

# **Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling**

Trishul M. Chilimbi, Matthias Hauswirth

ASPLOS'04, October 9–13, 2004, Boston, MA, USA.

---

2017/04/18

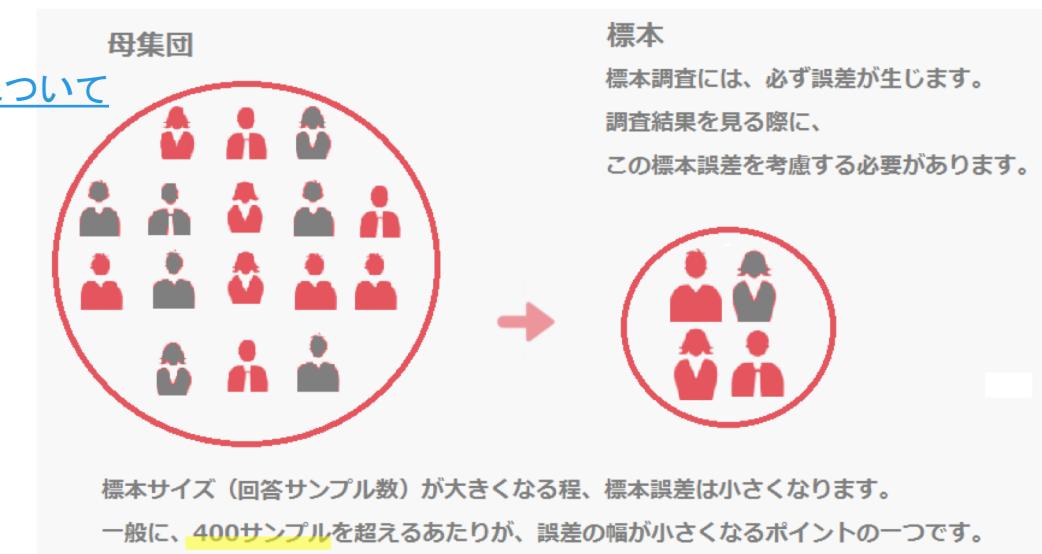
# 概要

---

- サンプリング(標本調査)をバグのチェックに応用したい
  - 低コスト化
    - →長時間稼働するプログラムにも適用可能
  - 局所的にバグを引き起こすコードがあると、正確性が下がる
    - →使用頻度に応じて、コードのサンプリングを調整
      - →使用頻度に反比例 → 使用頻度の低いコードはサンプリング頻度を上げる([ABT](#))
- [SWAT](#)というリーク検出ツールを実装、評価
  - メモリ割り当て/解放をトレース
  - 適応プロファイリングインストラクチャ(サンプリングの本体)を使用
    - 低オーバーヘッドでオブジェクトへのロードストアを監視
    - →採用技術:[ABT](#)(Adaptive Bursty Tracing)
  - 最終的に、staleオブジェクトをリークとみなして報告
    - 修正を行うわけではない
    - Stale: 最終アクセスから時間が経っている状態
  - 低オーバーヘッドなので(実行が5%未満、空間が10%未満)、長時間の運転にも対応可
  - 報告には、リークオブジェクトに最後にアクセスした番地を含めて、デバッグを容易に

# サンプリング(標本調査)

- ・全体ではなく一部を調査する手法
  - 長所: オーバーヘッドの削減
  - 短所: 正確性の欠如(標本誤差)
    - バグ検出では、とりわけ低頻度で実行されるコードにバグが集中していると対応が難しい
      - バグの9割が1%のコードに集中している場合など
  - イメージ出典:
    - <https://help.questant.jp/hc/ja/articles/203989368-必要回答サンプル数について>



# Bursty Tracing(BT)

- ・メソッド単位でサンプリングを行うCBS
  - CBS: counter-based-sampling
- ・元のコードに適用すると2つのコードを生成
  - checking code: サンプリングしないコード
  - instrumented code: 実際にサンプリングするコード
- ・どちらのコードに入るかの判定に以下の要素を追加
  - エントリチェック
  - ループバックエッジチェック
  - 全メソッドに追加される

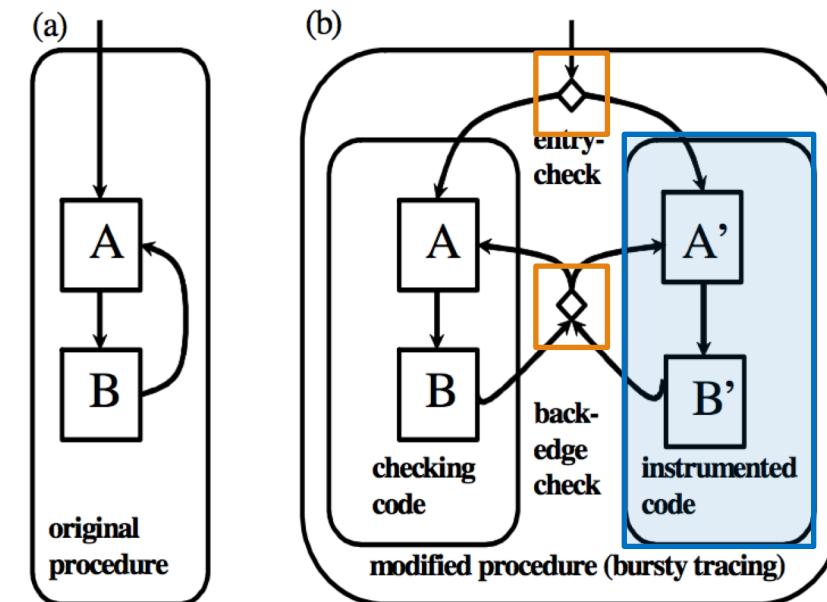


Figure 1. Instrumentation for low-overhead temporal profiling

# Bursty Tracing(BT)

- ・エントリ、ループバックで二つの変数を評価

- nCheck: ずっとばし期間
  - nCheck0: ユーザ定義
  - nInstr: バースト長、サンプリング期間
    - nInstr0: ユーザ定義

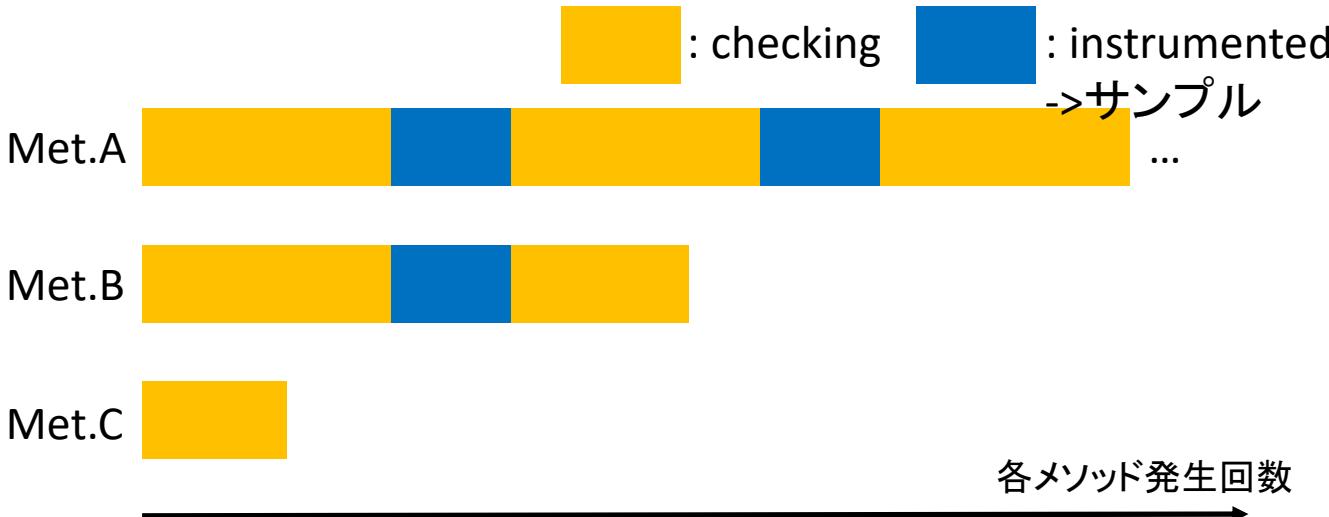
- 初期値
  - nCheck = nCheck0;
  - nInstr = 0;
- checking実行毎にnCheckをデクリメント
  - →0になつたらinstrumentedに移行
  - instrumented実行毎にnInstrをデクリメント
    - →0になつたらcheckingに移行

- サンプリングレート
  - $r = \frac{nInstr}{nCheck0 + nInstr0}$
  - ex) nCheck0 = 90, nInstr0 = 10ならレートは10%

メソッド単位→

繰り返し→

```
// fast path, executed on every check
nCheck--;
if(nCheck ≠ 0) goto targetchecking; // we were in checking code and stay there
// fall-through to remainder of check, executed infrequently
nCheck = 1;
if(nInstr == 0){
    nInstr = nInstr0;
    goto targetinstrumented; // transition from checking to instrumented code
}
nInstr--;
if(nInstr ≠ 0) goto targetinstrumented; // we were in instrumented code and stay there
nCheck = nCheck0;
goto targetchecking; // transition from instrumented to checking code
```



# Adaptive Bursty Tracing(ABT)

- BTの欠点: 低頻度コードのサンプリングが不確実

- 前図のMet.Cなど
- 使用頻度によってレートを変える(ABT)
  - 低頻度コードのサンプリングにも対応する
  - 次ページの漸化式

- 本研究(SWAT)の採用技術

- 最終的に、サンプリング結果を出力
  - さらに右図Foo.dll(カスタムDLL)を介してログとしてSWATに出力

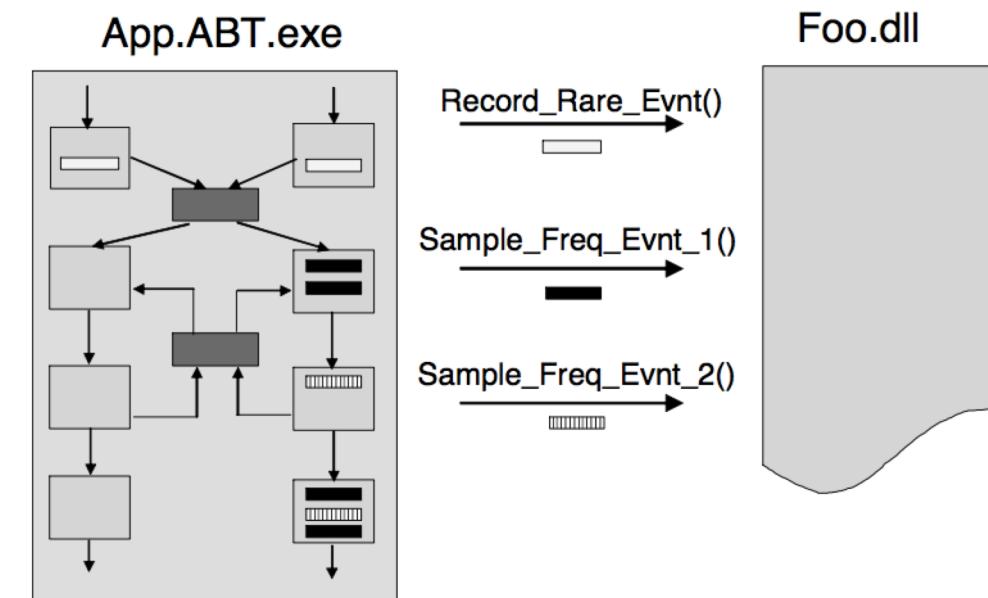


Figure 2. Adaptive Bursty Tracing Framework

# Adaptive Bursty Tracing(ABT)

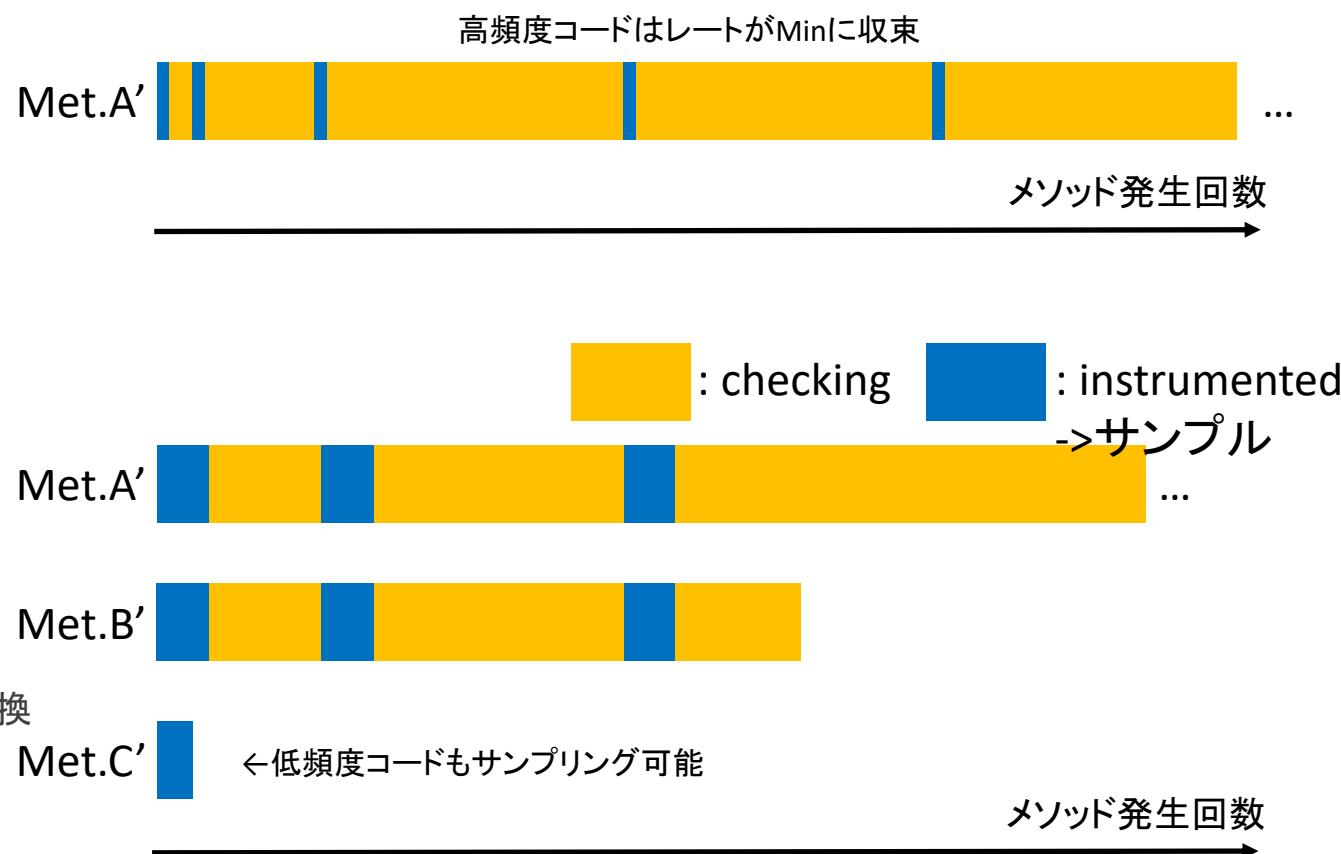
- nCheckを漸化式で変化

- $nCheck_0(n) = (Decr^{n-1} - 1) * nInstr_0$ 
  - instrumentedが実行される度にnが増えていく
  - Decr: ユーザ定義のレート減少率
  - Ex) Decr=10

$nCheck_0(1)=0$	$r=100\%$
$nCheck_0(2)=9*nInstr_0$	$r=10\%$
$nCheck_0(3)=99*nInstr_0$	$r=1\%$
$nCheck_0(4)=999*nInstr_0$	$r=0.1\%$

- Min: ユーザ定義のレート下限

- Ex) Decr=10, Min=0.001(0.1%)
  - $\rightarrow nCheck_0(5)=0.01\% < \text{Min}$  なので  $nCheck_0(5)=\text{Min}$  に置換



# サンプリング検出法の適正

---

- ・論文内でサンプリングに適したバグの種類を定義している
  - 以下の2つの条件を満たせば適している
    - バグBとイベントEが関連している前提
    - Soundness condition
      - $B \Rightarrow \neg E \wedge |False\ negative| = 0$
      - バグが生じれば、イベントは生じず、見逃しは発生しない
    - Preciseness condition
      - $\neg B \Rightarrow |E| > \frac{1}{SamplingRate} \wedge |False\ positive| = low$
      - バグが生じなければ、イベント数はレートの逆数を超え、誤検出は少ない
- ・メモリリークバグはこの2つの条件を満たしている(本当?)
  - 要するに、メモリリークはサンプリングによって見逃しなく、かつ誤検出を少なく検出できる(らしい)
  - 見逃しについては特に疑問

# 保守的と積極的

## 保守(PRECISE CONSERVATIVE)

- ・既存技術に多
- ・明確なリークのみ判定
- ・誤検出はないが、見逃しは発生

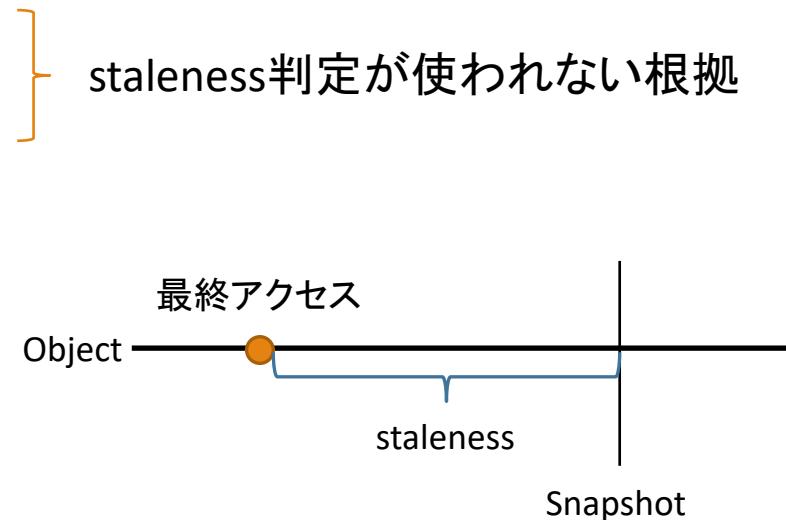
## 積極

- ・本提案手法(SWAT)
- ・リークか不明な部分も積極的に判定
- ・見逃しはないが、誤検出は発生
  - →見逃しについては疑問符



# staleness判定

- 長期間アクセスされていないオブジェクトはリークであるという直感
  - スナップショットの段階で
- staleness判定の欠点
  - ①: 全てのヒープデータアクセスの監視は高オーバーヘッド
  - ②: 誤検出が多くなる可能性
- 欠点への対処
  - ①: 適応プロファイリングの使用でオーバーヘッドの削減
    - → 実用に耐えられるレベルに
  - ②: 本手法の誤検出は顕著な差を出さない
    - → 誤検出 = 生きているオブジェクトへのアクセスを見逃す
      - → 一般にオブジェクトへのアクセスは複数 = サンプリングも複数
      - → 複数回のサンプリング全てでアクセスを見逃す確率は低い
      - → 誤検出の確率は低い
    - → 誤検出 = OSが意図的に解放しない
      - → バグではないが、メモリの使い方に改善の余地を示してくれているのでは？



# SWATの大まかな流れ

swatinstr.exe: 対象アプリにSWATの適用

- →app.swat.exeの出力

▪ app.swat.exe: ABTを用いて、サンプリング

- →割り当てログ、サンプルをswatruntime.dllに出力

▪ swatruntime.dll(swat.dll):

- ヒープモデル(後述)の更新

- 定期的なスナップショット作成 & リーク判定

- →ヒープモデルを読み込む

- →最終的にスナップショットは視覚化して出力

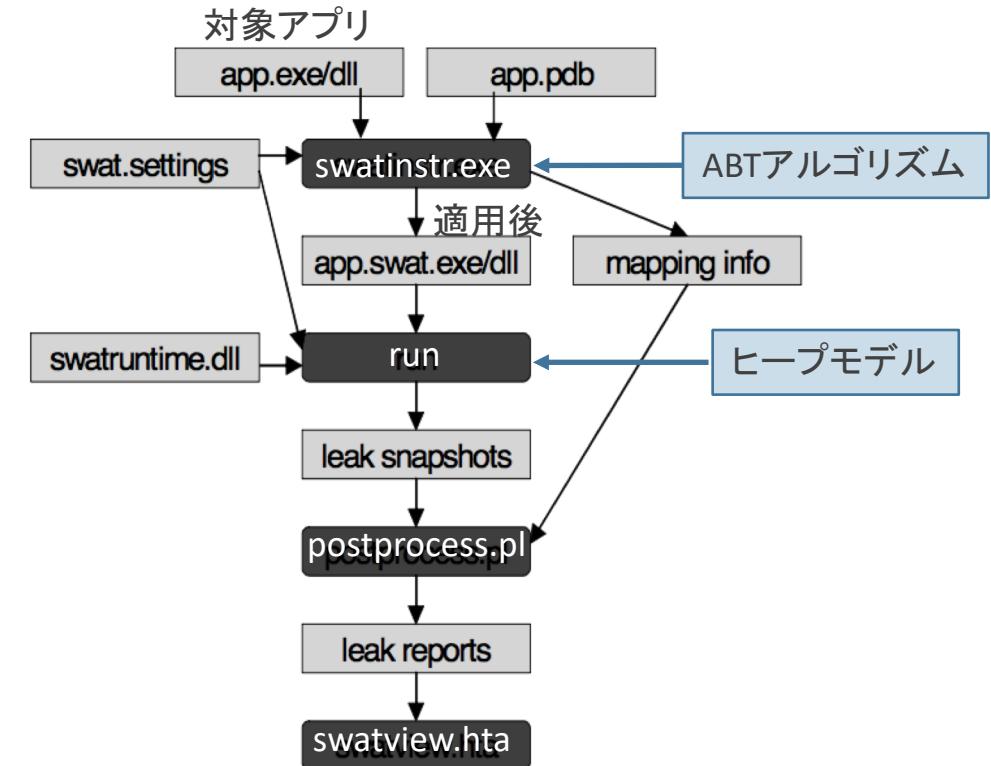


Figure 3. SWAT Infrastructure

# Heap Model

---

- ・入力: 割り当てログ(Record\_Alloc())、ヒープアクセスのサンプルセット(Sample\_Road/Store())
- ・割り当てられたオブジェクトにオブジェクト記述子(object descriptor)を付与
  - 内容:
    - オブジェクトの割り当てサイト →
    - オブジェクトへの最終アクセス時間 →
    - オブジェクトへの最終アクセスサイト →
- ・オブジェクト記述子の更新用の関数
  - *AllocateObject(ip, startAddress, size)*
  - *FreeObject(ip, startAddress)*
  - *FindObject(ip, address)*
  - *GetObjectIterator()*

# FindObject(ip, address)

---

- ・頻繁に呼ばれるため、オーバーヘッドの最小化が重要
- ・オブジェクト記述子はオブジェクトのヘッダにインライン実装される
  - →参照(ロード/ストア)を受け取った際に、オブジェクトの先頭(記述子)を如何に見つけるか
  - →各オブジェクトにアドレスツリー構造を保存(図5)
    - アドレスツリーからヘッダの位置情報が検索可能
    - →ロードストアアドレスと合致するアドレスツリーを見つけ、アドレスツリーから記述子の位置を得る
    - →アドレスツリーは大きな空間オーバーヘッドを発生
    - →特定のアライメントで葉と祖先ノードを破棄(図6)
  - →最も実行オーバーヘッドが小さいのは、ロードストアアドレスとヘッダアドレスが等しい場合
  - →最悪ツリー全体を走査する可能性
  - →そのアドレスツリーが頻繁にアクセスされていれば、キャッシュにいる可能性

# アドレスツリー図

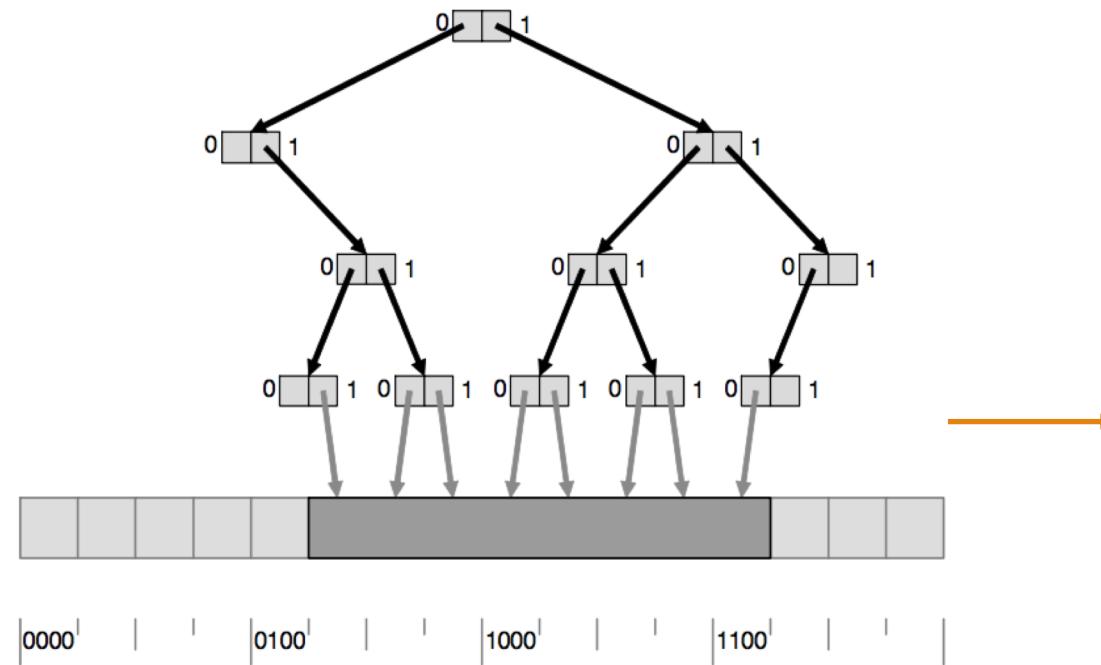


Figure 5. Part of complete address tree for object with start address 0101 and size 8.

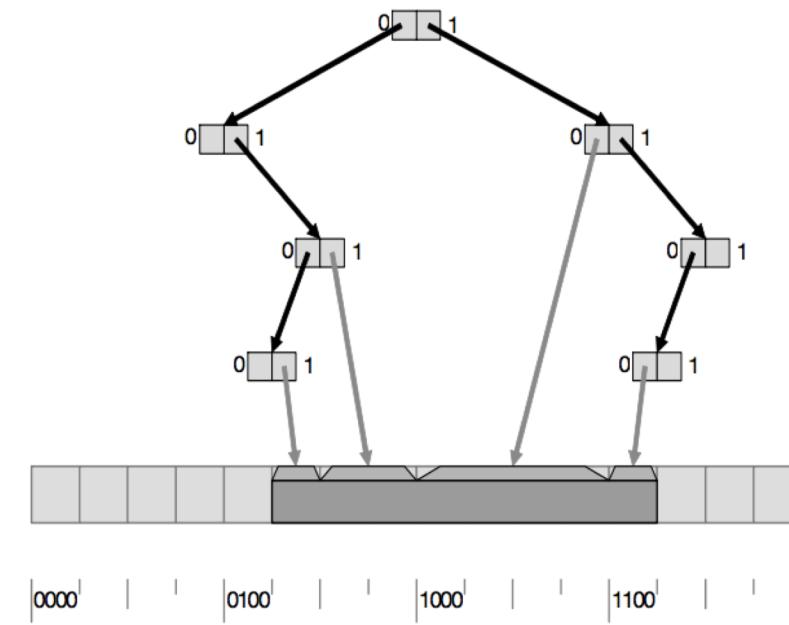


Figure 6. Part of an optimized address tree for an object with start address 0101 and size 8. All four unnecessary nodes are removed, and interior nodes directly reference the object descriptor.

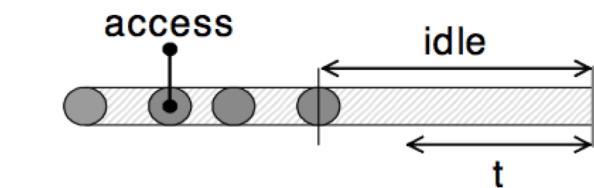
# Staleness predicates(staleness述部)

- ・割り当てられているオブジェクトがリークか判定
  - @スナップショット
  - 以下の述部のいずれかを判定に使用
    - Never Accessed
      - オブジェクト生成後、一度もアクセスされていない
    - Constant Time
      - 最終アクセスから一定時間経過している
    - Active Time
      - 初アクセスから最終アクセスまでの時間(active)の定数倍が経過している
- ・オブジェクトで適切な述部は違うというツッコミから発展
  - → Detecting Memory Leaks through Introspective Dynamic Behavior Modeling using Machine Learning
    - 機械学習で適切なstaleness述部を作る

## Never accessed



## Constant time (idle time > t)



## Active time

(idle time > n \* active time)

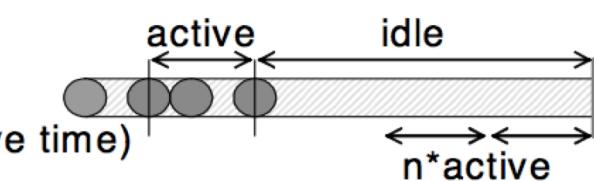


Figure 7. Staleness Predicates.

# 評価

---

- ・実験は4セット

- 対象: SPECInt2000
- 環境: OS: WindowsXP, RAM: 2GB, Proc: Pentium4(2.7GHz)
- ①: 実行オーバーヘッドの評価
- ②: 空間オーバーヘッドの評価
- ③: サンプリングの導入による誤差の評価
- ④: staleness述部の評価

- ・及びケーススタディ

- ①: 大規模対話型Webソフトウェアアプリ
- ②: マルチメディアアプリ
- ③: 自前のシミュレーションゲーム
- ④: 第三者の戦略ゲーム

# Experiment: Runtime Overhead

- ・前提: (Decr, Min, nInstr0)
  - (10, 0.1%, 10): 図の右グラフ
  - (10, 0.01%, 10): 図の左グラフ
  - →ABTのユーザ定義
- ・RuntimeOverheadの内訳:
  - アクセスの監視(ABT)
  - ヒープモデルの構築(HeapModel)
  - (リークレポートの生成、提出は含まれない)
    - →プログラム終了時に調整している
- ・結果: 5/8のベンチマークで5%未満を達成
  - &0.1%と0.01%で差がほぼない
    - →オーバーヘッドの大半はモデルの構築?

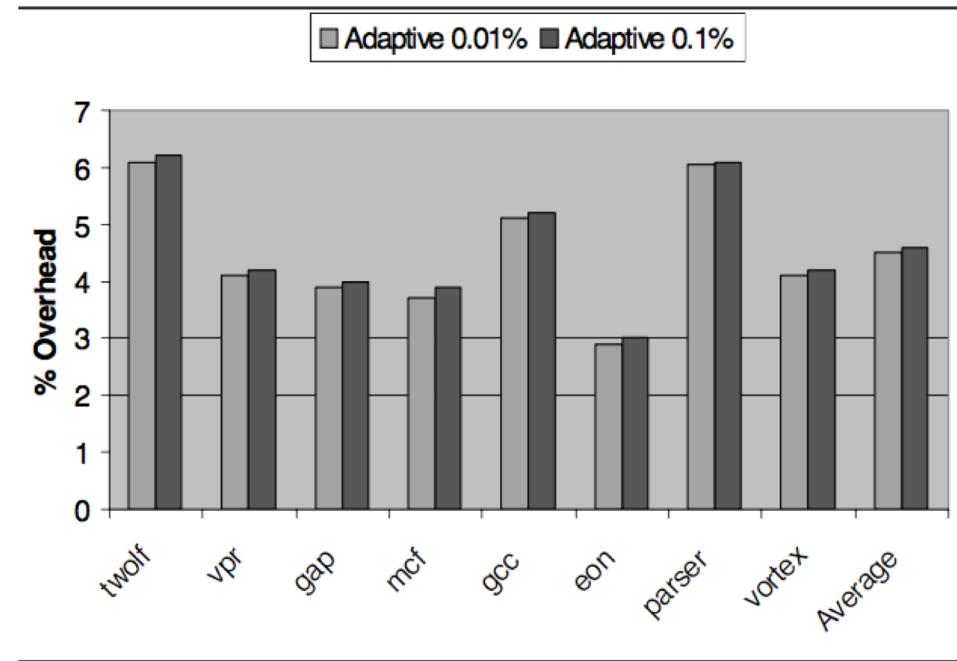


Figure 8. Runtime Overhead of SWAT.

# Experiment: Space Overhead

- SpaceOverheadの内訳:
  - アドレスツリー
  - ヒープに関連付けられたデータ
    - アドレスツリーを除く
    - 最終アクセスの識別用
- 結果: twolfを除き平均10%未満を達成
  - →twolfは小さなオブジェクトがめちゃ多い
    - →各オブジェクトにツリーやらが割り当てられるので高価に
  - 特にgap, mcf、gcc、parserでは0.1%未満を達成
- ABTの設定などが表記されていない不思議
  - →評価序文(4)では4setの評価と書いてあった
    - →次のベンチマーク項(4.1)ではSpaceOHを除く3setと記述
  - 何か怪しい

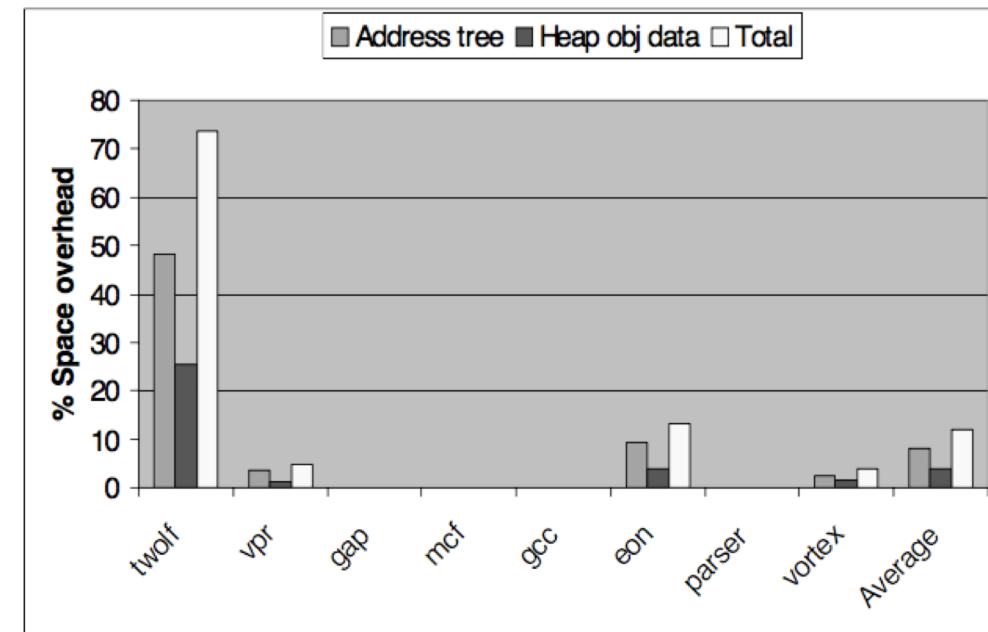


Figure 9. Space overhead of SWAT.

# Experiment: False positive introduced by sampling

- ・サンプリングによる誤検出の影響調査

- ・方法:

- ①: 動的解放をランダムに10%削除
- ②: ①をレート100でサンプリング
  - Staleness述部はIdleGt100Million
    - →最終アクセスから1億命令経過していればリーク
- ③: Minの値を変え、②と比較調査
  - 1%, 0.1%, 0.01%
- $FalseNegative = \frac{③ - ②}{②}$

- ・結果: 0.01%, 0.1%は上々、1%は誤検出多

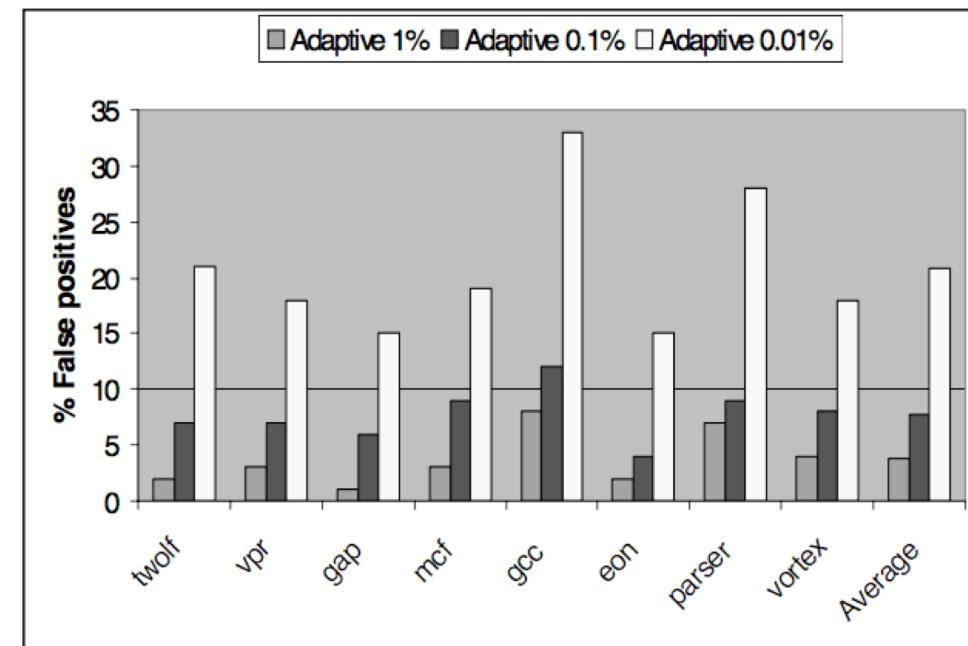


Figure 10. False positives introduced by sampling.

# Experiment: Staleness Predicates Evaluation

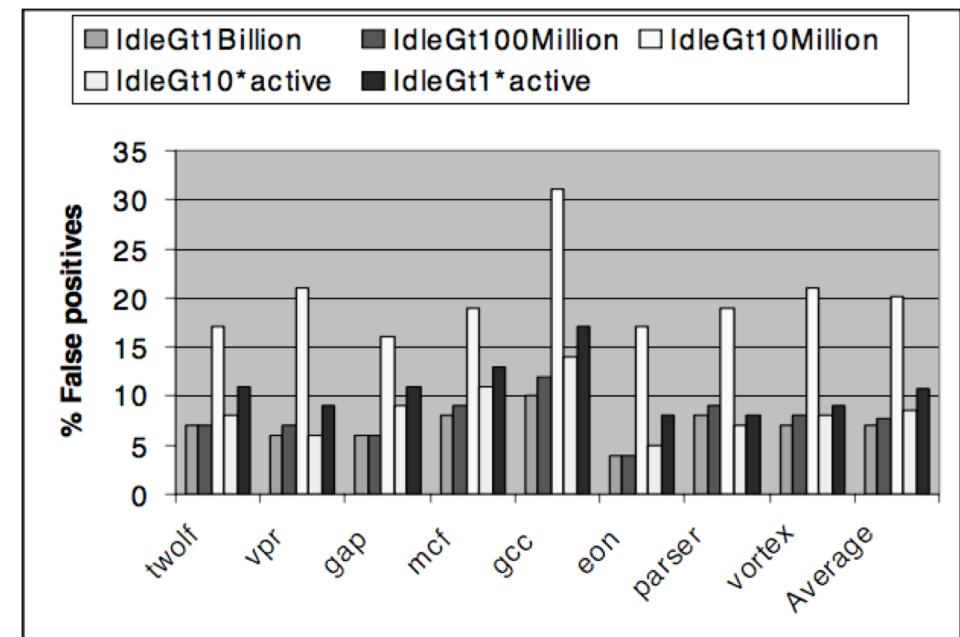
- staleness述部の評価
- 方法: サンプリングの誤検出調査と同様

## ▪ 対象:

- ConstantTimePredicates:
  - 1Billion: 最終アクセスから10億命令経過ならリーク
  - 100Million: "から1億命令"
  - 10Million: "から1000万命令"
- ActiveTimePredates:
  - 10\*active: 最終アクセスから10\*アクティブ時間経過ならリーク
  - 1\*active: 最終アクセスからアクティブ時間経過ならリーク

## ▪ 結果:

- 誤検出率:
  - $1\text{Billion} \leq 100\text{Million} \leq 10^*\text{active} < 1^*\text{active} < 10\text{Million}$  (parserを除く)
  - parserでは $10^*\text{active} < 1\text{Billion} < 100\text{Miillion}$
- ただし、1Billionでは短時間プログラムでリークを見逃す可能性



**Figure 11.** False positives introduced by various staleness predicates.

# Case Study

---

<b>Benchmark</b>	<b>% Runtime Overhead</b>		<b>% Space Overhead</b>		
	<b>Adaptive 0.01%</b>	<b>Adaptive 0.1%</b>	<b>Address tree</b>	<b>Heap object data</b>	<b>Total</b>
Interactive Web App	NA	NA	0.18	0.27	0.45
PC Game (Strategy)	4.3	4.71	8.2	2.21	10.41
Multimedia App	4.78	5.23	0.0	0.13	0.13
PC Game (Simulation)	4.42	4.97	3.25	0.78	4.03

**Table 1: SWAT overhead for real-world applications.**

# Case Study: Interactive Web App

Benchmark	%Runtime Overhead		%Space Overhead		
	Adaptive	0.01%	Address Tree	Heap Object data	Total
Interactive Web App	N/A	N/A	0.18	0.27	0.45

- ・恣意的にリークを注入(解放をスキップさせる)
  - 34個のリークを注入
  - 結果: 34+1個のリークを報告
    - はみ出た一個は誤検出かと思いきやガチリークでした
- ・SWAT適用後のアプリを日常的に使用
  - 都合上実行オバヘは観測できなかったが、体感では元と区別できないほどオバヘは低い
  - 空間オバヘも上図より十分低い
  - →6ヶ月使用して、20件のリークを発見(ただし誤検出かどうかは未検証)

# Case Study: 3<sup>rd</sup> Party PC Game(strategy)

Benchmark	%Runtime Overhead		%Space Overhead		
	Adaptive0.01%	Adaptive0.1%	Address Tree	Heap Object data	Total
PC Game(Strategy)	4.3	4.71	8.2	2.21	10.41

- 3rd Party: 第三者
- 出荷前に実験に協力してもらった
  - 適用バージョンは問題なく実行可能
    - →メモリ資源を多く使う近年のゲームでも、実行オバヘが許容範囲内(<5%)
- リークレポートは初回実行の2時間後に生成
  - →内容のほとんどは”誤検出”認定
    - →OSが再利用のために意図的に残したリークのため
    - 一方で、レポートから、ほとんどアクセスされていないキャッシュオブジェクト(≒メモリ管理の甘さ)などが読み取れた
    - →開発主任が修正に着手

# Case Study: Multimedia App

Benchmark	%Runtime Overhead		%Space Overhead		
	Adaptive0.01%	Adaptive0.1%	Address Tree	Heap Object data	Total
Multimedia App	4.78	5.23	0.0	0.13	0.13

- ・リリース前のベータ版に適用
- ・長時間の実行を伴うアプリへの実験
  - →SWAT適用バージョンと元のアプリの使用感は区別できないほど
  - →長時間の連續運転への適用にも堪える性能をSWATは保持している
- ・6件のガチリークを見つめ  
◦ →開発者が修正に着手

# Case Study: 1<sup>st</sup> Party PC Game(Simulation)

Benchmark	%Runtime Overhead		%Space Overhead		
	Adaptive0.01%	Adaptive0.1%	Address Tree	Heap Object data	Total
PC Game(Simulation)	4.42	4.97	3.25	0.78	4.03

- First Party: 製品提供者、ここではSWAT制作陣
- 約2時間分のゲームプレイのシナリオをキャプチャ&再生
  - → 実行オバヘは十分小さい(<5%)
  - 最終的に5件の潜在的リークを検出
    - → 手動でコードを走査した結果、これは誤検出ではないという結論に
- リークレポートがコード品質の検査にも役立つであろうという例

# Future Work & Conclusion

---

- Future work

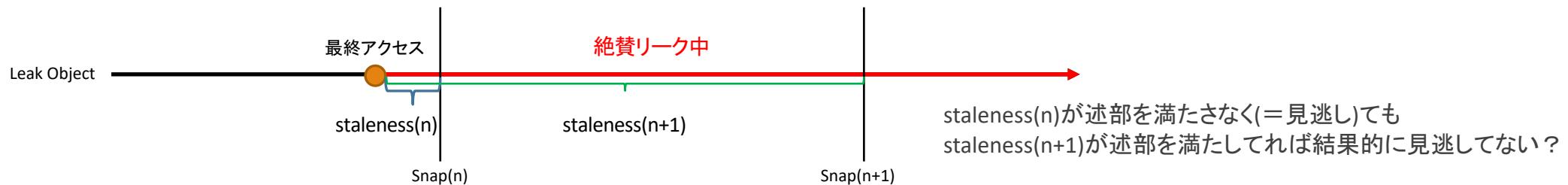
- リーク以外のバグの検出への応用
- アドレスツリーによるSpaceOverheadのさらなる削減
- 他のリーク検出技術との併用
  - ランダムサンプリング技術
  - issue++: リークオブジェクトを解放できる直近の場所を特定する
  - SWATはリークオブジェクトに最後にアクセスしたサイトを報告

- Conclusion

- オーバーヘッドを抑える適応プロファイリング技術をリーク検出に応用
- アルゴリズムを組み込んだSWATの実装＆実験
- 十分に低オーバーヘッド＆正確さを持っているのでは
  - マイクロソフト内のいくつかのグループで採用されていた

# 本論文へのツッコミ

- ・見逃し(False negative)が絶対に発生しないという根拠が曖昧
  - →n回目のスナップショットで見逃してもn+1回目のスナップショットで拾えるという発想？
  - Staleness述部の評価内で1Billionがリークを見逃す可能性があるとか言ってるし...
  - →もちろん1Billionが述部としては不適と言いたいだけかもしれないが



- ・各評価内でユーザ定義可能な要素が不明瞭
  - 例えばStaleness述部の評価内で、Minの値を0.1%にしたんだか1%にしたんだか分からん
    - →流れ的には0.1%だろうけど...
  - また、スナップショットのタイミングについて言及されていないのが不安
    - スナップショットのタイミング如何でstaleness判定に影響あるでしょ？