

高性能 CPU の研究・開発動向 (2022/11/07)

塩谷 亮太

東京大学 大学院 情報理工学系研究科

創造情報学専攻

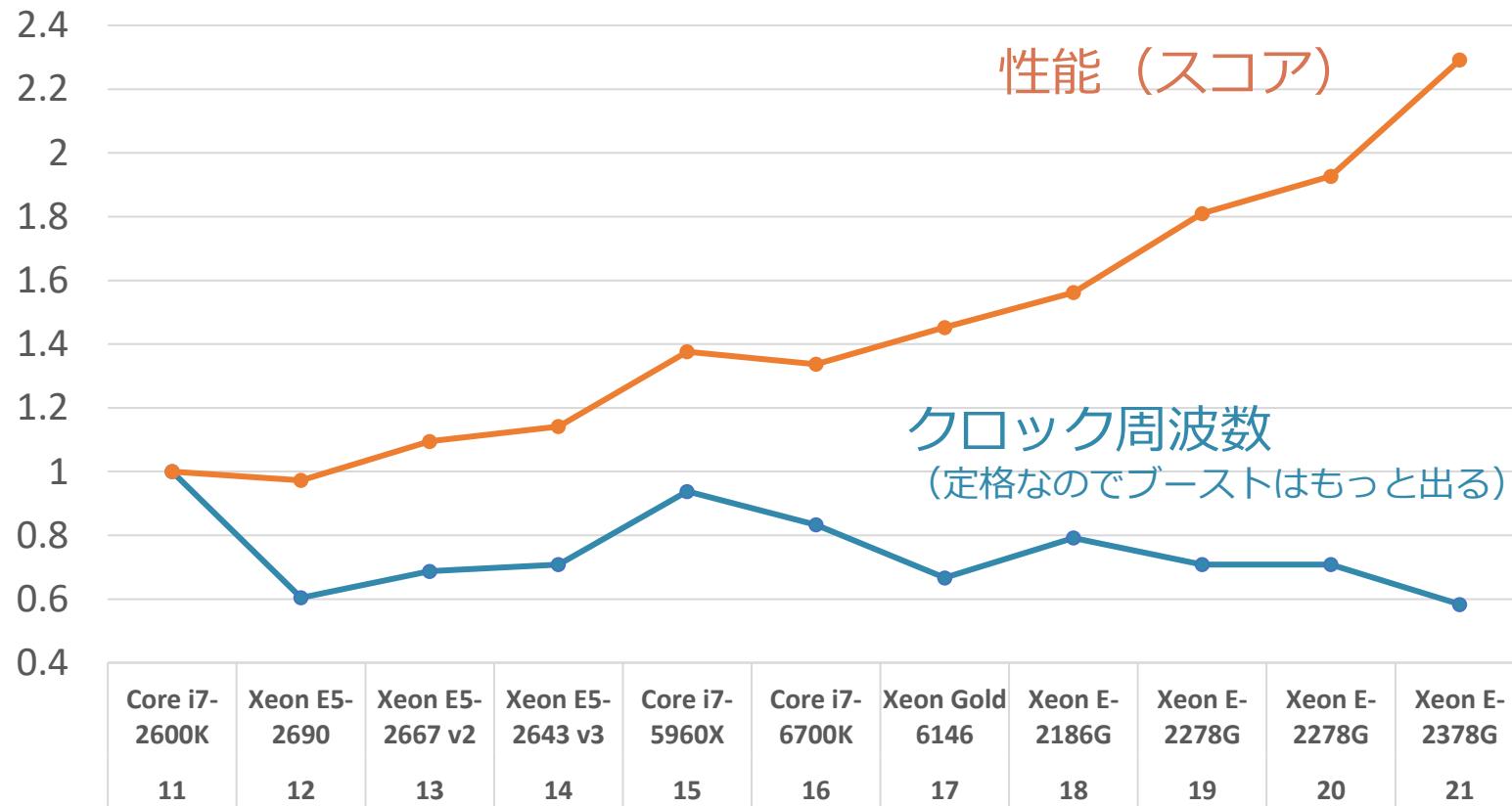
shioya@ci.i.u-tokyo.ac.jp

はじめに

- 塩谷の専門：
 - ◊ コンピュータ・アーキテクチャ
 - 特に CPU や GPU のマイクロアーキテクチャ
 - ◊ 最近は言語処理系やセキュリティに関わる研究も多いです
- 導入の話題：
 - ◊ 「CPU の性能 → シングルスレッド性能」は最近どうなっているのか？

過去10年のシングルスレッド性能の遷移

各年ごとの SPEC CPU int speed base の最高スコア (2006, 2017 を補正して結合)



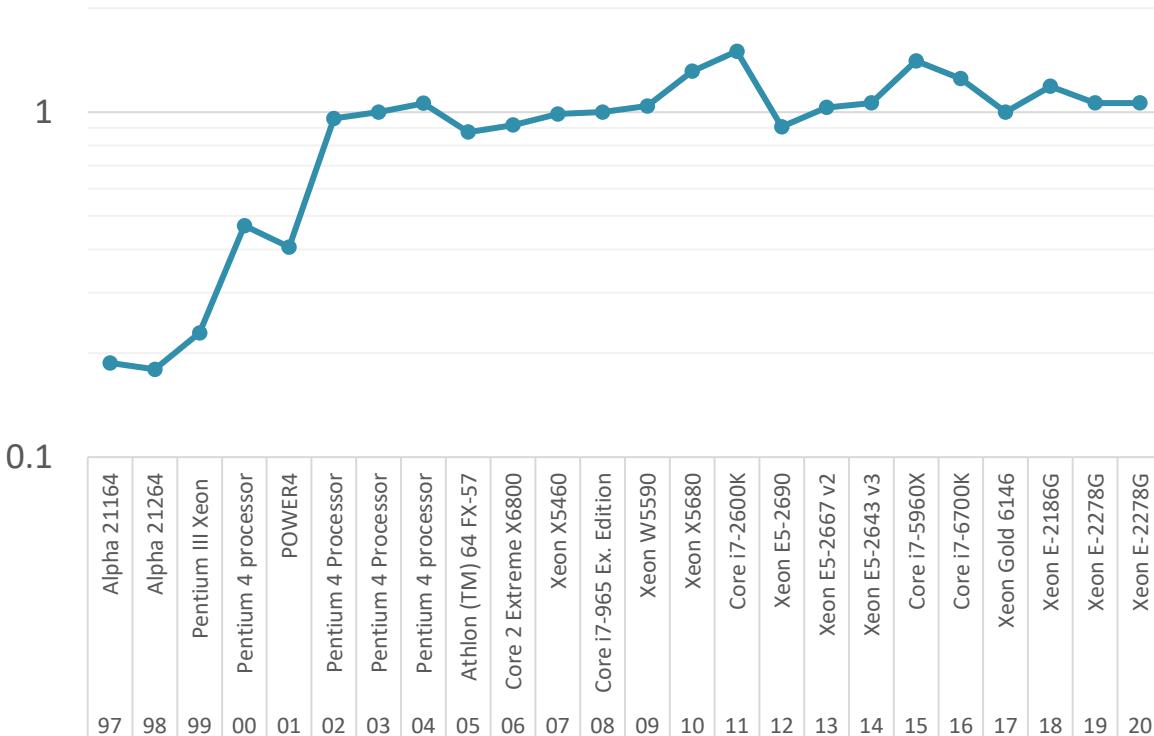
- 直近5年ぐらいは毎年11%程度の性能向上
- 以前に比べるとペースは落ちてるが、コンスタントに伸びている

クロック周波数とシングルスレッド性能

- クロック周波数は10年前からあまり上がっていない
 - ◊ 半導体の動作電圧をほぼ下げられなくなってしまったため
 - ◊ (最近はまた少し上がっててきた
 - 定格 3GHz 程度, ブースト時で 4GHz ぐらいが典型的
 - CPU によっては最大で 5GHz 程度で稼働
 - ◊ 現代の CPU はクロック周波数の向上に（あまり）頼らず性能を上げている
- 背景として、クロック周波数の変遷について簡単に紹介

クロック周波数

- CPU のクロック周波数：
 - ◇ 1秒間に何回処理を行えるかを表す
 - ◇ 性能を大きく左右
- 2002年頃からほぼ上がっていない



周波数向上がストップ

- なぜ？ → 電圧が下げられなくなつたから
 - ◊ 消費エネルギーの壁にぶつかった
- 話題：
 1. 半導体のスケーリング
 2. CPU の消費エネルギーとは何なのか
 3. ダークシリコン問題

ムーアの法則

- 「半導体の集積度は3年ごとに4倍になる」
 - ◇ トランジスタのサイズを1/2に縮小（スケーリング）
面積： $1/2 \times 1/2 = 1/4$

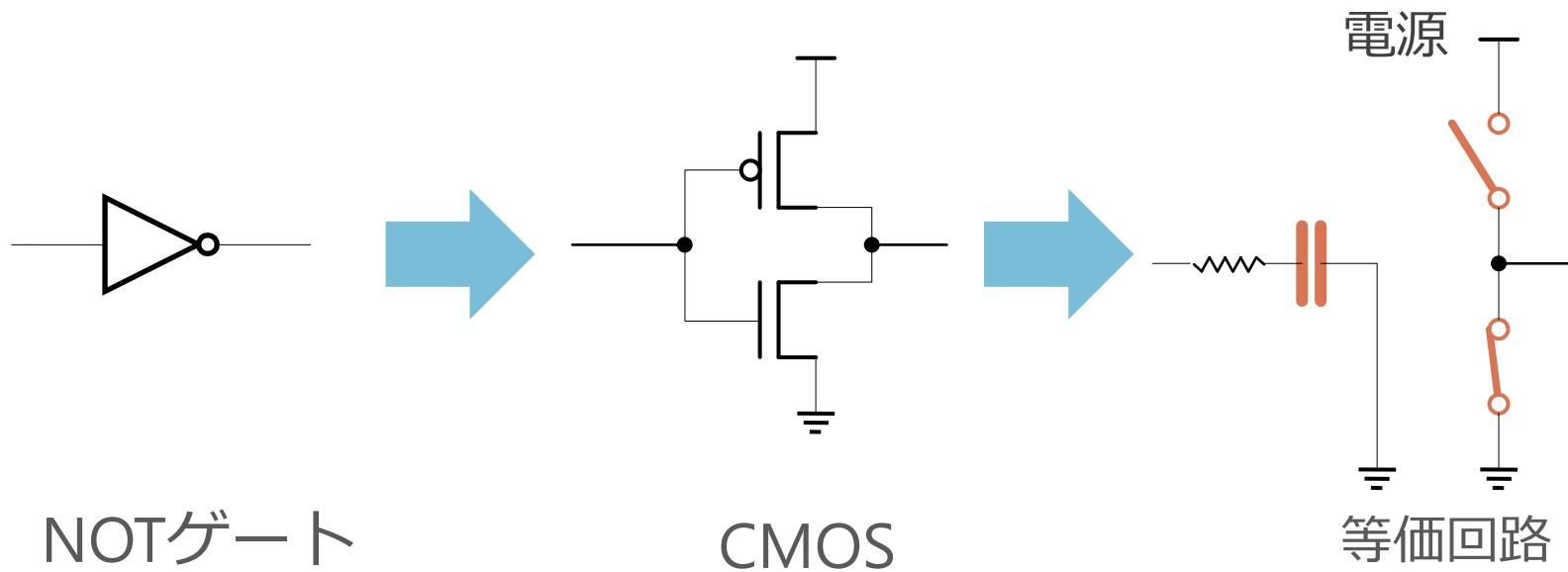


- すごいけど、数が増えるだけなの？

スケーリングの効果

- トランジスタあたりで、
 - ◊ 消費エネルギーは $1/8$ に
 - ◊ 遅延も $1/2$ に
- 等価回路を使って説明

CMOS ゲートの等価回路



- コンデンサと、連動したスイッチによって表せる
 - ◊ 充電：下のスイッチがON
 - ◊ 放電：上のスイッチがON
- CPU の計算で消費されるエネルギー：
 - ◊ スイッチ ON/OFF するためのコンデンサへの充放電

消費エネルギーと遅延

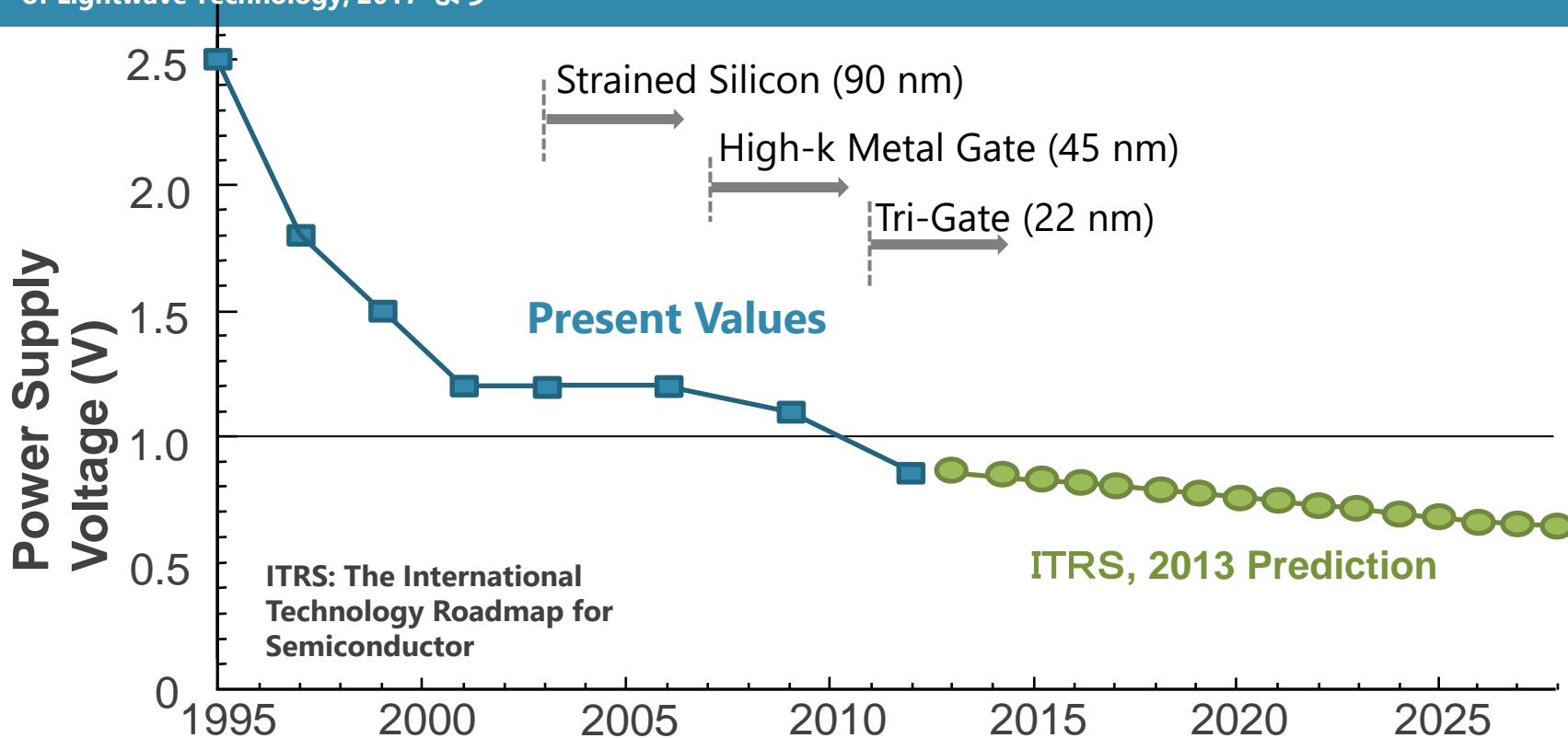
- コンデンサへの充電に必要なエネルギー
 - ◊ $\frac{1}{2} CV^2$
- サイズと電圧を $1/K$ 倍にスケーリングした場合
 - ◊ $C:$ $1/K$ (面積 $1/K^2$, 厚さ $1/K$ より)
 - ◊ $V:$ $1/K$
 - ◊ エネルギー : $1/K^3$
- C が小さくなり充電にかかる時間が減るので, 遅延も $1/K$ に
 - ◊ 結果として, 周波数は K 倍に

チップ全体の消費エネルギー

- $1/K$ 倍にスケーリングした場合, チップ全体では,
 - ◊ 個々のトランジスタのエネルギー : $1/K^3$
 - ◊ トランジスタの個数 : K^2
 - ◊ 周波数 (動作回数) : K
 - ◊ 全体の消費エネルギー : 1
- まとめると, スケーリングによって
 - ◊ 同じ大きさのチップに搭載できる回路量が増えるのはもちろん,
 - ◊ 消費エネルギーは一定のまま,
 - ◊ 周波数もすごい向上 !

ところが、電圧が下げられなくなった...

グラフは K. Sato : , Realization and Application of Large-scale Fast Optical Circuit Switch for Data Center Networking, IEEE Journal of Lightwave Technology, 2017 より



- 2000 年過ぎから、電圧低下がほぼとまった
 - ◊ 電圧が下がりすぎて、ON / OFF の境界の閾値が 0V 付近に
 - ◊ OFF にしたいときも、閾値との距離がとれない
 - ◊ 微妙に ON になり電流が漏れてしまう（リーク電流という）

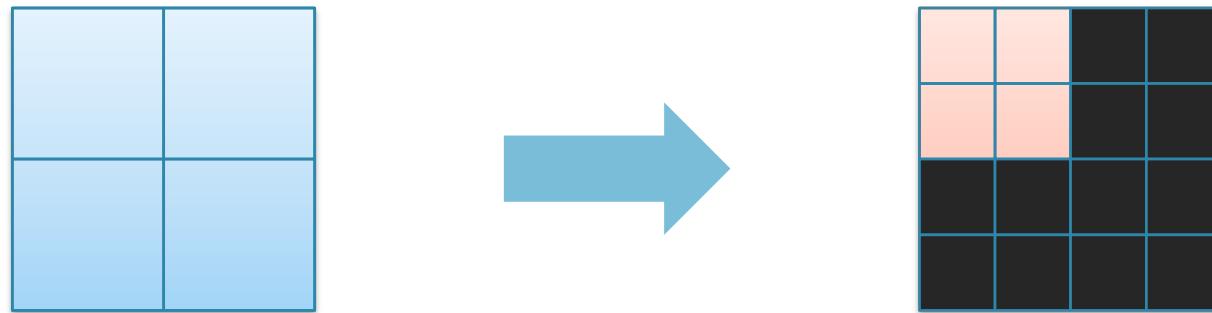
電圧が下がらないとどうなるのか

- $1/K$ 倍にスケーリングした場合
 - ◊ 回路面積: $1/K^2 \rightarrow 1/K^2$
- トランジスタ 1 個の 1 回のスイッチにかかるエネルギー
 - ◊ C: $1/K \rightarrow 1/K$
 - ◊ V: $1/K \rightarrow 1$
 - ◊ エネルギー : $1/K^3 \rightarrow 1/K$ ($E = \frac{1}{2}CV^2$ より)
- トランジスタ 1 個の単位時間あたりのエネルギー
 - ◊ スイッチ 1 回 : $1/K^3 \rightarrow 1/K$
 - ◊ 周波数 (動作回数) : $K \rightarrow K$
 - ◊ エネルギー : $1/K^2 \rightarrow 1$
- トランジスタは小さくなったのに、エネルギーが減ってない！

行き着く先：ダークシリコン

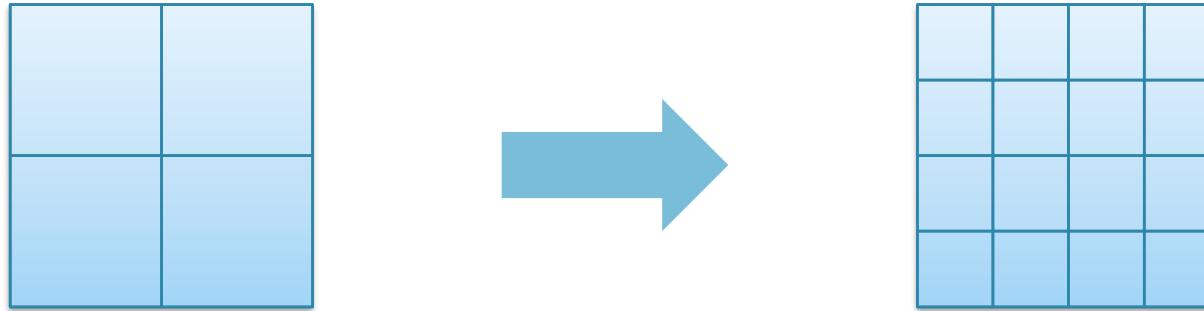
[Goulding10, Esmaeilzadeh12]

- トランジスタは小さくなったのに、エネルギーが減らない！
 - ◊ サイズだけ小さくして、電圧を下げていないから
- 電力密度があがってしまう
 - ◊ スケール前のチップ全体と、スケール後の赤い部分は同じエネルギーを消費



- チップへの電力供給はもう限界で、これ以上増やせない
 - ◊ 電源ピンに流せる電流量や冷却能力の限界
 - ◊ 使えない（ダーク）シリコンが...

とはいえ、いまのところ



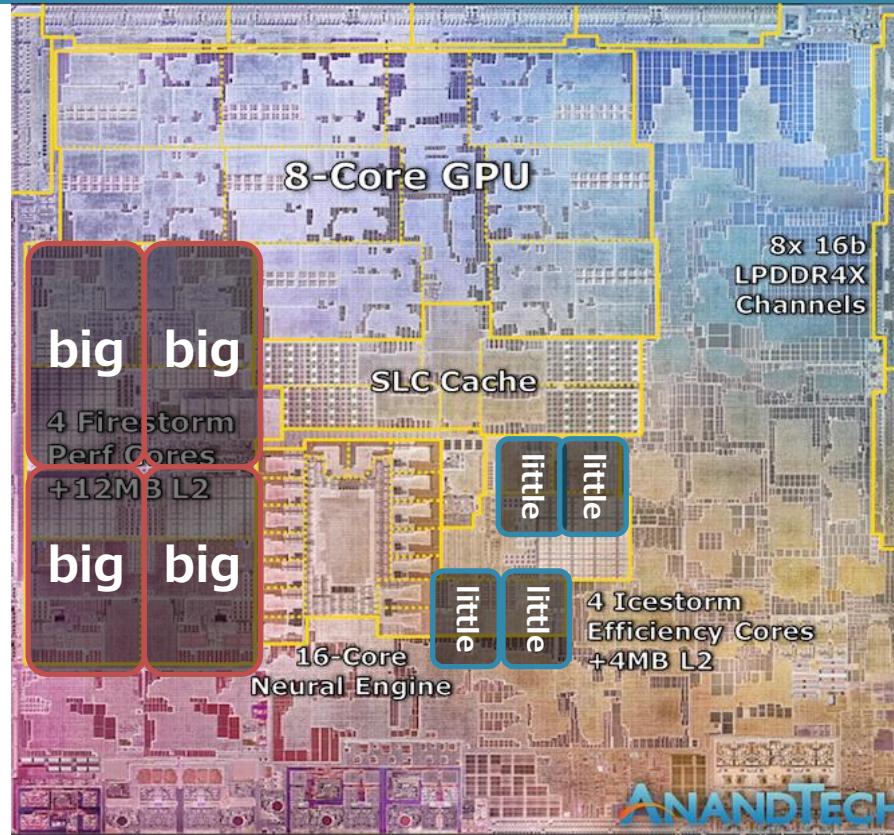
- まだ、ダークにまでは至っていない
 - ◊ デバイス技術が進歩
 - ◊ 電圧が一応ちろちろ下がってはいる
- 周波数を上げなくした
 - ◊ 動作回数が減るので、電力にダイレクトに効く
- まとめ：電圧が下げられなくなったので、電力密度増大をおさえるために周波数を上げるのをやめた

クロック周波数とシングルスレッド性能

- クロック周波数は10年前からあまり上がっていない
 - ◊ 半導体の動作電圧をあまり下げられなくなってしまったため
- 性能向上の要因：マイクロアーキテクチャの改善
 1. より大きく複雑な回路を導入
 - レジスタ1つあたりで見ると小さくなり続いている
 - 全部を同時に動かさなければ平気
 2. 方式自体の論理的な改良

Apple M1

写真は <https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive> より



- ◊ Apple M1（2020年11月発売の MacBook Pro に搭載）の big core は現時点で最も（論理的に）大きな CPU の1つ
- ◊ チップ面積の大部分は GPU や様々な専用回路に割かれている
 - big コアが少々大きくなつてももはや問題にならない

シングルスレッド性能向上の方法

- アプリケーションごとにそれぞれ異なる：
 1. HPC アプリケーション：
 - 例：スーパーコンピュータで実行されている科学技術計算
 2. 整数系アプリケーション
 - 例：PC やスマホで実行されているアプリ
- スーパーコンピュータはPC/スマホの上位互換ではない
 - ◊ それぞれで求められる性質が大分違う

HPC アプリケーション（1）

- HPC アプリケーション：スループット指向
 - ◊ アプリケーション自体に自明な並列性が豊富に含まれる
 - ◊ プログラムが（相対的には）単純である一方、データ量が多い
- 例：スーパーコンピュータのランキング TOP500 で使われる HPL というベンチマークの場合
 - ◊ 計算の大部分は浮動小数点の密行列積
- CPU コアの性能は（相対的には）理論最大性能の影響を大きく受ける
 - ◊ プログラムが単純なので人手で限界まで最適化される
 - ◊ コアごとの演算器の数 × 駆動周波数、メモリバンド幅

HPC アプリケーション（2）

- 「CPU コア」の性能向上：
 - ◊ レイテンシが多少伸びても良いからスループットを増やす
 - 浮動小数点演算器の個数（あるいは SIMD の幅）
 - キャッシュやメモリのバンド幅
- なるべく演算器をたくさん積んで、メモリからデータをたくさん転送できる口をつける
 - ◊ 人手の最適化を阻害するので、あまりハードウェアには自動で動いて欲しくない
 - ◊ 投機実行などは好まれない

整数系アプリケーション（1）

■ 整数系アプリケーション：レイテンシ指向

- ◊ 簡単に取り出せる並列性はあまりない
- ◊ コードは複雑である一方、データの量は（相対的に）小さい

■ 理論最大性能よりも全く低い性能しかでない

- ◊ プログラムの挙動が複雑すぎて人手でも並列化できない
 - 複雑な制御フローや依存関係
- ◊ 典型的には、理論最大性能の数分の $1 \sim 1/10$ もでない
 - 単純にピーク性能を増やすことは意味が無い

整数系アプリケーション（2）

- コアごとの実効性能は、ハードによる動的な並列処理次第
 - ◊ どうやって高効率に並列性を抽出するか
 - ◊ どうやって投機実行やプリフェッチをするか
- 整数演算器の数を増やすことは相対的には容易
 - ◊ 整数演算器は浮動小数点演算器よりも大幅に小さい
 - INT 64bit ADDER : 6k vs. FP DP FMA 150k トランジスタ [1,2]
 - (MIPS R3000 が 115k トランジスタ)
 - ◊ レジスタ・ファイルやスケジューラ、予測器の方が大きい
 - スループットではなくレイテンシが重要なため改善が難しい

1. S. Kao et al., "A 240ps 64b Carry Lookahead Adder in 90nm CMOS," ISSCC 2006

2. Vangal et al., "A 6.2- GFlops Floating-Point Multiply-Accumulator With Conditional Normalization," Journal of Solid-State Circuits 2006

両者の違いのまとめ（1）

- HPC アプリケーション（スパコン）：スループット指向
 - ◊ ピーク性能を引き上げることや、データ操作の改善が重要
 - ◊ ハードウェアには投機とかあまり余計なことはして欲しくない
 - ◊ ある程度は人力の最適化で性能を引き出す前提
- 整数系アプリケーション（PC/スマホ）：レイテンシ指向
 - ◊ ピーク性能だけを引き上げてもほぼ意味がない
 - ◊ ハードウェアによる動的な命令スケジューリング方法や投機実行の改善が重要
 - ◊ 人力の最適化で並列化は一般にあまり行われない

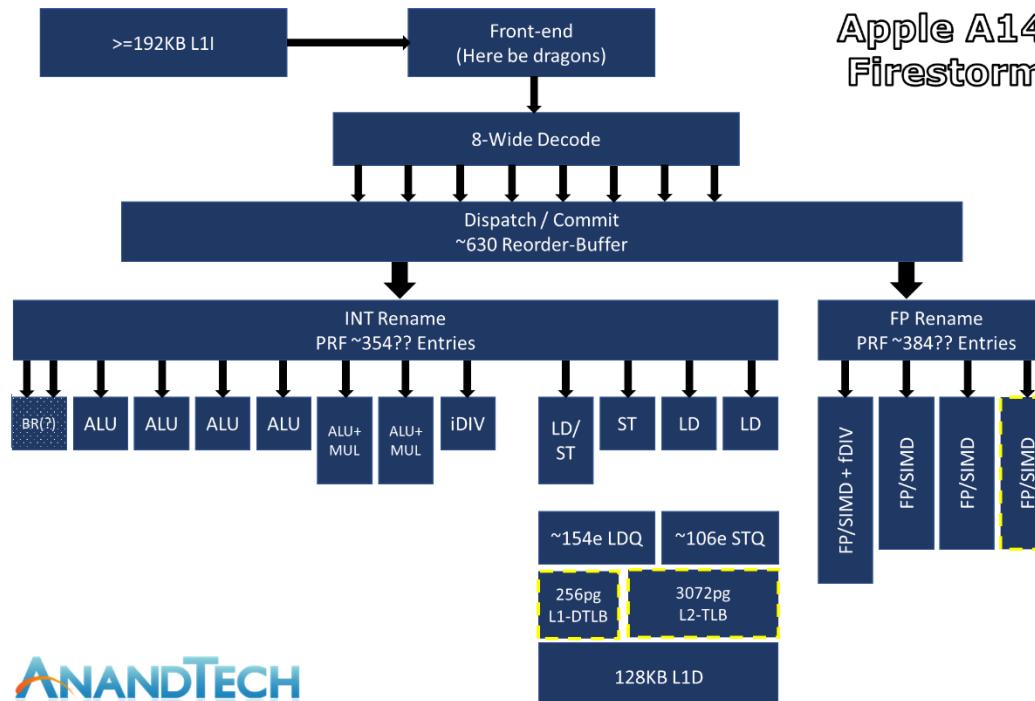
両者の違いのまとめ

- HPC アプリ（スーパーコンピュータ）と整数系アプリ（PC/スマホ）では性質もやるべき事も全然違う
 - ◊ 大量輸送を目的とするダンプカーと、道なき道を行くラリーカーの違いのようなもの
 - ◊ タイヤやエンジンの基盤技術は共通でも、上位互換ではない
 - ◊ 「コンピュータの性能や展望」の話をする際は HPC アプリを前提に話がなされてしまうことは結構ある
- 今日の残りの主題は整数系アプリケーション

整数系アプリケーションの性能向上

整数系アプリケーションの性能向上

- Apple や Intel, AMD, ARM はどんどんコアを大型化
 - ◊ 主に整数系アプリケーションのシングルスレッド性能向上のため
- 例：Apple M1 Firestorm コア
 - ◊ 最大630命令スケジューリング、13整数+4 FP 命令同時発行 可能
 - ◊ iPad Pro に、現時点で最も大きな CPU が乗っている



なぜ Apple がシングルスレッド性能を上げるのか

- PC やスマホ、タブレットで実行されるアプリのレスポンスを重視
 - ◊ 特に WEB ブラウザ上で動作する JavaScript で動くもの

近年の WEB アプリケーションの発展

- HTML5 の普及と共に、ここ 5 年ほどで大きく発展
 - ◊ 大規模アプリケーションの開発方法の確立
 - フレームワークや設計パターンの発展
 - ◊ もはや PC のデスクトップ・アプリに引けを取らない
 - これまで GUI ツールを色々作って来たが、圧倒的に楽で高機能なものを作れると感じる
- サーバーからクライアントに大きく処理を移す傾向
 - ◊ 昔) サーバーサイドの HTML レンダリング
今) WEB ブラウザ上で JavaScript の制御の元で画面を生成
 - ◊ レスポンスや使いがって向上のため
- 実行されるスクリプトのサイズが MB 単位であることも普通

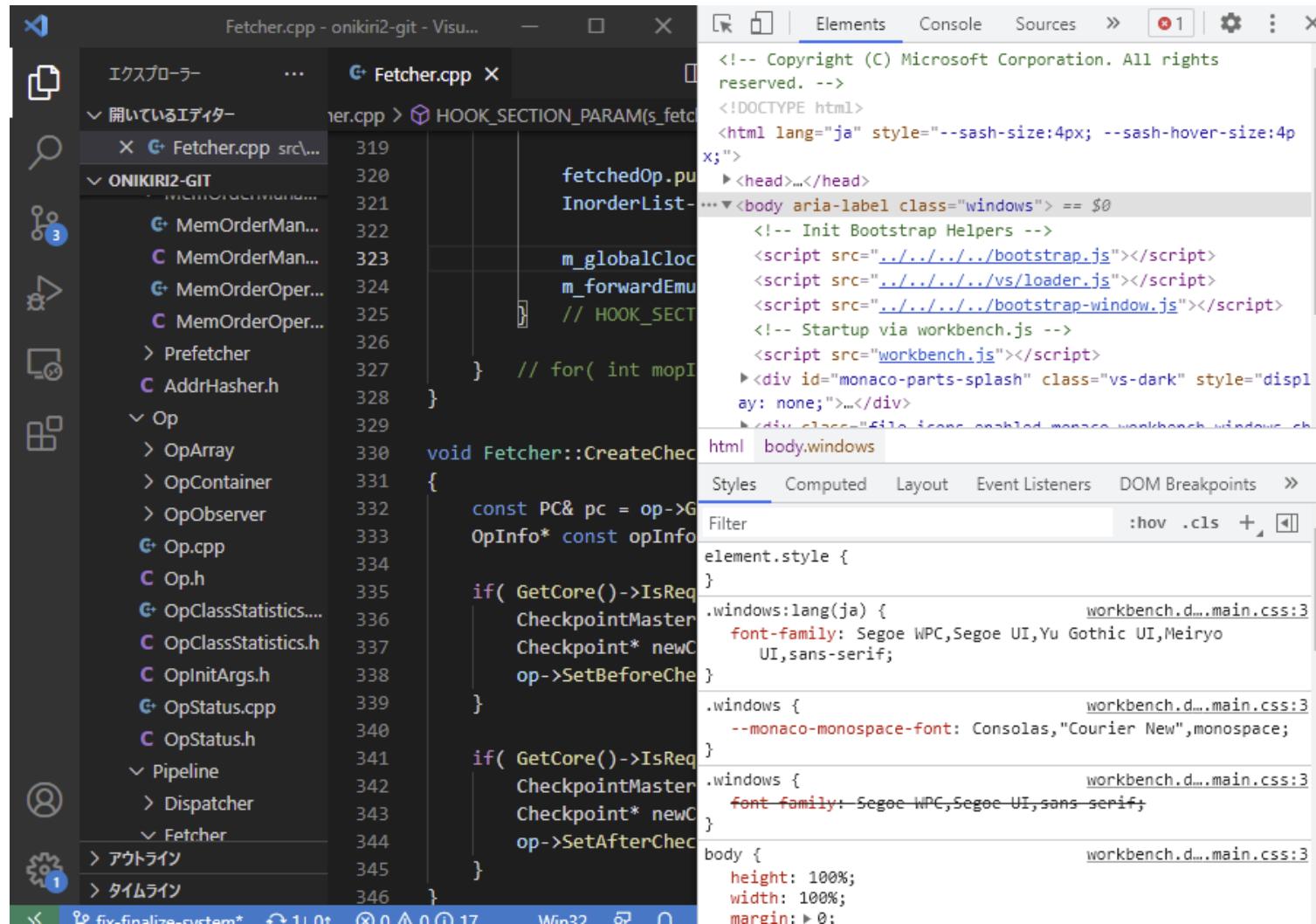
Electron

- 通常のアプリに見えて、WEB アプリケーションであるものも多い
 - ◊ MS Teams や vscode, slack, skype...
- これらはみな Electron と呼ばれるフレームワーク上で動作
 - ◊ デスクトップ・アプリケーションのためのフレームワーク
 - ◊ Google Chromium + Node.js を元に作られている
 - ◊ HTML+JavaScript で開発



vscode の HTML 表示

■ メニューから開発者ツールを ON にすると HTML が見える



The screenshot shows the Visual Studio Code interface with the developer tools open. The left sidebar shows a file tree with several files, including Fetcher.cpp, MemOrderManager.h, and Op.h. The main editor area displays code from Fetcher.cpp. On the right, the developer tools are open, specifically the Elements tab of the DevTools panel. This tab shows the DOM structure of the current page, which is the Visual Studio Code splash screen. The body element has the class "windows". The styles tab shows CSS rules applied to various elements, such as "windows:lang(ja)" and ".body.windows". The bottom status bar indicates the file is "fix_finalize_system*", the line is "110", and the column is "17".

```
<!-- Copyright (C) Microsoft Corporation. All rights reserved. -->
<!DOCTYPE html>
<html lang="ja" style="--sash-size:4px; --sash-hover-size:4px;">
  <head>...</head>
  <body aria-label="windows" style="background-color: #f1f1f1; font-family: Segoe WPC, Segoe UI, Yu Gothic UI, Meiryo, sans-serif; height: 100%; margin: 0; width: 100%;">
    <!-- Init Bootstrap Helpers -->
    <script src="../../../../../bootstrap.js"></script>
    <script src="../../../../../vs/loader.js"></script>
    <script src="../../../../../bootstrap-window.js"></script>
    <!-- Startup via workbench.js -->
    <script src="workbench.js"></script>
    <div id="monaco-parts-splash" class="vs-dark" style="display: none;">...</div>
  </body>
</html>
```

Styles Computed Layout Event Listeners DOM Breakpoints

```
element.style {
```

```
.windows:lang(ja) { font-family: Segoe WPC, Segoe UI, Yu Gothic UI, Meiryo, sans-serif; }
```

```
.windows { --monaco-monospace-font: Consolas, "Courier New", monospace; }
```

```
.body { font-family: Segoe WPC, Segoe UI, sans-serif; }
```

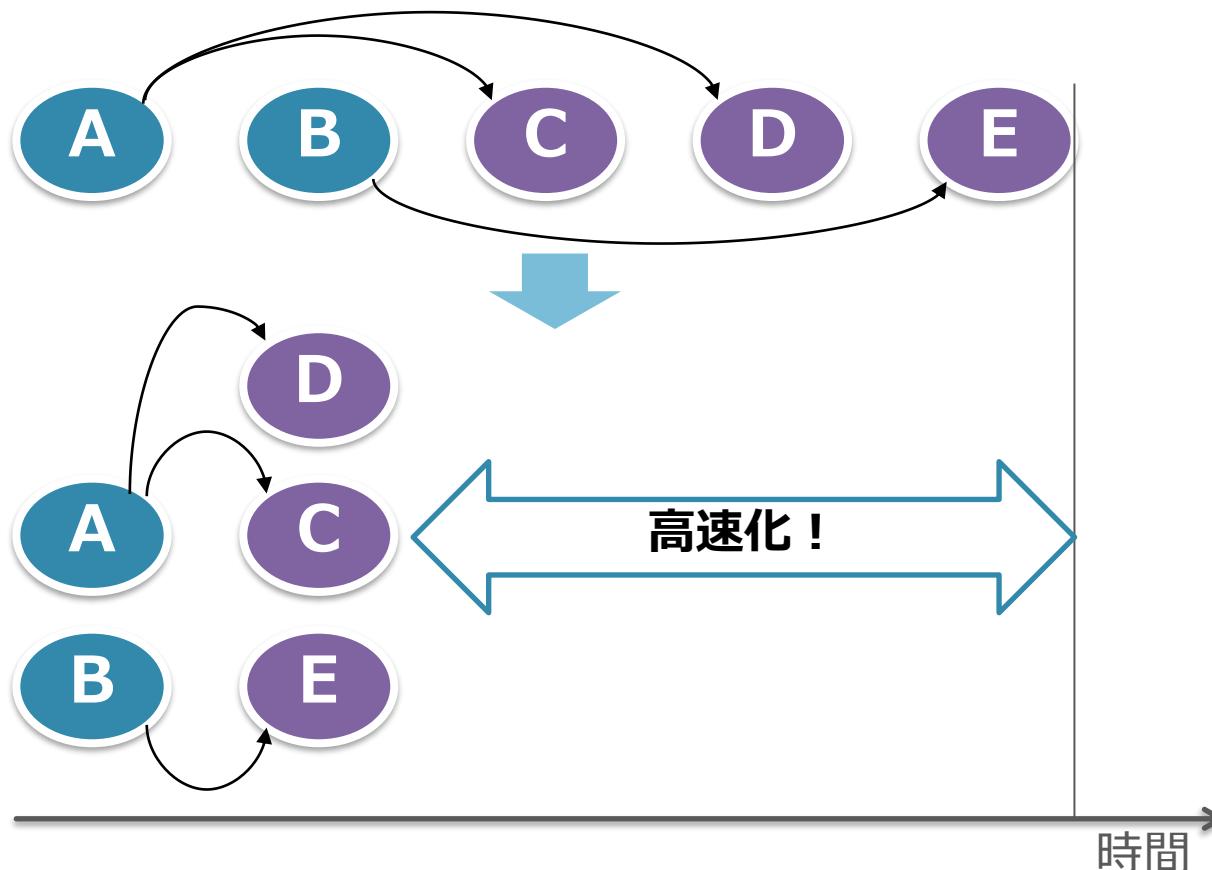
```
body { height: 100%; width: 100%; margin: 0; }
```

JavaScript でできた WEB アプリの性質

- C や C++ などと比べて基本的に数倍～10倍は遅い
 - ◊ (Google v8 エンジンの場合であり、普通はもっと遅い)
 - ◊ 投機処理がより困難
 - 言語処理系が間に挟まっているため、各種予測器が働きにくい
 - ◊ 処理量が多いが、内包する命令レベル並列性は高い
 - 型の動的チェックやハッシュ・テーブルの多用
- コード・フットプリントの増加
 - ◊ ブラウザ上での JIT の影響が大きい
- 実効性能を上げる余地は、ネイティブ・バイナリよりも大きい
 - ◊ プログラム本来の計算に加えて、裏で同時に色々なものが動く

Out-of-order スーパスカラ・プロセッサ

- 並列処理できる部分を探して、並び替えて実行
 - 型チェックやハッシュ・テーブルへのアクセス（紫）が本来の処理（青）に付随して多く含まれる
 - それらを動的に並べ変えて並列に実行



V8 ランタイムの実行の様子 (Chrome の JavaScript エンジン)

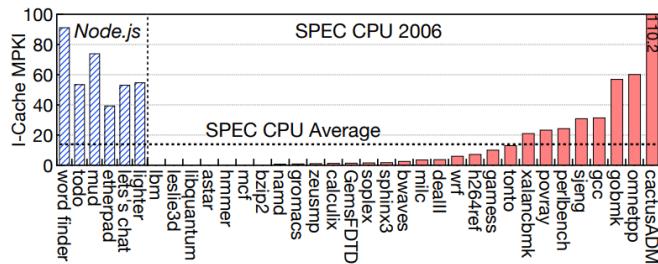
- パイプラインチャート
 - ◇ 縦軸が命令、横軸が時間



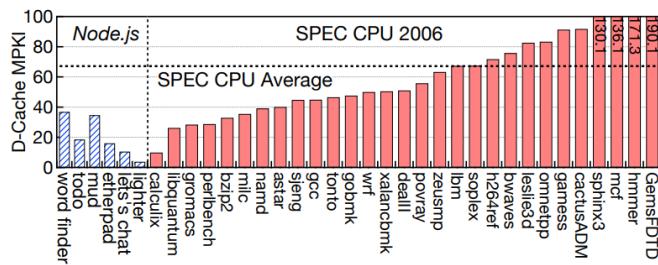
Node.js と SPEC CPU 2006 の違い

左上 I\$/D\$ ミス数、右上 I-/D-TLB ミス数、左下 分岐予測ミス率

Zhu, Yuhao et al., Microarchitectural Implications of Event-driven Server-side Web Applications, MICRO 2015 より



(a) Node.js applications show worse l1-cache locality than SPEC CPU 2006



(b) Current designs already capture the data locality of *Node.js*.

Figure 6: I- and D-cache MPKIs comparison for *Node.js* applications and SPEC CPU 2006.

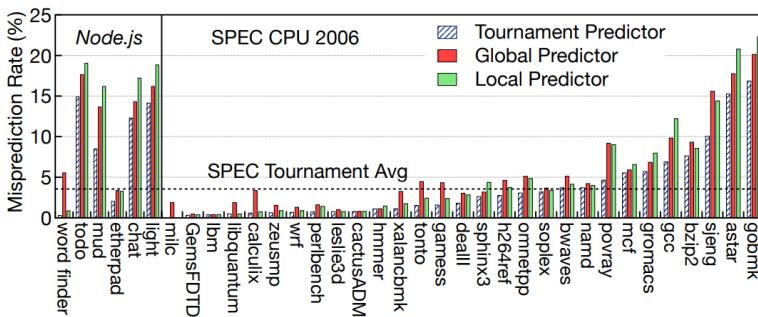
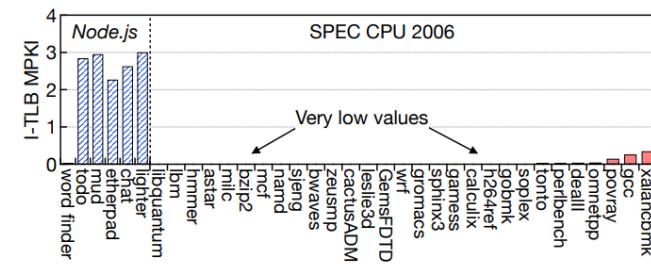


Figure 10: Comparison of *Node.js* and SPEC CPU 2006 applications under three classic branch predictor designs.



(a) I-TLB MPK

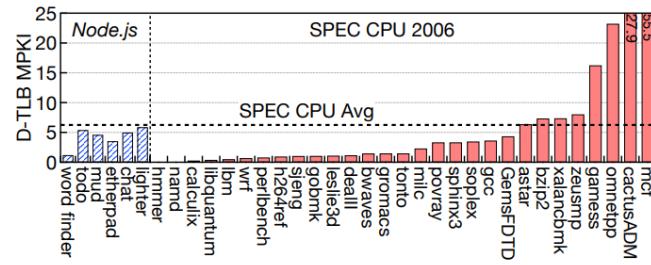


Figure 13: I-TLB and D-TLB MPKIs for the *Node.js* applications and SPEC CPU 2006 applications.

- ◊ Node.js はサーバーサイドで使用する JavaScript 処理系
 - ◊ Node.js (各グラフ内左側) の方が
 - 命令供給や分岐予測がボトルネック
 - データ供給はあまり問題にならない

間接分岐予測器の改良による インタプリタの性能改善

- E. Rohou et al., “Branch prediction and the performance of interpreters — Don’t trust folklore”, CGO 2015
 - ◇ インテル（Nehalem, Sandy Bridge, Haswell）とITTA GE 予測器の比較
 - ◇ Haswell で予測ミスが激減

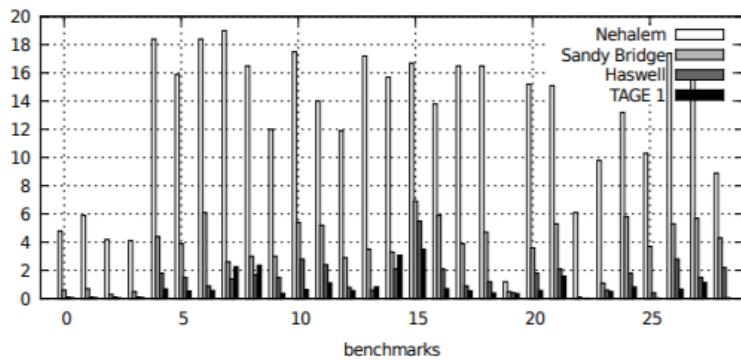


Figure 4. Python MPKI for all predictors

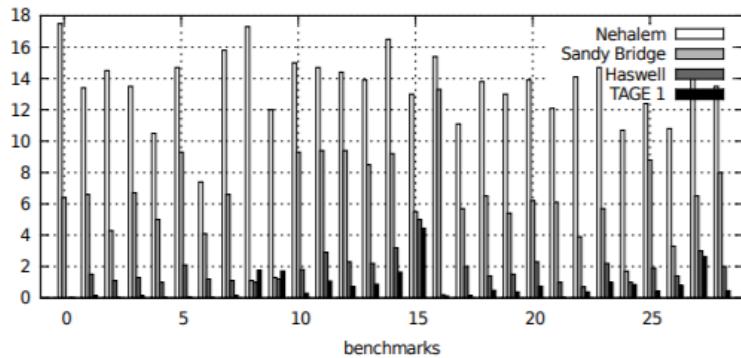


Figure 5. Javascript MPKI for all predictors

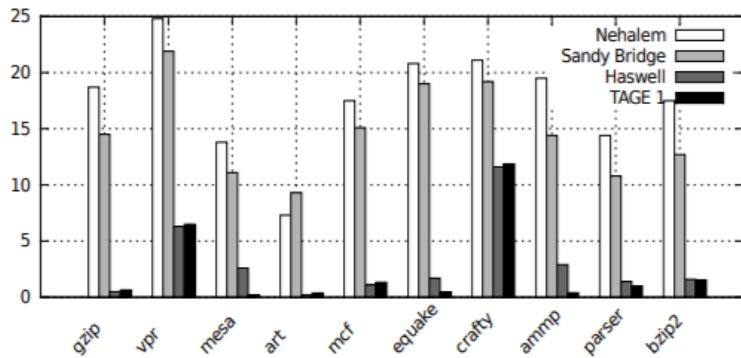


Figure 6. CLI MPKI for all predictors

重たい整数系アプリケーション

- サーバーサイドでも動的言語を使うことは多い
 - ◊ Python, JavaScript, PHP...
- 静的言語でもプログラムの巨大化や抽象化が進むことによる速度低下が進行
 - ◊ コードを整理するために、抽象化レイヤが間にに入る
 - ◊ データベース・エンジンなどで顕著
- アプリケーションは高機能化すると共に、重くなる一方
 - ◊ 「WEB ブラウザが動けば十分」とはよく言ったものの、いま一番重たいアプリは WEB ブラウザ

重たい整数系アプリケーションの高速化

- Out-of-order スーパスカラ・プロセッサによる実行
 - ◊ 動的に命令をスケジュールし、並列実行
 - ◊ さまざまな投機処理を行う
- これ以外の（ハードウェアで）実効的に意味がある方法は、いまのところあまりない

もくじ

1. 「現代の」Out-of-order スーパスカラ・プロセッサの構造
2. 最近の研究

「現代の」Out-of-order スーパスカラ・プロセッサの構造

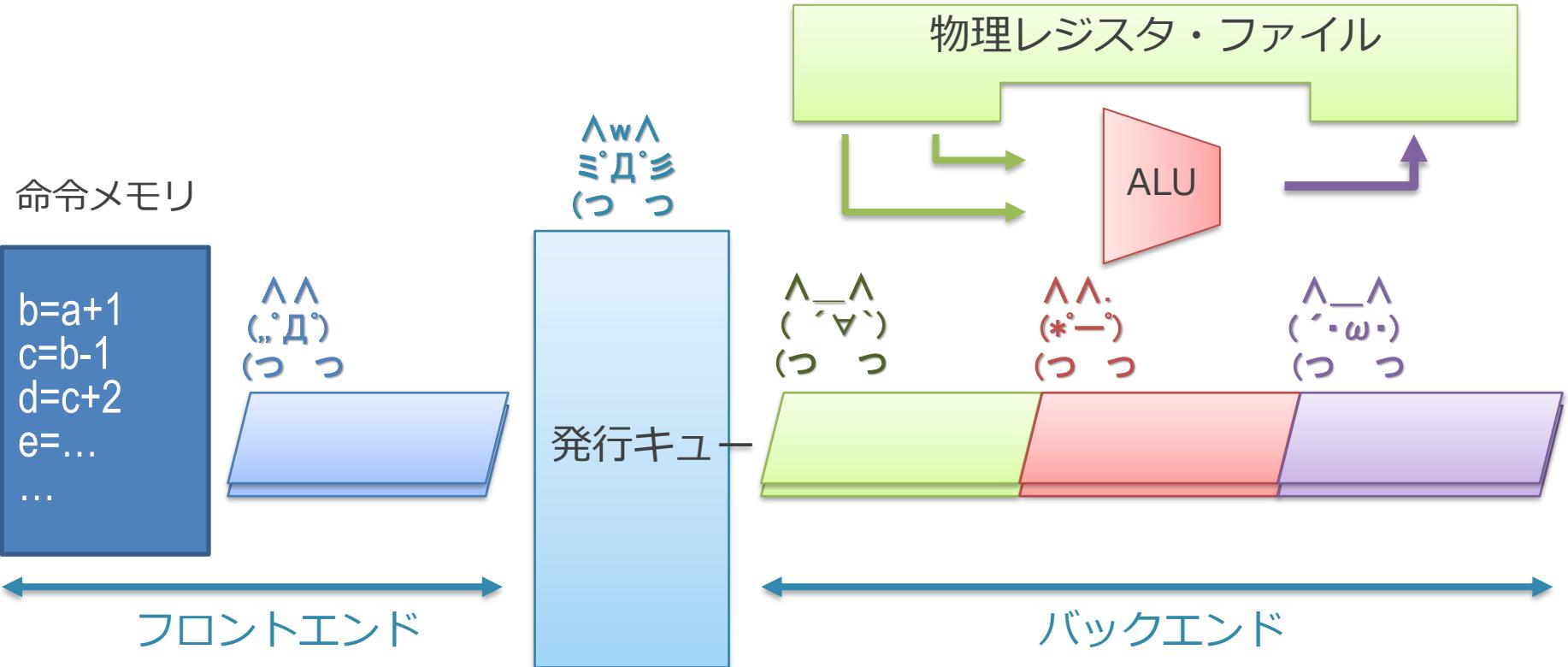
Out-of-order スーパスカラ・プロセッサ

- 今日とりあげる研究についての話題：
 - ◊ Out-of-order スーパスカラ・プロセッサに関わるもの
 - ◊ 長いので、以降では「OoO プロセッサ」と書きます
 - ◊ その前提として、基本を簡単に説明・復習
- キーワード：
 - ◊ 物理レジスタ方式
 - ◊ マトリクス・スケジューラ

OoO プロセッサの方式

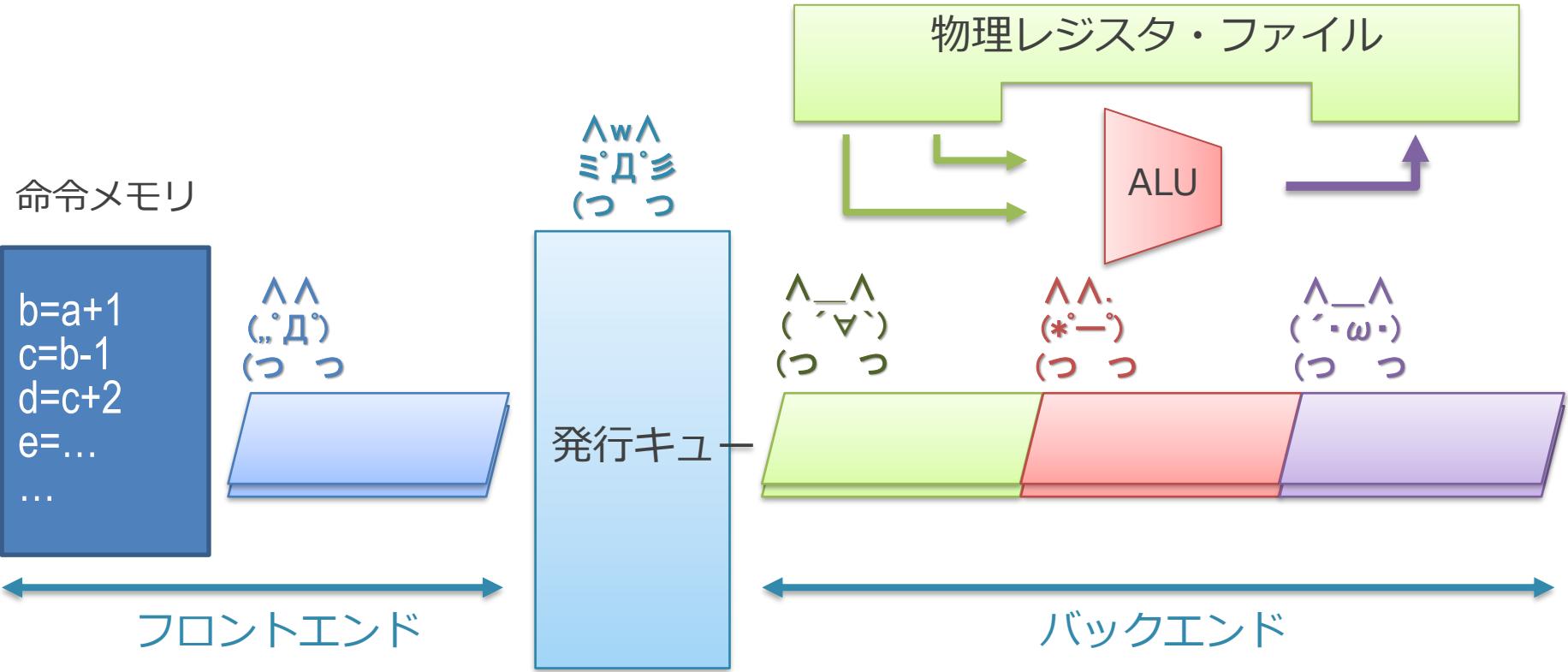
1. Reservation Station (RS) 方式
 - ◊ トマスロ (Tomasulo) のアルゴリズムとして知られるものの発展
 - ◊ IBM System/360 で初めて実装された
 2. 物理レジスタ方式
 - ◊ MIPS R10000 で初めて実装された（と思う）
- 現在主流なのは後者の物理レジスタ方式
- ◊ 構造がより単純で設計しやすく性能も出やすいから
 - ◊ 文献では 2 者の関係があまり書かれていないが、これらはかなり構造が違う

物理レジスタ方式の OoO プロセッサの構造



- 発行キューによって前後に分離されたパイプラインを持つ
 1. フロントエンド： 命令をフェッチ, リネーム
 2. 発行キュー： 発行待ち命令の待ち合わせのバッファ
 3. バックエンド： レジスタを読んで命令を実行

大ざっぱな動作



1. フロントエンドで命令を順にフェッチしてリネーム
2. 発行キューにディスパッチ
3. 発行可能なものから順にバックエンドに命令を発行
4. レジスタを読み、演算器で実行し、書き戻す

レジスタ・リネーム

- 目的：出力依存と逆依存を取り除く
 - ◊ 真の依存にのみ従って発行を行うことができるよう

1. 真の依存 (RAW: Read after Write) :

I1: add $x_1 \leftarrow x_2 + 1$

I2: add $x_3 \leftarrow x_1 + 1$ // I1 の結果がなければ I2 は実行できない

2. 逆依存 (WAR: Write after Read) :

I1: add $x_2 \leftarrow x_1 + 1$ // I2 は I1 の結果に依存していないが、

I2: add $x_1 \leftarrow x_3 + 1$ // I2 を先に実行すると x_1 が壊れる

3. 出力依存 (WAW: Write after Write) :

I1: add $x_1 \leftarrow x_2 + 1$ // I2 は I1 の結果に依存していないが、

I2: add $x_1 \leftarrow x_3 + 1$ // I1 を後で実行すると x_1 が壊れる

レジスタ・リネーム

- 方針：レジスタの名前を付け替える
 - ◊ 偽の依存の原因 = 同じレジスタの使い回し
 - 上書きにより破壊される
 - ◊ ディスティネーションにその命令専用のレジスタを動的に毎回新しく割り当てる
 - ◊ レジスタ番号がかぶらないので、他の命令との間で出力依存や逆依存は生じなくなる

I1: mul $x_3 \leftarrow x_2 * 4$

I2: add $x_3 \leftarrow x_1 + 1$

I3: sub $x_1 \leftarrow x_5 - 1$

I4: and $x_6 \leftarrow x_7 \& 1$



I1: mul $p_{20} \leftarrow p_{12} * 4$

I2: add $p_{21} \leftarrow p_{11} + 1$

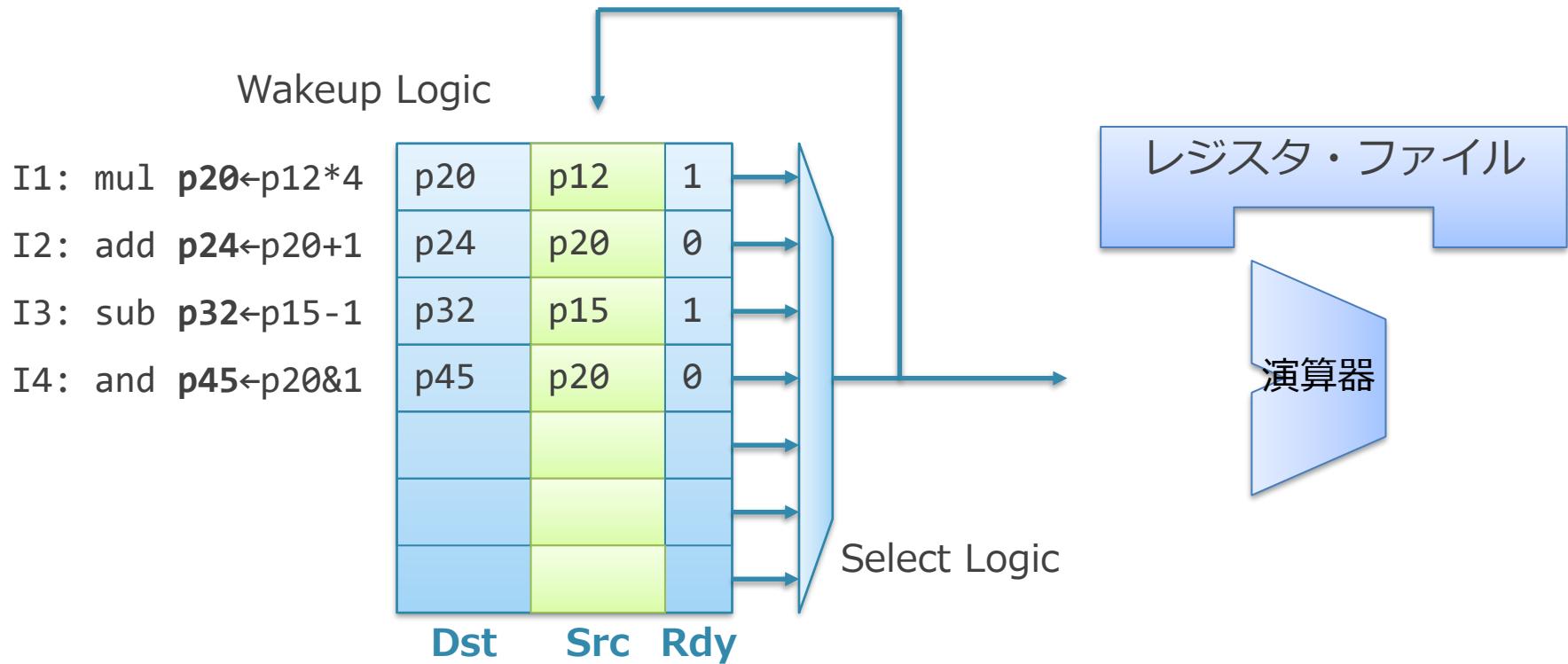
I3: sub $p_{22} \leftarrow p_{15} - 1$

I4: and $p_{23} \leftarrow p_{17} \& 1$

バックエンドへの命令の発行

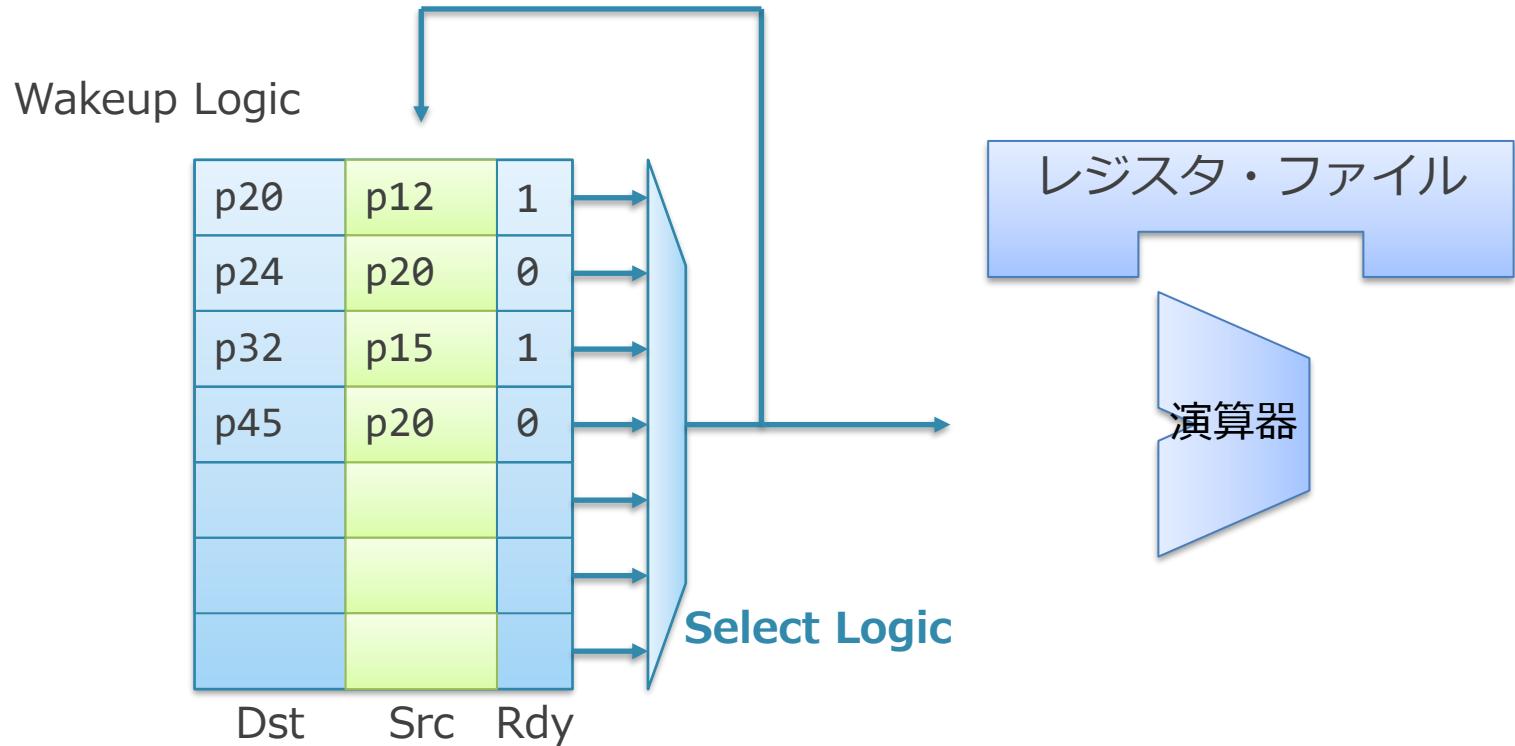
- 命令スケジューリング
 - ◊ 依存関係が解けた命令から順に演算器に発行する
- スケジューラの作り方
 1. 連想検索を使う方法
 2. 行列を使う方法

連想検索による実装



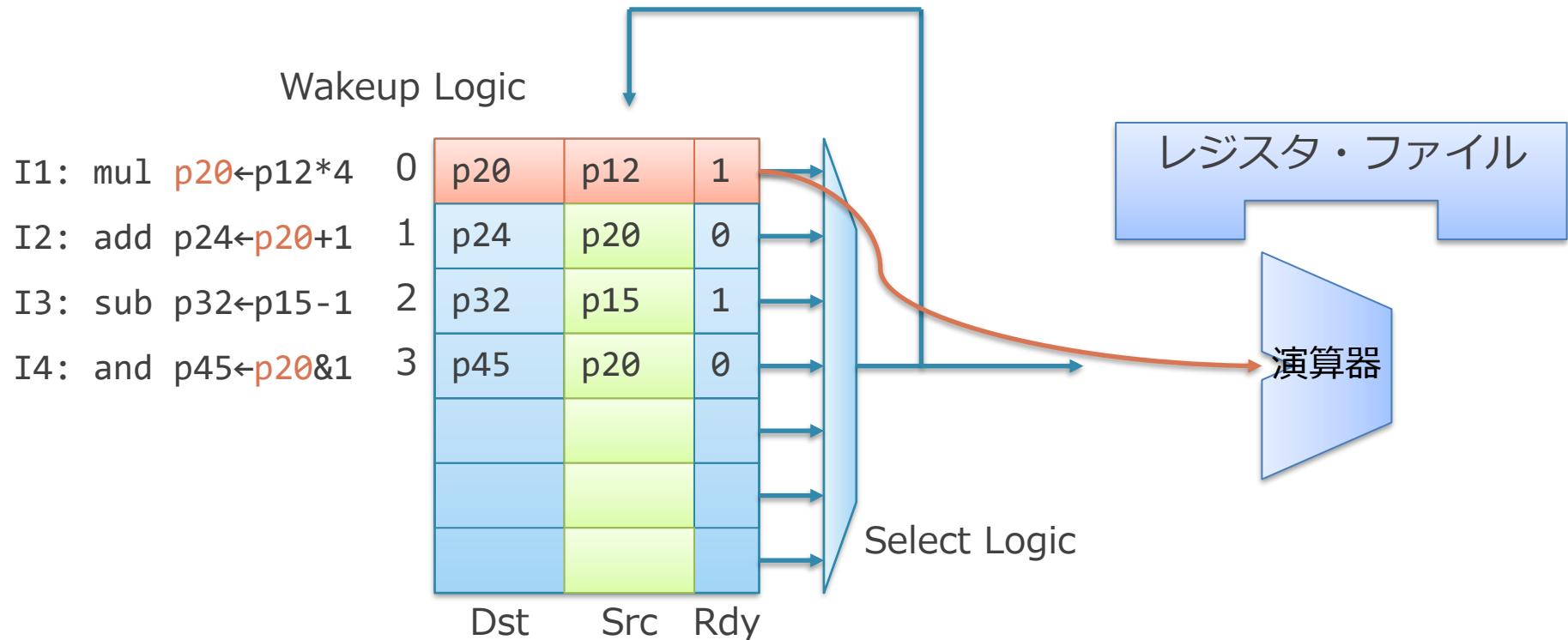
- Wakeup Logic : 寝ている（発行待ち）命令を起こす機構
 - ◊ Dst : ディスティネーションの物理レジスタ番号
 - ◊ Src : ソースの物理レジスタ番号（簡単のためここでは 1 つ）
 - ◊ Rdy : 依存元命令が実行され、発行準備ができているか

連想検索による実装



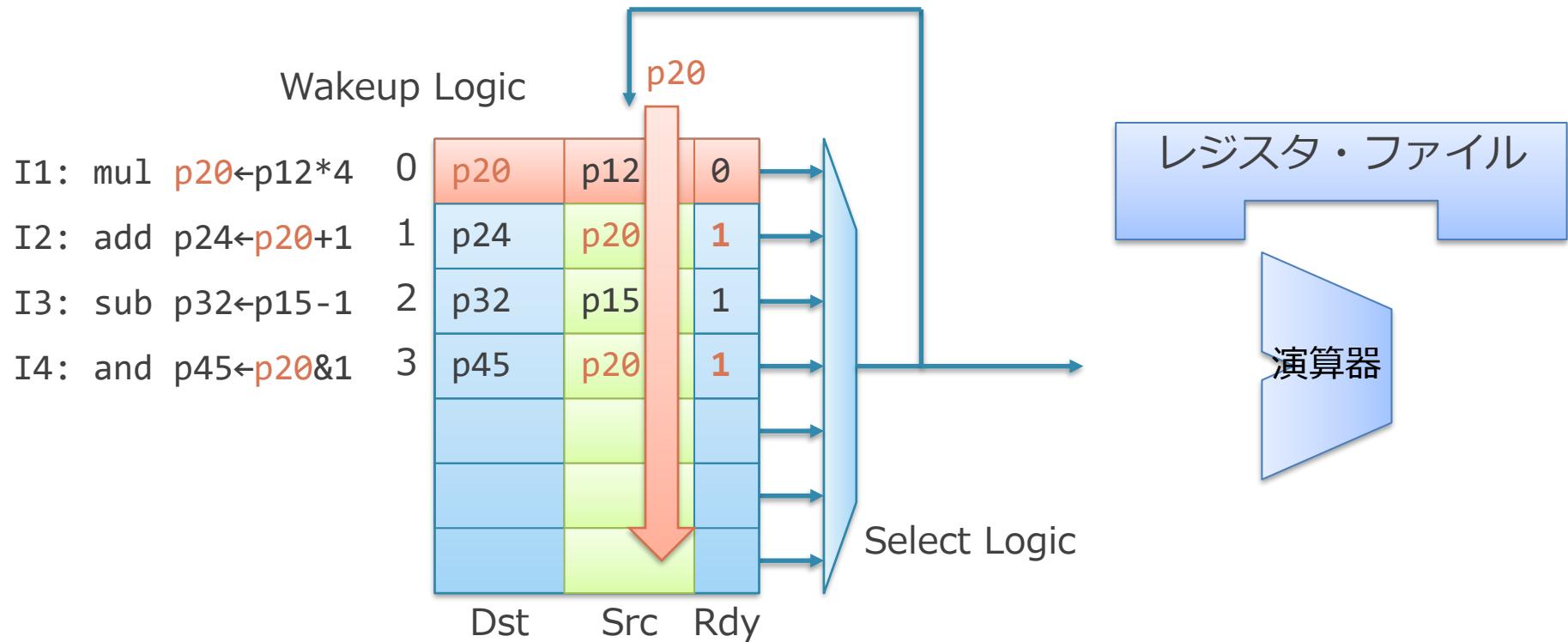
- Select Logic : 起きた命令から発行する命令を選ぶ機構
 - ◊ 発行準備ができている命令 ($Rdy=1$) から 1 つを選んで発行
 - ◊ インテル用語だと Picker

発行キューの動作（1）：Select



- 現在エントリ 0 番と 2 番で発行準備ができている
 - ◊ Select Logic により、0 番を選択
 - ◊ 0 番の内容（命令）がバックエンドに発行される
- （理想的にはプログラム内の命令順に優先度を付けて選ぶが、ハードが複雑なので適当に番号が若いものを選ぶこともある）

発行キューの動作 (2) : Wakeup



- Select して発行した命令のディスティネーション番号を
Wakeup Logic 全体にブロードキャスト
 - ◊ 各エントリのソースオペランドの番号と一致比較を行う
 - ◊ 一致した場合は依存が満たされた = Rdy を 1 に (wakeup)

バックエンドへの命令の発行

- 発行キューの作り方
 - 1. 連想検索を使う方法
 - 2. 行列を使う方法

行列を使った方式

- モチベーション：連想検索を使った方式の問題
 - 1. 同時発行数に比例して回路が大きくなる
 - ブロードキャストする信号線と比較器が増加
 - 2. 消費電力が大きい
 - ものすごい数の比較器が同時に稼働している
 - 3. 遅延が大きい
 - 命令のセレクト後にブロードキャストすべき物理レジスタ番号を別のテーブルから読み出す必要がある

行列を使った方式

■ マトリクス・スケジューラ

- ◊ Masahiro Goshima et al., "A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors", MICRO 2001
- ◊ Sassone, Peter G. et al., "Matrix Scheduler Reloaded", ISCA 2007

■ 行列状の構造を使って wakeup を行う

- ◊ 各命令が依存元命令のビットベクタもつ
- ◊ レジスタ番号を介さずに命令 to 命令で直接 wakeup
- ◊ インテルや IBM の CPU ではこれを使っている

マトリクス・スケジューラ

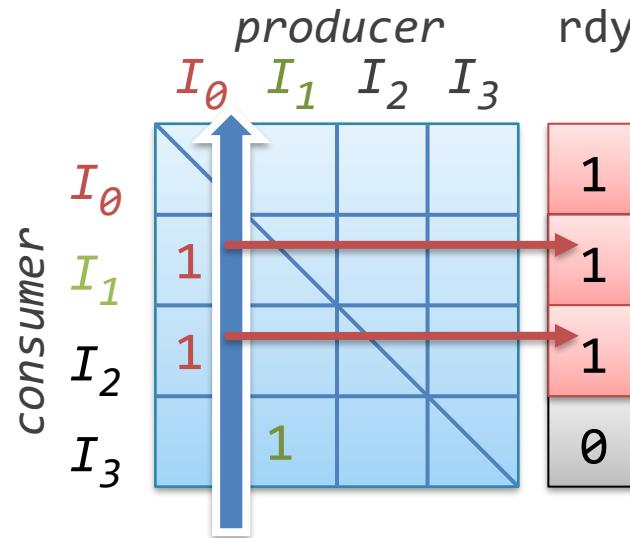
```
 $I_0$  ld   p11  $\leftarrow$  *(p10)  
 $I_1$  sll  p12  $\leftarrow$  p11 << 1  
 $I_2$  sll  p13  $\leftarrow$  p11 << 2  
 $I_3$  add  p14  $\leftarrow$  p12 + 1
```

		producer				rdy
		I_0	I_1	I_2	I_3	
consumer	I_0	1				1
	I_1		1			0
	I_2			1		0
	I_3				1	0

- 行、列がそれぞれ発行キューのエントリに対応
 - ◊ 行：コンシューマ　　列：プロデューサ
- プロデューサとコンシューマの交点により依存関係を表す
 - ◊ 各行は、その命令が依存している列に 1 を立てる

マトリクス・スケジューラ

I_0 ld $p_{11} \leftarrow *p_{10}$
 I_1 sll $p_{12} \leftarrow p_{11} \ll 1$
 I_2 sll $p_{13} \leftarrow p_{11} \ll 2$
 I_3 add $p_{14} \leftarrow p_{12} + 1$



- Wakeup の動作
 - ◊ 発行された命令の列をアサートする
 - ◊ 各行のその列のビットが立っていたら rd़y を 1 に

- RAM の読み出しに近い構造で実現できる
 - ◊ Wired-OR により、複数ソース・オペランドを同時に扱うこともできる（いくつかバリエーションがある）

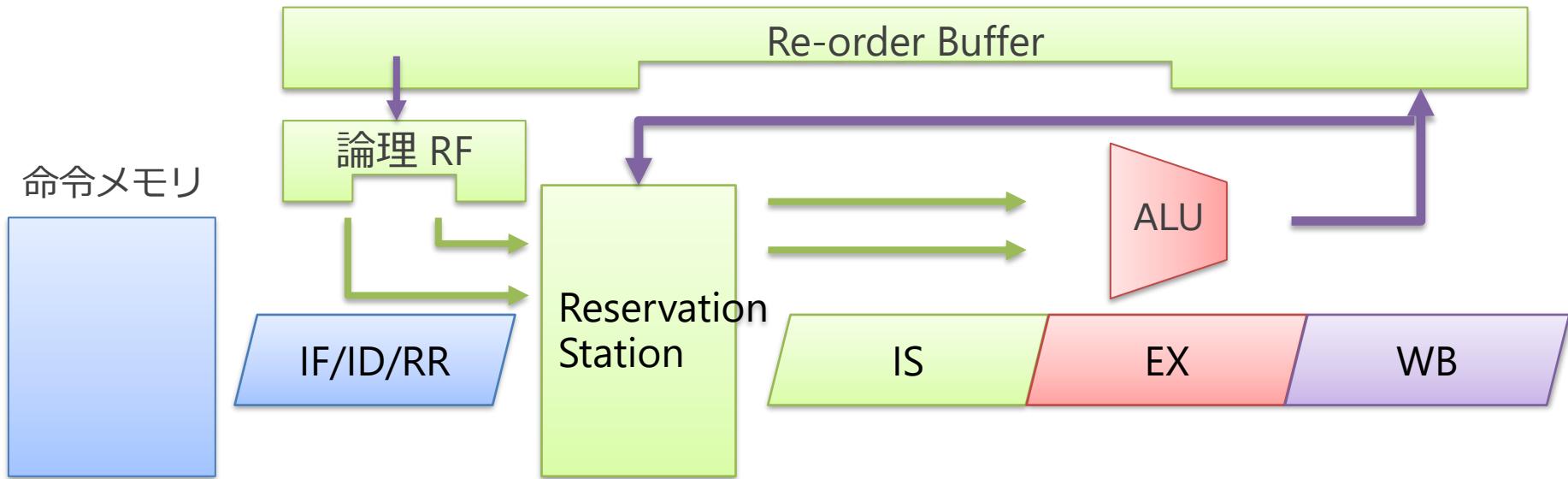
マトリクス・スケジューラの利点

- 発行幅と回路構成が無関係
 - ◊ 発行幅が増えても、同時にアサートする列の数が増えるだけ
- ブロードキャストを伴う比較回路を含まない
 - ◊ 消費電力が小さい
- ブロードキャストすべきレジスタ番号を読み出す必要がない
 - ◊ 遅延が小さい

Out-of-order 発行の方式

- ここまで話してきたのは物理レジスタ方式
 - ◊ 現在はこれが主流
- 比較のために Reservation Station (RS) 方式も説明
 - ◊ もともとトマスロのアルゴリズムがベース

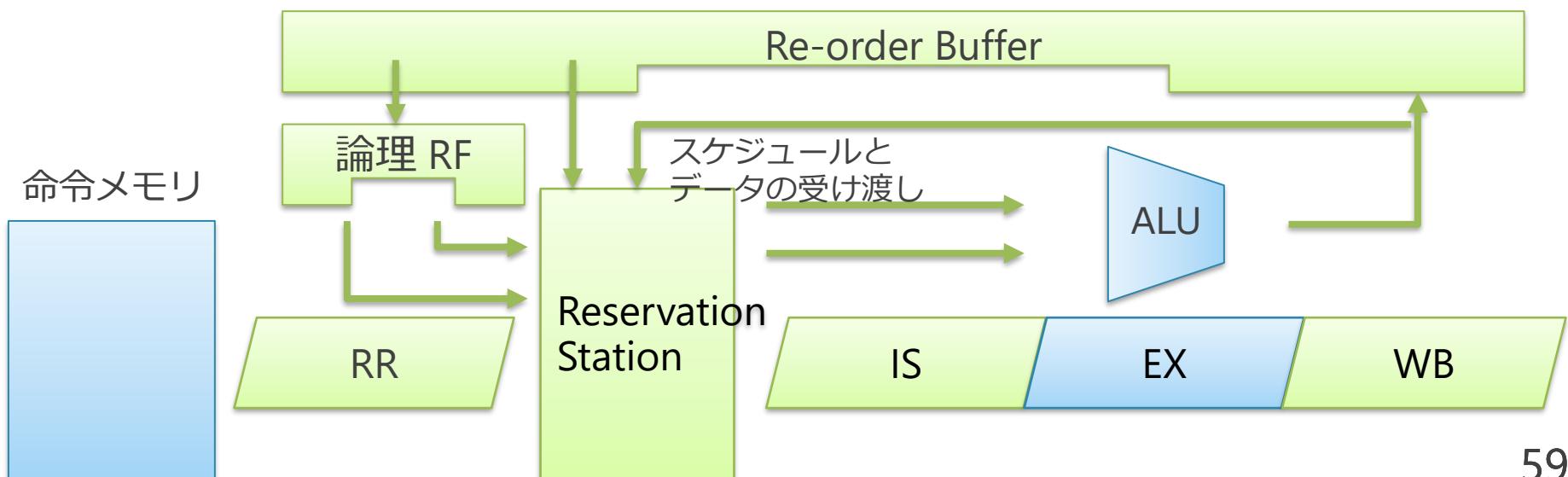
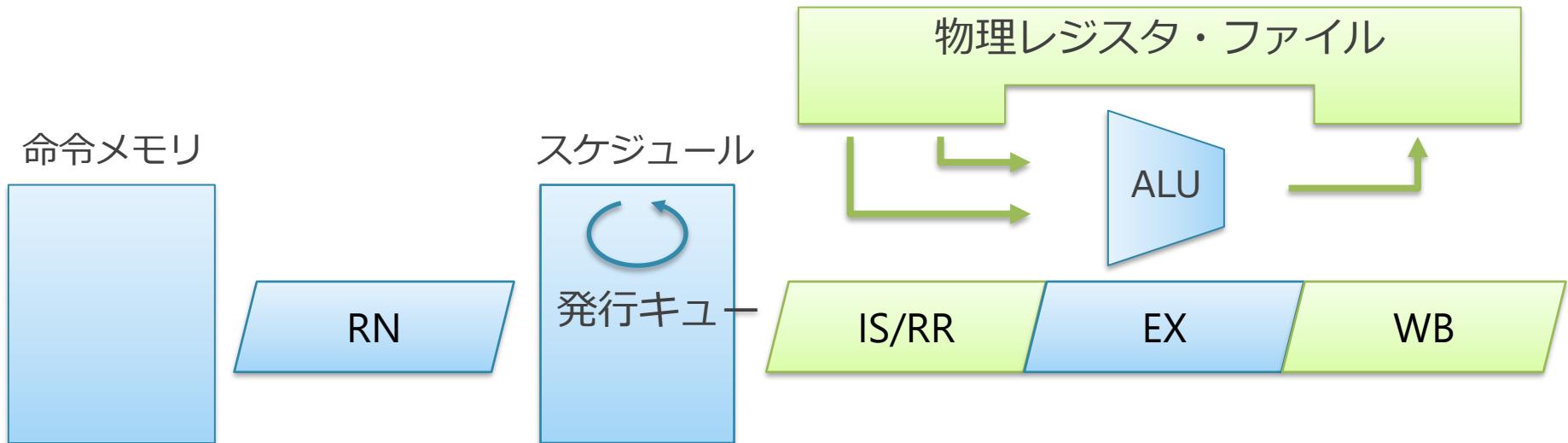
RS 方式



- ◇ 論理レジスタ・ファイル
 - デコード後にフロントエンドで論理レジスタ番号でアクセスされる
 - レジスタ値 or タグがとれる（タグ：命令を識別する ID）
- ◇ Reservation Station
 - ソース・オペランドのレジスタ値 or タグが置かれ「値」の待ち合わせを行う
 - 演算結果の値とタグをブロードキャスト
- ◇ Re-order Buffer (ROB)
 - 演算器で演算された結果が out-of-order に書き込まれ、論理 RF に in-order に結果を書き戻す

物理レジスタ方式（上）と RS 方式（下）の違い

RS 方式はデータ・パス（緑部分）がかなり複雑



物理レジスタ方式と RS 方式の違い

- 物理レジスタ方式は、データとスケジュールを完全に分離
 - ◊ まずリネームをしてしまう
 - ◊ リネーム結果の真の依存にのみ従ってスケジュール
- RS 方式は、in-order 発行を行う CPU からの延長
 - ◊ 論理レジスタ・ファイルを拡張して、現在待ち合わせ中の場合はそれをタグとして書いておく
 - ◊ 実行結果の値そのものをバッファ（リザベーション・ステーション）で待ち合わせる

物理レジスタ方式と RS 方式の違い

■ 物理レジスタ方式

- ◊ レジスタの値は単一の物理レジスタ・ファイルに格納
- ◊ データはバックエンドでのみアクセス

■ RS 方式

- ◊ レジスタの値は複数箇所に複製されて存在
 - 論理RF, RS, ROB
- ◊ データはフロントエンドとバックエンドでアクセス

0o0 プロセッサの変化

- RS 方式から物理レジスタ方式への変化
 - ◊ 余計な値の複製や移動がなくなる
 - ◊ 値のブロードキャストがなくなる
- マトリクススケジューラの導入
 - ◊ 物理レジスタ番号のブロードキャストがなくなる

大昔のスーパスカラと VLIW との比較

- Tetsuya Hara et al., Performance Comparison of ILP Machines with Cycle Time Evaluation, ISCA 1996
 - ◊ VLIW と OoO プロセッサの比較を行った論文
 - ◊ OoO プロセッサは周波数低下が避けられないとしていた
 - RS へのブロードキャストと比較による遅延が要因
- しかし、その後ブロードキャストの問題は解決された
 - ◊ 物理レジスタ方式などの採用が大きい

最近の研究

もくじ

1. 「現代の」Out-of-order スーパスカラ・プロセッサの構造
2. 最近の研究

ここ数年のトップカンファレンスに出ている研究

- ISCA, MICRO, HPCA, ASPLOS より選択
 - ◊ OoO プロセッサの性能向上や電力効率改善の話題をピックアップ

話題

1. クリティカルな命令への対処
2. キャッシュ周りのあまり着目されてこなかった部分
3. コア・アーキテクチャ自体の改良

話題

1. クリティカルな命令への対処
 1. 値予測
 2. 分岐のプレディケートへの変換
 3. 予測ミスを意識した命令発行方法
2. キャッシュ周りのあまり着目されてこなかった部分
3. コア・アーキテクチャ自体の改良

値予測

- Sumeet Bandishte et al.,
“Focused Value Prediction”, ISCA 2020
- インテルのインドとイスラエルのチーム

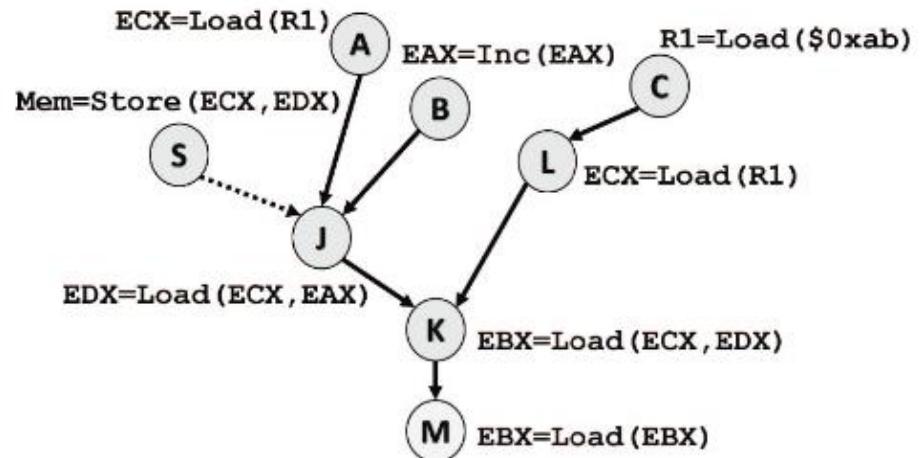
- 値予測：
 - ◊ 命令の演算結果を予測し、その依存先を投機的に実行
 - ◊ 真の依存を超えて OoO 実行を行い、性能を向上
- 既存のアプローチ：
 - ◊ とにかく予測のカバー率を増やそうとしている
 - 色々な予測器をハイブリッドにするとか
 - 大きなオーバーヘッドが生じる
 - ◊ その割に、実際には性能を向上させない
 - 多くの場合は OoO 実行により既に遅延が隠蔽されている

アプローチ

- 値予測の効果がある命令に絞って予測
 - ◊ OoO プロセッサでボトルネックになる命令のみに焦点を絞る
 - クリティカル・パス上の、特定のクリティカル命令に絞る
 - LLC ミスを起こす命令の先祖など
 - ◊ 同じアドレスへのストア → ロードを検出して予測
 - メモリ・リネーミングの一種

実装

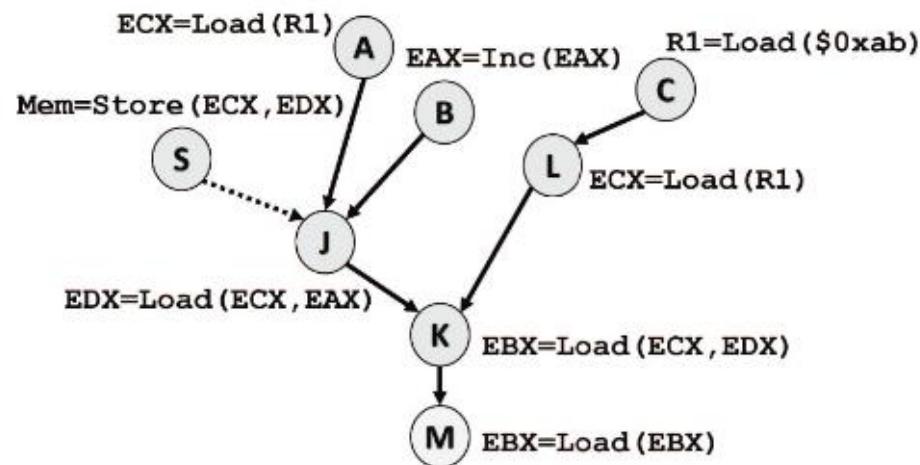
- クリティカル・パスの root (図の「M」) となっている命令の検出
 - ◊ 実行時に ROB の末尾付近にいるかどうかをチェック
 - 末尾にいる = ROB フルでストールを発生させている
 - ◊ 末尾にいることが多い場合、クリティカル・パスの root と判定



実装

- root の依存元を辿り、値予測可能かどうかを判定
 - ◊ RMT を拡張し、各レジスタの値を生成した命令の PC を記録
 - ◊ 先祖の PC が値予測可能かどうかを予測してみて判定
 - 予測不能な場合、その先祖をさらに遡る
 - ◊ 別途、メモリを介した依存もおいかける

- 予測そのものは last value 予測と分岐履歴を使った context last value 予測で行う
 - ◊ 予測はロード命令のみ行う？



評価：性能向上とカバレッジ（ロード命令の予測率）

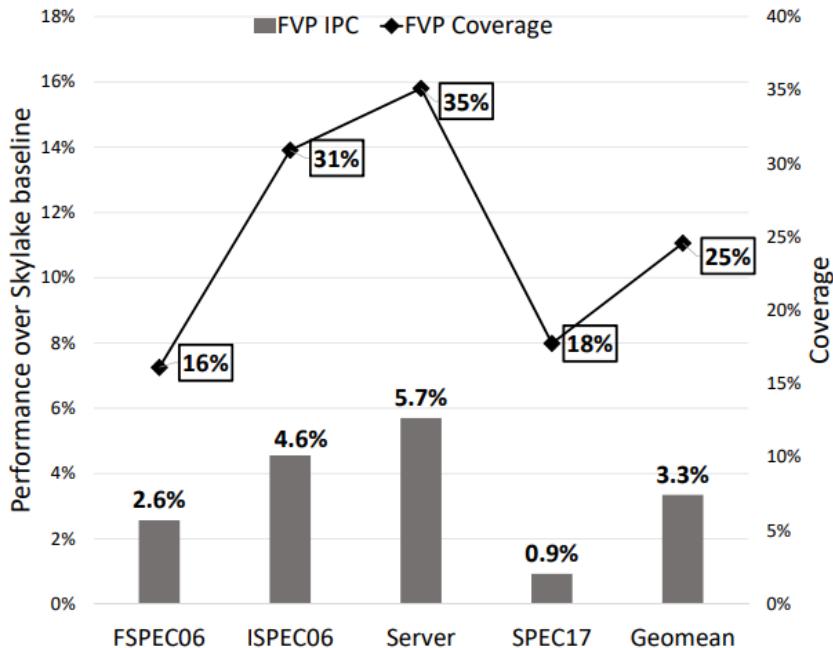


Fig. 6. Performance and Coverage of FVP on Skylake

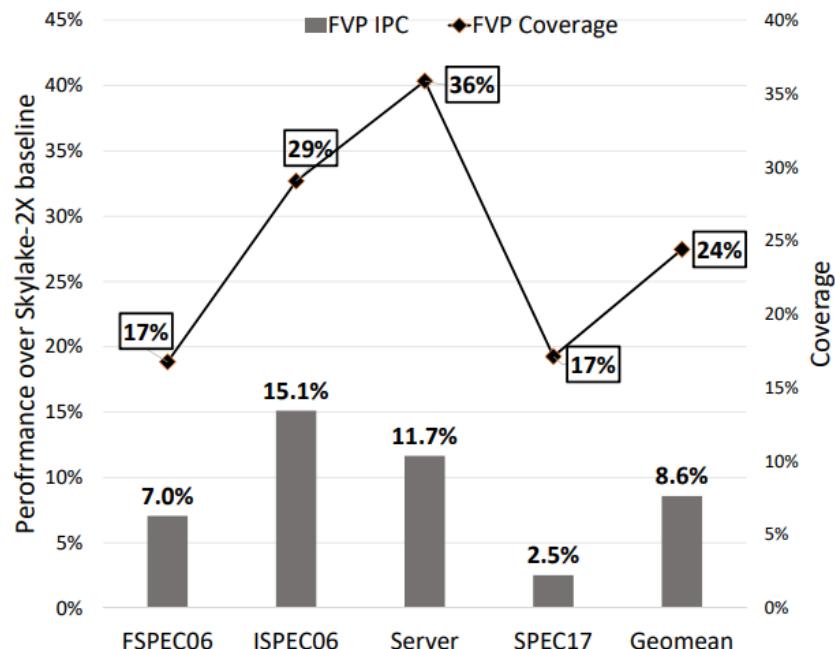


Fig. 7. Performance and Coverage of FVP on Skylake-2X

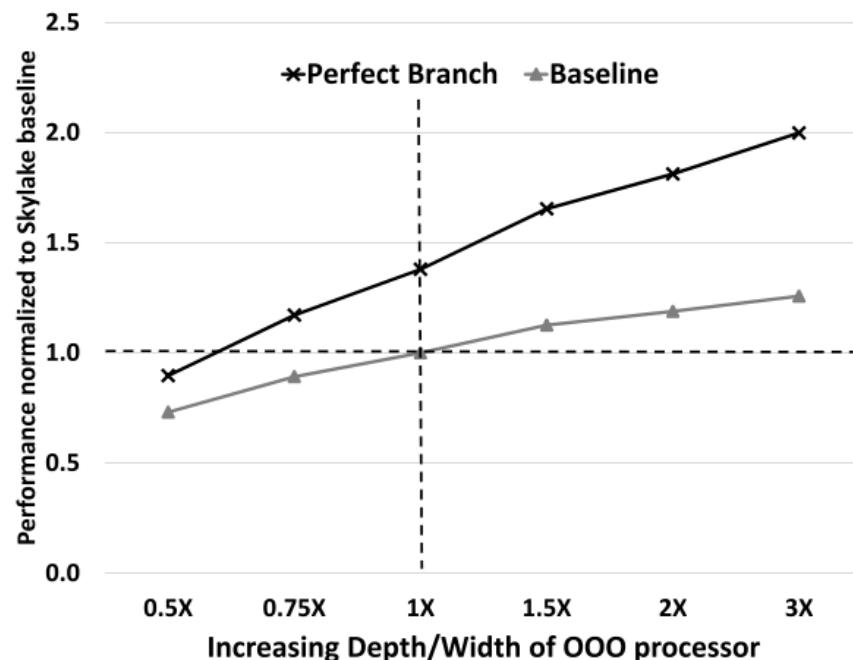
- ◇ 1.2KB の追加回路のみで実現
- ◇ 図左) ベースライン (Skylake) に対して 3.3% 性能向上
- ◇ 図右) 将来現れるだろう幅などを倍にしたプロセッサでは 8.6% 性能向上
- クリティカル・パスの命令の影響がよりあらわになる

分岐のプレディケートへの変換

- Adarsh Chauhan et al.,
"Auto-Predication of Critical Branches", ISCA 2020
- インテルのインドとイスラエルのチーム

背景

- 分岐予測器が大きく発展
 - ◊ パーセプトロン予測器, TAGE 予測器
- しかし、どうしても予測困難な分岐は存在
 - ◊ 過去の履歴と相関のないものは本質的に予測不能
- プロセッサの性能を大きく制限
 - ◊ 特に深さや幅が増すほど影響が大きくなる
 - ◊ Baseline は Intel Skylake



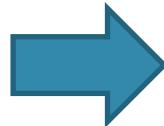
アプローチ

■ アプローチ：

- ◊ ハードウェアで動的にプレディケート実行に変換
- ◊ IBM POWER8 には既に簡単なものが搭載

```
if (r1) {  
    r2 = r3 - r4  
} else {  
    r2 = r3 + r4  
}
```

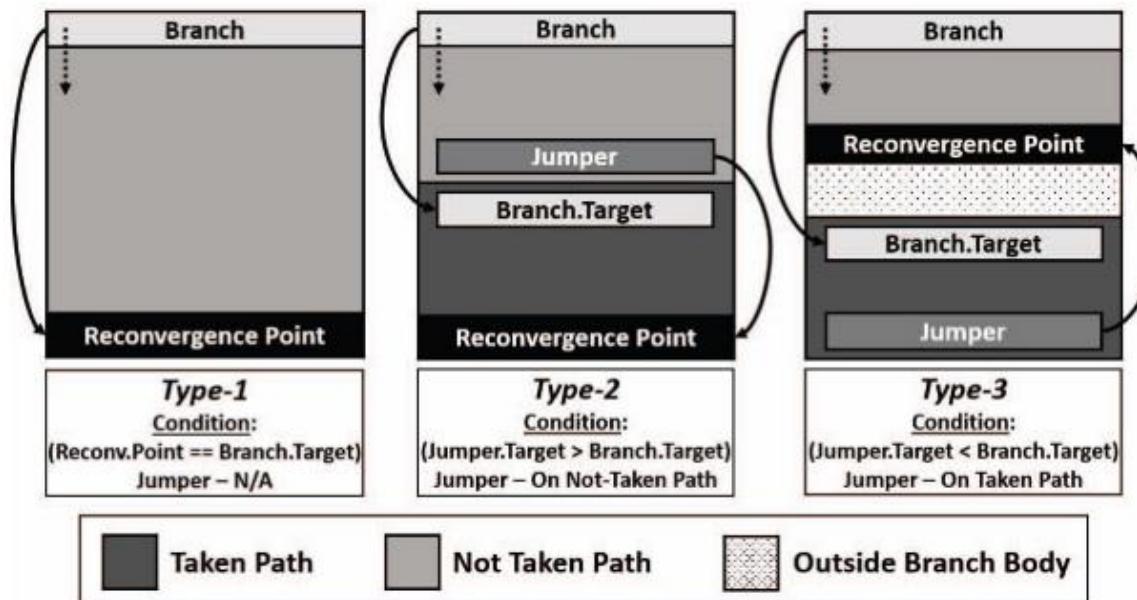
```
i1: bne r1 L1  
i2: add r2 < r3,r4  
i3: jump L2  
L1:  
i4: sub r2 < r3,r4  
L2:
```



```
i2': (!r1) r2 = r3 + r4  
i4': ( r1) r2 = r3 - r4
```

動的な変換の方法

- 合流点を持つかどうかを判別（下記 Type-1 の場合）
 1. 最初の Branch の飛び先のアドレスを記録
 2. Not-taken 時に一定命令以内に PC が 記録されたアドレスと一致するかどうかで判別
- 両パスをフェッチ・実行する（Type-1 では 2. は必要ない）
 1. 分岐予測結果にかかわらず not-taken のパスの命令をフェッチ
 2. 合流点に達したら taken のパスに PC を遷移させフェッチを行う
 3. 合流点に再度達したら、通常の実行に戻る



動的な変換の制御

- 分岐予測が当たる場合は性能が下がる
 - ◊ 命令レベル並列性が下がる
 - データ依存が増えるため
 - 以下の例では, r1 への依存が加わっている
 - ◊ 両方のパスの命令を実行する必要がある
 - then/else パートが大きいと問題に
- 動的に分岐予測の確信度や性能などをモニタし, 変換を制御

```
i1: bne r1 L1
i2: add r2 ← r3,r4
i3: jump L2
L1:
i4: sub r2 ← r3,r4
L2:
```



```
i2': (!r1) r2 = r3 + r4
i4': ( r1) r2 = r3 - r4
```

評価

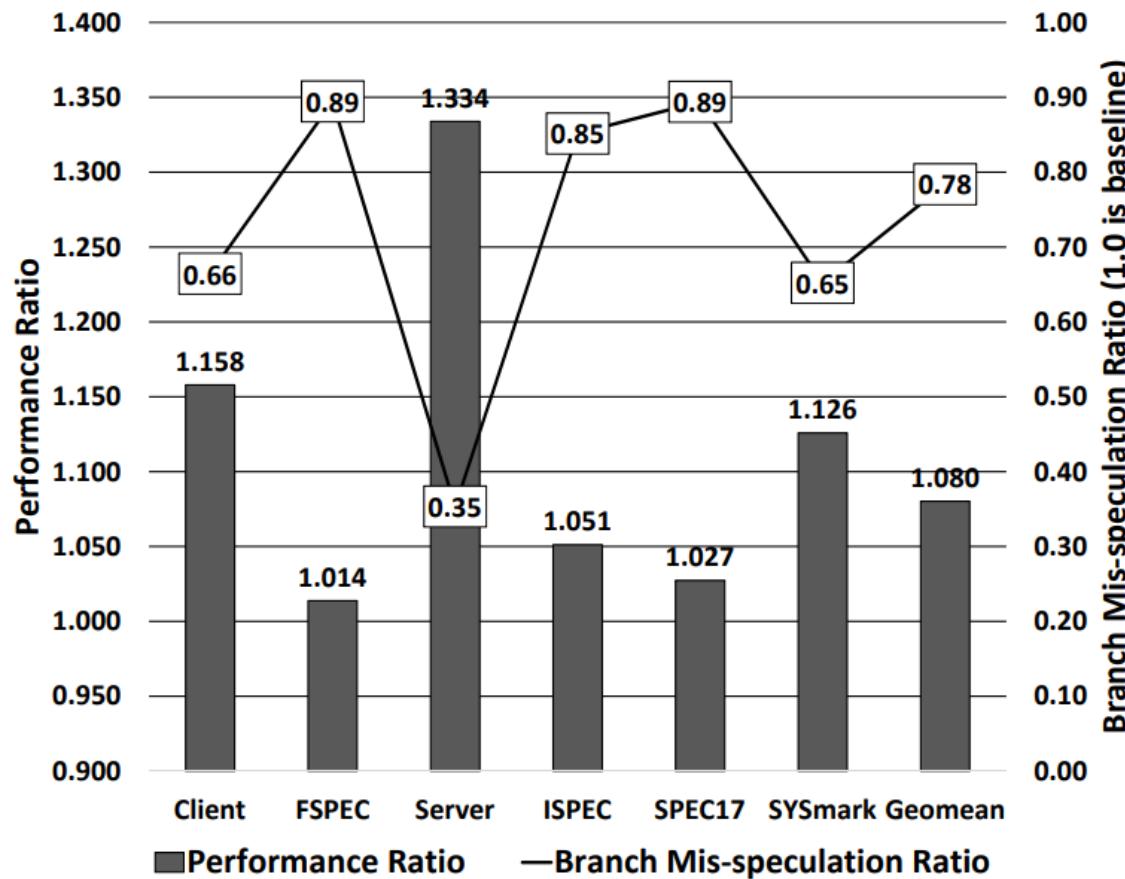


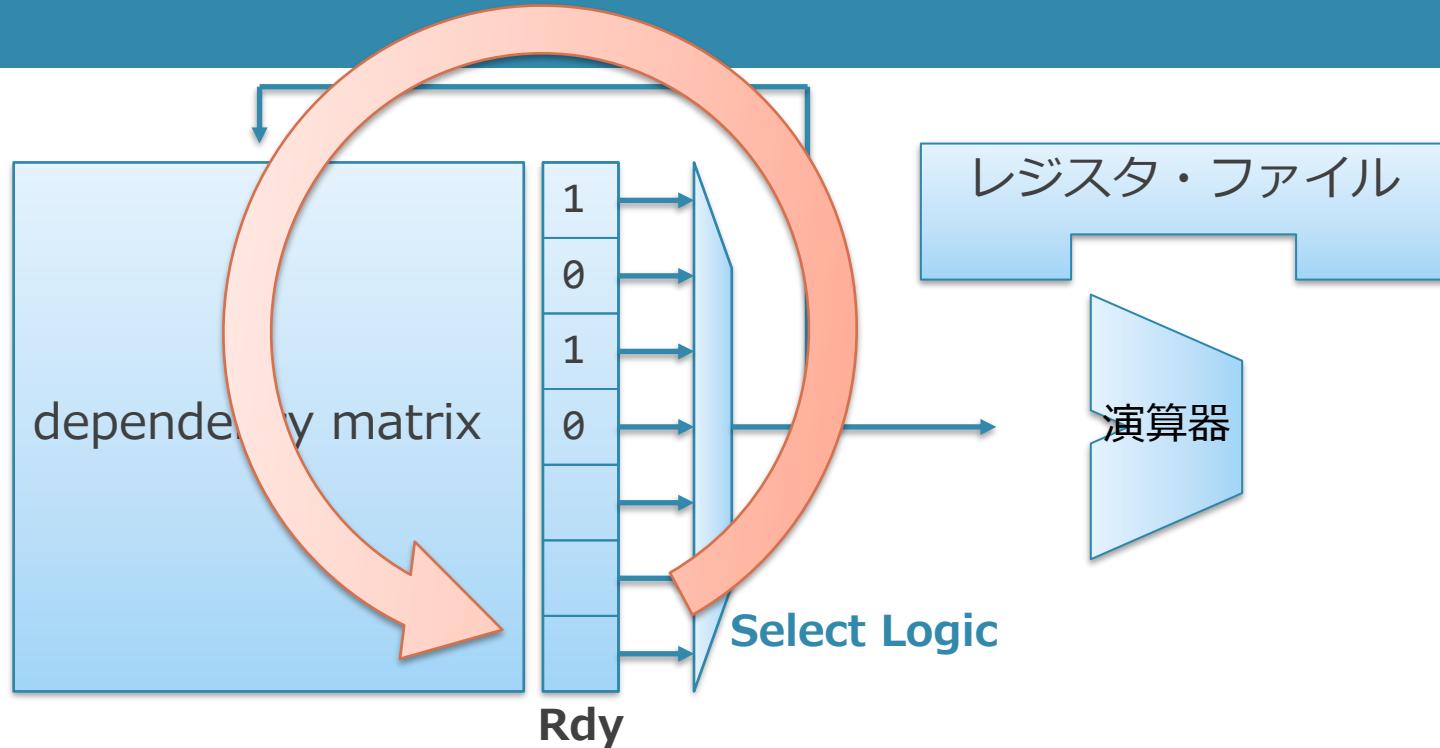
Fig. 6. All workloads (category-wise) results for ACB.

- 幾何平均で予測ミスを 22% 削減, 性能が 8% 向上

クリティカルな命令の優先発行

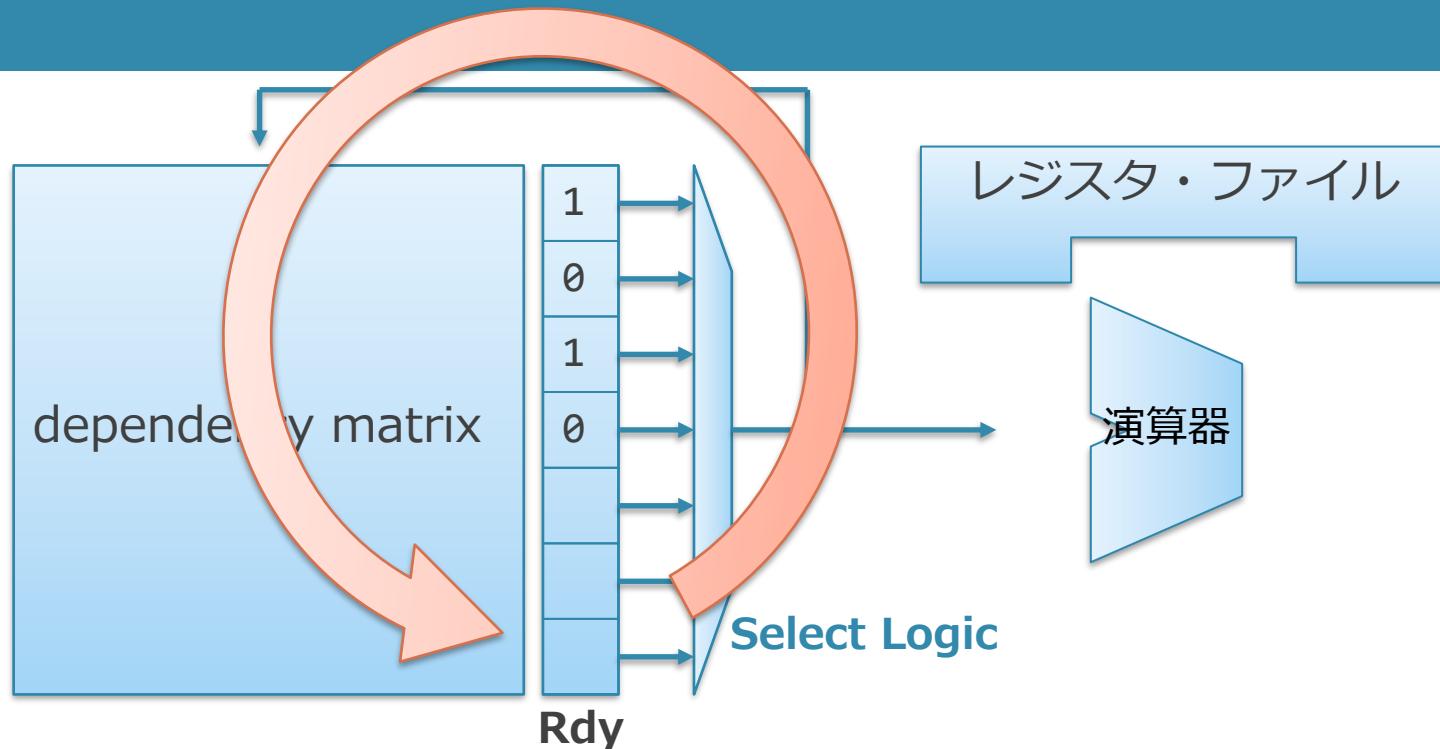
- Hideki Ando, "Performance Improvement by Prioritizing the Issue of the Instructions in Unconfident Branch Slices", MICRO 2018
- 名古屋大の安藤先生. 塩谷の元ボス

背景



- 発行時のセレクト論理は高速性が要求される
 - ◊ wakeup-select のループを 1 サイクル以内に終える必要がある
- 単純な回路が用いられる
 - ◊ 固定優先順位, (不完全な) プログラムオーダー順

背景



- たとえば単純にエントリ番号が若いものを優先した場合
 - ◊ 挿入位置はその時空いていたエントリなので、優先順位は実質ランダム
- 問題：予測ミスを起こす命令やその依存元が後回しにされた場合
 - ◊ 性能が大きく低下
 - ◊ スケジューリング可能命令数の増加と共に、影響が顕在化

アプローチ

- 方針：
 - ◊ 予測の確信度が低い分岐命令を検出
 - ◊ 低確信度の分岐命令とその先祖を優先して発行

実装

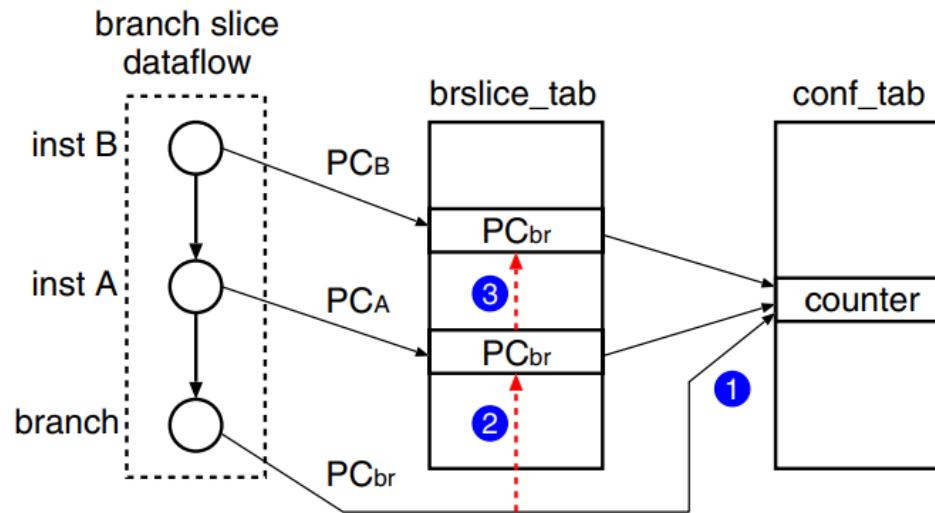
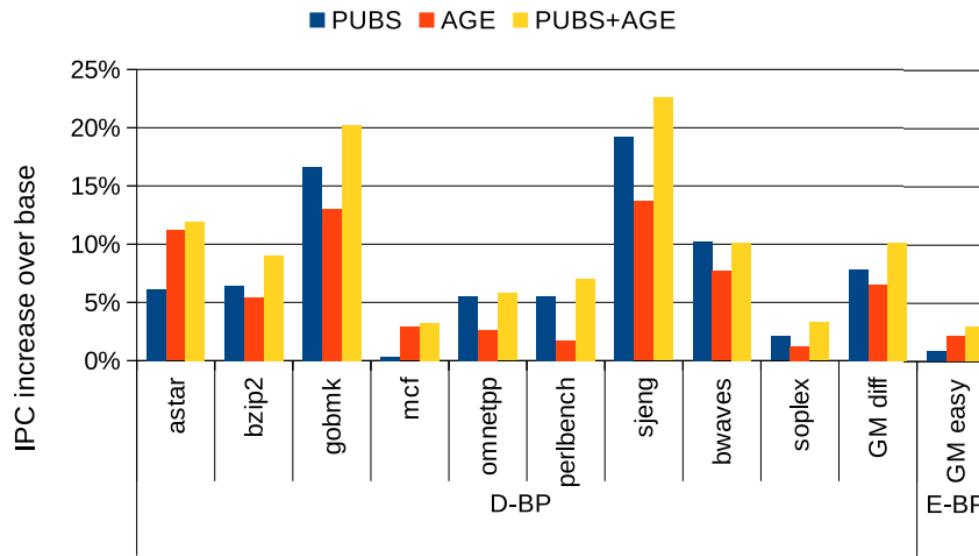


Fig. 3. Structures for predicting unconfident branch slice instructions.

■ アルゴリズム：

- ◊ 分岐命令ごとに分岐予測のヒット/ミス数をカウンタで計測 (①)
 - 低確信度な分岐命令を検出
- ◊ RMT を拡張等して、先祖に低確信度命令の PC を伝搬 (② ③)
 - 先祖は子孫の低確信度命令の PC を使ってカウンタにアクセス
- ◊ カウンタが閾値以下なら、発行キュー内の優先度の高いエントリに入れる

評価



■ 評価

(a) IPC

- ◊ パーセプトロン分岐予測を使用
- ◊ ランダム優先度, age matrix (プログラムオーダで古いものを優先) と提案 (PUBS) を比較

■ 5~10% 程度性能が向上

クリティカルな命令への対処

- 共通した話題：クリティカルな命令への対処
 1. 値予測, プレディケート変換, 命令発行方法
- なぜいまクリティカルな命令に着目しているのか？
 - ◊ CPU が大きくなると, 性能に特に効いてくる
 - ◊ スケジューリングのウィンドウサイズや, 同時発行幅
- 変化
 - ◊ 10年前：
 - Intel Sandy Bridge : 168 命令スケジューリング & 6 命令同時発行
 - ◊ いま：
 - Apple M1 : 630 命令スケジューリング & 17 命令同時発行
 - Intel の次期コア Golden Cove もこれに近い

話題

1. クリティカルな命令への対処
2. キャッシュ周りのあまり着目されてこなかった部分
 1. ストアバースト実行のプリフェッч
 2. TLB 置き換えアルゴリズム
3. コアアーキテクチャ自体の改良

ストア・バッファのためのプリフェッチ

- Juan M. Cebrian et al., "Boosting Store Buffer Efficiency with Store-Prefetch Bursts" MICRO 2020
- スペインのムルシア大学の人達

背景

- ストア・バッファ (Store Buffer: SB)
 - ◊ ストアのレイテンシは SB によって隠蔽
 - ◊ SB への書き込みのみでストアはコミットできる
- (この SB は文脈的にコア内の SQ を指しているものと思われる
 - ◊ SQ 上にコミット・ポインタと書き戻しポインタを持つ実装
- 問題：
 - ◊ SB がフルになるとストールが発生
 - ストアのキャッシュ・ミスなどにより起きる
 - ストール・サイクルの3分の1程度を占めることがある
 - ◊ 高速にフォワーディングを行うために、SB を大きくできない

アプローチ

- 観察：
 1. SB によるストールの大半は、ごく少数の命令に由来
 2. それらの大半はコピー操作等 (memcpy とか) であり、アドレスは容易に予測できる
 3. しかし強烈な積極度でプリフェッチを行わなければ遅延を隠蔽できない
 - ロードもストアも区別しない既存のプリフェッチではダメな理由
 - 普通はポリューションを恐れて、特に L1 ではそこまで強くしない
- 方針：
 - ◇ 連續領域への集中した書き込みがあるとストアのバーストとして検出
 - アンロールやインタリーブによる順序入れ替えを考慮する
 - ◇ 検出したら現在のそのページの残りを全部プリフェッチ

実装

- 構造 : Sat カウンタ (4 bits)
 - ◊ アクセスが「隣接」したラインへ遷移した回数を数える
 - ◊ 遠くに飛んだ場合はリセット
- 動作 : ストアのコミット時に直前ストアのライン・アドレスとの差分を見る
 - ◊ 0: 同じライン. 何もしない
 - ◊ 1: 隣のライン. Sat をインクリメント
 - ◊ それ以外 : Sat をリセット
 - アクセスされるライン・アドレスが +0～+1 の範囲から外れるときリセット
 - つまり, ストアが局所的に集中しなければカウンタは増えない
- 一定ストア毎に, カウンタの値が閾値以上であればバースト検出

評価

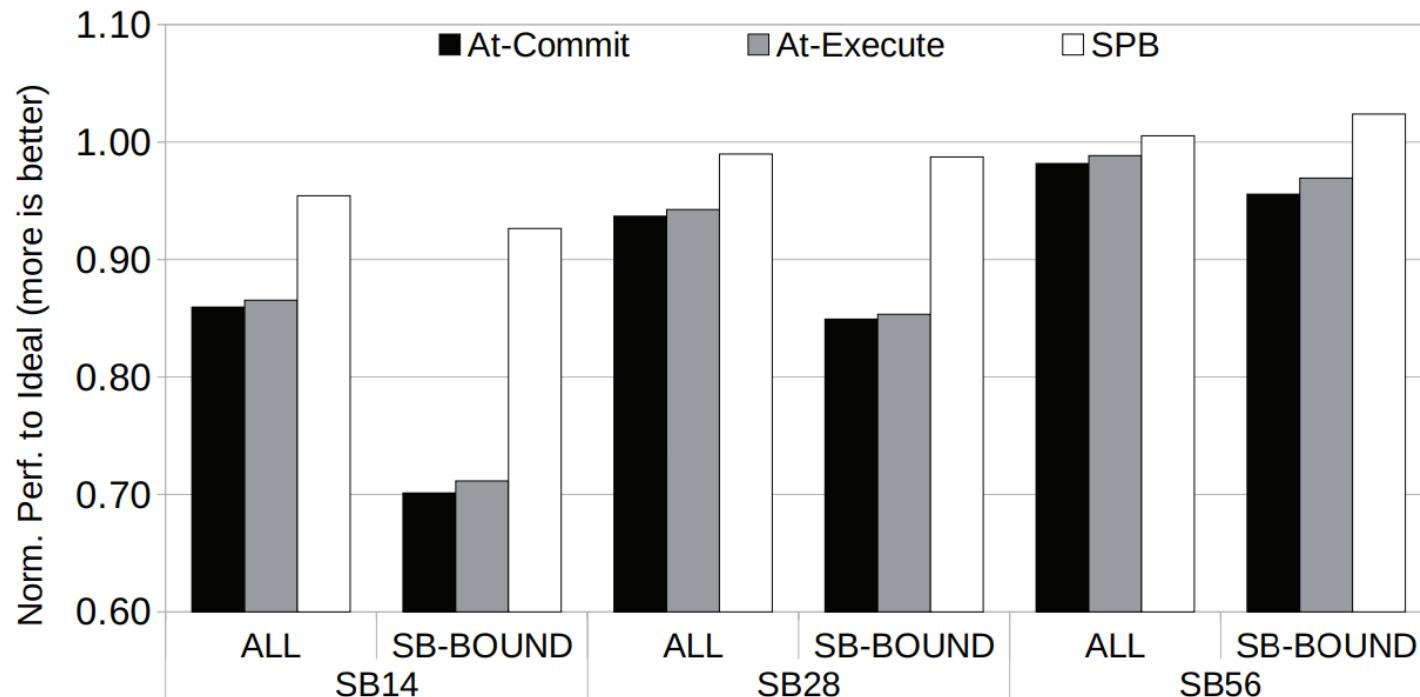


Fig. 5: Norm. performance to Ideal for different SB sizes

- ◇ 理想モデルを 1.00 とした相対値（数字が高いほど高性能）
- ◇ コミット時や実行時にプリフェッチを行うモデルと比較
- ◇ 特に SB が小さい時に大きく性能を向上

TLB 置き換えアルゴリズム

- S. Mirbagher-Ajorpaz et al., "CHiRP: Control-Flow History Reuse Prediction", MICRO 2020
- テキサスA&M大学

背景

- TLB (Translation Lookaside Buffer)
 - ◊ アドレス変換のためのページ・テーブルのキャッシュ
- ミス・ペナルティは大きい
 - ◊ L2 TLB のミス・ペナルティ：
 - 212 cycles for Skylake (2015)
 - 272 cycles for Broadwell Xeon (2016)
 - 230 cycles for Coffee Lake (2017)
- ワーキング・セットの増大により、TLB ヒット率がより重要に
 - ◊ 今日ではシステム実行時間の 20%-50% を TLB ミスが占めている

アプローチ

- TLB の置き換えアルゴリズムは、そもそもあまり研究されていない
- 方針：再利用予測に基づく置き換え
 - ◊ エントリが再び使用されるかどうかを予測
 - ◊ LLC や BTB の置き換えアルゴリズムでは良く使用されている
 1. エントリの再利用性と強く相関を持つ特徴を利用

TLB のエントリの再利用性に相関のある特徴

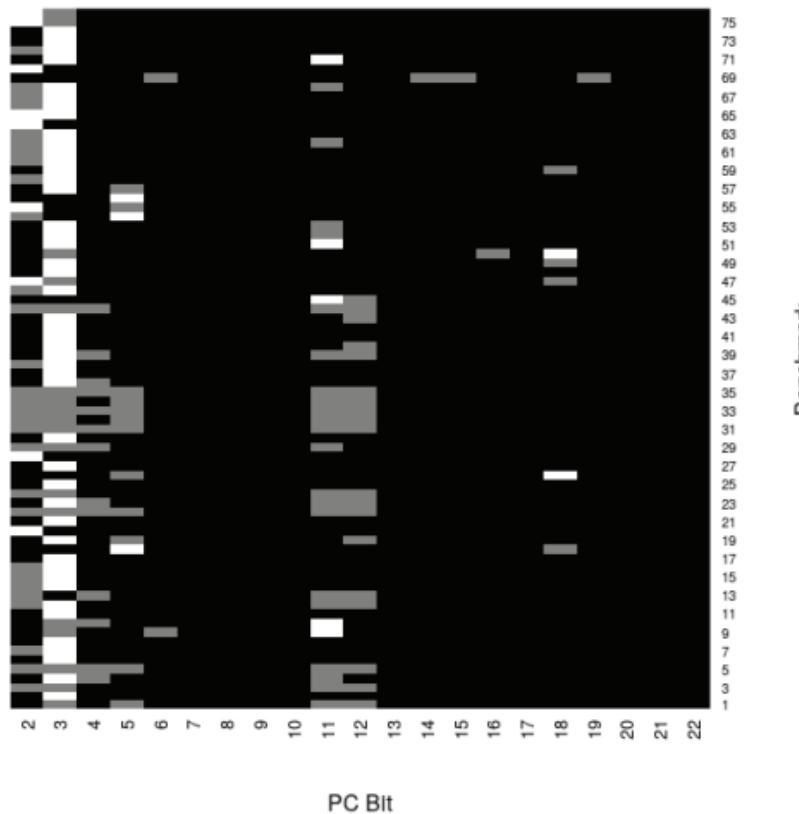


Fig. 3. Each row represents an offline-trained ADALINE weight vector for one benchmark. The *x*-axis shows the PC bit used as input. The white boxes show reuse prediction in TLB entries is strongly correlated with bits 2 and bit 3 of the PC.

- PC の 2, 3 ビット目は再参照の有無に関して強い相関を持つ

実装

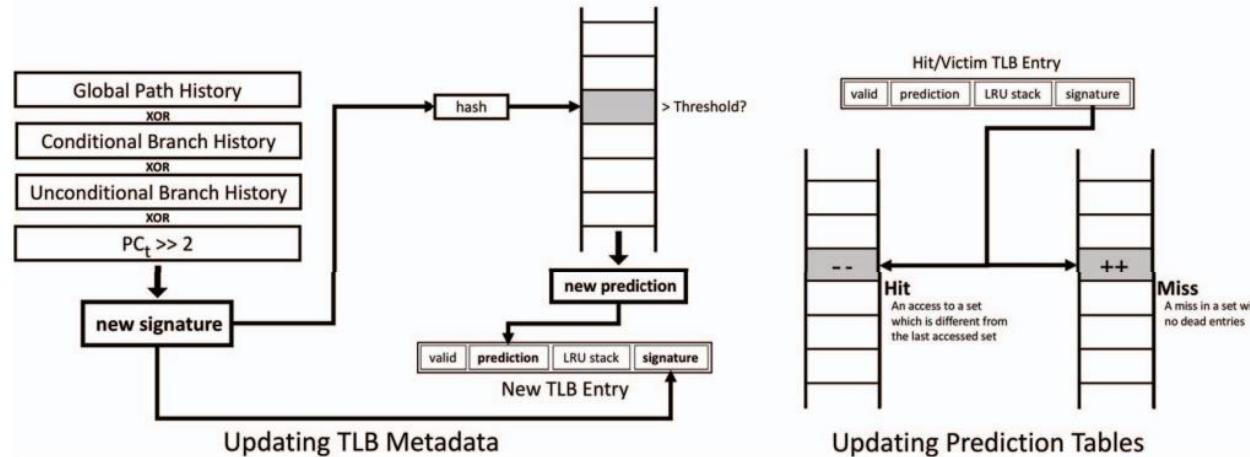


Fig. 4. CHiRP TLB metadata and prediction table update flow using a signature.

1. 相関を持つ特徴をハッシュ関数で畳み込んでシグニチャを計算
2. アクセス毎に、最後に触った命令のシグニチャを TLB のエントリに記録
3. 学習：「エントリに最後に触ったのはあのシグニチャの時」を学習する
 1. TLB から追い出されたエントリに記録されたシグニチャに対応したテーブルのカウンタをインクリメント
4. 予測：シグニチャからテーブルを読み出す
 1. カウンタが閾値以上なら dead なので、優先的に追い出す

評価：1024エントリ 8way L2 TLB での MPKI (misses per kilo instructions) と性能

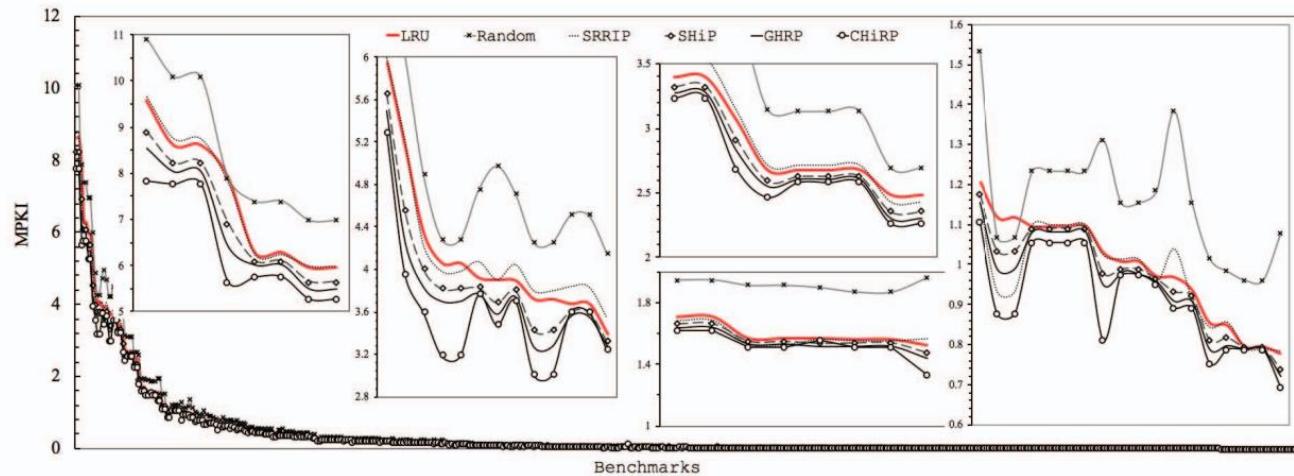


Fig. 7. MPKI comparison of various policies. The horizontal axis shows the benchmarks in the order of sorted MPKI for LRU. Multiple zoomed-in areas of the graph are shown in insets.

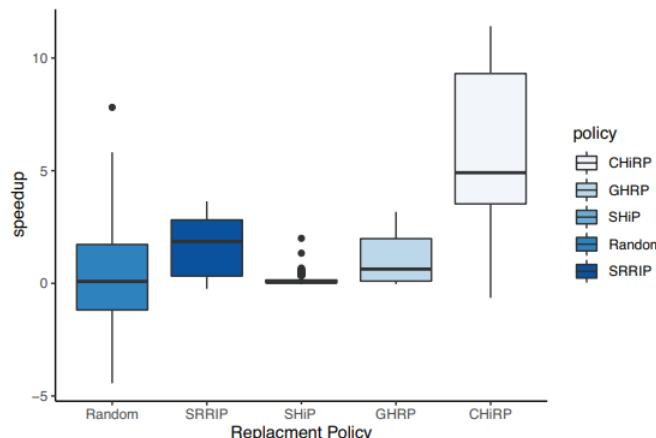


Fig. 8. Speedup for 870 traces.

- ◇ 平均 MPKI :
 - 1.51 (LRU), 1.47 (Random), 1.35 (SRRIP),
1.08 (CHiRP)
- ◇ 性能 :
 - 平均で 4.8% 向上
 - メモリ・インテンシブなベンチマークでは 10% 程度向上
- ◇ 8KB (L2 TLB の 10% 程度の大きさ) のテーブルを使った場合の結果
 - 256B でもそこそこきちんと働く

キャッシュ周りのあまり着目されてこなかった部分

- キャッシュ周りのあまり着目されてこなかった部分
 1. ストアバースト実行のプリフェッチ
 2. TLB 置き換えアルゴリズム
- CPU 全体の速度向上による
 - ◊ 今までボトルネックとなり得なかった部分がボトルネックに

話題

1. クリティカルな命令への対処
2. キャッシュ周りのあまり着目されてこなかった部分
3. コアアーキテクチャ自体の改良
 1. リネームの省略
 2. 高性能な in-order 実行
 3. 投機的なベクトル命令実行

レジスタ・リネームの省略

- Hidetsugu Irie, Toru Koizumi, Akifumi Fukuda, Seiya Akaki, Satoshi Nakae, Yutaro Bessho, Ryota Shioya, Takahiro Notsu, Katsuhiro Yoda, Teruo Ishihara, and Shuichi Sakai:
"STRAIGHT: Hazardless Processor Architecture without Register Renaming", MICRO 2018
- 東大 入江先生, 塩谷, 富士通の方など

背景：レジスタ・リネーミング

- OoO プロセッサのボトルネックの一つ
 - ◊ 多数の命令を同時にリネームするためには非常に多ポートの RAM が必要
 - ◊ 必要ポート数 : 同時リネーム命令数 × (総オペランド数 + 1)
- 最先端の商用プロセッサでもリネーム幅はボトルネック

	Intel Sunny Cove	AMD Zen2	IBM POWER9
フェッチ幅	6	8	8
リネーム幅	5	6	6
発行幅	8	7	9

発行幅は FP を除く

通常のレジスタ・リネーム

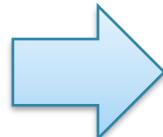
- 方針：レジスタの名前を付け替える
 - ◊ 偽の依存の原因 = 同じレジスタの使い回し
 - 上書きにより破壊される
 - ◊ ディスティネーションにその命令専用のレジスタを動的に毎回新しく割り当てる
 - ◊ レジスタ番号がかぶらないので、他の命令との間で出力依存や逆依存は生じなくなる

I1: mul $x_3 \leftarrow x_2 * 4$

I2: add $x_3 \leftarrow x_1 + 1$

I3: sub $x_1 \leftarrow x_5 - 1$

I4: and $x_6 \leftarrow x_7 \& 1$



I1: mul $p_{20} \leftarrow p_{12} * 4$

I2: add $p_{21} \leftarrow p_{11} + 1$

I3: sub $p_{22} \leftarrow p_{15} - 1$

I4: and $p_{23} \leftarrow p_{17} \& 1$

STRAIGHT アーキテクチャ

RISC

```
ADDi r1, $zero, 5  
ADDi r10, $zero, 8  
ADD r2, r1, r10
```

STRAIGHT

```
ADDi $zero, 5  
ADDi $zero, 8  
ADD [2], [1]
```



- レジスタ番号ではなく距離でオペラントを指定する ISA
 - ◊ ディスティネーションを命令と 1 : 1 でシーケンシャルに割り当てる
 - 物理レジスタはリングバッファ状の構造を取る
 - リングを 1 周しなければ）同じレジスタの使い回しがなくなる
 - ◊ ソースを「何命令前の結果」のように相対距離で指定
 - 命令間の相対距離は不变
- レジスタ・リネーミングなしに高効率な out-of-order 実行を実現
 - ◊ レジスタへの上書きが存在しないため、偽の依存が生じない

実はけっこう歴史が長い

1. 五島 正裕 他, 「Dualflow アーキテクチャの命令発行機構」, 情報処理学会論文誌 2001
 - ◊ リングバッファと距離で指定することによる方式は, この時点で発明
 - ◊ ソースの参照ではなく, ディスティネーションの送信先を命令間距離で指定
2. 一林 宏憲 他, 「逆 Dualflow アーキテクチャ」, ACS 論文誌 2008
 - ◊ 現在の STRAIGHT と同様にソースを距離で表す方式の発明
 - ◊ 通常の命令セットからハードウェアで動的に変換し, キャッシュする
3. Ryota Shioya et al., "Energy Efficiency Improvement of Renamed Trace Cache through the Reduction of Dependent Path Length", ICCD 2014
 - ◊ 変換結果をキャッシュする際のオーバーヘッド削減
4. Hidetsugu Irie et al., "STRAIGHT: Hazardless Processor Architecture without Register Renaming", MICRO 2018
 - ◊ 動的な変換ではなく, 静的にコードを生成する方式に
 - ◊ コード生成方法などを定式化

高性能な in-order 実行

- Ipoom Jeong et al., "CASINO Core Microarchitecture: Generating Out-of-Order Schedules Using Cascaded In-Order Scheduling Windows", HPCA 2020
- 韓国 延世大学とサムスンのチーム

- OoO プロセッサ
 - ◊ 高いシングルスレッド性能を得るために欠かせない
- 問題：電力効率がよくない
 - ◊ スケジューラやロード・ストア・キューなど
 - ◊ 多ポートの RAM や CAM からなる

アプローチ

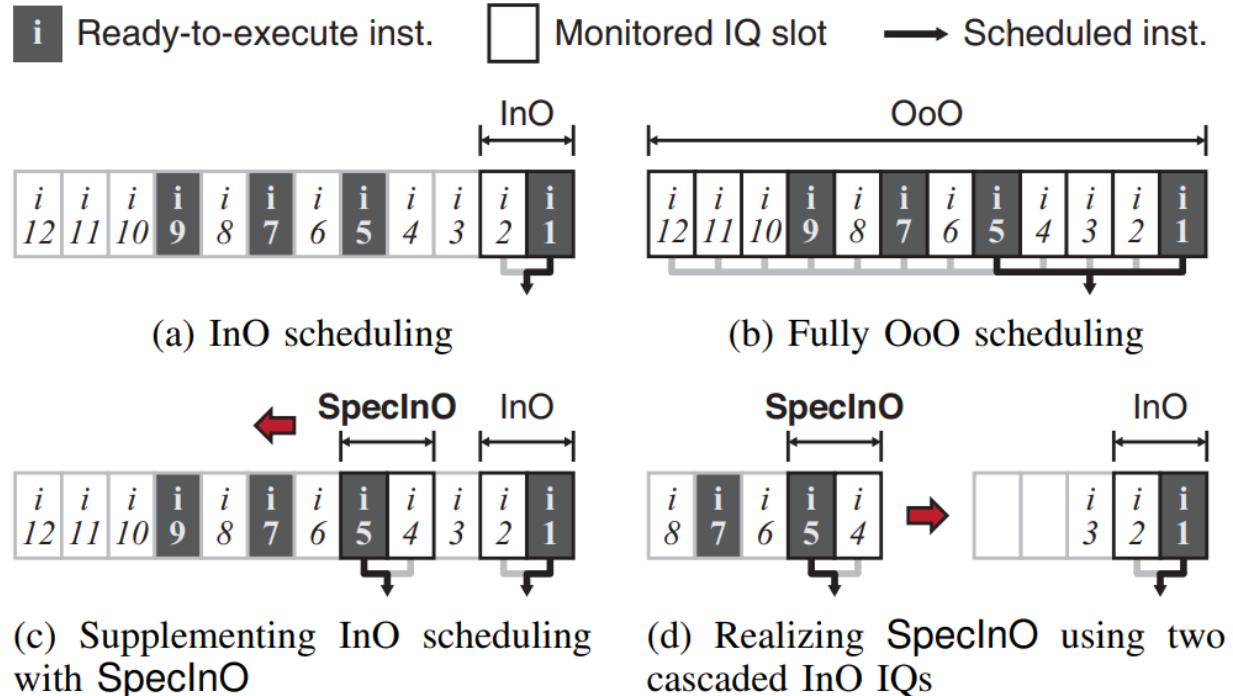


Figure 1: Examples of instruction scheduling schemes

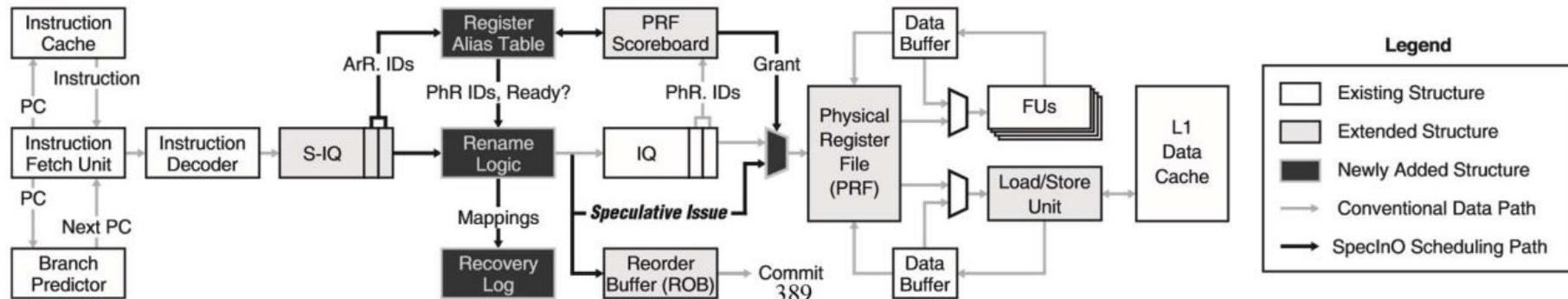
InO と OoO

- ◊ (a) InO はプログラムオーダの最も古い部分だけを実行できる
- ◊ (b) OoO は (a) InO と違い後続の命令 (i5,i7,i9) を実行できる

方針 :

- ◊ (c) InO に加えて、実行できる若い部分だけを投機的に実行 (SpecInO)
- ◊ (d) SpecInO と InO をカスケードして実行
- SpecInO すぐに実行できない長レイテンシを後ろの InO に送る

S-IQ と IQ がカスケードされた構造 (CAScaded IN-Order: CASINO)



- デコードされた命令は S-IQ に入る
 - ◊ 既に依存が満たされている場合は、そこで投機的に発行
 - ◊ メモリに関して投機をかけている（だと思う）
 - ◊ LSQ を使わずに、ロード再実行系の投機/検証を行う
- S-IQ ですぐ発行できなかった命令は S-IQ を抜けて IQ に
 - ◊ ここでは in-order に実行

評価結果

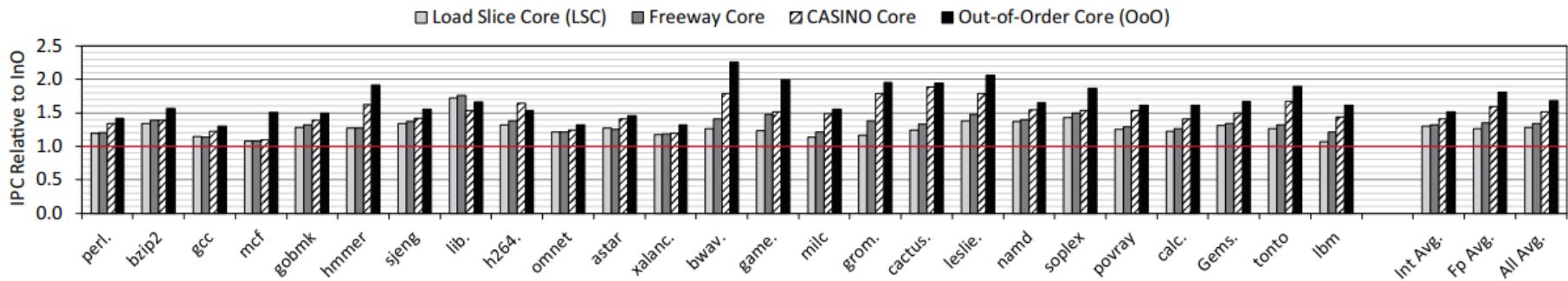


Figure 6: IPC comparison between LSC [15], Freeway [16], CASINO, and OoO normalized to the baseline InO

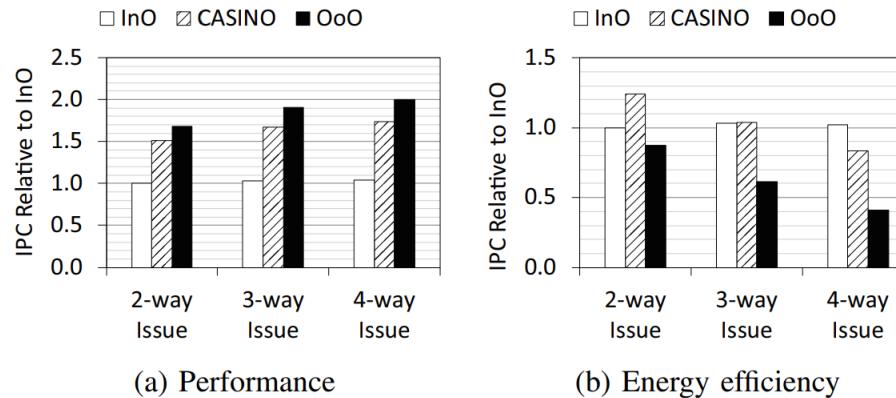


Figure 11: Evaluation on wider-issue designs

- InO よりは大分性能が高く、OoO に迫る性能が出せる

関連研究：InO パイプがカスケードされたアーキテクチャ

Flea-flicker
InO + InO

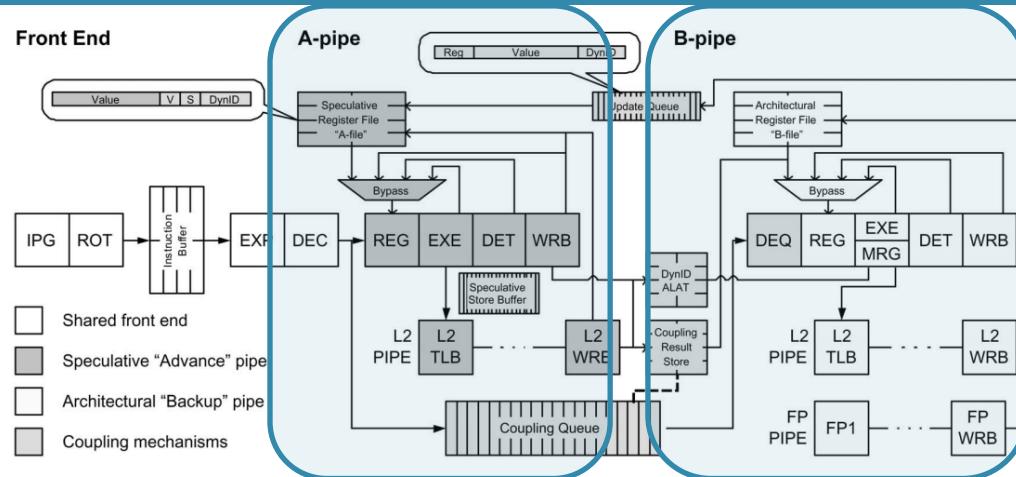


Fig. 3. Two-pass pipeline design.

FXA
多段 InO + 小 OoO

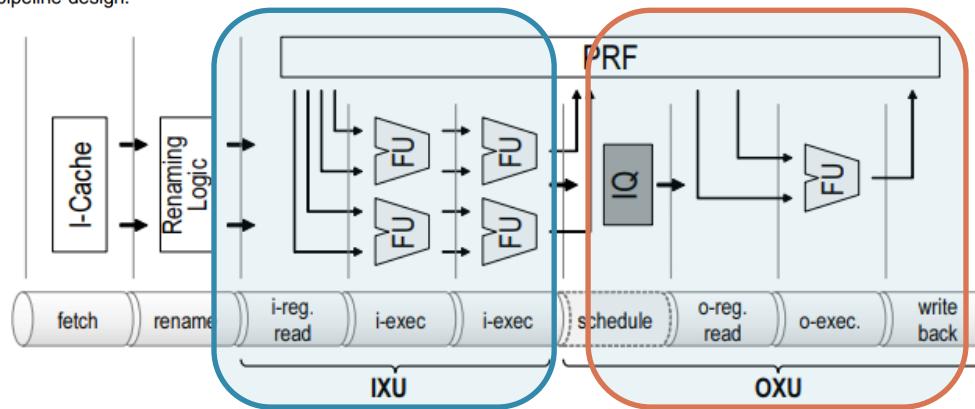


Figure 2: Front-end execution architecture.

- ◇ Ronald D. Barnes et al. "Flea-flicker" Multipass Pipelining: An Alternative to the High-Power Out-of-Order Offense, MICRO 2005
- ◇ Ryota Shioya et al., "A Front-end Execution Architecture for High Energy Efficiency", MICRO 2014

投機的なベクトル実行

- Peng Sun et al., "Speculative Vectorisation with Selective Replay", ISCA 2021
- ケンブリッジ大学と ARM のチーム

背景

- SIMD 命令
 - ◊ AVX, SVE など
 - ◊ 基本的には人手で使うもの
 - ◊ HPC アプリでは一部コンパイラで対応できる
- 汎用アプリのコンパイラによる自動 SIMD 命令化はうまくいかない
 - ◊ 安全とわかる場合しか変換できない
 - ◊ コンパイラによるメモリのエイリアス解析の限界

```
1 /* Read integers from the standard input. */
2 int *x = read();
3 for (i = 0; i < N; i++) {
4     a[x[i]] = a[i] + 2;
5 }
```

Listing 1: Example code with read-after-write cross-iteration dependences every four iterations when `read()` returns {3, 0, 1, 2, 7, 4, 5, 6, 11, 8, 9, 10...}.

アプローチ

- 観察：SPEC CPU などのアプリを解析
 - ◊ すべてのループを SIMD 命令化できた場合は速度が平均 2.1 倍に
 - ◊ メモリ依存が不明なループを除外した場合は 1.02 倍
 - ◊ SIMD 命令化されないループの 70% 以上がメモリ依存を持つ
- 方針：ハードウェアによる投機的実行と選択的に再実行
 - ◊ SIMD 命令化されたコードの領域を投機的に実行
 - ◊ アクセスされたメモリ・アドレスを監視し、依存の違反を特定
 - ◊ 誤ったデータを受け取ったレーンだけを選択的に再実行

実装

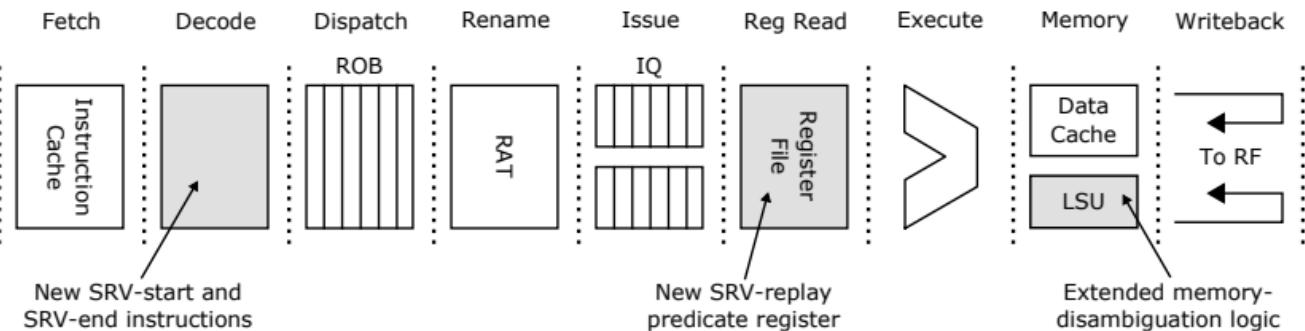


Fig. 1: Logic for SRV added to a standard out-of-order superscalar pipeline.

```
1 Loop:  
2   srv_start  
3   v_load  v0, a[i:i+15]  
4   v_add    v0, 2  
5   scatter v0, a[x[i]:x[i+15]]  
6   srv_end  
7   inc     i, 16  
8   comp    i, N  
9   bne    Loop
```

Listing 2: Pseudo-code for listing 1 using SRV.

- `srv_start/srv_end` で挟まれた区間を投機実行
 - ◊ 特定の SIMD 命令などのみが実行可能
- LSU で依存違反を動的に検出
 - ◊ 依存違反があった場合、その命令から違反があったレーンだけリプロセス

評価結果

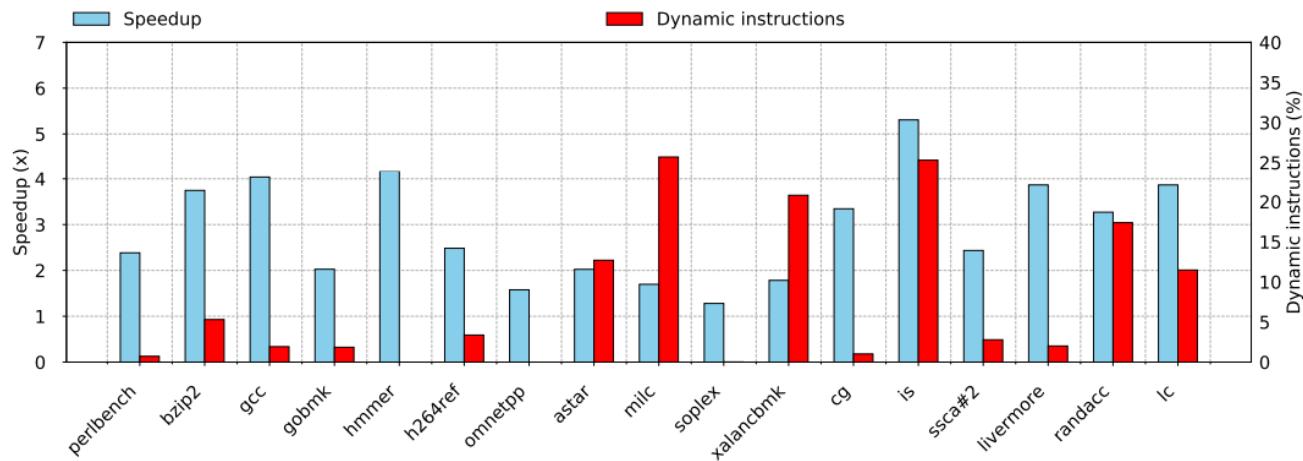


Fig. 6: Per-loop speedup for all SRV-vectorisable loops in each benchmark and their corresponding coverage in dynamic instructions compared to a baseline out-of-order microarchitecture.

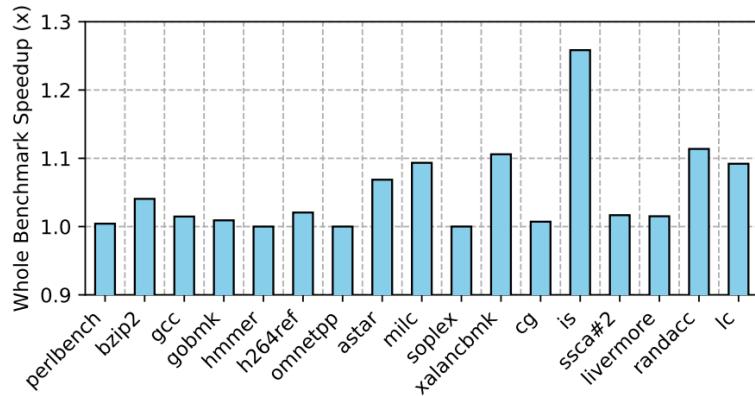


Fig. 7: Whole-program speedup for each benchmark compared to vectorisation with SVE.

- ◊ ループ単位では大きく性能が上がっているものがある
- ◊ オーバーオールでも 10% 程度の速度向上に繋がるものも
- ◊ SPEC CPU int でも効果がでている

コアアーキテクチャ自体の改良

- コアアーキテクチャ自体の改良
 - ◊ リネームの省略
 - ◊ 高性能な in-order 実行
 - ◊ 投機的なベクトル命令実行

今日取り上げなかつた，それ以外の話題

- マイクロアーキテクチャレベルでの攻撃と，その対策の話は多い
 - ◊ Spectre/Meltdown 等の投機的な実行を利用したもの
 - ◊ スレッド間で共有される資源を介したもの
- データや命令のプリフェッチ
 - ◊ Data Prefetch Championship 3,
Instruction Prefetch Championship 1
- メモリ圧縮，マイクロ命令，予測器，run-ahead 実行
- 一時に比べると，マイクロアーキテクチャの話は割と増えた

まとめ

■ 今日の内容

1. プログラムの複雑化とシングルスレッド性能の向上
2. 「現代の」 Out-of-order スーパスカラ・プロセッサの構造
3. OoO プロセッサに関する最近の研究

まとめ

- 今後の展望：
 - ◊ 当面はボトルネック解消の色々な技術の研究が進む
 - 個々の技術の効果は一見それほど大きくない (5~10%)
 - 各技術は基本的に直交しているため,
この 5~10% の積み重ねが効いていく
 - ◊ 長期的には、自動マルチスレッド化を目指すことになると思う
 - 使える回路資源自体は大量にある
 - 回路量に性能をスケールさせる方法が、まだない
 - プログラミング言語のモデルも含めたアプローチが必要かも