ハードウェアトランザクショナルメモリを用いたRuby仮想機械の大域インタープリタロックの除去

大平 怜
日本アイ・ビー・エム東京基礎研究所,

Jose G. Castanos
IBM Research – Watson Research Center

スクリプト言語の大域インタープリタロック

- スクリプト言語（Ruby、Pythonなど）の普及
  → 高まる性能要求

- シングルスレッド性能向上を目指す多くの既存研究
  - JITコンパイラ（Rubinius、ytljit、PyPy、Fioranoなど）
  - HPC Ruby
- マルチスレッド性能は大域インタープリタロックにより制約
  - 同時に1スレッドしかインタープリタを実行できない

☺ インタープリタとライブラリで並列プログラミングが不要

☹ マルチコア上でのスケーラビリティがゼロ

ハードウェアトランザクショナルメモリ(HTM)が市場に登場

- 大域ロックをTMに置き換えるだけで大きく性能向上（？）
- ソフトウェアTMより低いオーバーヘッドをハードで実現

IBM

Blue Gene/Q
2012

Sun Microsystems

Rock Processor
開発キャンセル

IBM

zEC12
2012

Intel

Transactional
Synchronization
eXtensions, 2013

本発表の目的

- GILをHTMで除去した場合の実アプリの性能は？
  - 大域インタープリタロック＝Global Interpreter Lock (GIL)
- そのために必要な変更点・新技術は？

本発表の目的

- GILをHTMで除去した場合の実アプリの性能は？
  - 大域インタープリタロック＝Global Interpreter Lock (GIL)
- そのために必要な変更点・新技術は？

✓ RubyのGILをzEC12のHTMを用いて除去
✓ Ruby NAS Parallel Benchmarksで評価

```
atomic { }
```

zEC12

関連研究

- PythonのGILをHTMで除去
  - ☹ 非サイクル精度なシミュレータでマクロベンチマークのみ [Rileyら,2006]
  - ☹ サイクル精度シミュレータだがマイクロベンチマークのみ [Blundellら, 2010]
  - ☹ Rockの制約の多いHTM上でマイクロベンチマークのみ [Tabba, 2010]
- Ruby, PythonのGILを細粒度ロックで除去
  - JRuby, IronRuby, Jython, IronPythonなど
  - ☹ 大きな実装の手間
  - ☹ クラスライブラリのマルチスレッド実行がオリジナルと非互換

✓ RubyのGILを・・・
✓ 実機のHTMを用いて除去し・・・
✓ マイクロベンチマークより大きなプログラムで評価
✓ 少ない実装の手間
✓ クラスライブラリの互換性の問題無し

本発表の流れ

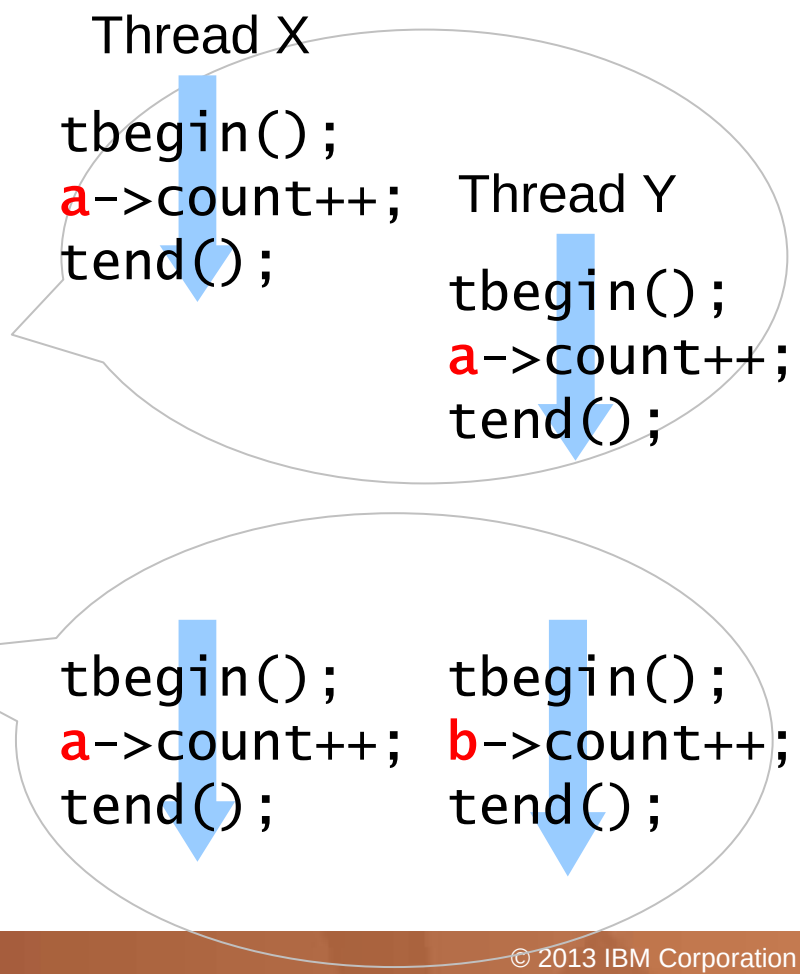
- 概要
- トランザクショナルメモリについて
- RubyのGILについて
- HTMによるGILの除去
- 実験結果
- 結論

トランザクショナルメモリ

```
lock();
a->count++;
unlock();
```

```
tbegin();
a->count++;
tend();
```

- プログラミング時
  - クリティカルセクションをトランザクション開始・終了命令で囲む
- 実行時
  - トランザクション中のメモリ操作は1ステップで行われたかのように他スレッドからは観測
  - 複数のトランザクションはメモリ操作が衝突しない限りは並列実行可
    →ロックよりも高い並列性



zEC12のHTM

```
TBEGIN
if (cc!=0)
  goto abort handler
...
...
TEND
```

- 命令セット
  - TBEGIN: トランザクションの開始
  - TEND: トランザクションの終了
  - その他、TABORT、NTSTGなど
- 実装
  - メモリ読み込み集合をL1とL2データキャッシュに保持 (~2MB)
  - メモリ書き込み集合をL1とストアバッファに保持 (8KB)
  - キャッシュコヒーレンスプロトコルを利用した衝突検知
- 以下の場合はアボートしてTBEGIN命令の直後に戻る
  - 読み込み集合の衝突、書き込み集合の衝突
  - 読み込み集合のサイズあふれ、書き込み集合のサイズあふれ
  - 禁止命令（システムコールなど; ほとんどのユーザ命令は許可）
  - その他、外部割込みなど

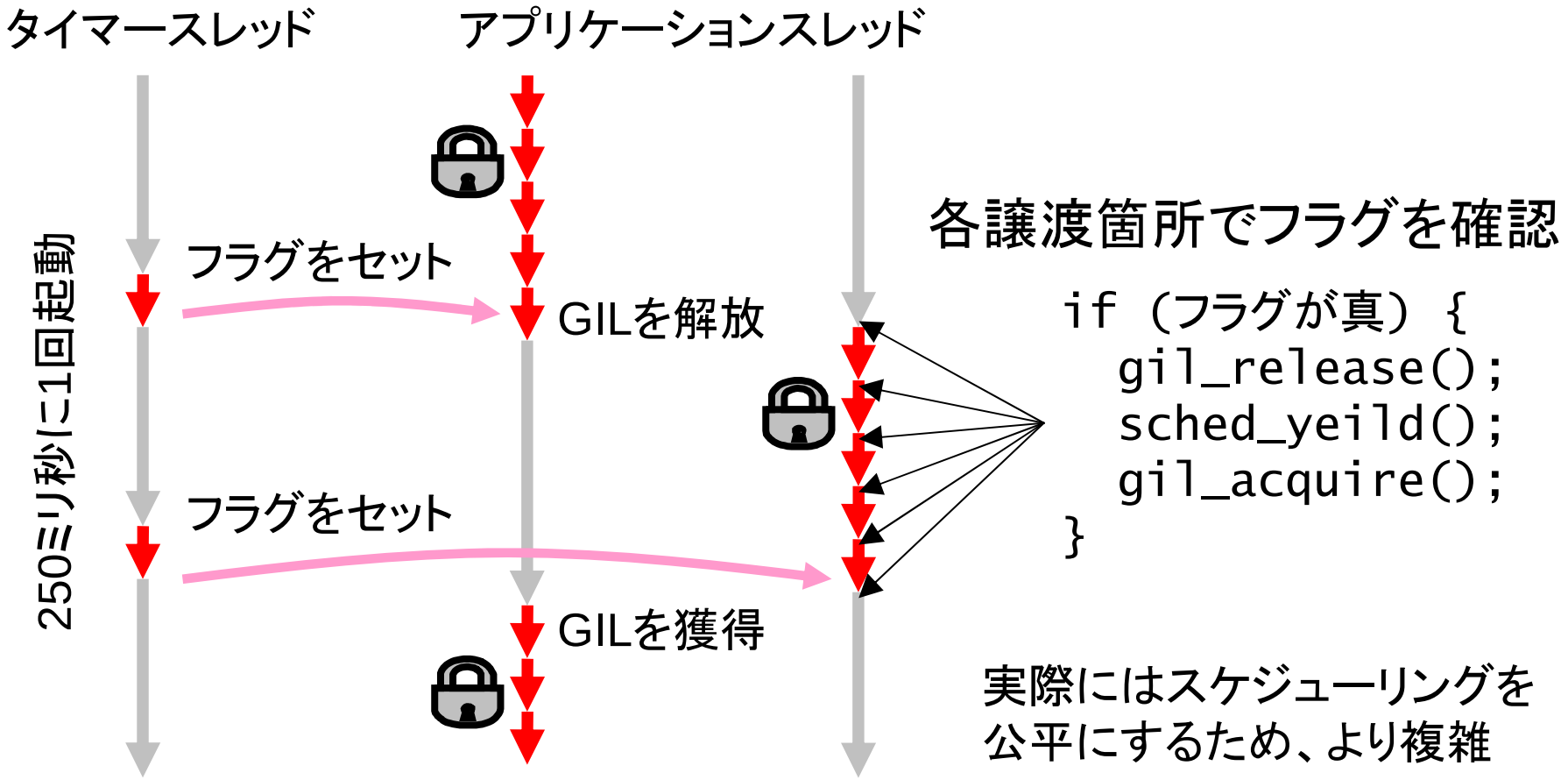
Rubyマルチスレッドプログラミングと GIL 1.9.3-p194準拠

- Ruby言語
  - Thread, Mutex, ConditionVariableクラスを用いてプログラム
- Ruby仮想機械
  - Rubyスレッドをネイティブスレッド(Pthread)に割り当て
  - しかしGILの存在により同時に1スレッドのみインタープリタ実行可

- 各スレッドは開始時にGILを獲得、終了時に解放
- ブロックする操作の間はGILを譲渡(I/Oなど)
- 特定の譲渡箇所バイトコード命令でもGILを譲渡
  - 無条件・条件ジャンプ、メソッド・ブロック出口

RubyのGIL譲渡の仕組み

- 譲渡箇所で常にGILを譲渡すると高オーバーヘッド
  → タイマースレッドを用いて250ミリ秒に1回譲渡

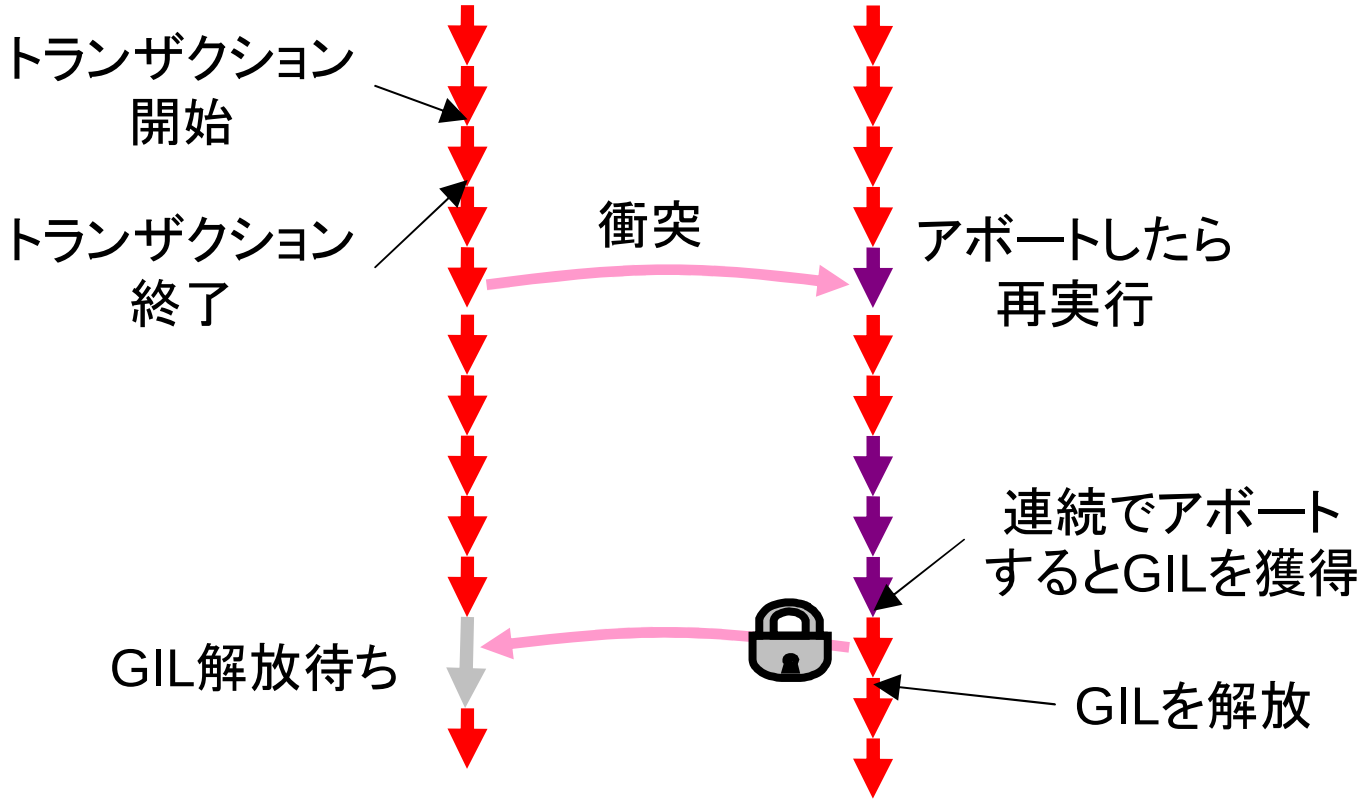


本発表の流れ

- 概要
- トランザクショナルメモリについて
- RubyのGILについて
- HTMによるGILの除去
  - 基本アルゴリズム
  - トランザクション長の動的な調節
  - 衝突の除去
- 実験結果
- 結論

HTMによるGIL除去の概要

- 最初はトランザクションとして実行
  - 開始・終了箇所はGILの獲得・解放・譲渡箇所と（基本）同じ
- トランザクションが何回もアボートしたらGILを獲得



トランザクションの開始

- 恒久的なアボート
  - サイズあふれ
  - 禁止命令
- それ以外は一時的なアボート
  - 読み込み・書き込み集合衝突など
- アボート理由はCPUから報告される
  - 特定メモリ領域を使用

```
if (TBEGIN()) {
  /* トランザクション */
  if (GIL.acquired)
    TABORT();
} else {
  /* アボート処理 */
  if (GIL.acquired) {
    if (16回再実行した)
      GILを獲得;
    else
      GIL解放を待って再実行;
  } else if (恒久的な理由でアボート) {
    GILを獲得;
  } else { /* 一時的な理由でアボート */
    if (3回再実行した)
      GILを獲得;
    else
      再実行;
}}
Rubyコードを実行;
```

トランザクションの開始・終了箇所

- GILを獲得・解放・譲渡する箇所と同じとすべき
  - クリティカルセクション境界として安全が保障されているから

☹ ただし元の譲渡箇所は疎粒度過ぎてサイズあふれなどが頻発

- バイトコード境界は安全なクリティカルセクション境界（なはず）
  - バイトコードは任意の順番で生成されうる
  - よって、バイトコード境界をまたがるインタープリタ内部のクリティカルセクションは考えづらい
  - ☹ 正しく同期されていないRubyプログラムは挙動が変わる可能性

→ 以下のバイトコード命令の直前をトランザクション境界として追加

- `getinstancevariable, getclassvariable, getlocal, send, opt_plus, opt_minus, opt_mult, opt_aref`
- 基準：頻繁に現れる、もしくは処理が複雑

トランザクションの終了と譲渡箇所の処理

トランザクションの終了

```
if (GIL.acquired)
  GILを解放;
else
  TEND();
```

譲渡箇所(トランザクション境界)の処理

```
if (--yield_counter == 0) {
  トランザクションを終了;
  トランザクションを開始;
}
```

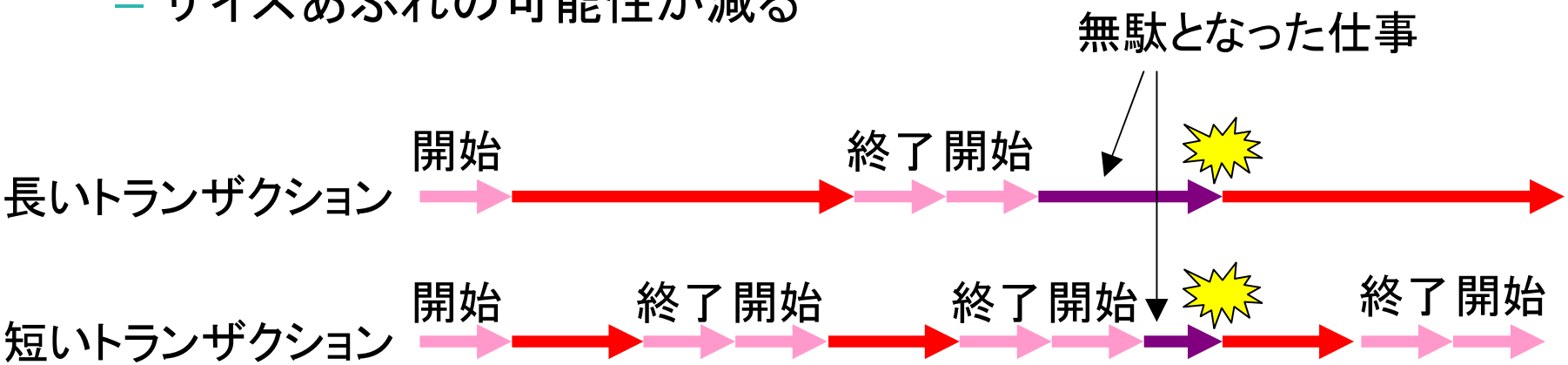
トランザクション長の動的な調節(後述)

✓ ソースコードの変更点は一部のファイルに限定
←→ 細粒度ロックは変更点が全体に散らばる

トランザクション長のトレードオフ

- 譲渡箇所で常にトランザクションを終了開始する必要はない
  ＝ トランザクション長は譲渡箇所の粒度で可変

- 長いほうが開始終了の相対オーバーヘッドが小さい
- 短いほうがアボートのオーバーヘッドが小さい
  - アボート時に無駄となって巻き戻される仕事量が減る
  - サイズあふれの可能性が減る



トランザクション長の動的な調節

譲渡箇所毎に、そこから開始するトランザクション長を調節

1. 初期長として大きな長さ(255)を割り当てる
2. 譲渡箇所毎に以下の2つを数えてアボート率を算出:
   - そこから開始したトランザクションの数
   - そこから開始したトランザクションがアボートした数
3. アボート率が閾値(1%)を超えたらトランザクション長を縮めて(x 0.75)、2に戻る
4. アボート率が閾値を超える前に一定数(300)トランザクションを開始したら、その譲渡箇所は調節終了

主な5つの衝突箇所と衝突の除去 (1/2)

✓ 性能を上げるためにはソースコードを全く無変更というわけにはいかない。が、変更はそれぞれ高々数十行

- 現在実行中のスレッドを指す大域変数
  - 譲渡箇所毎に書き込まれるので激しく衝突
  - ✓Pthreadのスレッドローカル領域に移動
- インラインキャッシュミス時の更新処理
  - メソッド呼び出し時とインスタンス変数アクセス時にハッシュ表を引いた結果をキャッシュ
  - 全スレッドで共有なので更新すると他スレッドと衝突
  - ✓インラインキャッシュミスを減らす手法を実装

主な5つの衝突箇所と衝突の除去 (2/2)

- オブジェクト割り付け時の大域フリーリスト操作
  - ✓スレッドローカルフリーリストを導入
    - 空になったら大域フリーリストから256個取ってくる
- ゴミ集め
  - 複数スレッドがゴミ集めしようとすると衝突
  - 単一スレッドがゴミ集めしてもトランザクションサイズ溢れ
  - ✓Rubyヒープの初期サイズを400MBに広げて頻度を低下
    - ヒープが縮小しないように改造
- スレッド構造体(rb_thread_t)のfalse sharing
  - 色々とスレッドローカル情報を格納するように改造
    → 同じキャッシュラインにある他のデータと衝突
  - ✓スレッド構造体をキャッシュラインにアラインして割り付ける

環境固有の変更点

- Pthread関数でGIL獲得待ちする前にスピンウェイトを挿入
  - オリジナルのGILは低頻度でしか獲得・解放されない
  - HTMの代替手段としてのGILは遥かに高頻度で獲得・解放

  → 毎回OSで寝ていると並列度が著しく下がる
  - 環境によっては同様の最適化をPthread実装が既に含む
- 禁止命令を含まない独自実装の`setjmp()`を使用
  - z/OSの`setjmp()`はアドレス空間操作命令を含む
  - Rubyでの`setjmp()`の用途には汎用レジスタの保存で十分

本発表の流れ

- 概要
- トランザクショナルメモリについて
- RubyのGILについて
- HTMによるGILの除去
- 実験結果
- 結論

実験環境・比較対象

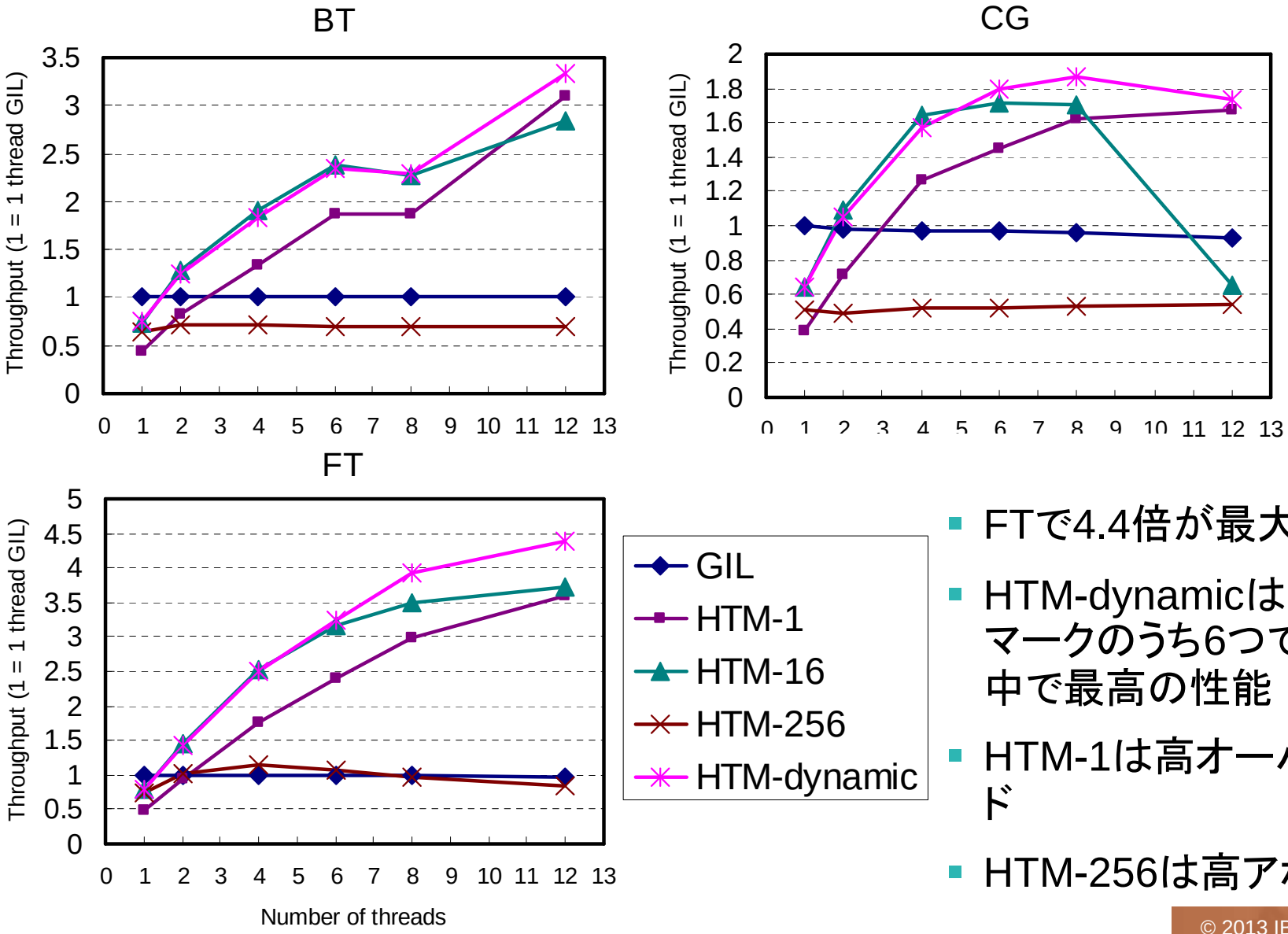
- Ruby 1.9.3-p194へ実装
- z/OS 1.13のUNIX System Servicesへ移植
  - EBCDICのためにフル機能のRubyは未だビルドできず
  - コアのクラスライブラリのみサポートするminirubyで実験
- 8コア5.5 GHzのzEC12で実験
  - 1コア1ハードウェアスレッド
- 比較対象
  - GIL：オリジナルのRuby
  - HTM-*n* (*n* = 1, 16, 256)：
    固定トランザクション長（*n-1*個の譲渡箇所をスキップする）
  - HTM-dynamic：動的なトランザクション長調節

ベンチマークプログラム

- マイクロベンチマーク2つ
  - Whileとイテレータで値を加算する完全並列プログラム
  - HTMは1スレッドGILに対して12スレッドで11倍の性能向上
  - シングルスレッドのオーバーヘッドは最低5-14%（後述）
- Ruby NAS Parallel Benchmarks (NPB) [野瀬ら,2012]
  - 全部で7つのプログラム
  - シングルスレッド区間とマルチスレッド区間が混在

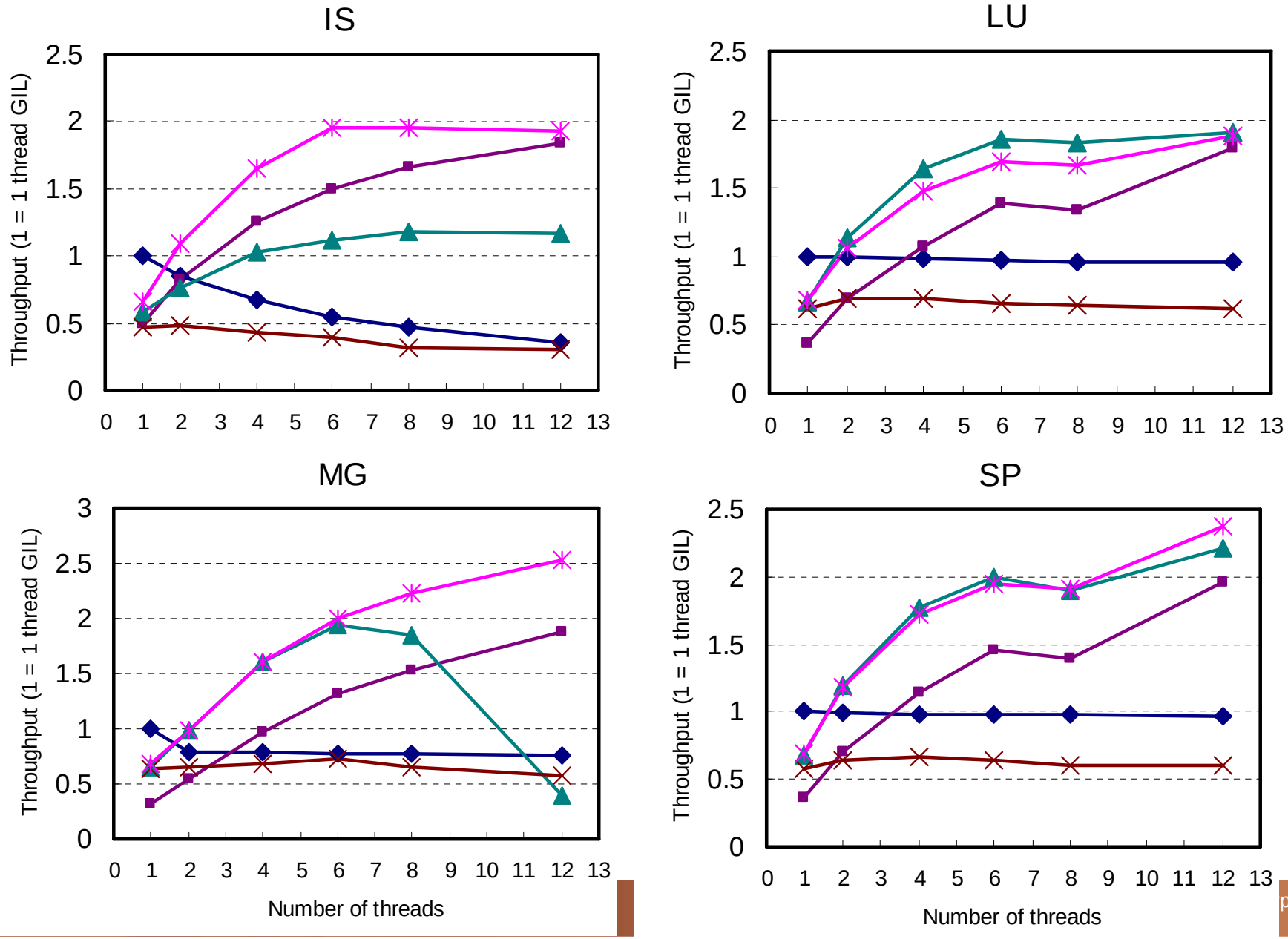
  → GILを完全除去しても完全なスケーラビリティは示さない
- Webベースのワークロードは計測中

Ruby NPBのスループット (1/2)



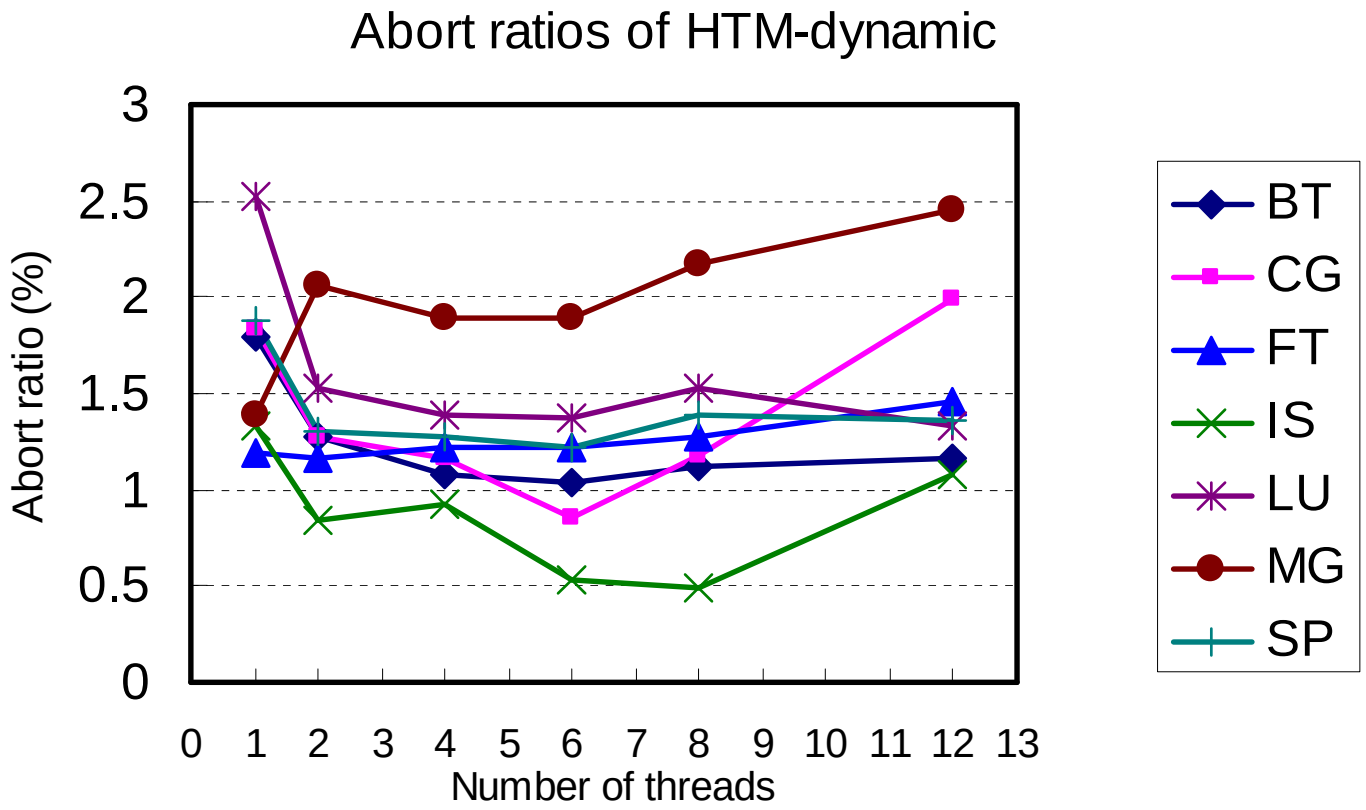
- FTで4.4倍が最大
- HTM-dynamicは7ベンチマークのうち6つでHTMの中で最高の性能
- HTM-1は高オーバーヘッド
- HTM-256は高アボート率

Ruby NPBのスループット (2/2)



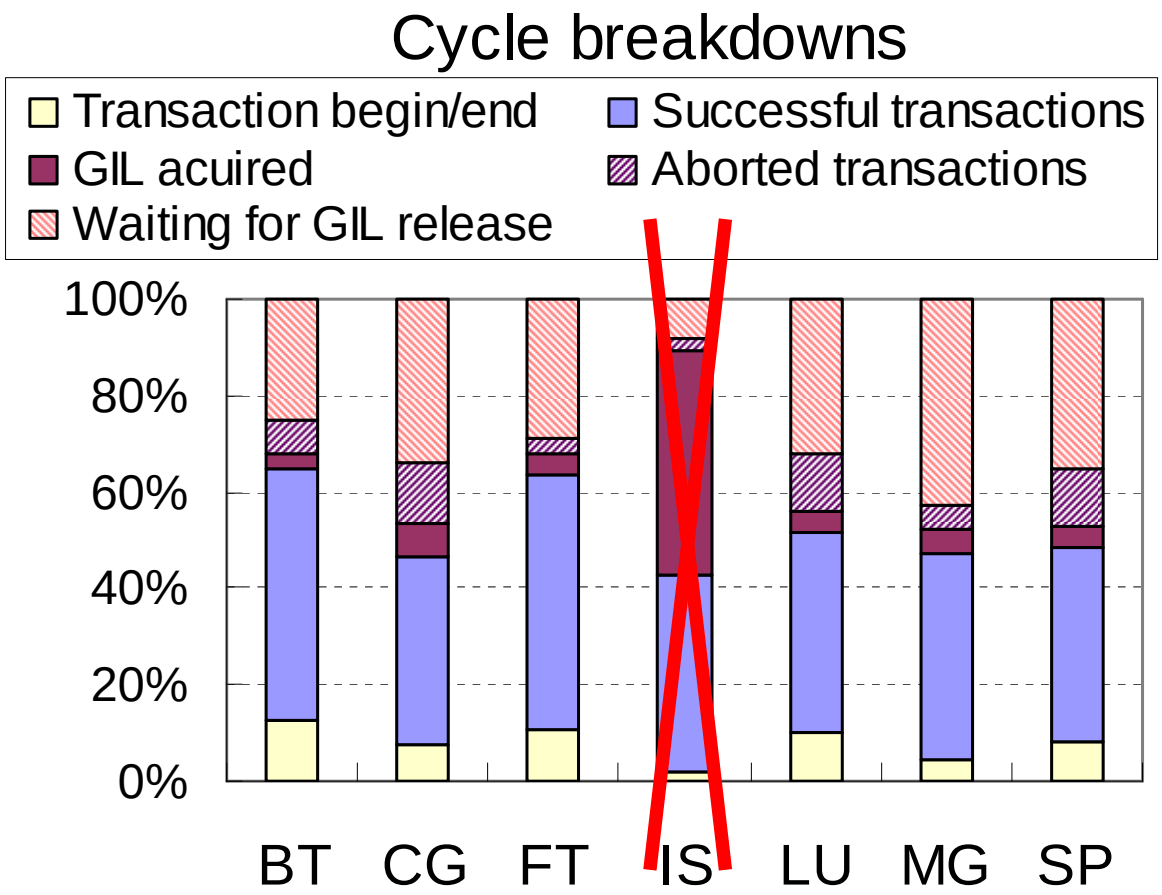
アボート率

- HTM-dynamicがトランザクション長をうまく調節することで目標値の1%程度に抑えている
- スケーラビリティとの相関は見られない



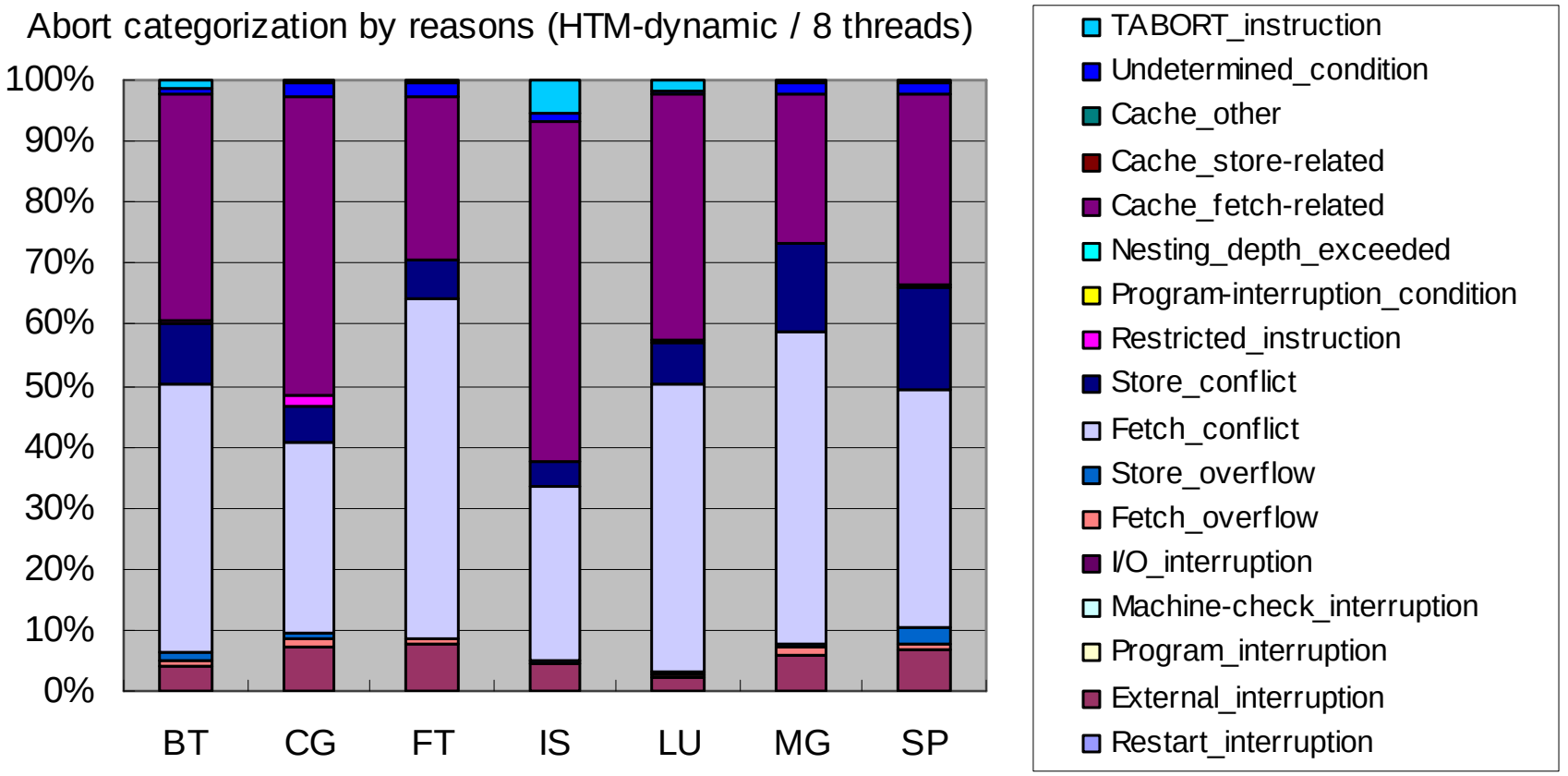
CPUサイクルの分類

- スケーラビリティとの相関は見られない
  - ISは実行時間の79%を計測区間外の初期化処理が占めているので、CPUサイクルの分類は不正確



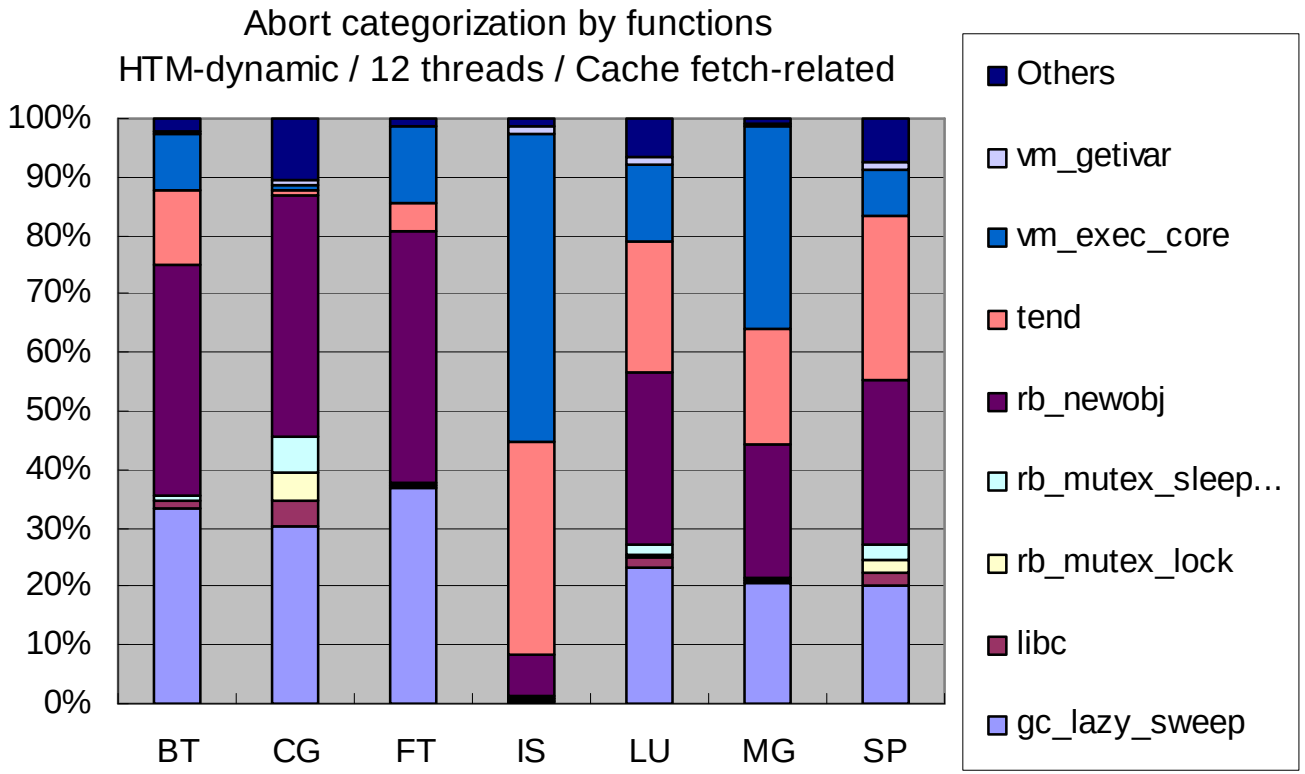
アボート理由による分類

- 読み込み集合の衝突がアボート理由の大半を占める
  - Cache fetch-related + Fetch conflict

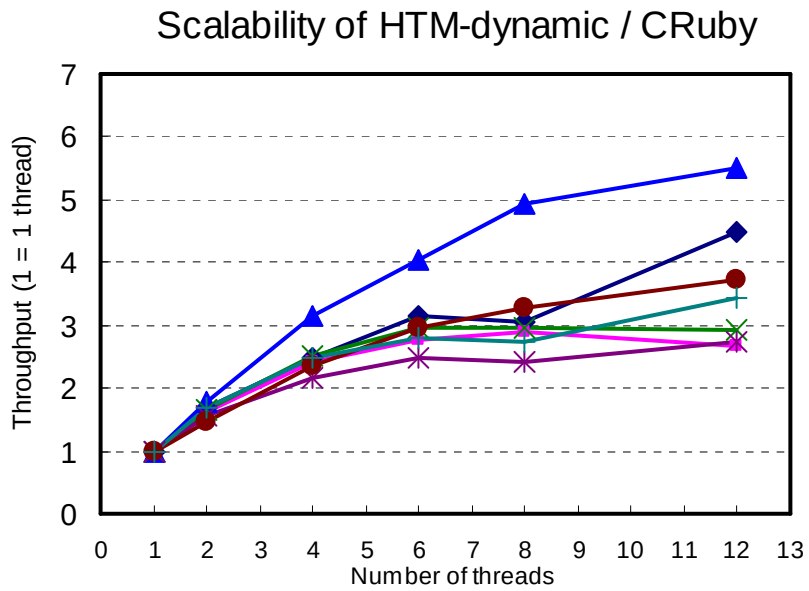


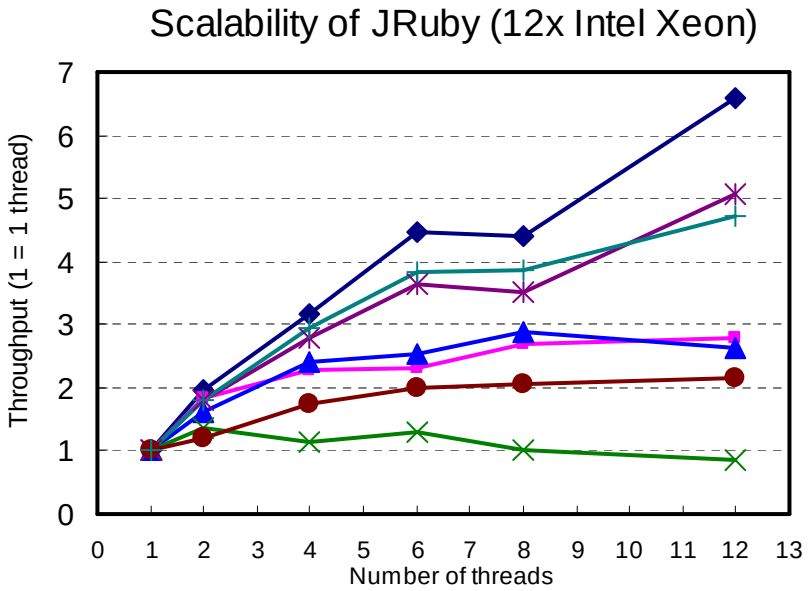
読み込み集合の衝突を起こした関数による分類

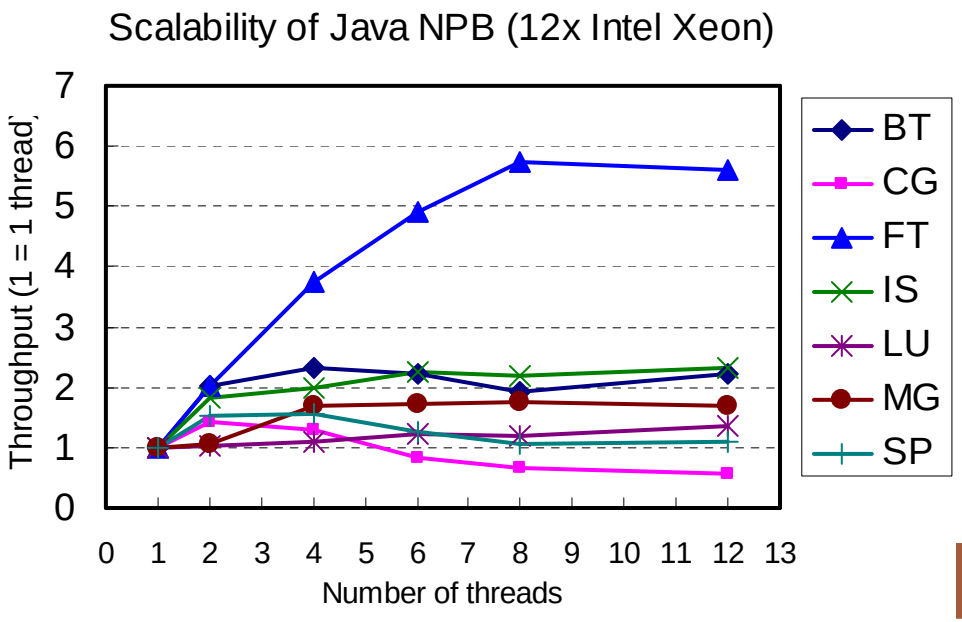
- 大域フリーリストの操作(rb_newobj)と遅延スイープ(gc_lazy_sweep)が衝突の半分以上を起こしている
  - 大部分がFloatオブジェクトなので2.0のFlonumで解決？



CRuby、JRubyとJavaのスケーラビリティ比較







- CRuby (HTMを使用) はJava (Java VM内部のボトルネックはほぼ無し)に類似

→ CRubyにおけるスケーラビリティの頭打ちは各プログラムに固有のボトルネック

- CRubyとJRuby (細粒度ロックを使用)は異なるスケーラビリティの特徴を示す
- 平均ではCRubyとJRubyは同じスケーラビリティを達成

シングルスレッドオーバーヘッド

- スクリプト言語のシングルスレッド性能は重要

- 1スレッドしかない場合、HTMではなくGILを用いる最適化

→ マイクロベンチマークでなお5-14%のオーバーヘッド

- オーバーヘッドの原因:
  - 譲渡箇所でのチェックと新たな譲渡箇所の追加
  - Pthreadのスレッドローカル領域へのアクセスの遅さ（z/OS固有）

結論

- GILをHTMで除去した場合の実アプリの性能は？
  - 12スレッドで最大4.4倍
- そのために必要な変更点･新技術は？
  - HTMを用いるために必要な変更は一部ファイルに限定
  - 衝突を除去するための変更はそれぞれ高々数十行
  - 動的にトランザクション長を調節する仕組みを提案

✓ 少ないソースコード変更でGILを上回る性能を
少なくとも一部のプログラムで達成する有用な手法