Intel Xeon プロセッサにおける Cache Coherency 時間の測定方法と大規模システムにおける実測結果

Performance Measurement Method of Cache Coherency Effects on a large Intel Xeon Processor System

河辺　峻[†1]　　古谷英祐[†2]

KAWABE Shun,　FURUYA Eisuke

要旨

現在のプロセッサの構成は，メモリを共有するマルチコア化が進んでいる。それぞれのコアには他のコアと共有しない専用のキャッシュを持っている。このため他のコアと共有するエリアを更新し，他のコアがこのエリアをアクセスするときキャッシュ間の内容の一貫性を保つための論理回路が動作する。この時間を cache coherency 時間とすると，Linux カーネルで提供されている atomic_inc 関数を用いてこの時間を測定する方法を考案し, Intel Xeon プロセッサの 64 コアの大規模システムにて実測を行いその結果を分析した。

1. はじめに

プロセッサは現在マルチコア化による高速化が進んでいる。これは 1 つのコアによる性能向上が難しくなりつつあるためである。マルチコア化により並列処理が可能なプログラムやスループットを主とする多重プログラムにとっては高速化が期待できる。さらにデータベースの大規模化などにより、多数のコアによる並列処理が必要になっている。しかしプロセッサ内部のキャッシュ構成は複雑化しており、それぞれのコアが所有するキャッシュは、Level1 cache(L1)、Level2 cache(L2)、Level3 cache(L3)に階層化されている。さらに更新されたデータを階層化されたキャッシュすべてとメモリに反映する”writethrough”方式と、更新されたキャッシュのみに反映する”writeback”方式とがある。最新の Intel Xeon プロセッサチップでは、L1 と L2 キャッシュはコア間では共有せず、L3 キャッシュは同一プロセッサチップのコア間で共有している。また複数のプロセッサチップを搭載するサーバでは、L3 キャッシュ間で情報の交信を行っている。更新方式は”writeback”方式である。このような構成において、コア間で共通のエリアを更新する場合、cache coherency（キャッシュ間の内容の一貫性）を保つ

†1 明星大学・東京大学　　†2 明星大学

Meisei University・Tokyo University　　Meisei University

ためのオーバーヘッドが課題となりつつある。しかしこのオーバーヘッドに関する数値はほとんど公開されていない。この論文ではこの cache coherency の時間を、Linux カーネルで提供されている atomic_inc 関数を用いて測定する方法を新たに考案し、実測プログラムを作成した。実測はプロセッサチップが 1 つの小規模システムから、プロセッサチップが 2 つの中規模システム、さらにメモリを共有するプロセッサチップが 8 つの 64 コアの大規模システムまで実測を行い、その結果の分析により特にプロセッサの性能指標である CPI(Clock cycle Per Instruction)に与える影響について考察した。

cache coherency の時間を実測する研究は、メモリやキャッシュの latency やバンド幅を実測する研究の中で主に行われてきた。例えば Molka 他の論文[1]では、転送バイト数に対する latency やバンド幅の測定を行っている。この中でキャッシュは MESIF (Modified, Exclusive, Shared, Invalid, Forwarding) のいずれかの状態になるが、プログラムで強制的にキャッシュを常に Exclusive の状態にして access latency を測定することにより、cache coherency の時間を実測している。これに対して本研究では atomic_inc 関数を用いることにより、cache coherency の時間を実測した。64 コアなどの大規模システムでは、この方法の方がより容易に実測できると考える。

2. atomic_inc 関数を用いた性能測定方法

はじめに、新しい測定方法の提案として atomic_inc 関数を用いて cache coherency の時間を測定する方法について述べる。

2.1 Linux カーネルの atomic_inc 関数動作

Linux カーネルの atomic 操作は、変数の読み出しと書き込み（更新）を不可分な操作として扱うものである。マルチスレッドで動作する場合に、Intel の x86 アーキテクチャでは共通にアクセスする変数にハード的に lock をかけて他からのアクセスを禁止して更新を行う。C 言語で用いる atomic_inc 関数は指定した変数に lock をかけて変数の値を＋１する機能である。例えば、

```
#define LOCK "lock ; "    /*ハード的に lock をかける指示 */
typedef struct {volatile int counter; } atomic_t;
        atomic_t abc;

        static __inline__ void atomic_inc(atomic_t *v) {
            __asm__ __volatile__(
                        LOCK "incl %0"
                        :"=m" (v->counter)
                        :"m" (v->counter));
        }
```

としておいて、

```
            atomic_inc(&abc.counter);
```

と書くと変数 abc.counter の値が＋１される。

ここで 2 つのコアにおいて、メモリ上の共通変数に交互に atomic_inc 関数を実行させると、各コアで交互に排他的にメモリ上の共通変数が＋１される。さらにその時、各コアにある cache coherency（一貫性）を保つための論理回路が必ず動作する。したがってマルチスレッドプログラミングを用いて、指定したコアで atomic_inc 関数を交互に実行させると、cache coherency（一貫性）の時間が測定可能になる。

2.2 atomic_inc 関数の性能測定方法とプログラム

マルチスレッドプログラミングは Linux の C 言語の Pthread を用いて行った。まず実行するコアを指定する必要があるが、これは affinity 機能を使用してコアの指定を行った。

・affinity 機能を使用したコアの指定(コア 0 を指定した例)

```
cpu_set_t mask0;
CPU_ZERO(&mask0);
CPU_SET(0,&mask0);
rv=sched_setaffinity(0,sizeof(mask0),&mask0)
```

次にマルチスレッドの各スレッドの測定部分のプログラムであるが、これは gettimeofday 関数を持ちいて tr 回のループを測定した。

```
gettimeofday(&st,NULL);
for(a=0;a<tr;a++)
    {
        atomic_inc(&abc.counter);
    }
gettimeofday(&et,NULL);
```

gettimeofday 関数はマイクロ秒(μs)単位の時間を測定する関数である。1 回あたりの atomic_inc 関数の時間は、ns で小数点以下 1 桁程度の精度が必要になる。そのための精度を保つために 1000 万回のループを計測した。

2.3 同一プロセッサチップ内の atomic_inc 関数の動作

まず同一プロセッサチップ内からはじめ、異なるプロセッサチップ間さらに大規模構成へと進めていく。

図 1 は同一プロセッサチップ内の atomic_inc 関数の動作を示したものである。図 1 において core0(C0)から atomic_inc 関数を実行する。まず Memory にあるデータ a を L1 まで持ってくる。この時、L2、L3 の対応エリアも a の値になる。Intel Xeon プロセッサの cache 制御は”writeback”方式であるので、atomic_inc 関数により値が更新されるのはこの場合 L1 のみで、L1 の値が a+1 に更新される。次に core1(C1) から atomic_inc 関数を実行する。この場合、真の値は C0 の L1 にあるので、まず C0 の L1 の内容の値 a+1 を C0 の L2 および L3 に書き込む。

この動作の後、C1 は L3 から a+1 の値を L1 まで持って来て値を a+2 に更新する。同じようにして次は core0(C0) から atomic_inc 関数を実行する。この場合、真の値は C1 の L1 にあるので、まず C1 の L1 の内容の値 a+1 を C1 の L2 および L3 に書き込む。
この動作の後、C0 は L3 から a+2 の値を L1 まで持って来て値を a+3 に更新する。

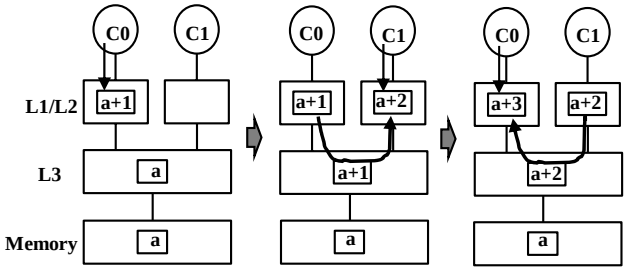

図 1 同一プロセッサチップ内の atomic_inc 関数の動作

このように、同一プロセッサチップ内の atomic_inc 関数の動作は、共有する L3 を介して行われる。更新された最新の値はそのコアの L1 のみにある。

2.4 異なるプロセッサチップ間の atomic_inc 関数の動作

図 2 は異なるプロセッサチップ間の atomic_inc 関数の動作を示したものである。図 2 において core0(C0)から atomic_inc 関数を実行する。まず Memory にあるデータ a を L1 まで持ってくる。この時、L2、L3 の対応エリアも a の値になる。Intel Xeon プロセッサの cache 制御は writeback 方式であるので、atomic_inc 関数により値が更新されるのはこの場合も L1 のみで、L1 の値が a+1 に更新される。次に core4(C4) から atomic_inc 関数を実行する。この場合、真の値は C0 の L1 にあるので、まず C0 の L1 の内容の値 a+1 を C0 の L2 および C0 の L3 に書き込む。この動作の後、C4 は C0 の L3 から a+1 の値を QPI 経由で C4 の L3 から L1 まで持って来て値を a+2 に更新する。同じようにして次は core0(C0) から atomic_inc 関数を実行する。この場合、真の値は C4 の L1 にあるので、まず C4 の L1 の内容の値 a+1 を C4 の L2 および L3 に書き込む。この動作の後、C0 は C4 の L3 から a+2 の値を QPI 経由で C0 の L3 から L1 まで持って来て値を a+3 に更新する。

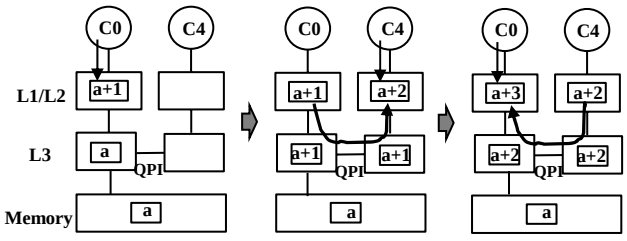

図 2 異なるプロセッサチップ間の atomic_inc 関数の動作

このように、異なるプロセッサチップ間の atomic_inc 関数の動作は、QPI を経由してそれぞれが所有する L3 を介して行われる。更新された最新の値はそのコアの L1 のみにある。このため atomic_inc 関数の動作時間は、同一プロセッサチップ内よりも QPI を経由して情報を交換する異なるプロセッサチップ間の方が大きいと考えられる。

2.5 評価に用いたプロセッサ(1 ボード構成)

図 3 に今回の評価で用いた中規模システム（1 つのボードに 2 つのプロセッサチップを搭載）のプロセッサ構成図を示す。

Intel E5620(Nehalem Westmere-EP)プロセッサは図 3 に示すように、1 つのプロセッサに 4 つのコアがありコアごとに 32KB の L1 のデータおよび命令キャッシュと 256KB の L2 キャッシュを持ち、各コアが共有する 12MB の L3 キャッシュを持っている。周波数は 2。40GHz で TPD は 80W である。このプロセッサチップが 1 つのボード上に 2 つ搭載されており、Quick Path Controller を通して QPI(Quick Path Interconnect)でプロセッサチップ間の情報の交信を行っている。また、それぞれのコアが HT(Hyper Threading)機能を持っている。このためプログラムからは論理的には 16 のコアがあるように見える。

今回のプログラムでは affinity 機能を用いて使用するコアを指定した。OS は Linux の Fedora14(64b) (kernel 2.6.35)を使用した。また gcc の version は 4.5.1 である。

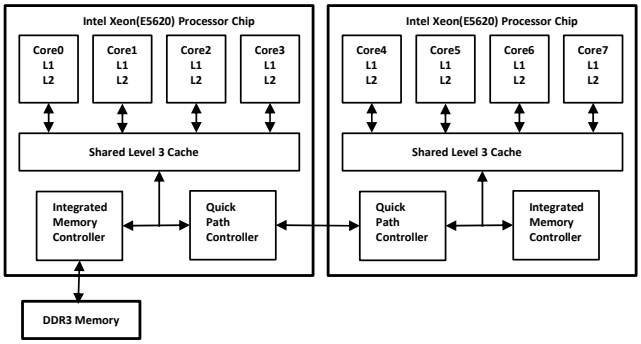

図 3 評価に用いたプロセッサ(1 ボード)構成図

コア番号の指定に当たっては、cat コマンドを利用して、コア番号を定めた。図 3 の構成図のプロセッサでは、

$ cat /proc/cpuinfo | grep “physical id”

とすると次の Processor、core id、physical id の情報が表示される。

Processor	core id	physical id	定めたコア番号
0	0	0	core0[C0]
1	0	1	core4[C4]
2	1	0	core1[C1]
3	1	1	core5[C5]
4	9	0	core2[C2]

5	9	1	core6[C6]
6	10	0	core3[C3]
7	10	1	core7[C7]
8	0	0	core8[C8]
9	0	1	
10	1	0	
11	1	1	
12	9	0	
13	9	1	
14	10	0	
15	10	1	

この情報よりコアの番号を、次のように定めた。

[0, 0, 0]⇒C0、[2, 1, 0]⇒C1、[4, 9, 0]⇒C2、[6, 10, 0]⇒C3

[1, 0, 1]⇒C4、[3, 1, 1]⇒C5、[5, 9, 1]⇒C6、[7, 10, 1]⇒C7

[8, 0, 0]⇒C8

2.6 1ボード構成の性能実測結果と考察

性能測定は次の 4 つのケースについて行った。

(1)atomic_inc 関数の単体性能 [測定 1]

(2)cache coherency 動作を伴わない atomic_inc 関数の性能 [測定 2]

同一プロセッサチップ内(on die)の atomic_inc

(3)関数の性能 [測定 3]

(4)同一ボード内（異なるプロセッサチップ間）の atomic_inc 関数の性能 [測定 4]

測定結果

測定 1:core0[C0]にて atomic_inc 関数を実行させて性能を測定する。結果は 10.52ns となった。ちなみにハード的に lock をかけずに実行すると、結果は 3.97ns であった。

測定 2 : core0[C0]の同一コア内で HT(Hyper Threading)機能を利用して、[C0,C8]のペアで 2 つの atomic_inc 関数を実行させて性能を測定する。この場合は L1、L2 キャッシュを共有しているので cache coherency 動作は伴わない。結果は平均して 11.13ns となった。

測定 3 : 同一プロセッサチップ内(on die)の atomic_inc 関数の動作として、[C0,C1]、[C0,C2]、[C0,C3]のペアで 2 つの atomic_inc 関数を実行させて性能を測定する。結果は平均して 38.53ns となった。

測定 2 で得られた値 11。13ns をこれから引いた値 38.53-11.13=27.40ns が同一プロセッサチップ内の cache coherency 時間の平均値と見なすことができる。

測定 4 : 同一ボード内（異なるプロセッサチップ間）の atomic_inc 関数の動作として[C0,C4]、[C0,C5]、[C0,C6]、[C0,C7]のペアで 2 つの atomic_inc 関数を実行させて性能を測定する。結果は平均して 124.84ns となった。

測定 2 で得られた値 11.13ns をこれから引いた値 124.84-11.13=113.71ns が同一ボード内（異なるプロセッサチップ間）の cache coherency 時間の平均値と見なすことができる。表 1 にこれらの 1 ボード構成における測定結果のまとめを示す。

表 1 1 ボード構成における測定結果のまとめ

	測定構成	時間(ns)	備考
測定 1	core0[C0]	10.52	
		3.97	lock なし
測定 2	[C0,C8]HT	11.13	
測定 3	[C0,C1]	39.81	
(on die)	[C0,C2]	37.18	平均 38.53ns
	[C0,C3]	38.62	
測定 4	[C0,C4]	120.27	
(1 hop)	[C0,C5]	127.38	平均 124.84ns
	[C0,C6]	126.96	

考察

CPI(Clock cycle Per Instruction)に与える影響について考察する。CPI は 1 命令の実行に要するクロックサイクル数で、

性能（実行時間）=CPI/周波数＊実行命令数

の関係があるので CPI は小さい程性能（実行時間）が良い。

まず cache coherency 時間は、同一プロセッサチップ内(on die)では平均 27.40ns(65.76cyc)、異なるプロセッサチップ間(QPI 1hop)では平均 113.71ns(272.90cyc)となる。1 命令において基本 CPI を 2.0 としたとき、cache coherency の命令あたりの発生頻度を横軸にとり、縦軸に CPI をとったグラフを図 4 に示す。

これから分かるように、同一プロセッサチップ内で発生する cache coherency が CPI に与える影響は比較的軽微である。

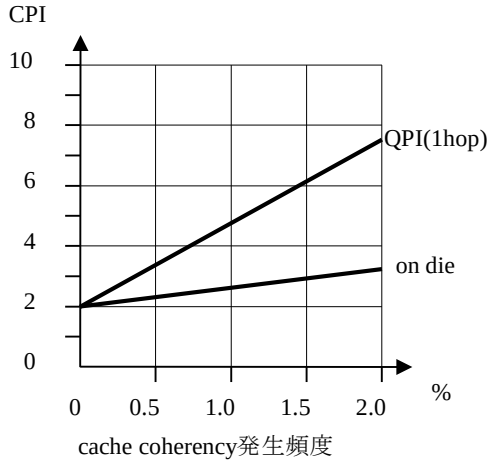

図 4　1 ボード構成における CPI に与える影響

しかし異なるプロセッサチップ間で発生する cache coherency が CPI に与える影響は非常に大きい。これは異なるプロセッサチップ間で発生する cache coherency 時間が、同一プロセッサチップ内で発生する cache coherency 時間の 4.15 倍にもなっていることによる。
また Molka らによる論文[1]では、転送バイト数に対する access latency の測定を行っている。この中ではプログラムで強制的にキャッシュを Exclusive の状態にして access latency を測定することにより、cache coherency の時間を実測している。測定した構成は図 3 に近く、プロセッサは Nehalem Xeon X5579(2.9GHz)である。これによると同一プロセッサチップ内(on die)では 28.3ns、異なるプロセッサチップ間(1 hop)では 102-109ns と、本報告の 27.4ns および 113.7ns と近い結果になっている。

3　大規模構成での測定

次に大規模システムの構成の 64 コアシステムを測定する。

3.1 メモリを共有する 64 コアの大規模システム構成

評価で用いたプロセッサの構成図を示す。

Intel X7560(Nehalem)プロセッサは図 5 に示すように、1 つのプロセッサに 8 つのコアがありコアごとに 32KB のデータおよび命令キャッシュと 256KB の L2 キャッシュを持ち、各コアが共有する 24MB の L3 キャッシュを持っている。周波数は 2.26GHz で TPD は 130W である。このプロセッサチップが 1 つのボード上に 2 つ搭載されており、Quick Path Controller を通して 3 つの QPI(Quick Path Interconnect)でプロセッサチップ間の情報の交信を行っている。また、それぞれのコアが HT(Hyper Threading)機能を持っている。このためプログラムからは論理的には 16 のコアがあるように見える。

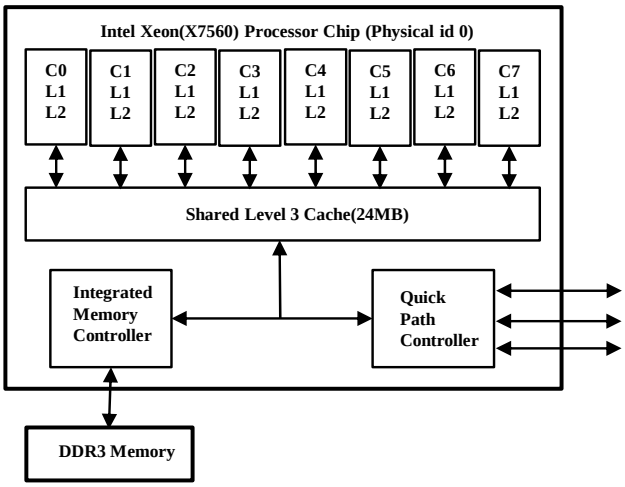

図 5　プロセッサチップの構成図

今回のプログラムでは affinity 機能を用いて使用するコアを指定した。OS は Red Hat Enterprise Linux 5(kernel 2.6.18)を使用した。また gcc の version は 4.1.1 である。

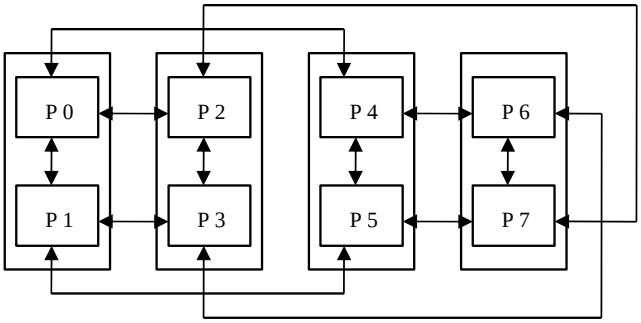

図 6　QPI 接続による 8 プロセッサチップ構成図

図 5 に示す Intel X7560(Nehalem)プロセッサチップは、ID がつけられており、図 5 のプロセッサチップは Physical id0 の例である。これを P0 と略す。64 コアの大規模システムでは、図 6 に示すように 1 つのボードに 2 つのプロセッサチップが搭載されている。全体では図 6 の左から[P0, P1] [P2, P3] [P4, P5] [P6, P7]と 4 つのボードから構成されている。さらに 1 つのプロセッサチップから他のプロセッサチップへは、QPI を使用して 1 hop ないしは 2 hops で接続できる構成になっている。

コア番号の指定に当たっては、cat コマンドを利用した。

$ cat　/proc/cpuinfo　|　grep “physical id”

とすると Processor、core id、physical id に関して 128 行の情報が表示される。同じ physical id 番号に対して 16 行の情報があり、physical id 番号が 0～7 に対応して 128 行の情報が表示さ

れる。physical id 番号を i としたときの場合の 3 列 16 行の表示を次に示す。

Processor	core id	physical id	定めたコア番号
0+i*16	0	i	core0[C0]
1+i*16	0	i	
2+i*16	1	i	core1[C1]
3+i*16	1	i	
4+i*16	2	i	core2[C2]
5+i*16	2	i	
6+i*16	3	i	core3[C3]
7+i*16	3	i	
8+i*16	8	i	core4[C4]
9+i*16	8	i	
10+i*16	9	i	core5[C5]
11+i*16	9	i	
12+i*16	10	i	core6[C6]
13+i*16	10	i	
14+i*16	11	i	core7[C7]
15+i*16	11	i	

この情報をもとにして physical id 番号ごとにコアの番号を定めた。また表示に関しては、例えば次のように表示することにする。
(P0.C0)：physical id0 のプロセッサチップにおける core0

性能測定は次の 5 つのケースについて行った。

(1)atomic_inc 関数の単体性能　[測定 1]
(2)cache coherency 動作を伴わない atomic_inc 関数の性能　[測定 2]
(3)同一プロセッサチップ内(on die)の atomic_inc 関数の性能　[測定 3]
(4)異なるプロセッサチップ間(1 hop)の atomic_inc 関数の性能　[測定 4]
(5)異なるプロセッサチップ間(2 hops)の atomic_inc 関数の性能　[測定 5]

3.2 測定結果

測定 1：physical id0 のプロセッサチップにおける core0 にて atomic_inc 関数を実行させて性能を測定する。結果は 11.94ns となった。
ちなみにハード的に lock をかけずに実行すると、結果は 5.35ns であった。
測定 2：physical id0 のプロセッサチップにおける core0 の同一コア内で HT(Hyper Threading)機能を利用して、2 つの atomic_inc 関数を実行させて性能を測定する。この場合 L1、L2 キャッシュを共有しているので cache coherency 動作は伴わない。結果は平均して 13.29ns となった。
測定 3:同一プロセッサチップ(P0)内の atomic_inc 関数の動作として、[C0,C1]、[C0,C2]、[C0,C4]

のペアで 2 つの atomic_inc 関数を実行させて性能を測定する。結果は平均して 38.88ns となった。
測定 2 で得られた値 13.29ns をこれから引いた値 38.88-13.29=25.59ns が同一プロセッサチップ内の cache coherency 時間の平均値と見なすことができる。
測定 4：異なるプロセッサチップ間(1 hop)の atomic_inc 関数の動作として[(P0.C0), (P1.C0)]、[(P0.C0), (P1.C2)]、[(P0.C0), (P1.C7)]、[(P0.C0), (P2.C0)] 、[(P0.C0), (P4.C0)]のペアで 2 つの atomic_inc 関数を実行させて性能を測定する。結果は平均して 218.32ns となった。
測定 2 で得られた値 13.29ns をこれから引いた値 218.32-13.29=205.03ns が異なるプロセッサチップ間(1 hop)の cache coherency 時間の平均値と見なすことができる。
測定 5：異なるプロセッサチップ間(2 hops)の atomic_inc 関数の動作として[(P0.C0), (P3.C0)]、[(P0.C0), (P5.C0)]、[(P0.C0), (P6.C0)]、[(P0.C0), (P7.C0)] 、[(P0.C0), (P7.C7)]のペアで 2 つの atomic_inc 関数を実行させて性能を測定する。結果は平均して 305.62ns となった。
測定 2 で得られた値 13.29ns をこれから引いた値 305.62-13.29=292.33ns が異なるプロセッサチップ間(1 hop)の cache coherency 時間の平均値と見なすことができる。
表 2 にこれらの 4 ボード構成における測定結果のまとめを示す。

表 2　4 ボード構成における測定結果のまとめ

	測定構成	時間(ns)	備考
測定 1	(P0.C0)	11.94	
		5.35	lock なし
測定 2	[(P0.C0), (P0.C0)HT]	13.29	
測定 3	[(P0.C0), (P0.C1)]	34.68	
(on die)	[(P0.C0), (P0.C2)]	40.92	平均 38.88ns
	[(P0.C0), (P0.C4)]	41.05	
測定 4	[(P0.C0), (P1.C0)]	216.14	
(1hop)	[(P0.C0), (P1.C2)]	219.27	平均 218.32ns
	[(P0.C0), (P1.C7)]	219.01	
	[(P0.C0), (P4.C0)]	218.85	
測定 5	[(P0.C0), (P3.C0)]	276.56	
(2hops)	[(P0.C0), (P5.C0)]	315.03	
	[(P0.C0), (P6.C0)]	311.67	平均 305.62ns
	[(P0.C0), (P7.C0)]	284.26	
	[(P0.C0), (P7.C7)]	340.58	

3.3 考察

CPI(Clock cycle Per Instruction)に与える影響を考察する。
まず cache coherency 時間は、同一プロセッサチップ内(on die)では平均 25.59ns(57.83cyc)、異なるプロセッサチップ間(QPI 1hop)では平均 205.03ns(463.37cyc)、QPI 2hops では平均

292.33ns(660.67cyc)である。異なるプロセッサチップ間(QPI 1hop)は、ボード内で cache coherency を処理する場合とボード間で処理する場合があるが、cache coherency 時間は両者でほとんど変わらなかった。1 命令において基本 CPI を 2.0 としたとき、cache coherency の命令あたりの発生頻度を横軸にとり、縦軸に CPI をとったグラフを図 6 に示す。

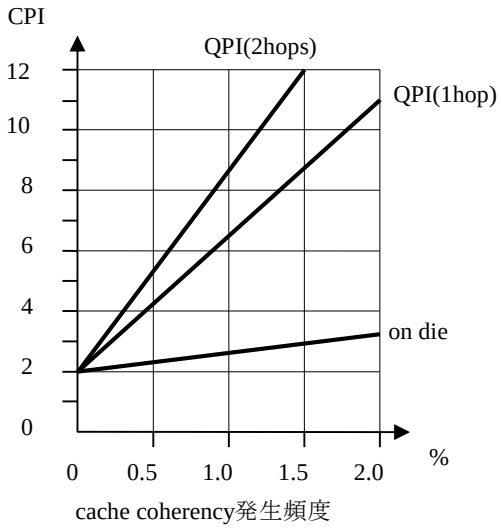

図 6　4 ボード構成における CPI に与える影響

これから分かるように、同一プロセッサチップ内で発生する cache coherency が CPI に与える影響は比較的軽微であるが、異なるプロセッサチップ間ではこの影響は極めて大きくなる。図 4 の場合と比較しても、QPI 1hop の値も悪くなっており、大規模システムの場合は cache coherency の論理回路がより複雑になることにより、特に QPI 2hops の値は極めて悪化する。

4　cache coherency 時間の改善

Intel Xeon プロセッサについて Nehalem マイクロアーキテクチャについての測定を行った。Intel はその後マイクロアーキテクチャを、Sandy Bridge、Haswell と進化させている。これらについて同一プロセッサチップ内(on die)での cache coherency 時間を測定した結果を、図 7 に示す。

図 7 を見るとマイクロアーキテクチャの進化に応じて、cache coherency 時間も改善されてきている。cache coherency 時間については公開された資料は少ないが、Intel のマニュアル[2]では Sandy Bridge の L3 の dirty hit access latency は 60cycles 以上という記述がある。Intel が定義している dirty hit access latency は、cache の内容が他のプロセッサなどによる更新により最新の状態になっていない(dirty hit)ということで最新の状態に更新するアクセス時間である。これはここで定義している cache coherency 時間そのものである。従ってプロセッサの周波数が 3.4GHz であるので 17.65ns 以上という値になり、測定値の 19.12ns とほぼ一致する。

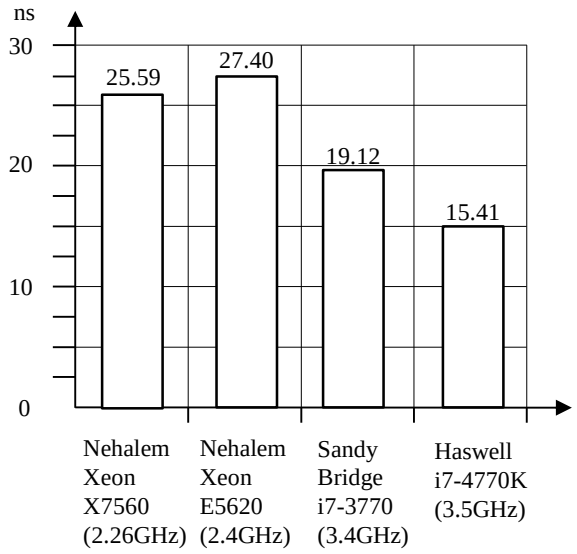

図 7　on die における cache coherency 時間

5 結論

Linux カーネルで提供されている C 言語の atomic_inc 関数を用いることにより、cache coherency の時間を実測する方法を考案し実測を行った。この方法は比較的簡単なプログラミングで小規模システムから大規模システムまで測定が可能であり、測定値についても妥当な結果が得られた。

cache coherency の時間については、プロセッサ構成と cache coherency を行うプロセッサ間の距離により大きく異なる。まずプロセッサチップが 1 つの小規模システムの場合は、cache coherency の処理がチップ内(on die)で行われるため、この時間は 15～28ns となり、CPI に与える影響は比較的軽微である。またマイクロアーキテクチャの進化に応じて、cache coherency 時間も改善されてきている。

次にプロセッサチップが 2 つで 1 ボードの中規模システムの場合は、cache coherency の情報がボード内で伝達されるため、この時間は Intel Xeon E5620(Nehalem)では平均 113.71ns となり、CPI に与える影響は大きくなる。

そしてプロセッサチップが 8 つで 4 ボードの大規模システムの場合は、cache coherency の情報がボード間でも伝達されるため、異なるプロセッサチップ間(1 hop)で平均 205.03ns となる。この場合、ボード内で cache coherency を処理する場合とボード間で処理する場合があるが、cache coherency 時間は両者でほとんど変わらない。

さらに QPI 2hops では平均 292.33ns と cache coherency 時間が極めて大きくなる。このため、CPI に与える影響は非常に大きくなる。

メモリを共有したマルチコア化の方向は今後も進むと考えられるが、コア間で共通のエリア

を更新する場合は性能に関して十分注意が必要である。
特に大規模システムの場合は、cache coherency の論理回路がより複雑になり、プロセッサ間の距離によって cache coherency 時間が大きく異なる。特に cache coherency の情報がボード間で伝達される異なるプロセッサチップ間(2 hops)の場合は非常に大きくなる。
このようにプロセッサ間の距離を意識してプログラミングを行うのは、非常な困難を伴うが、コア間で共通のエリアを更新する場合は、できる限りプロセッサチップ内(on die)で行うのが望ましく、異なるプロセッサチップ間(特に 2 hops)は避けるべきである。

謝辞　本研究の一部は、内閣府最先端研究開発支援プログラム「超巨大データベース時代に向けた最高速データベースエンジンの開発と当該エンジンを核とする戦略的社会サービスの実証・評価」の助成による。研究の機会を与えて頂いた東京大学生産技術研究所／国立情報学研究所　喜連川優教授に感謝するとともに、特にメモリを共有するプロセッサチップが 8 つの 64 コアの大規模システムの実測にご協力頂いた東京大学生産技術研究所　合田和生特任准教授に、感謝の意を表します。また内容について議論し、貴重なご意見を頂いた東京大学生産技術研究所　小高俊彦客員教授に謹んで感謝の意を表します。

参考文献

1)　Molka, D.et.al,: Memory Performance and Cache Coherency Effects on an Nehalem Multiprocessor System, 18th ICPACT, pp.261-270, 2009
2)　Intel 64 and IA-32 Architectures Optimization Reference Manual, Intel, July 2013(online).　available from <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>　(accessed 2014-2-3)