

# コンパイラとランタイムによる ソフトウェアキャッシュの 更新オーバヘッド隠蔽手法

三好 健文

電気通信大学/JST

吉瀬 謙二

東京工業大学

入江 英嗣

電気通信大学

吉永 努

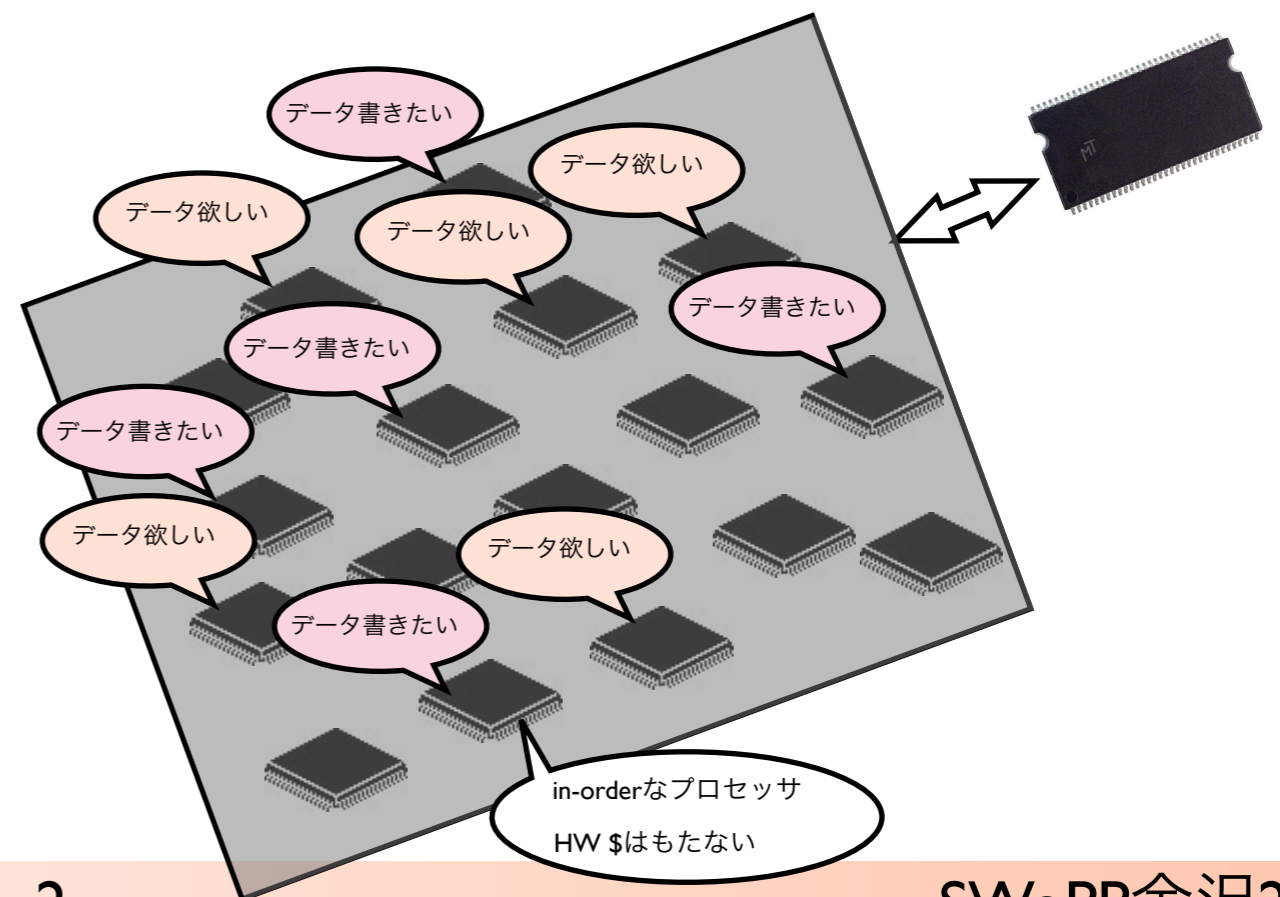
電気通信大学

# メニーコアプロセッサ時代のメモリ問題

🔍 コア数 >> メモリアクセス口数

▶ Cell/B.E.(7-8 SPEs v.s. 1メモリコントローラ)

▶ SCC(48-cores v.s. 4メモリコントローラ)

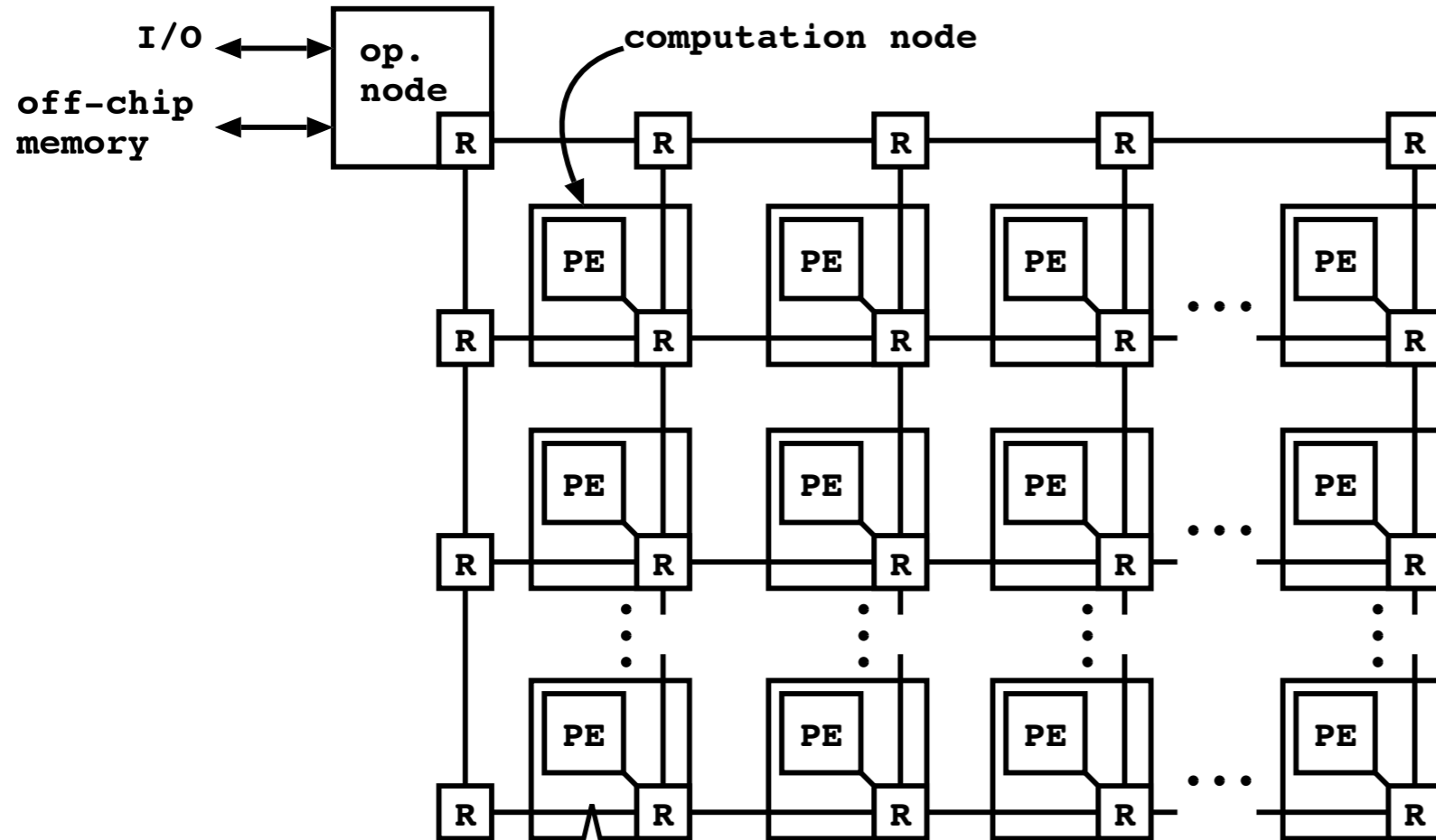


# どのくらい問題か？

- 🔊 衝突がある/ない時の実行時間を比較
  - ▶ 評価環境
    - ▶ M-Coreアーキテクチャ<sup>[14]</sup>
    - ▶ ベンチマークソフトウェア
      - ▶ seq/mm/fftを各コアで実行
      - ▶ 各プログラムはSW\$でデータ転送

# 予備実験 実験環境

## 🔍 ベースシステム(= M-Coreアーキテクチャ [14])



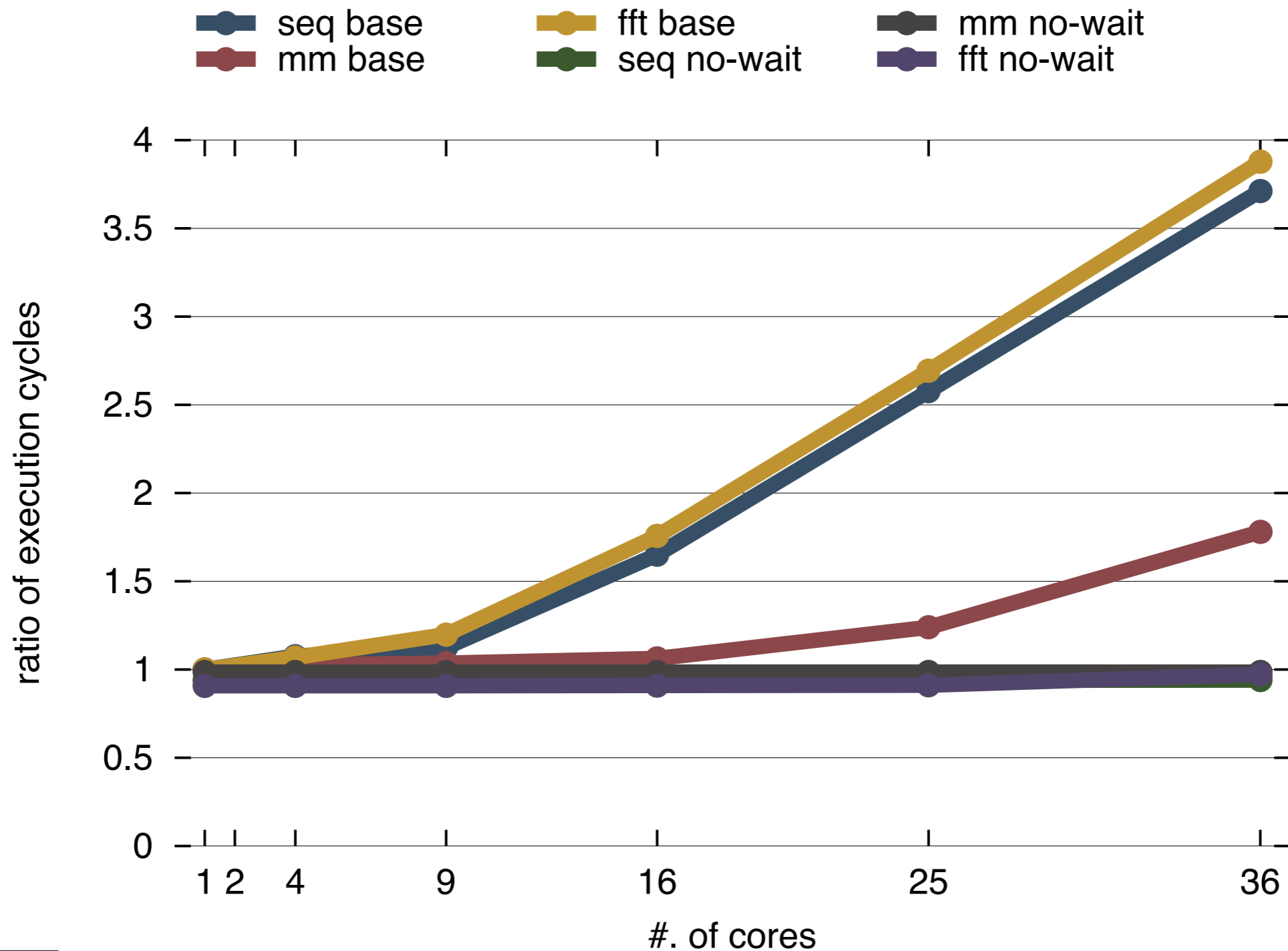
(1) seq	32M write/read	12.5%
(2) mm	256x256の行列の積	4.4%
(3) fft	512K点FFT	14.4%

命令セット	MIPS32互換
オフチップメモリ	4GB
	100サイクル
ローカルメモリ	256KB
	1サイクル
ルーティング	1サイクル

SW \$ 方式	4-way set-associative
	ライトバック
SW \$ 容量	32KB
SW \$ ライン幅	32B

# 予備実験 結果

## 📌 衝突がある(=base)/ない(=no-wait)時



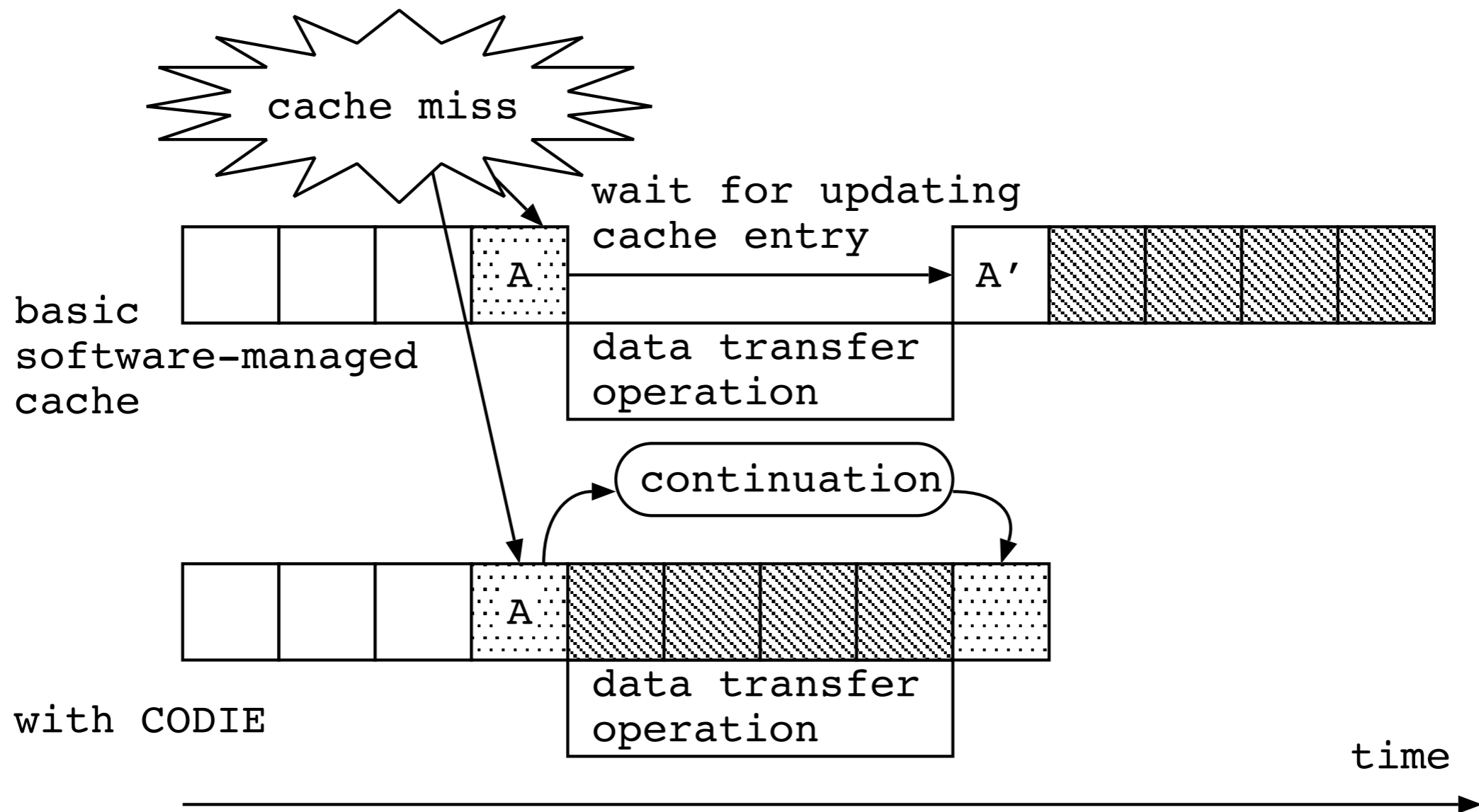
# 衝突の影響を軽減したいけど

- ▶ プログラミングは難しくせず
- ▶ SW \$で楽々メモリアクセス
- ▶ チューニング不要
- ▶ 追加HWの必要もないまま
- ▶ 単なるin-orderプロセッサ

# 通信オーバーヘッド削減手法:CODIE

- ▶ ソフトウェアキャッシュをベース
- ▶ キャッシュミスに係る命令を再実行
- ▶ コンパイル時に...
  - ▶ キャッシュミスに係る命令を抽出
  - ▶ 再実行コードの埋込み

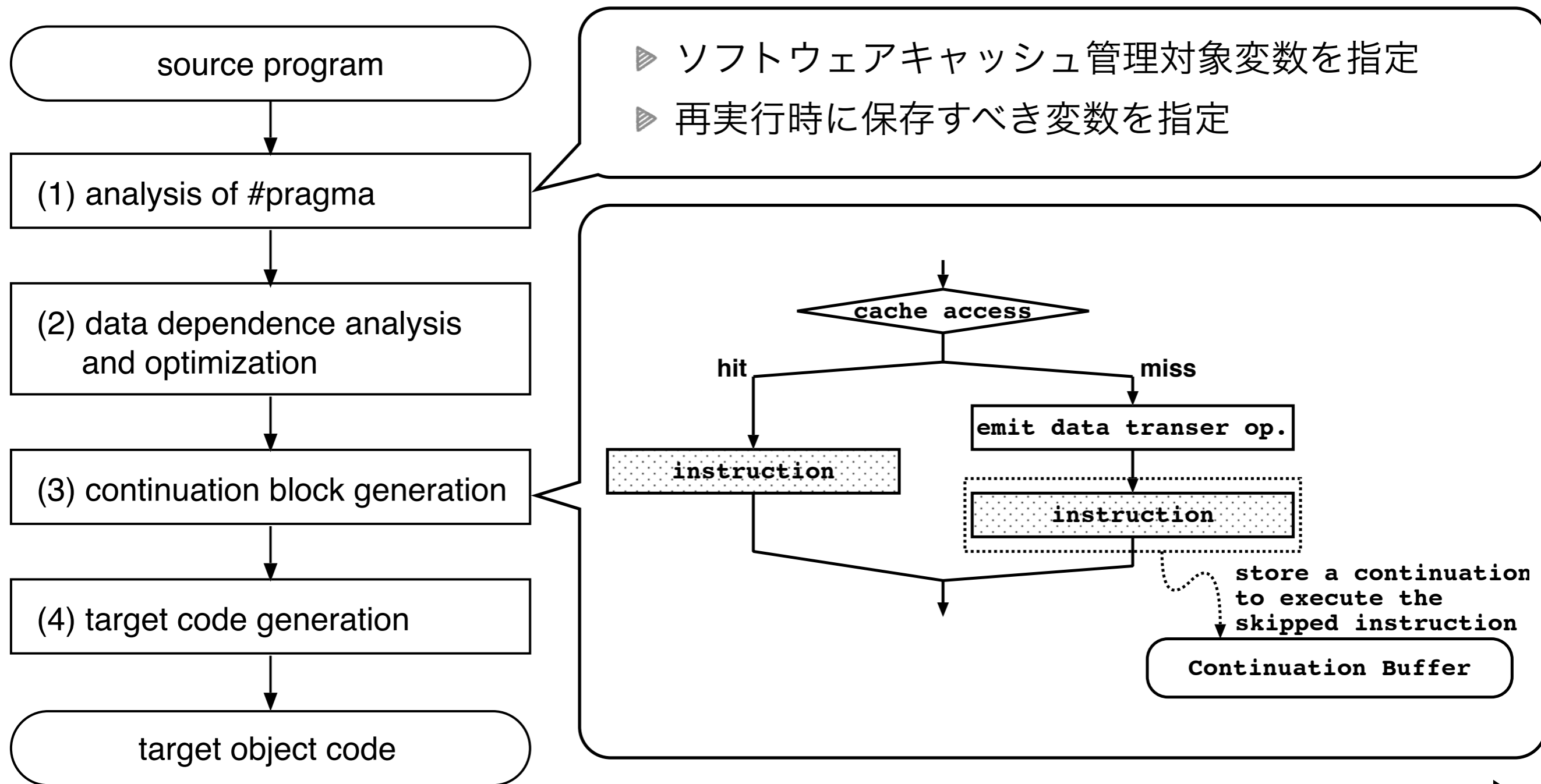
# CODIEによる命令実行の流れ

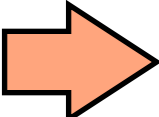




# CODIEの埋め込み手法

## 📌 コンパイルフローと再実行コードの埋込み



次ページにサンプル 

# CODIE適用例

```
#pragma codie cache(a, 32, 10)
volatile int a[10];

#pragma codie context main(i)
int main(){
    volatile int i = 0;
    volatile int tmp;
    tmp = a[i];
    printf("a[%d] = %d\n", i, tmp);
    printf("Hello World\n");
    return 0;
}
```

```
typedef struct{
    void *codie_ptr;
    int status;
    int i;
} main_context_t;
```

```
if(cache_read_int_only_accessible(i, &tmp)){
    printf("a[%d] = %d\n", i, tmp);
}else{
    CHECK_CODIE_BUFFER_FULL();
    codie = &main_context_buffer[CODIE_BUFFER_NEXT];
main_label_0:
    switch(codie->status){
        case 0:
            cache_read(i);
            codie->i = i;
            codie->status = 1;
            codie->codie_ptr = &&main_label_0;
            POINTER_INCREMENT(CODIE_BUFFER_NEXT);
            break;
        case 1:
            if(cache_read_int_only_accessible(i, &tmp)){
                printf("a[%d] = %d\n", codie->i, tmp);
                codie->status = 0;
                POINTER_INCREMENT(CODIE_BUFFER_TOP);
            }
            goto *codie_normal_flow;
    }
}
```

# 再実行のタイミング

- ▶ バッファがいっぱいになったとき
  - ▶ バッファトップを一つ実行
- ▶ 関数~~呼び出し~~呼び出し元に返る直前
  - ▶ バッファは関数フレームで保持
  - ▶ バッファがすべてクリアできるまで

# 再実行の仕組み

## ▶ バッファトップを一つ実行

```
#define CHECK_CODIE_BUFFER_FULL() \  
{ \  
    volatile void *macro_codie_tmp = codie_normal_flow; \  
    main_buffer_check: \  
        codie_normal_flow_ptr = &&main_buffer_check; \  
        if(__builtin_expect(CODIE_BUFFER_TOP == CODIE_BUFFER_NEXT, 0)){ \  
            codie = &main_context_buffer[CODIE_BUFFER_TOP]; \  
            goto *(codie->codie_ptr); \  
        } \  
        codie_normal_flow = macro_codie_tmp; \  
}
```

## ▶ バッファをクリア

```
#define MAIN_CODIE_FLUSH() \  
{ \  
    codie_epilogue: \  
        codie_normal_flow = &&codie_epilogue; \  
        if(CODIE_BUFFER_TOP != CODIE_ENTRY_INDEX){ \  
            codie = &main_context_buffer[CODIE_BUFFER_TOP]; \  
            goto *(codie->codie_ptr); \  
        } \  
}
```

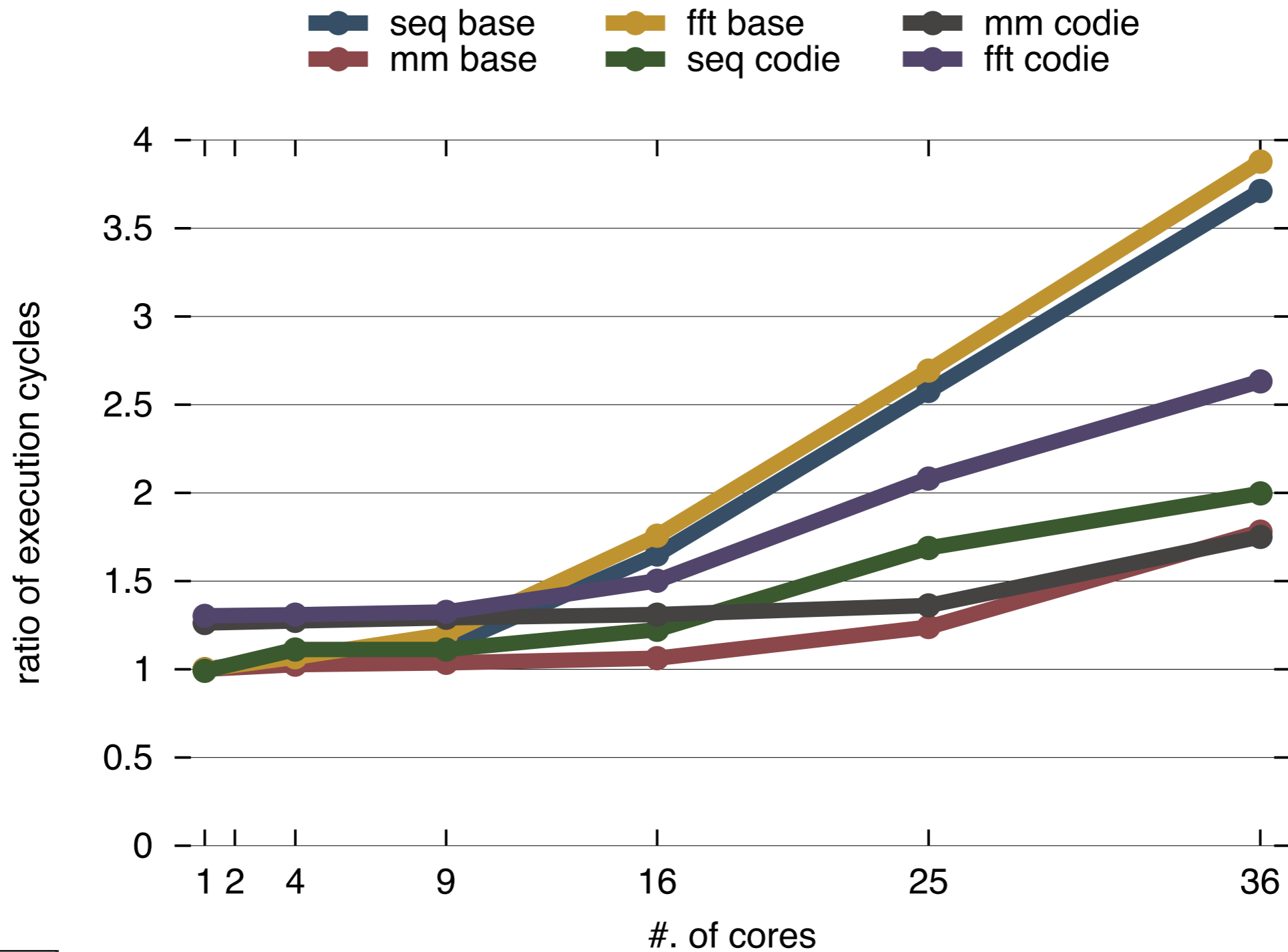
- ▶ 継続の融合
  - ▶ 複数データを要する式に対し一括で継続/再実行
- ▶ 命令スケジューリング
  - ▶ キャッシュアクセスする命令を適度に分散させる

# 実験と結果

- ▶ 対象アーキテクチャ
  - ▶ M-Coreアーキテクチャ
  - ▶ Cell/B.E.(@PS3®)
- ▶ ベンチマークプログラム
  - ▶ マイクロベンチマーク (seq/mm/fft)

# 実験と結果

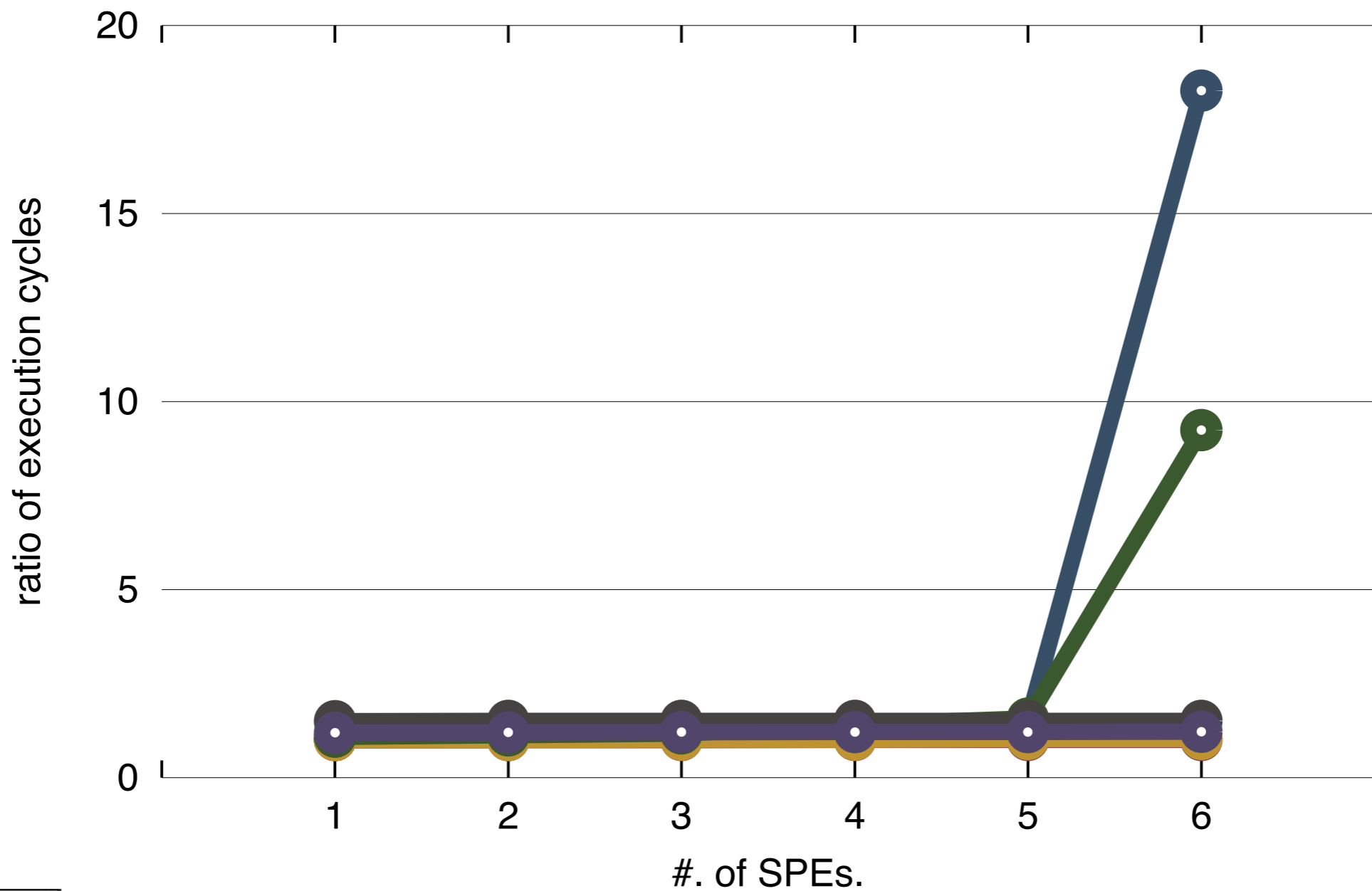
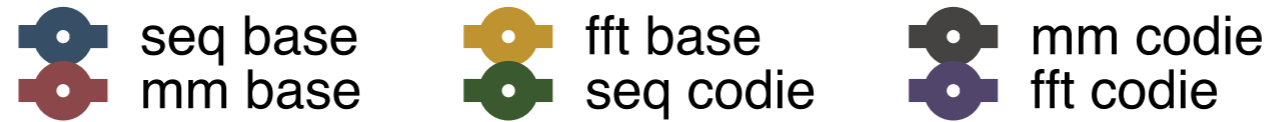
## M-Coreアーキテクチャ



# 実験と結果



## Cell/B.E. (@PS3®)





# 提案手法の制約

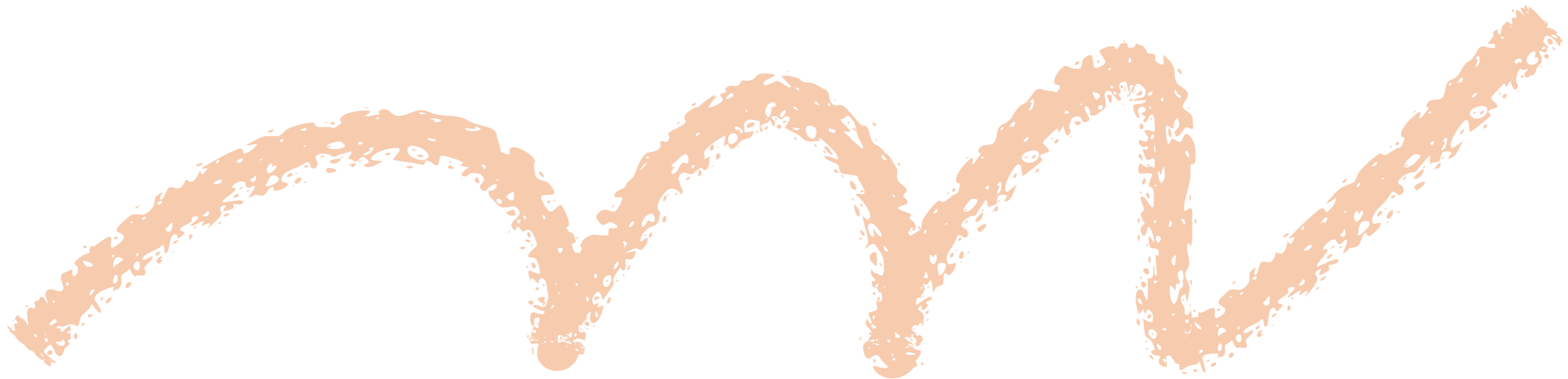
- ▶ キャッシュに係る命令と係らない命令が適度に混在している必要がある
- ▶ 依存関係にある命令を適度に分割する必要
  - ▶ Scala Expansion
  - ▶ Scala Privatization
- ▶ コンパイル可能なプログラムに対する制約
  - ▶ ポインタエイリアスの解決

# 関連研究

- ▶ Out-of-Orderプロセッサ
- ▶ プリフェッチ
- ▶ おかしなデータでも計算を継続
  - ▶ Runahead Execution
  - ▶ CFP, iCFP
- ▶ うまく譲り合う
  - ▶ CAER

# まとめ

- ▶ キャッシュミスをした命令を後で再実行することでアクセス衝突の影響を軽減させる手法CODIEを提案
- ▶ CODIEはコンパイル時に埋込まれる
- ▶ オーバヘッドが隠蔽できたことをベンチマークプログラムの実行によって評価



# 評価環境

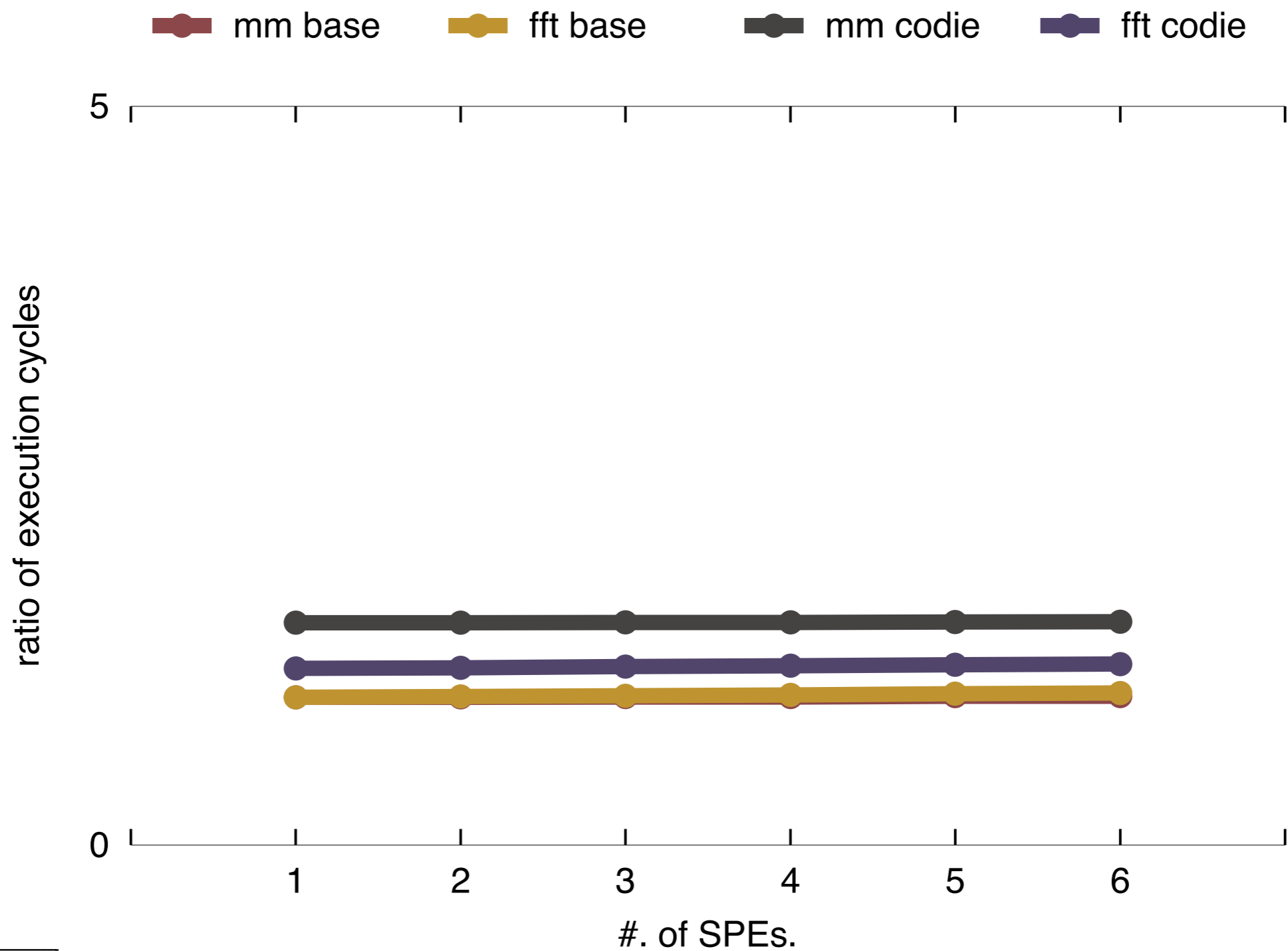
## Cell/B.E. (@PS3®)

CPU	Cell/B.E. 3.2GHz, 6SPEs
メモリ	256MiB, XDR <u>DRAM@3.2GHz</u>
システム転送速度	25.6GB/s
OS	Yellow Dog Linux 6.1
フロントエンドコンパイラ	LLVM2.7+clang-1.1+original
バックエンドコンパイラ	gcc 4.1.1

# 実験と結果



## Cell/B.E. (@PS3®)



# 今後の課題

- ▶ より現実的なアプリケーションでの評価
- ▶ オーバヘッドの軽減
- ▶ コンパイラによる最適化の検討
- ▶ コンパイル可能なプログラムに対する制約の緩和