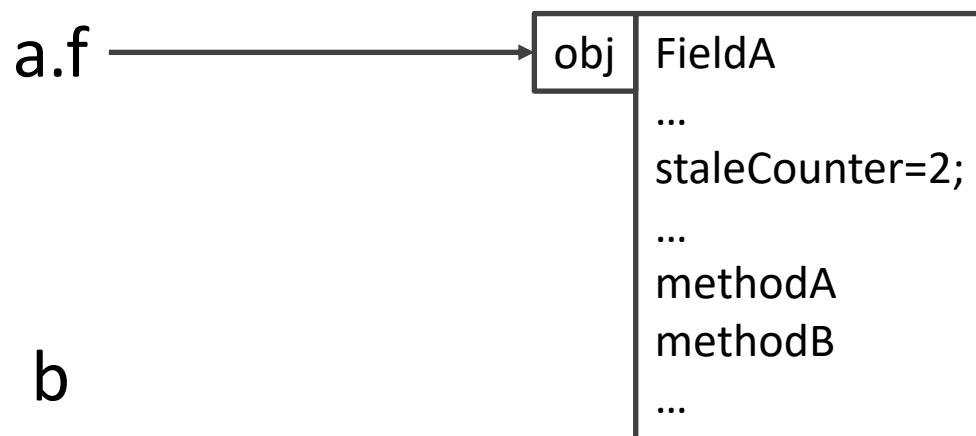
- 情報を探してプログラム終了
- メモリ不足の警告
- 剪定先のデータ構造
 - 右図では型cと2*型dの情報
- 仮にPOISONINGでプログラム終了してもセマンティクス違反にはならない
- 余談: セマンティクス違反



実装: 4-1 リーク判定アルゴリズム①

OBSERVE状態になると、各オブジェクトにstaleCounter(sc)が付与される

- staleCounter: 3bitで表現され、最終アクセスから経過したGCの回数の対数に大まかに対応
 - おおよそ 2^{sc} 回のGCが最終アクセス以降実行されたことを表現
 - インクリメント条件: 2^k 分割するフルヒープGCが実行されたとき、オブジェクトのstaleCounterが $sc == k$ を満たす
 - リセット条件: オブジェクトへのアクセスがあった場合
 - コンパイラを修正してリードバリアを付与

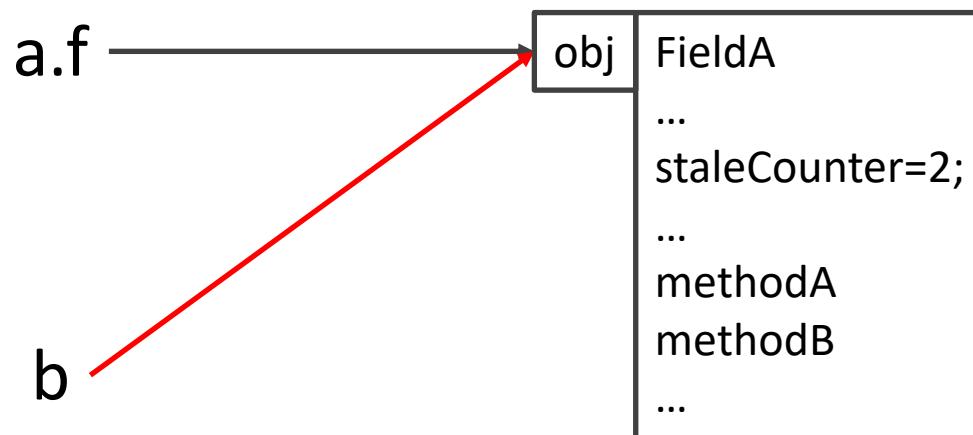


```
b = a.f;    //App code
//-----read barria----//
if(b.staleCounter > 0){
    b.staleCounter = 0;
}
//-----read barria----//
```

実装: 4-1 リーク判定アルゴリズム①

OBSERVE状態になると、各オブジェクトにstaleCounter(sc)が付与される

- staleCounter: 3bitで表現され、最終アクセスから経過したGCの回数の対数に大まかに対応
 - おおよそ 2^{sc} 回のGCが最終アクセス以降実行されたことを表現
 - インクリメント条件: 2^k 分割するFull GCが実行されたとき、オブジェクトのstaleCounterが $sc == k$ を満たす
 - リセット条件: オブジェクトへのアクセスがあった場合
 - コンパイラを修正してリードバリアを付与

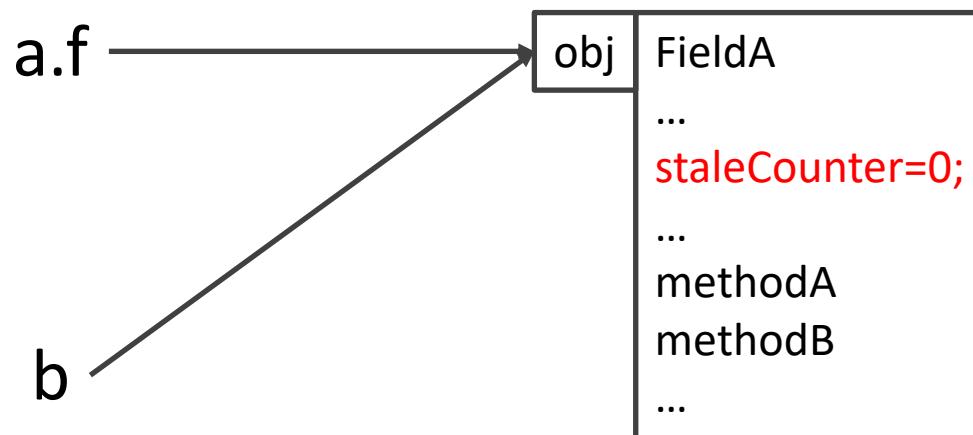


```
b = a.f; //App code
//-----read barria----//
if(b.staleCounter > 0){
    b.staleCounter = 0;
}
//-----read barria----//
```

実装: 4-1 リーク判定アルゴリズム①

OBSERVE状態になると、各オブジェクトにstaleCounter(sc)が付与される

- staleCounter: 3bitで表現され、最終アクセスから経過したGCの回数の対数に大まかに対応
 - おおよそ 2^{sc} 回のGCが最終アクセス以降実行されたことを表現
 - インクリメント条件: 2^k 分割するFull GCが実行されたとき、オブジェクトのstaleCounterが $sc == k$ を満たす
 - リセット条件: オブジェクトへのアクセスがあった場合
 - コンパイラを修正してリードバリアを付与



```
b = a.f; //App code
//-----read barria----//
if(b.staleCounter > 0){
    b.staleCounter = 0;
}
//-----read barria----//
```

実装: 4-1 リーク判定アルゴリズム②

エッジテーブル:

- 参照元と参照先の関係をオブジェクトタイプに基づいてまとめたもの
 - **maxStaleUse(msu)**: OBERVE状態へ推移以降全期間内の tgt_{class} の最大staleCounter
 - →OBSERVE状態の間更新される
- **bytesUsed**: 各エッジ内でmaxStaleUseの条件を満たした参照先のみから到達可能なオブジェクトの合計サイズ
 - =各エッジの**再利用可能なサイズ**
 - →SELECT状態で確定
 - →条件は次ページ

src_{class}	tgt_{class}	maxStaleUse	bytesUsed
Country	Prefecture	2	*
Prefecture	City	2	*
City	Population	4	*
Prefecture	Population	2	*
Mountain	River	0	*
...

実装: 4-1 リーク判定アルゴリズム③

SELECT状態で各エッジの tgt_{class} のオブジェクトのうち、

- (1) $staleCounter \geq maxStaleUse + 2$ を満たすオブジェクト参照(エッジ)を検索
 - 前述の条件
- (2) 条件を満たすエッジのみから辿れるオブジェクトの合計サイズを $bytesUsed$ に格納
 - 要するにに解放可能な合計サイズを格納
- 全エッジの内、この $bytesUsed$ が最大となるエッジをプルーニング対象とする
 - プルーニング対象となったエッジは、上の条件を満たす箇所すべてが切られる
 - 切られたエッジはPOISONINGが設定される(3-3)
 - POISONINGへのアクセスはリードバリアを設置

src_{class}	tgt_{class}	$maxStaleUse$	$bytesUsed$
Country	Prefecture	2	300
Prefecture	City	2	200
City	Population	4	20
Prefecture	Population	2	100
Mountain	River	0	0
...

実験結果 : 5-1: 実験環境

VM:

- Jikes RVM 2.9.2
 - 最適化コンパイル
 - タイマーベースのため非決定的
 - →リプレイコンパイルに変更

GC:

- Memory Management Toolkit(MMTk)
- 世代別マーク&スイープGC
 - 平行処理、stop-the-world

ベンチマーク:

- DaCapo(ver2006-10)
- SPECjbb2000
- SPECjvm98

} オーバーヘッド測定用

プラットフォーム:

- Linux 2.6.20.3
- Dual Core 3.2GHz Pentium4
 - 64-byte L1, L2 cache line size
 - 16-KB 8-way set associative L1 data cache
 - 12-K μ ops L1 instruction trace cache
 - 1-MB unified 8-way set associative L2 on-chip cache
- Core 2 Quad 2.4 GHz
 - 64-byte L1 and L2 cache line size
 - 8-way 32-KB L1 data/instruction cache
 - each pair of cores shares a 4-MB 16-way L2 on-chip cache

各実験の記録について

- 5回以上実行した結果の平均値

実験結果: 5-2: 実行 オーバーヘッド

Pentium4

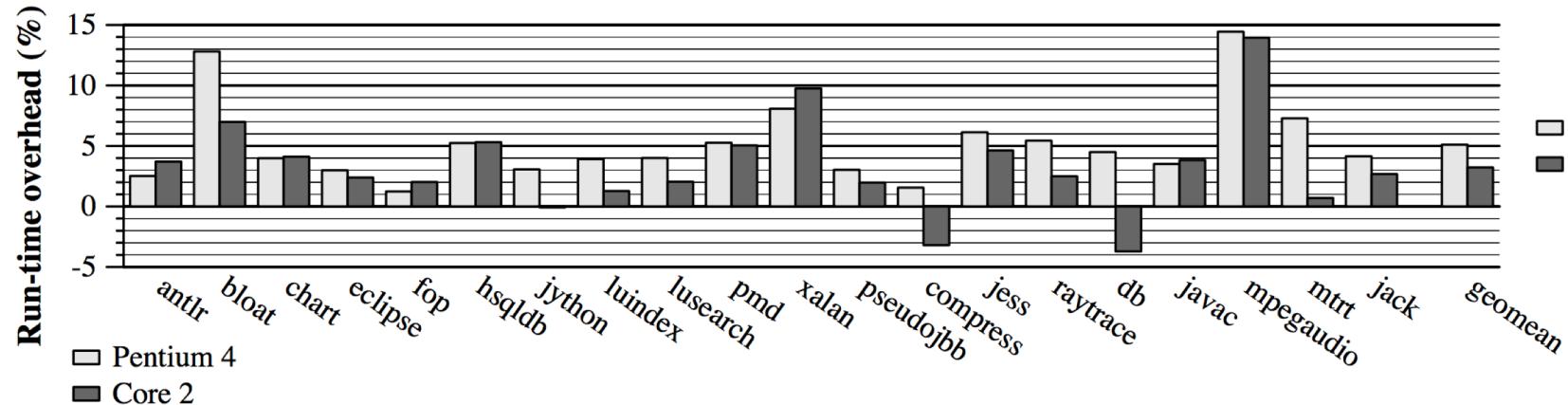
Core2

SPECは5-1参照

ヒープ上限は最小実行可能サイズの2倍

GCのオーバーヘッドを含む

各バーは5回の試行の中央値



目的:

- 実行オーバーヘッドの低さを示し、ネイティブの実行と遜色ない速度を維持することの証明(私感)

前提:

- リークが発生しなくてもSELECT状態へ移行
 - 明記はないので閾値をいじっているか、定期的に遷移させているか

得られた内容:

- 平均してオーバーヘッドは+5%(Pentium4), +3%(Core2)
- オーバーヘッドの大半はOBSERVE状態でのリードバリア
 - =アクセスの監視
 - SELECT状態のオーバーヘッドは無視できるレベル

実験結果: 5-2

GC オーバーヘッド

環境はPentum4

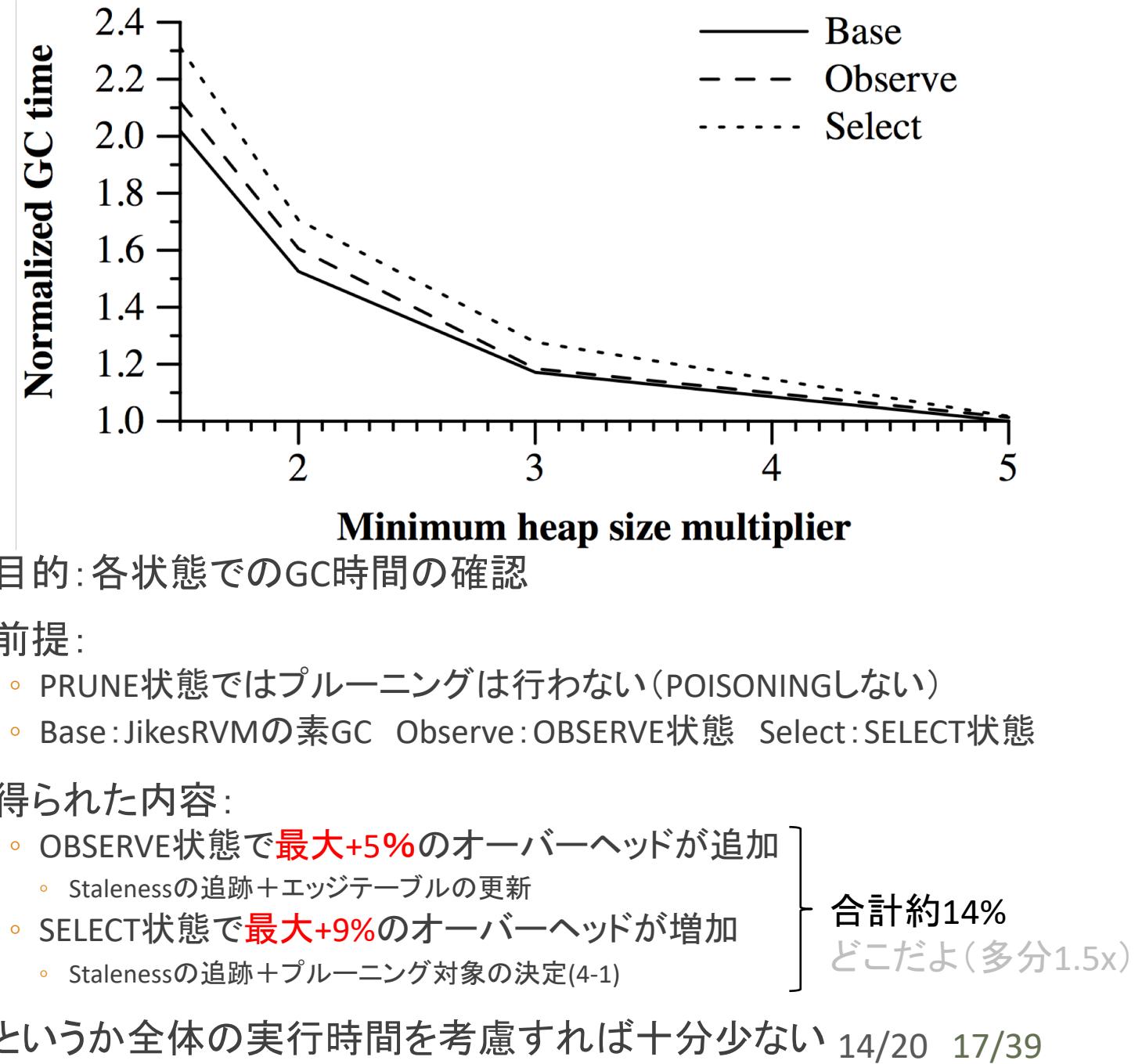
Core2は同様の値なので省略

横軸: ヒープサイズ上限

最小ヒープサイズから見た倍率
(1.5x ~ 5.0x)

縦軸: 正規化GC実行時間

すべてのベンチマークの
幾何平均 $\sqrt{\sum t_i^2}$
ヒープ5倍、Baseの幾何平均が1
(明記なし)



実験結果 : 5-3 リークの実例 全体

Leak (LOC)	Effect	Reason
EclipsesDiff (2.4M)	Runs >200X longer	Almost all reclaimed
ListLeak (9)	Runs indefinitely	All reclaimed
SwapLeak (33)	Runs indefinitely	All reclaimed
EclipseCP (2.4M) MySQL (75K) SPECjbb2000 (34K) JbbMod (34K) Mckoi (95K)	Runs 81X longer	Almost all reclaimed
	Runs 35X longer	Most reclaimed
	Runs 4.7X longer	Some reclaimed
	Runs 21X longer	Most reclaimed
	Runs 1.6X longer	Some reclaimed
	No help	None reclaimed
Delaunay (1.9K)	No help	Short-running

Table 1. Ten leaks and leak pruning's effect on them.

Open-source: EclipseDiff, EclipseCP, MySQL, Mckoi

Benchmark: SPECjbb2000, JbbMod

Micro-benchmark: ListLeak, SwapLeak, DualLeak

同僚のリークアプリケーション: Delaunay

24時間以上
実行

普通よりは
長く実行

長く実行
できなかった

実験結果 : 5-3 リークの実例 EclipseDiff

横軸: 繰り返し回数

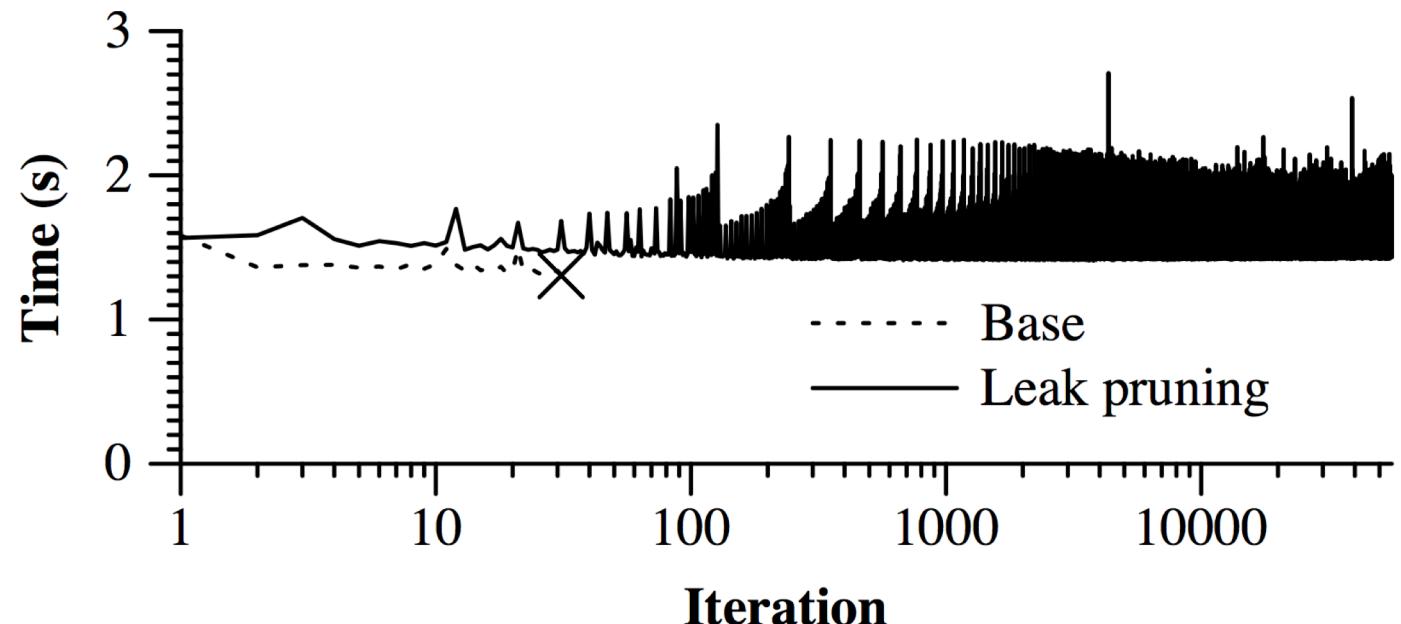
繰り返し = リークを誘引するプログラム作業の固定量

縦軸: 繰り返し一回の時間

$\sum Time$ は稼働時間と近似

Base: プルーニング適用なし

Leak Pruning: 適用あり



目的: 実際にリークするプログラムが延命されることの証明
◦ & 予測アルゴリズムが正しくリーク判定できているかの証明(時間視点)

EclipseDiff: Eclipse3.1.2であったリークバグの再現

◦ NavigationHistory -> ResourceCompareInputという参照が主原因

前提: バツ印はプログラムが停止した印

得られた内容:

- プルーニング適用後は24時間の実行を確認
 - = メモリ不足を引き起こさなかった
 - = メモリリークの改善
- 長期的なスループットは一定

実験結果 : 5-3 リークの実例 EclipseCP

横軸: 繰り返し回数

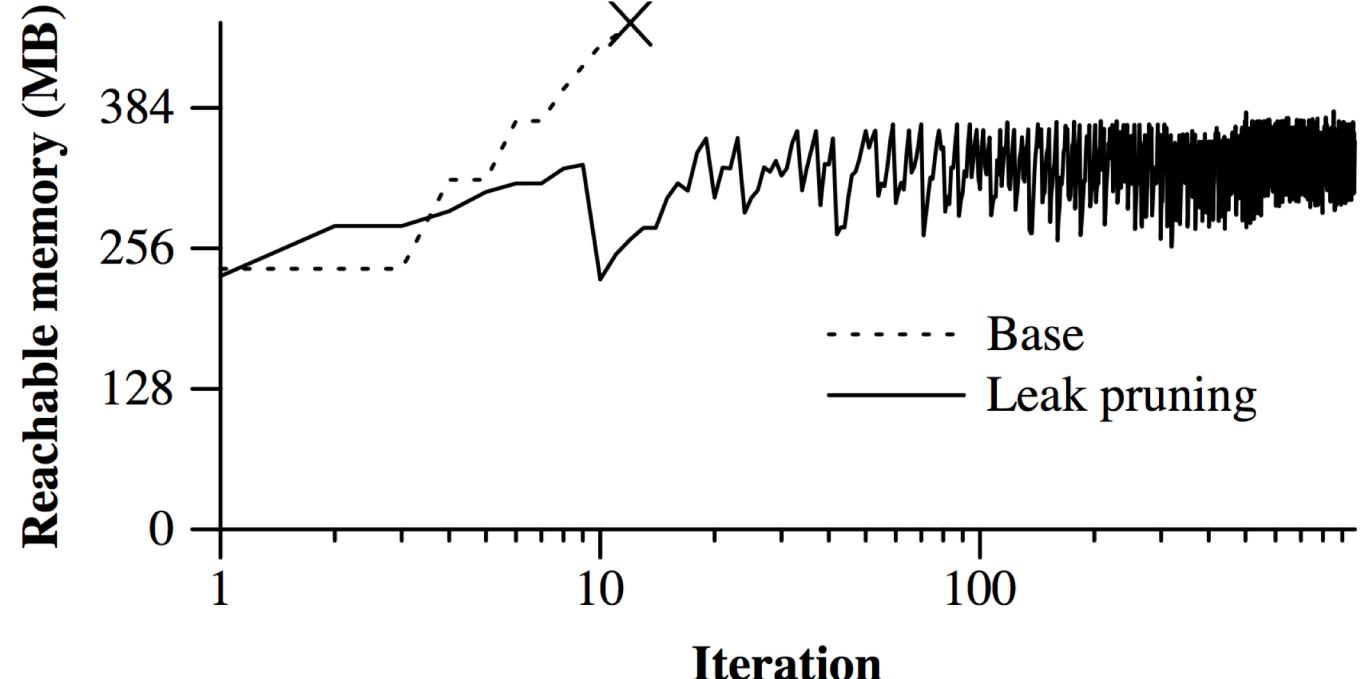
繰り返し = リークを誘引するプログラム作業の固定量

縦軸: 繰り返し一回の時間

$\sum Time$ は稼働時間と近似

Base: プルーニング適用なし

Leak Pruning: 適用あり



Iteration

目的: 実際にリークするプログラムが延命されることの証明

- & 予測アルゴリズムが正しくリーク判定できているかの証明(メモリ視点)

EclipseCP: Eclipseにあったリークバグの再現

- テキストカット→ファイル保存→テキストペースト→ファイル保存→リーク
- が生じるバグが存在

得られた内容:

- 適用なしでは11回目の繰り返し後メモリが枯渇
- 適用ありでは971回(9.5h)稼働
 - 序盤はある二つのエッジを何度もプルーニング
 - 終了までに100種類以上のエッジをプルーニング
 - 最後はPOISONINGを再アクセスして終了

実験結果 : 5-4 他の判定方法 との比較

Most Stale: オブジェクト単位で、
最大のstaleCounterを持つオブ
ジェクトへの参照を全てプルーニ
ング

Indiv refs: エッジテーブルを考慮
せず、参照先単体のstaleCounter,
サイズで判定

Leak Pruning Default: 提案手法

表内の値は全て繰り返しの回数

Leak	Base	LS [35] & Melt [8] Most stale	Indiv refs	Leak pruning Default	Default edge types
EclipseDiff	259	228	3,380	$\geq 55,780$	1,817
ListLeak	110	108	$\geq 2.7M$	$\geq 2.7M$	56
SwapLeak	5	5	11	$\geq 11,368$	83
EclipseCP	11	10	41	971	2,203
MySQL	18	35	114	634	230
SPECjbb2000	135	97	625	632	197
JbbMod	204	41	911	4,267	209
Mckoi	44	47	71	72	308
DualLeak	145	149	144	143	69

目的: 作った予測アルゴリズムが他のアルゴリズムよりも優れて
いることの証明

得られた内容:

- EclipseCPでの主なプルーニング対象の差異
- Indiv refs: String -> char[]
- Leak Pruning:
 - org.eclipse.jface.text.DefaultUndoManager\$TextCommand -> String
 - org.eclipse.jface.text.DocumentEvent -> String
- Stalenessとデータ構造の評価はよりリーク判定の精度が上がる

関連研究: 星取り表

	静動	対象言語	手法	セマンティクス	可用性	外部メモリ	備考
<i>Static detecting leaks</i>	Static	C/C++	Allocation /deallocation評価	保持	リーク判定のみ	必要なし	
<i>Dynamic detecting leaks</i>	Dynamic	C/C++	Allocation /deallocation評価 staleness評価	保持	リーク判定のみ	必要なし	
<i>Cyclic memory allocation</i>	Dynamic	C	Live-object数評価	違反有り	増大	必要なし(?)	アロケーションサイト単位でオブジェクト数を事前に予測
<i>Plug</i>	Dynamic	C/C++	アロケーション修正	保持	増大	必要あり	GC情報なし 物理メモリに退避
<i>Melt</i>	Dynamic	Java/C#	staleness評価	保持	増大	必要あり	リーク判定オブジェクトを外部に退避
<i>Leak Pruning</i>	Dynamic	Java/C#	Staleness +データ構造評価	保持	増大	必要なし	提案手法

まとめ

リークプルーニングは多くのプログラムでセマンティクス違反なく可用性を強化する

- =プログラムの挙動を変えず実行時間を延長する

リークプルーニングの新規性

- 新規予測アルゴリズム(Staleness+データ構造)
- セマンティクスを保持しつつリークオブジェクトを解放するプルーニングの着想

性能評価

- 実行オーバーヘッドは平均+3%~+5%
- ベンチマークのうち8/10で実行時間の延長が確認
- 残りの二つは短時間に終了するプログラムと、ライブオブジェクトが増加したために延長は叶わず

参考文献

Leak Pruning

- Michael D. Bond, Kathryn S. McKinley
- ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation

- Huu Hai Nguyen, Martin Rinard
- ISMM'07, October 21–22, 2007, Montréal, Québec, Canada

Plug: Automatically Tolerating Memory Leaks in C and C++ Applications

- Gene Novark, Emery D. Berger , Benjamin G. Zorn

Tolerating Memory Leaks

- Michael D. Bond, Kathryn S. McKinley
- OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

ガベージコレクション 自動的メモリ管理を構成する理論と実装(和訳版)

- Richard Jones, Antony Hosking, Eliot Moss
- 訳:前田敦司、鶴川始陽、小宮常廉

提案手法: 3-3

余談: セマンティクス違反

セマンティクス違反:

- プログラムの内部処理が変わってしまうこと
 - →線引は曖昧な部分があるが、一般的には意図しない挙動を引き起こす状態を違反とする

Q: Leak Pruningはセマンティクス違反ではないのか？

- →だって勝手に解放してるやん！

A: 違反していません

- POISONINGした箇所がリークの場合: 当然アクセスされないので違反なし
- POISONINGした箇所がリークではなかった場合:
 - プルーニングが行われるのはメモリ不足の直前
 - →プルーニングしなければメモリ不足でプログラム終了
 - POISONINGした箇所へ再アクセスした場合はメモリ不足のエラーをスローするので、結局出力としては同じ
 - 再アクセスまでメモリ不足の延命をしただけなので違反なし

実装: 4-1

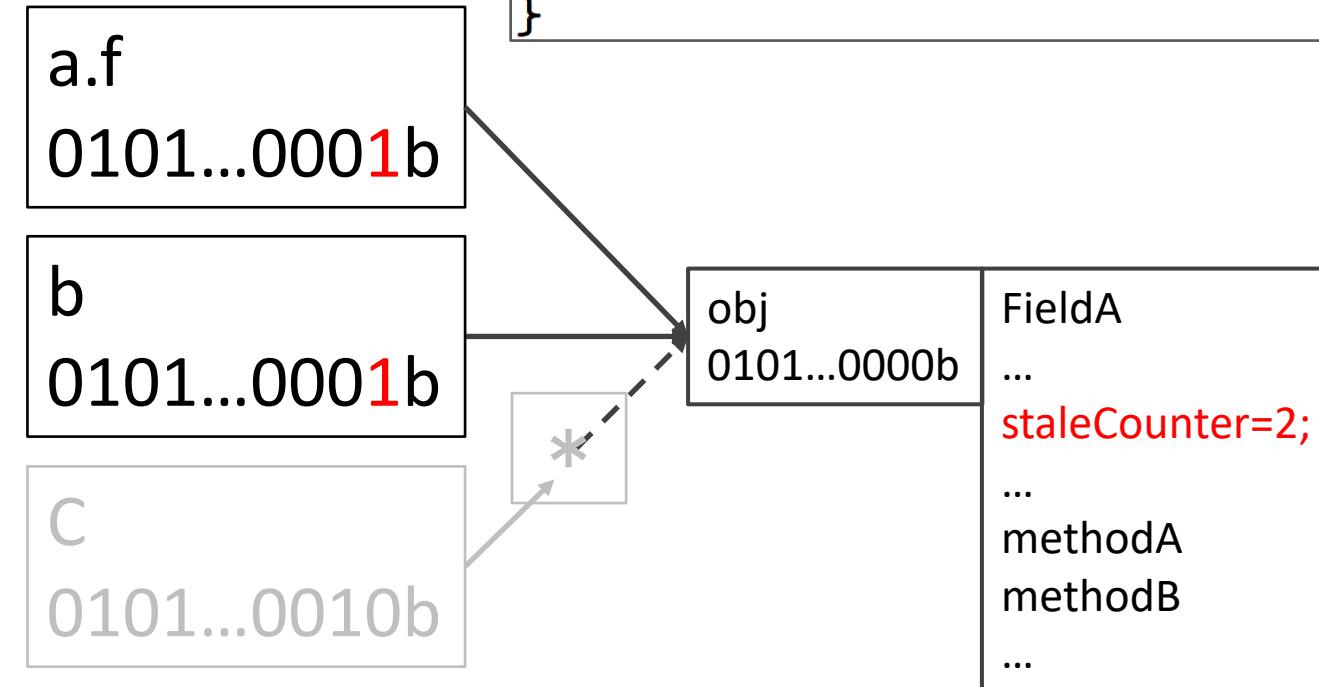
余談: 下位2ビット

各ポインタ先のアドレス下位2ビットに判定情報を付与

- → Javaのアライメントで下位2ビットが00bとなることを利用
- **最下位ビット==1**
 - → 参照先のstaleCounter > 0
 - 下位2ビット目==1
 - → 参照先がPISONING

フラグ管理が効率的にできる

```
b = a.f;           // Application code
if (b & 0x1) {    // Read barrier
    // out-of-line cold path
    t = b;          // Save ref
    b &= ~0x1;      // Clear lowest bit
    a.f = b; [iff a.f == t] // Atomic
    b.staleCounter = 0x0; // Atomic
}
```



実装: 4-1

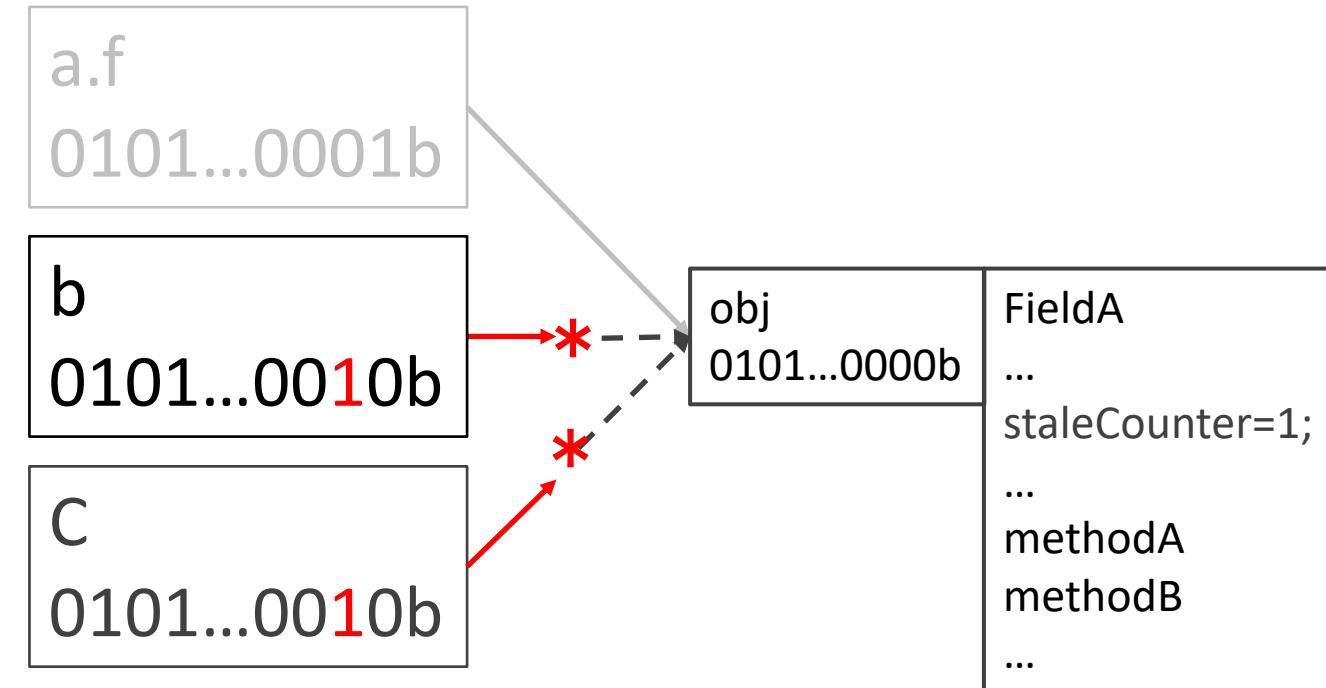
余談: 下位2ビット

各ポインタ先のアドレス下位2ビットに判定情報を付与

- →Javaのアライメントで下位2ビットが00bとなることを利用
- 最下位ビット==1
 - →参照先のstaleCounter > 0
- **下位2ビット目==1**
 - →参照先がPISONING

フラグ管理が効率的にできる

```
if (b & 0x2) { // Check if poisoned
    InternalError err = new InternalError();
    err.initCause(avertedOutOfMemoryError);
    throw err;
}
/* rest of read barrier cold path */
```



実装: 4-1

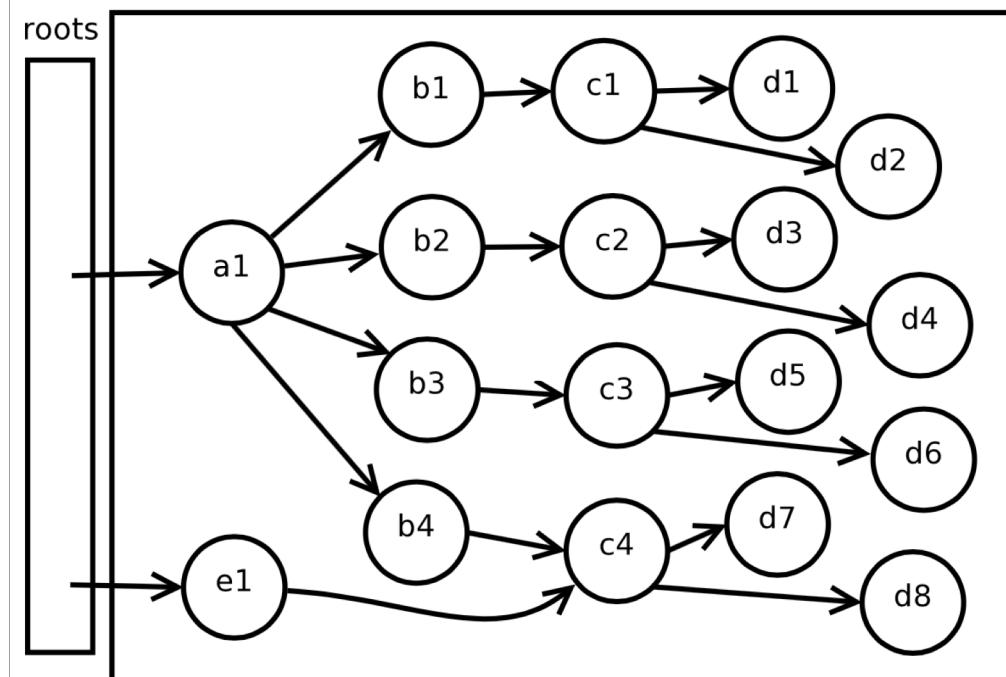
余談: リーク判定アルゴリズム: 実例

1: SELECT 状態遷移時に maxStaleUse(msu) が確定

o

o

o



B → C	maxStaleUse = 0	bytesUsed = 0
E → C	maxStaleUse = 2	bytesUsed = 0

実装: 4-1

余談: リーク判定アルゴリズム: 実例

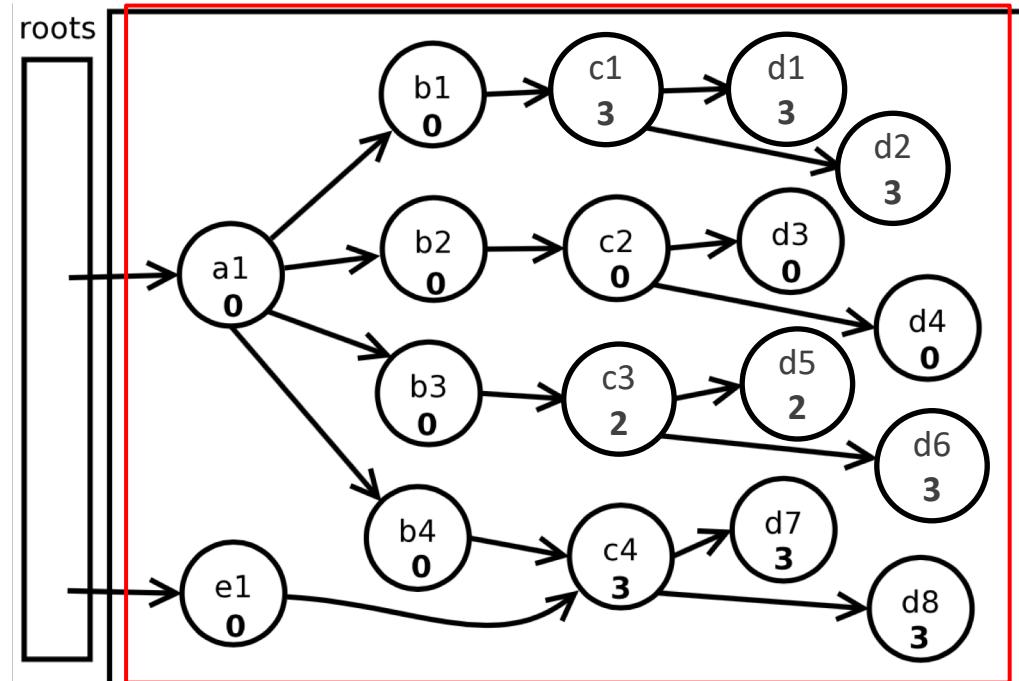
1: SELECT状態遷移時にmaxStaleUse(msu)が確定

2: SELECT状態での各staleCounter(sc)が確定

o

o

o



B → C	maxStaleUse = 0	bytesUsed = 0
-------	-----------------	---------------

E → C	maxStaleUse = 2	bytesUsed = 0
-------	-----------------	---------------

実装: 4-1

余談: リーク判定アルゴリズム: 実例

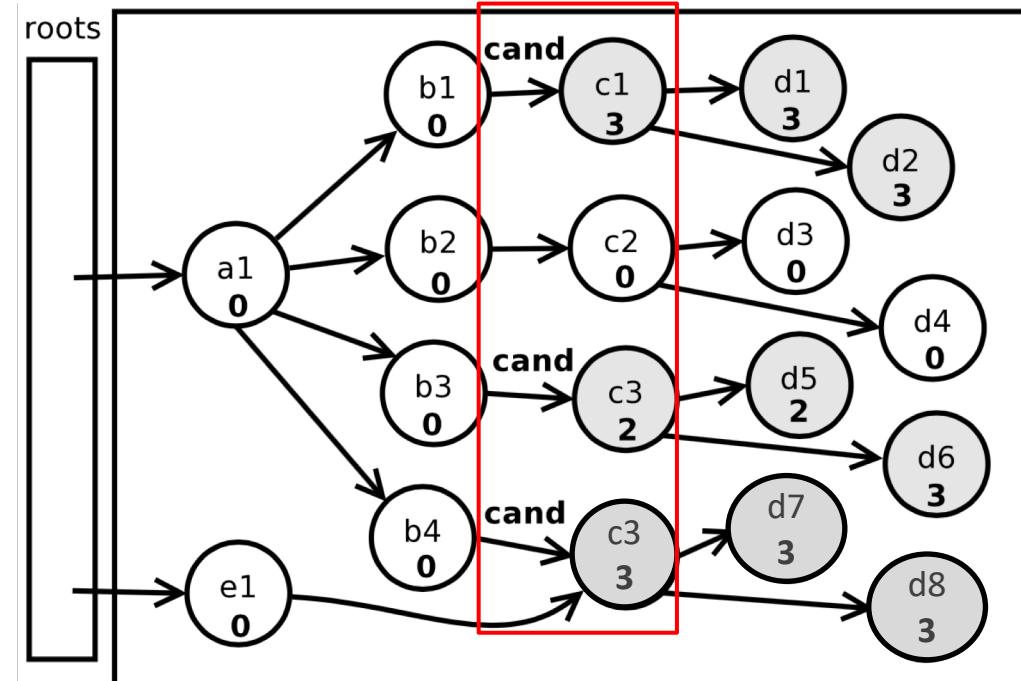
1: SELECT状態遷移時にmaxStaleUse(msu)が確定

2: SELECT状態での各staleCounter(sc)が確定

3.1: 各エッジについてターゲットのscとmsuを比較

- 3.2: 条件を満たしていれば候補参照認定
- 3.3: 候補参照から辿れるオブジェクトをマーク

◦



$B \rightarrow C$	maxStaleUse = 0	bytesUsed = 0
$E \rightarrow C$	maxStaleUse = 2	bytesUsed = 0

実装: 4-1

余談: リーク判定アルゴリズム: 実例

1: SELECT状態遷移時にmaxStaleUse(msu)が確定

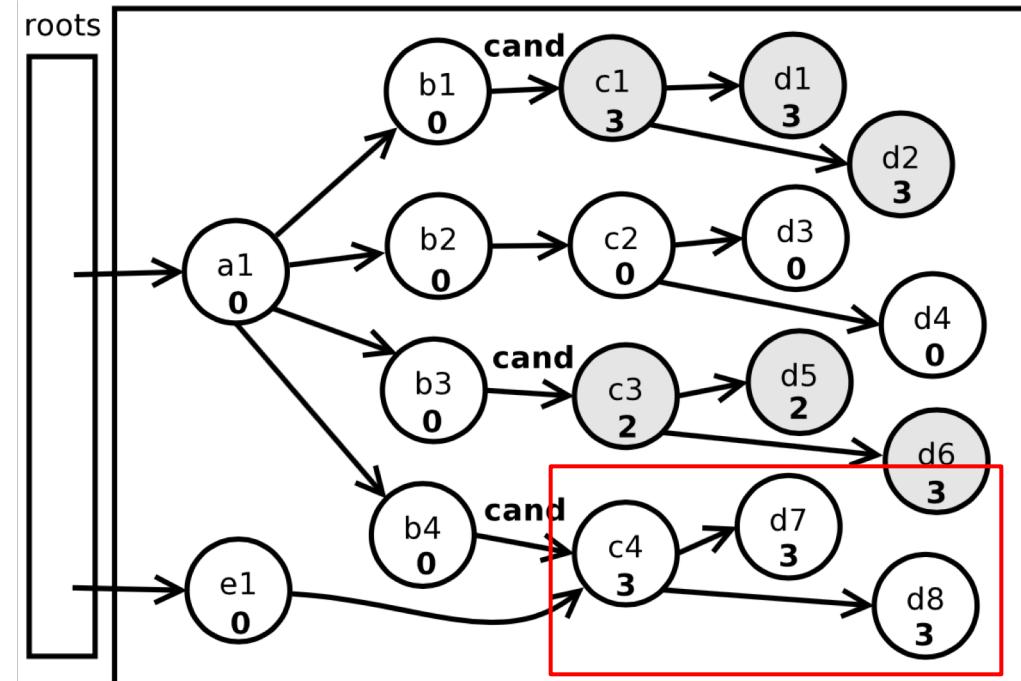
2: SELECT状態での各staleCounter(sc)が確定

3.1: 各エッジについてターゲットのscとmsuを比較

- 3.2: 条件を満たしていれば候補参照認定
- 3.3: 候補参照から辿れるオブジェクトをマーク

4: ルートからオブジェクトをたどり、**候補参照を介さず到達可能なオブジェクトのマークを外す**

◦



$B \rightarrow C$	maxStaleUse = 0	bytesUsed = 0
$E \rightarrow C$	maxStaleUse = 2	bytesUsed = 0

実装: 4-1

余談: リーク判定アルゴリズム: 実例

1: SELECT状態遷移時にmaxStaleUse(msu)が確定

2: SELECT状態での各staleCounter(sc)が確定

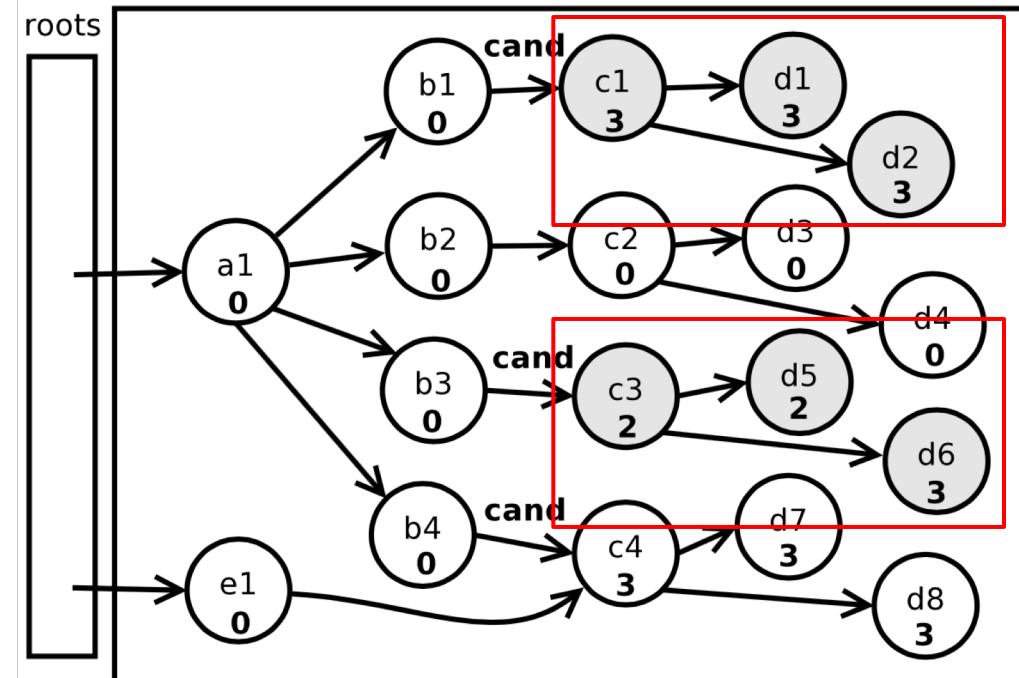
3.1: 各エッジについてターゲットのscとmsuを比較

- 3.2: 条件を満たしていれば候補参照認定
- 3.3: 候補参照から辿れるオブジェクトをマーク

4: ルートからオブジェクトをたどり、候補参照を介さず到達可能なオブジェクトのマークを外す

5: エッジ単位でマークされたオブジェクトの合計
サイズをbyteUsedに格納

- 各オブジェクトのサイズはすべて20と仮定

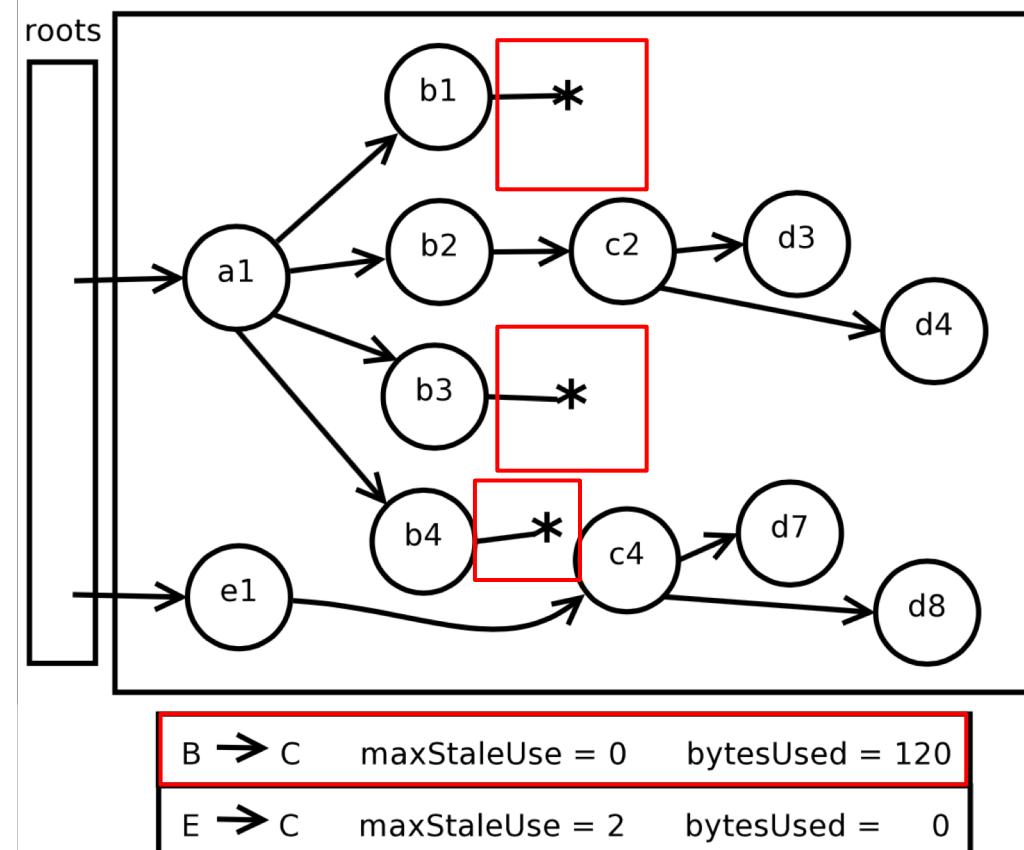


B → C	maxStaleUse = 0	bytesUsed = 120
E → C	maxStaleUse = 2	bytesUsed = 0

実装: 4-1

余談: リーク判定アルゴリズム: 実例

- 1: SELECT状態遷移時にmaxStaleUse(msu)が確定
- 2: SELECT状態での各staleCounter(sc)が確定
- 3.1: 各エッジについてターゲットのscとmsuを比較
 - 3.2: 条件を満たしていれば候補参照認定
 - 3.3: 候補参照から辿れるオブジェクトをマーク
- 4: ルートからオブジェクトをたどり、候補参照を介さず到達可能なオブジェクトのマークを外す
- 5: エッジ単位でマークされたオブジェクトの合計サイズをbyteUsedに格納
 - 各オブジェクトのサイズはすべて20と仮定
- 6: 最大のbyteUsedを持つエッジの候補参照をPOISONING(@PRUNE状態)



実験結果: 5-2

余談: コンパイルオーバーヘッドと 空間オーバーヘッド

コンパイルオーバーヘッド:

- リードバリアの計装のためにオーバーヘッドが増大する
- → 平均 +17% (時間)
- → しかし全体から見れば十分小さいオーバーヘッド

空間オーバーヘッド

- 新たに加わる要素は
 - エッジテーブル(ハッシュ)
 - 各エッジに4word(SourceClass, TargetClass, maxStaleUse, bytesUsed)
 - 今回の実験では最大数千種類のエッジが格納(Eclipse)