Championship Branch Prediction における取り組み v5

塩谷 亮太[1]，小泉 透[2]，前川 隼輝[2]，
水野 将成[2]，黒木 地球[1]，津邑 公暁[2]

[1] 東京大学，[2] 名古屋工業大学

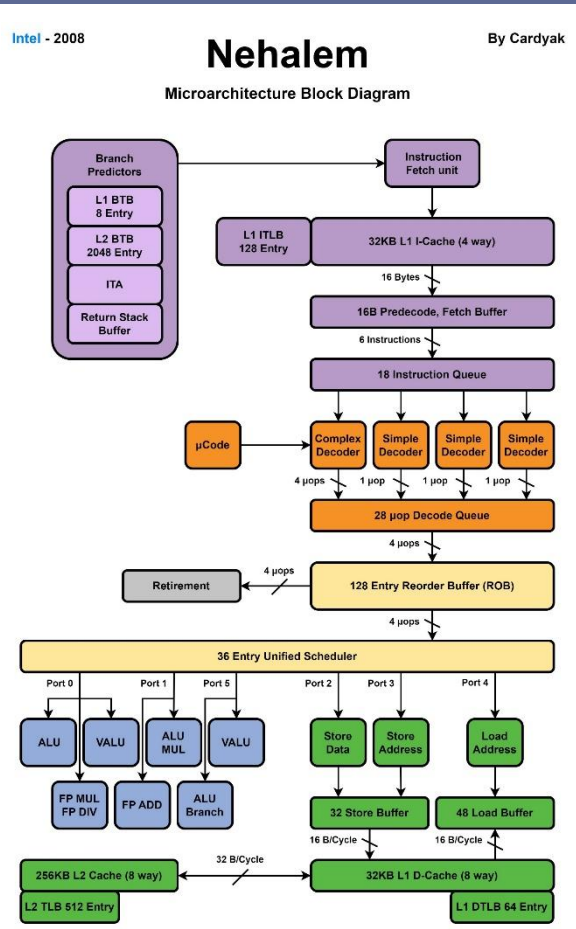
今日の話題

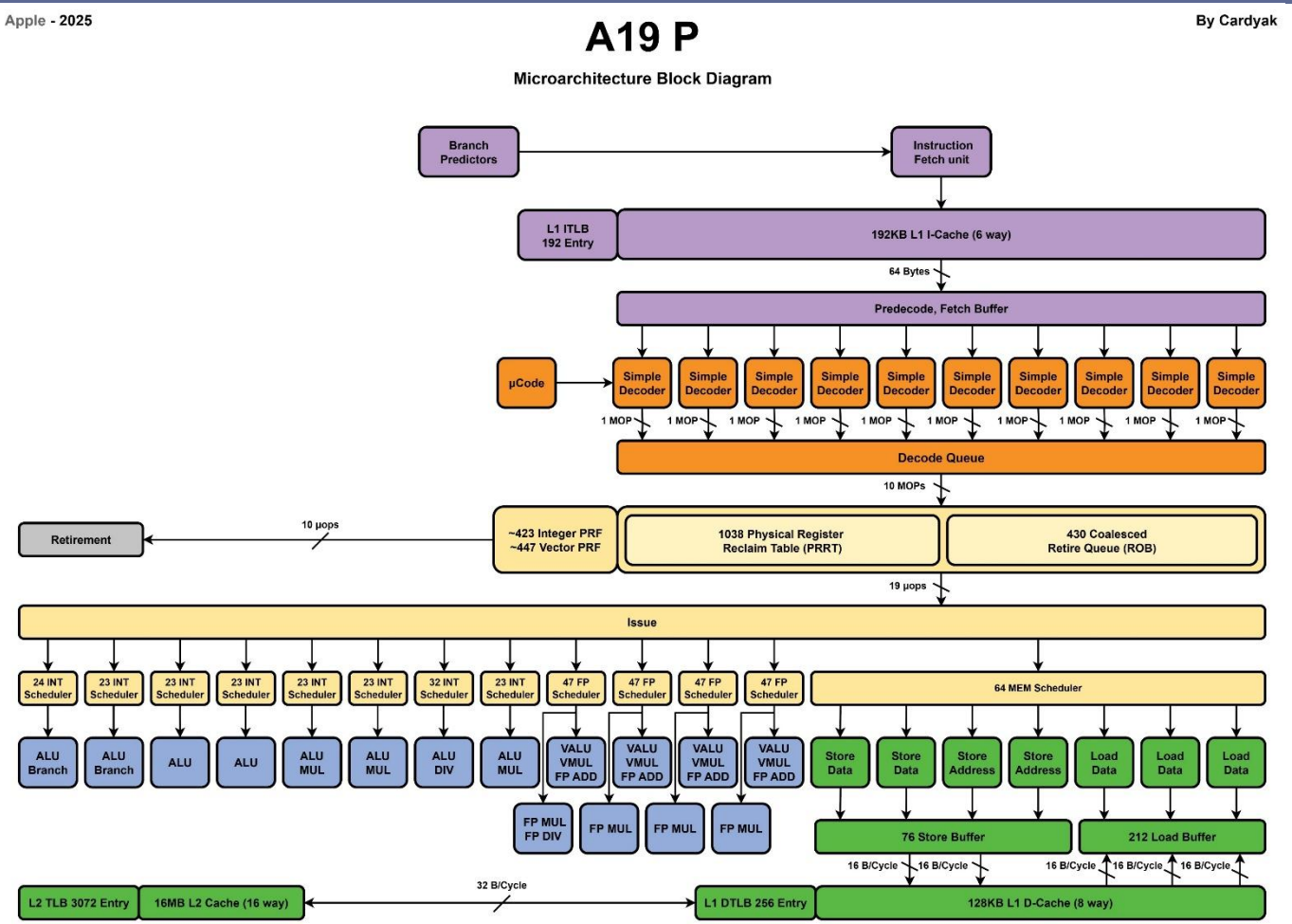
- 6th Championship Branch Prediction（CBP2025）が6月に開催
  - 分岐予測器のアルゴリズムを競う大会
    - 国際会議 ISCA に併設で開催
  - 提案する RUNLTS 予測器により，我々はこれに優勝
- 上記への参加の経験を通じて，以下のような話をしたい
  - どうやって短期間で成果を出すか？
  - どうやって経験の浅い学生が有意に貢献できるようにするか？

もくじ

- Championship Branch Prediction (CBP)
  - 現代の CPU と分岐予測器
  - CBP の歴史やルール
  - 提案した RUNLTS 予測器の概要
- 我々の取り組み：
  - 試行を確実かつ高速に回す環境
  - 可視化や解析のための環境
- 研究インフラを整える事が大事

背景：現代の CPU の大型化



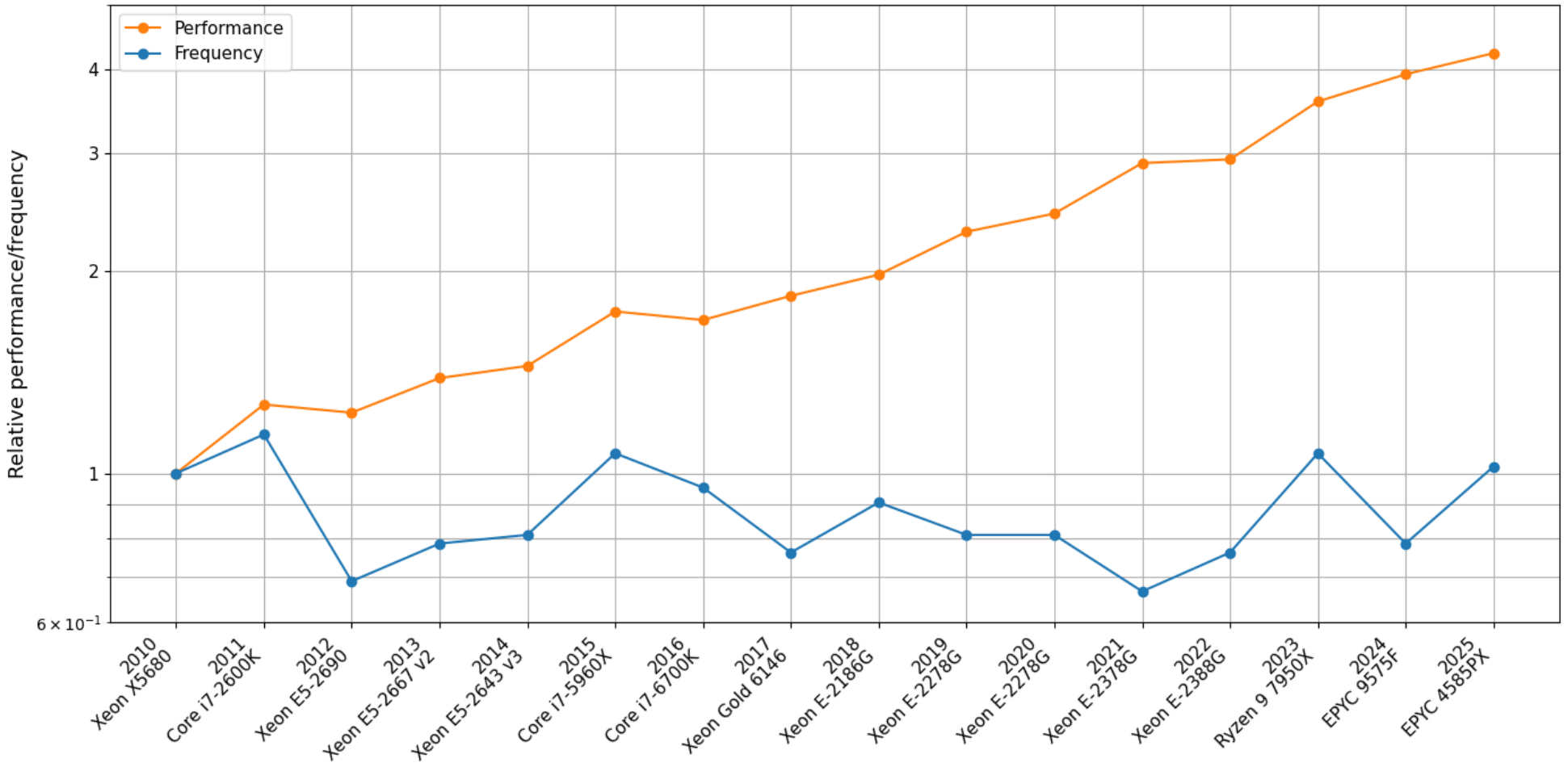


Intel Core2　(2008):
6 命令発行, 128 命令ウィンドウ

Apple A19 P (2025):
19命令発行, 1038 命令ウィンドウ（マイクロ命令）

図は Cardyak's Microarchitecture Cheat Sheet より
https://docs.google.com/spreadsheets/d/18ln8SKIGRK5_6NymgdB9oLbTJCFwx0iFI-vUs6WFyuE/edit?gid=1076260513#gid=1076260513

CPU のシングルスレッド性能は伸び続けている



- 上記は SPEC CPU 2006/2017 INTEGER の各年ごとの最高値と周波数（対数軸）
- 周波数は変わらないが，性能はコンスタントに向上（15年で4倍ぐらい）

過去15年の CPU の変化

- 周波数はあまり変わらないまま性能が向上し続けている
  - 発行幅や命令ウィンドウの大型化
    - 最新の CPU では 1000 命令以上のスケジューリングが可能
  - 様々なマイクロアーキテクチャ技術の導入
    - 分岐予測器の精度向上

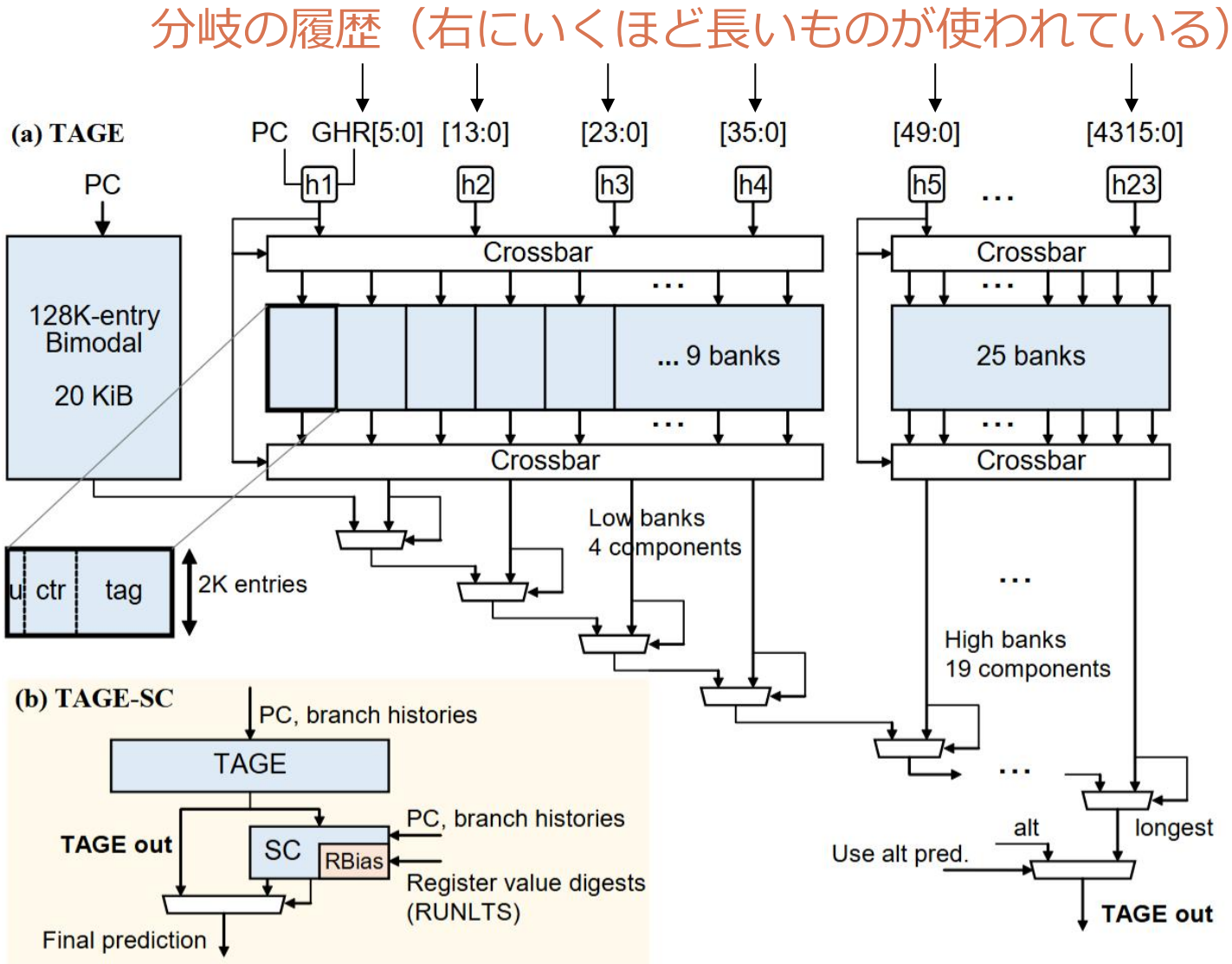
現代の分岐予測器

- 分岐予測器は消費電力面でも重要
  - 最新の予測器：1000 命令実行ごとに 3 回程度の予測ミスが起きる精度
  - 1000 命令実行ごとに 3000 命令が余計に実行され取り消されうる
    - （命令ウィンドウが 1000 命令で，最悪の場合

- 現代主流の予測器：TAgged Geometric history length (TAGE) Predictor
  - Andre Seznec 先生（INRIA，フランス）らにより提案
  - 近年の Intel, Apple, AMD, ARM, IBM などの高性能 CPU はみな搭載
    - この会場にあるほぼすべての PC やスマホはこれを搭載

TAGE の構造：トーナメント式に結合されたテーブル

- 基本：過去に実行された分岐結果の履歴に従って，次の分岐の方向を予測
- トーナメント式：
  - 幾何級数で伸ばした異なる履歴長で学習
  - 長い履歴を学習したテーブルの予測結果を優先
- TAGE-SC：
  - TAGE の出力を統計的補正器（SC）で補正
  - 現在最も強力と言われる



（RBias は後で述べる我々の提案

Championship Branch Prediction (CBP)

- TAGE などの分岐予測器の発展を牽引
  - 2004 年に初めて開催，2025 年で 9 年ぶり 6 回目の開催
- 第 2 回目以降ずーっと Seznec 先生が優勝
  - 各回で提案されたさまざまアイデアを TAGE に取り込み，強くなり続けた
- Seznec 先生が強すぎた
  - みんな萎えてしまい，しばらく開催されていなかった（個人の感想です）

CBP 2025 開催の動機

- ARM が主催：
  - 委員も産業界の人が多い
  - ARM, Apple, AMD, Intel…
- 産業界の人達の動機：
  - 産業界は分岐予測について
    大きな課題に直面している
  - 学術界との協業こそ重要であり
    分岐予測への関心を戻したい

Organizing Committee	Program Committee
Rami Sheikh (ARM), Chair	Alaa Alameldeen (Simon Fraser University)
Saransh Jain (ARM)	Muawya Al-Otoom (Apple)
	Saransh Jain (ARM)
	Gabriel Loh (AMD)
	Pierre Michaud (Inria)
	Eric Rotenberg (NCSU), Chair
	Rami Sheikh (ARM)
	Dam Sunwoo (ARM)
	Chris Wilkerson (Intel)

CBP 2025のルール

- おおよそ次のルールのもとで実施：
  - CPU のシミュレータが与えられ，ここに予測器を実装して競う
  - 使用可能な記憶容量は合計 192 KiB（過去最大
    - 命令キャッシュより大きい予測器は，現代では割と普通
  - 事前に与えられたトレースは合計 105 個
    - compress，fp，infra，int，media，web の 6 区分にカテゴライズ
    - 本番測定では未知のトレースも使用される

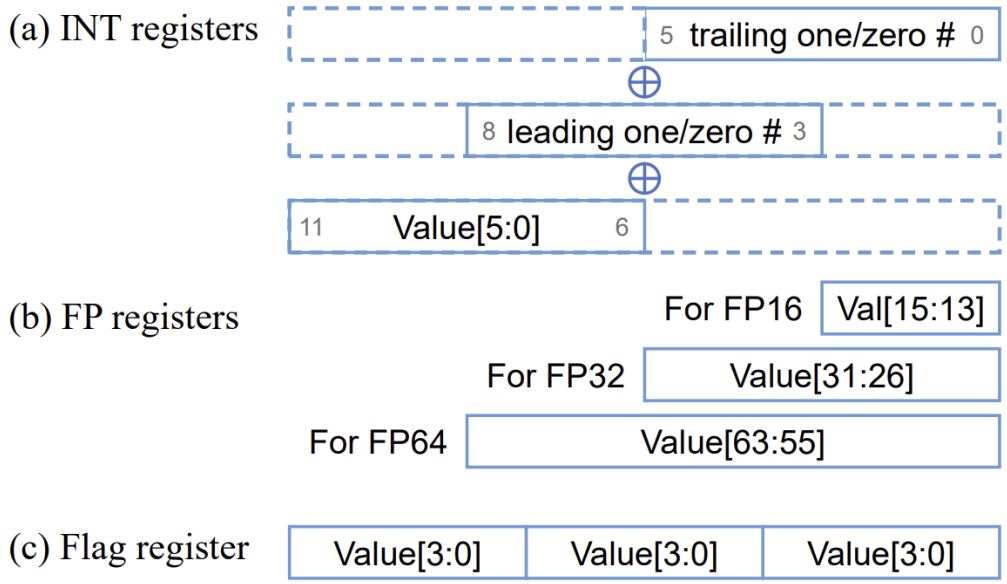
提案：RUNLTS 予測器

- TAGE-SC 予測器をベース
  - TAGE 部分の改良：
    - 幾何級数ではない履歴長セットを導入
    - エントリの確保量を動的に制御
  - 統計的補正器（SC）部分の改良：
    - 従来はブラックボックスとして扱われがちだった
      - SC はパーセプトロン予測器
    - どういう時になにが有効かを解析し，改良調整
- RBias 機構の導入：
  - レジスタ値と分岐方向の相関を直接捉える新しい方法を導入

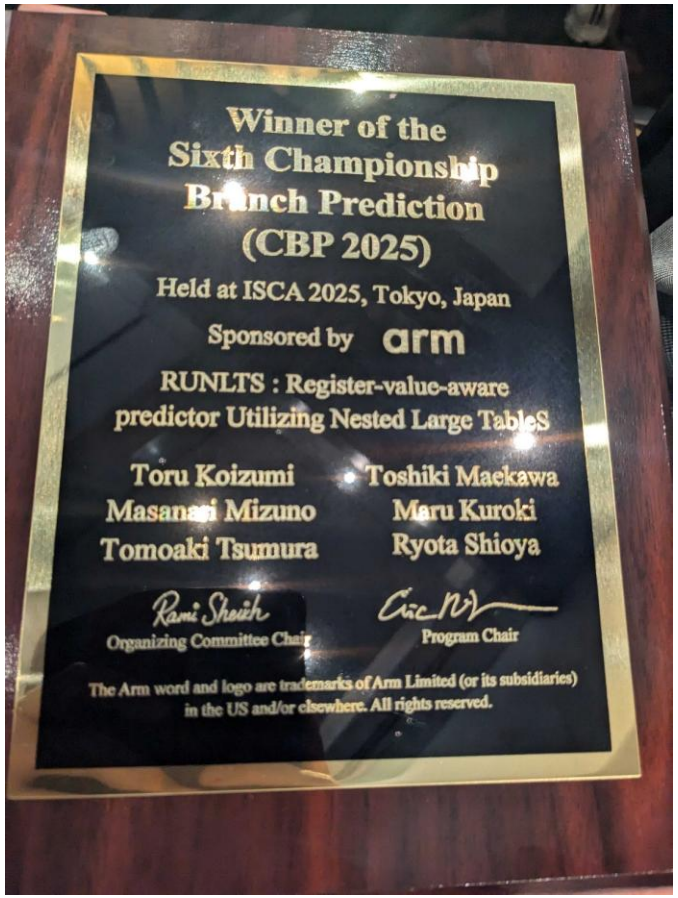
RBias の概要

- レジスタ値からダイジェストを生成し，分岐方向との相関を拾う
  - 最下位ビット列，最上位/最下位の連続した 0/1 の数，
    FP の符号や指数部などをエンコード
  - 小さい値やアラインされたポインタなどの特徴を捉える
  - これをパーセプトロン予測器にいれて分岐方向との相関を拾う

- 実行結果のレジスタ値とか予測時には見えないのでは？
  - 発見：分岐予測ミスはバーストで起きがち
  - １回フラッシュが走ると，それより古い値がみえるので２個目以降が救える
  - 予測ミス時の実行結果からも相関を拾えて未来視みたいな事までできる



CBP 2025 優勝

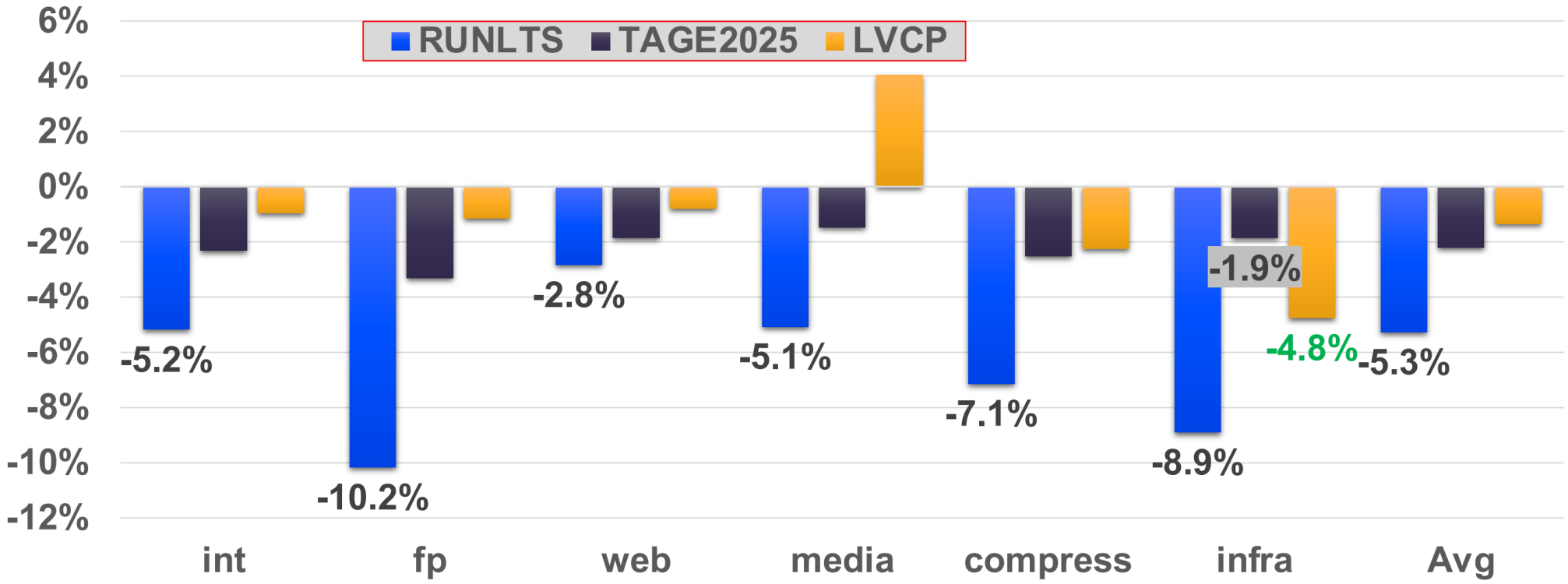


- 特に RBias の貢献が大きい
  - ノイズまみれの中から実際に有効な相関を拾うのは容易ではなかった
  - これを実現した小泉先生はえらい

大きく差をつけて勝った（我々の予測器が青色の棒）

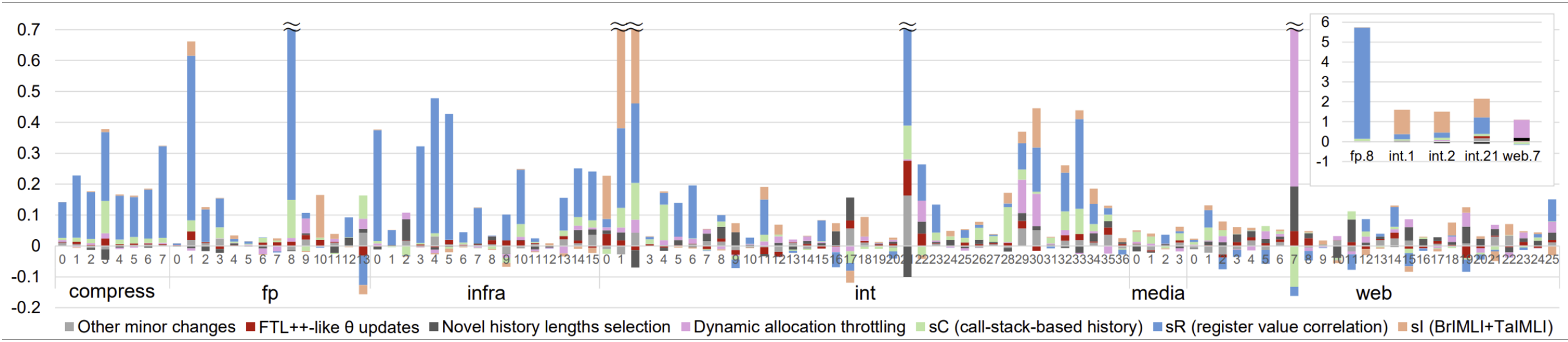
グラフは CBP2025 Closing Remarks, Results, and Awards より

Results – Top-3 Per Workload Category (BrMisPKI, Full)



グラフはベースラインの 192KiB TAGE-SC-L 予測器からのミス数削減率
BrMisPKI は 1000 命令あたりミス数
グラフ上で下にあるほど良い

提案予測器における要素ごとの精度向上の寄与



- 縦軸は1000命令あたりの予測ミス数の削減量
  - （１ページ前とは逆に，グラフ上で上にあるほど良い
- RBias（青色）は，幅広いワークロードで精度を大きく向上している

ここまでのまとめ

- よりよい分岐予測を目指して CBP 2025 が開催
  - 分岐予測器はいまでも超重要
    - CPU は大型化しており，消費電力面でもインパクトが大きい
    - TAGE ベースの予測器が使われている
  - 産業界からの分岐予測やろうぜの呼びかけ
- RUNLTS 予測器を提案して優勝
  - 開催期間 3ヶ月ぐらいで良い成果が出た

本題：どうやって成果をだすか？
インフラの整備が大事

過去のチャンピオンシップへの参加

- The Third Data Prefetch Championship（DPC3）2019
  - 良いアイデアが出た
    - 「あえてプリフェッチを遅らせる」
  - チューニング不足で惨敗
- The First Instruction Prefetch Championship（IPC1）2020
  - 良いアイデアが出た
    - 「関数リターンの連続回数を使ったハッシュ」
  - チューニングも出来たので入賞

動機１：良いアイデアが出れば勝てる！・・・が，

- それが出来れば苦労はしない
  - アイデアがでるかは基本的には運ゲー
- 良いアイデアがでる確率をなんとかして上げる：
  1. 当たるまでたくさんガチャを回す
     - 分析と試行を可能な限り高速に回す
  2. 可視化による分析が有効
     - めんどくさいので，あまり皆やらない
     - 知られてない特性に気づく可能性が高い

動機２：チューニングに関する反省

- チャンピオンシップ終盤はこうなりがち
  - 評価データが再現できない：
    - 実装やパラメータが適切に保存されていない
    - いろいろ実験してるうちに，何でとったのかわからなくなる
  - 「なんで数字があわないの？！」「うるせー！」みたいなやりとりがおきてツラい
- 確実かつ高速に評価を行う環境がいる

動機３：時間と人手がたりない

- CBP2025 は開催アナウンスから提出締切まで 3 ヶ月間
  - 分析や実装，チューニングなど無数にやることがある
  - チャンピオンシップ古参兵の博士の学生さん達はみんな卒業した
    - 新しい学生さん達に手伝ってほしい
    - M1 × 2 人 + M2 × 1 人 が参加

学生さんのよくある状況

- ４月ごろ，研究室配属後にとりあえずシミュレータを動かしてみる
  - 定例ミーティング：
    - １週目：「コンパイルが通らないです」 … コンパイルオプションがおかしい
    - ２週目：「変なエラーがでるんです」 … ライブラリが不足でリンク失敗
    - ３週目：「性能がなぜか悪化します」 … パラメータがおかしい
    - …
    - 季節はめぐり，暑くなって肌寒くなり，年の瀬も迫ってくる
- これでは戦場にたどり着くころには戦が終わってしまう
  - 有効な実験をすぐ開始できる環境整備がいる

これまでの経験に基づく取り組み

- インフラ整備を頑張った：
  - すぐ開始できる実験環境
    - とにかくコマンド一発で立ち上がる環境を整備
  - 試行（ガチャ）を高速かつ確実に回す環境の整備
    - シミュレーションの高速化
    - 実験実行ツール Art2
  - 可視化などの解析環境の整備

1. すぐ開始できる実験環境の整備

- 実験準備の手順：
  - リポジトリのクローン
  - その中の launch.sh を実行 → 全てが揃った仮想環境が立ち上がる
  - （トレース入力のダウンロードスクリプトを実行
- 実験の手順：
  - シミュレータの make
  - art2 コマンド（後述）の実行

環境の実装方法

- シェルスクリプトや Makefile，Dockerfile を気合で整備する
  - リポジトリから pull した後に，必要なリビルドやインストールがきちんと走るように
  - とにかくユーザーに更新を意識させない
- Dev Container に近い
  - より依存が少ない形になっていると思う

自動で一貫した環境が作れることは重要

1. 気兼ねなくライブラリを導入できる
   - 後述する最適化やツールは多くのライブラリに依存
   - 依存を満たした環境を維持するのは結構ハードルが高い
     - 特に経験が浅い学生さんはトラブル解消に時間がかかる

2. 実験結果を一貫させられると検証が容易に
   - コンパイラやライブラリのバージョン違いにより，わずかにプログラムの挙動が変わってしまう場合が結構ある
   - 小数点以下がなんかずれるとか

3. 新規環境を気軽に立ち上げられる
   - ちょっとノート PC でも簡単な解析をやる，みたいな事がすぐできる

インフラの整備

1. すぐ開始できる実験環境
   - コマンド一発で立ち上がる環境を整備
2. 高速かつ確実に試行を回す環境
   - シミュレーションの高速化
   - 実験実行基盤ツール art2
3. 可視化などの解析環境の整備

学生さんのよくある状況 その２

- シミュレータも動いたので，いろいろ分析をしてみる
  - 定例ミーティング：
    - １週目：「～の値を振って測ってみようか」
    - ２週目：「まだ測定が終わってないです．もうちょっとです」
    - ３週目：「データとれましたが微妙でした」
      「いやそこはそうじゃなくて，～を変えてみようか」
    - ４週目：「ちょっとスクリプトがバグってて～」
    - ５週目：「今週は就活が忙しくて～」
    - …
  - 季節はまためぐり，もう１回ぐらい暑くなって寒くなってくる

- これではやっぱり戦が終わってしまう
  - 高速かつ確実に測定や集計を行える環境がいる

シミュレータを高速化する

- CBP で提供されているシミュレータ：
  - C++ で記述されている
  - プログラムの実行トレースを読みながらシミュレーションを行う
  - CPU パイプライン全体や 3 レベルのキャッシュまでシミュレーション
    - 分岐予測器のみを再現するシミュレータと比べると遅い
- Perf でのボトルネック解析と最適化を実施

シミュレータの最適化

- トレース圧縮形式の変更：
  - gz の展開がボトルネックだったので，ZStandard に変更した
    - gz よりずっと高い圧縮率でかつ３～４倍程度高速
  - 圧縮フォーマット変換も自動で走るように
- メモリアロケータの置換：
  - 小さなオブジェクトの頻繁な生成・破棄がボトルネック
  - jemalloc を採用
- 内部のシーケンシャル検索の置換：
  - ハッシュテーブルに置き換えなど
- デバッグ用文字列生成の抑制：
  - 文字列生成が裏で常に走っていたのがわかったので，無効化した

最適化の結果

- それとは別に，分岐予測のシミュレーションだけを行う簡易版を実装
- 結果：全トレースの実行時間 @ AMD EPYC 7713 (64core) x2
  - オリジナル：　　9分
  - 最適化版：　　2分（4.5倍速）
  - 簡易版：　　35秒（15倍速）
- ちゃんとプロファイラを使って観察するのは重要
  - 気づいてしまえば，しょうもない点で重くなっている事が多い

最適化の意味

- 35秒でワンショットの実験ができる
  - なにかソースを書き変えて様子を見る，がかなり高速に回せる
  - 35秒は集中力が切れない
    - 5分待つだと，ネットを見に行って1時間戻ってこないとかになりがち
- パラメータスイープ時のスループットが大きく上がる
  - 一晩で試せる量が段違い

インフラの整備

1. すぐ開始できる実験環境
   - コマンド一発で立ち上がる環境を整備
2. 高速かつ確実に試行を回す環境
   - シミュレーションの高速化
   - 実験実行ツール art2
3. 可視化などの解析環境の整備

動機：シミュレーションによる評価環境

- 典型例：
  - シェルスクリプト等からシミュレータを起動
  - 結果を別のスクリプトから集計・・・みたいなものをアドホックに作る
- 問題：
  - コピペやコメントアウトの繰り返しで破綻しやすい
  - 複数パラメータを振った組合せを柔軟に記述しにくい
    - 毎回大がかりな手作業が必要になる
  - 既存結果の再現が困難になりがち
    - スクリプトのソースコードの変更が適切に保存されない事が多い
    - 突貫修正により，既に得た数値を再現できないような事故が多発

実験実行ツール Art2

- 目的
  - 再現性の担保
  - 作業の一元化
- art2 コマンド
  - シミュレーション実行，結果の集計，グラフ生成などを一貫して実行する CLI
  - Git のようなサブコマンドで各操作を行う
  - タブ補完等の機能も充実

```
$ art2 init                # セッション雛形の生成
art2cfg.jsonc has been created.
...

$ vim art2cfg.jsonc        # セッション情報の修正
...

$ art2 run                 # 実行
Setting up a session 0001 (1/3) 'Test-ALL-BPrNmTAGE..'
...
--- Queuing jobs (backend: parallel) ...
Total jobs to execute: 315
Executing job (1/315): /cbp2025/our_work/test/result/
0001/job/int_19_trace.exec.sh ...
Executing job (2/315): /cbp2025/our_work/test/result/
0001/job/int_17_trace.exec.sh ...
...

$ art2 stat        # 集計
$ art2 graph       # グラフ生成
```

Art2 の設定ファイル

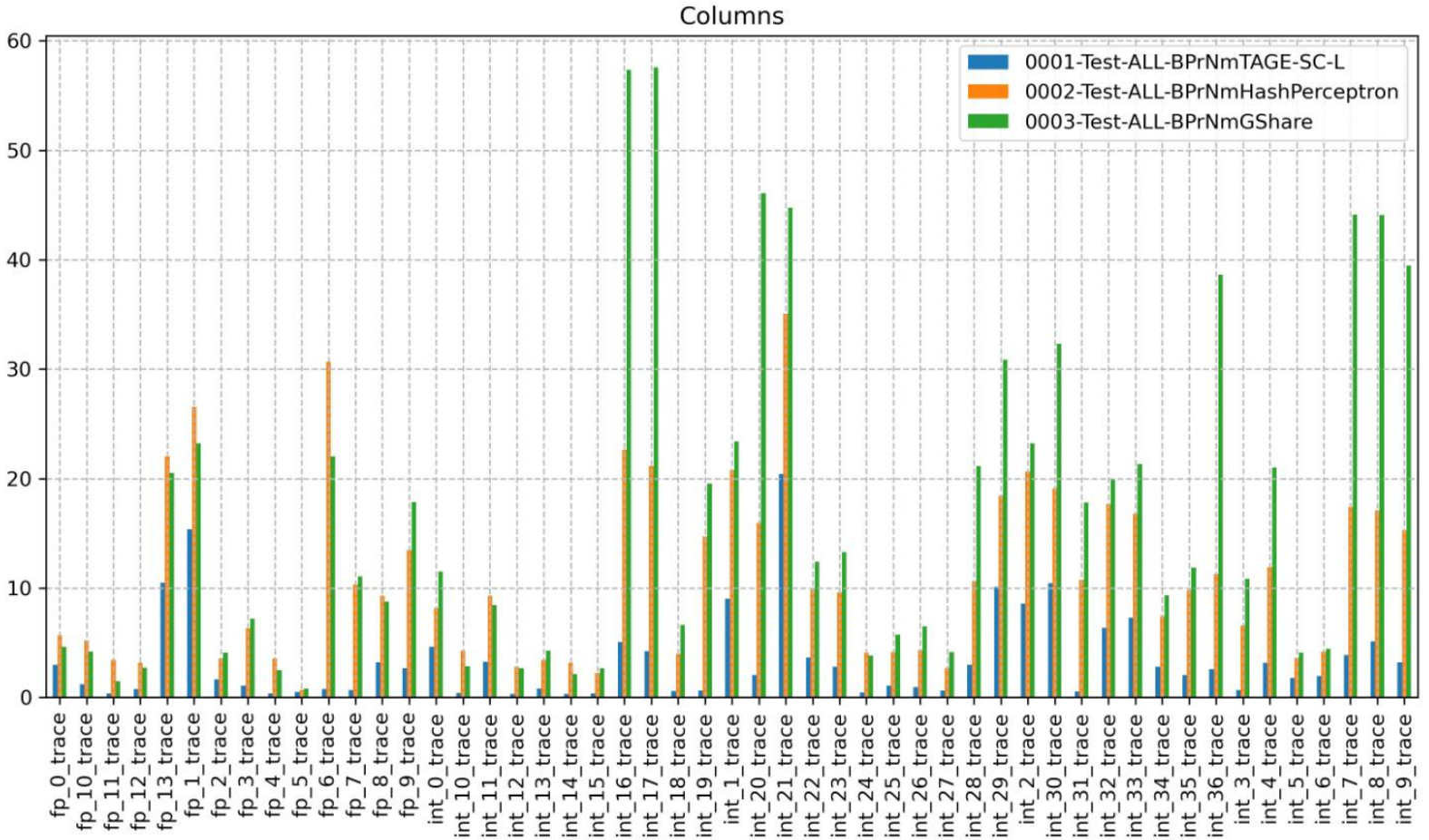
- パラメータなどを宣言的に記述
  - 範囲を振って指定なども柔軟に可能
  - 右の例では，予測器を 3 種類に振って全トレースを評価
- ローカルマシン or クラスタで並列実行
  - slurm ジョブスケジューラも使用可能

```
{ // art2cfg.jsonc
  // 実行バイナリとジョブスケジューラバックエンドの指定
  "binary_file": "$(ROOT_DIR_PATH)/cbp",
  "backend": "parallel", // serial|parallel|slurm

  // 指定したパスの .zst がつくファイルを入力トレースに選択
  "input_file_patterns": [
    { "path": "$(TP)/int/", "pattern": ".+\\.zst" },
    { "path": "$(TP)/fp/",  "pattern": ".+\\.zst" }
  ],

  // セッション設定例
  "sessions": [{
    "name": "Test-ALL",
    // TAGE-SC-L,HashPerceptron,GShare を実行
    "parameters": [
      { "entry": "BranchPredictorName",
        "range": "TAGE-SC-L,HashPerceptron,GShare" }
    ]
  }]
}
```

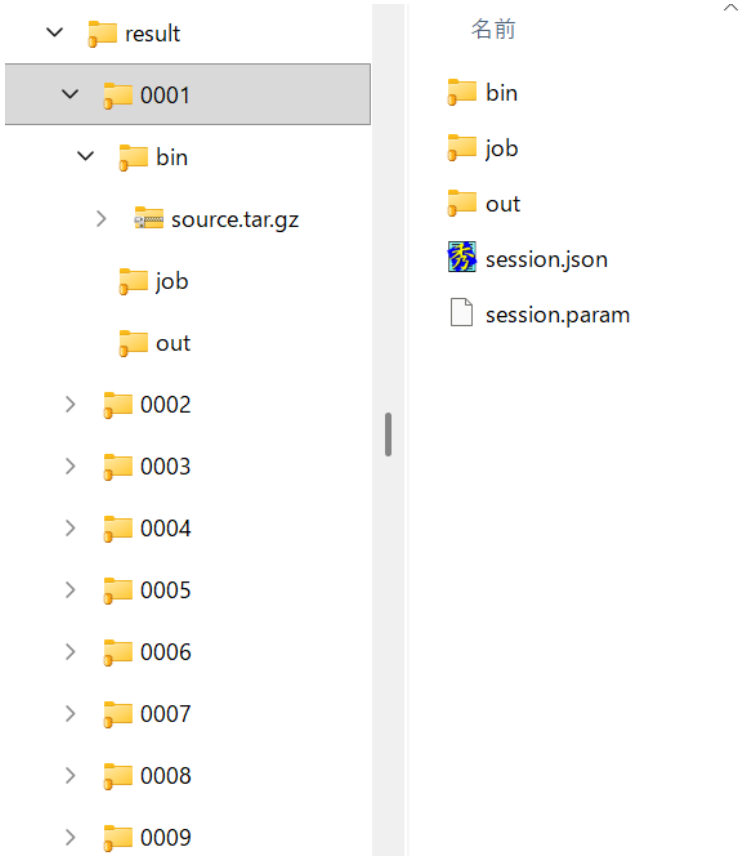
結果の集計やグラフの生成



- シミュレータ出力のパースと CSV への集計，
  ミス数の平均数やグラフの生成をコマンド一発で実行可能

実験環境の保持機能

- 測定のたびに測定環境をすべてコミットしておけばよい
  … が，みんな面倒なのでやらない
  - ちょっとソースを書き換えて実験，とかやりがち
  - 実験が再現できない
- Art2 は評価セッションごとに，パラメータや
  ソースコード，バイナリの完全なコピーを保存
  - 同一の結果を再現可能
- ソースコード差分を表示する機能も整備
  - 何が結果の差を生んでいたかなどを容易に追跡可能
- 特定用途向けの軽量な git のような機能



Art2 の効果

- ほとんど全ての実験の状況をカバーできた
- 実験結果の再現に起因する問題は完全に抑え込めた
- ソースコード差分表示は意外と活躍した

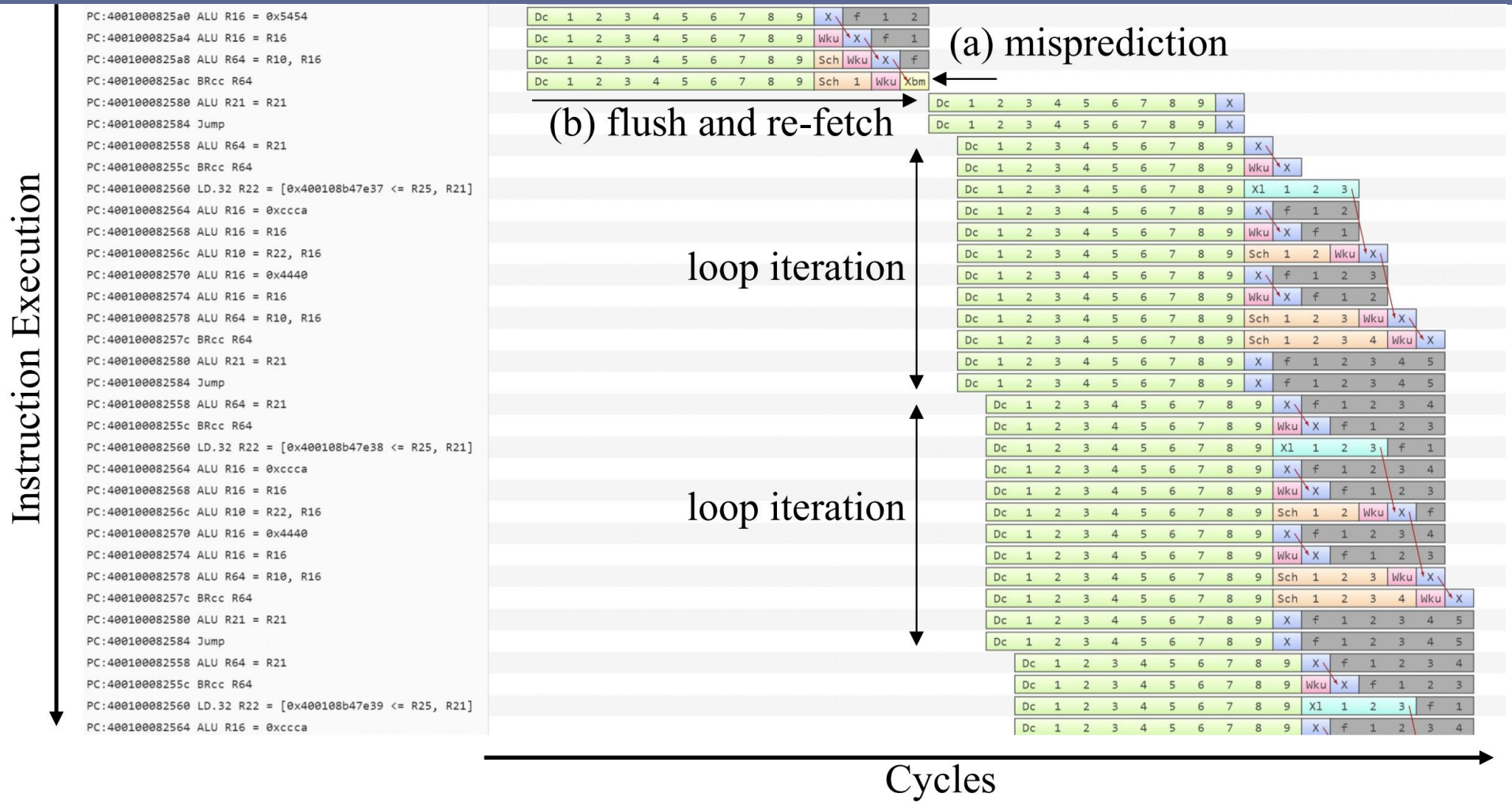
インフラの整備

1. すぐ開始できる実験環境
   - コマンド一発で立ち上がる環境を整備
2. 高速かつ確実に試行を回す環境
   - シミュレーションの高速化
   - 実験実行基盤ツール art2
3. 可視化などの解析環境の整備

可視化と解析

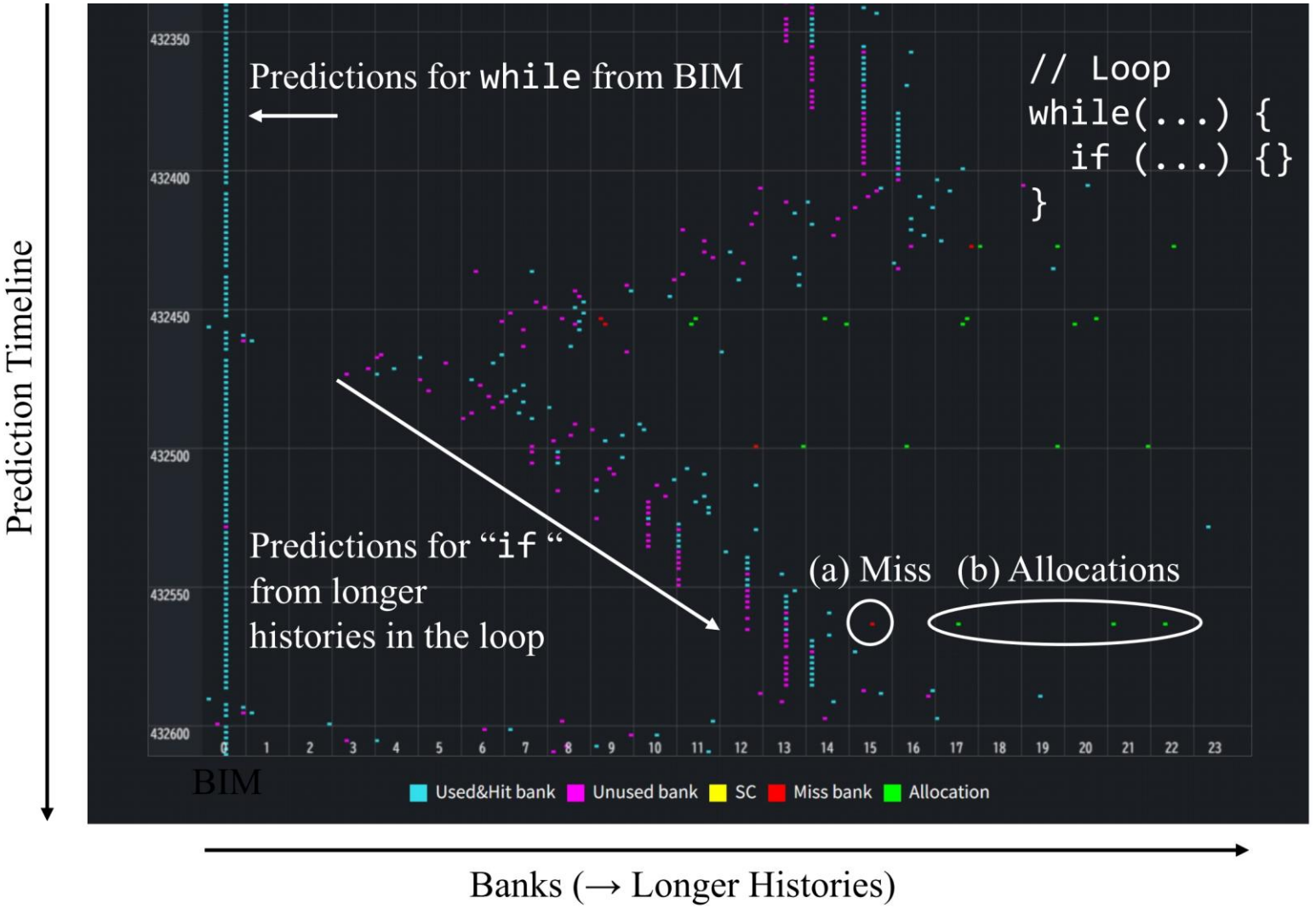
- さまざまツールを開発
  - Konata：パイプライン可視化
  - Sazanami：予測器のテーブル可視化
  - トレースの文字出力
  - トレースからの逆アセンブル生成

Konata によるパイプライン可視化



- https://github.com/shioyadan/Konata
- プロセッサ内で何が起きているのかを掴むのに広範囲に活躍
- まずは Konata トレースの出力をシミュレータに実装するのが我々の定石に

Sazanami による予測器テーブルアクセスの可視化



- TAGE 予測器内で何が起きているのかを掴むのに活躍
- https://github.com/shioyadan/sazanami2

文字列アクセスをダンプするツール

- バイトアクセスを行うロード命令の値をダンプ
  - 文字列処理の様子がとれる
- 文字列からトレースの元のプログラムがわかる事がある
  - SunSpider, Speedometer, SPECjbb

文字列アクセスのダンプ例

- ある web カテゴリのトレースにおいて，なぜか DNA ぽい文字列が現れる
- SunSpider（JavaScript ベンチマーク）の regexp-dna だと判明

regexp-dna のソースコード

入力の DNA 文字列

```
47  var l;
48  var dnaInput = ">ONE Homo sapiens alu\n\
49  GGCCGGGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGGCGGA\n\
50  TCACCTGAGGTCAGGAGTTCGAGACCAGCCTGGCCAACATGGTGAAACCCCGTCTCTACT\n\
51  AAAAATACAAAAATTAGCCGGGCGTGGTGGCGCGCGCCTGTAATCCCAGCTACTCGGGAG\n\
52  GCTGAGGCAGGAGAATCGCTTGAACCCGGGAGGCGGAGGTTGCAGTGAGCCGAGATCGCG\n\
53  CCACTGCACTCCAGCCTGGGCGACAGAGCGAGACTCCGTCTCAAAAAGGCCGGGCGCGGT\n\
54  GGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGGCGGATCACCTGAGGTCA\n\
55  GGAGTTCGAGACCAGCCTGGCCAACATGGTGAAACCCCGTCTCTACTAAAAATACAAAAA\n\
56  TTAGCCGGGCGTGGTGGCGCGCGCCTGTAATCCCAGCTACTCGGGAGGCTGAGGCAGGAG\n\
57  AATCGCTTGAACCCGGGAGGCGGAGGTTGCAGTGAGCCGAGATCGCGCCACTGCACTCCA\n\
58  GCCTGGGCGACAGAGCGAGACTCCGTCTCAAAAAGGCCGGGCGCGGTGGCTCACGCCTGT\n\
59  AATCCCAGCACTTTGGGAGGCCGAGGCGGGCGGATCACCTGAGGTCAGGAGTTCGAGACC\n\
60  AGCCTGGCCAACATGGTGAAACCCCGTCTCTACTAAAAATACAAAAATTAGCCGGGCGTG\n\
61  GTGGCGCGCGCCTGTAATCCCAGCTACTCGGGAGGCTGAGGCAGGAGAATCGCTTGAACC\n\
62  CGGGAGGCGGAGGTTGCAGTGAGCCGAGATCGCGCCACTGCACTCCAGCCTGGGCGACAG\n\
63  AGCGAGACTCCGTCTCAAAAAGGCCGGGCGCGGTGGCTCACGCCTGTAATCCCAGCACTT\n\
64  TGGGAGGCCGAGGCGGGCGGATCACCTGAGGTCAGGAGTTCGAGACCAGCCTGGCCAACA\n\
65  TGGTGAAACCCCGTCTCTACTAAAAATACAAAAATTAGCCGGGCGTGGTGGCGCGCGCCT\n\
66  GTAATCCCAGCTACTCGGGAGGCTGAGGCAGGAGAATCGCTTGAACCCGGGAGGCGGAGG\n\
67  TTGCAGTGAGCCGAGATCGCGCCACTGCACTCCAGCCTGGGCGACAGAGCGAGACTCCGT\n\
68  CTCAAAAAGGCCGGGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG\n\
69  CGGGCGGATCACCTGAGGTCAGGAGTTCGAGACCAGCCTGGCCAACATGGTGAAACCCCG\n\
70  TCTCTACTAAAAATACAAAAATTAGCCGGGCGTGGTGGCGCGCGCCTGTAATCCCAGCTA\n\
71  CTCGGGAGGCTGAGGCAGGAGAATCGCTTGAACCCGGGAGGCGGAGGTTGCAGTGAGCCG\n\
72  AGATCGCGCCACTGCACTCCAGCCTGGGCGACAGAGCGAGACTCCGTCTCAAAAAGGCCG\n\
73  GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGGCGGATCACC\n\
74  TGAGGTCAGGAGTTCGAGACCAGCCTGGCCAACATGGTGAAACCCCGTCTCTACTAAAAA\n\
75  TACAAAAATTAGCCGGGCGTGGTGGCGCGCGCCTGTAATCCCAGCTACTCGGGAGGCTGA\n\
```

データ 3 倍水増し

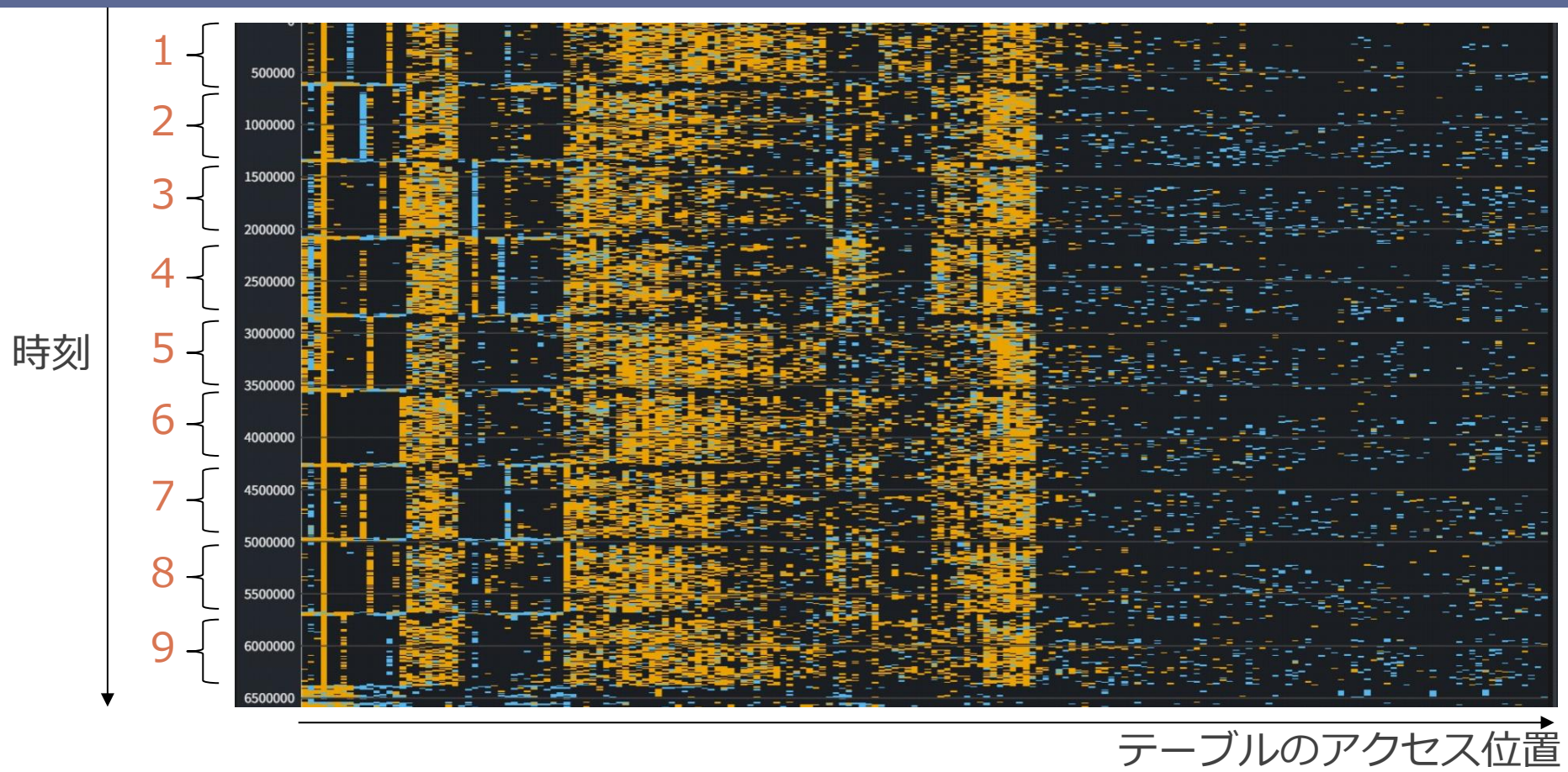
```
1720  dnaInput = dnaInput + dnaInput + dnaInput;
1721
1722  var ilen, clen,
1723   seqs = [
1724    /agggtaaa|tttaccct/ig,
1725    /[cgt]gggtaaa|tttaccc[acg]/ig,
1726    /a[act]ggtaaa|tttacc[agt]t/ig,
1727    /ag[act]gtaaa|tttac[agt]ct/ig,
1728    /agg[act]taaa|ttta[agt]cct/ig,
1729    /aggg[acg]aaa|ttt[cgt]ccct/ig,
1730    /agggt[cgt]aa|tt[acg]accct/ig,
1731    /agggta[cgt]a|t[acg]taccct/ig,
1732    /agggtaa[cgt]|[acg]ttaccct/ig],
1733   subs = {
1734    B: '(c|g|t)', D: '(a|g|t)', H: '(a|c|t)', K: '(g|t)',
1735    M: '(a|c)', N: '(a|c|g|t)', R: '(a|g)', S: '(c|t)',
1736    V: '(a|c|g)', W: '(a|t)', Y: '(c|t)' }
```

9 個の正規表現

順々にマッチをかけている

```
1746  for(i in seqs)
1747      dnaOutputString += seqs[i].source + " " + (dnaInput.match(seqs[i]) || []).length + "\n";
1748  // match returns null if no matches, so replace with empty
```

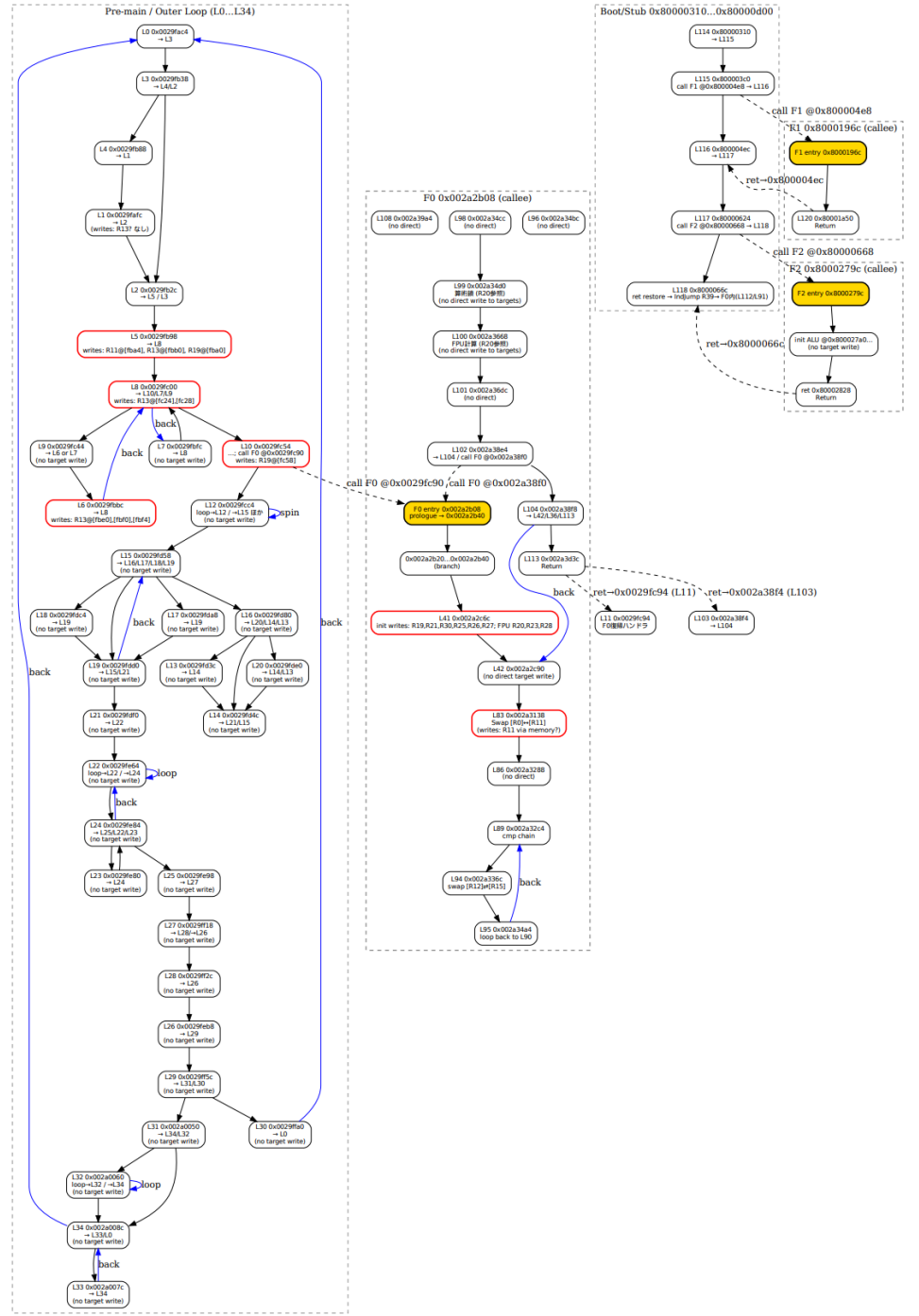
可視化結果とあわせて予測器の挙動の説明がつく



- 全体が９個のフェーズに分かれる（９個の正規表現マッチに対応
- 各フェーズは途中から予測が凄い当たる
  - 文字列が３倍水増しされてるので，途中から同じパターンが繰り返される
  - このトレースでは１回見たものを素早く憶えて正確に繰り返すのが重要

アセンブリコードや CFG の復元ツール

```
6802 L124:    ; from:L126(008acec4)↓
6803      008ace80:    LD.32 R0 = [R24]              ; id:0 exec:122↓
6804      008ace84:    LD.32 R1 = [R25]              ; id:0 exec:122↓
6805      008ace88:    [CMP] ALU R64 = R0, R1        ; id:0 exec:122↓
6806      008ace8c:    BRcc R64    -> <unknown>        ; id:0 exec:122 taken:0 untaken:122↓
6807      008ace90:    [SUB] ALU R2 = R1, R0         ; id:0 exec:122↓
6808      008ace94:    LD.64 R1 = [R29]              ; id:0 exec:122↓
6809      008ace98:    ALU R2 = R2                   ; id:0 exec:122↓
6810      008ace9c:    LD.32 R23 = [R29]             ; id:0 exec:122↓
6811      008acea0:    ALU R28 = R2                  ; id:0 exec:122↓
6812      008acea4:    [ADD] ALU R3 = R1, R21        ; id:0 exec:122↓
6813      008acea8:    [ADD] ALU R0 = R3, R0         ; id:0 exec:122↓
6814      008aceac:    ALU R3 = R27, R0              ; id:0 exec:122↓
6815 L125:    ; from:L126(008acec4)↓
6816      008aceb0:    ALU R0 = R3                   ; id:0 exec:3416↓
6817      008aceb4:    ALU R2 = R28                  ; id:0 exec:3416↓
6818      008aceb8:    ALU R1 = 0x0                  ; id:0 exec:3416↓
6819      008acebc:    ALU R23 = R23                 ; id:0 exec:3416↓
6820      008acec0:    Call -> F0                    ; id:0 exec:3416 target:F0(004002c0)↓
6821 L126:    ; from:L172(01134f80)↓
6822      008acec4:    [ADD] ALU R3 = R0, R22        ; id:0 exec:3416↓
6823      008acec8:    [CMP] ALU R64 = R23, R19      ; id:0 exec:3416↓
6824      008acecc:    BRcc R64    -> L125             ; id:0 exec:3416 taken:3294 untaken:122 target:L125(008aceb0)↓
6825      008aced0:    LD.64 R0 = [R29]              ; id:0 exec:122↓
6826      008aced4:    ALU R20 = R20                 ; id:0 exec:122↓
6827      008aced8:    [ADD] ALU R21 = R21, R0       ; id:0 exec:122↓
6828      008acedc:    [CMP] ALU R64 = R20, R26      ; id:0 exec:122↓
6829      008acee0:    BRcc R64    -> L124             ; id:0 exec:122 taken:120 untaken:2 target:L124(008ace80)↓
6830      008acee4:    ALU R23 = R0                  ; id:0 exec:2↓
6831      008acee8:    LD.64 R26 = [R29]             ; id:0 exec:2↓
6832      008aceec:    ST.64 [R29] = R29, R27        ; id:0 exec:2↓
```



- プログラムの構造がわかると，予測器の挙動が説明がつく

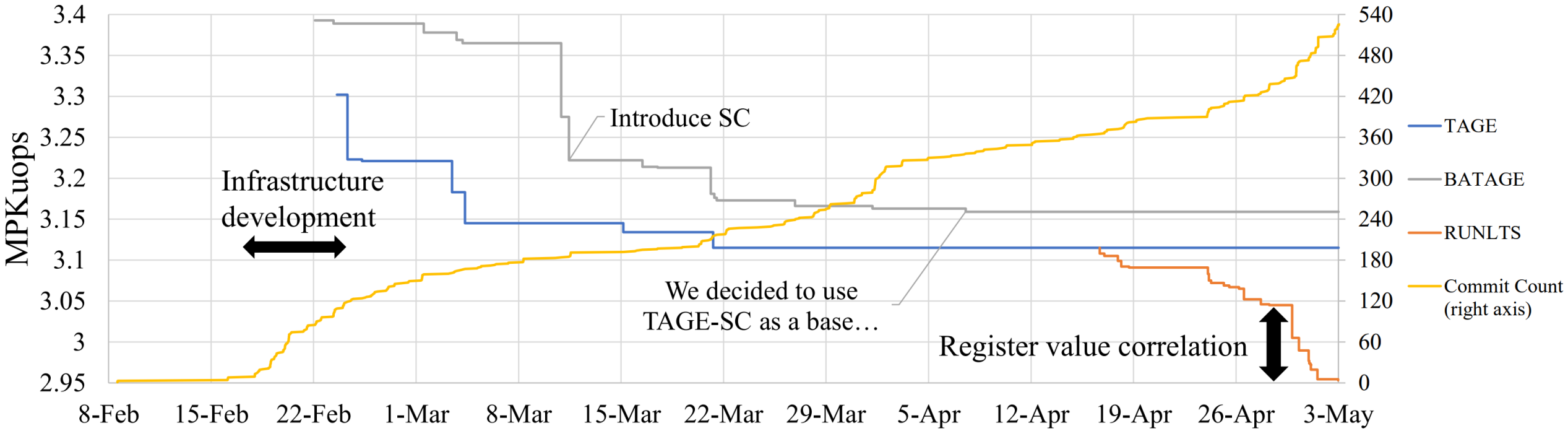
ツールの開発

- Python や JavaScript (JS)/TypeScript(TS) で作成
  - グラフ作成は Python (Jupyter+DataFrame+matplotlib)
  - 巨大トレースファイルの解析は JS（Node.js）
    - C 言語の数分の 1 ぐらいの速度で動く
  - インタラクティブな可視化ツールは JS/TS で HTML 上に構築
    - 最近の WEB ブラウザはなんでもできる
      - ログ解析等まで含めて，全てをブラウザ上で処理
    - GB クラスの大きさの入力ファイルも扱える

LLM が活躍

- 可視化ツールの作成やグラフ作成には極めて強力
  - 自力でやると 1 週間はかかるようなツールがすぐ作れる
  - matplotlib でヒートマップ出力 → 「HTML に移植してインタラクティブに操作できるようにして」とかもできる
  - アセンブリからの逆コンパイルみたいな作業も得意
- 反面，予測器本体やアイデア出しにはほとんど貢献していない

予測器のミス数（左側軸，低いほどよい）と，リポジトリへのコミット数の推移



- 最初の三週ぐらいはインフラの開発が多い（黄色の立ち上がり）
- ベースラインのチューニングでかなり良くなっている（青色の遷移）
- 最も重要な提案は，締め切り数日前に実装成功（右端オレンジが急に落ち込む）
  - その後の数日で適切に改良が進んだのも，実験インフラ整備のおかげ

まとめ

- ポイント：
  - 高速かつ確実に試行を回し，いろいろ解析もやって，
    有効なアイデアがでる確率をなんとかして上げる
  - 不慣れな人でも有意な貢献ができるように環境を整える

- やったこと：インフラ整備
  1. すぐ開始できる実験環境
     - とにかくコマンド一発で立ち上がる環境を整備
  2. 高速かつ確実に試行を回す環境
     - シミュレーションの高速化
     - 実験実行ツール art2
  3. 可視化などの解析環境の整備
     - HTML ベースの可視化は非常に強力

まとめ２

- ３ヶ月で得られたもの
  - （たぶん）トップ会議に通る提案
  - 博論を書けるぐらいのボリュームの知見や新しい提案
    - １００ページでは足りない
- 学生さんの貢献もとても大きい
  - さまざまな実装や解析をしてもらった
- 学生さんの研究で，教員側でこのような環境整備ができると良い
  - しかし，ここまでやるのは無理
  - 出来る範囲で学生さんが各々インフラを整えるのはとても有効だと思う
    - 手作業を減らしつつ回転を上げよう，可視化してみよう
    - ただし整備にのめり込んで研究本体が進まなくならないように注意

ふろく

うまく働く例

```
1: X1 = …
2: R1 = f(X1);              // ながい関数呼び出し
3: if (R1 > 0) {            // ここの分岐予測が外れてパイプラインフラッシュ
4:     if (X1 < 0.0) …      // ここの予測時には X1 がフロントエンドで見える
5: }
```

- ４行目の `if (X1<0.0)` において，レジスタ X1 の値と分岐方向の相関を直接学習
  - X1 のダイジェストは符号をエンコードしているので，相関を学習できる
- 予測ミスはバーストで起きがち
  - １回ミスが起きると，パイプラインがフラッシュされて予測からやりなおし
  - ＝それより前の命令の実行結果が見えてる状態で次の予測ができる
    - ３行目の if で予測ミスが起きると４行目で X1 の値がつかえる