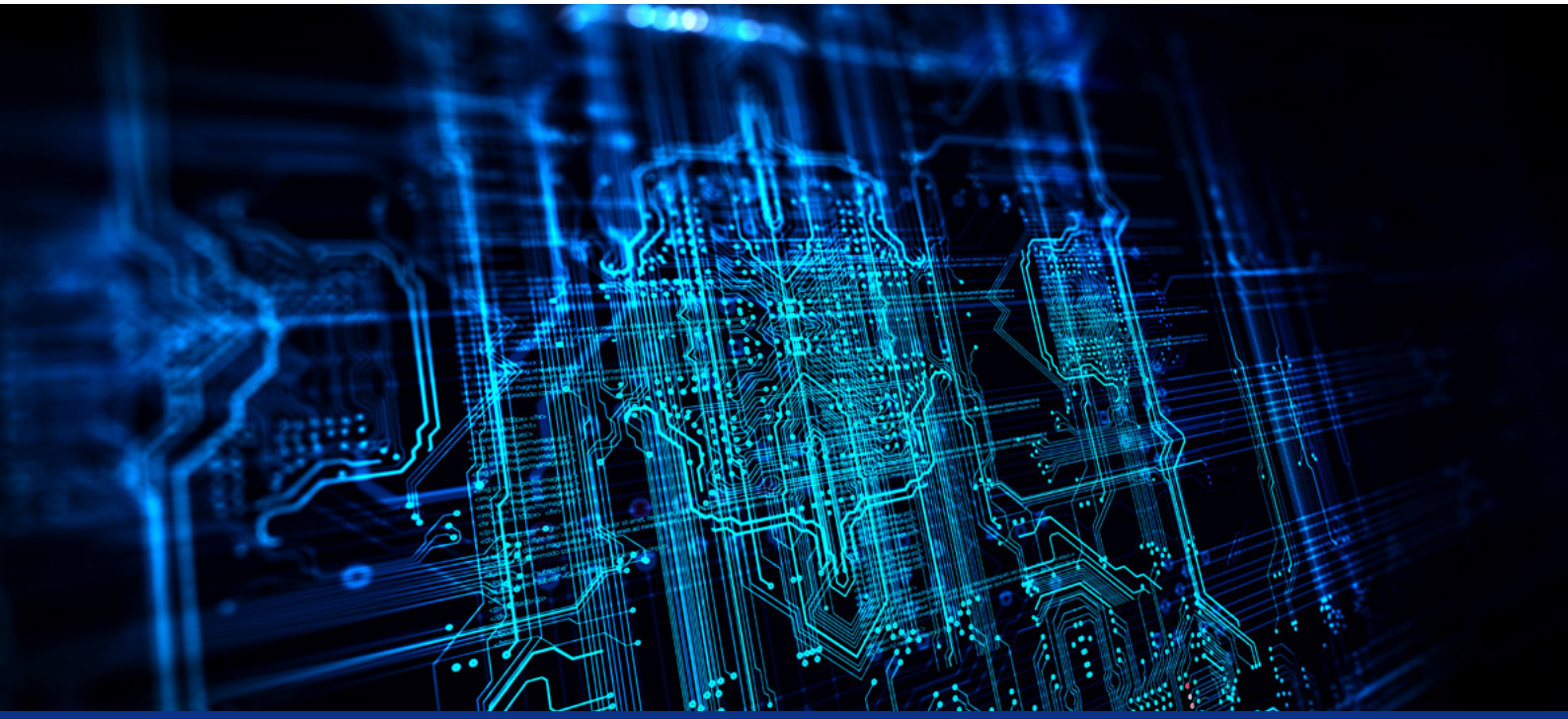


マルチプロセッサーでフォルス・シェアリングを検出し軽減する

マルチスレッド・アプリケーションのパフォーマンスを大幅に向上

Ramesh Peri インテル・コーポレーション シニア主席エンジニア

現在、コアごとに複数のローカルキャッシュ階層を持つ、マルチコア、シングルノードのシステムが一般的になっています。平均的なラップトップには少なくとも 2 つ以上のコアが備わっており、サーバーシステムでは数十のコアに至ります。ソフトウェアは、マルチスレッドによってこれらのコアを利用します。スレッドは共通アドレス空間の共有データ構造で動作するため、異なるコアのローカルキャッシュに存在するデータのコピーを互いに調整する必要があります。例えば、2 つのスレッドが同じメモリー位置にアクセスする場合、メモリー・サブシステムはスレッドがローカルキャッシュに存在するデータのコピーを更新する際に、検証して更新する必要があります (これは、マルチスレッド・プログラムが共有メモリーのアトミック更新を保証するために使用する同期メカニズムとは関連性がありません)。メモリー・サブシステムは、複雑なキャッシュ一貫性プロトコルを実装

することで、この一貫性を保証します。これは、複数のスレッドが共有メモリーにアクセスして更新を行う際に、パフォーマンスに影響します。

この記事では、キャッシュの一貫性プロトコルと、それらがマルチスレッド・プログラムのパフォーマンスに与える影響について説明します。

キャッシュとキャッシュの一貫性プロトコル

キャッシュは、メインメモリーの頻繁にアクセスされる場所のコピーを保持するため、CPU が備える少容量ですが高速なメモリーの一種です。マルチコアシステムでは、すべての CPU にメインメモリーのコピーを保持する個別のキャッシュが備わっています。プログラムの正当性を維持するため、CPU は実行中にこれらのキャッシュの内容を相互に同期および更新します。データは通常、64 バイトまたは 128 バイトのブロック単位でメモリーからキャッシュに転送されます。CPU がメモリーを読み書きする場合、最初にキャッシュラインのいずれかにそのアドレスが存在するか確認します。存在する場合、CPU はそのキャッシュラインをフェッチします。存在しない場合は、以降に使用することを想定し新しいキャッシュラインのエントリーが割り当てられます。

マルチコアマシンでは、CPU がメモリーからデータを取り込むかどうか決定する前に、ローカルキャッシュだけでなく他のコアのキャッシュをチェックする必要があります。すべてのコアのすべてのキャッシュにわたってメモリーアクセスごとに生じるチェックの回数を減らすため、すべてのキャッシュラインには特定の情報が保持されています。この情報は MESI (変更 (Modified)/排他 (Exclusive)/共有 (Shared)/無効 (Invalid)) として知られています。

キャッシュラインが最初にロードされると、CPU はそのキャッシュラインを「排他 (Exclusive)」アクセスとしてマークします。排他モードである限り、以降のロードはすべてこのデータにアクセスします。同じキャッシュラインが別の CPU によってロードされると、キャッシュラインはすべての CPU のキャッシュで「共有 (Shared)」としてマークされます。CPU が「共有 (Shared)」としてマークされたキャッシュラインに書き込みを行うと、キャッシュラインは「変更 (Modified)」としてマークされ、他のすべての CPU に「無効 (Invalid)」メッセージが送られます。「変更 (Modified)」としてマークされたキャッシュラインに別の CPU がアクセスすると、キャッシュラインはメモリーにストアされ「共有 (Shared)」としてマークされます。同じキャッシュラインにアクセスする他の CPU はキャッシュミスを引き起こし、メインメモリーから更新されたコピーをフェッチします。

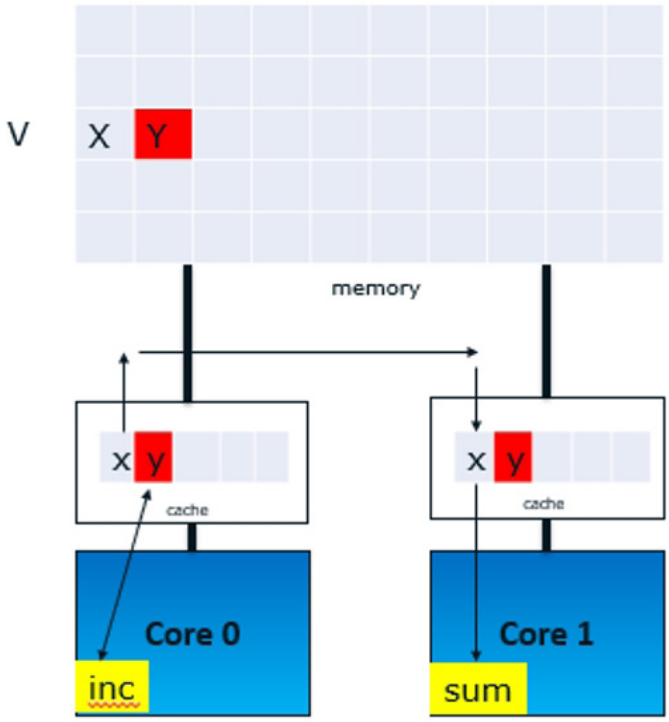
別の CPU で「変更 (Modified)」状態のキャッシュであるメモリー位置に CPU がアクセスすると、以下が発生することが分かります。

- メモリーへの書き込み
- メモリーからの読み取り

異なるコアで実行される複数のスレッドがアクセスするメモリー位置では、これは最新の値を提供し、メモリーモデルの一貫性を保証するために必要です。

効率のため、キャッシュの一貫性プロトコルは、個別のバイト単位ではなくキャッシュライン単位で動作します。これにより、フォルス・シェアリング (偽共有) と呼ばれる現象が発生します (**図 1**)。フォルス・シェアリングでは、

```
struct {
  int x;
  int y;
} v;
/* sum & inc を並列に実行 */
int sum(void)
{
  int i, s = 0;
  for (i = 0; i < 1000000; ++i}
    s+=v.x;
  return s;
}
void inc(void)
{
  int i;
  for (i=0; < 1000000; ++i)
  v.y++;
}
```



1 マルチコアシステムのフォルス・シェアリング

同じキャッシュラインに存在する異なるメモリー位置にアクセスするスレッドにもキャッシュ一貫性プロトコルのメカニズムが適用されるため、パフォーマンスが低下します。場合によっては、このパフォーマンス・ペナルティーは大きくなる可能性があります。

図 1 では、コア 0 とコア 1 で実行されるスレッド **sum** と **inc** は、同じキャッシュラインにある異なる 2 つのメモリー位置にアクセスします。キャッシュ一貫性プロトコルは、キャッシュライン単位で動作するため **inc** スレッドが **y** にアクセスした後に、**sum** スレッドが **x** にアクセスするたびに、2 つのスレッドが同じ変数にアクセスしているかのようにパフォーマンス・ペナルティーが生じます。

キャッシュ間の共有を検出

インテル® プロセッサーのパフォーマンス監視ユニット (PMU) には、プログラムの実行中にマイクロアーキテクチャーで発生する各種イベントを監視する機能があります。システム上の他の CPU キャッシュの「変更 (modified)」、「排他 (exclusive)」、または「共有 (shared)」状態のメモリー位置の参照に関連するいくつかのイベントがあります。これらのイベントは **'HIT<M/E/S>'** という文字列が示されます。このイベントを監視することで、コア間のキャッシュライン相互参照があるか確認できます。例えば、開発コード名 **Goldmont** (英語) マイクロアーキテクチャー・ベースのプラットフォームでは、**MEM_UOPS_RETIRED.HITM** イベントを監視することで、他の CPU キャッシュの変更状態のキャッシュラインへの参照数を知ることができます。また、開発コード名 **Skylake** マイクロアーキテクチャー・ベースのプロセッサーでは、いくつかのイベントを調べそれらに数式を適用することで、フォルス・シェアリングを監視できます (**インテル® VTune™ Amplifier** は、必要なすべてのメトリックを自動的に計算します)。

フォルス・シェアリングを示すマイクロベンチマーク

図 2 のコードは、指定する数のスレッドを生成し、スレッドは指定されたパターンで配列にアクセスします。このコードは、引数としてスレッド数とスレッドが配列にアクセスする開始インデックスを受けとります。引数 **4 0 1 2 3** を指定してコードを実行すると、**図 2** に示すアクセスパターンとなります。ここでは、4 つのスレッドがすべて同じ 64 バイトのキャッシュラインにある異なる配列要素を操作していることが分かります (**図 3**)。このアクセスパターンにより、プロセッサーはこれらのスレッドを実行する 4 つのコアのローカルキャッシュ間でキャッシュラインを " 相互に返し "、並列パフォーマンスを制限する可能性があります。

引数 **4 0 64 128 192** でコードを実行すると、アクセスパターンは**図 4** のようになります。各行は異なるキャッシュラインをアクセスします。すべてのスレッドが、異なるキャッシュラインにある配列要素を操作していることが分かります。

両方のインスタンス (**図 3** と **4**) では、リタイアした命令数、およびメモリーのロードとストアは、すべてのスレッドでまったく同一であることに注意することが重要です。また、これらのスレッドはすべて、常に配列の異なる要素をアクセスしています。そのため、プログラムの 2 つの実行におけるパフォーマンスの違いは、キャッシュ・サブシステムの影響であると結論付けることができます。

フォルス・シェアリング・マイクロベンチマークのパフォーマンスを評価

マイクロベンチマークは、フォルス・シェアリングがある場合とない場合で、インテル® VTune™ Amplifier パフォーマンス解析ツールで実行されました。**図 5** はフォルス・シェアリングがある状況で、**図 6** はない状況を示しています。どちらの場合も、実行の作業量は同じであることが分かります (両方の実行で同じ命令数 (およそ 50 億) が実行されています)。しかし、実行のクロックサイクル数は異なります。フォルス・シェアリングがある場合は約 124 億サイクルで、ない場合は約 35 億サイクルであり、およそ 3 倍の差があります。スレッドは、キャッシュラインのフォルス・シェアリングが原因で他のスレッドのストアが完了するのを待機するため、インテル® VTune™ Amplifier はこれを 100% ストア依存として検出およびレポートすることに注意してください。

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

// shared array accessed by all threads
int a[256] __attribute__ ((aligned(64)));

/* this function is run by the second thread */
void *thread_func(void *vptr)
{
    int indx = *((int *)vptr);
    for (int j=0;j<1000;j++)
       for (int k=0;k<10000;k++)
          for (int i=indx;i<(indx+32);i+=4)
             a[i]=a[i]+1;
    return NULL;
}

void usage(char *name)
{
    printf("USAGE: %s <n> <offset-1> ... <offset-n>\n",name);
    exit(0);
}

int main(int argc, char **argv)
{
    int num_threads;
    pthread_t *threads;
    int *access_pattern;
    if (argc < 2) usage(argv[0]);
    num_threads = atoi(argv[1]);
    if (argc != (num_threads+2)) usage(argv[0]);
    // allocate space for threadids and the access pattern
    access_pattern = (int *)calloc(num_threads,sizeof(int));
    threads = (pthread_t *)calloc(num_threads, sizeof(pthread_t));
    for (int i=0;i<num_threads;i++)
       access_pattern[i]=atoi(argv[i+2]);
    printf("Running %d threads with access pattern: ",num_threads);
    for (int i=0;i<num_threads;i++)
       printf("%d ", access_pattern[i]);
    printf("\n");
    // create threads
    for (int i=0;i<num_threads;i++)
       if (pthread_create(&threads[i], NULL, thread_func, &access_pattern[i])) {
          fprintf(stderr, "Error creating thread %d\n",i);
          return(1);
       }
    // wait for all threads to finish
    for (int i=0;i<num_threads;i++)
       if (pthread_join(threads[i], NULL)) {
          fprintf(stderr, "Error joining thread %d\n",i);
          return(2);
       }
    return 0;
}
```

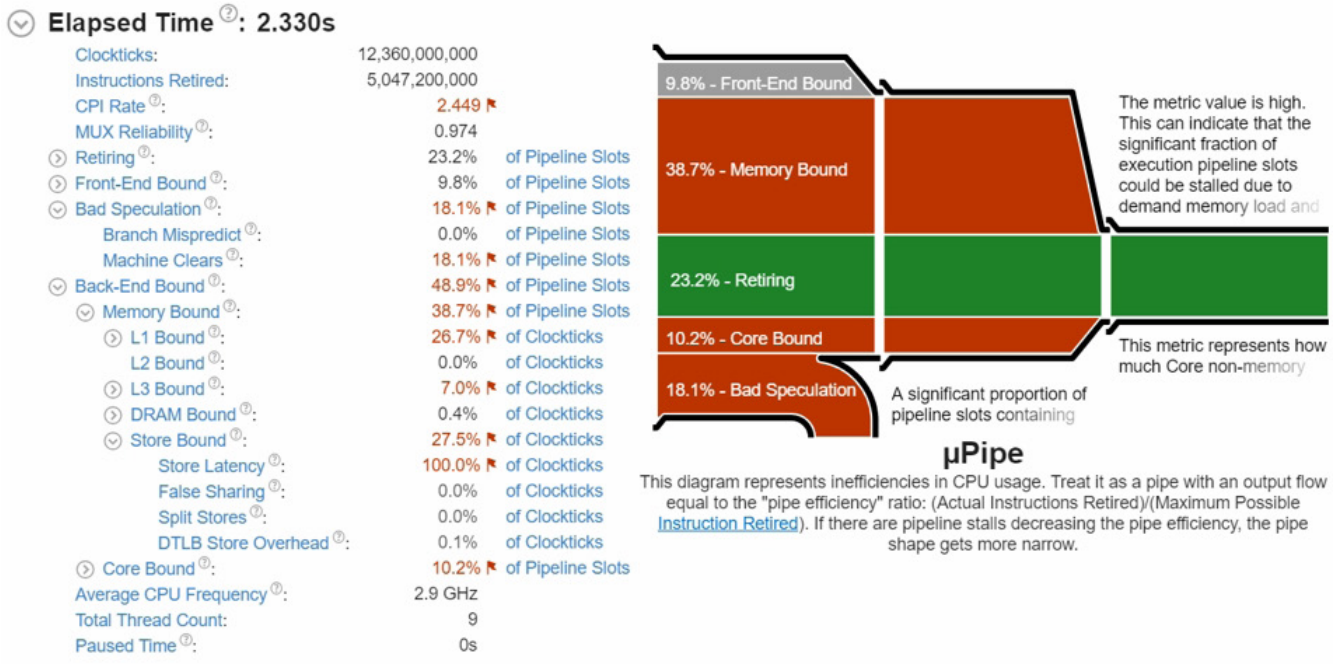
2 **さまざまなメモリー・アクセス・パターンを行うマイクロベンチマーク**

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	....
																...
																...
																...

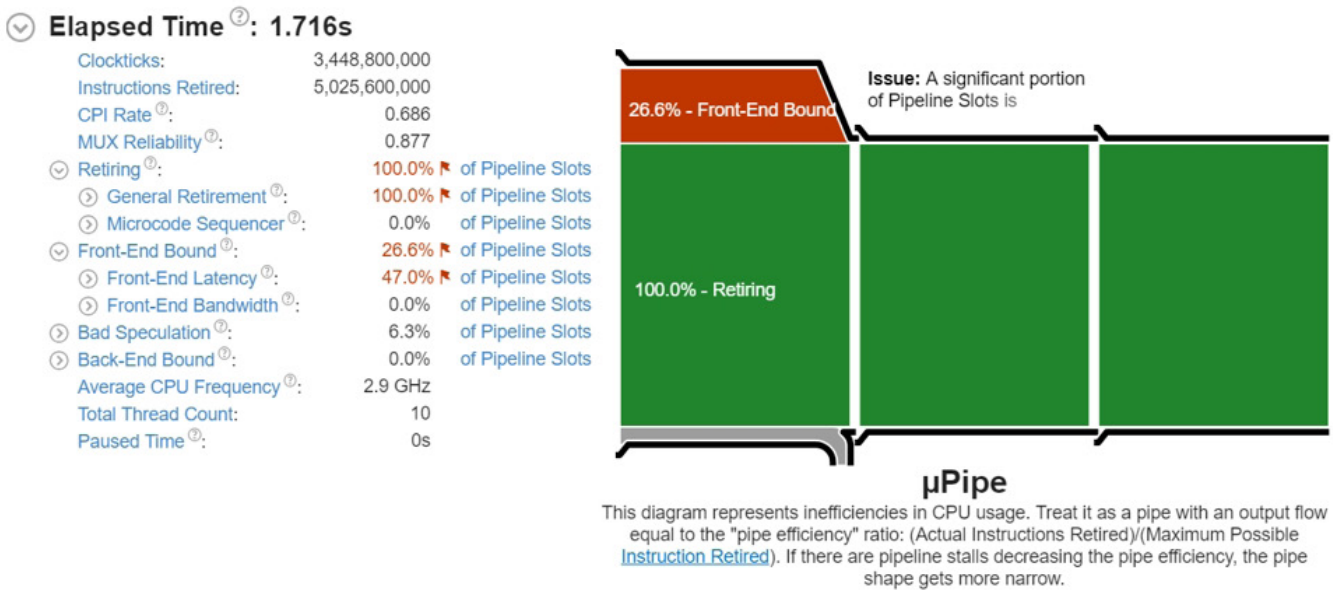
3 **図 2 のコードを引数 `4 0 1 2 3` で実行した際のメモリー・アクセス・パターン。64 バイトのキャッシュラインでフォルス・シェアリングが発生します。**

1				1				1				1				1				...
	2				2				2				2				2			...
		3				3				3				3				3		...
			4				4				4				4				4	...

4 **図 2 のコードを引数 `4 0 64 128 192` で実行した際のメモリー・アクセス・パターン。64 バイトのキャッシュラインでフォルス・シェアリングは発生しません。**



5 **フォルス・シェアリングを示すメモリー・アクセス・パターン (図 3) についてのインテル® VTune™ Amplifier のサマリー・ビュー。**



6 **フォルス・シェアリングがないことを示すメモリー・アクセス・パターン (図 4) についてのインテル® VTune™ Amplifier のサマリービュー。**

フォルス・シェアリングを排除する

フォルス・シェアリングは、マルチスレッド・アプリケーションのパフォーマンスを大幅に制限する可能性があり、コンパイラーまたはユーザーが共有データ構造を配置する方法により発生することがあります。フォルス・シェアリングにより、マイクロベンチマークのパフォーマンスは 1/3 に低下する可能性があります。インテル® VTune™ Amplifier などのプロファイル・ツールには、フォルス・シェアリングを検出して開発者に警告する機能が備わっています。

参考資料

1. **MESI プロトコル (英語)**
2. **インテル® VTune™ Amplifier**
3. **インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル (英語)**

インテル® VTune™ Amplifier
現代のプロセッサーのパフォーマンス解析
無料評価版のダウンロード