キャッシュ性能安定性について

電気通信大学　今村 俊幸

1　準備

本記事の読者は現在殆どのマイクロプロセッサでほぼ同一の方式のキャッシュが搭載されていることをご存知であろうか? 「ハードウェア」や「プロセッサ」と名の付いた多くの書籍に詳細は書かれているので, 本原稿では最小必要限の内容を準備としてから解説を進めていこう.

図 1 は, 一般的なキャッシュの構造を模式化したものである. テキストによっては, 記法が異なる場合があるが, 本記事ではこの方式で説明を進めたい. 多くのプロセッサで採用されているキャッシュの構造は **n-way セットアソシアティブ (n 群連想記憶方式)** キャッシュと呼ばれる. 図 1 は, 4-way 構成した場合のキャッシュの概念図になり, n-way 構成の場合キャッシュ全体は n 個の**バンク (way)** に分けられそれぞれのバンクでは更に**ライン**と呼ばれる単位に分割される. 各ラインには**データ**と**タグ**が対応し, データ部分にはプロセッサがアクセスする主記憶データのコピーが格納される. 更に, タグ部分にはラインの仮想記憶上でのアドレス[1] の上位ビットを記憶する. そして, 列位置 (図では set x と表記) を**インデックス** (x) と呼び, 同一インデックスの n 個のラインの集合を**セット**と呼ぶ.

一般に, ライン数やラインサイズは **2 のベキ**であり, 本稿ではそれぞれ 2^s, 2^l と表記する (つまりライン数, ラインサイズはそれぞれ s, l ビット幅で表現される). 仮想記憶上のアドレスとセットとの対応は, アドレスの下位 $s+l$ ビットのうち上位の s ビット値のセットで一意に決定される. ただし, このままでは s ビットが一致するアドレスは無数に存在するため, 複数の格納場所を設けなくてはならない. これが, バンクであり最大で n 個の s ビットが一致するアドレスをキャッシュ上に格納することができる. 逆に言えば n 個しか保持できない強い制約が存在するともいえよう.

アドレスの下位 $s+l$ ビットを落とした値をタグ部分に格納した way が存在すれば, キャッシュ上の (set,way) で定まる場所に指定アドレスを含むデータが格納されていることになる. 一方, いずれの way にもタグ部分に対応するアドレス情報がない場合は, キャッシュミスの状態にあると云う. どの way を選択するかは, 通常 **LRU(Least Recently Used, 最長未使用時間)** 方式で行われる. 「最近もっとも使われていない」way が選択され主記憶上のキャッシュラインが格納されることになる. 実際利用者がどの way を利用するかは制御できないし, 特に注意を払う必要もない (但し, これ以降の議論では制御できると便利な面もあるのだが...).

[1] マイクロプロセッサによって物理記憶上のアドレスとなる場合があるが, ここでは仮想記憶上のアドレスを前提とする.

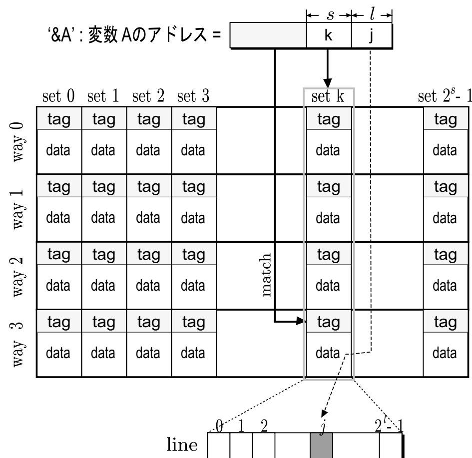

図 1: 4-way セットアソシアティブキャッシュの概念図

次節以降の説明を簡略化するために次のような表記方法を導入する. もし, データ A がキャッシュ上の set x, way y に対応するとき, ただし way y は LRU などによって適時予想がつく際に y を決めることとし, 定まらない場合は可能性のあるいずれかの way が対応するものと約束する. このとき, **データ A はキャッシュ上の (x,y) に格納されると表記する**ことにする.

さて, 近代的なプロセッサでは, キャッシュは階層構造をなしており, L1(レベル 1) から L2, L3 の 3 階層のものも存在する. L1 はプロセッサに最も近い位置に設置され高速かつ低レインテシであるが容量は小さい. 逆に, L2,L3 となるに従って大容量かつ低速になる. 更に, キャッシュと主記憶の間には仮想記憶のページと物理記憶装置との対応表を格納する **TLB(Translation Lookaside Buffer)** が存在する. 近代的な OS では物理メモリではなく仮想記憶によるページ単位のメモリ管理がされており, そのページ管理システムとして殆どのマイクロプロセッサには備わっている. TLB はキャッシュと同様の構造をしているが, より高価なフルセットアソシアティブ方式 (場合よってはフルではない) をとることが殆どであるため, エントリ数はキャッシュのライン数よりもずっと少ない. また, TLB ミスのペナルティはキャッシュミスよりもずっと大きいといわれている.

2 キャッシュミスによる性能不安定化

本節では幾つかの例を挙げながらキャッシュミスによる性能不安定化の原因とその処方箋について説明をする. 説明の中で fortran プログラムを基にするがこれは fortran の多次元配列の持つ連続性や整合寸法の特徴を利用するためである. C 言

語などに精通した読者は, fortran の配列のとり方に注意をしながら読んでいけば特に問題はないであろう.

先節の説明にあるように、キャッシュは主記憶に比べ小規模ではあるがアクセスが数クロック単位でなされるため, 高性能計算ではデータを如何にキャッシュ上にとどめておくかが鍵になる. データがキャッシュ上にない状態つまり「キャッシュミス」を避けることが重要なのであるが, では, どのような場合にキャッシュミスが起こるのであろうか? 参考書によく現れるのが次に 3 つのミスである.

1. 初期参照ミス (compulsory miss) キャッシュラインを最初にアクセスするときに起こるミス.
2. 容量性ミス (capacity miss) キャッシュしたいライン数がキャッシュ容量を上回ることで起こるミス.
3. 競合性ミス (conflict miss) 同じインデックスをもつ異なるキャッシュラインへのアクセスが発生することで起こるミス.

2.1 初期参照ミス (compulsory miss)

1. のミスはプロセッサとキャッシュの関係から避けることができない. 通常大きな配列は動的に (`malloc` などで) 確保されるが, 確保した段階ではメモリ領域にタッチしないためメモリがキャッシュにロードされることはない. 実際, 初めてそのメモリ領域にアクセスするときにメモリ–キャッシュ間の転送が発生する. 1 は避けることができないキャッシュミスではあるが, 最近のマイクロプロセッサにはプリフェッチ (prefetch) 機能が備わっており, データを実際に利用するよりもずっと早い段階でアクセスしてキャッシュへのロードを完了させておくことができる. コンパイラの最適化やマイクロプロセッサが持つハードウェアプリフェッチの機能があるので, 利用者は特に気にする必要がないものである.

2.2 容量性ミス (capacity miss)

2. のミスは利用者がプログラム中で利用するメモリサイズとキャッシュサイズのアンバランスから生じるもので, その処方箋は幾つか存在する. プログラム上で利用するデータをキャッシュ上に留めておき, キャッシュの効果 (短時間でのデータアクセス) を得ることがもともとのキャッシュの考え方である. 一方, キャッシュは非常に高価でメモリサイズと比較してほんの僅かしかプロセッサ上には搭載されないため, 通常のプログラムではプログラム実行中常に全データをキャッシュ上に留めることはできない. そこで, プログラムのある区間に限定しデータの一部をできる限りキャッシュに留める (局所化する) ことでキャッシュの恩恵を得ることができる. よく知られた手法として, 行列·行列積のブロック化がある. また, ブロック化

ができなくてもプリフェッチの恩恵が得られるような場合には, 少々のアンバランスがあっても気にする必要は無い.

2.3 競合性ミス (conflict miss)

2. のミスの処方箋は局所化により利用するデータ (ブロック) をキャッシュサイズに抑えることであり, それを意識的にプログラムに反映することが容易である. 一方, 3. のミスはキャッシュの構造に由来するものであることから, その原因を理解することが容易ではない. プログラムの性能不安定性の主要な部分を占めるにも関わらず, 原因不明と処理されてしまうことも多いのではないだろうか. (実際, プロセッサのタイミングカウンタを見ることで初めてキャッシュミスが性能劣化の原因と判明した場合, このタイプのキャッシュミスであることがある)

簡単な例を紹介しよう. あるプログラムのコアループ中で 1 次元配列 A, B, C, D が利用されているとする. このとき, 各配列がページアライメントされていると, 各配列の同一インデックス要素は同じキャッシュラインへのアクセスに格納される. ここでは, 各配列の先頭 A(1) などが (0,*) に対応するアライメントを仮定する[2]. キャッシュが 4way で 1 ラインが 2 ワードに相当するとき, 各配列は次図 2 のように格納されるであろう.

	(0,*)		(1,*)		(2,*)		···
(*,0)	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	···
(*,1)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	···
(*,2)	C(1)	C(2)	C(3)	C(4)	C(5)	C(6)	···
(*,3)	D(1)	D(2)	D(3)	D(4)	D(5)	D(6)	···

図 2:

この状態で, 配列 A,B,C,D は全く衝突することなくキャッシュに収まることが理解できます. また, コアループが終了しても配列 A,B,C,D の総量がキャッシュサイズを超えなければ, 配列 A,B,C,D はキャッシュ内に留まることも理解できるであろう. 再度コアループに入ったとしても, キャッシュ内に留まるデータを高速に利用できるため, 性能はほぼ最高性能に達すると期待できる.

一方, このコアループにもう一つ配列 E(同じくページアライメントされているとする) の利用が追加された場合に, キャッシュの利用はどうなるだろうか? 答えは「キャッシュミスが発生し性能は数十分の一に劣化する」である.

まず, ライン (0,*) の利用を考えて見よう. (0,*) を利用するのは

[2]ここでは容量の小さい L1 キャッシュ利用を想定するが, ページサイズがキャッシュ1way 分よりも大きいことを仮定する. 大容量のキャッシュの場合はページアライメントはキャッシュ上ランダムになるためこの限りではないが, 基本的議論は同様におこなえるだろう.

{A(1),A(2)},{B(1),B(2)},{C(1),C(2)},{D(1),D(2)},{E(1),E(2)}

です. ここで, 中括弧で囲んだ単位がキャッシュラインに相当します.

A(1)→B(1)→C(1)→D(1)→E(1)→A(2)→ ···

のようなアクセスがループ内でされたら, (0,*) は 4way しかないので D(1) までのアクセスについては問題なく進行するが, E(1) をアクセスした時点で A(1) が入っているライン (0,0) をメモリに書き戻して, その部分に E(1)(実際は {E(1),E(2)} のライン) を格納する. そして, 次の A(2) のアクセス時には {A(1),A(2)} のラインはもはやキャッシュ上にないので, メモリにアクセスしないといけない. このとき, B(1) が入ったライン (0,1) をメモリに書き戻して, A(2) の入ったラインを (0,1) に格納することになる. この様に, 常にキャッシュラインの奪い合いが発生し, 連鎖的なキャッシュミスが発生する. このような現象をキャッシュスラッシング **(Cache thrashing)** と呼ぶことがある.

キャッシュスラッシングはハードウェアから見たらキャッシュの連想性 (way 数) の低さが原因ともいわれるが, ソフトウェアからこの現象をある程度解決することができる. まず, 解消すべきはデータ利用時の同一キャッシュラインへのアクセスをなくすことである. もし, 配列 E がページアライメントされてなく E(1) がキャッシュ上の (1,*) にあったとしよう. このとき,

A(1)→B(1)→C(1)→D(1)→E(1)→A(2)→ ···

のようなアクセスがあったとしても, それらのアクセス中にはキャッシュ競合は起こらないことは理解できるであろう. A(1), B(1), C(1), D(1), E(1) がアクセスされたときのキャッシュの状態は以下のとおりである (図 3).

	(0,*)	(1,*)	(2,*)	···
(*,0)	A(1) A(2)	E(1) E(2)		···
(*,1)	B(1) B(2)			···
(*,2)	C(1) C(2)			···
(*,3)	D(1) D(2)			···

図 3:

先の例の様に, E(1) のアクセス時に A(1) をメモリに追いやって ···, といったラインの競合の連鎖は結果として発生しなくなる. ただし, 容易にわかるようにコアループを終了し再度同じコアに突入する際には, 配列 E の殆どはキャッシュ上に残っていないことが判る. しかし, 配列 A,B,C,D はキャッシュ上にあるためその分はキャッシュの恩恵を得ることができる. ここで, A は E 利用時に上書きされるため B,C,D の効果のみがあることに注意したい. 1. の説明でも言及したように, A,E などのメモリ–キャッシュ間アクセスが発生しても, 一般にプリフェッチによってア

クセス待ちのペナルティを隠すことができる. ただし, プリフェッチは数ステップ先までのデータ利用について発行されるので, 上の例の様にE(1)が(1,*)に対応していては, Aのプリフェッチされたデータと Eのプリフェッチされたデータが衝突することになり, 結局両者のプリフェッチ効果が得られなくなる. したがって, このような場合にはプリフェッチの先読みよりも十分離れた位置にE(1)がくるようにアライメントすればよいということになる. アライメントの方法はいくつか候補があるが, 動的に確保する配列の場合キャッシュの1行分 (1way分) を余分にとった上で先頭アドレスから数ライン分ずらした位置を配列の先頭として利用すればよいということになる. なお, 静的な場合も同様の手法が使える.

2.4 キャッシュ性能不安定性とチューニング

ここまで簡単な競合性ミスの例を説明してきた. 基本的には1次元配列で生じることを例示したが, より高次元配列でも当然同様の事例が発生する. また, 性質が悪いことに多次元配列であるが故に発生する場合もある. さらに, 本節のタイトルに挙げたように性能を上げるために行うチューニング作業の結果として, ある特定の条件化で競合性ミスが発生することが知られている. ここでは, チューニングと結びつけて解説をしていく.

まず, 次に挙げるfortranプログラムは特に何のチューニングを施していない行列-ベクトル積を計算するループである.

```
integer :: LDA, N
real    :: A(LDA,N), X(N), Y(N)
Y(1:N)=0.0
do J=1,N
   do I=1,N
      Y(I)=Y(I)+A(I,J)*X(J)
   enddo
enddo
```

このプログラムをチューニングする際に, 外側のループを展開するループアンローリングを行うことがある. Nが2で割り切れると仮定して, 次の様に変形する.

```
integer :: LDA, N
real    :: A(LDA,N), X(N), Y(N)
Y(1:N)=0.0
do J=1,N,2
   do I=1,N
      Y(I)=Y(I)+A(I,J)*X(J)+A(I,J+1)*X(J+1)
   enddo
```

```
      enddo
```

この場合, A(I,J)*X(J) の計算量に対して Y(I) へのアクセス数を比較すると, プログラム変形前に比べて半分になることがわかる. その分 (つまり計算に要する時間が増加するという意味で), 配列 Y がキャッシュ上に長く留まるようになるため性能が向上するのである. 筆者は文献 [1] で, A が対称行列のときの性能測定を行っている. その際のプログラムは以下のようなものである.

```
integer :: LDA, N
real    :: A(LDA,N), X(N), Y(N)
do J=1,N
   do I=1,N
      Y(J)=Y(J)+A(I,J)*X(J)
      Y(I)=Y(I)+A(I,J)*X(J)
   enddo
   Y(J)=Y(J)+A(J,J)*X(J)
enddo
```

このプログラムを以下のようにループアンローリングによりチューニングを行った. (ただし, 下三角要素は 0 であると仮定する)

```
integer :: LDA, N
real    :: A(LDA,N), X(N), Y(N), D(N)
do J=1,N
   D(J)=A(J,J); A(J,J)=0.0
enddo
do J=1,N,2
   do I=1,J
      Y(J+0)=Y(J+0)+A(I,J+0)*X(J+0)
      Y(J+1)=Y(J+1)+A(I,J+1)*X(J+1)
      Y(I)=Y(I)+A(I,J+0)*X(J+0)+A(I,J+1)*X(J+1)
   enddo
enddo
do J=1,N
   Y(J)=Y(J)+D(J)*X(J); A(J,J)=D(J)
enddo
```

このプログラム変形が正しいことは読者の皆さん各自で行って欲しい. この段階ではループアンローリングは 2 段しか行われていないが, これをプロセッサが有するレジスタ資源が許す限りの段数で行えば性能が向上していくことが知られている.

ここで, この性能の上昇は一般的に正しいのであるが, ある特定の状況下ではNGであることを確認できる．文献 [1] では, 行列の次元を 4032 から 4160 次元まで 1 次元毎に (4096 次元を中心に ±64 次元) 性能測定がなされている．測定した計算機は日立製の SR8000F1 モデルの 1PE で, プロセッサは Power3 に準拠し擬似ベクトル機能を追加したものである．また, キャッシャは L1 のみで 128KB の 4way 構成である．ループアンローリングは最大 15 段まで可能であり, その測定結果を次図 4 に示す (図中の ‘M-数字’ の, 数字部分がアンローリング段数を指す).

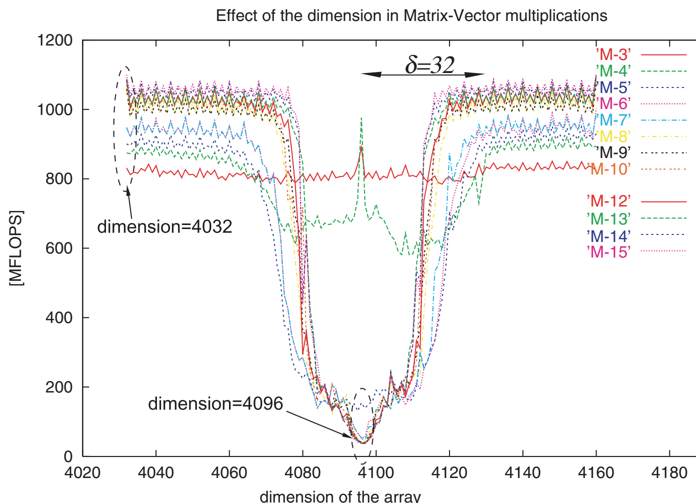

図 4: SR8000F1 での行列ベクトル積の性能測定結果

図 4 は横軸に行列の次元 (サイズ), 縦軸に実行時間から算出した行列ベクトル積ルーチンの性能 (MFLOPS, FLOPS は 1 秒間あたりに行われた浮動小数点演算量, M は 10^9 を表す単位) をプロットしたものである．なお, 同図は性能測定結果のほんの一部分を拡大したに過ぎないものである．4080 次元から 4120 次元あたり (4096 の前後 32 次元) で大きな性能の劣化が見られるが, ほとんどの次元では性能はほぼ安定している．

- では, この性能曲線の落ち込みの原因は何だろうか?
- また, 底の中心である 4096 次元の 4096 という数値には何の意味があるのだろう?

答えを先に示してしまうと, 前節で解説した競合性ミスによるキャッシュスラッシングが発生しているのである．先に示したプログラムで LDA なる整合寸法が指定されていたが, この値は N と同じ値を採っていた事が問題を生じさせることとなったのである．これについて説明しよう．fortran での多次元配列は, 整合寸法

配列 (英語では leading dimension とも呼ばれることがある) により, 1 次元データを折り返し多次元配列表現をしている．このとき, A(1,1) と A(2,1) は必ず連続メモリ上にとられる (仮想記憶でのページ境界の場合を除く), 一方 A(1,1) と A(1,2) は整合寸法配列語分離れている．ループ中に A(I,J),A(I,J+1) といったアクセスがあった場合, A(I,J+1) は A(I,J) から **(整合寸法) % (キャッシュサイズ/way 数/語長 (倍精度なら 8 バイト))** 離れた位置 (語長単位で) に格納されることになる．

SR8000F1 モデルの場合, **(キャッシュサイズ/way 数/語長) は 4096 語 (=128K/4/8)** に相当する．したがって, N=LDA=4096 のとき A(I,J) や A(I,J+1) などのアクセスは同じキャッシュラインを要求することになる．一方, way 数は 4 しかないので, 4 段のループアンローリングまでは配列 A によるキャッシュスラッシングは発生しないが, 5 段以上ではキャッシュスラッシングは避けられない．実際にはベクトルデータが 2 本 (X と Y) あるためそのアクセスも含めると, 3 段までしかスラッシングを回避できないことになる．

では, 次に N が 4096 の近傍のときはどう解釈すべきなのか? N=4097 のとき配列 A がキャッシュに収まる様子を調べてみる．SR8000F1 のキャッシュラインは 128 バイト (倍精度浮動小数点で 16 語) なので, これまでの図とは若干異なる．図 5 中は添え字のみの省略形で書き出した．

	(0,*)					···
(*,0)	(1,1)	(2,1)	···	(15,1)	(16,1)	···
(*,1)		(1,2)	···	(14,2)	(15,2)	···
(*,2)			···	(13,3)	(14,3)	···
(*,3)			···	(12,4)	(13,4)	···

図 5:

4 段アンローリングを施した場合の J=1 でのキャッシュの様子を示している．この状態でも, 配列 A のみでライン (0,*) を占有しているため, キャッシュスラッシングは必ず発生します．このキャッシュの充填は規則的であり, 賢明な読者であれば配列の折り返しである整合寸法を LDA=4096+6=4102 とすれば, 4 段目の A(1,4) は A(1,1) から 6*3=18(> 16) 離れており, (0,3) ではなく (1,3) に格納されるためスラッシングを回避できることに気づくであろう (図 6).

ただし, 先のグラフを見るところでは 4 段のアンローリングの性能は 4130 次元程度まではよくない．これは, SR8000 がプリフェッチをかなり先読みしているためである．グラフから見て取れるように, もう 2 ライン分ほど大きめにとらないと駄目だという事が判る．

したがって, 本件の ad hoc な処方箋として

- 多次元配列の整合寸法は 4096+ラインサイズ*C (C は 1 より大きな整数)

	(0,*)					(1,*)			···
(*,0)	(1,1)	(2,1)	···	(15,1)	(16,1)	(17,1)	(18,1)	···	···
(*,1)			···	(9,2)	(10,2)	(11,2)	(12,2)	···	···
(*,2)			···	(3,3)	(4,3)	(5,3)	(6,3)	···	···
(*,3)							(1,4)	···	···

図 6:

が挙げられる. ここで, ラインサイズ*C を加えた別の理由として, キャッシュを先頭から必ず利用することでキャッシュ容量を最大限に使い切ることを意図している.

ここで挙げた方法は, 他の多くのプロセッサにあてはまるはずである (ただし, 数値 4096 はキャッシュの構造に依存する).

2.5 スラッシングはあらかじめ予測できないのか?

では, ここで「k 段のアンローリングをしたけれども, ループ内に登場する配列 A の整合寸法が N_A のときスラッシングは起こるのか?」という判定をしたくなる. この様な判別が容易にできれば, 性能劣化のポケットからプログラムを救助でき, 結果的に性能向上をソフトウェアの視点から実施できるはずである.

この判定については, 先のキャッシュの充填パターンから判るように, ある種の規則性 (周期性) を持つので, 簡単な剰余計算で判別することができる. 詳細は文献 [2] によるが, 判別式のみを書けば次のようになる.

> **仮定:** n-way 連想記憶キャッシュを想定し 1way には 2^L 語格納可能とする (つまり, キャッシュの総容量は $n * 2^L$ 語である). また, 整合寸法を N_A と書くことにする.
>
> 今, コアループ中で k 段のループアンローリングを施した際, 以下の不等式を成立させるような整数 i が存在するとき, キャッシュスラッシングが起こる可能性がきわめて高い. ただし, 定数 δ はラインサイズもしくはその数倍を指す.

$$0 < \exists i < k/n, |\mathrm{mod}(i * N_A + 2^L, 2^{L-1}) - 2^{L-1}| < \delta \quad (1)$$

したがって, 上記の不等式を成立させないような N_A を選んで, 配列を確保すればよいのである.

2.6　さらに精密なキャッシュ性能安定化

ここまで熟読された読者は, 先節は配列 A 単体のアンローリングに対する処方箋であったことを見抜いた事であろう. そして, 処方箋の要点は「データのアクセスパターンに応じてデータレイアウトに細心の気をつけよ」ということを理解したであろう. 実は, 複数配列が同一ループ内に登場した場合には配列単体で登場する以上に競合性ミスの可能性は高くなる.

先節の記号と手法を流用することにしよう. 2 次元配列 A, B が同時にループ内に登場し, さらにそのループをアンローリングしたら競合性ミスの最も簡単な例で示したように way 数を超えた同一ラインへのアクセスが発生する. そこで, B(1,1) のアドレスを A(1,1) から最も離れた位置 (2^{L-1},*) に設定すれば多段のアンローリングを設定しても競合の問題は起こりにくくなる. 配列数がさらに多くなった場合は, お互いの距離が最大となるように配置すれば競合の可能性は下がるはずである. なお, 配列のアクセスパターンが決定的であっても配列数が増加したり高次元配列利用となると完全に競合を起こさない配置を決めることは困難になる. 可能であればループ内の配列数の利用を適切なものに制限するなどしたほうがよいであろう.

3　さいごに

キャッシュミスは容量性ミスばかりが気にされがちであるが, 極端な性能劣化は競合性ミスによって誘発されるキャッシュスラッシングによるものであることが多い. それらは発見が困難であり, ハードウェアカウンタなどを用いない限り見落とされがちである. 本記事の主張は「注意すべきキャッシュ不安定性はデータアクセスパターンとデータレイアウトが相互に影響しあうことで起こるキャッシュスラッシングによるもの」ということであり, それらにも注意をしたプログラミングや性能解析を行うべきであるということである. それらの多くは, 記事内で紹介したように配列データを注意深く配置することによりソフトウェア側からそれを制御し, 性能安定化を行うことが可能である. 本記事の読者の皆様はこの点に注意を払って高性能プログラムの作成に役立てて頂ければと願います.

参考文献

[1] 今村俊幸, 直野健：性能安定化を目指した自動チューニング型固有値ソルバーについて, 先進的計算基盤シンポジウム SACSIS2003 論文集, pp.145-152, 2003.

[2] 今村俊幸, 直野健：キャッシュ競合を制御する性能安定化機構内蔵型数値計算ライブラリについて, 情報処理学会論文誌コンピューティングシステム, No. SIG 6 (ACS 6), Vol. 45, pp. 113-121, 2004.