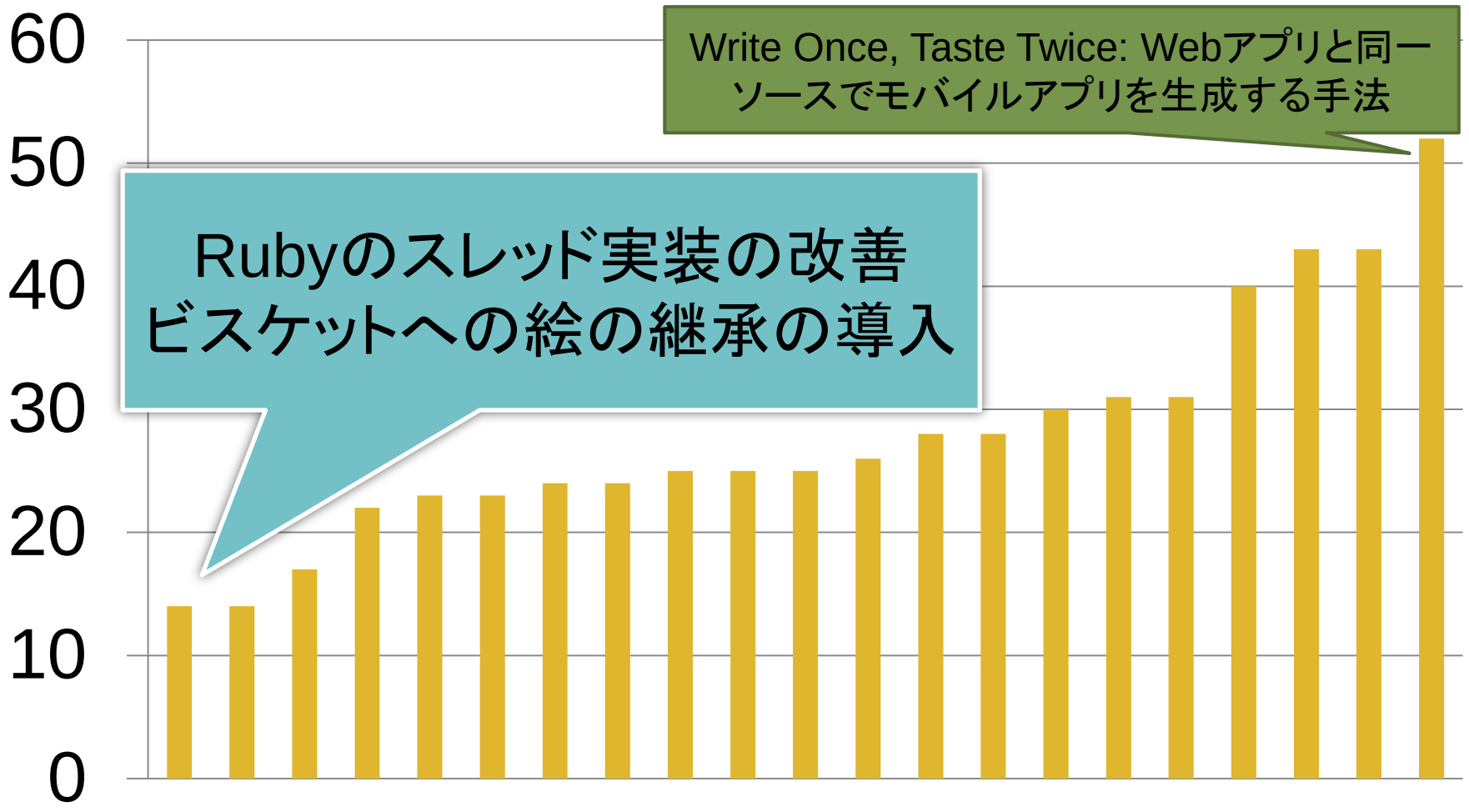
Rubyのスレッド実装の改善

東京大学大学院
情報理工学系研究科 創造情報学専攻
笹田耕一
<sasada@ci.i.u-tokyo.ac.jp>

始める前に
プロシン原稿担当幹事から2つ

- 予稿集の「査読付論文」明記出来ず失礼しました
- 結果「査読付論文」のシンボルは流行の3D仕様
- まだ「査読付論文」シールは余っています.

始める前に
プロシン原稿担当幹事から2つ

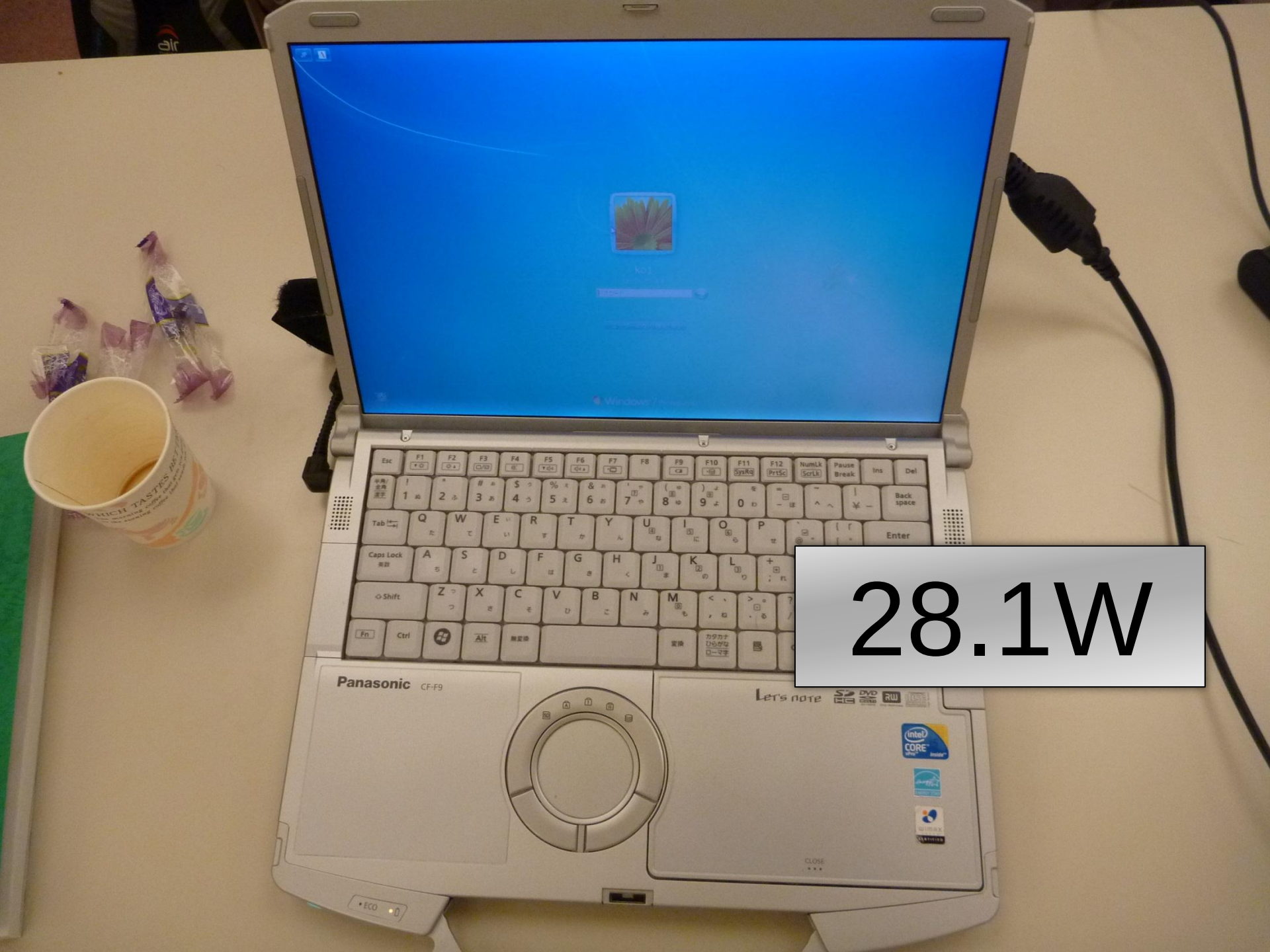


本発表のキーワード

低消費電力

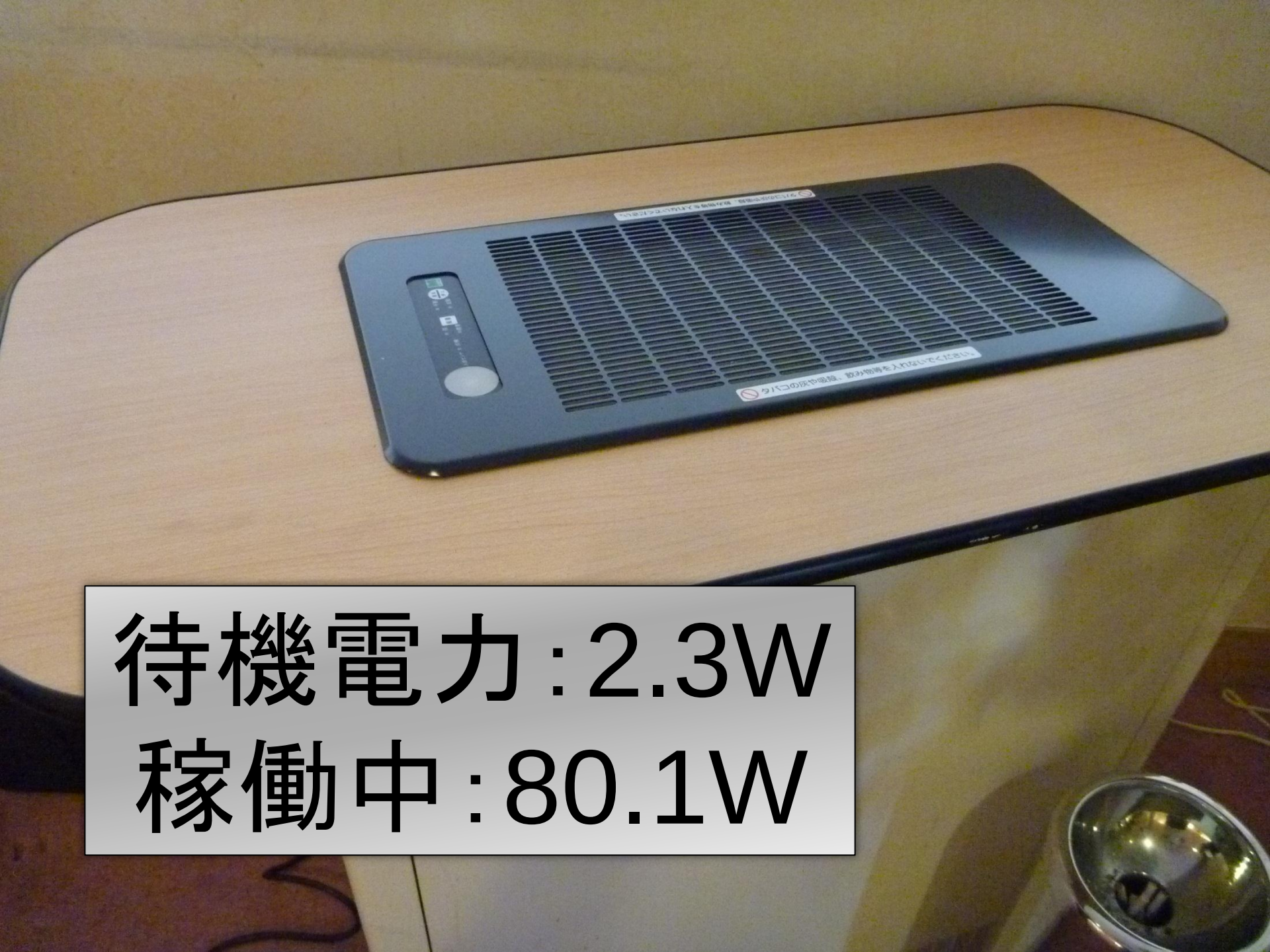
マルチコア


ES
watts up? PRO
145.7
WATTS
LOG
SELECT
MODE


28.1W
Panasonic
CF-F9
Let's note


保温（上）
だけ
145.7W
UNIMAT OFFISCO
適温
保温(下)
保温(上)
入
切
入
切
UNIMAT OFFISCO


第52回プログラミング・シンポジウム
様
25.0W


待機電力:2.3W
稼働中:80.1W

本発表のキーワード

低消費電力

マルチコア

本発表の成果

低消費電力

0.3W くらい電力削減

マルチコア

バグを修正

想定されていたシナリオと実際

想定されていたシナリオ

プロシン用にキャッチーなネタで
さくっと論文を書いてしまおう

実際

なんでこうなるのかわかんねー
たくさんの人に助けられました

得られた成果

0.3W 電力削減
BugFix

＊有名なソフトウェア工学の
ブランコの絵から引用

では，本編

背景
Ruby

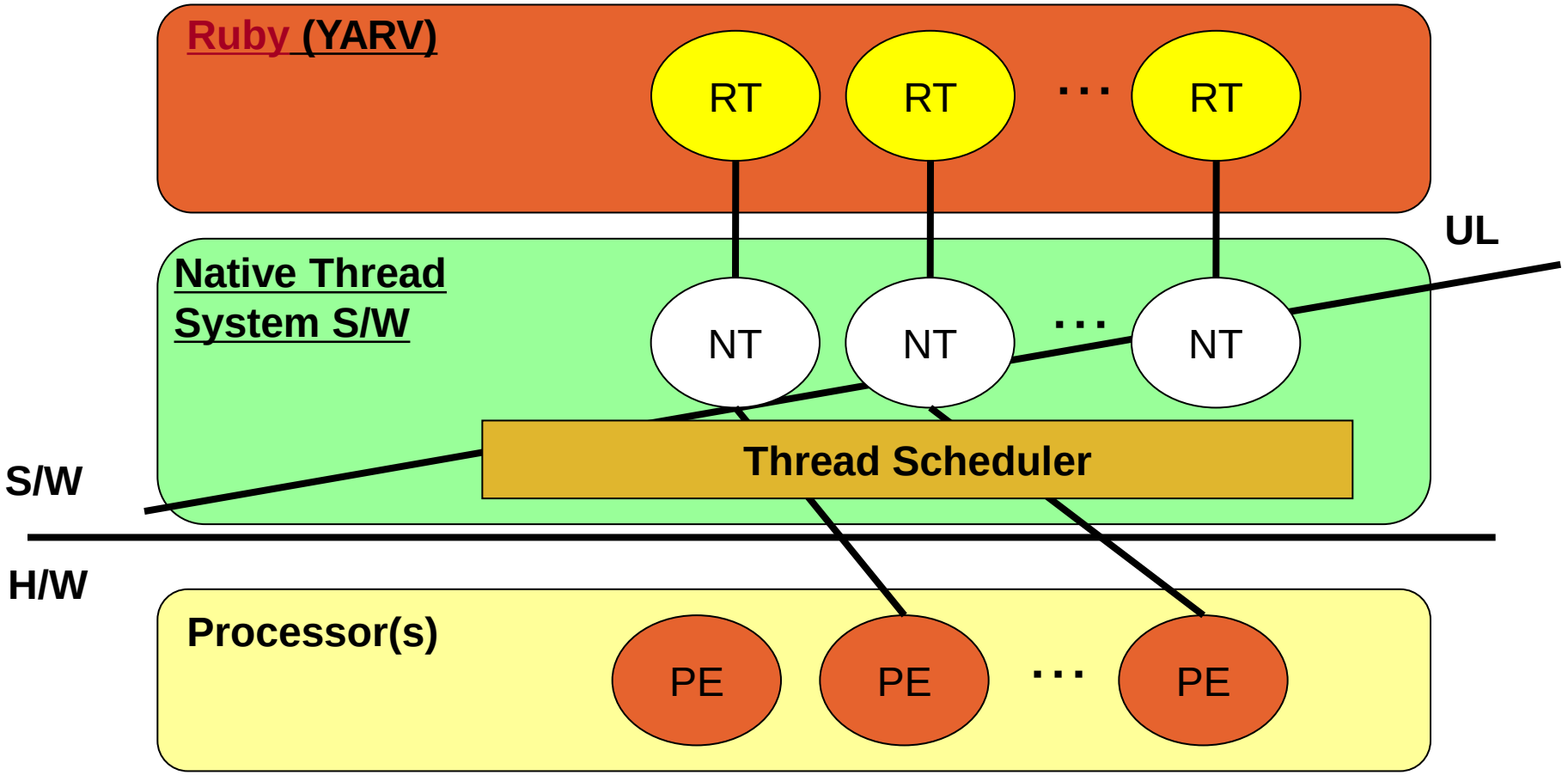
- プログラミング言語Ruby
  - オブジェクト指向で云々，Ruby on Rails で云々
  - スレッドプログラミングが可能
  - 2010/08/18 Ruby 1.9.2 リリース
  - 2010/12/25 Ruby 1.8.7，1.9.2 パッチリリース
  - Ruby 1.8：安定版（多分，一番使われている）
  - Ruby 1.9：最新版（RoR3 も動く（らしい））
  - 今回は，Ruby 1.9 のお話
- Ruby 1.9
  - M17N（Multilingualization）
  - VM（YARV）
  - **ネイティブスレッドを利用**

背景
スレッドの実装

- 用語の整理
  - Rubyスレッド（RT）：Rubyプログラムで作るスレッド
  - ネイティブスレッド（NT）：システムのスレッド

- **ネイティブスレッドを利用したRubyスレッド**
  - 1つの NT で 1つのRT（1:1モデル）
  - 1.8 までは1つのネイティブスレッドですべてのRubyスレッド（1:Nモデル）

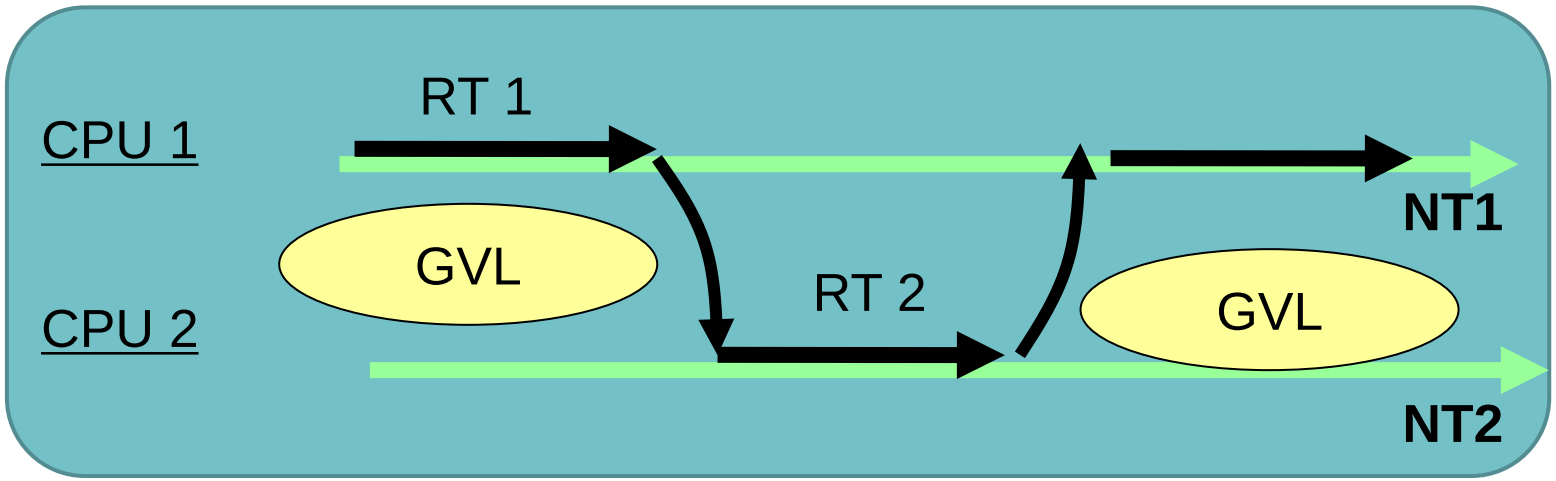
背景
Rubyスレッドとネイティブスレッド



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

背景
ジャイアントVMロック（GVL）

- ジャイアントVMロックで並列実行はさせない
  - Python などでは GIL（Giant Interpreter Lock）と呼称
  - （本発表とは関係ないが）並列に実行するRubyスレッドはRubyプログラミング（かんたんプログラミング）的にまずいんじゃないかと思っている
    - （D論のテーマはそれだったんだけど）



背景
タイマスレッドと割り込み

- タイマスレッド
  - 定期的に起動し，GVLを放棄するように，GVL所有スレッドのフラグをセット
  - GVL所有スレッド == 現在動いている Ruby スレッドは定期的に（特定のVM命令で）そのフラグをチェック．必要なら GVL を解放して他に譲る
- 割り込み（シグナルとかThread#raise）の対応
  - ブロックするようなシステムコール（e.g. select）を「安全に」キャンセルするためには trick が必要
    - キャンセルを確実に行うために，定期的な処理が必要
  - 詳細は予稿集を参照（説明が面倒）
  - （キャンセルできないものも中には…（未解決））

Rubyのスレッド実装に関する問題（の中で実際にクレームがあった2つ）

(1)タイマースレッドによって計算機の消費電力が増える（というクレーム）

(2)マルチコアマシンでGVLが正しく受け渡しが行われないことがある（1つのRubyスレッドだけが延々と走り続ける）

問題1：消費電力
タイマスレッドによる消費電力増

- 一定時間（t = 10ms）ごとにフラグをセット
  → 現在の実装では，**while (1) {sleep(t); flag = 1}** のような実装
- **誰も flag をチェックしないような状況でも，定期的にタイマスレッドだけは定期的に起動**
  - 1 thread しか走っていないとき
  - すべての Thread が sleep しているとき
- CPUが低消費電力状態（halt）を維持出来ない
  - システム全体がアイドルな場合（**アクセスのないサーバ**等）
  - 最近のCPUが持つDVFS（動的電圧・周波数制御）
  - powertop というツールを使うと簡単に観測可能

問題1：消費電力
解決案の検討

- GVL解放タイミングをどのように通知するか？
  (1) setitimer など，タイマシグナルを活用
  (2)「GVLを待っているRubyスレッド」が通知
  　　Pythonの新しい版はこうしているらしい
  (3) やっぱりタイマスレッド
  　　ただし，誰もGVLを待っていない時は起きないように
- 議論
  - (1) は計算機・OSの構成によって定期的にシグナルが来ない時があった（**フェアネスに問題）**
  - (2) は，たくさんスレッドが要る場合面倒
  - というわけで，(3) に

問題1：消費電力
解決策の詳細

- 必要なときだけタイマスレッドを活性化
  - 普段は無限に待つ（名前無しセマフォ）
  - GVLを待つスレッドが現れたら定期的に起きるように
  - シグナルなんかも同じ仕組み

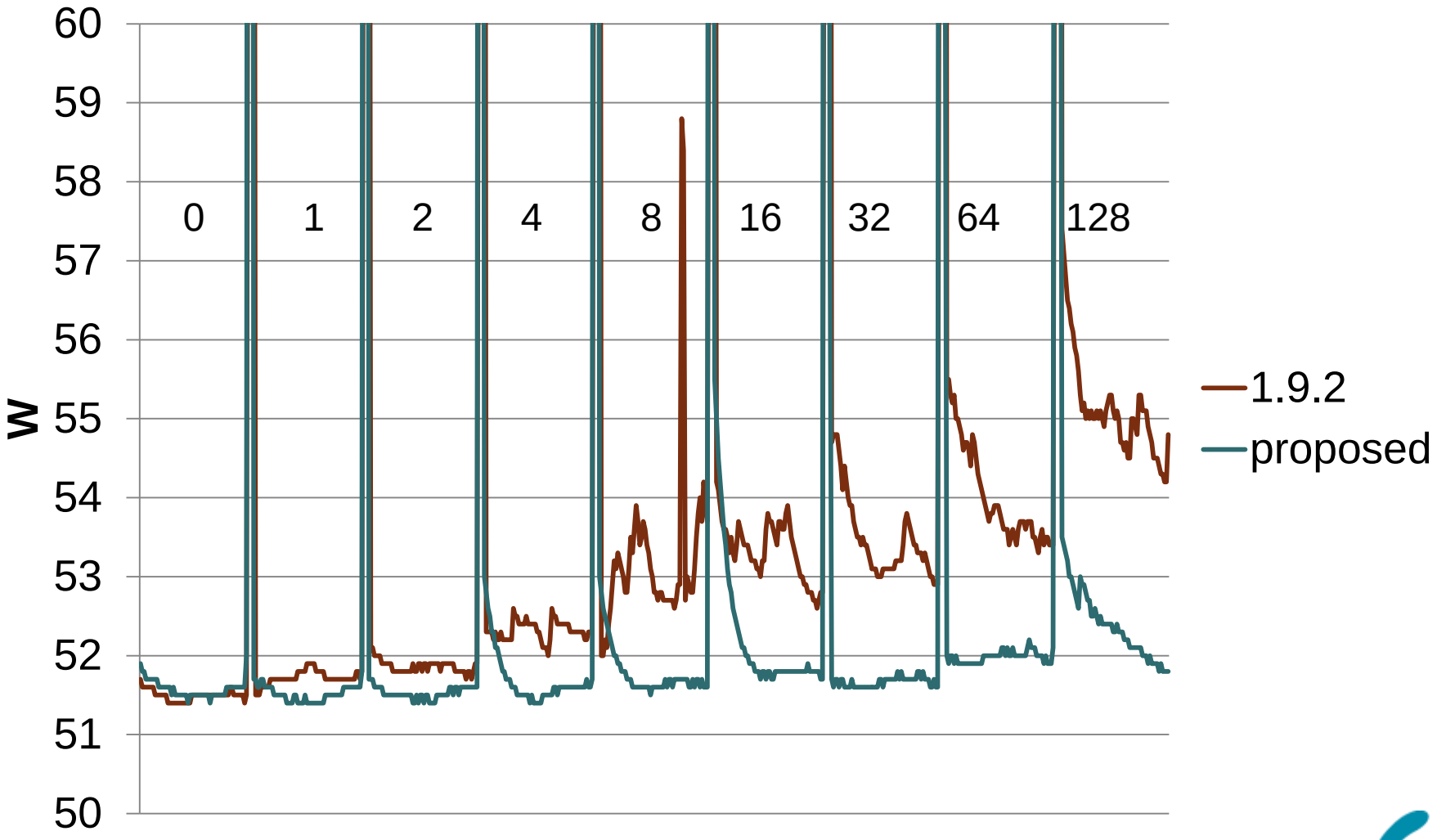
- セマフォが使えない場合は特別のケアが必要
  - 例えば MacOS X では名前無しセマフォはサポートしていないらしい

問題1：消費電力
解決した結果の評価

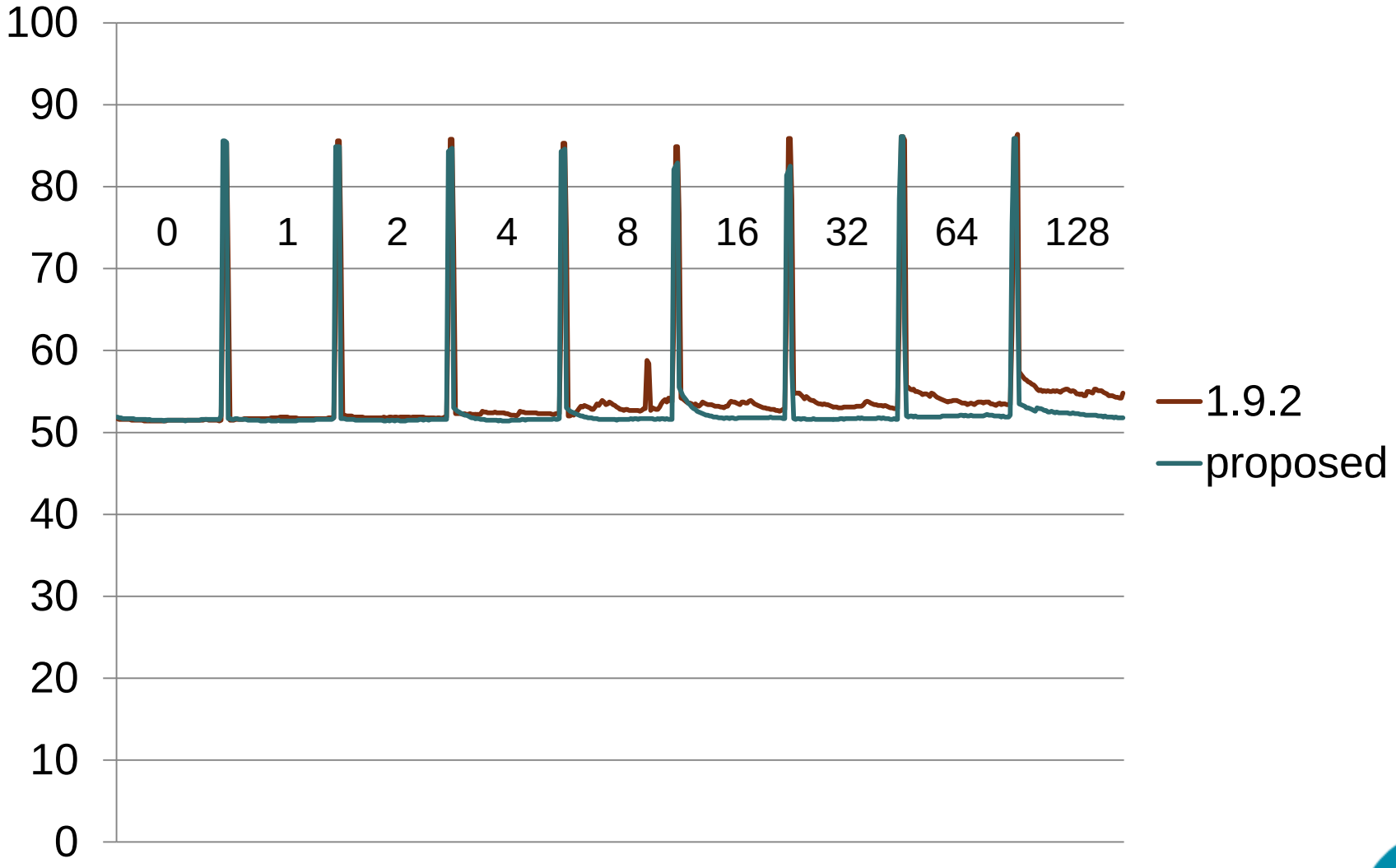
- というか，実際どれくらい「電力消費が上がっていたのか？」の評価
  - 「苦情」来たけど，実際どうなのよ
  - 解決していれば電力消費が 0 になるから自明
- 評価環境
  - Intel Core i5（4 core）
  - Ubuntu Server 10.10
  - 電力計：Watts Up? PRO
    電力消費/秒のログをUSBで取得可能

- Idle Ruby process（sleep するだけ）がどれくらい電力を消費するのか，を測定

問題1：消費電力
消費電力の様子（？）



問題1：消費電力
消費電力の様子（真）



問題1：消費電力
電力消費のまとめ

- IdleなRubyプロセスの消費電力0.3Wくらい削減
  - Idleシステムは **50W** くらい（Intel の速いCPUのPC）
- 電気代（東京電力）
  - 17.87円/1kWh (*1) × 0.3W/1000 × 30 × 24 = 3.86円（1月の電気代の節約）
  - 3.86 * 12 = 46.31円（1年の電気代の節約）
  - 10,000世帯にホームサーバ＋Ruby 1.9.2設置なら 46.31 * 10,000 = **463,190 円（年間）**

この研究は45万円ほどの成果

*1: 最初の 120kWh まで

問題2：GVL受け渡し問題

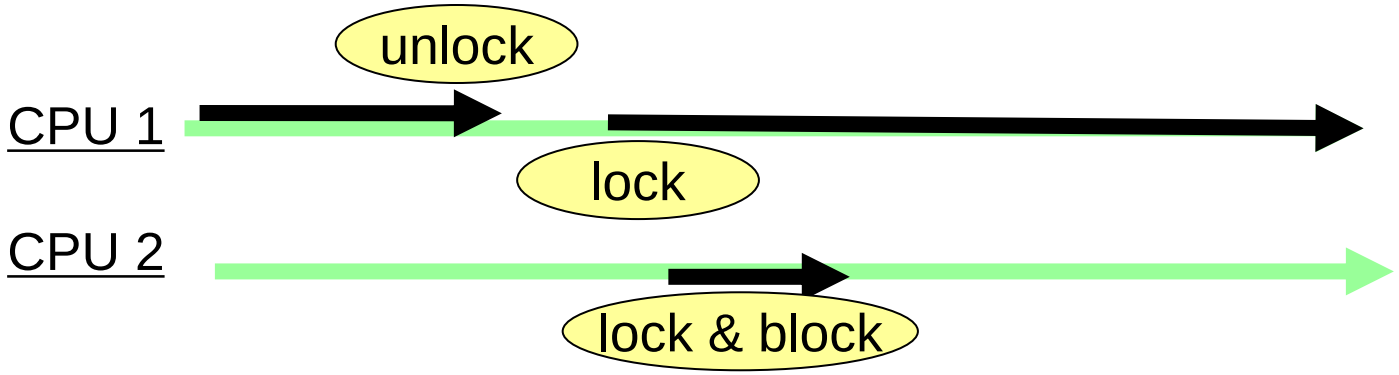
- マルチコアマシンでGVLが正しく受け渡しが行われないことがある（1つのRubyスレッドだけが延々と走り続ける）

```
flag = true
Thread.new{
  while flag; end
}
flag = false
```

- 現象
  - マルチコア環境でRubyスレッドが別々のCPUにスケジューリング
  - あるRubyスレッドがGVLを解放
  - 期待：他のRubyスレッドがGVLを獲得
  - **実際：同じRubyスレッドがGVLを再獲得**

問題2：GVL受け渡し問題
受け渡らない理由

- GVLの実装：pthread_mutex_lock/unlcok
  - Linux等では，CPUコアごとにスケジューラを持つ
  - 各ネイティブスレッドは各スケジューラに所属
  - ロックを外すと，待ちスレッドが resumeし，ロックを取ろうとするが，他のCPUコアで起きたスレッドはこの競争に負ける



  - そもそも，FIFOで渡すことはPOSIXでは保証していない
  - ちなみに，mutexではなく条件変数やセマフォも同じ
  - ちなみに，WindowsのMutexはFIFOを保証している（らしい．挙動から見て）

問題2：GVL受け渡し問題
確実な受け渡し方法の検討

- 最初の実装
  - 各Rubyスレッドが条件変数を持たせる
  - Ruby処理系レベルで「次に実行するRubyスレッド」をスケジューリング（FIFO）し，GVL受け渡し時に，次に実行するべきRubyスレッドを起動
- ちゃんと動いた
  - しかも，priority も Rubyプロセスが完全に制御可能
- しかし，遅い！
  - スケジューリングだけで凄い時間を食っている
  - 例：n.times{} を複数Rubyスレッドで実行すると2倍

問題2：GVL受け渡し問題
遅い理由の分析

- **CPUをまたぐ**スケジューリングは遅い
  - Linux等ではCPUごとにスケジューラを保持
  - OSのスケジューラの上にもう一つのスケジューラ
    - そもそも，GVL を用いた単一スレッドのみ実行させるようなスケジューリングをOSは想定していない
  - CPUをまたいで実行権を委譲するのは遅いらしい
    - OSのバージョンに大いに影響を受けるとは思うが…
  - I/O時などでGVLを離すだけでこの**重い処理**が走る
- 問題の再定義
  - **ユーザレベルで確実に効率的なスケジューリングを実現するには？（OSに手をいれずに）**
  - まさに，Scheduler Activations 等が解決する話
（でも**普通のOSには無い**し）

問題2：GVL受け渡し問題 解決策（最終決定はまだ）

- いろいろな方法を検討
  - 予稿集には4つ
  - 実装は8つくらい
- 採用案
  - GVLは単なるロックとして実装
  - **GVLをただ離す時（I/O実行時など）は，ロックを外すだけ**
  - 他のRubyスレッドへ実行権を委譲する時は，同じスレッドが再獲得しないように工夫（工夫の詳細は予稿に）
- そもそも，Rubyスレッド切り替え間隔 10ms は小さすぎる → 100ms へ
- そもそも実行するCPUコアを1つに限定という手も…

（ポータブルでないが）

おまけ
シグナルの扱い

- POSIXシグナルをタイマスレッドに集約していた
  - POSIX Thread では，スレッドごとにシグナルマスク
  - シグナルはマスクの掛かっていない**どれか**のスレッドへ
  - システムコールのエラー EINTR（シグナル到着のためにキャンセル）にちゃんと対応しているかわからないため
  - でも，バグでそうなってなかった
- すべてのRubyスレッドが受信するように変更
- EINTRにちゃんと対応しているか，きちんとテスト
  - すべてのシステムコールが100回くらい EINTR を返すようにしてテスト，1箇所だけ未対応部分を確認
  - ptrace を利用（ruby-ptrace + fakeeintr）

先に謝辞

- この研究（とくに，問題2）についての実装のアイデアはRuby開発コミュニティに助言を受けました.
- というか，数日間，深夜5時くらいまで，延々とチャットで「この実装だとどうだ」という話につきあってくれた小崎さん（富士通），樽家さん，遠藤さん（東芝）ありがとうございました.
- オープンソースソフトウェア開発っぽい？

まとめ

低消費電力

0.3W くらい電力削減

マルチコア

バグを修正（調整中）

まとめ（得られた知見）

- 消費電力の話
  - ユーザレベルでのタイマ的なものは 0.3W くらい消費
  - 利用者からのクレームは，あんまり根拠がない

- マルチコアにおけるロックの話
  - GVLのような構造は，単純なロックではうまくいかない
  - コアをまたぐGVLの確実な受け渡しは大きなコスト
  - 意外と難しい（完璧な解決はまだ）
  - OS非依存に性能を出すのはPOSIXの範囲ではそろそろ難しいかも

課題と展望

- Ruby 1.9.3（今年中？）に実装を整理して実装

- そもそも，Rubyスレッドの実行をGVLで1つに制限しているからこんな話が

  →やっぱりRubyスレッドの並列実行？

おわり

Rubyスレッド実装の改善

ささだこういち
<ko1@rvm.jp>