メニーコア・プロセッサとそれを支える要素技術

井上弘士[1]　木村啓二[2]　松谷宏紀[3]

[1]九州大学大学院システム情報科学研究院 情報知能工学部門
[2]早稲田大学 理工学術院 情報理工学科
[3]東京大学大学院 情報理工学系研究科

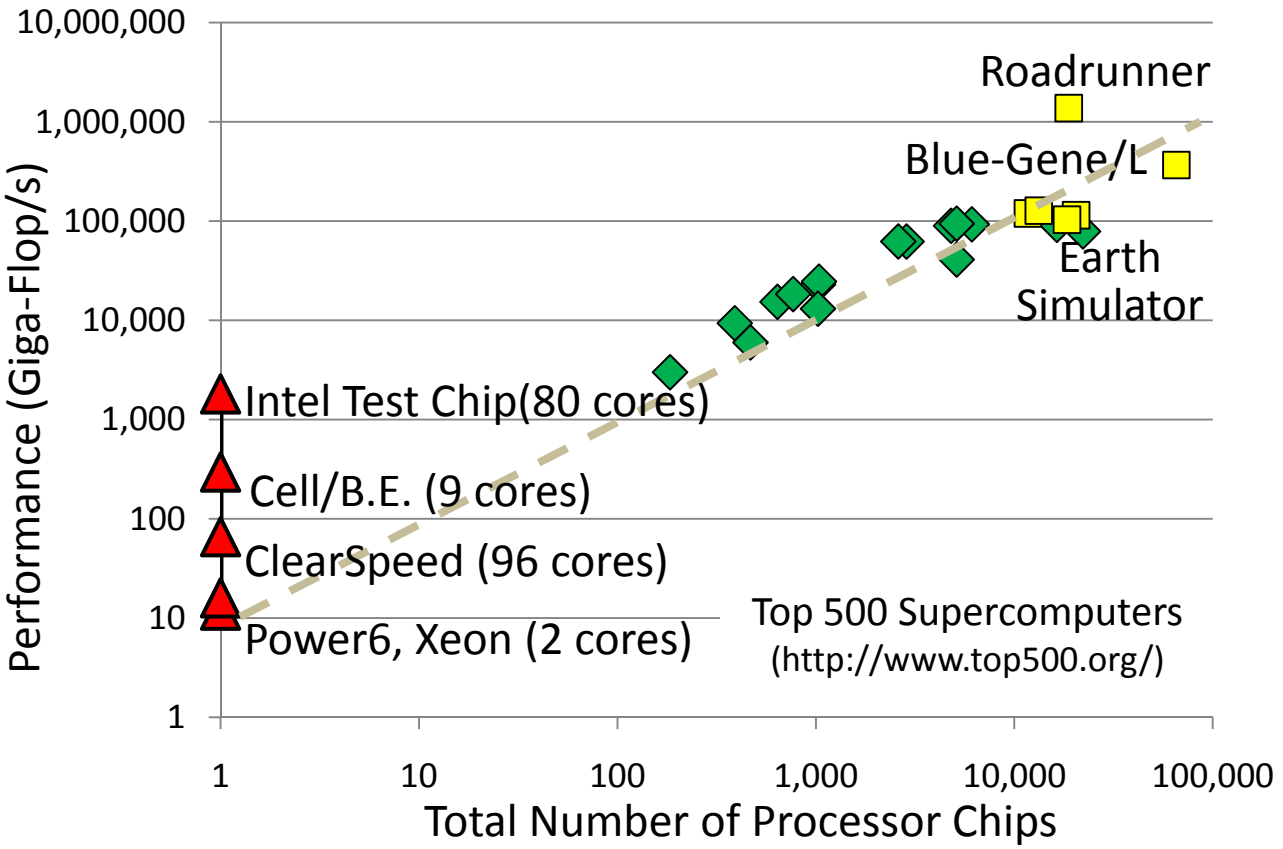
チュートリアル内容

- なぜメニーコアなのか?～「マルチ」から「メニー」の世界へ～
- メニーコアを支える要素技術
  - プロセッサ/メモリ・アーキテクチャ
  - ネットワーク・オン・チップ
  - 自動並列化コンパイラ
- 今後の展望

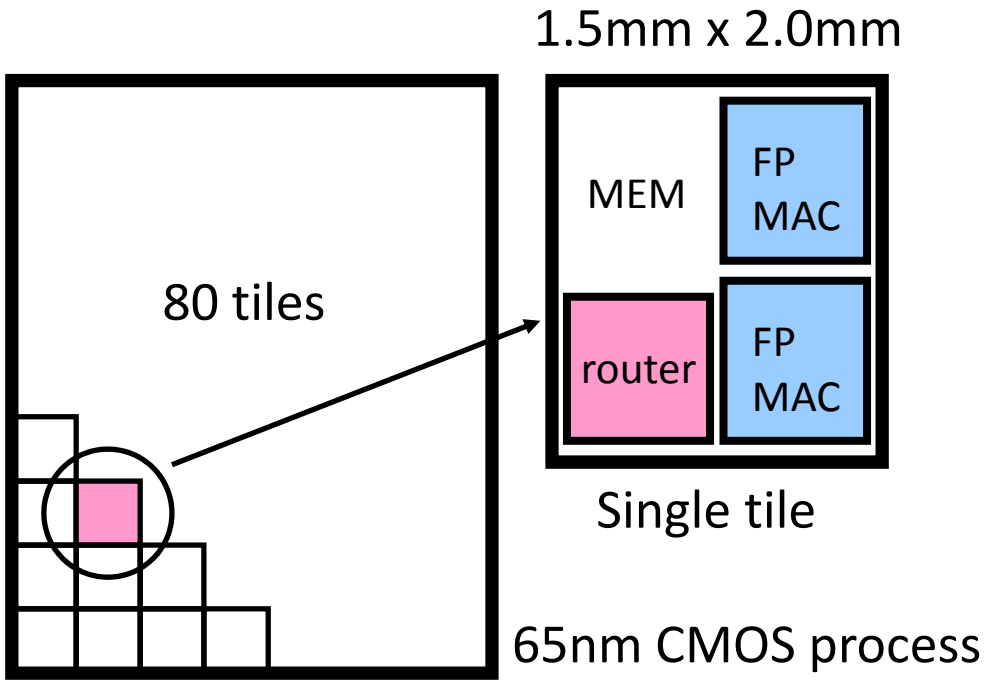
トランジスタを如何に有効活用し性能を向上するか?

- ポイント1:理論ピーク性能を高くする!
  - 如何に「より多くの演算」を「より高速に」処理するか?
    - 並列性の活用
    - 動作周波数の改善
    - などなど
- ポイント2:実効性能を理論ピーク性能に近づける!
  - 如何に「性能阻害要因を排除」するか?
    - メモリ階層の最適化(キャッシュの搭載など)
    - 予測技術に基づく投機実行サポート
    - などなど

「マルチ」から「メニー」の世界へ!



代表的なメニーコアの「実」チップ



- 80コア＋NOC
- 1.81TFlop/s@ 5.7GHz
- 1.01TFlop/s@ 3.16GHz

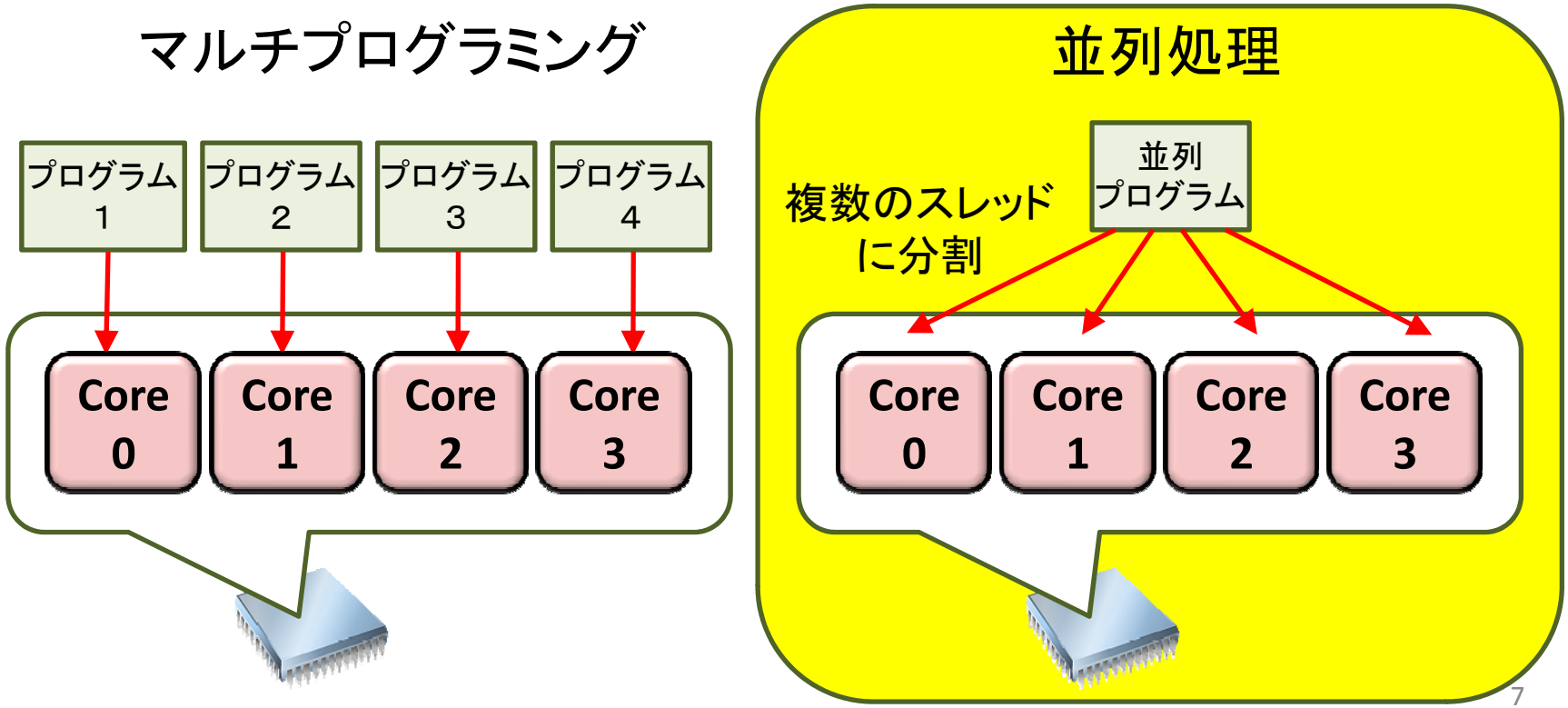
See
http://www.legitreviews.com/article/460/1/
http://techresearch.intel.com/articles/Tera-Scale/1449.htm

なぜメニーコアなのか?ポイントは?

- メニーコアの利点は?
  - コア数(PE数)に比例して理論ピーク性能が向上!
  - 設計が比較的容易!
- メニーコアの欠点は?
  - コア数(PE数)に比例してピーク消費電力が増加!
  - 実効性能を理論ピーク性能に近づけることがより困難に!
- ポイントは?
  - どのようなコア(とメモリ)を搭載するのか?
  - どのようにコアを接続するのか?
  - どのようにコアを活用するのか?

本チュートリアルで対象とする オンチップ並列処理モデル

- １つのチップ上に複数のプロセッサコアを搭載
- 同時に複数のコアで実行することにより性能向上



チュートリアル内容

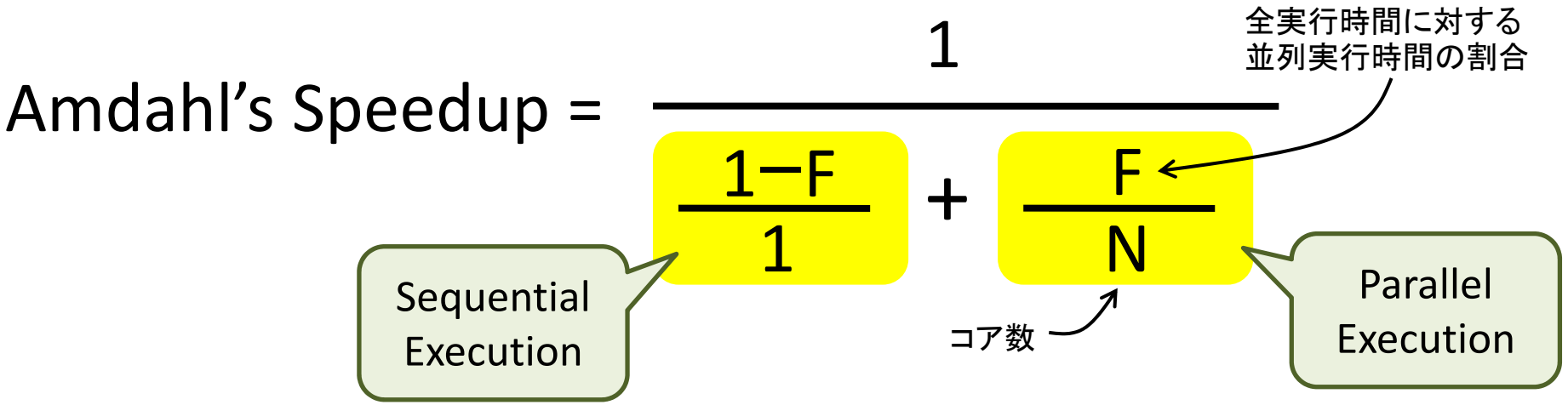
- なぜメニーコアなのか?～「マルチ」から「メニー」の世界へ～
- メニーコアを支える要素技術
  - プロセッサ/メモリ・アーキテクチャ
  - ネットワーク・オン・チップ
  - 自動並列化コンパイラ
- 今後の展望

詳細はこちらをご覧下さい

M. D. Hill and M. R. Marty, “Amdahl’s Law in the Multicore Era,” IEEE Computer, pp.33-38, July 2008.
M. D. Hill, HPCA’08 Keynote, http://pages.cs.wisc.edu/~markhill/includes/publications.html#year2008

どのようなコアを搭載するか?
～性能の観点から～

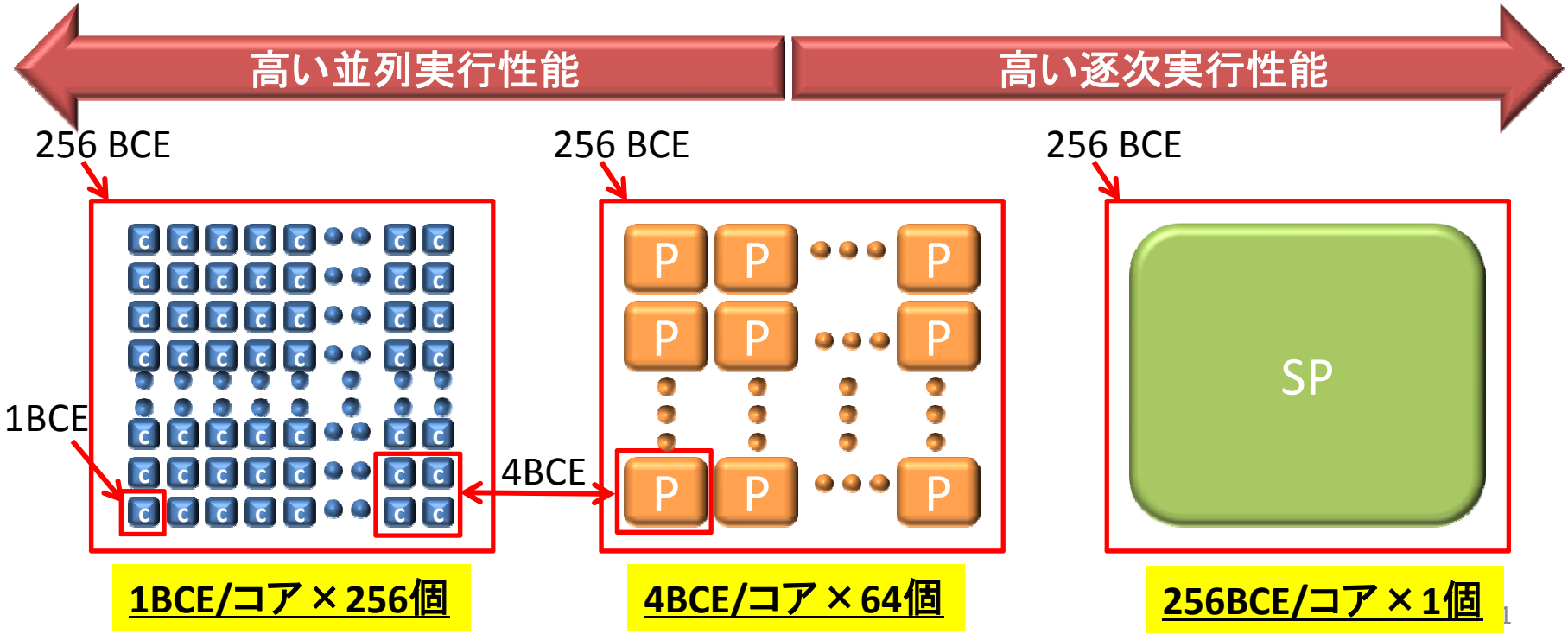
アムダールの法則(を思い出そう)



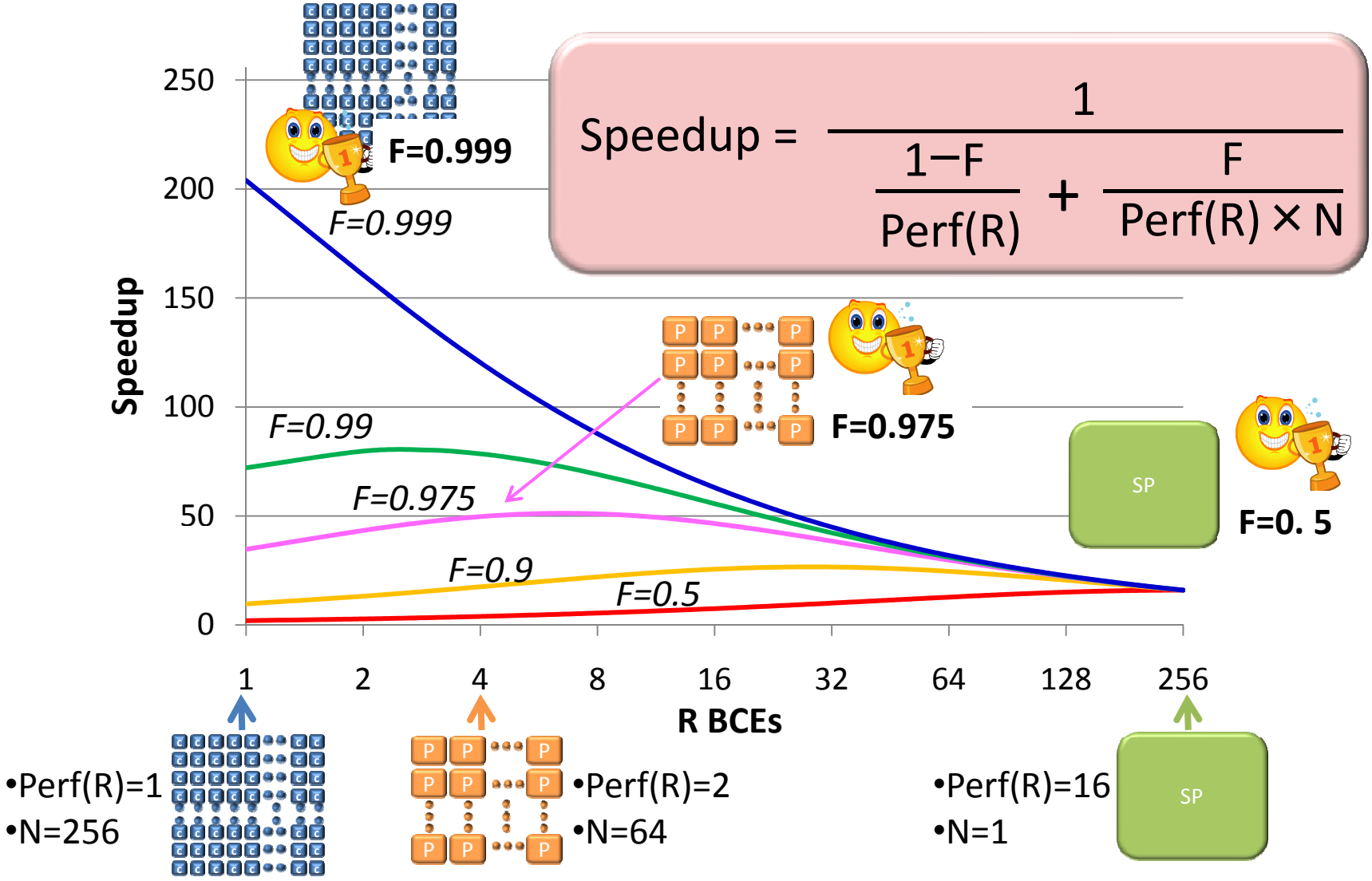
- 高性能化を実現するには?
  1. F(並列実行可能部分)を大きくする!
  2. 逐次実行時間を短縮する!
  3. 並列実行時間をF/Nに近づける!

Symmetricメニーコアの性能ポテンシャルは?

- 1BCE（Base Core Equivalent）は最小コア「c」の実装に要するHW量
- R-BCEでの逐次実行性能＝Perf（R）＝square root（R）
- メモリによる影響等は無視（理想状態を想定）



Symmetricメニーコアの性能ポテンシャルは?



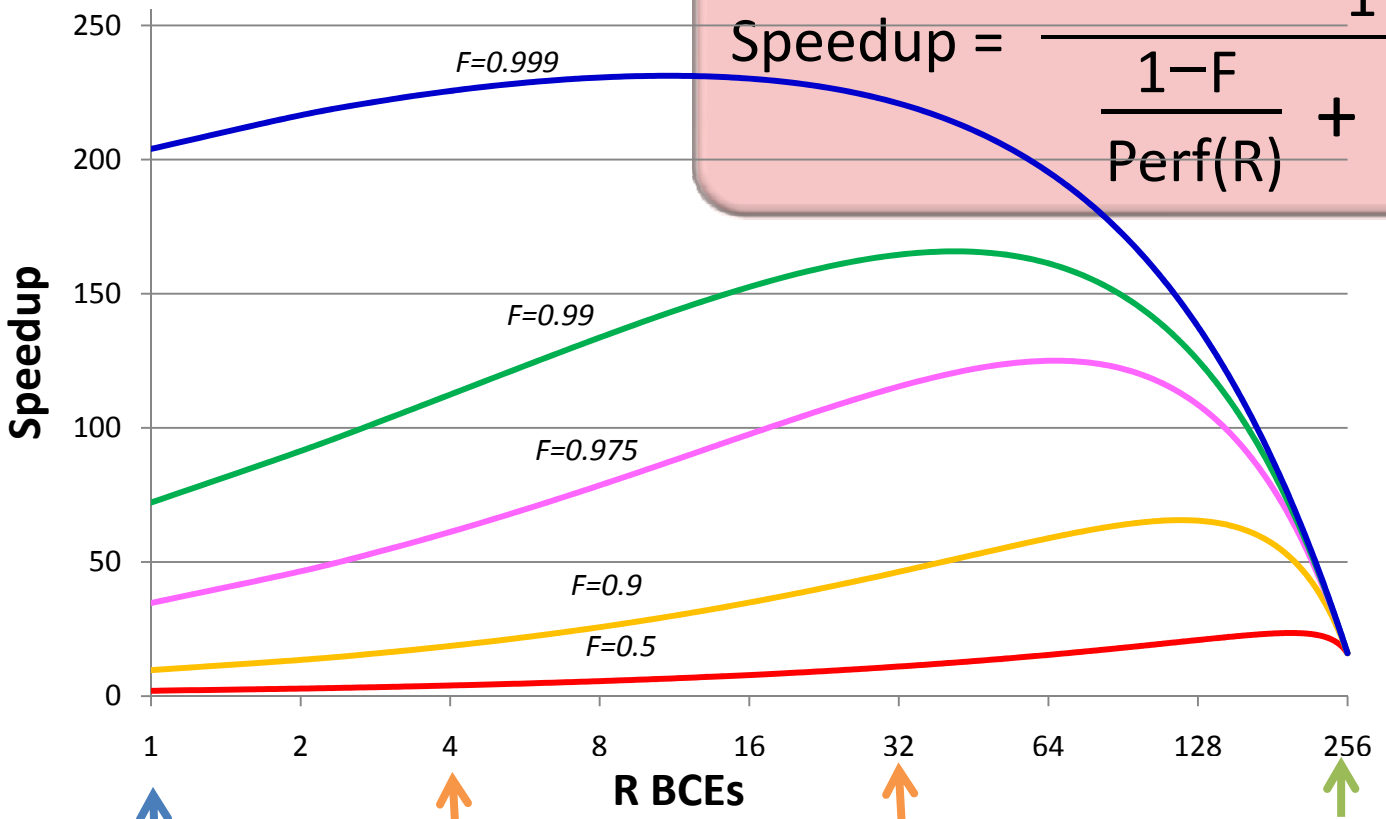
Mark D. Hill, HPCA'08 Keynote, http://pages.cs.wisc.edu/~markhill/includes/publications.html#year2008

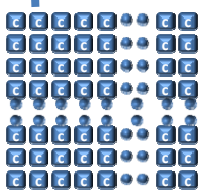
Asymmetricメニーコアの性能ポテンシャルは?

- 1BCE（Base Core Equivalent）は最小コア「c」の実装に要するHW量
- R-BCEでの逐次実行性能＝Perf（R）＝square root（R）
- メモリによる影響等は無視（理想状態を想定）

Asymmetricメニーコアの性能ポテンシャルは?

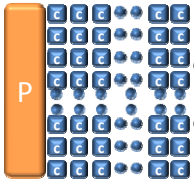
$$\text{Speedup} = \frac{1}{\frac{1-F}{\text{Perf}(R)} + \frac{F}{\text{Perf}(R)+(N-R)}}$$





•Perf(R)=1
•N=256

•Perf(R)=2
•N-R=252



•Perf(R)=5.7
•N-R=224

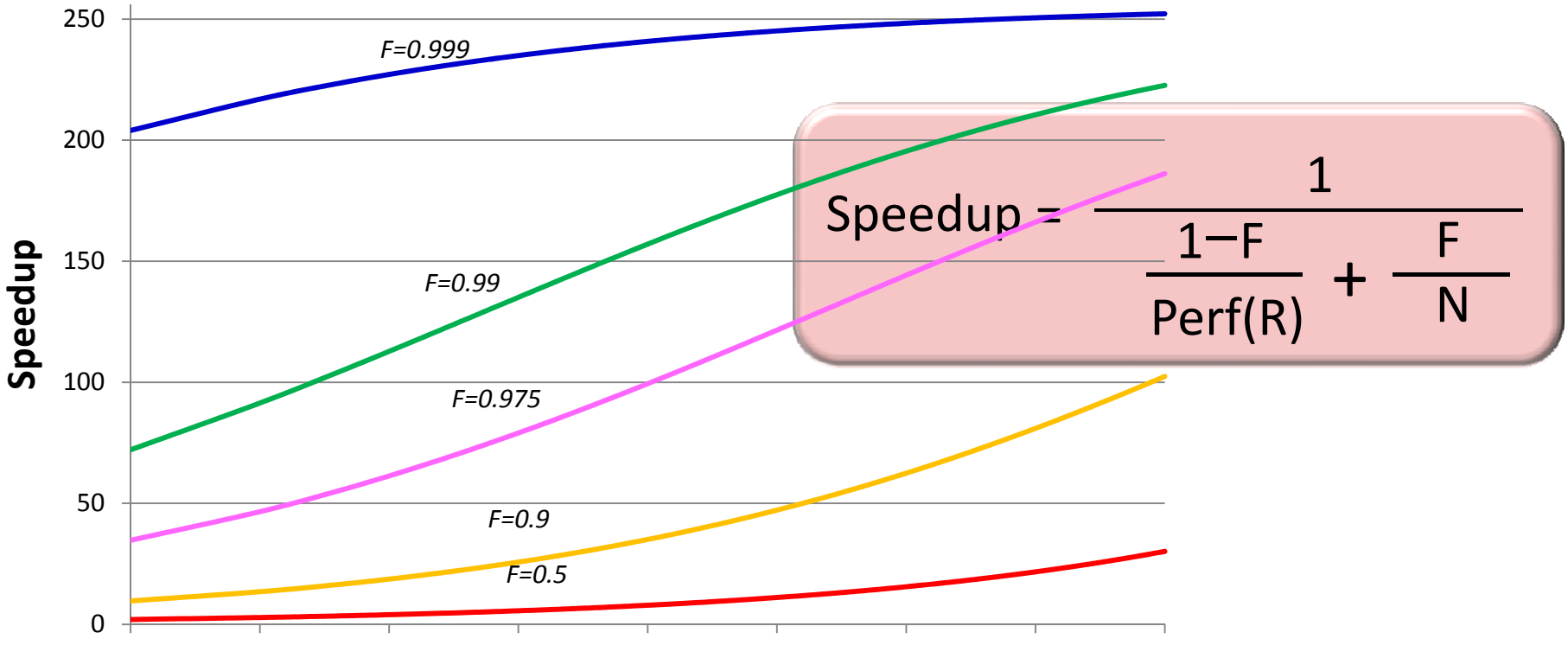
•Perf(R)=16
•N-R=0

Mark D. Hill, HPCA'08 Keynote, http://pages.cs.wisc.edu/~markhill/includes/publications.html#year2008

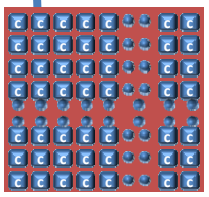
Dynamicメニーコアの性能ポテンシャルは?

- 動的なコア活用法の切替え(ができたとしたら・・・)
  - 並列部:全てのHW資源を最小コア「c」で活用
  - 逐次部:「P」もしくは「SP」で実行

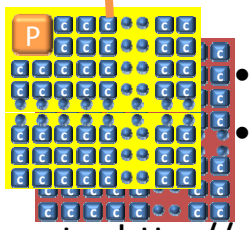
Dynamicメニーコアの性能ポテンシャルは?



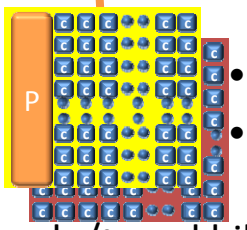
$$\text{Speedup} = \frac{1}{\frac{1-F}{\text{Perf}(R)} + \frac{F}{N}}$$



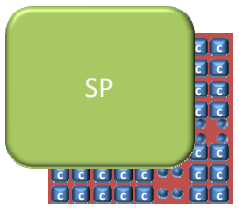
•Perf(R)=1
•N=256



•Perf(R)=2
•N=256



•Perf(R)=5.7
•N=256



•Perf(R)=16
•N=256

Mark D. Hill, HPCA'08 Keynote, http://pages.cs.wisc.edu/~markhill/includes/publications.html#year2008

メニーコア性能のポイント

- 単純なメニーコア構成であれば，並列実行可能部分を「極めて高く（F>0.95）」する必要がある!
  - 如何にしてこの並列性をアプリケーションから抽出するか?
  - 如何にして並列化効率を阻害する要因を排除するか?
- 「並列処理の最適化」＋「逐次処理の最適化」を同時に考慮する必要あり!

詳細はこちらをご覧下さい

D. H. Woo and H. S. Lee, “Extending Amdahl’s Law for Energy-Efficient Computing in the Many-Core Era,” IEEE Computer, pp.24-31, Dec. 2008.

どのようなコアを搭載するか?
～性能/電力/エネルギーの観点から～

メニーコア・プロセッサの電力/エネルギー効率を探る!!

- メニーコアは低消費電力/消費エネルギーの観点から得策なのか?
- ポイント
  - 逐次実行時に多くのプロセッサがアイドル状態
  - どのようなコアを如何に活用すべきか?

性能/消費電力/消費エネルギー ～「p*」タイプ・メニーコアの場合～

P

1コア実行での実行時間=1
1コア実行での総消費E=1

1コア実行での総消費エネルギー

逐次実行時のアイドルコア数

コア稼働時に対するアイドル時電力比（$0 \le k \le 1$）

$$Perf = \frac{1}{(1-f)+\frac{f}{n}}$$

並列実行時のコア数

$$W = \frac{1+(n-1)k(1-f)}{(1-f)+\frac{f}{n}}$$

$$\frac{Perf}{W} = \frac{1}{1+(n-1)k(1-f)}$$

$$\frac{Perf}{J} = \frac{1}{(1-f)+\frac{f}{n}} \times \frac{1}{1+(n-1)k(1-f)}$$

性能/消費電力/消費エネルギー
～「c*」タイプ・メニーコアの場合～

P

1コア実行での実行時間=1
1コア実行での総消費E=1

P に対する c の・・・・

相対性能（$0 \le S_C \le 1$）

消費電力比
（$0 \le w_C \le 1$）

逐次実行時の
アイドルコア数

c の稼働時に対する
アイドル時電力比（$0 \le k_C \le 1$）

$$Perf = \frac{S_C}{(1-f)+\frac{f}{n}}$$

並列実行時のコア数

$$W = \frac{w_C + (n-1)w_C k_C (1-f)}{(1-f)+\frac{f}{n}}$$

$$\frac{Perf}{W} = \frac{S_C}{w_C + (n-1)w_C k_C (1-f)}$$

$$\frac{Perf}{J} = \frac{S_C}{(1-f)+\frac{f}{n}} \times \frac{S_C}{w_C + (n-1)w_C k_C (1-f)}$$

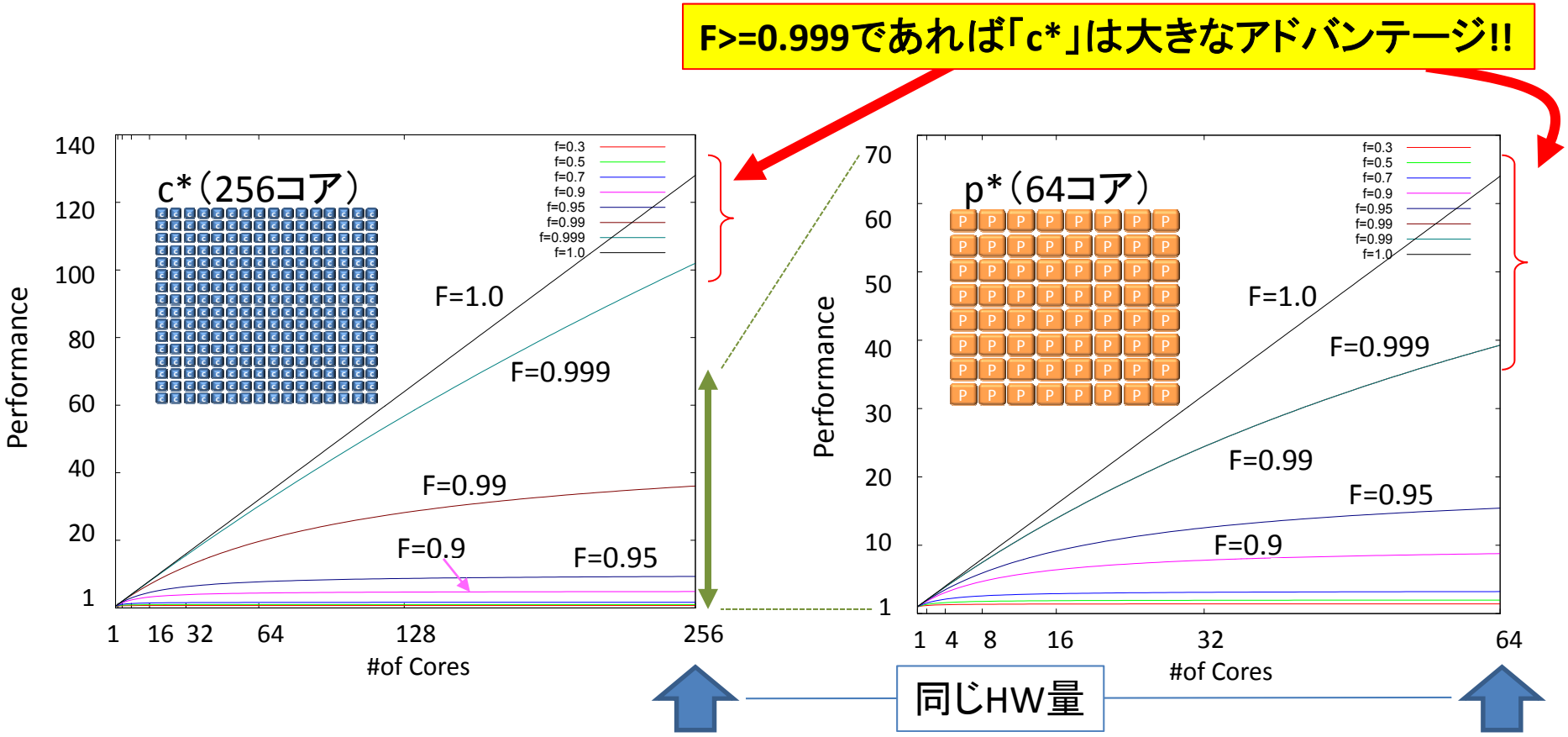
「p*メニーコア」vs「c*メニーコア」 ～ 仮定～

	P 高性能/高消費電力コア	c 低性能/低消費電力コア
トランジスタ数（面積）	1	0.25
性能	1	S_C：0.5（ポラックの法則）
稼働時消費電力	1	W_C：0.25（Power ∝ #Tran.）
アイドル時消費電力	0.3（稼働時消費電力比）	K_C：0.25（稼働時消費電力比）

1/4（トランジスタ数）
1/2（性能）
1/4（稼働時消費電力）
30%（稼働時消費電力→アイドル時消費電力）
25%（W_C→K_C）

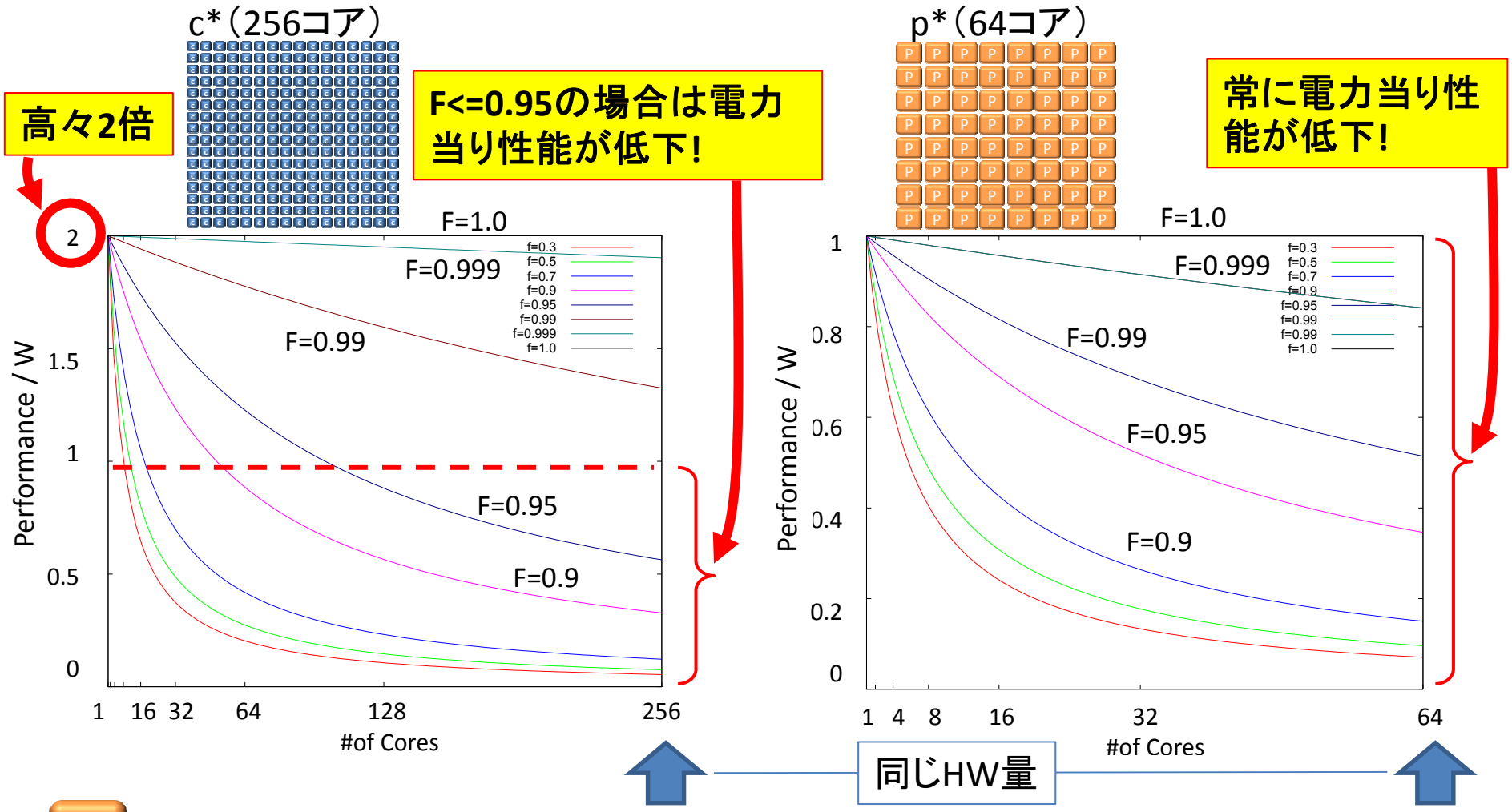
- プロセッサコア特性は上表の通り
- メモリアクセスやオンチップ/オフチップ通信などの影響は無視（理想状態）

「p*メニーコア」vs「c*メニーコア」
～ 性能を比較する!～

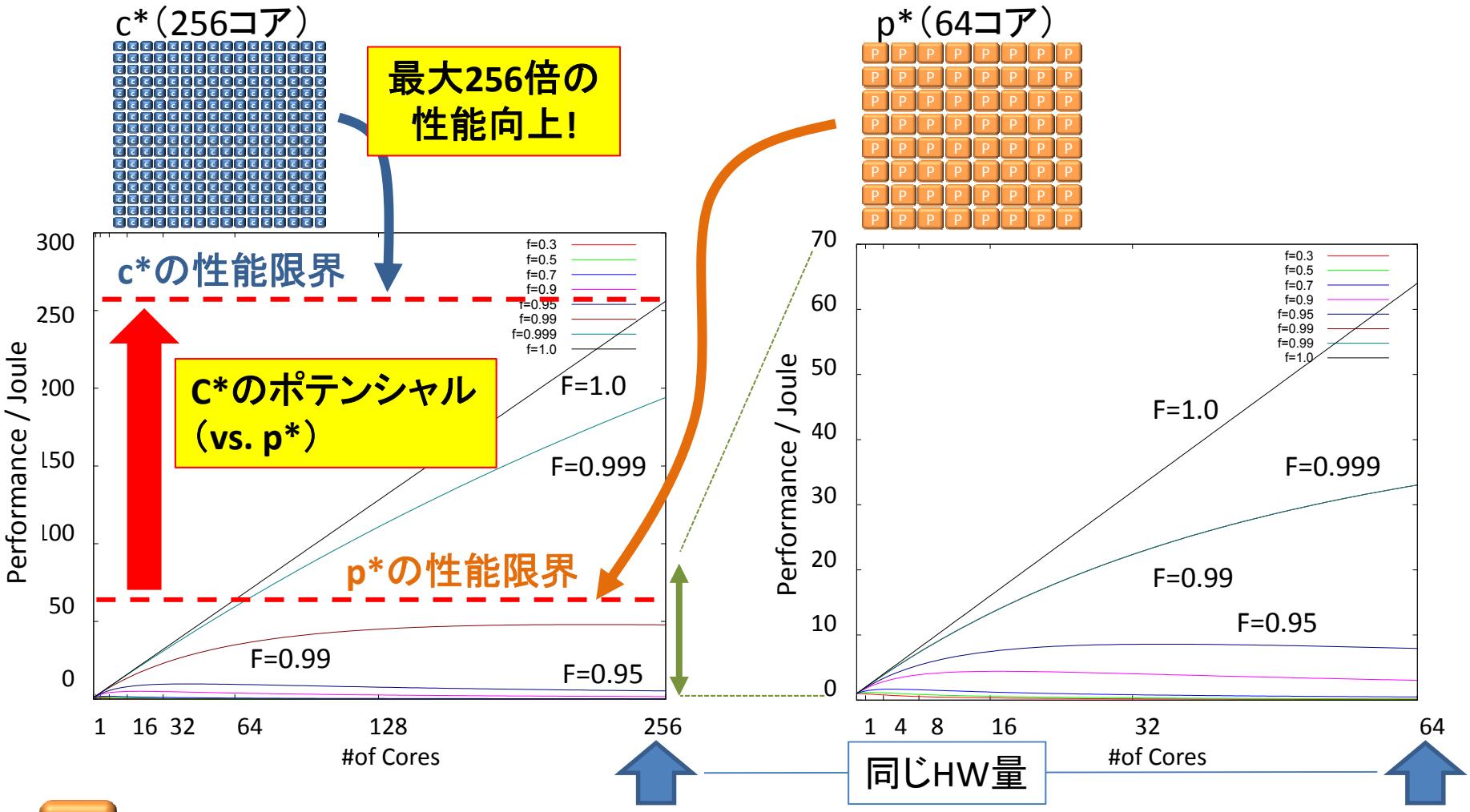


P 1コアに対する相対性能

「p*メニーコア」vs「c*メニーコア」
～ 電力当りの性能を比較する!～



「p*メニーコア」vs「c*メニーコア」
～ エネルギー当りの性能を比較する!～



P 1コアに対する相対性能

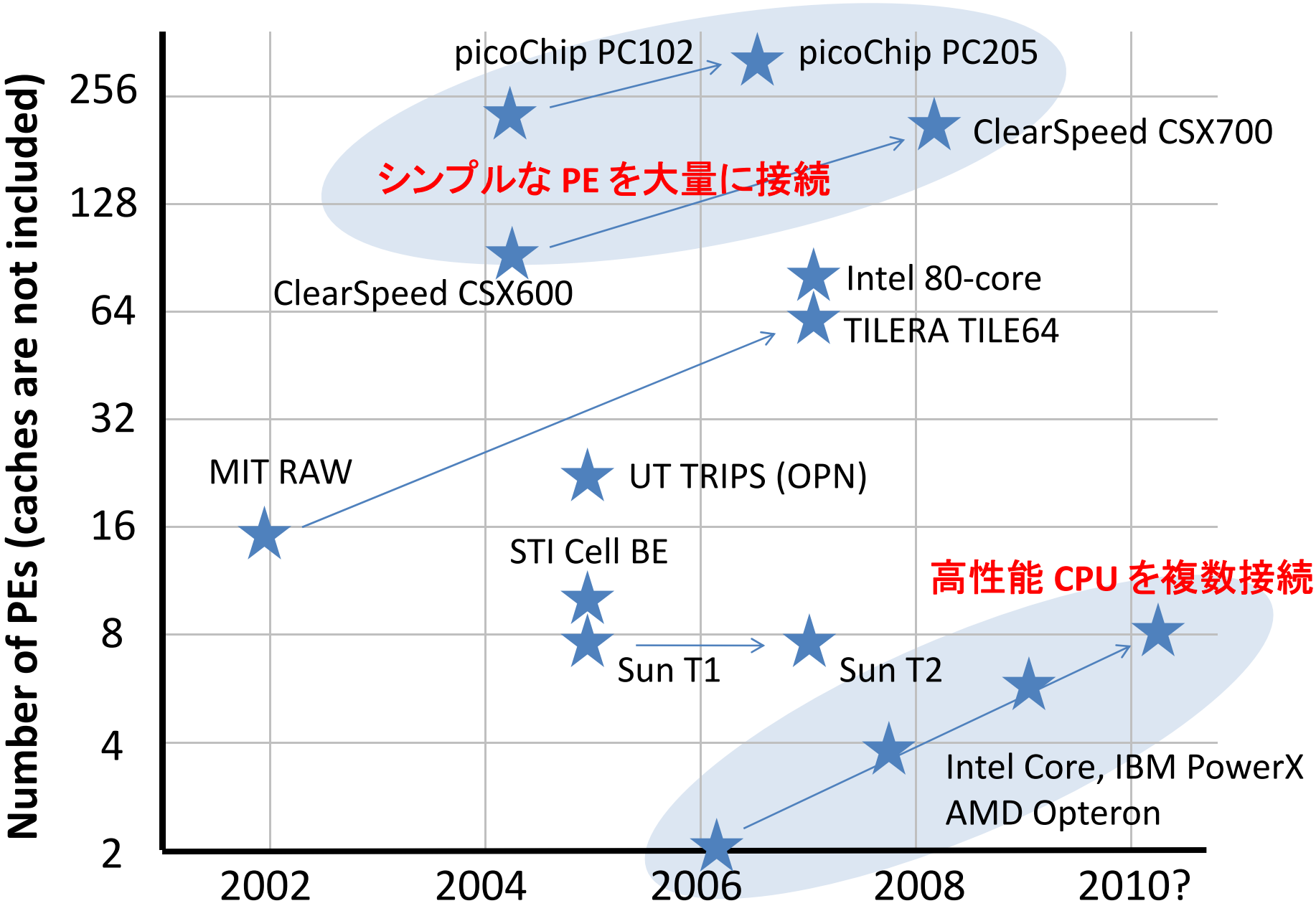
「p*メニーコア」vs「c*メニーコア」 ～ 結局のところ・・・～

- 「c*」の本質は?
  - 超並列実行の実現
  - 「性能/電力の向上（つまり，消費エネルギー削減）」ではなく，「性能/エネルギー（つまり，エネルギー遅延積の削減）」に大きなアドバンテージ!
- 「c*」のアプリケーション・ターゲットは?
  - 高い性能と省エネルギーが求められる場合
- 「p*」が優位な場合は?
  - 十分な並列性が抽出できない場合

チュートリアル内容

- なぜメニーコアなのか?～「マルチ」から「メニー」の世界へ～
- メニーコアを支える要素技術
  - プロセッサ/メモリ・アーキテクチャ
  - ネットワーク・オン・チップ
  - 自動並列化コンパイラ
- 今後の展望

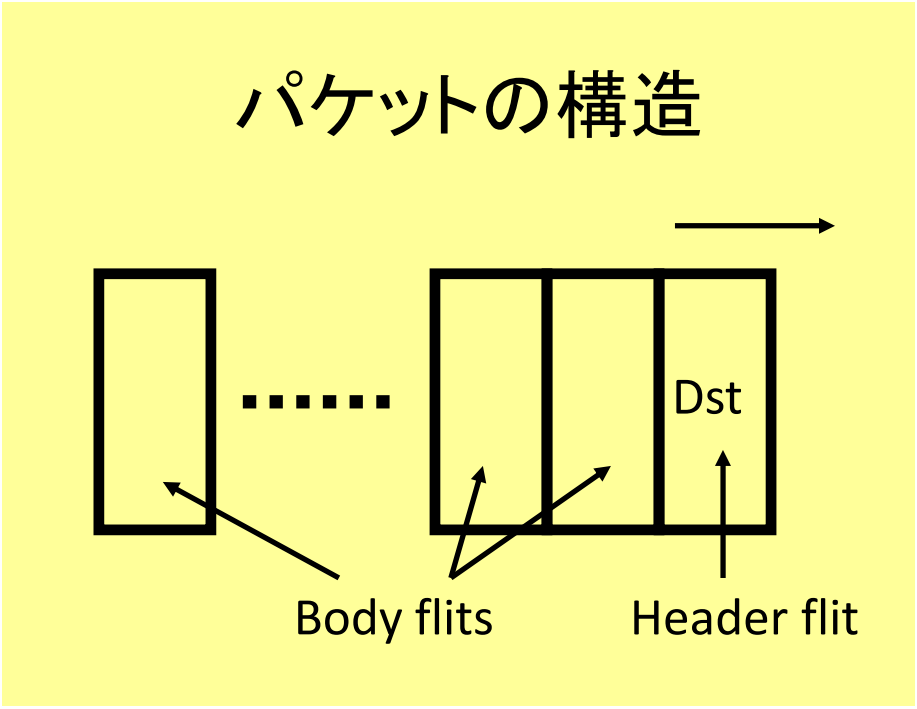
オンチップ・ネットワークの話し

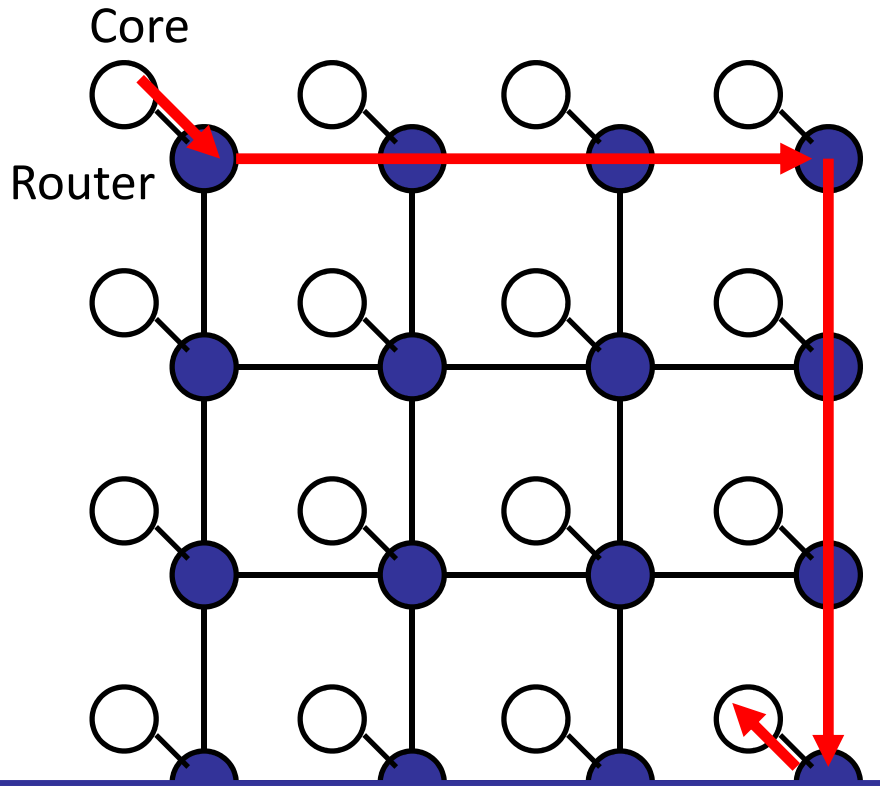


NOCの内部構造を探る!
～要素技術の解説～

コアの接続方式: バス vs. ネットワーク

- オンチップバス
  - 全コアがリンクを共有
  - 1度に1対のみ通信可能
- Network-on-Chip (NoC)
  - ネットワーク状に接続
  - パケットスイッチング

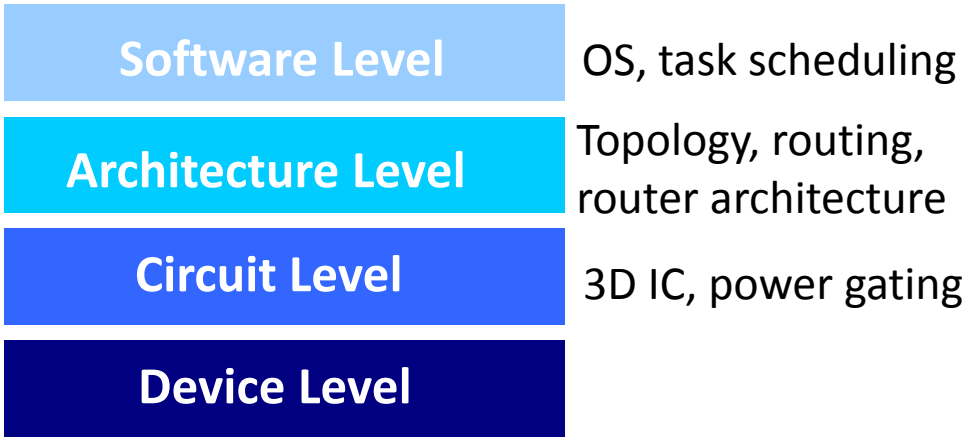




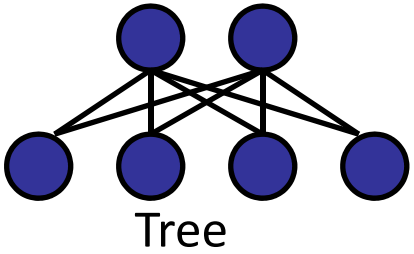
オンチップバスに代わる結合網として Network-on-Chip (NoC) が注目

オンチップネットワークの要素技術

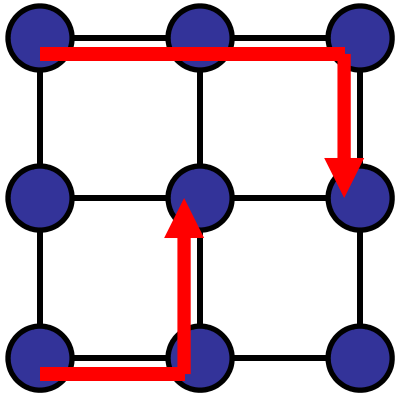
- いろいろな研究分野
  - ソフトウェア レベル
  - アーキテクチャ レベル
  - 回路 レベル



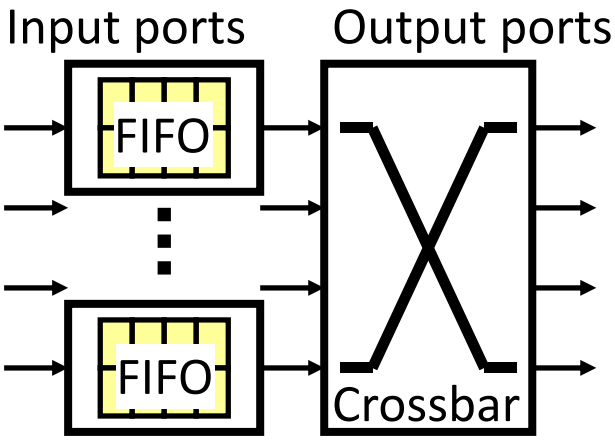
- ネットワークアーキテクチャ



ネットワークトポロジ



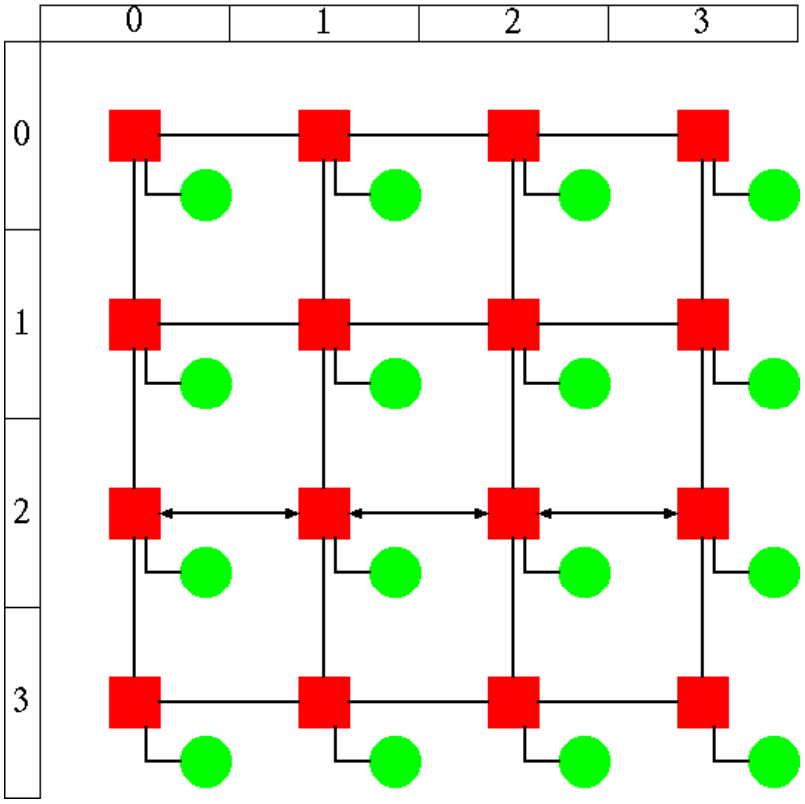
ルーティング,フロー制御



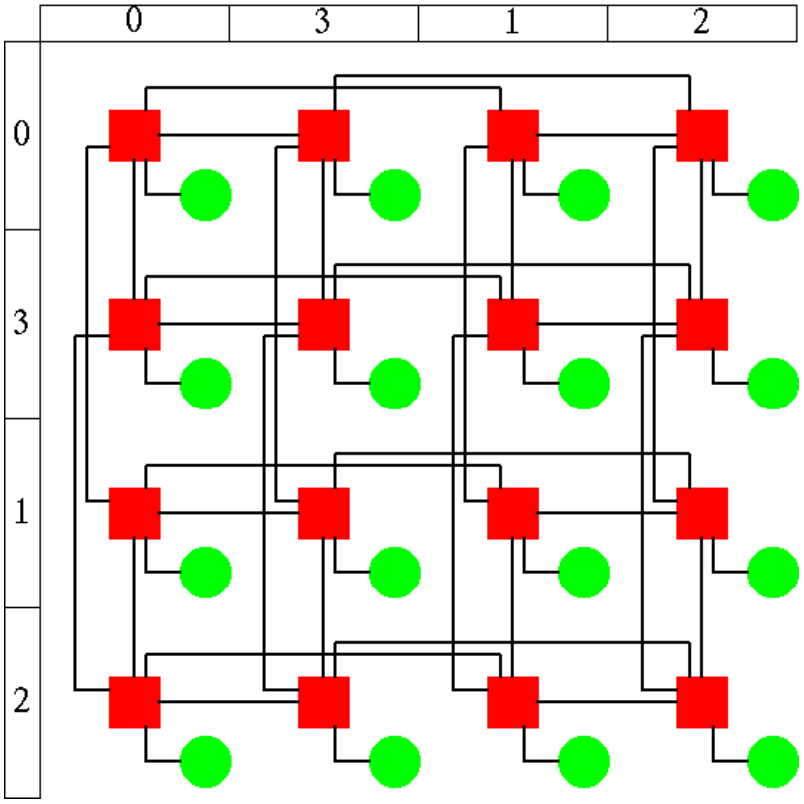
ルータアーキテクチャ

要素技術: ネットワークトポロジ

- 2-D Mesh
  - 各ノードは東西南北の隣接ノードと接続



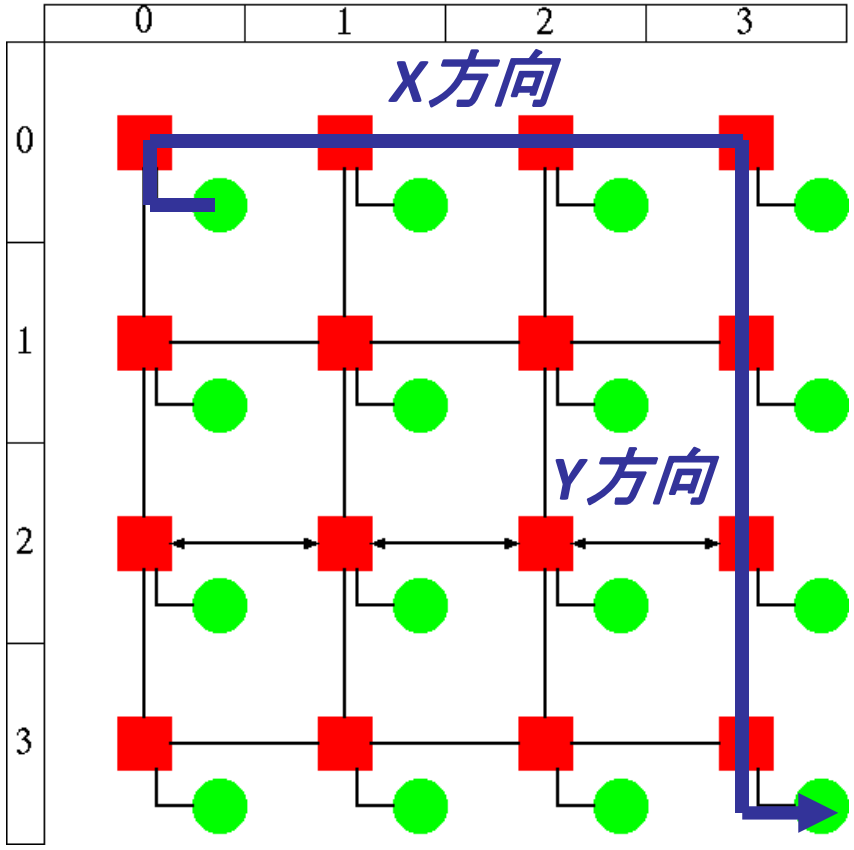
- 2-D Torus
  - 端と端をつなぐリンクを追加
  - メッシュの2倍の帯域



ルータ　計算コア

要素技術: パケットルーティング

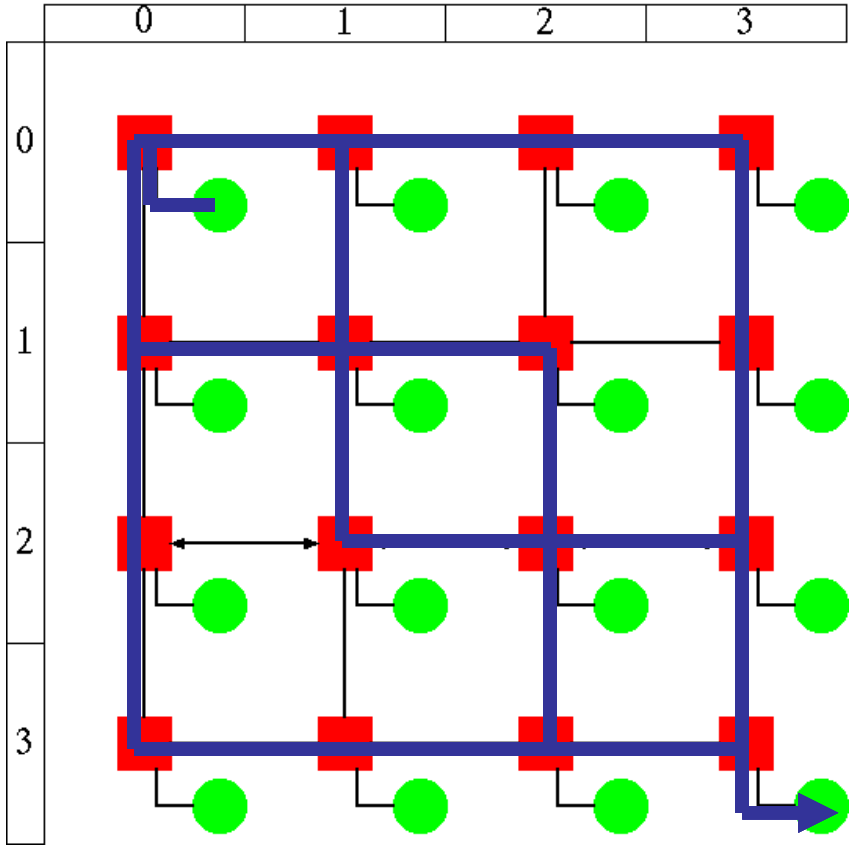
- 固定型ルーティング
  - Source-destination 間の経路は1つに固定
- ランダム型ルーティング
  - Source-destination 間に複数の経路
  - ランダムに1つを選択
- 適応型ルーティング
  - Source-destination 間に複数の経路
  - 混雑に応じて1つを選択



例) 次元順ルーティング

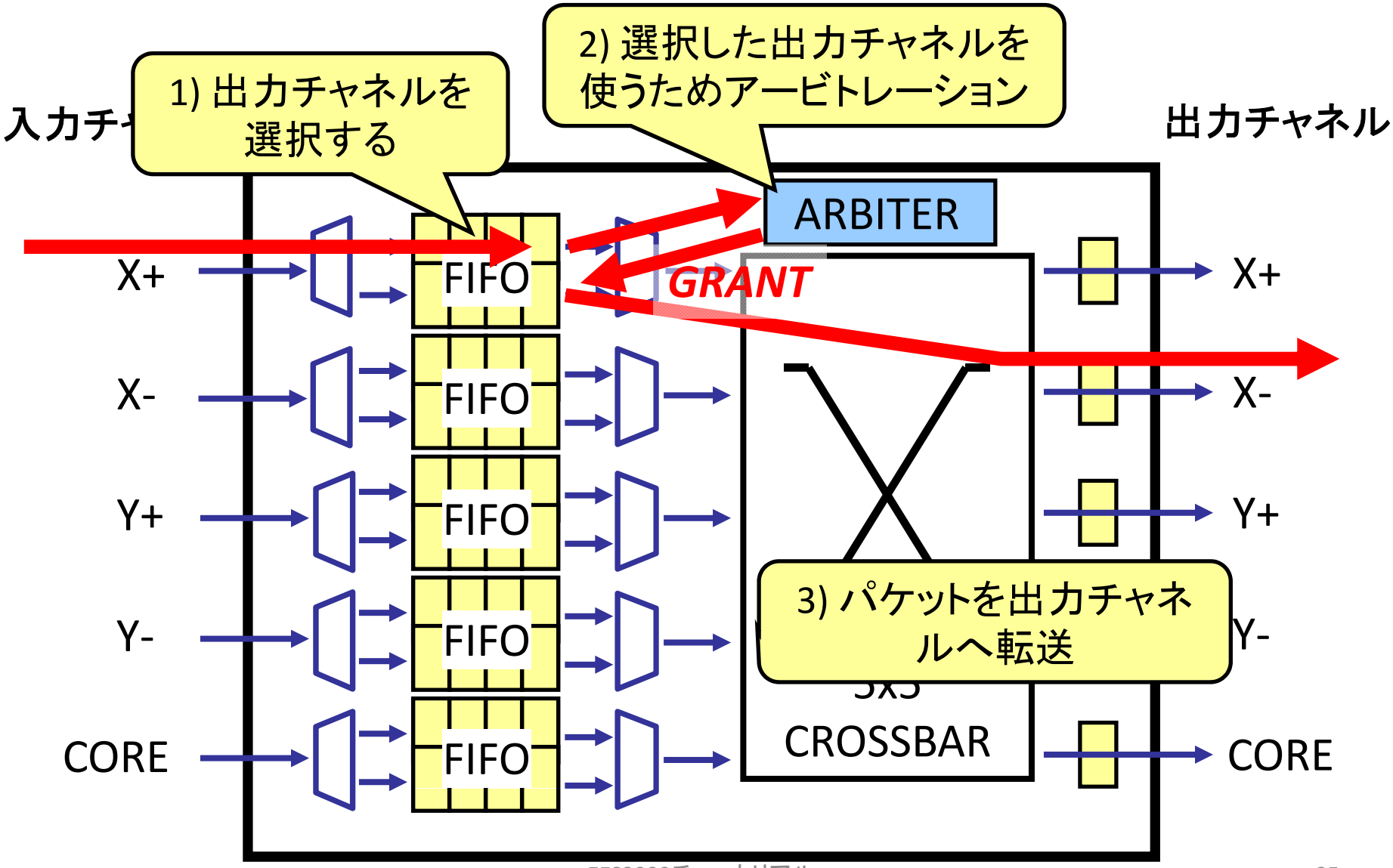
要素技術: パケットルーティング

- 固定型ルーティング
  - Source-destination 間の経路は1つに固定
- ランダム型ルーティング
  - Source-destination 間に複数の経路
  - ランダムに1つを選択
- 適応型ルーティング
  - Source-destination 間に複数の経路
  - 混雑に応じて1つを選択



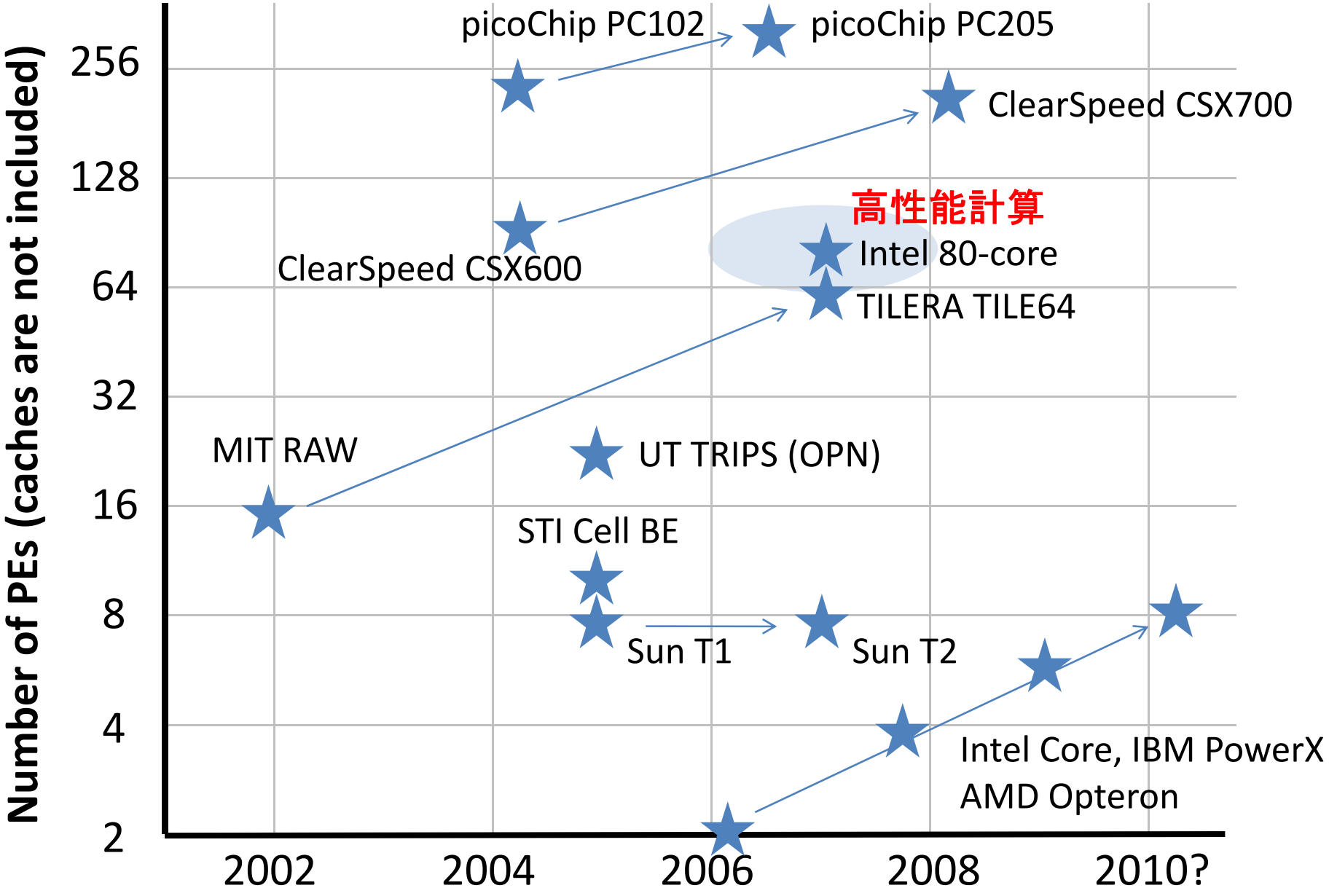
例) Turn-model (West-first, Odd-even), Opt-y, Duato's protocol

要素技術: ルータアーキテクチャ

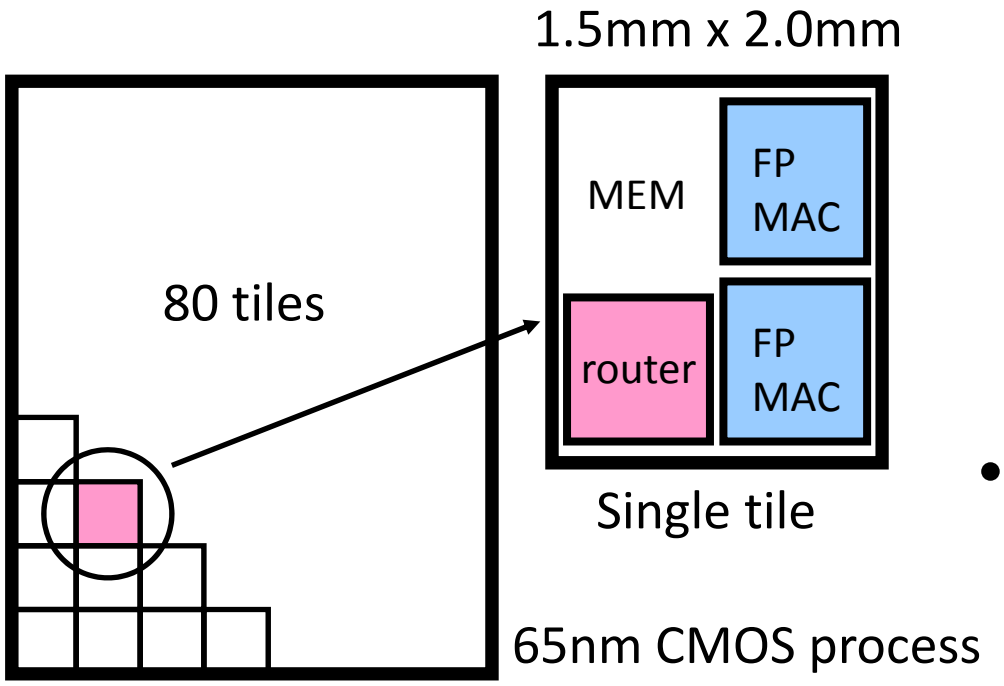


どのような「実」システムが存在するのか?

実システムの例: Intel 80-core chip



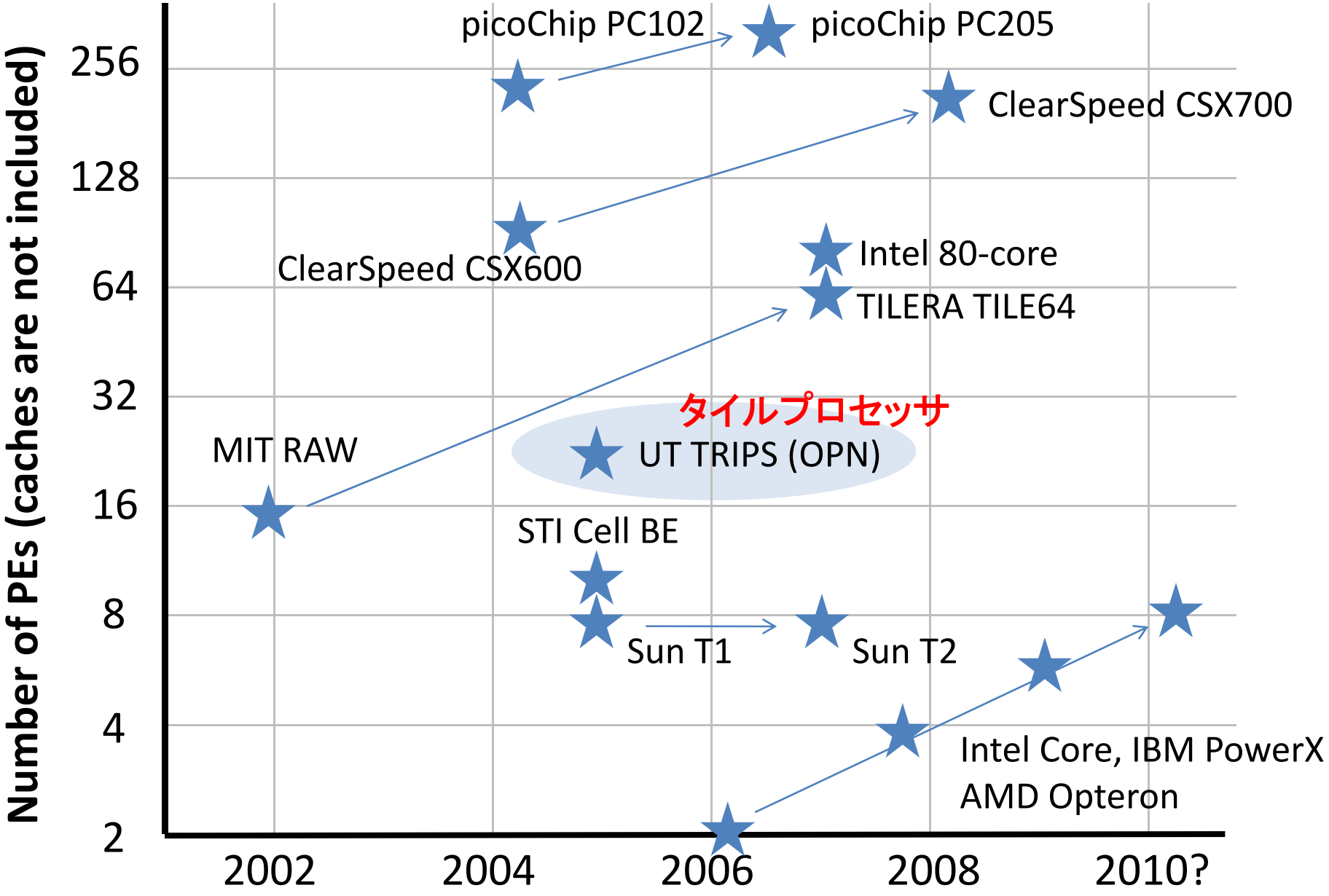
実システムの例: Intel 80-core chip



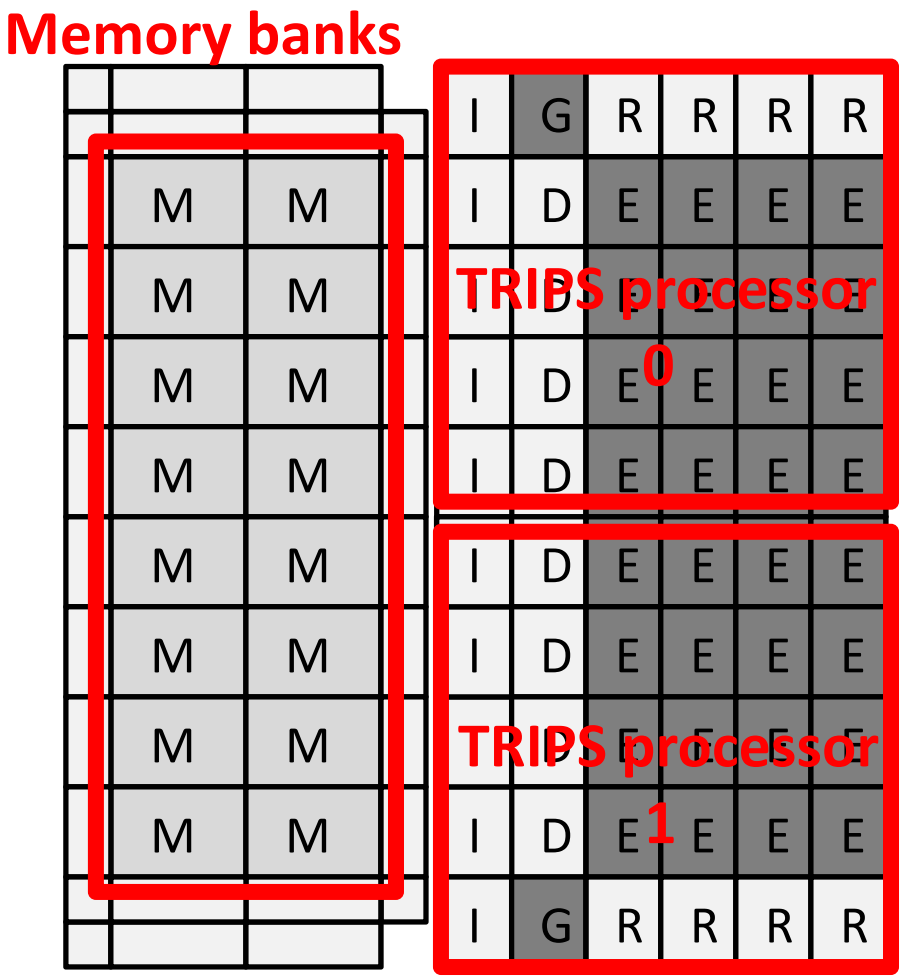
Intel 80-core chip [Vangal,ISSCC'07]

- タイルの構成
  - 浮動小数点演算ユニット (FPMAC) 2個
  - メモリ
  - オンチップルータ
- ネットワークの特徴
  - 8x10 mesh
  - リンクデータ幅 32-bit
  - 次元順ルーティング
  - Wormhole スイッチング

実システムの例: UT Austin TRIPS chip



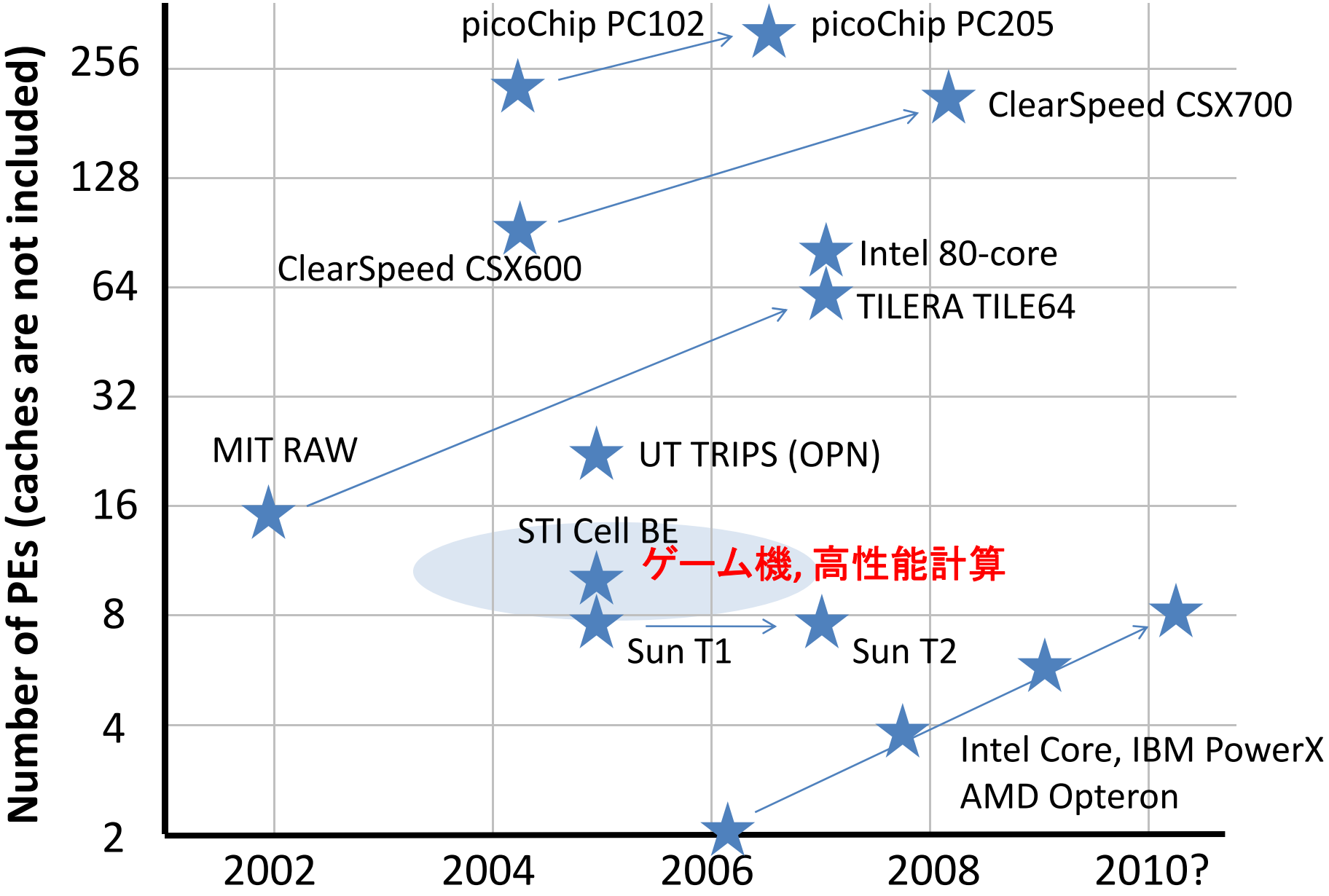
実システムの例: UT Austin TRIPS chip



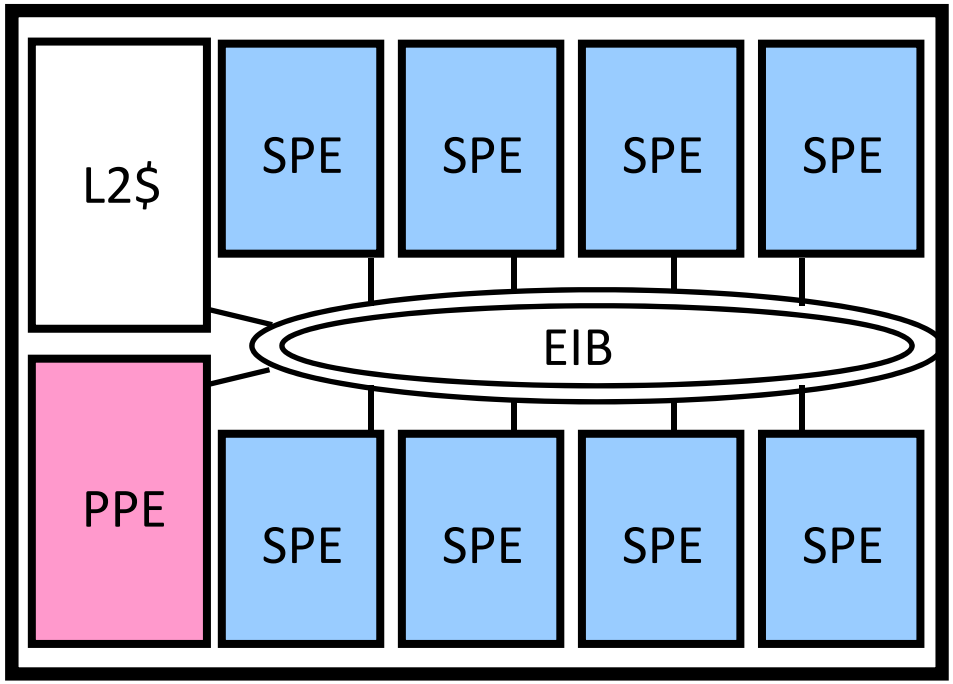
TRIPS chip diagram [Gratz, IEEE MICRO'07]

- On-chip network (OCN)
  - TRIPS プロセッサ 2個
  - メモリバンク 16個
- Operand network (OPN)
  - TRIPS プロセッサ内
  - タイル 25個
- OCN と OPN の特徴
  - 2次元メッシュ
  - YX ルーティング
  - Wormhole スイッチング

実システムの例: Sony,Toshiba,IBM Cell BE



実システムの例: Sony,Toshiba,IBM Cell BE

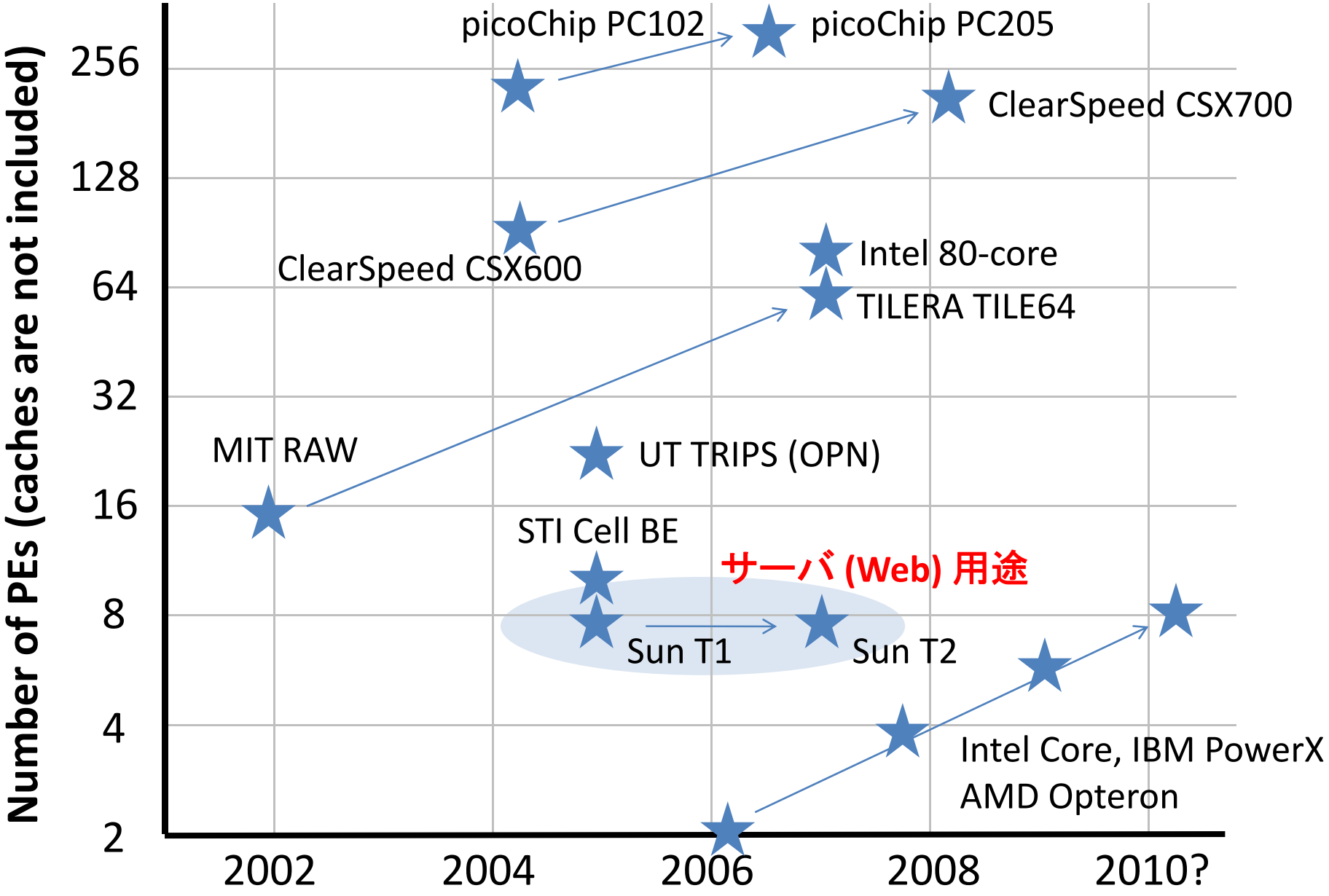


Cell Broadband Engine

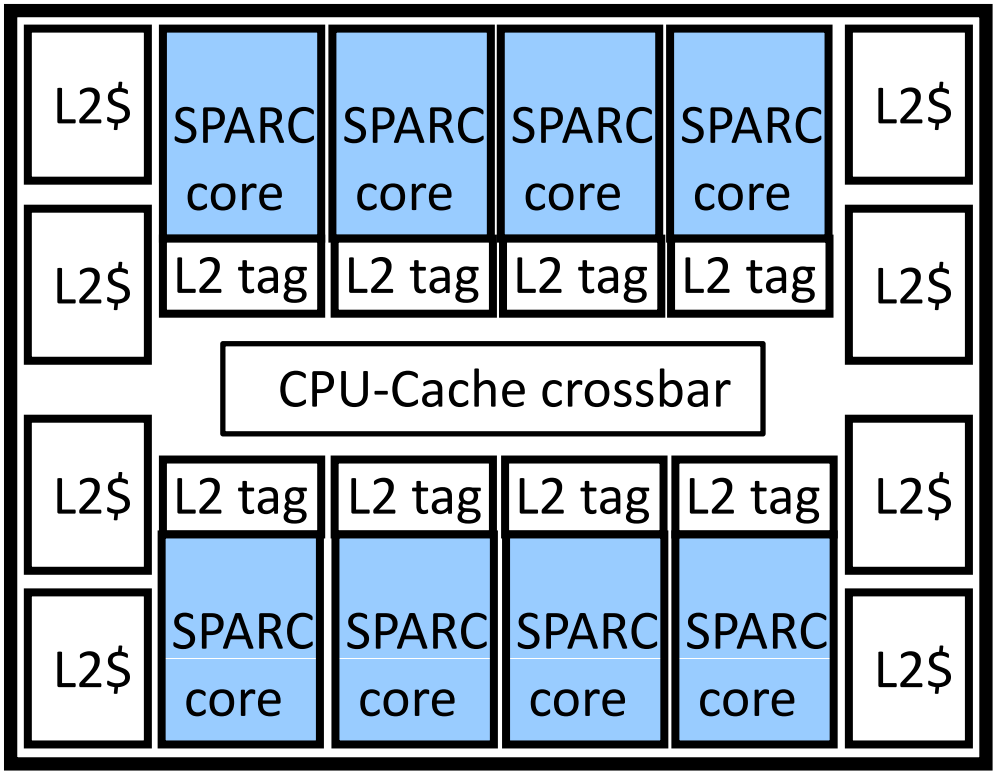
- Cell BE の構成
  - PPE (PowerPC) 1個
  - SPE (SIMDプロセッサ) 8個
  - メモリ, I/O
  - リングバス EIB で接続
- Element Interconnect Bus
  - 単方向リング 4本
  - 中央にアービタ
  - リンクデータ幅 128-bit

http://domino.research.ibm.com/comm/research_projects.nsf/pages/multicore.CellBE.html

実システムの例: OpenSPARC T2



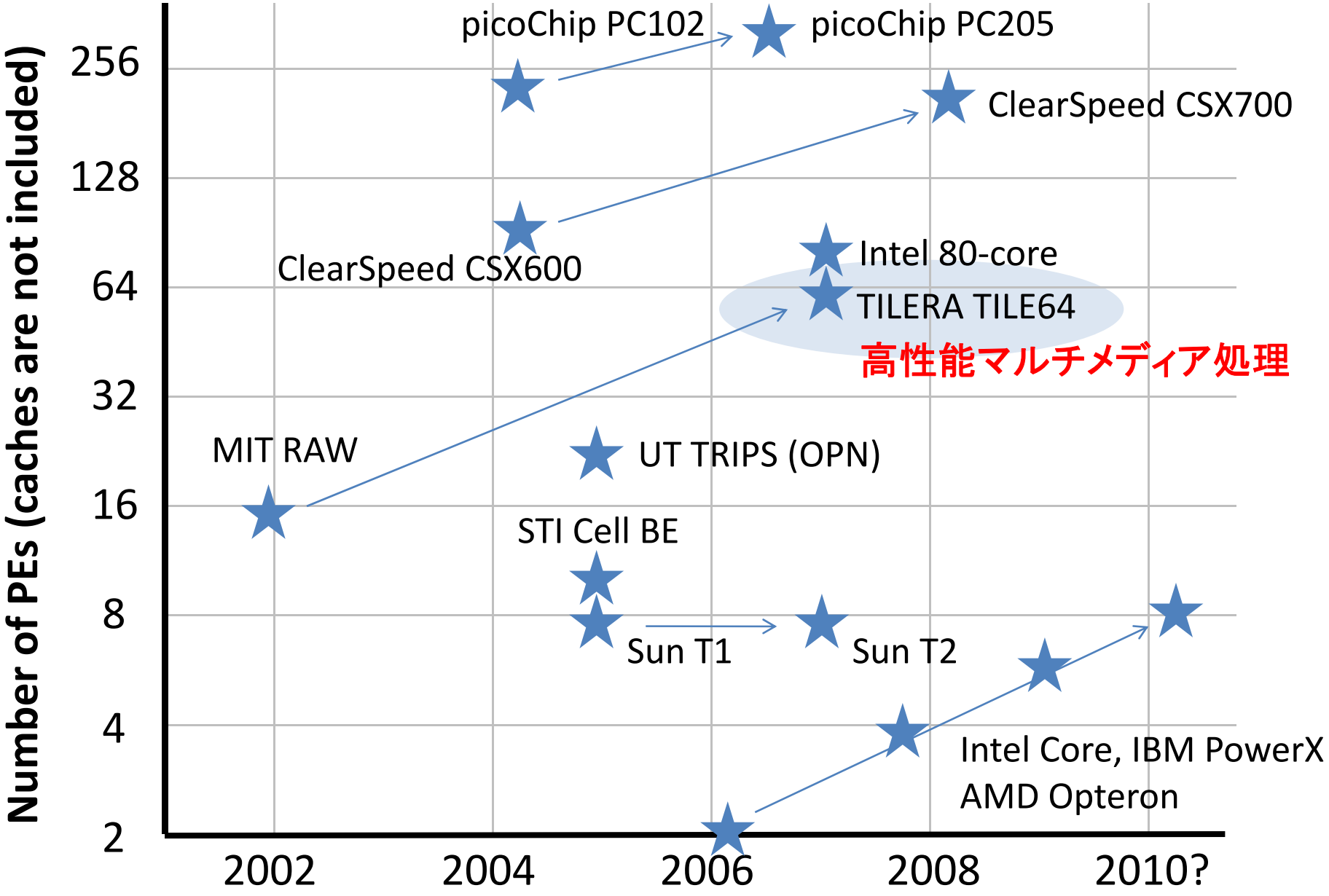
実システムの例: OpenSPARC T2



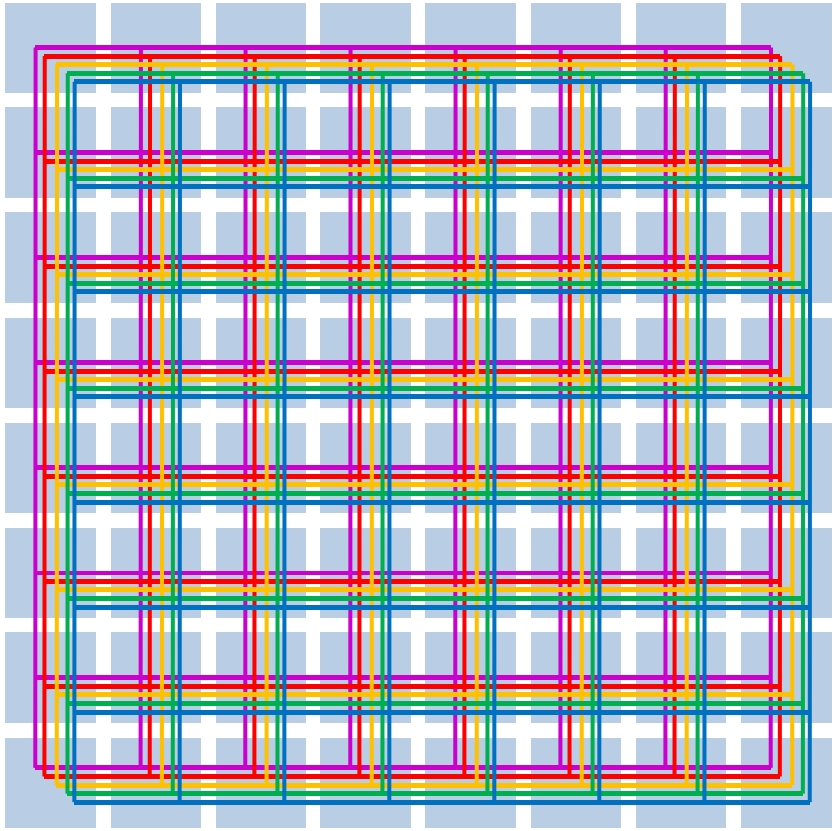
http://www.opensparc.net/

- Sun UltraSPARC T2 のオープンソース版
- OpenSPARC T2 の構成
  - SPARC プロセッサ (最大8スレッド処理) 8個
  - L2 キャッシュバンク 8個
  - クロスバで接続
- CPU-Cache Crossbar (CCX)
  - リンクデータ幅 128-bit

実システムの例: TILERA TILE64



実システムの例: TILERA TILE64



5種類の静的&動的ネットワーク:
MDN TDN UDN IDN STN

- タイルの構成
  - VLIW プロセッサ
  - L1 / L2 キャッシュ
  - オンチップルータ
- ネットワークの特徴
  - 8x8 mesh
  - 5系統のネットワーク
  - リンクデータ幅 32-bit
  - 次元順ルーティング
  - Wormhole スイッチング

http://www.tilera.com/products/TILE64.php

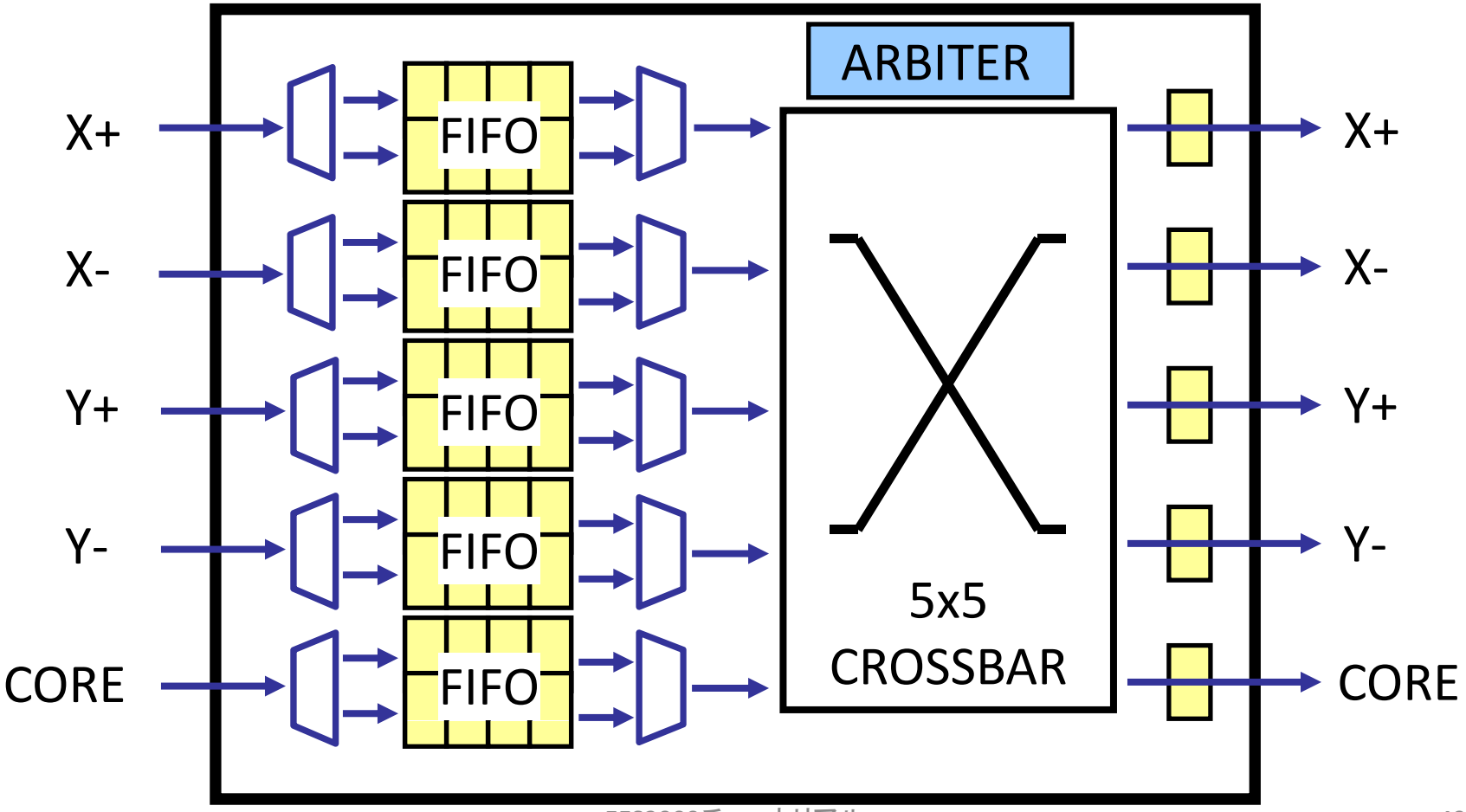
如何にして「低消費電力NOC」を実現するのか?

低消費電力化に向けて

- 電圧と周波数の制御 (DVFS)
- 電力供給のストップ (Power gating)
- バッファリング回数の削減
  - 平均ホップ数の小さいトポロジの採用
  - 中継ルータのスキップ
- 3次元化 (複数のチップを積み重ねる)
  - パケット移動距離の短縮

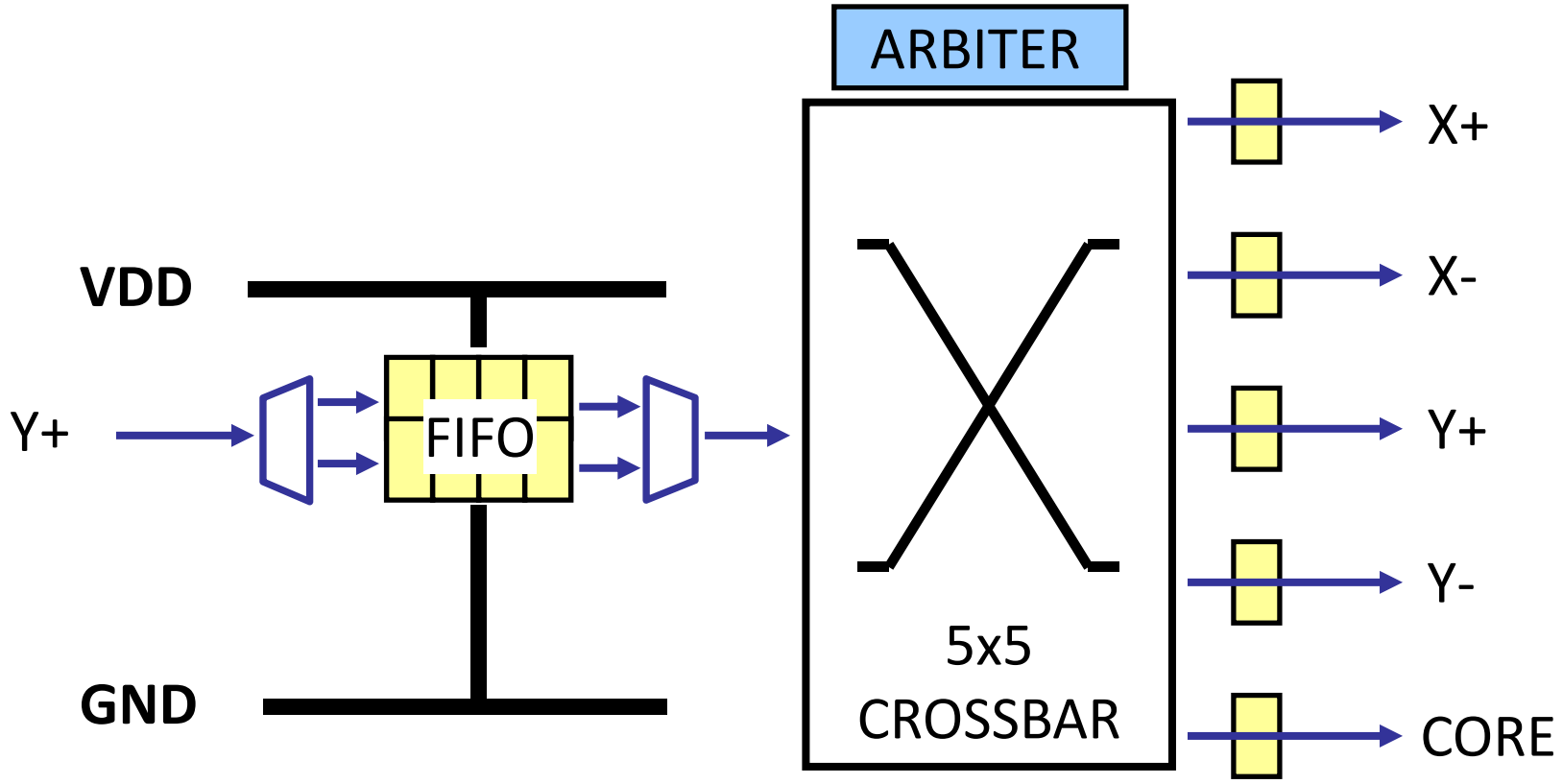
低消費電力化: Runtime Power Gating

- パケットが来たらバッファの電源 ON　通過したら OFF



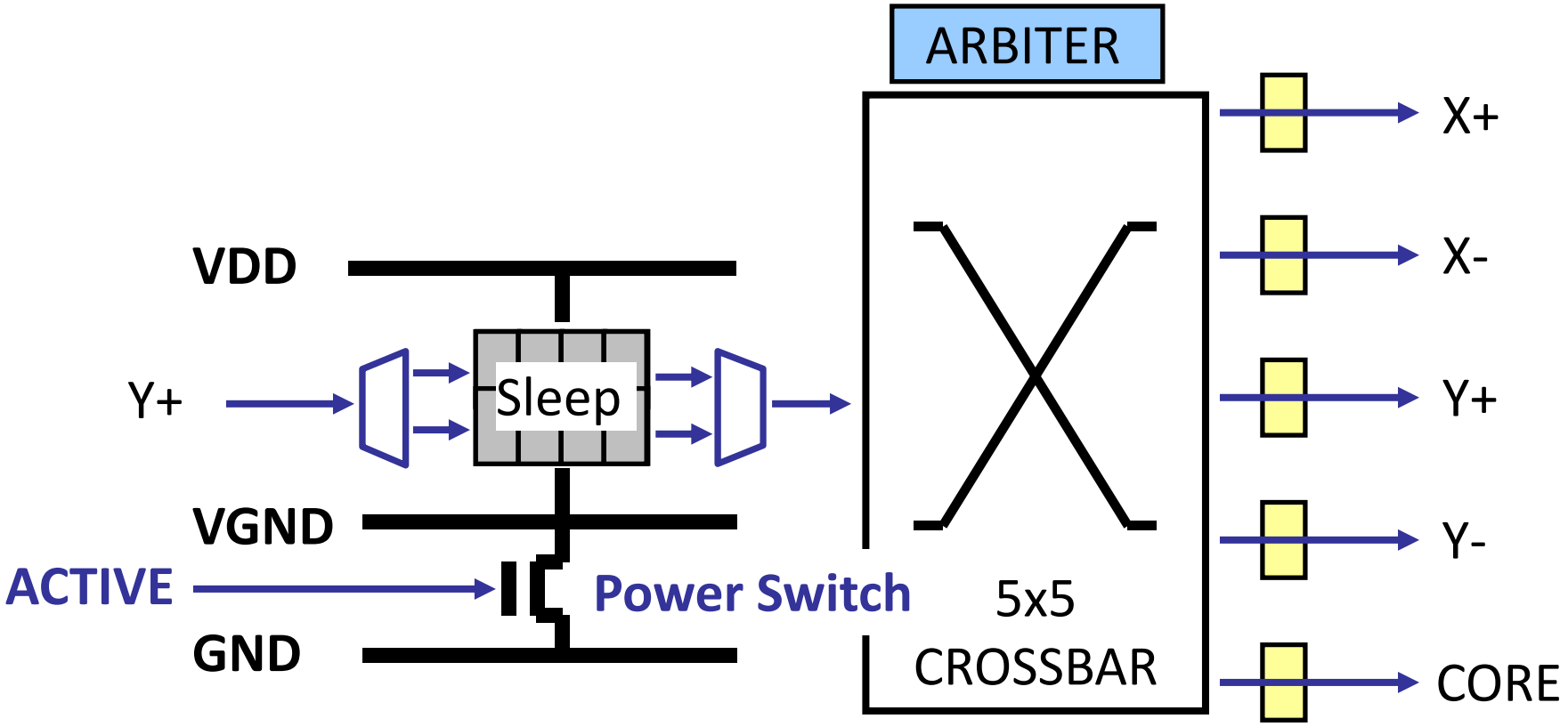
低消費電力化: Runtime Power Gating

- パケットが来たらバッファの電源 ON　通過したら OFF



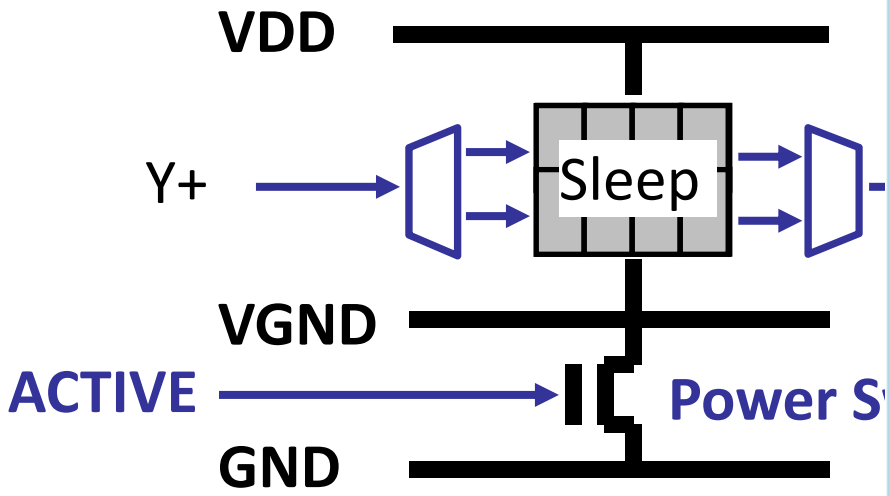
低消費電力化: Runtime Power Gating

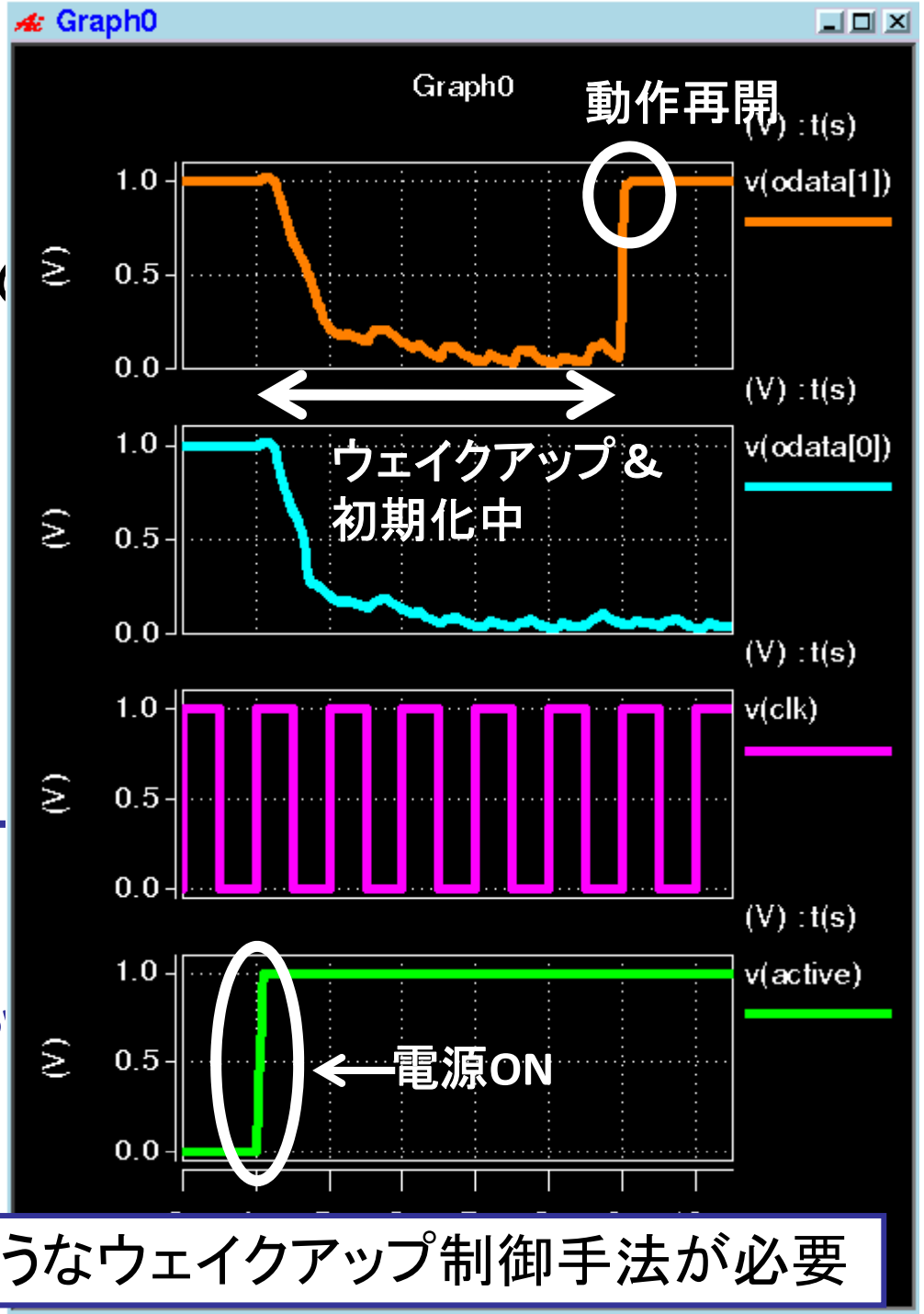
- パケットが来たらバッファの電源 ON　通過したら OFF



低消費電力化:

- パケットが来たらバッファ(





ウェイクアップ遅延を隠蔽するようなウェイクアップ制御手法が必要

低消費電力化に向けて: 他にも課題は多い

- 電圧と周波数の制御 (DVFS)
- 電力供給のストップ (Power gating)
- バッファリング回数の削減
  - 平均ホップ数の小さいトポロジの採用
  - 中継ルータのスキップ
- 3次元化 (複数のチップを積み重ねる)
  - パケット移動距離の短縮

チュートリアル内容

- なぜメニーコアなのか?～「マルチ」から「メニー」の世界へ～
- メニーコアを支える要素技術
  - プロセッサ/メモリ・アーキテクチャ
  - ネットワーク・オン・チップ
  - 自動並列化コンパイラ
- 今後の展望

プログラムを「並列化する」とは?

メニーコア時代における自動並列化コンパイラの可能性

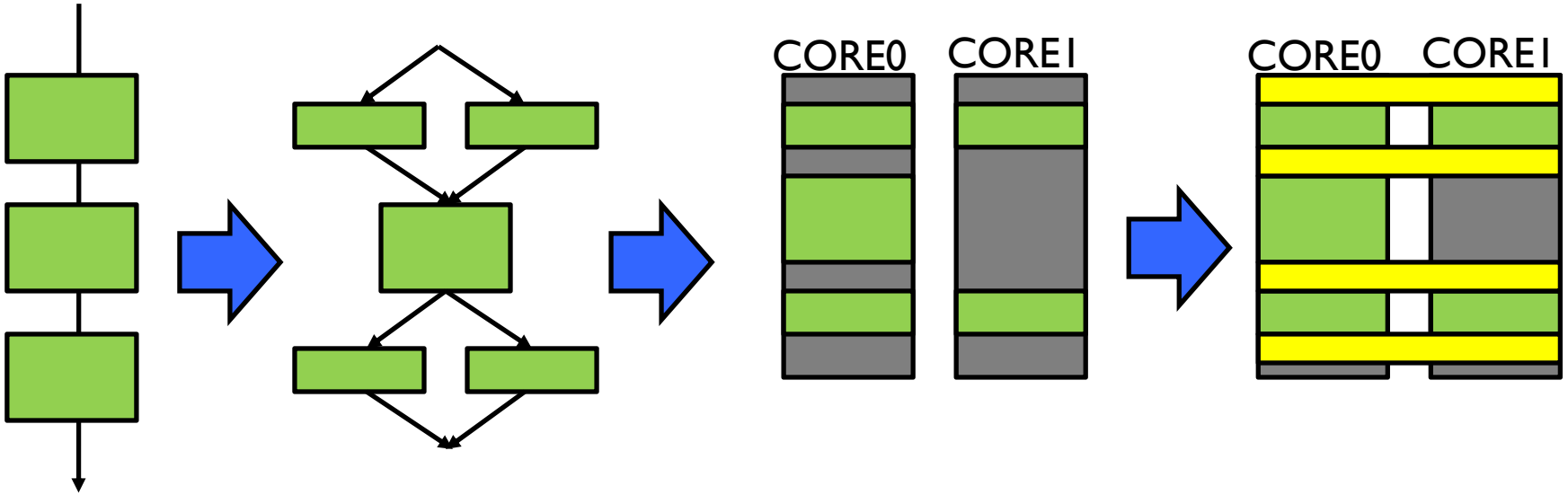
- メニーコアの足音が聞こえてきます
  - 128コアあれば性能を128倍にしてほしい
  - でも消費電力は上げないで
- ソフトウェア開発は誰がやるの？
  - とりあえず並列化プログラムを記述しやすくするプログラミング言語とかあるらしい
    - Erlangとか
  - なんとかなりそう

ほんと？

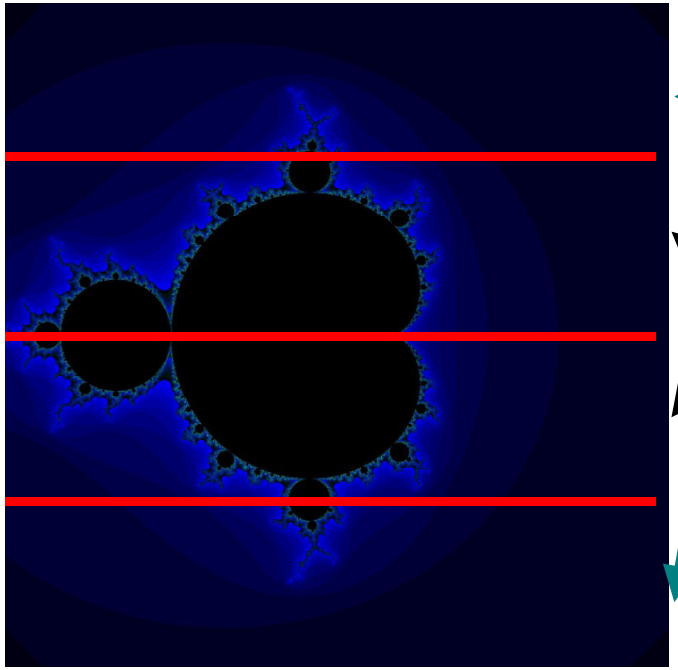
コンパイラで自動的に並列化したい

プログラムの並列化とは

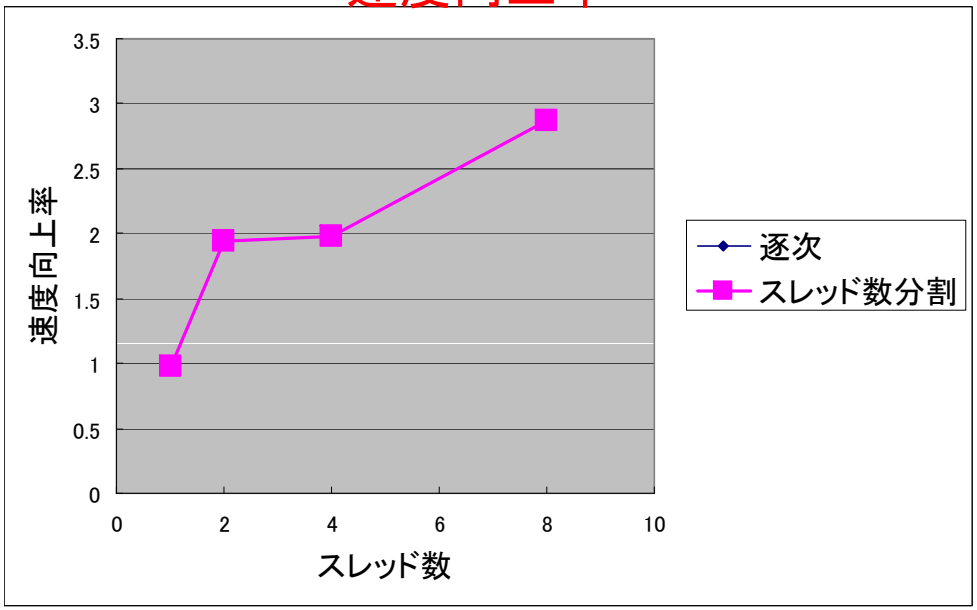
- 並列化可能な箇所の特定
- 並列処理単位(タスク)への分割
- タスクのコアへの割り当て(スケジューリング)
- 同期コード・(必要なら)データ転送コードの挿入



負荷分散
（マンデルブロ集合並列化を例に）



スレッド数を増やしたときの速度向上率



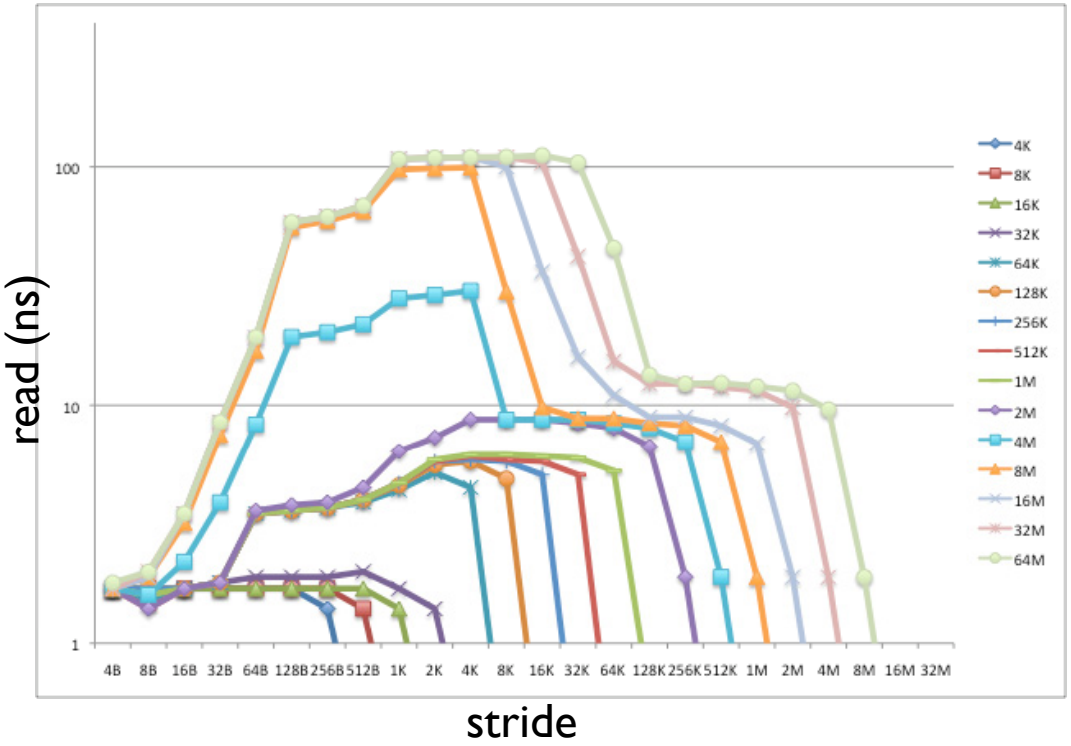
- 以下の複素数列が2に達するまでの計算回数

$$Z_n = \sqrt{Z_{n-1}{}^2 + C}$$

- 複素平面上の各点に対して計算
  - 各点の計算は完全に独立

遠くなるメモリ

- メモリはどんどん遠くなる
  - 近接メモリ（キャッシュ、スクラッチパッドメモリ等）の効率的な利用が重要
- チップ内外を問わずバンド幅は限られる
  - コア間共有データ、コア固有データをどのように保持するべきか？
  - データ転送はどうする？



ストライドを変えたメモリアクセスによりキャッシュ周りのパラメータを測定した結果（参考：ヘネパタ第4版）

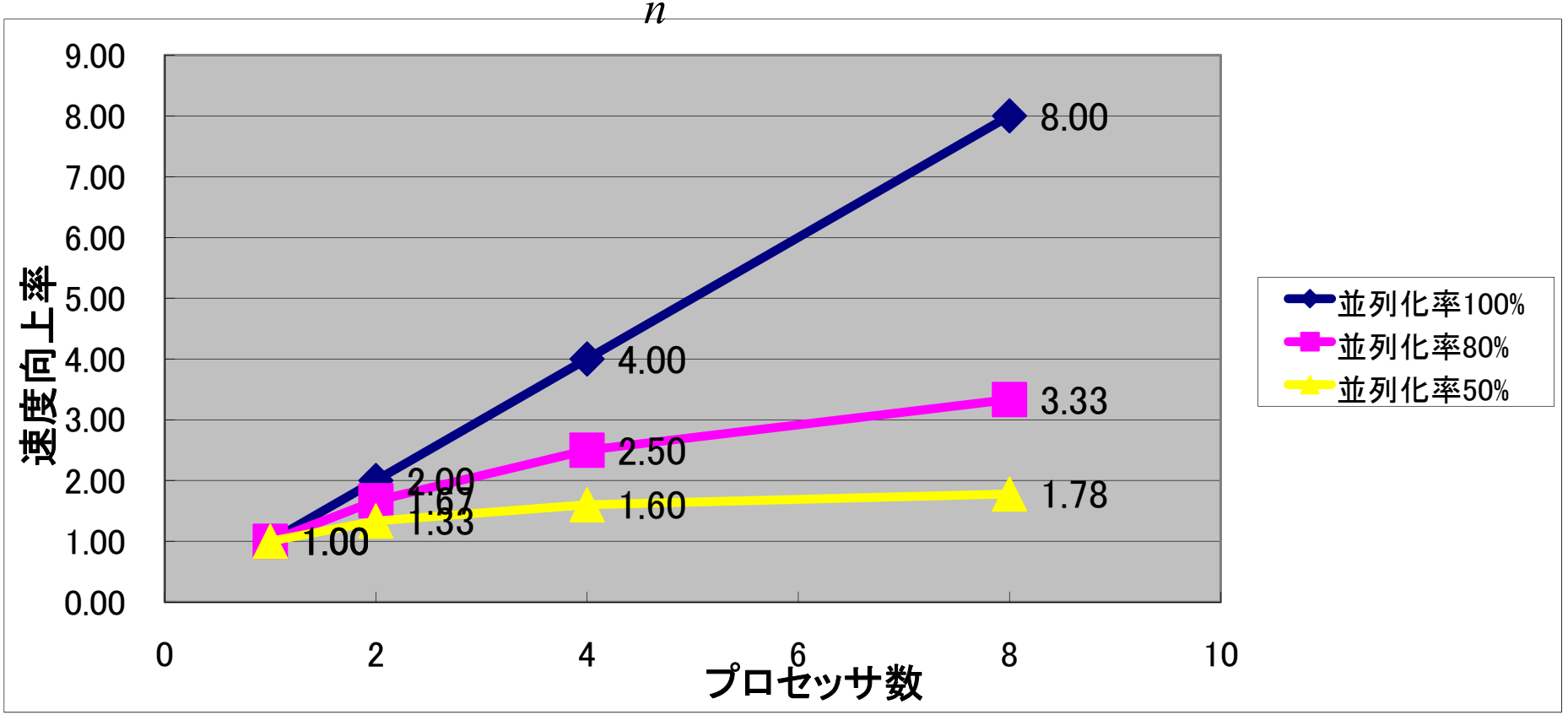
測定環境：
Intel xeon x5365@3GHz (4x2 core)
L1Dcache 32KB
L2cache 4MB/2core

アムダールの法則

- 最適化の効果は最適化可能な箇所の割合に制限される

$$speedup = \frac{1}{1 - \frac{n-1}{n} p}$$

p: 並列化率, *n*: プロセッサ（コア）数



並列化コンパイラ

- 入力プログラムを並列化する
  - 並列化可能な箇所を特定する
  - 処理を分割する
  - 処理をスレッドに割り当てる
- 一般的にループを並列化のターゲットとする

```
for (i=0; i<1000; i++)
  sum += a[i];
```

CPU0用

```
for (i=0; i<250; i++)
  sum0 += a[i];
barrier(var); /*待ち合わせ*/
sum = sum0+sum1+sum2+sum3;
```

CPU1用

```
for (i=250; i<500; i++)
  sum1 += a[i];
barrier(var);
```

CPU2用

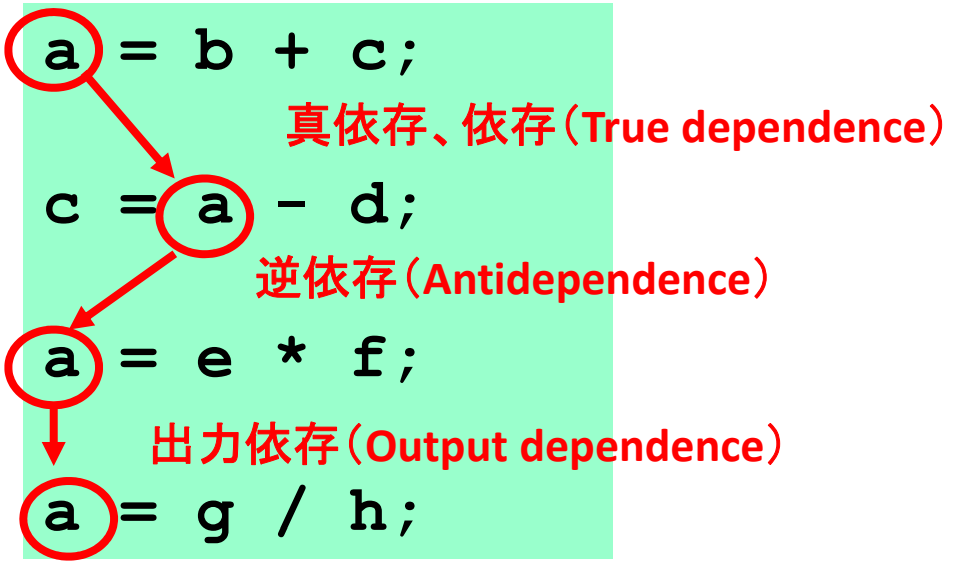
```
for (i=500; i<750; i++)
  sum2 += a[i];
barrier(var);
```

CPU3用

```
for (i=750; i<1000; i++)
  sum3 += a[i];
barrier(var);
```

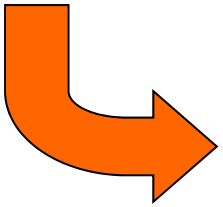
並列化可能とは？

- 複数の処理が同時に実行できる
  - 複数の処理を任意の順番で実行しても結果が変わらない（処理が依存しない）
- 処理が依存するとは？



ループの並列化

```
for (i=0; i<n; i++) {
  a[i]   = b[i] + c[i];   // S1
  c[i+1] = a[i] + d[i];   // S2
  e[i]   = c[i+2]+c[i+1]; // S3
  a[i+1] = f[i] + 1.0;    // S4
}
```



```
S1(0): a[0] = b[0] + c[0];
S2(0): c[1] = a[0] + d[0];
S3(0): e[0] = c[2] + c[1];
S4(0): a[1] = f[0] + 1.0;
        O    A
S1(1): a[1] = b[1] + c[1];
S2(1): c[2] = a[1] + d[1];
S3(1): e[1] = c[3] + c[2];
S4(1): a[2] = f[1] + 1.0;
        O    A
S1(2): a[2] = b[2] + c[2];
S2(2): c[3] = a[2] + d[2];
S3(2): e[2] = c[4] + c[3];
S4(2): a[3] = f[2] + 1.0;
```

ループ繰り越し依存
(loop carried dependence)

参考：中田育男、「コンパイラの構成と最適化」、朝倉書店

「実」自動並列化コンパイラ：OSCAR

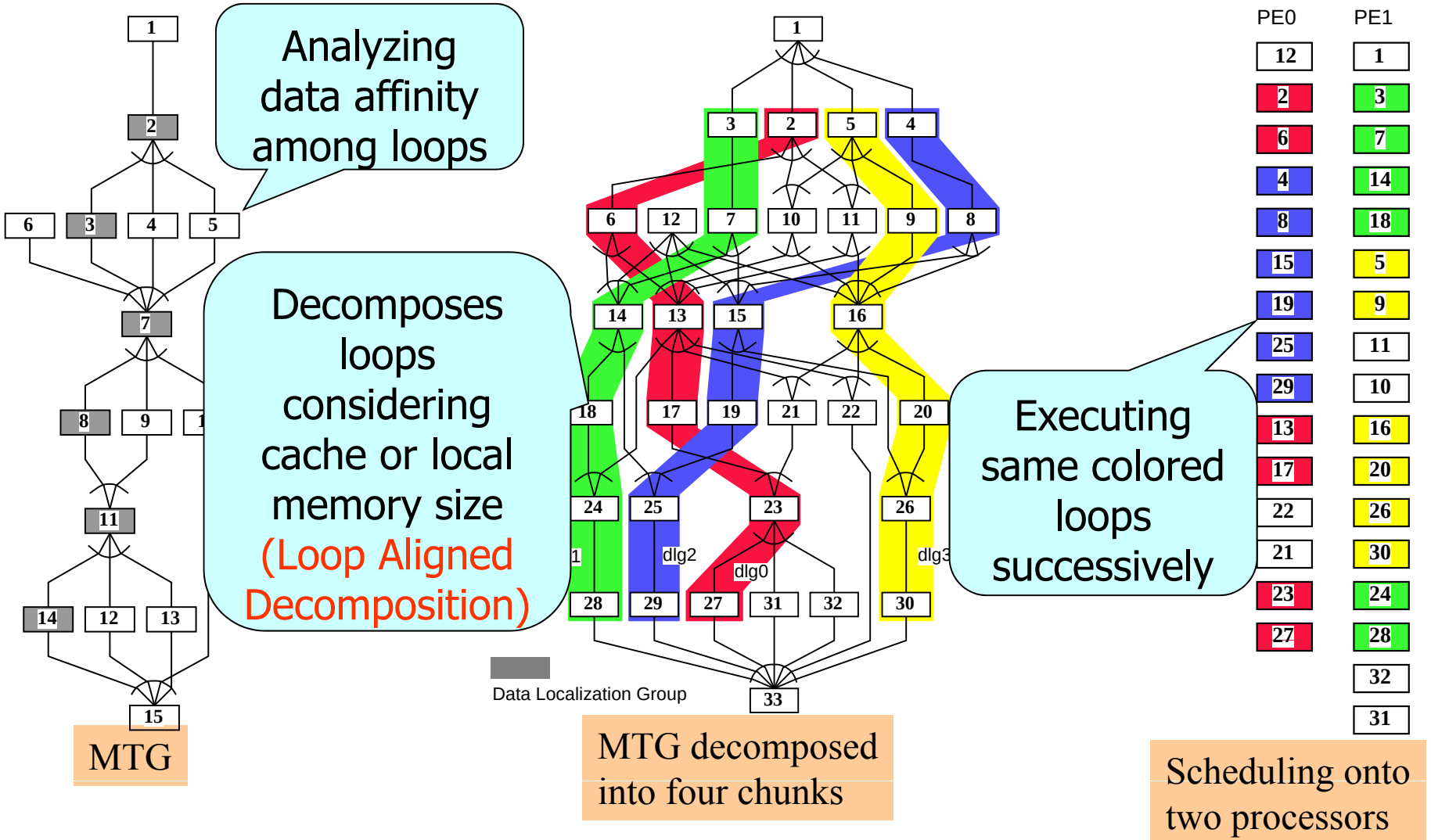
OSCARコンパイラ

- 早稲田大学 笠原・木村研で開発している自動並列化コンパイラ
- マルチグレイン並列化
  - プログラム全域にわたる並列化
- メモリ最適化
  - キャッシュ・ローカルメモリ利用の最適化
- データ転送最適化
  - データ転送隠蔽技術
- 低消費電力化
- ヘテロジニアス・スケジューリング

マルチグレイン並列処理

- プログラム全域にわたる最適化
  - ループイタレーションレベル並列化
    - 通常の並列化コンパイラ
  - 粗粒度タスク並列化
  - 近細粒度並列化

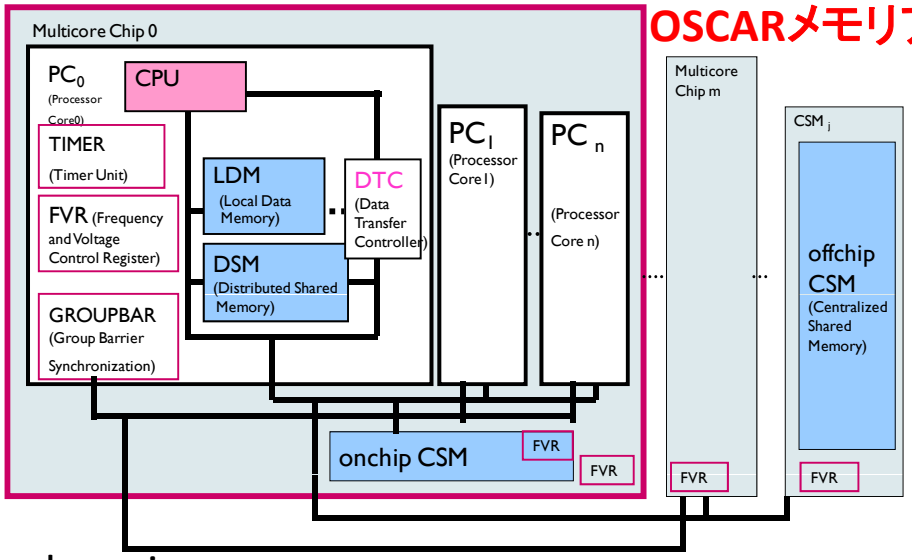
データローカリティ最適化



OSCAR API

- 主にリアルタイム情報家電機器が対象
  - 様々なメモリアーキテクチャ
    - SMP, ローカルメモリ, 分散共有メモリ, …
- 産官学連携
  - 日立、ルネサス、富士通、東芝、パナソニック、NEC
  - 経済産業省・NEDOのサポート
- OpenMPのサブセットがベース
  - 共有メモリモデルの、サーバ機等で非常によく使われる並列化API
  - OpenMPコンパイラでコンパイル可能
- 6つのカテゴリ
  - 並列実行
  - メモリ配置
  - データ転送
  - 電力制御
  - タイマ
  - 同期

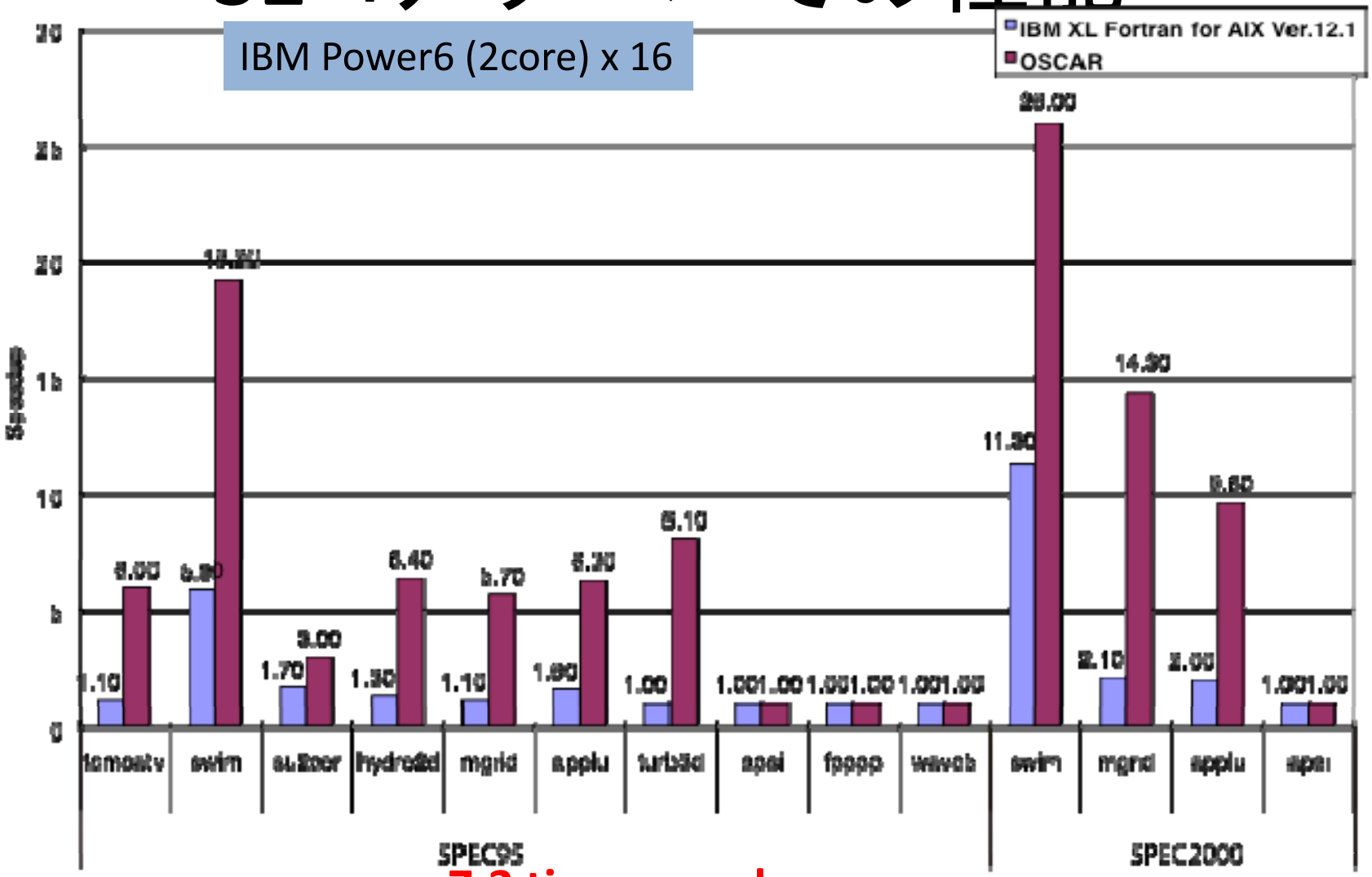
リファレンスとなる
OSCARメモリアーキテクチャ



以下で公開:
http://www.kasahara.cs.waseda.ac.jp

32コアサーバでの性能

IBM Power6 (2core) x 16



7.3 time speedup on average

3.3 times acceleration over XL Fortran

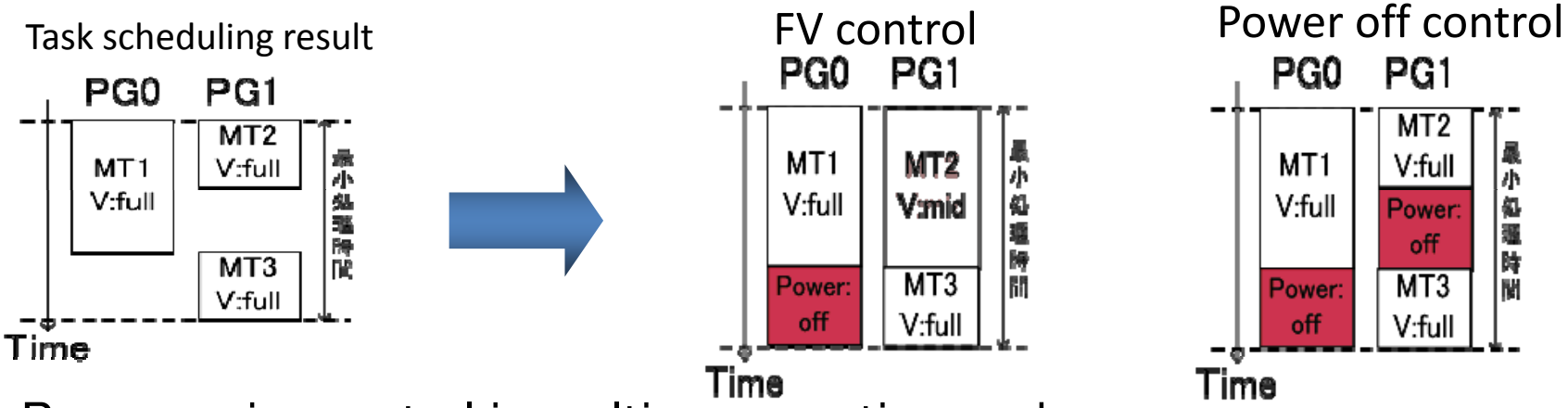
メニーコアを支える今後のコンパイル技術

これからのコンパイラ技術

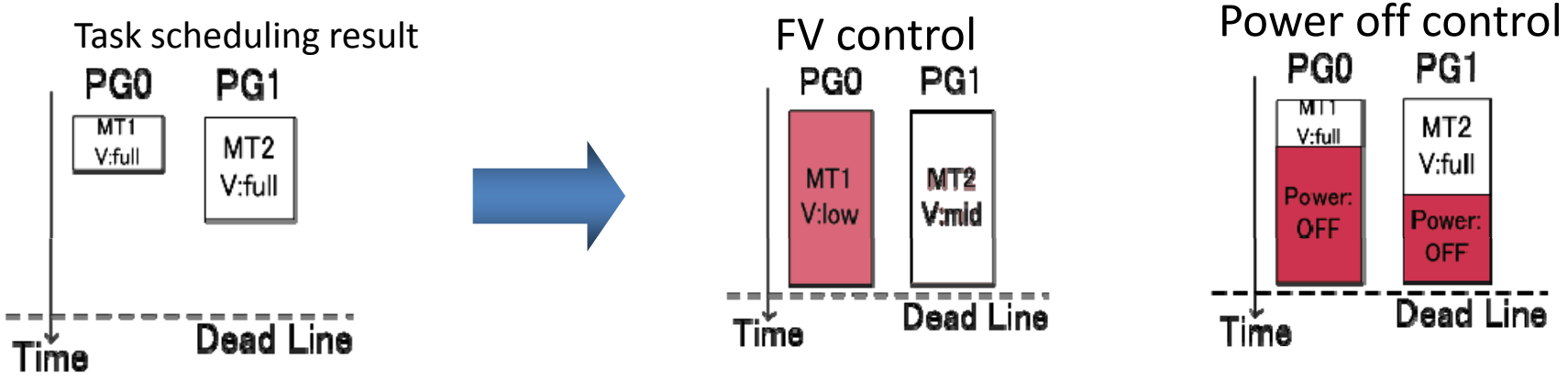
- コンパイラによる低消費電力化
  - コア数が増えても消費電力は抑えたい
- ヘテロジニアスマルチコア用コンパイル技術
  - いろいろなコアが混載される状況に対応
- Parallelizable C
  - 並列処理用プログラミングガイドライン
    - コンパイラ技術そのものではないですが…

コンパイラによる低消費電力化

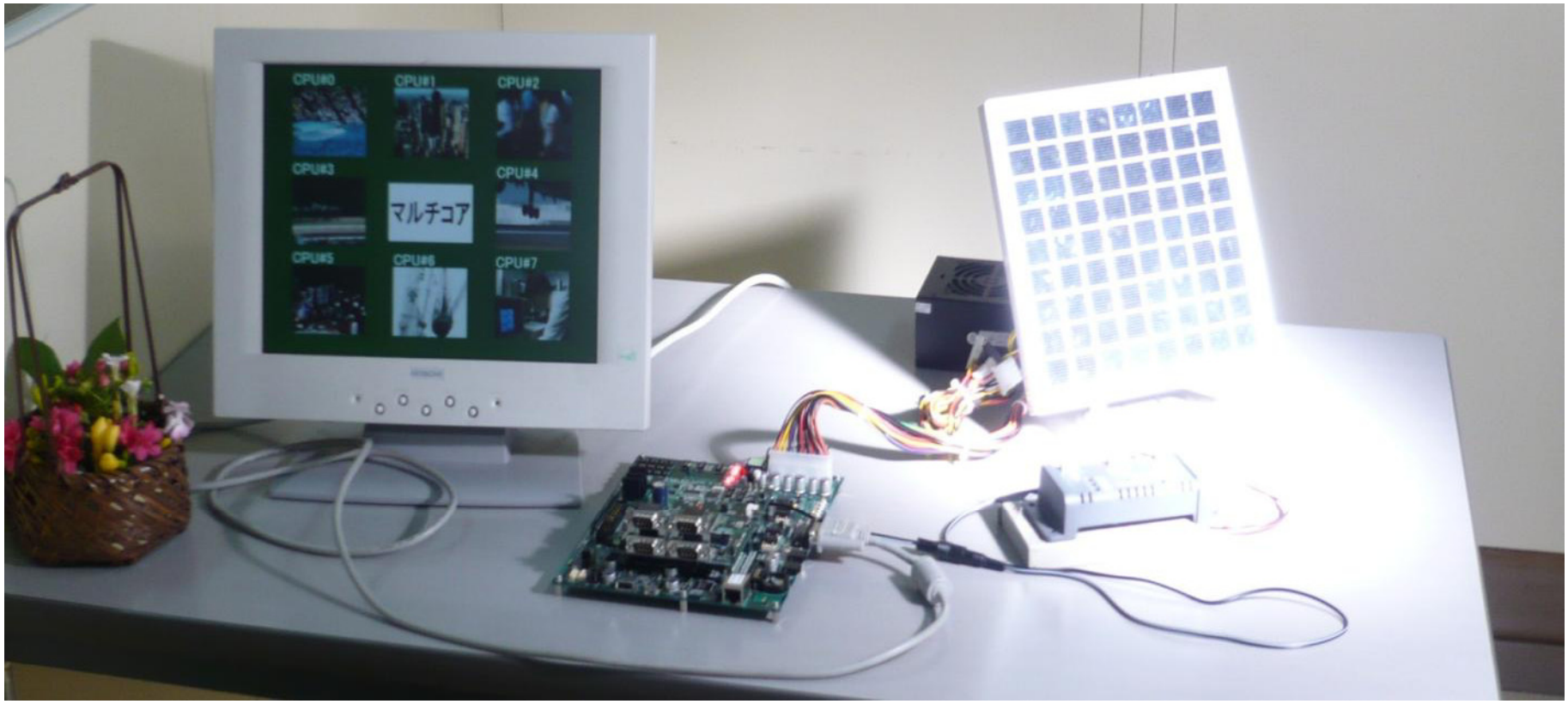
- Power saving control in the fastest execution mode



- Power saving control in realtime execution mode

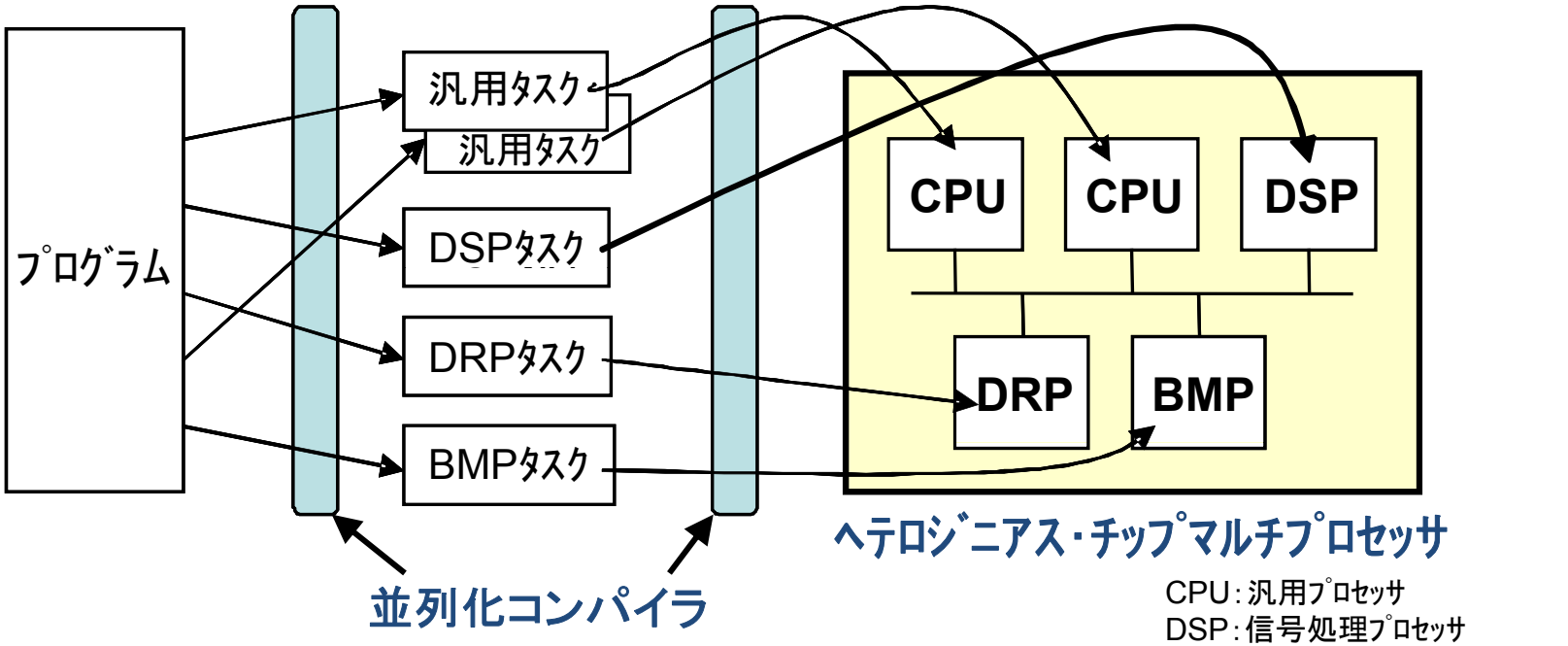


Solar Powered RP2 Multicore Chip



ヘテロジニアスマルチコア用コンパイラ技術

- 多種類の計算エンジン(プロセッサ)を1チップに集積したSoCアーキテクチャ
- プログラムの並列性を抽出し、各プロセッサの特徴に適したタスクの分割と配置を行う並列化コンパイラ

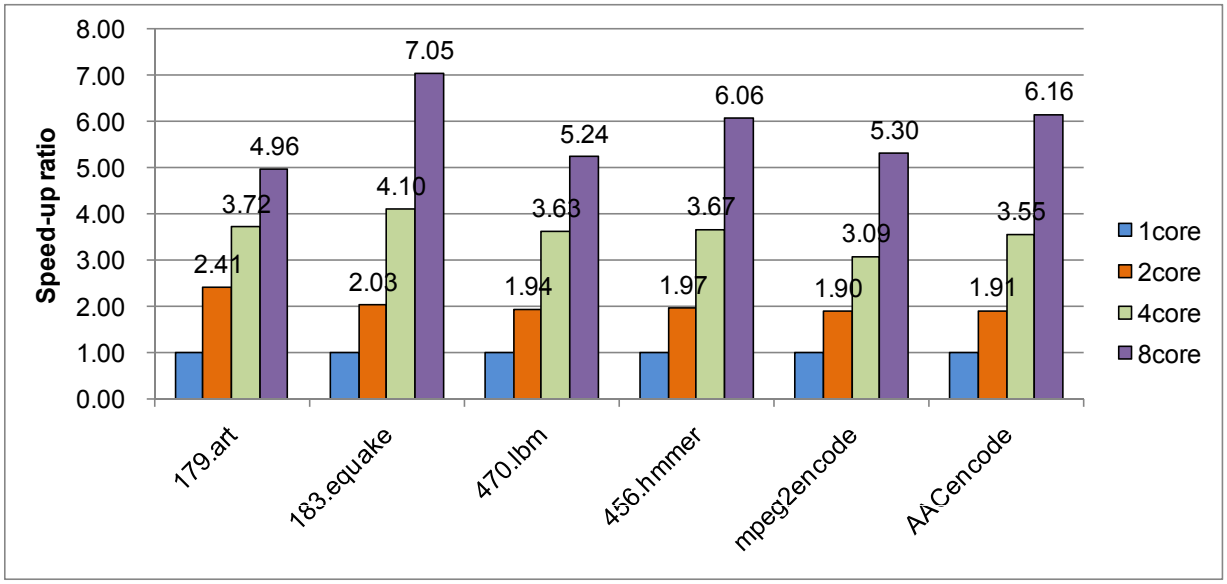


CPU:汎用ﾌﾟﾛｾｯｻ
DSP:信号処理ﾌﾟﾛｾｯｻ
DRP:動的再構成可能ﾌﾟﾛｾｯｻ
BMP:ﾋﾞｯﾄ処理ﾌﾟﾛｾｯｻ

Parallelizable C

- 並列処理用プログラミングガイドライン
- Cで記述されたプログラムは解析が困難
  - ポインタの柔軟性に起因
  - ポインタの利用に制限を設ける

Cで記述されたプログラムの解析が可能に

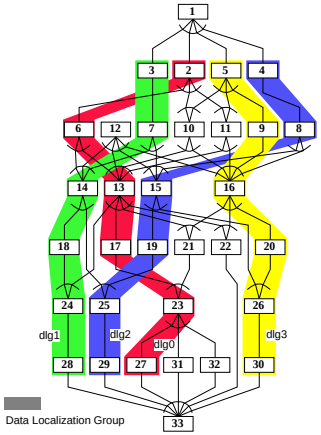


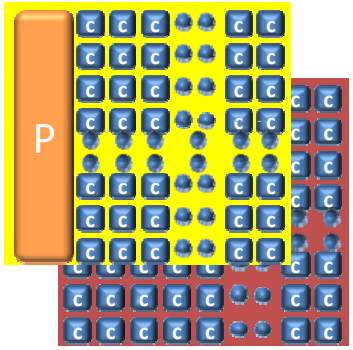
IBM Power5+
8コアWSでの性能

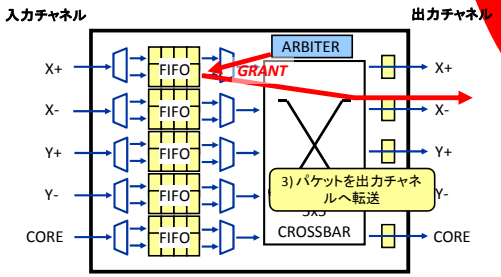
チュートリアル内容

- なぜメニーコアなのか?～「マルチ」から「メニー」の世界へ～
- メニーコアを支える要素技術
  - プロセッサ/メモリ・アーキテクチャ
  - ネットワーク・オン・チップ
  - 自動並列化コンパイラ
- 今後の展望

実装からプログラミングレベルまでの統合的な技術開発が必要!!







応用技術
（On-Line画像処理など）

プログラミング技術
（API標準化, Tuningなど）

コンパイラ/OS技術
（自動並列化, VMなど）

アーキテクチャ技術
（CPU/メモリ構成, NOCなど）

回路設計技術
（Sub-Threshold回路など）

半導体製造/実装技術
（3次元積層など）