*Ruby*用仮想マシン*YARV*における並列実行スレッドの実装

笹田耕一（発表者）
（東京大学大学院情報理工学系研究科）
まつもとゆきひろ（NaCl）
前田敦司（筑波大）
並木美太郎（農工大）

情報処理学会プログラミング研究会 2006 10/13

Agenda

- 背景と問題点
- 並列実行スレッドの検討
  - マッピングモデル
  - 並列実行のための排他制御
- YARVにおける並列実行スレッドの設計
  - Rubyスレッドのネイティブスレッド対応
  - Rubyスレッドの並列実行
- 実装した並列実行スレッドの評価
- まとめ

背景

- スクリプト言語Ruby
  - 使いやすい / 気軽に使えるオブジェクト指向言語として世界中で広く利用
  - スレッドにも当然対応
    - Rubyレベルで気軽に生成できるスレッド（**Ruby**スレッド/**RT**）を並行実行
- 並列実行計算機のコモディティ化
  - 「普通のパソコン」にデュアルコアCPU
  - 数年後にはマルチコア（>2）が「普通」に
  - 並列計算機を活用できる**S/W**環境が必要

```
Thread.new{
  # … Thread 2
}
# … Thread 1
```

問題点

- Rubyスレッドの並列実行は不可能
  - ユーザレベルスレッド
  - 現状はあくまで「並行」実行支援（v.s. Java）
- ユーザレベルスレッドの非効率な実装
  - 移植性を向上するためスレッド切り替えは「setjmp/longjmp」＋「マシンスタックコピー」
    ブロッキングするOSコールの利用不可
  - 例えば I/O に関してはselectでポーリング

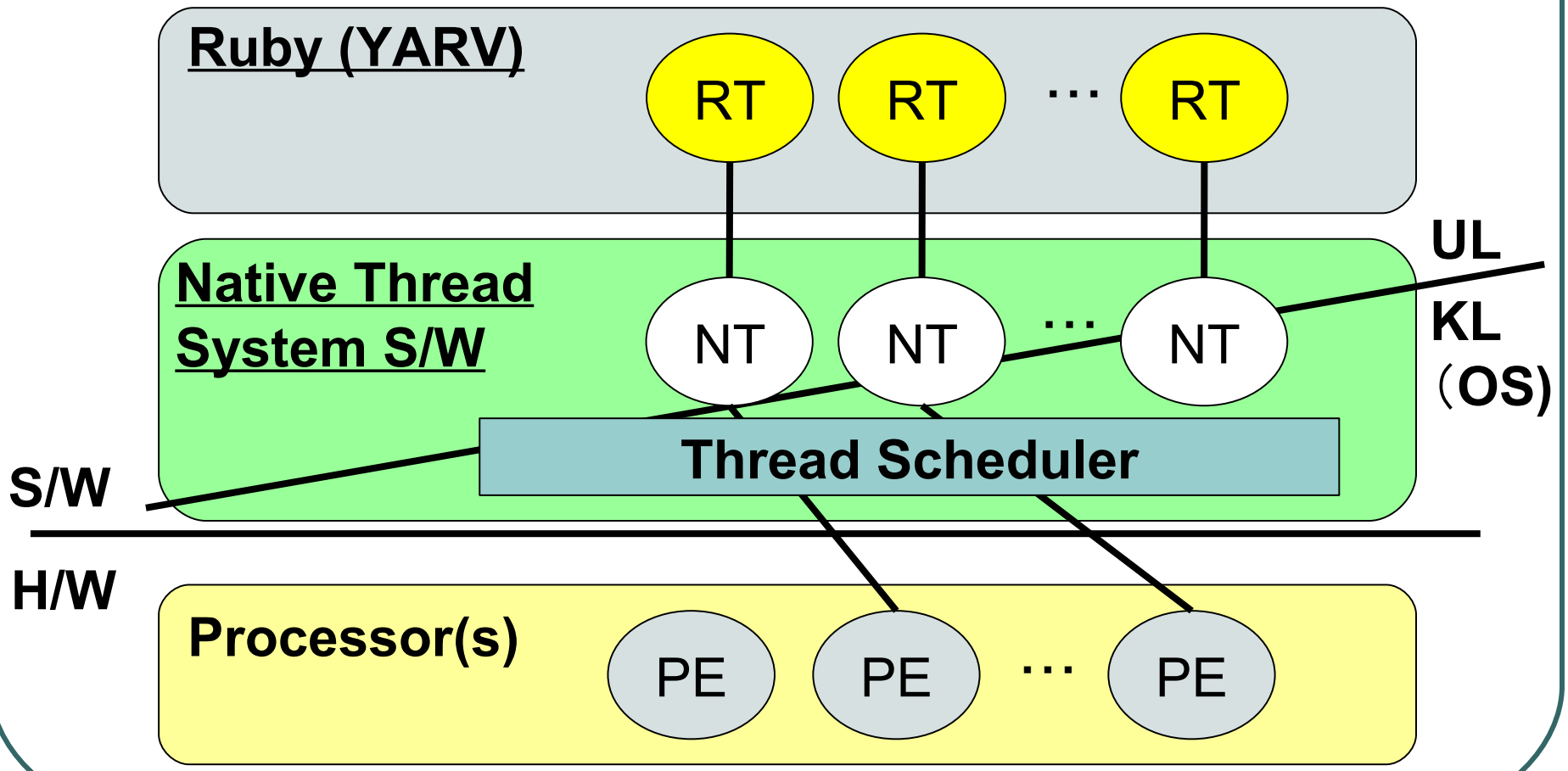
解決の方針

- ネイティブスレッド処理機構の利用
  - 本機構で管理するスレッド：ネイティブスレッド/NT
- （大体）移植性の確保
  - UNIX系：POSIX Thread（Pthread）
  - Windows系：MS-Windows Win32API
- システム依存の効率的なスレッド管理
- 並列計算機上で並列実行可能
- ブロッキングするようなOSコールも利用可能

検討:
ネイティブスレッド利用

- Rubyスレッドとネイティブスレッドのマッピングモデル(1:1, M:N)の検討
  - 1:1→スレッド管理を全てNTで
  - M:N →切り替えなどをユーザレベルで
  - (1:N → ユーザレベルスレッド)
- シンプルにするために 1:1 モデルで
  - ネイティブスレッド処理機構にスレッド制御の性能を担保(現在も研究されている分野なので、まだまだ性能向上は見込める)

システムの全体像
Rubyスレッドとネイティブスレッド



PE: Processor Element, UL: User Level, KL: Kernel Level

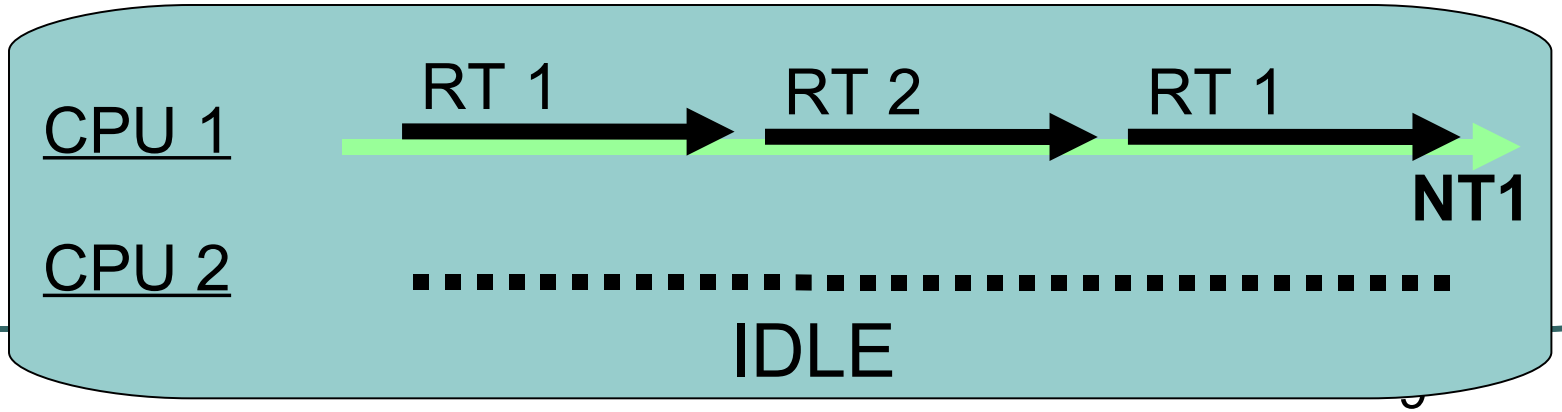
検討:
並列実行における排他制御

- 並列実行と排他制御
  - 排他制御は必要
  - 排他制御はオーバヘッド
- いくつか考えられる方針
  - (a) 細粒度ロック(FGL)のみ
  - (b) 細粒度ロック＋ジャイアントロック(GL)
  - (c) ジャイアントロックを持つRTのみ実行
  - (d) 排他制御しない(Rubyプログラマ任せ)

```
str = "foo"
Thread.new{
  N.times{str << "bar"}
}
N.times{str << "baz"}
```

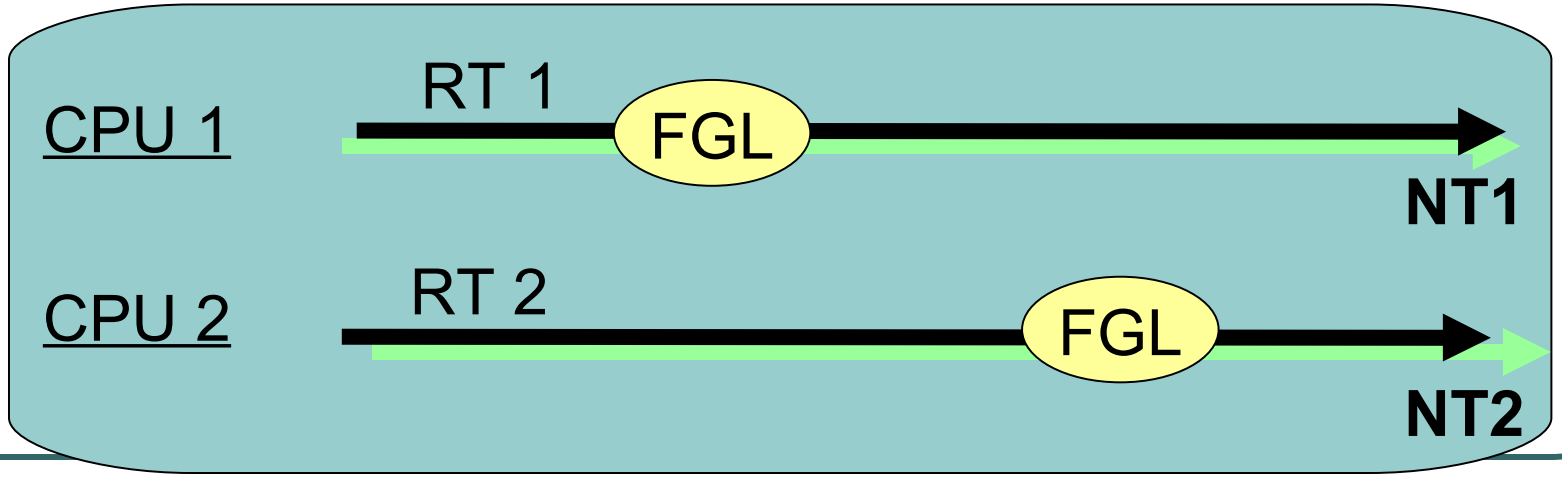
現在のスレッド

- 並列実行が不可能
- スレッド切り替えのタイミングをコントロールできるため、排他制御が不要
  - そのような前提で書かれた膨大なCで記述されたメソッド（Cメソッド）が存在



検討：並列実行における排他制御
(a) 細粒度ロックのみ

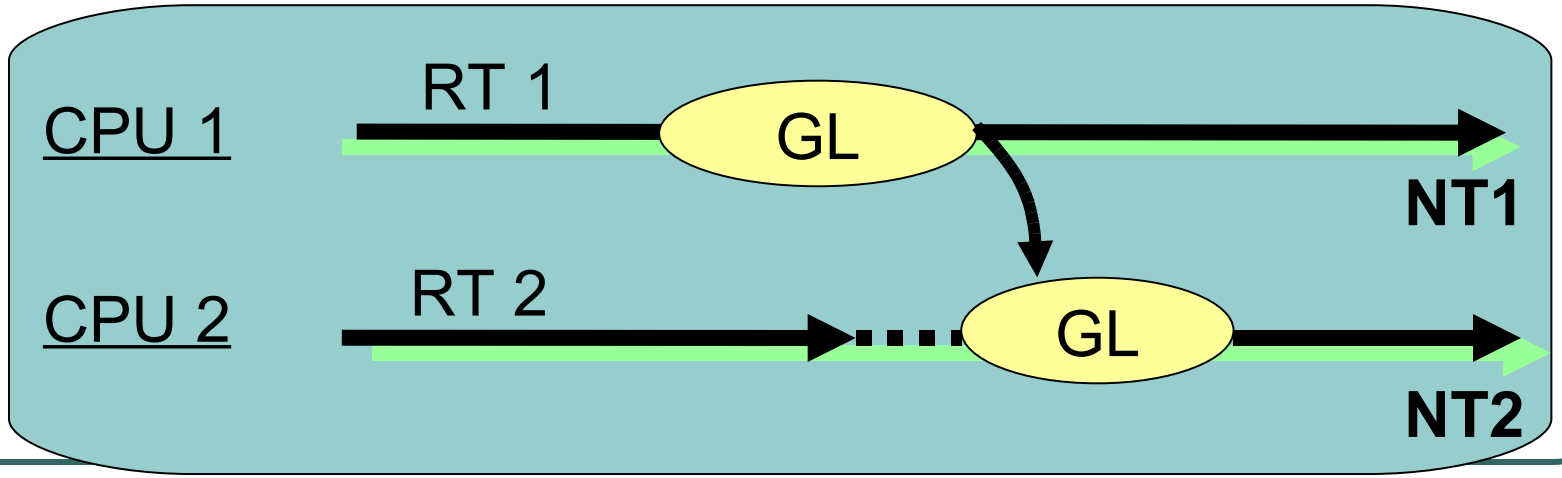
- ○ 十分な並列性を発揮
- × 全ての既存のコードに対し、FGL等によるスレッドセーフ（TS）コードへ変更が必須
  →「豊富なRubyライブラリ」が利用不可に



FGL: Fine-Grained Lock

検討：並列実行における排他制御
(b) 細粒度ロック＋ジャイアントロック

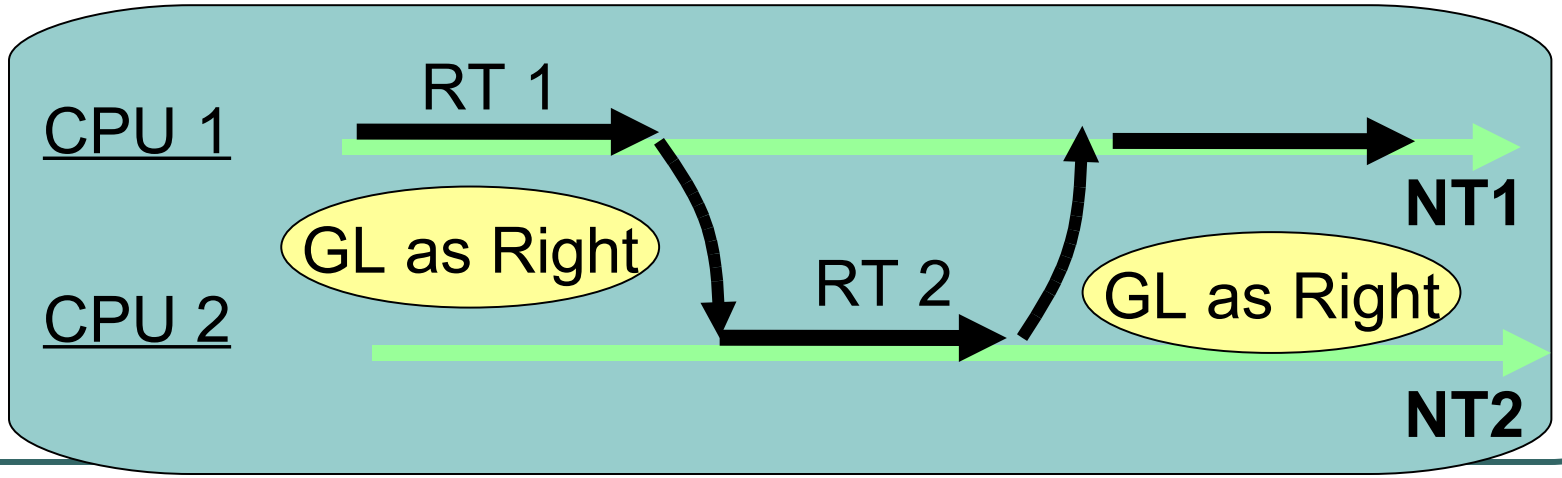
- TSでない既存のコードはGLで保護
- ○ 既存のコードが利用可能（並列実行不可）
- ○ とりあえず今までどおりCメソッド記述可能
- ○ 必要に応じてFGLによるTSコードへ移行可



GL: Giant Lock

検討：並列実行における排他制御
(c) ジャイアントロックを持つRTのみ実行

- GLを実行権として扱い、持つRTのみ実行
- ○ 排他制御不要
- × 並列実行不可能



検討：並列実行における排他制御
(d) 排他制御しない（Rubyプログラマ任せ）

- ○ 排他制御不要
- × 処理系エラー（SEGV等）頻発
- 低級言語的アプローチ

検討:
排他制御の各方式のまとめ

Rubyらしさ(?)

	スケーラビリティ（台数効果）	逐次処理の性能	既存コードの再利用	やさしさ
(a)	◎	○	×	○
(b)	○	○	○	○
(c)	×	◎	○	○
(d)	◎	◎	○	×

性能追求だけなら

Single Core用

Hacker用？

設計：ネイティブスレッドを用いた並列実行可能な**Ruby**スレッド処理機構

- ネイティブスレッドを 1:1 で利用
  - Pthread or Win32APIを利用
- 現状の Ruby API をほぼ実現
  - Rubyスレッドの割り込みも可能
- システム側でスレッドセーフに
  - スレッドセーフに書き換え可能
  - TSではないCメソッドも実行可能

設計：**Ruby**スレッドのネイティブスレッド対応生成管理

- Rubyスレッドの生成・終了・排他制御→ネイティブスレッドのAPI利用
  - 実行終了後、スレッドオブジェクトはGCまで生存
- Rubyスレッドの合流→独自実装
- RTのスケジューラ→ NTのスケジューラ
  - RubyAPIの優先度とネイティブスレッドの優先度をマッピング

設計：**Ruby**スレッドのネイティブスレッド対応
Rubyスレッドへの割り込み

- Rubyスレッドは割り込み可能
  - 他スレッドに例外可能
    - Timeout 処理に利用
      特に I/O のタイムアウト
  - シグナルハンドラの実行

```
t = Thread.new{
  #...
}
t.raise(exception)
```

- ネイティブスレッドには「割り込み」インターフェースは無し

設計：**Ruby**スレッドのネイティブスレッド対応 **Ruby**スレッドへの割り込み**(cont.)**

- (1) 割り込みフラグのポーリング
  - VM命令のいくつかの箇所でチェック
- (2) ブロック状態になる前にブロック解除関数を登録
  - ポーリングしない部分でも割り込みに対応
  - 例：selectシステムコールを中断させるブロック解除→当該スレッドにシグナル送信（select は EINTR で中断）

設計：**Ruby**スレッドのネイティブスレッド対応 管理スレッドの用意

- UNIXでのシグナル受信
  - PthreadではどのNTに配送されるか不明→シグナル受信専用の管理スレッド
  - 他のNTはシグナルマスクで受信不可
  - 受信したらmainスレッドへRubyスレッドへの割り込み機能を利用して割り込み

設計：
Rubyスレッドの並列実行

- スレッド情報へのポインタをTLSに
- 適切な同期・排他制御の導入
  - TSなコードの登録
  - 非TSなコードのための適切なGL制御
- GCのための同期
- ロック競合を避けるための工夫

設計：**Ruby**スレッドの並列実行
スレッド情報へのポインタを**TLS**に

- スレッド情報へのポインタをネイティブスレッドが提供するTLSに格納
  - TLS: Thread Local Storage
    - GCC 拡張
    - Windows VC 拡張
    - pthread_getspecific の利用

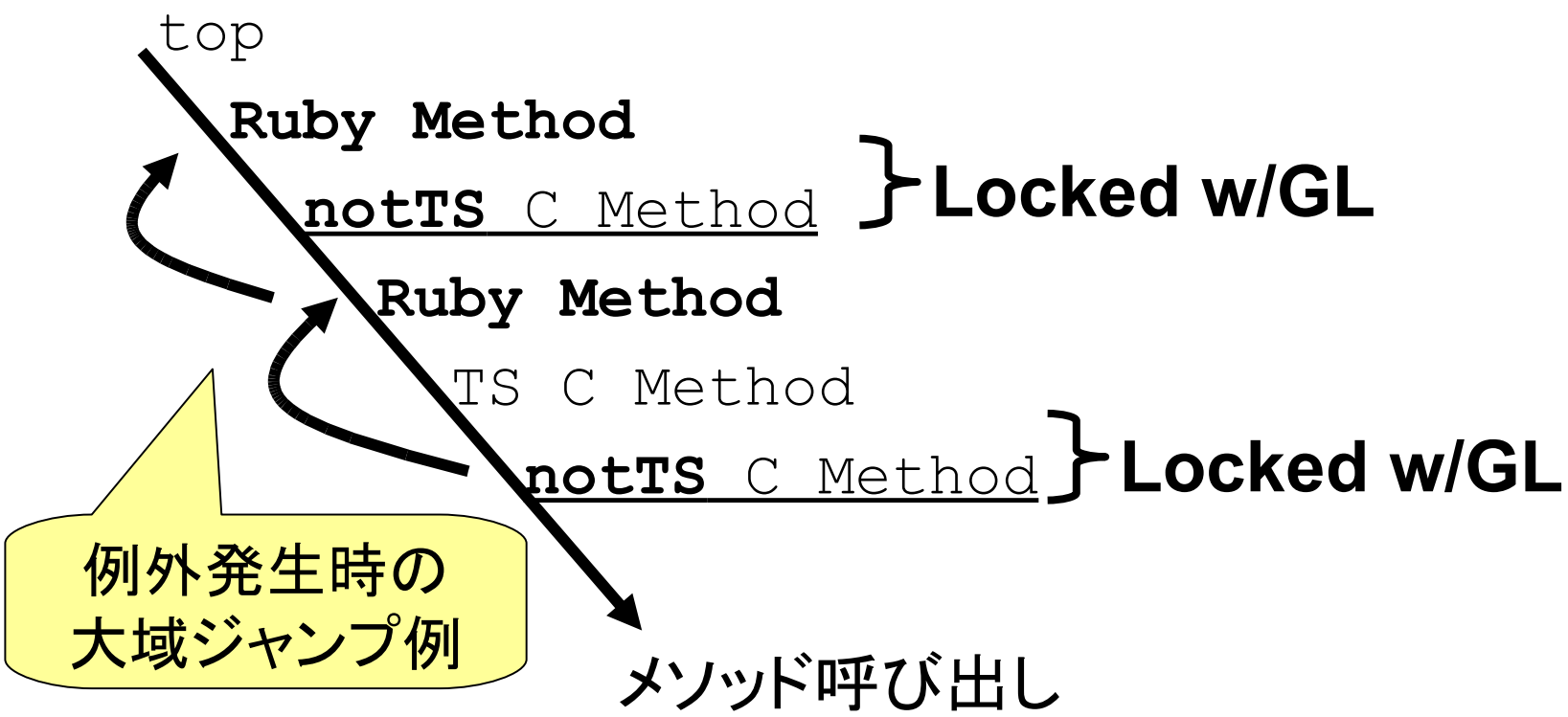
設計：**Ruby**スレッドの並列実行
TSなコードの登録

- TS（スレッドセーフ）メソッドの登録
  - 本質的にロック不要なCメソッド、細粒度ロックを利用して作り直したCメソッドはTSメソッドとして登録（起動時にGL獲得不要）
  - 具体的なTS化はライブラリ依存
  - Cメソッド登録APIに「TSメソッド登録」API追加
    - rb_define_method → rb_define_method_ts
  - 必要なCメソッドのみTS化することが可能
    - 性能に影響ある、頻繁に呼ばれるメソッドなど

設計：**Ruby**スレッドの並列実行
非**TS**なコードのための適切な**GL**制御

- 非TSなコード == 既存のCメソッド実装
  - メソッド起動時にGL獲得
- TSなコード ==
対処したCメソッド or Ruby メソッド
  - GL獲得なしで起動可能
- CメソッドはRubyメソッド呼び出し（逆も）可能
  - メソッド呼び出し時、GL獲得状態によって処理分岐
- 例外発生時のGL獲得情報に注意

設計：Rubyスレッドの並列実行
!TSなコードのための適切なGL制御



設計：**Ruby**スレッドの並列実行
GCのための同期

- RubyのGC → 保守的mark&sweep
  - GC時、すべてのRubyスレッドを停止
  - 割り込みのためのポーリングの機構を利用
  - ブロック時にはマシンコンテキストを保存
- 並列GCは今後の課題

設計：**Ruby**スレッドの並列実行
ロックコストを下げるための工夫

- 排他制御のためのロック→オーバヘッド
  - とくにロック競合は高いオーバヘッド
- 下げるためのいくつかの工夫
  (1) 単一スレッド実行時にはGL無視
  (2) スレッドローカルヒープの用意
  (3) ロック不要なメソッドキャッシュ
  (4) スピンロックの利用
  (5) 利用CPUの制限
  (6) 並列実行しない（本質的に並列実行に向かない場合）

設計：**Ruby**スレッドの並列実行
ロックコストを下げるための工夫**(cont.)**

- ロックを不要に
  - (1) 単一スレッド実行時にはGL無視
  - (2) スレッドローカルヒープの用意
    - オブジェクトアロケーション時、ロック不要
  - (3) ロック不要なメソッドキャッシュ
    - キャッシュヒットすれば余計なコスト不要
    - キャッシュミス時、コスト増

設計：**Ruby**スレッドの並列実行
ロックコストを下げるための工夫**(cont.)**

- (5) 利用CPUの制限
  - 競合を繰り返すRubyスレッドの実行CPUを制限
  - NPTLのpthread_setaffinity_npを利用
    - Windows では SetThreadAffinityMask
  - 一定時間過ぎたら制限を解除

    - ちょっと後ろ向きの工夫
      （v.s. TS なコードを増やす）

評価
評価環境

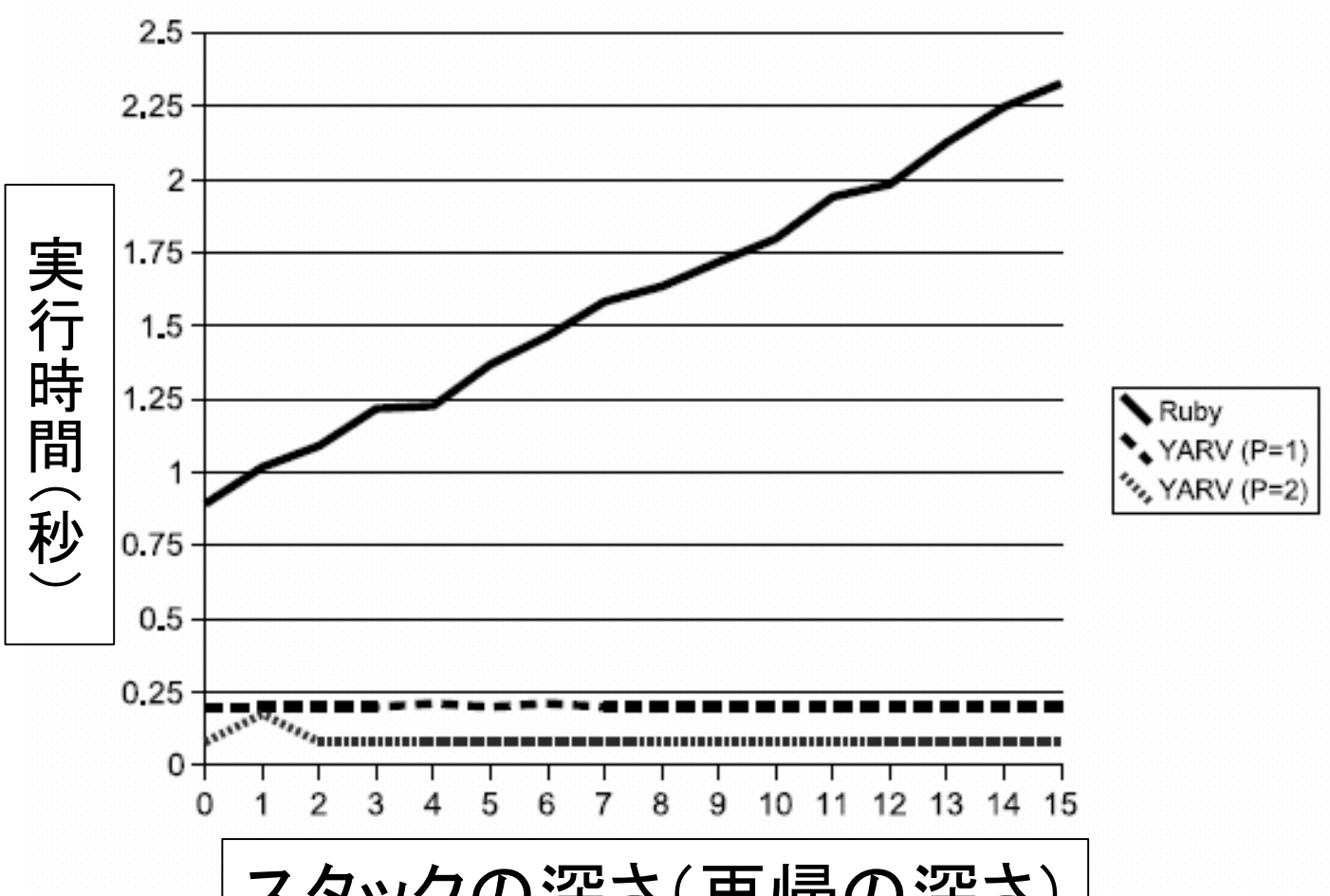
- 評価環境
  - CPU: Intel(R) Pentium(R) D CPU 3.46GHzプロセッサ（Dual Core/L2 2MB）、4GBメモリ、
  - OS: Linux 2.6.17-1.2174_FC5 SMP / NPTL
  - Compiler: gcc version 4.1.1 20060525 (Red Hat 4.1.1-1)
- 比較対象のRuby
  - ruby 1.9.0 (2006-04-08) [x86_64-linux]
- YARV最適化オプション
  - 融合操作とStack caching 以外すべて

評価
スレッド制御

- 生成・合流はNT制御オーバヘッドで低速に
- 排他制御はCで再実装したため高速化
- この性能はNTライブラリに依存

	Ruby (sec)	YARV (sec)
生成	1.82	5.06
合流	0.58	2.32
排他制御	3.32	0.43

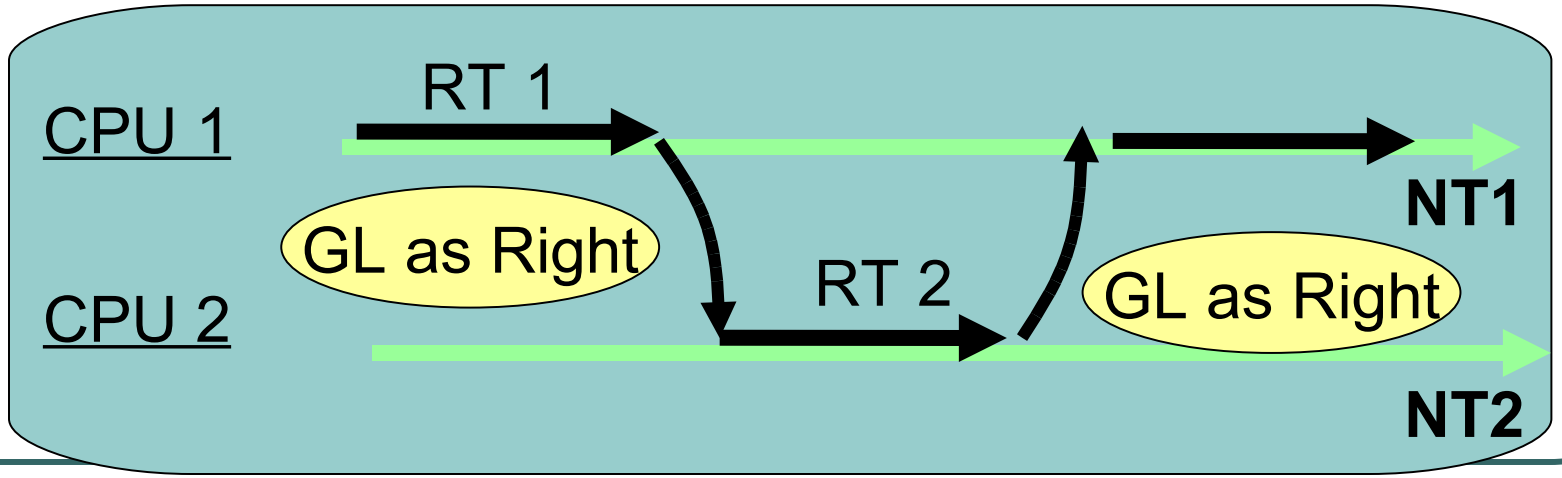
評価
スレッド切り替え時間



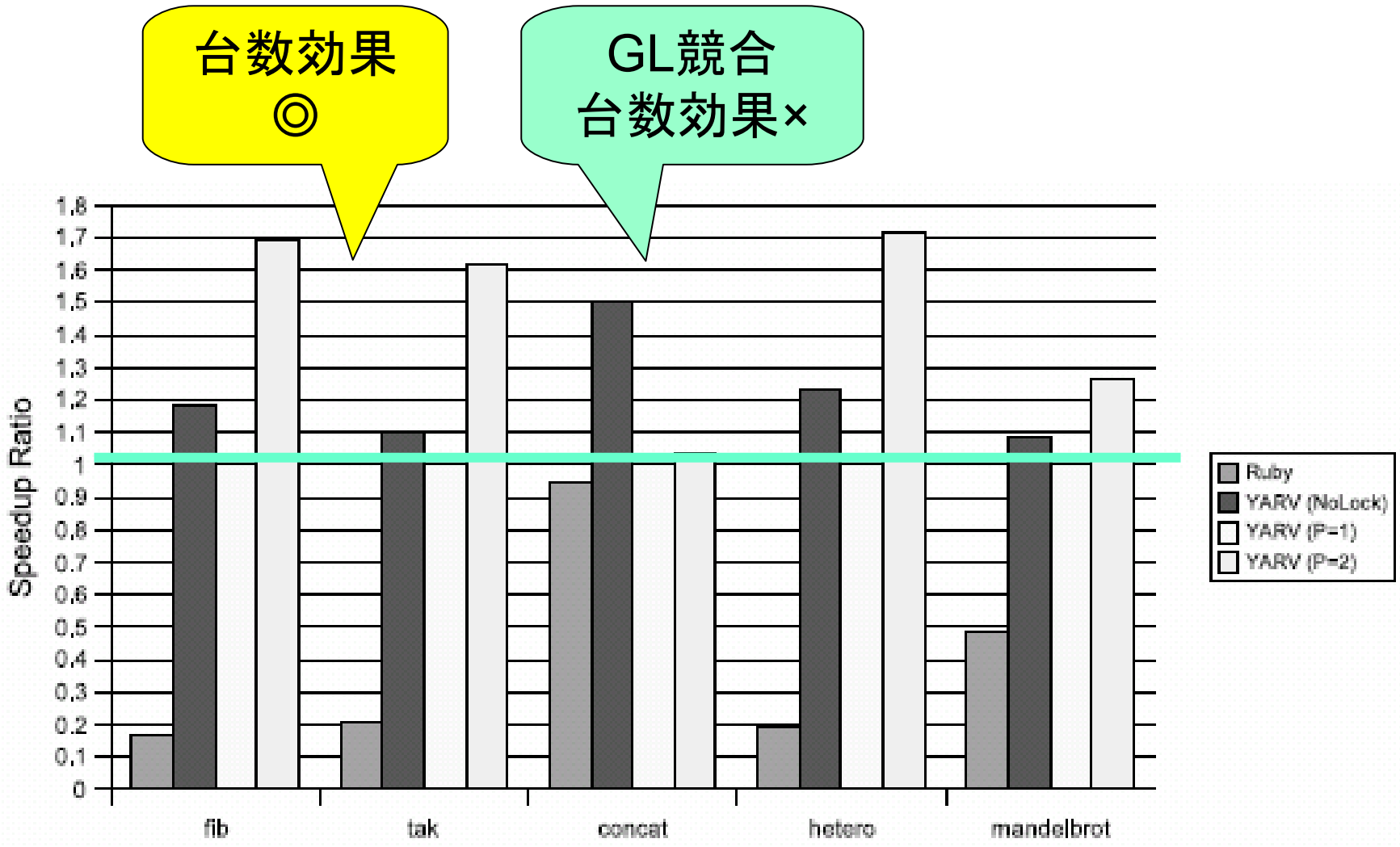
検討：並列実行における排他制御
(c) ジャイアントロックを持つRTのみ実行

- GLを実行権として扱い、持つRTのみ実行
- ○ 排他制御不要
- × 並列実行不可能

YARV (NoLock)



評価
並列実行(マイクロベンチマーク)



まとめ

- Rubyスレッドを並列実行可能に
  - **RTをNT**にマッピング（**1:1**）
  - 並列実行のために排他制御・同期を導入
    - GLを導入し既存のコードが利用可に
    - TSに書き換えることで段階的に並列度向上
    - ロックの競合回避のための工夫
  - 評価の結果、TSコードは台数効果確認

今後の課題

- 処理のTS化（特に文字列・配列処理）
  - 頻繁に利用するがとにかく規模が大きい
- 並列GCの実装
- >2 なCPUコア数での評価
- Rubyプログラミングモデルの変更
  - Rubyスレッド生成は「速い」→「遅い」
  - スレッドプールの利用など

おわり

ご清聴ありがとうございました。

笹田 耕一 / ささだこういち

sasada@ci.i.u-tokyo.ac.jp
ko1@atdot.net
http://www.atdot.net/yarv/

【謝辞】
YARV: Yet Another RubyVM の開発には
大変多くの方にご協力いただいております。
本プロジェクトでは IPA 未踏ソフトウェア創造事業の
支援を受けています。
この場を借りて深く御礼申し上げます。

以降、補足

Rubyを高速に実行するための処理系
YARV: Yet Another RubyVM

- 昨年度未踏ユース
- VM命令セットの設計
- コンパイラの設計・開発
- 仮想機械(RubyVM)の設計・開発
  - スタックマシン（JavaVM / .NET などと一緒）
- 高速化のためのさまざまな最適化
- オープンソースソフトウェアとして公開中
- http://www.atdot.net/yarv/

YARV完成度

- ほとんどのRubyプログラムが実行可能
  - テストスクリプトがほぼ動作

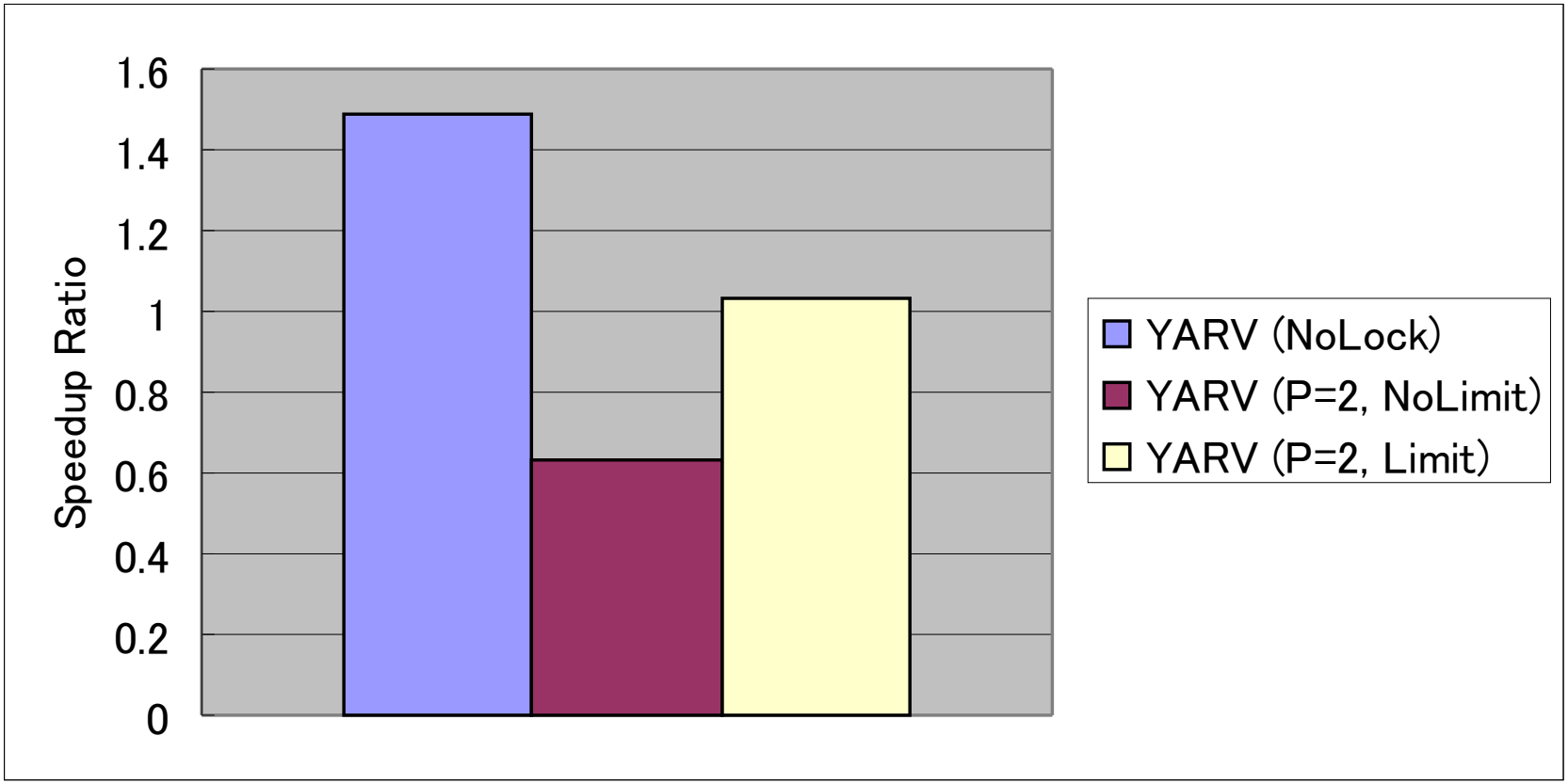
GL競合回避の工夫 その2
GL Stick

- 競合を繰り返すスレッドがGLを獲得し続ける(解放しない)Stick方式
- 一定時間でStickは解除
- 汎用的(CPU制限APIは必要なし)
- マクロベンチマークで若干いい性能
  - もちろん YARV (NoLock) には勝てない

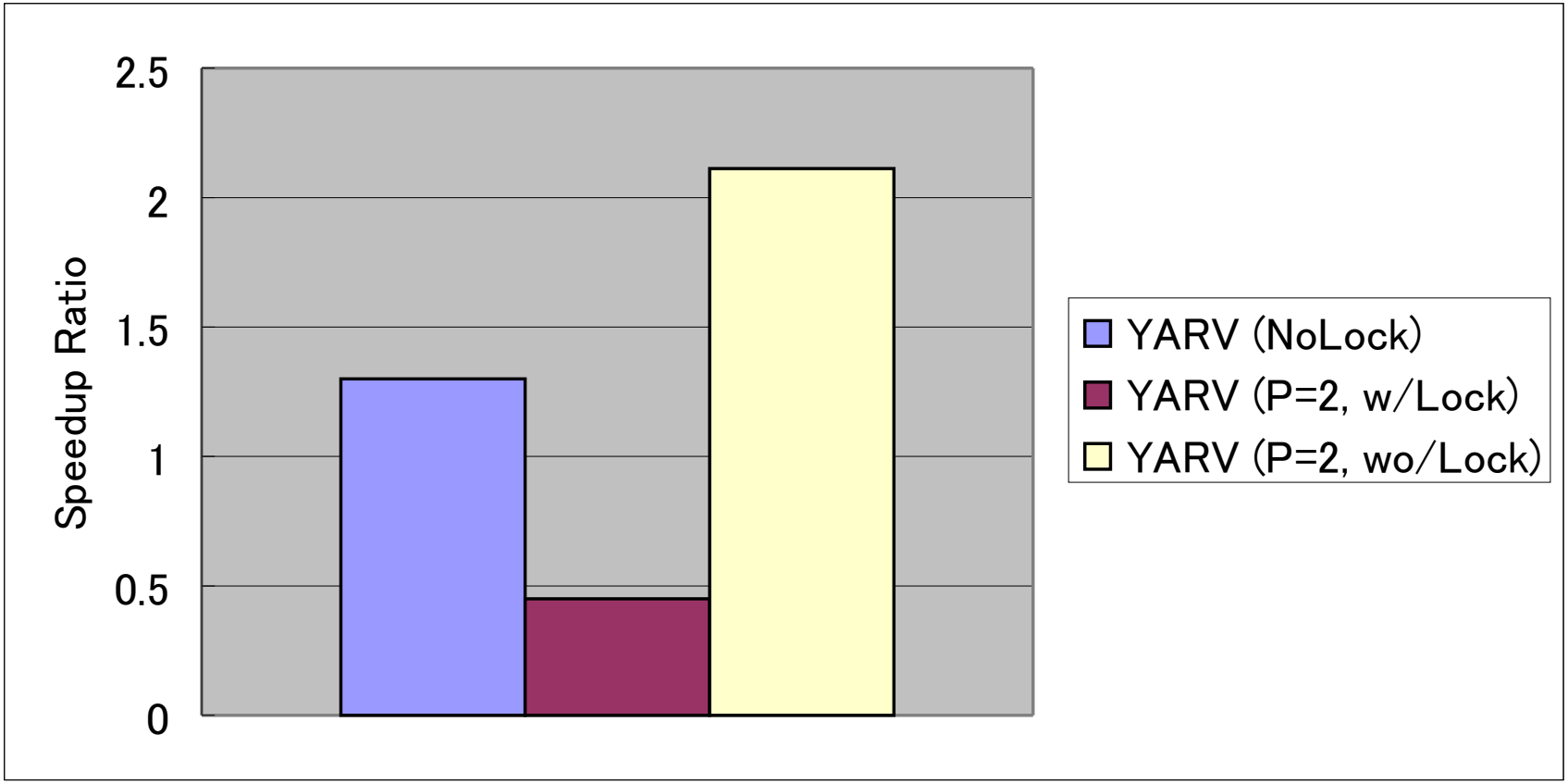
文字列・配列処理のTS化が面倒な理由

- 文字列・配列は生データを多くの場面でアクセス（参照）されているため、文字列・配列処理部分のみ作り直せばよいというわけではない
  - 文字列・配列のデータ型を変更し、コンパイルエラーによって上記変更が必要な箇所を洗い出して作り直す予定

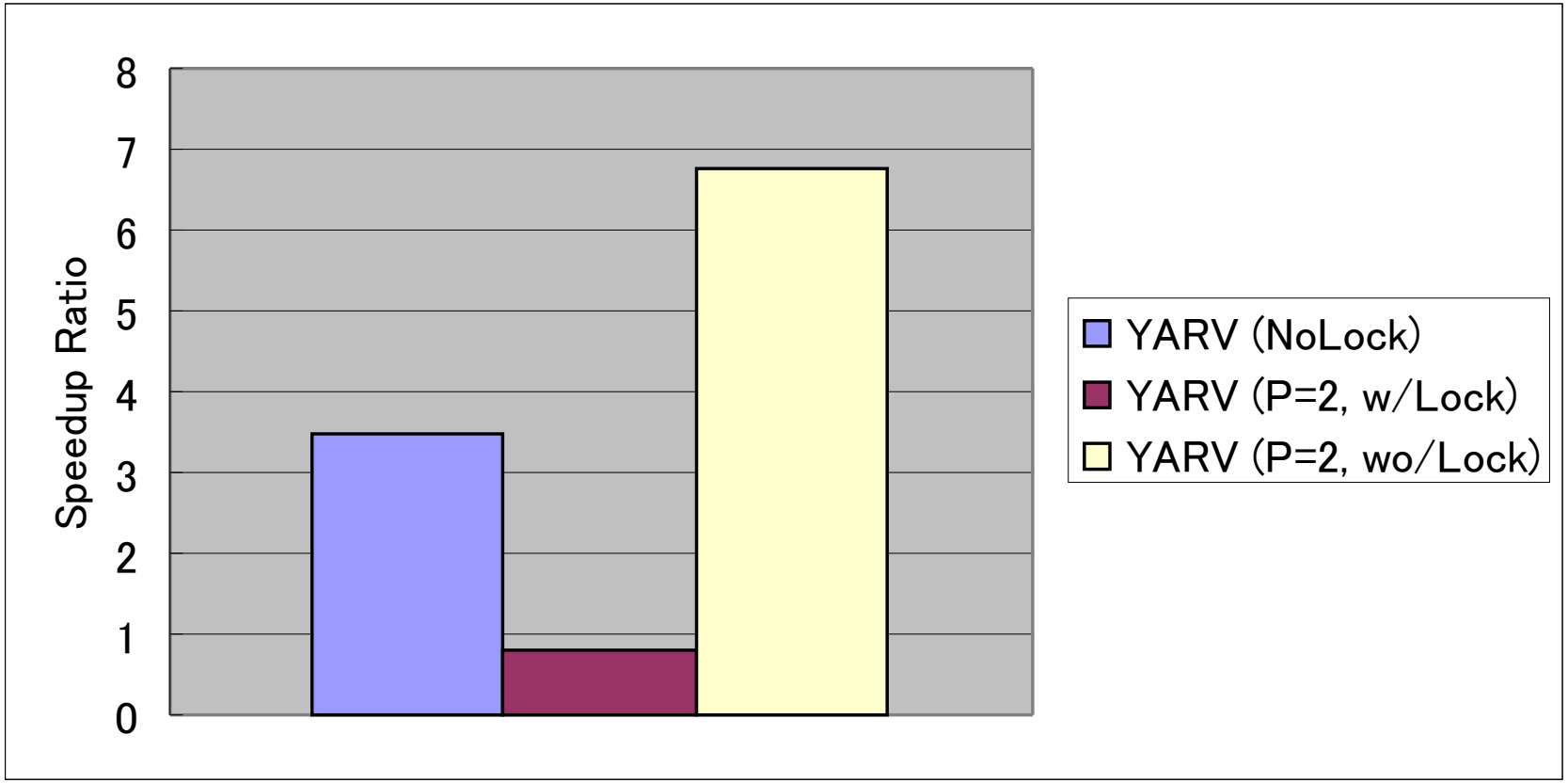
評価
CPU制限の効果


Speedup Ratio
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
YARV (NoLock)
YARV (P=2, NoLimit)
YARV (P=2, Limit)

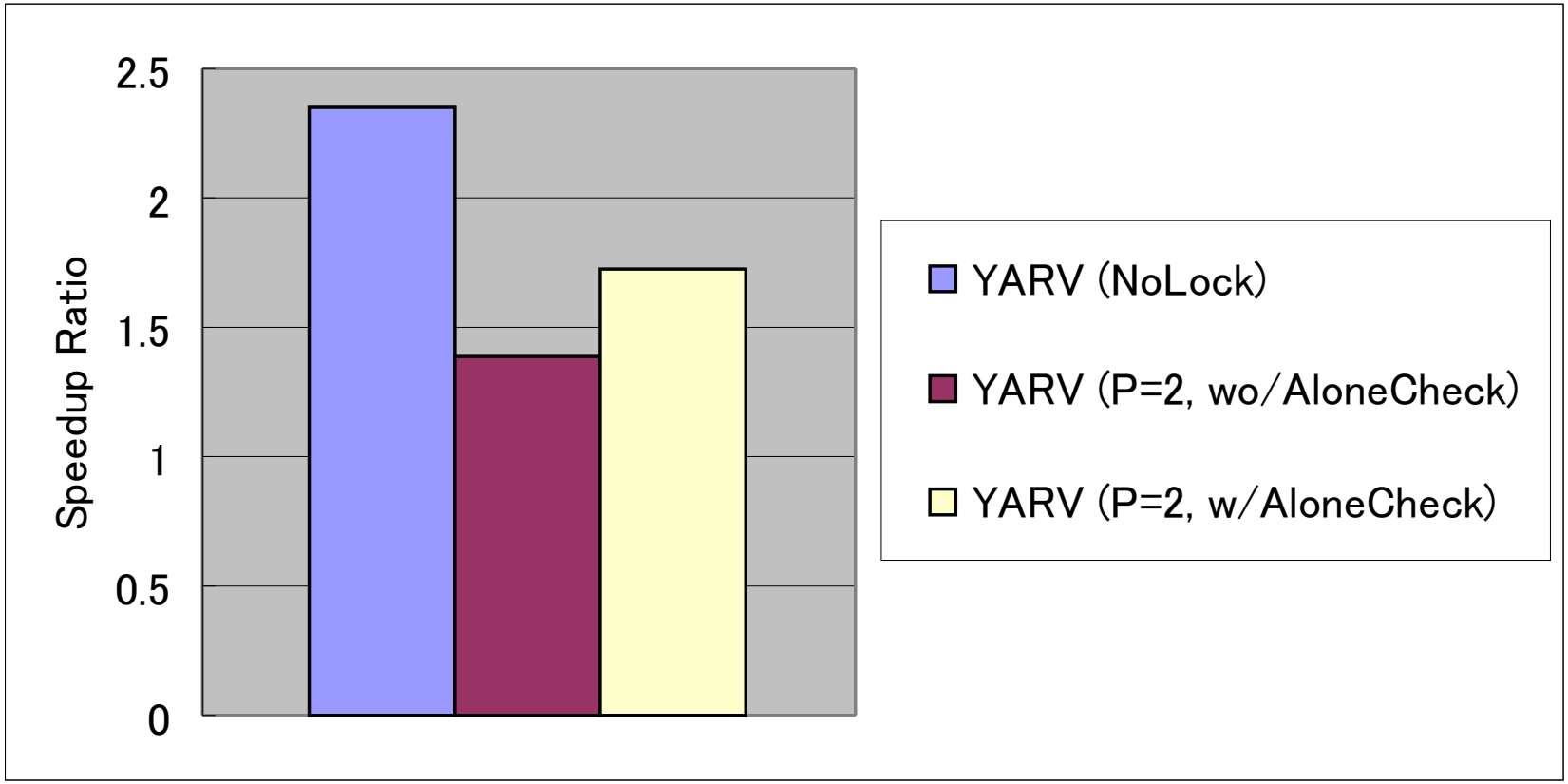
評価
スレッドローカルヒープの効果



評価
ロック不要なメソッドキャッシュの評価



評価
単一スレッド時にはGL制御しない



評価
GL獲得時スピンロックに