



# プロセッサー・キャッシュ最適化技法

インテル株式会社  
ソフトウェア&ソリューションズ統括部  
ソフトウェア製品部



# はじめに

L1キャッシュでのアクセスミスは**数十クロック**のペナルティーが生じる

L2キャッシュでのアクセスミスは**数十バスクロック**のペナルティーが生じる

キャッシュを有効利用するにはデータやコードの位置関係が重要となる

しかしそれは...

他の最適化技術に影響を及ぼす場合がある

# キャッシュに読み込まれるタイミング

1. アプリケーションが参照したメモリーの内容がキャッシュにない場合
2. アプリケーションがメモリーに書き込みを行った内容がキャッシュにない場合
3. アプリケーションがプリフェッチ命令を実行した場合
4. ハードウェア・プリフェッチャーが動作した場合

読み込み書き出しの最小単位はキャッシュライン(64バイト)

# ハードウェア・プリフェッチャー

ハードウェアは 2 つの方法でデータを取り込む

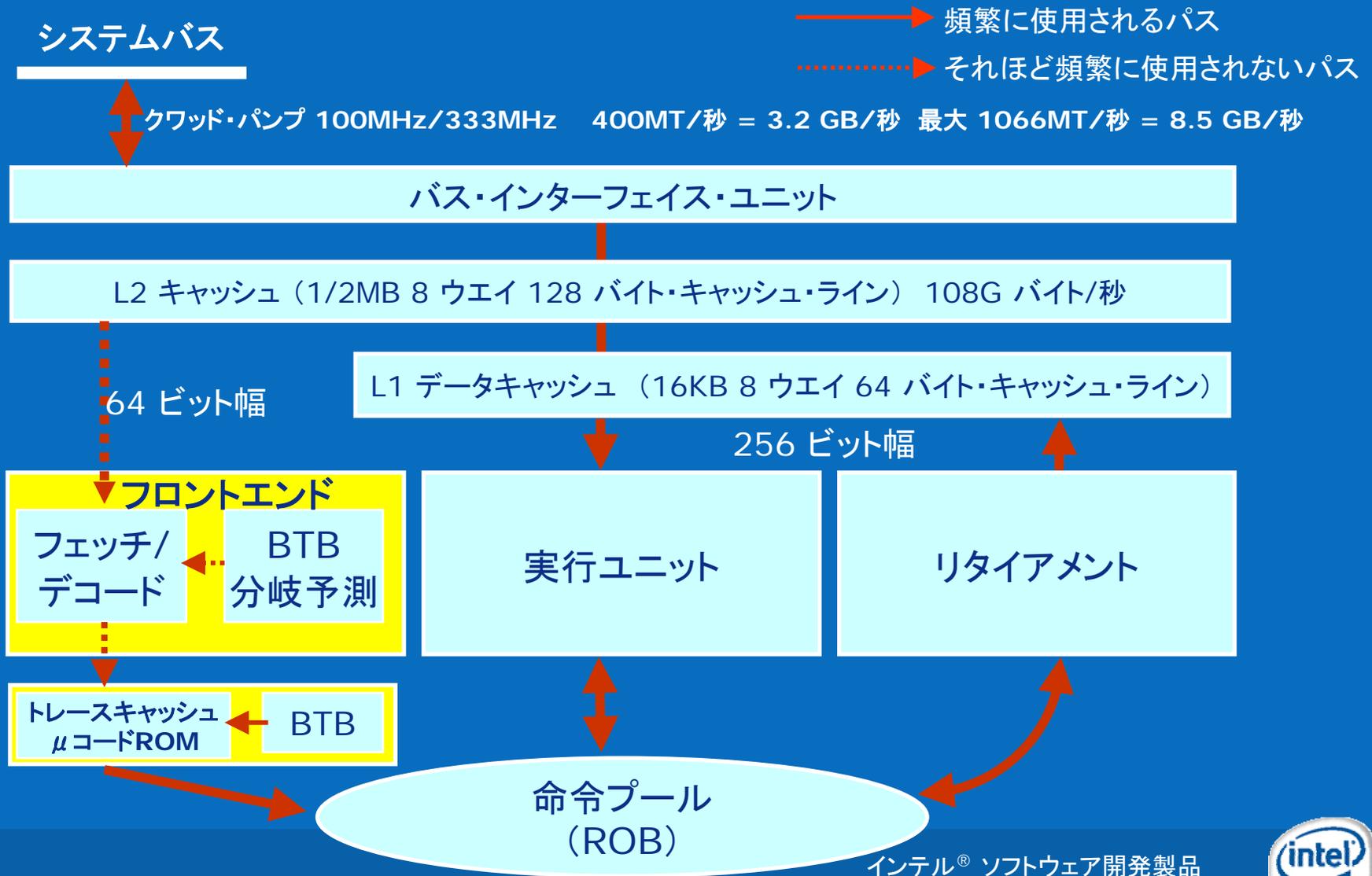
- ストライド・プリフェッチ

- データ・アクセス・パターンを基にキャッシュラインのプリフェッチを開始する
  - いくつかのキャッシュラインを取得する
  - 通常のデータ・アクセス・パターンのときによりパフォーマンスが得られるようにする
- キャッシュライン参照の読み込みミスがトリガーになる
  - しきい値内での 2 回のキャッシュミスがトリガーになる
    - 90 ナノ・メートル・テクノロジーのインテル® Pentium® 4 プロセッサでは 512 バイト
    - それ以前のインテル® Pentium® 4 プロセッサでは 256 バイト
  - 次のノードへの距離をトリガーとなる距離の 1/2 以上に保つ

- 隣接ライン・プリフェッチ

- 128 バイトのフェッチで L2(L3)ミスを考慮する
  - キャッシュミスが下位 64 バイトで起こったなら、隣接する上位 64 バイトを取り込む
  - キャッシュミスが上位 64 バイトで起こったなら、隣接する下位 64 バイトを取り込む
- キャッシュライン参照の読み込みミスがトリガーになる

# Intel NetBurst® マイクロアーキテクチャー



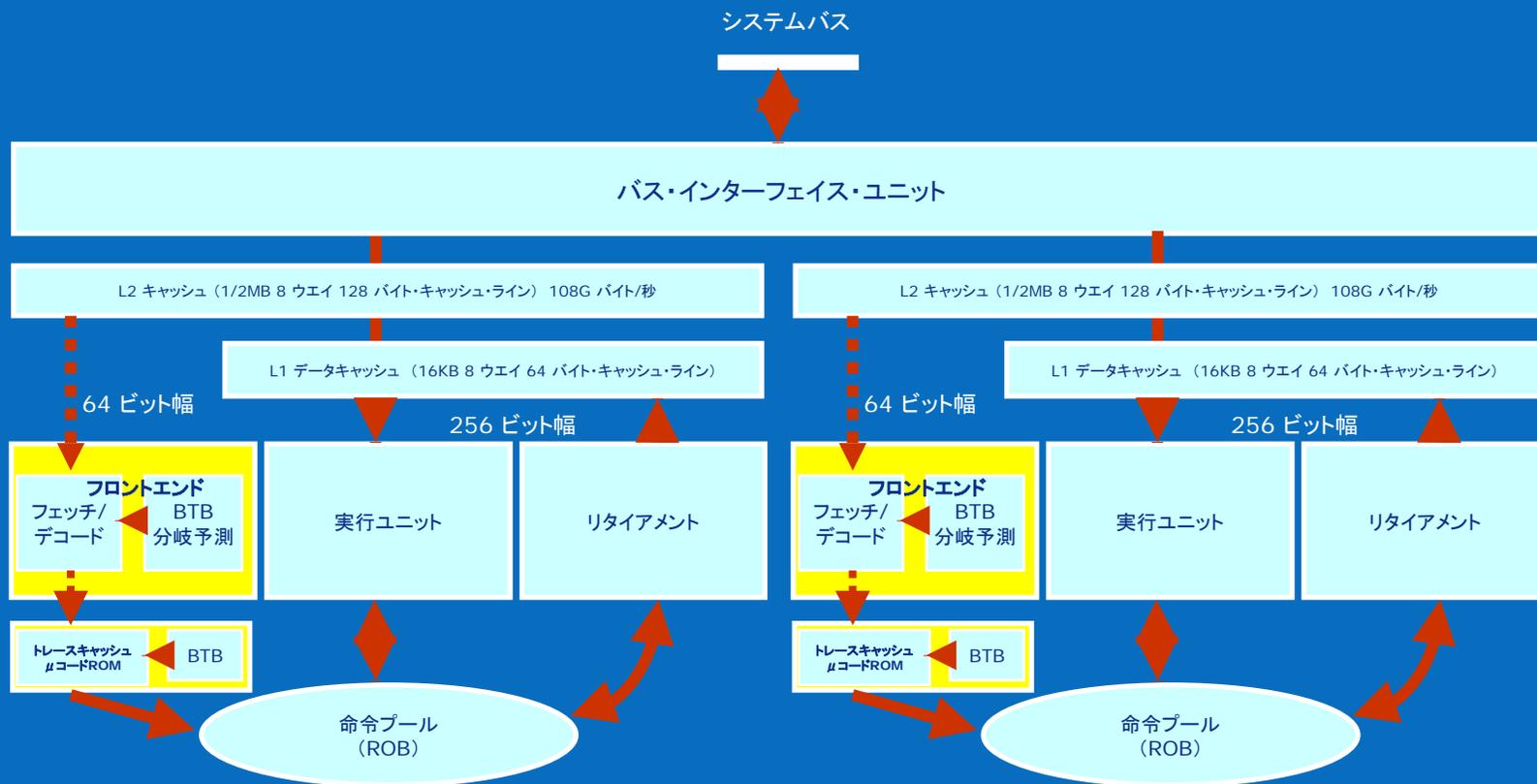
# インテル® モバイル・マイクロアーキテクチャー

→ 頻繁に使用されるパス  
→ それほど頻繁に使用されないパス



# 第1世代のデュアルコア・プロセッサ

Pentium® D、Pentium® Extreme Edition



# インテル® Core™ マイクロアーキテクチャー

システムバス

クワッド・パンプ 166MHz/266MHz/333MHz

667MT/秒 = 5.3 GB/秒、1066MT/秒 = 8.5GB/秒、1333MT/秒 = 10.6GB/秒

バス・インターフェイス・ユニット

アドバンスド・スマートL2 キャッシュ (2MB/4MB 8ウェイ 128バイト・キャッシュ・ライン)

256ビット幅

L1 命令キャッシュ  
(32KB 8ウェイ)

L1 データキャッシュ  
(32KB 8ウェイ)

L1 命令キャッシュ  
(32KB 8ウェイ)

L1 データキャッシュ  
(32KB 8ウェイ)

64ビット幅

64ビット幅

5 フロントエンド  
フェッチ/ BTB  
デコード 分岐予測

実行ユニット(5つ)

リタイアメント

5 フロントエンド  
フェッチ/ BTB  
デコード 分岐予測

実行ユニット(5つ)

リタイアメント

マクロ・オペレーションのフュージョン

マクロ・オペレーションのフュージョン

4 マイクロ・オペレーション(μOP)のフュージョン

4 マイクロ・オペレーション(μOP)のフュージョン

命令プール  
(ROB)

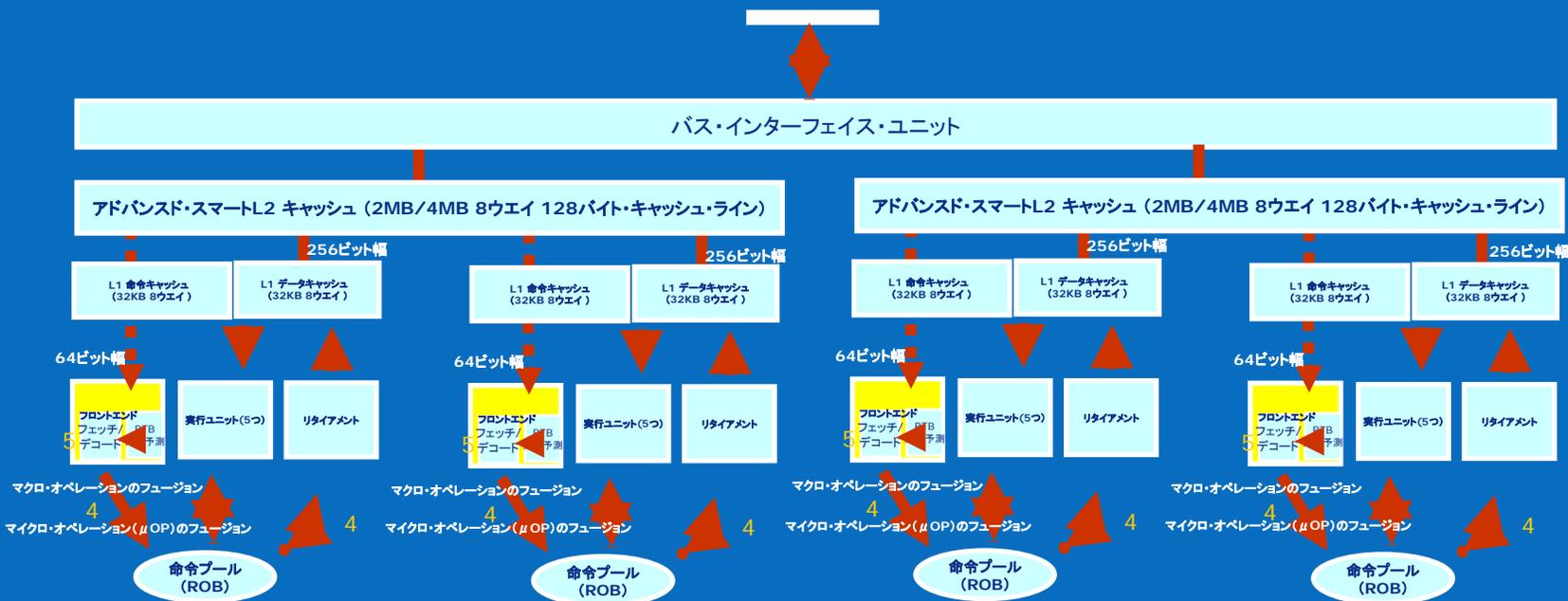
命令プール  
(ROB)

→ 頻繁に使用されるパス

→ それほど頻繁に使用されないパス

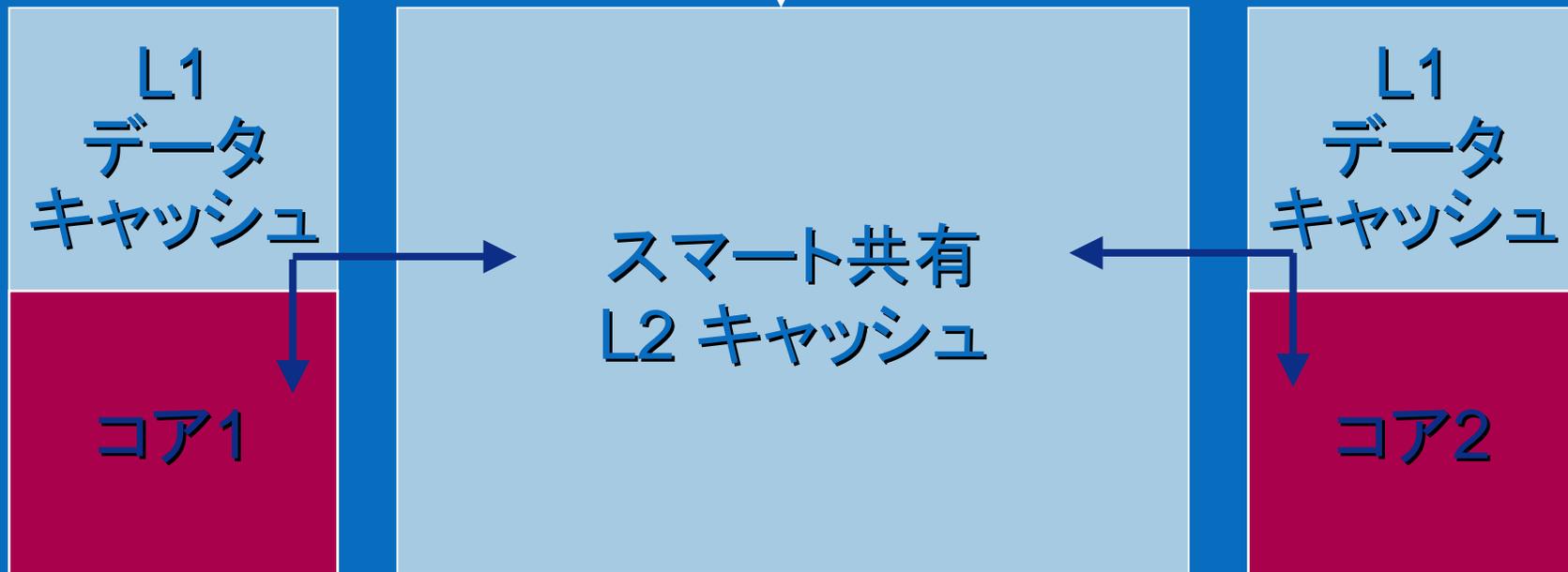
# インテル® Core™ マイクロアーキテクチャー クアドコア・プロセッサ

システムバス



# スマート・メモリー・アクセス

システムバス

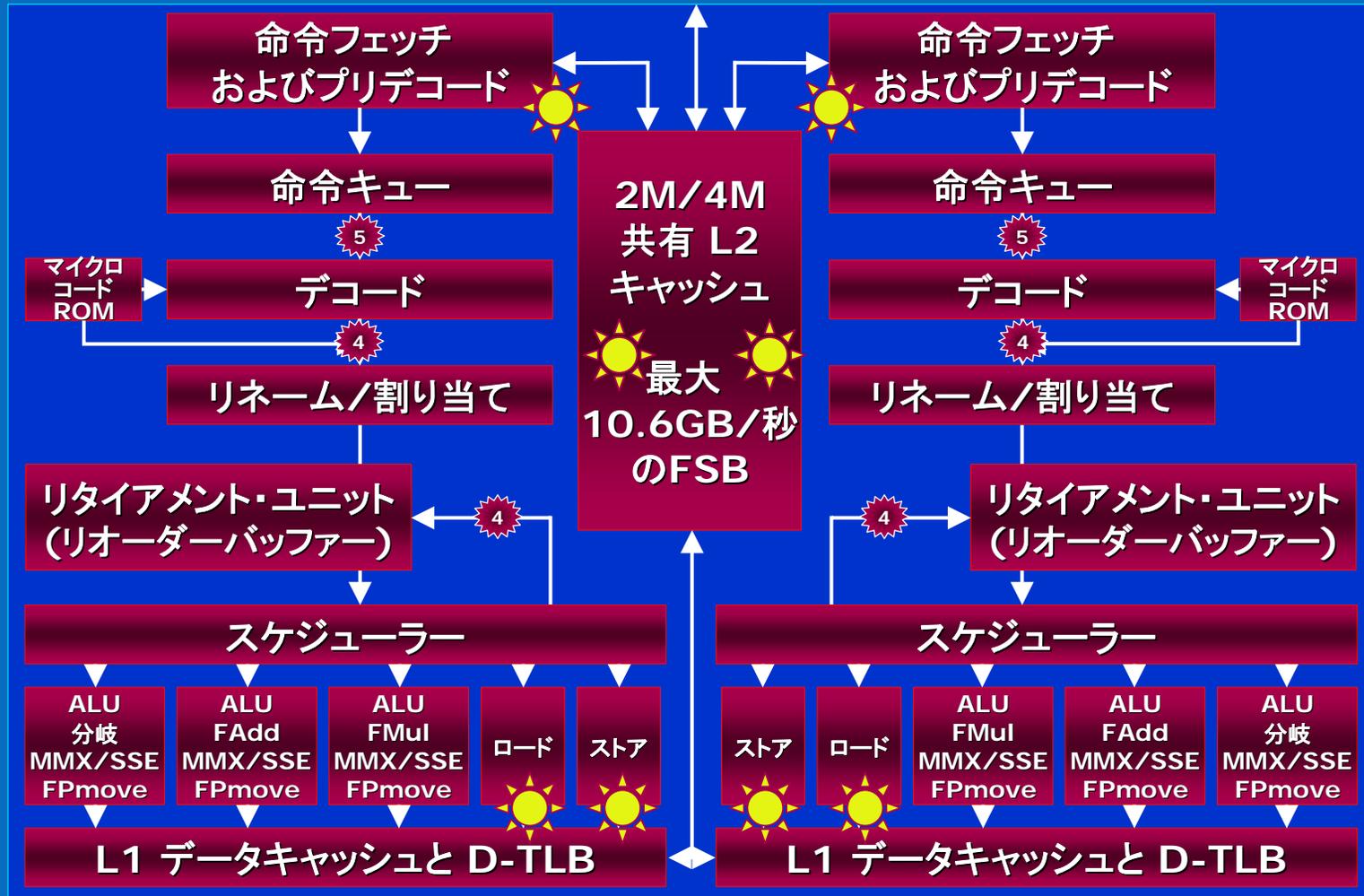


時間の局所性  
空間の局所性

データを可能な限り早く利用できるようにする  
データが可能な限り近くにあるようにする

メモリー・サブシステムのレイテンシーを隠蔽

# プリフェッチャーとマルチコア



動的に共有される 2 つの L2 プリフェッチャー



# アドバンスド・スマート・キャッシュ マルチコアに最適化



## スマートキャッシュの利点

- L2 が各コアの負荷に適応できる
- 高速データ共有
- 複製データがない

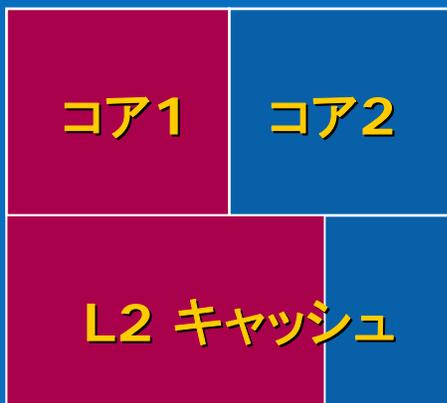
## 追加の利点

- L1 キャッシュに対する 2 倍の帯域幅

マルチコアに最適化された共有キャッシュ2 倍の帯域幅

# アドバンスト・スマート・キャッシュ ダイナミック・キャッシュ・アロケーション

## アドバンスト・ スマート・キャッシュ



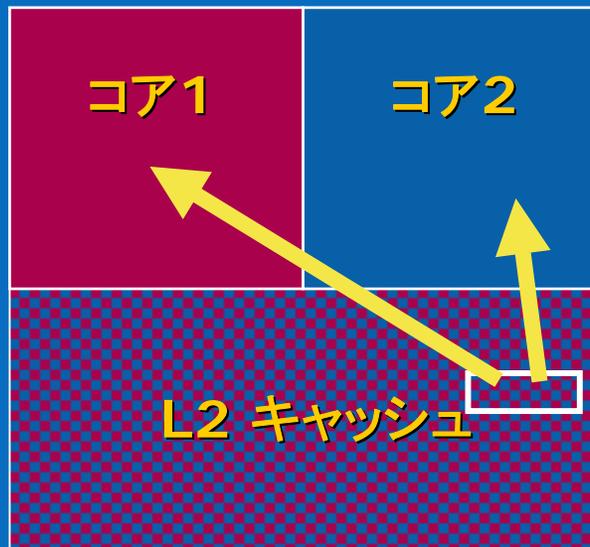
## 独立キャッシュ



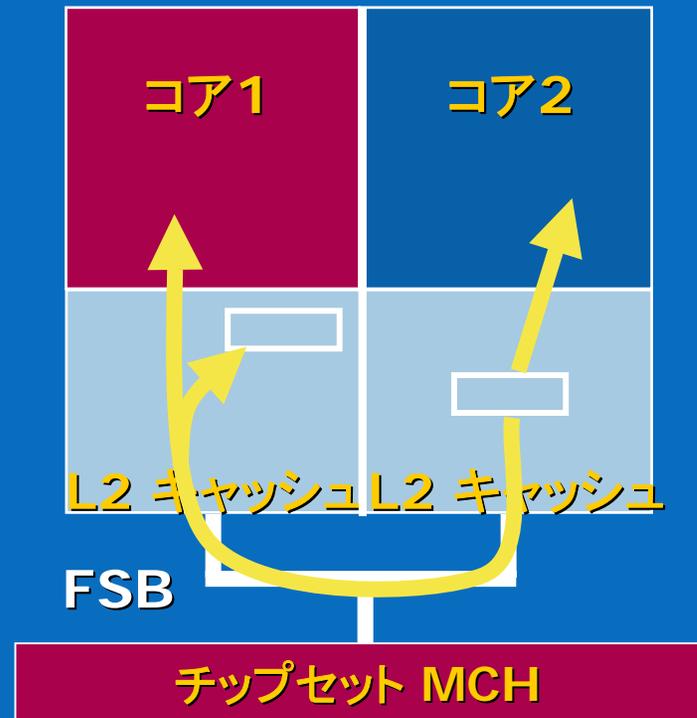
共有キャッシュは2つのコアからの不均衡な負荷に適応。しかし独立キャッシュは、一方のキャッシュの使用率が低く、キャッシュが空いていても、もう一方の高負荷のアプリケーションはその空きキャッシュを利用できずパフォーマンス向上が見込めない。

# アドバンスド・スマート・キャッシュ 効率的なデータ共有

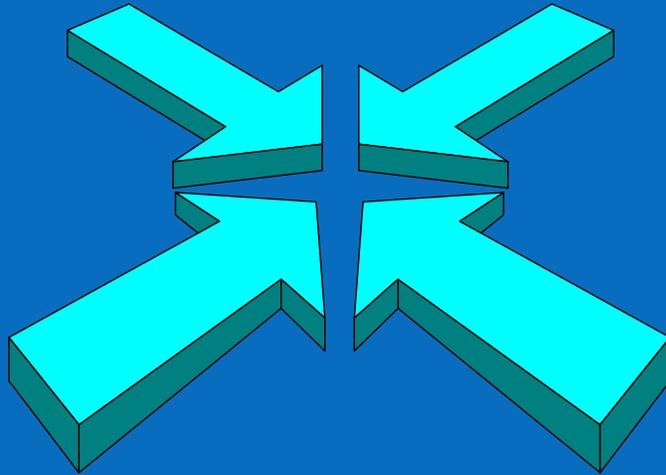
アドバンスド・  
スマート・キャッシュ



独立キャッシュ

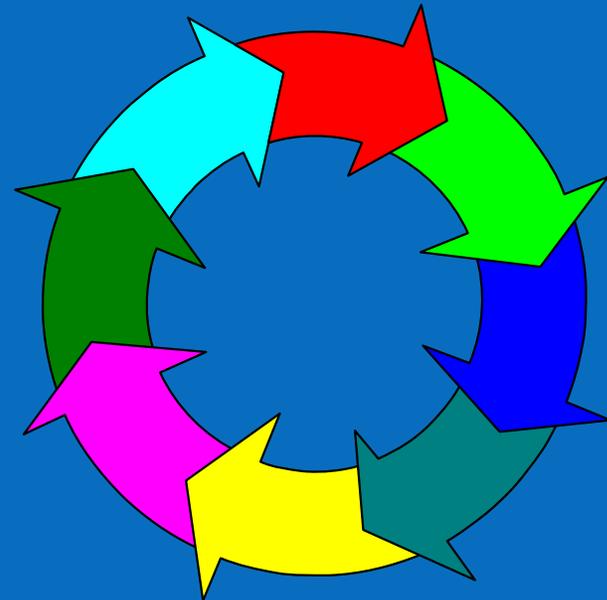


# 空間の局所性と時間の局所性



アクセスされたデータに隣接するデータは近い将来参照される可能性が高い

アクセスされたデータは近い将来再びアクセスされる可能性が高い



# キャッシュの構成

容量 (C)

ラインサイズ (B)

アソシアティビティ (A)

いくつかの不特定なメモリーの内容がキャッシュの特定のブロックに割り当てられる

$A = 1$

ダイレクトマッピング

$A = C / B$

フルアソシアティビティ

$1 < A < (C / B)$

セット アソシアティビティ

# メモリー中のスキップを避ける

メモリー・スキップするアクセスはパフォーマンスを低下させる

- IA-32 / インテル® 64 プロセッサのハードウェア・プリフェッチャーは、メモリー中のスキップを認識しない
  - TLB のエントリーには限りがある
  - 各アクセスにつき 1 つのキャッシュ・ラインが入出力される
    - キャッシュに未使用データが含まれる
    - 未使用データのためにより多くの帯域が必要
- ユニット・ストライドされないアクセスのため、効率的にベクトル化されない

# スキップの例

特定のスキップ

```
for (i = 0; i < MAX; i += 10) A[i];
```

外部次元中のループ

```
for (i = 0; i < MAX; i++) A[i][1];
```

ループ構造の 1 つの要素のみにアクセス

```
Struct Person[100] {  
    int ID;  
    char[50] Name;  
    char[100] address;  
}  
  
for (i = 0; i < 100; i++)  
    if (Person[i].ID==match) matchid = i;
```

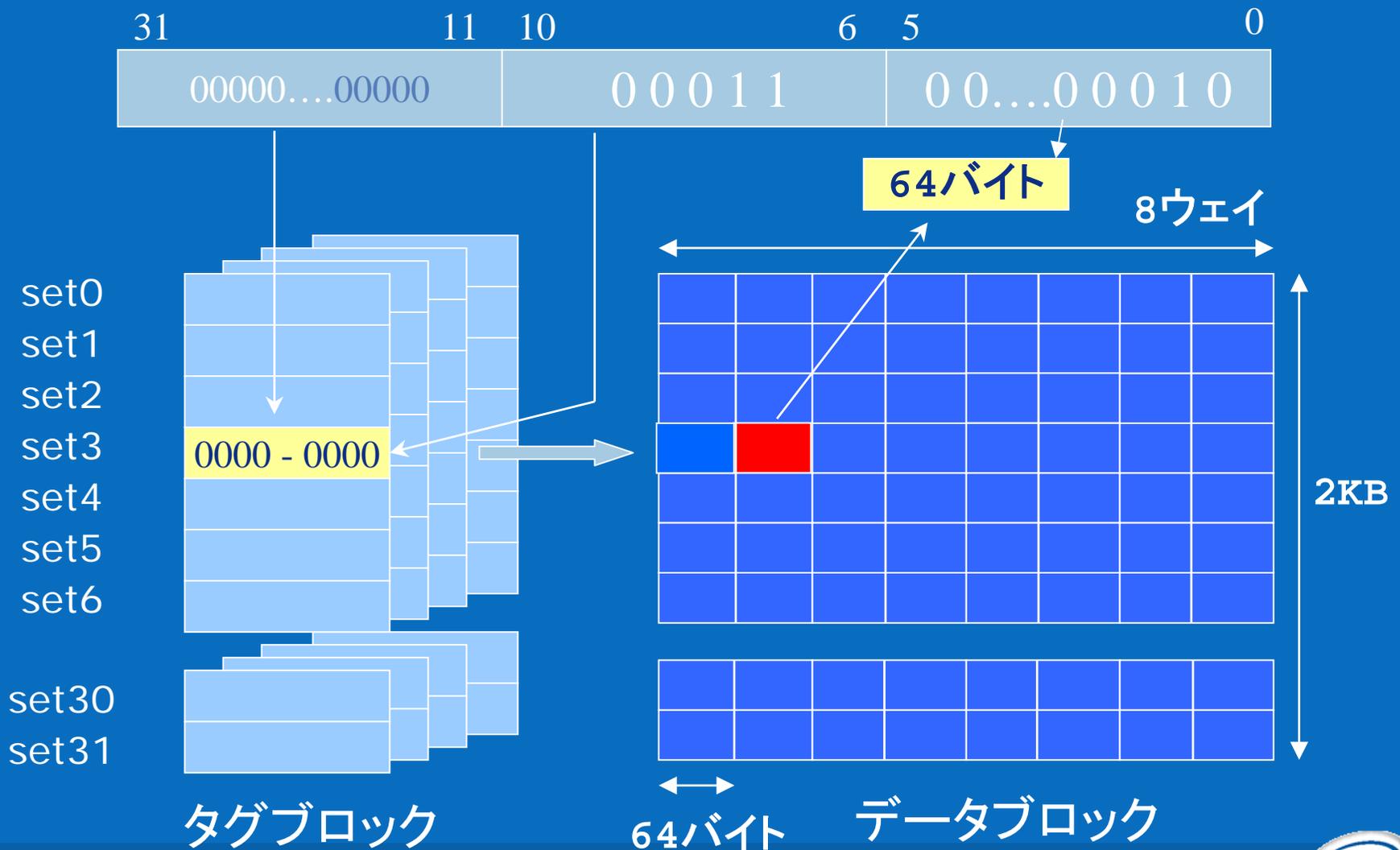
# 2<sup>n</sup> の法則

さまざまなアーキテクチャーの特性は 2<sup>n</sup> 離れたアドレスで明示される

- 大きな幅の 2<sup>n</sup> (キャッシュ・ラインよりも大きい場合) はキャッシュ・アクセスを招く
  - セット・アソシアティブ (連想方式) ・キャッシュ
  - キャッシュ管理
- 大きな幅の 2<sup>n</sup> のスキップは大きなパフォーマンス低下につながる!

2<sup>n</sup> の境界 (2K、4K、16K...256K) でメモリー・スキップを行うと、キャッシュ入れ替えの原因となる。正確なサイズはプロセッサとキャッシュ構成によって異なる

# 8ウェイ・セット・アソシアティブの場合

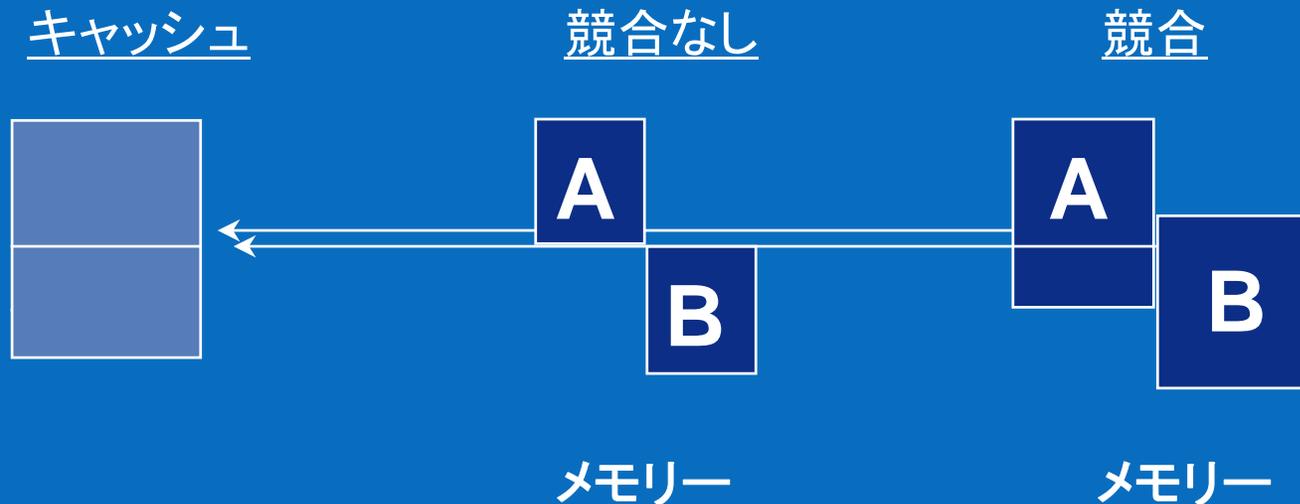


# キャッシュ・アクセス・ミスとは

初期参照ミス

容量ミス

競合ミス



# キャッシュ・アクセス・ミスを減らす最適化技術

配列のマージ

パディングとアライメント

パッキング

書き込み前の読み込み

ループインターチェンジ

ループフュージョン

ブロッキング

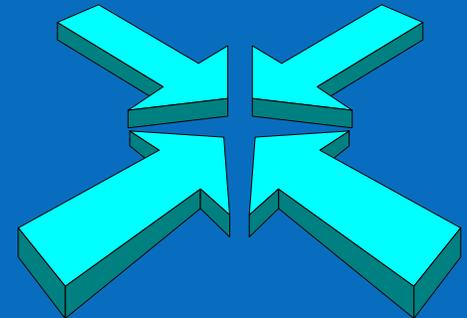
コードサイズの削除

```
struct {  
    int A[ ];  
    int B[ ];  
}
```

```
for(){  
  
}
```

# ローカリティを増やす

## コードとデータサイズを減らす



# SoA (配列構造体) と AoS (構造体配列)

Loop

s.a[i]

Loop

s.b[i]

```
struct {  
    int a[SIZE];  
    int b[SIZE];  
} s;
```

Loop

s[i].a

s[i].b

```
struct {  
    int a;  
    int b;  
} s[SIZE]
```

# SoA (配列構造体) と AoS (構造体配列)

```
struct {  
  int a[SIZE];  
  int b[SIZE];  
} s;
```



Loop

s[i].a

s[i].b

```
struct {  
  int a;  
  int b;  
} s[SIZE]
```



Loop

s.a[i]

Loop

s.b[i]

# パディング

```
struct {  
    int val[15];  
}
```

```
struct {  
    int val[15];  
    int pad;  
}
```

コンパイラによって自動的にアライメントされないデータ構造を考える

# 64 バイト (1 キャッシュライン)



```
struct {
    int val[15];
} s [SIZE]
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	0	1
2	3	4	5	6	7	8	9	10	11	12	13	14	0	1	2
3	4	5	6	7	8	9	10	11	12	13	14	0	1	2	3

```
struct {
    int val[15];
    int Pad;
} s [SIZE]
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	pad
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	pad
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	pad
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	pad

もしくは、

```
_declspec (align (16)) struct {
    int val[15];
} s [SIZE]
```



# 自然なアライメントと SIMD アライメント

C 言語の仕様では、構造体と配列内のデータは宣言したデータ要素が自然なサイズになるようにアライメントされる

- 他の方法 (パッキング) を指定した場合を除く
- 標準コードのメモリアクセスをスピードアップ

C 言語では、SSE のパフォーマンスを上げる SIMD アライメント (16 バイト) は強要しない

例:

- **float A[100]**
  - 16 バイトアドレスでアライメントする必要はない
  - 高速な SIMD movapd 命令は使用できない
  - インテル<sup>®</sup> コンパイラーは 16/32 バイト境界でアライメントしようとする



# コンパイラーの支援: データのアライメント(1)

```
declspec(align(16)) float B[MAX];
```

```
void funca(float * B) {  
    __assume_aligned(B,16);  
    A=_mm_malloc(sizeof(float) * MAX,16);  
  
    for (i = 0; i < MAX; i++)  
        A[i] += B[i];  
  
    _mm_free(A); }  
}
```

# コンパイラーの支援: データのアライメント(2)

```
_declspec(align(16)) float B[MAX];
```

```
void funca(float * B[]) {  
    A=_mm_malloc(sizeof(float)*MAX,16);
```

```
    #pragma vector aligned
```

```
    for ( i = 0 ; i < MAX; i++ )
```

```
        A[i] += B[i];
```

```
    _mm_free(A); }
```

# アライメント用のコンパイラー組み込み関数

## **\_\_declspec(align(base, [offset]))**

境界からのオフセット "offset" (バイト単位、デフォルトは 0) で、"base" バイト境界でアライメントされる変数を作成する

## **void\* \_mm\_malloc (int size, int n)**

"n" バイト境界でアライメントされるメモリーへのポインターを作成する

## **#pragma vector aligned | unaligned**

ベクトルアクセスにアライメントされている、またはアライメントされていないロードおよびストアを使用する

## **\_\_assume\_aligned(a,n)**

配列 "a" が "n" バイト境界でアライメントされると仮定する

# 構造体メンバーの再配置

```
struct unix_proc {  
    struct proc *next;  
    struct proc *back;  
    ...  
    int pid;  
    ...  
    int prio;  
    int nice;  
    ...  
    char uid;  
}
```



```
struct unix_proc {  
    struct proc *next;  
    int pid;  
    int prio;  
    struct proc *back;  
    ...  
    ...  
    int nice;  
    ...  
    char uid;  
}
```

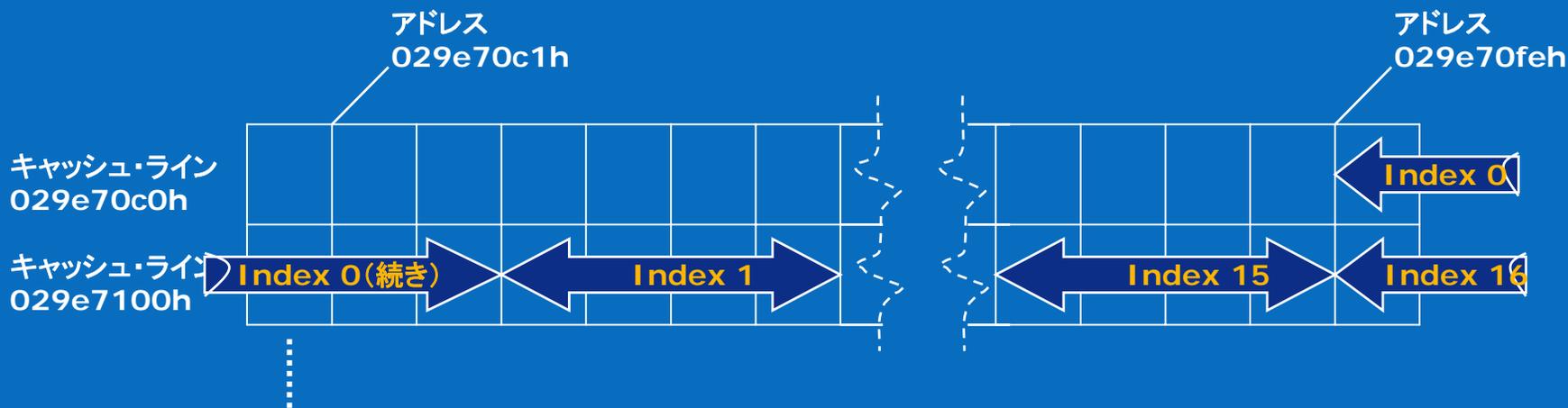
# キャッシュ・ラインの分割

定義: データ要素がキャッシュ境界をまたぐこと

影響: そのデータ要素にアクセスするには、1 回ではなく 2 回のメモリアクセスが必要になるため、パフォーマンスが低下する

インテル® Pentium® 4 プロセッサのキャッシュ・ラインは 64 バイト

SSE3 では、キャッシュ・ラインの分割問題を解消するために、新しい Iddqu 命令を追加



# ソフトウェア・プリフェッチの追加

## メカニズム

### `void _mm_prefetch(char const* a, int sel)`

(PREFETCH を使用)

指定されたアドレスからプロセッサに“近い” キャッシュ階層にデータのキャッシュ・ラインをロードする。値のセットは、プリフェッチ命令のタイプを指定する。プリフェッチ命令のタイプに合わせて、次の定数を選択する:

```
_MM_HINT_TO  
_MM_HINT_T1  
_MM_HINT_T2  
_MM_HINT_NTA
```

## 使用方法

- インテル® Pentium® 4 プロセッサのアーキテクチャーでは、CPU コアのリソースを要求しないでハードウェア・プリフェッチを行うことができるため、ソフトウェア・プリフェッチは通常、意味がない
- 細心の注意を払って試す。メモリーをランダムにアクセスするコードには有効



# ストリーミング・ストアー

```
#pragma vector nontemporal
```

```
for (i = 0; i < N; i++)
```

```
    a[i] = 1;
```

定義: キャッシュ中にデータを置かないストアーのこと

ループ回数が大きなループのパフォーマンスを著しく向上

注:  $a[i]$  は終了時にキャッシュ中に存在しないため、通常は  $a[i]$  に格納されたデータをすぐに使用しない

# データ・アクセス・パターンの理想

しきい値距離内のデータをアクセスすることで、ハードウェア・ストライド・プリフェッチャーは連続してアクセスできる

- L2 (L3) のヒットレートを向上させる

128 バイト以内に分散したデータを利用することで、隣接ライン・プリフェッチャーの恩恵を受けられる

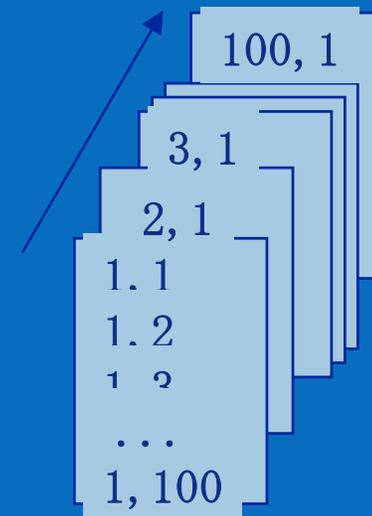
4K 以内にデータアクセスを集中することで、過剰なページングを防止する

# 実行コードによるキャッシュ・アクセス最適化

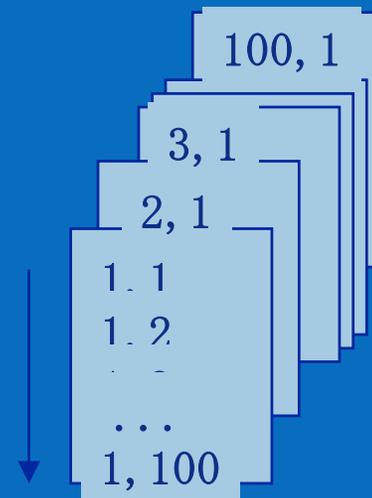


# ループインターチェンジ

```
for ( j = 0; j < 100; j++ )  
    for ( i = 0; i < 100; i++ )  
        a[i][j] = 2 * a[i][j];
```



```
for ( i = 0; i < 100; i++ )  
    for ( j = 0; j < 100; j++ )  
        a[i][j] = 2 * a[i][j];
```



# ループフュージョン

## ループフュージョン実行前;

```
for ( i = 0; i < N; i++ )  
    for ( j = 0; j < N; j++ )  
        a[i][j] = b[i][j] * c[i][j];  
for ( i = 0; i < N; i++ )  
    for ( j = 0; j < N; j++ )  
        d[i][j] = a[i][j] + c[i][j];
```

## ループフュージョン実行後;

```
for ( i = 0; i < N; i++ )  
    for ( j = 0; j < N; j++ ) {  
        a[i][j] = b[i][j] * c[i][j];  
        d[i][j] = a[i][j] + c[i][j];  
    }
```

# ループアンロール

```
for ( j = 0; j < M; j++)  
    for ( k = 0; k < N; k++ )  
        for ( i = 0; i < L; i++ )  
            C[k][i] += A[k][j] * B[j][i];
```

```
for ( j = 0; j < M; j += 4 )  
    for ( k = 0; k < N; k++ )  
        for ( i = 0; i < L; i++ )  
            C[k][i] += A[k][j] * B[j][i] +  
                A[k][j+1] * B[j+1][i] +  
                A[k][j+2] * B[j+2][i] +  
                A[k][j+3] * B[j+3][i];
```

# ストリップマイニング

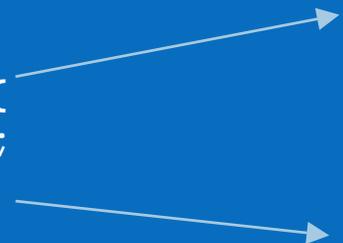
ストリップマイニング(ループセクション)はキャッシュを効率良く使用するためのループ変換手法であり、大きなループを小さなループに分割することで、データキャッシュの時間および空間の局所性を高める。

```
strip_mine(){
struct _vertex{ float x, y, z, nx, ny, nz, u, v;
} v[NUM];

for(i=0; i<NUM; i++){
    transform(&v[i]);
    Lighting(&v[i]);
}

}

for(i=0; i<NUM; i+=SM){
    for(j=i; j < min(NUM, i*SM); j++){
        transform(&v[j]);
        Lighting(&v[j]);
    }
}
```



# ブロッキング

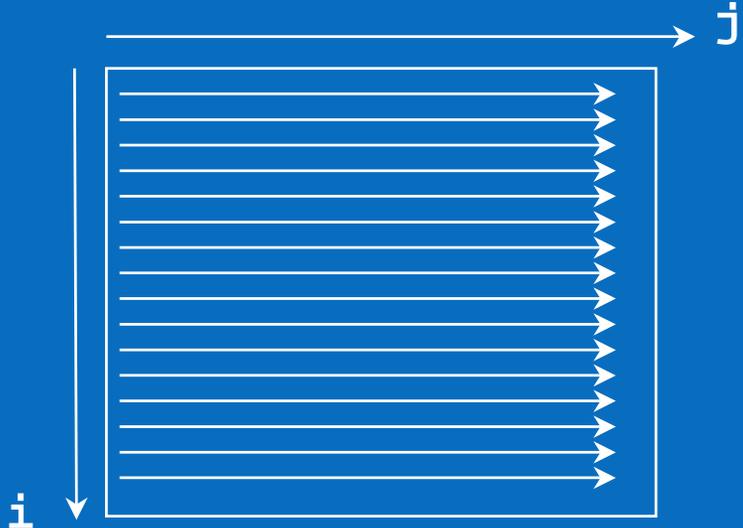
```
float A[MAX, MAX], B[MAX, MAX];  
  
for(i=0, i<MAX, i++){  
    for(j=0, j<MAX, j++){  
        A[i][j] = A[i][j] + B[j][i];  
    }  
}
```

配列AとBはシーケンシャルにアクセスされるが、配列Bはアクセスするたびにキャッシュミスが発生し、キャッシュの再利用がまったく考慮されていない。

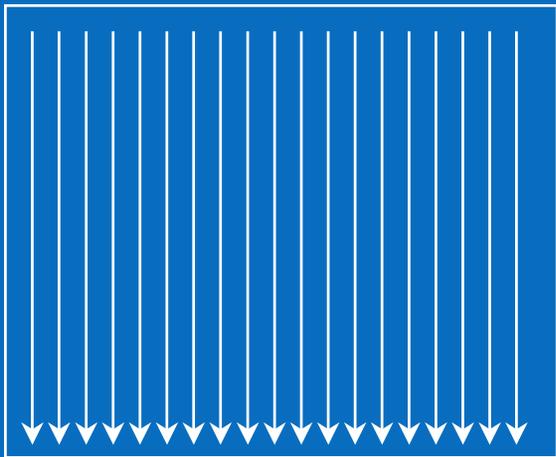
```
float A[MAX, MAX], B[MAX, MAX];  
  
for(i=0, i<MAX, i+=blocksize){  
    for(j=0; j<N; j+= blocksize){  
        for(ii=i; ii<i+blocksize; ii++){  
            for(jj=j; j<j+blocksize; jj++){  
                A[ii][jj] = A[ii][jj] + B[jj][ii];  
            }  
        }  
    }  
}
```

配列AとBをキャッシュサイズに収まるように断片的にアクセスすることにより、配列Bのキャッシュミスを減らす。Blocksizeを8にすると、各配列がブロッキングされる断片は8キャッシュラインとなる。

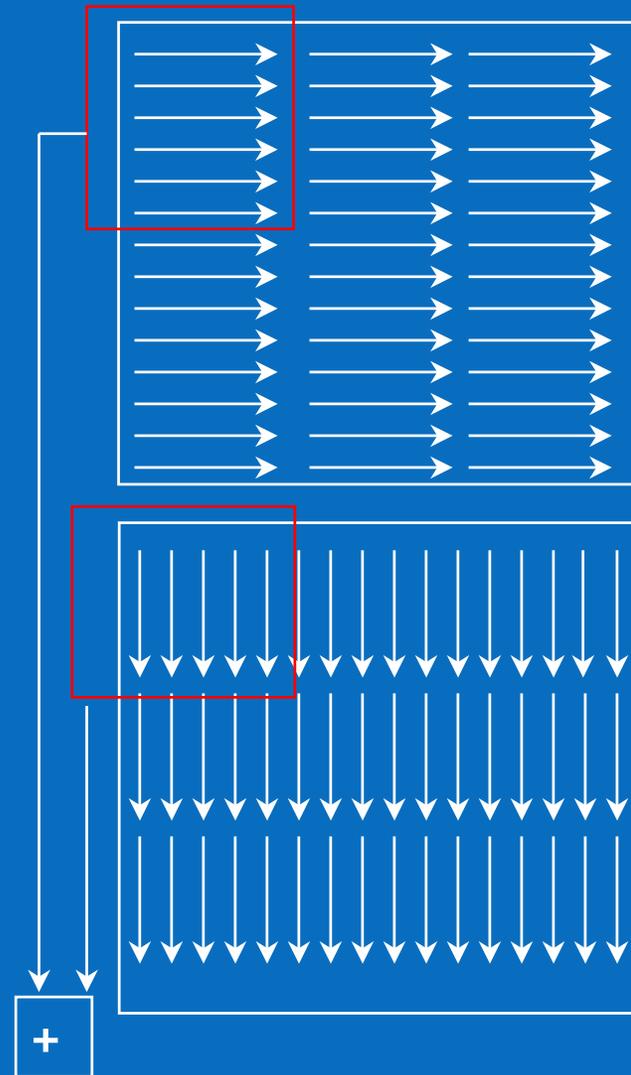
# A(i, j)のアクセスパターン



# B(i, j)のアクセスパターン



キャッシュサイズより小さい



# オルターネイトループ

```
void matrix(n){  
double a[100][100],b[100][100],C[100][100];  
int i,j,k;
```

```
    for(i=0; i < n; j++){  
        for(j=0; j < n; j++){  
            for(k=0; k < n; k++){  
                a[i][j] = a[i][j] + b[i][k] * c[k][j];  
            }  
        }  
    }  
}
```

ループの回数が不定であるため、ブロッキング等ができない

```
if(n > itval){
```

## ブロッキング化したコード

ループのコピーを作り、ループの回数がトランスフォーメーションを行なうのに十分な大きさがあれば、最適化したコードを実行する

```
    } else {  
        for(i=0; i < n; j++){  
            for(j=0; j < n; j++){  
                for(k=0; k < n; k++){  
                    a[i][j] = a[i][j] + b[i][k] * c[k][j];  
                }  
            }  
        }  
    }  
}
```

# コードサイズを減らす(アセンブリー)

複数サイクル命令を使用する

アドレス生成時にインデックス、オフセット

そしてスケールを使用する

短いオペコードを使用する

- $\pm 128$ 以下の即値
- `mov eax, 0` の代わりに  
`xor eax, eax` を使用
- レジスターはなるべく `eax` を使用

# まとめ

コード/データアクセスのローカリティ向上

コード/データサイズを減らす

キャッシュの性能を監視する:

- キャッシュの使用率のプロファイル
- ホットスポットの検索
- アクセスミスタイプ割り出し
- 適切な方法で最適化



# 参考資料



# プリフェッチ命令

プリフェッチ命令はデータの参照に先行してデータをフェッチすることにより、アプリケーション・コードのパフォーマンス・クリティカルな部分でのメモリアクセス遅延を隠すことができる。

## 非一時命令

prefetchnta

L0へ読み込み(L1)

## 一時命令

prefetcht0

すべてのレベルのキャッシュへ読み込み

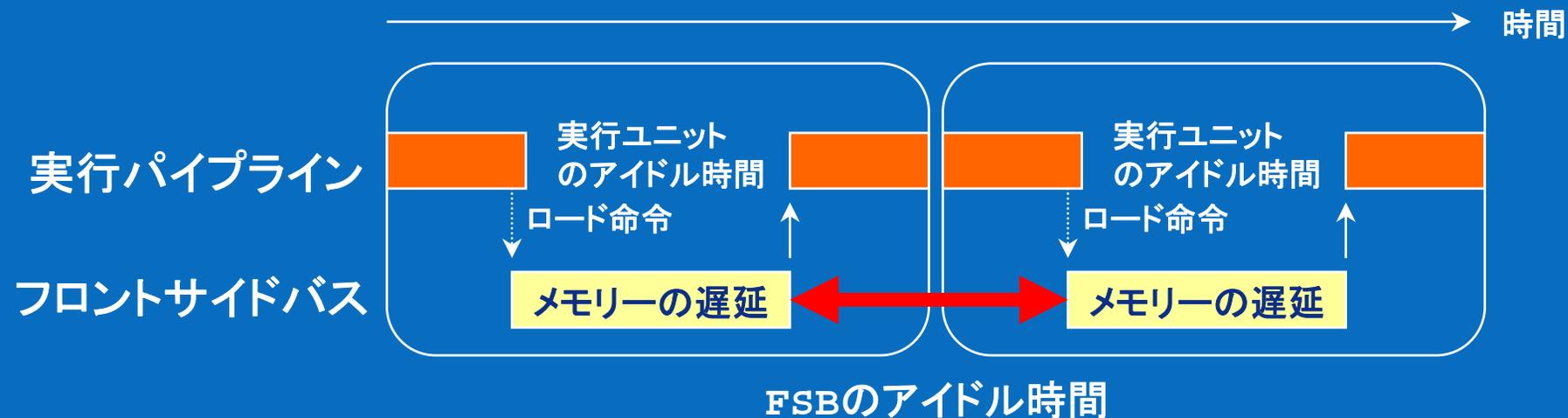
prefetcht1

L0を除くキャッシュへ読み込み(L2)

prefetcht2

L0, L1を除くキャッシュ(L2)へ読み込み

# メモリーアクセスによる遅延とプリフェッチ

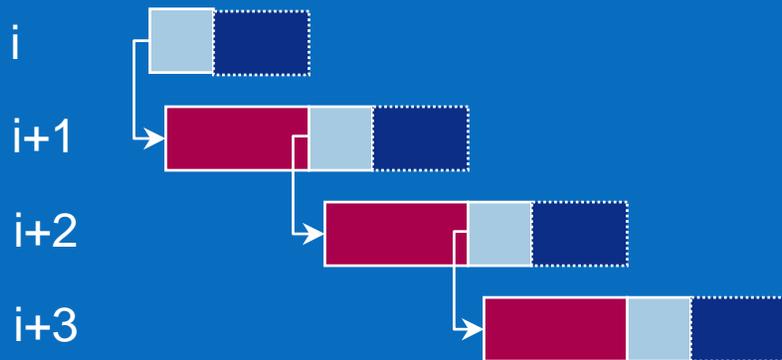


# プリフェッチ命令による メモリアクセスの最適化

- プリフェッチを実行する間隔
- プリフェッチの連結
- プリフェッチの最小化
- プリフェッチの分散
- キャッシュのブロッキング
- メモリーバンクのアクセス競合
- キャッシュ管理

# プリフェッチを実行する間隔

プリフェッチ命令を有効に使用するには、適切な実行間隔が設定されなければならない。間隔が狭すぎると、プリフェッチの時間を他の演算実行サイクルで隠すことができず、間隔が広すぎるとプリフェッチしたデータがキャッシュに無い可能性がある。



小さくてタイトなループはプリフェッチの恩恵を受けそうに見えるが、実際にはそうではない。

# プリフェッチ間隔 (PSD)

$$\text{Psd} = \frac{\text{Nlookup} + \text{Nxfer} \cdot (\text{N}_{\text{pref}} + \text{N}_{\text{st}})}{\text{CPI} \cdot \text{N}_{\text{inst}}}$$

Psd	プリフェッチのスケジュール間隔
Nlookup	ルックアップレイテンシークロック。 メモリーやチップセットに依存。
Nxfer	キャッシュラインの転送に要するクロック。
NprefとNst	プリフェッチとストアされるキャッシュライン数
Ninst	1回のループの命令数

$$\text{Psd} = \frac{60 + 25 \cdot (\text{N}_{\text{pref}} + \text{N}_{\text{st}})}{1.5 \cdot \text{N}_{\text{inst}}}$$



# プリフェッチの連結

ネストしたループでは、内側のループが終了し、外側のループ処理が始まるまでにパイプラインが途切れる可能性がある。

```
for (ii=0; ii=100; ii++){  
    for(jj=0; jj<32; jj+=8){  
        prefetch a[ii][jj+8];  
        computation a[ii][jj];  
    }  
}
```

この例では、`a[ii][0]`を含むキャッシュラインがプリフェッチされておらず、内側のループの最後で不要なプリフェッチが行われる。

# プリフェッチの連結(2)

プリフェッチの連結は、内側と外側の実行パイプラインを繋ぐブリッジの役目をはたす。内側のループの繰り返しを、ループ外で次のループで使用するデータのプリフェッチと共に行うことで、メモリーパイプラインの途切れによる性能低下を回避する

```
for (ii=0; ii<100; ii++){
    for(jj=0; jj<24; jj+=8){
        prefetch a[ii][jj+8];
        computation a[ii][jj];
    }
    prefetch a[ii+1][0];
    computation a[ii][jj];
}
```

# プリフェッチの最小化

過度なプリフェッチは次のような問題を招く。

- フィル・バッファに空きが無い場合、フィル・バッファ・エントリの割り当て待ちでプリフェッチがロードバッファ内に蓄積する。
- ロードバッファに空きが無いと、命令の割り当てがストールする
- 対象ループが小さな場合、プリフェッチによってオーバーヘッドが増大する。

```
Loop:
prefetchnta [edx+esi+32]
prefetchnta [edx*4+esi+32]
...
movaps    xmm1,[edx+esi]
movaps    xmm2,[edx*4+esi]
...
add       esi,16
cmp       esi,ecx
jl        Loop
```

```
Loop:
prefetchnta [edx+esi+32]
prefetchnta [edx*4+esi+32]
...
movaps    xmm1,[edx+esi]
movaps    xmm2,[edx*4+esi]
...
movaps    xmm1,[edx+esi+16]
movaps    xmm2,[edx*4+esi+16]
...
add       esi,32
cmp       esi,ecx
jl        Loop
```

# プリフェッチの分散

すべてのプリフェッチをループのはじめに実行すると、大幅な性能低下につながる可能性がある。

プリフェッチは他の演算と交互に配置する。

PentiumIIIプロセッサ(500MHz)では、20-25サイクル毎にプリフェッチ命令を挿入する。

Loop:

```
prefetchnta [ebx+128]
prefetchnta [ebx+1128]
prefetchnta [ebx+2128]
prefetchnta [ebx+3128]
...
prefetchnta [ebx+17128]
prefetchnta [ebx+18128]
prefetchnta [ebx+19128]
prefetchnta [ebx+20128]
...
mulps      xmm3, [ebx+4000]
addps      xmm1, [ebx+1000]
addps      xmm2, [ebx+3016]
mulps      xmm3, [ebx+2000]
mulps      xmm1, xmm2
...
...
add        ebx, 32
cmp        ebx, ecx
jl         Loop
```

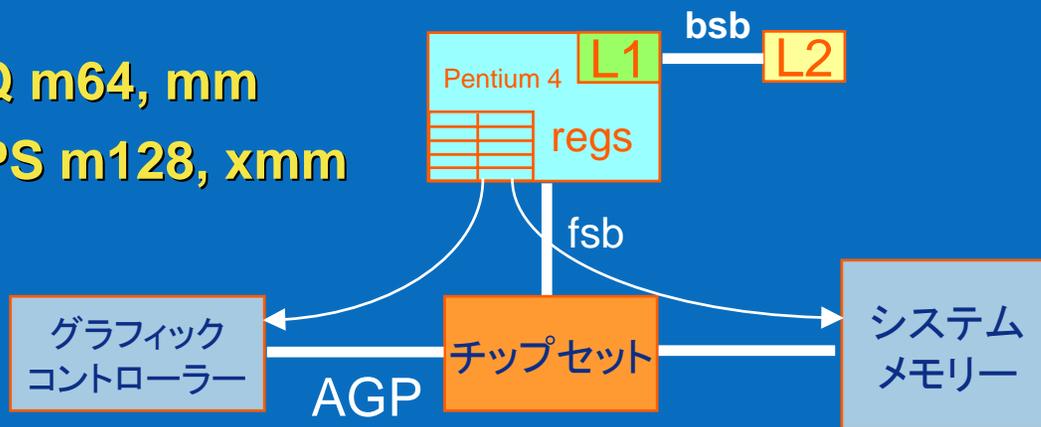
# ビデオデコーダにおけるキャッシュ管理

ここでのビデオデコーダでは、処理済のフレームデータをビデオメモリへ書き込み、その後データのコピーがプロセッサによってライトバックメモリに格納され、以降のデータ生成に利用されることを前提とする。

データはストリーミング・ストア命令でビデオメモリへ直接書き込むことにより、プロセッサキャッシュの汚染を防止する。その後プロセッサは、prefetchnta再読み込みされるため、使用可能なバンド幅は最大となる。NTA(非一時)読み込みを行うことで、キャッシュ内の他のデータへの影響を最小限に押さえる。

**MOVNTQ m64, mm**

**MOVNTPS m128, xmm**



本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。製品に付属の売買契約書『Intel's Terms and conditions of Sales』に規定されている場合を除き、インテルはいかなる責を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証 (特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他、知的所有権を侵害していないことへの保証を含む) に関しても一切責任を負わないものとします。

インテル製品は、予告なく仕様が変更されることがあります。

\* その他の社名、製品名などは、一般に各社の商標または登録商標です。

© 2007, Intel Corporation.

