DEWS2003 5-B-01

OLTP 性能向上を目的としたメモリプロファイリングツール

吉岡　弘隆†

† ミラクル・リナックス株式会社　〒 107-0052 東京都港区赤坂４－１－３０
E-mail: †hyoshiok@miraclelinux.com

あらまし　CPU の処理速度は年率数十%で向上しているが、メモリの処理速度の向上率は CPU に比較して低い。その結果、メモリをアクセスした時のペナルティが年々増加している。特にデータベース管理システムにおけるメモリアクセスはリスト構造の検索あるいはハッシュなどが多数をしめ、科学技術アプリケーションでみられる配列の単純なアクセスに対する最適化は効果がほとんどないことが知られている。本論文はトランザクション処理（OLTP) 性能向上を目的として、パフォーマンスチューニングに必要なメモリプロファイリングツールの開発し、DBMS 等においてその効果を確認した結果を報告する。

キーワード　性能評価, ベンチマーク, キャッシュ, 最適化

A Memory Profiling Tool for Performance Tuning of OLTP Workloads

Hirotaka YOSHIOKA†

† Miracle Linux Corporation　4-1-30 Akasaka, Minato-ku, Tokyo, 107-0052 Japan
E-mail: †hyoshiok@miraclelinux.com

Abstract　The performance of CPU has been improved more than 50 % per year but the performance improvement of memory in latency is much lower than those of CPU. Therefore the penalty of memory access has been increased every year. Recent works show optimization on scientific workloads is not effective on commercial workloads like DBMSs. We have developed a memory profiling tool to collect performance information for OLTP and evaluated the effectiveness of the tool.

Key words　Performance Evaluation, Benchmark, Cache, Optimization

1. はじめに

ＣＰＵの処理速度は年率数十%で向上しているが、メモリの処理速度の向上は年率数%といわれており、ＣＰＵの向上率に比較して低い。(図 1)

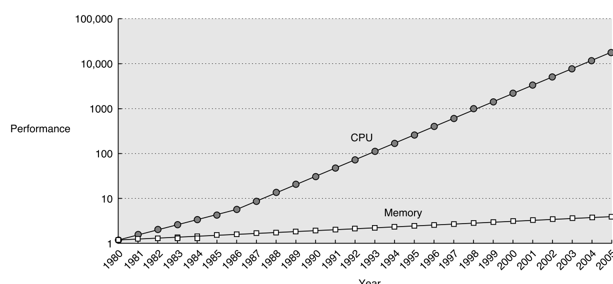

図 1　[1] 第 5 章図 2 からの引用

その結果、CPU とメモリの性能のギャップは年々広がっている。例えば、最近のプロセッサでは、メモリアクセス (キャッシュミス) のペナルティは 200 倍近くあり、メモリアクセスのコストは一定であるという仮定のもとのプログラミングモデルは破綻している。(表 1) 従って、より高性能なソフトウェアを実現にするためにメモリ階層を意識したプログラミングモデルが必要となってきている。

表 1　Intel Xeon 2GHz/Main Memory 1GB での実測値

メモリ階層	アクセスコスト
L1	1 nsec
L2	9 nsec
Main memory	196 nsec

90 年代の研究によれば、SPEC ベンチマークに代表する科学技術アプリケーションにおける CPI (Clock Per Instruction) に比べ、商用ワークロード (OLTP – On Line Transaction Processing) の CPI が大きいことが知られている。その要因の一つがメモリアクセス時のストールである。メモリへのアクセス待ちが CPI を上げるのである。([4])

特にＲＤＢＭＳ等におけるメモリアクセスは、ポインタを利用したランダムなアクセスが多く、科学技術アプリケーション

でみられる単純な配列の順アクセスに対する最適化は単純には適用できない。

そこでメモリアクセスに注目した性能向上ツールが必要とされているのだが、従来のタイマ割り込みによる PC(Program Counter) のサンプリング手法でのプロファイリングでは、キャッシュミス等のメモリの動的特性についての詳細情報を得ることができない。そのためプログラマは、おおまかなホットスポット (実行時間を多く消費している場所) の位置を推定することができても、何故そこで実行時間がかかっているかを特定することが困難であった。命令がストールしているのは、データメモリのキャッシュミスなのか？分岐予測の失敗なのか？それとも単に実行にコストがかかる命令を実行中だったのか等について情報が得られない。

さらに最近のプロセッサでは、投機的実行や out of order 実行、深いパイプラインなどにより、イベントを発生させた命令と PC の値が必ずしも一致しないため、上記のプロファイリングだけでは問題を特定することが益々難しくなっている。

Pentium 4/Intel Xeon では、上記の問題に対し、ハードウェアによるパフォーマンスモニタリングファシリティをそなえ、18 個のパフォーマンスモニタカウンタを持ち、各種メモリアクセスイベント (L1/L2 キャッシュミス/TLB ミス等) を測定できるようになっている。そして PEBS (Precise Event Based Sampling) と呼ばれる機能によって、以前のプロセッサでは不可能だった、より精密なプロセッサの状態のサンプリングが可能になった。

そこでわれわれは Pentium 4/Intel Xeon における上記の機能を利用してメモリプロファイリングツールを実装し、RDBMS (PostgreSQL) においてそのツールの有効性を検証した結果を示す。我々のツールを利用すれば従来のツールでは困難だったメモリイベント (L1/L2 キャッシュミスなど) のホットスポットを容易に発見することができた。

また、プログラムの小さな変更では、キャッシュミスが多発するプログラムとそうでないプログラムの差を厳密に見ることは、OS のタイマの精度がミリ秒以下の差を測定することが通常困難であるため難しかったが、我々のツールでは、キャッシュミスの回数を測定するので、容易にどちらのアルゴリズムがメモリアクセス特性について優れているか判定できた。そのため、アルゴリズムの抜本的な改良のみならず、OS のタイマの粒度では測定が難しい細かい逐次的改良の積み重ねなどの効果も一つ一つ確認しながら行える。

最後にプロジェクトの今後の方向および課題について述べる。

2. メモリプロファイリングツールの実装

Intel 社の 32 ビットマイクロプロセッサ (IA-32 と記す) はモデル固有のハードウェア・パフォーマンス・モニタリング機能を持つ。(付録 1. 参照) パフォーマンスカウンタ (PMC) は Pentium から実装され、さまざまなハードウェアイベントの計測を可能としている。計測できるイベントはアーキテクチャモデルによってことなる。最新の Pentium 4 および Intel Xeon プロセッサでは、18 個の 40 ビットのパフォーマンスカウンタを持ち、多くのイベントを同時に計測することができるようになった。

表 2　パフォーマンスファシリティ用パッチ例

名前	開発者名	URL
perfctr	Mikael Pettersson	[8]
brink_abyss	Brinkley Sprunt	[9]
Rabits	Don Heller	[10]
PAPI	Philip J. Mucchi, et. al	[11]

表 3　実験に使用したマシン

実験に使用したマシン	
CPU	Intel Xeon
Clock	2.0GHz
Memory	1GB
L1 cache (Data)	8KB (4 way set associative)
L1 line size	64 byte
Trace cache (Instruction)	12K μ OP
L2 cache (unified)	512KB (8 way set associative)
L2 line size	64 byte

(計測できるイベント例：分岐命令数、分岐予測失敗数、バストランザクション数、キャッシュミス数、TLB ミス数、実行命令数等々多数)

我々は、IA-32 のパフォーマンスモニタリングファシリティを利用して、メモリプロファイリングツールを Linux 上に実装した。([7])

ツールは以下のコンポーネントからなる。

（1） Linux Kernel へのパッチ

（2） ユーティリティ(xhardmeter および ebs)

（3） ユーザプログラム用 API (Application Programming Interface)

パフォーマンスモニタリングファシリティの機能は Linux ではデフォルトでは利用できないので、それを利用可能にするパッチが必要である。(表 2)

ただ上記パッチだけではイベントの設定について様々なレジスタへフラグを 16 進であたえなくてはいけないので繁雑である。そこでイベントの設定を簡単にするための GUI ツールおよびコマンドユーティリティ(ebs) 作成した。

計測する対象はユーザモード、カーネルモードどちらも可能である。

3. ベンチマークによる検証

PEBS (Precise Event Based Sampling) を利用したイベントサンプリングがどのくらい有用な情報を提供するか検証するために、以下のような Intel Xeon マシン (表 3) で実験をおこなった。

3.1 pgbench

PostgreSQL のディストリビューションに標準的に添付されている pgbench を利用して、下記 (表 4) のようなデータベースを作成し、その時の L1/L2 キャッシュミスを測定した。トランザクションのモデルは TPC-B 型の単純なモデルである。

今回の実験では十分速いディスクを用意できなかったため、scaling factor を小さくしないと、I/O ボトルネックが発生し、idle がみられた。CPU をほぼ 100 %使いきるために、scaling factor を 1 に設定してベンチマークを実行した。(下記実行例

表 4　pgbench

テーブル名	タプル数
branches	1
tellers	10
accounts	100000
history	0

参照)

pgbench の実行例

```
$ time /usr/local/pgsql/bin/pgbench -c 10 -t 1000 pgbench
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 1
number of clients: 10
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 185.972150 (including connections establishing)
tps = 186.100001 (excluding connections establishing)

real    0m53.790s
user    0m1.440s
sys     0m1.540s
```

表 5　L1 キャッシュミスの頻度と場所、PostgreSQL V7.3 2.4.18 kernel

頻度	アドレス	ルーチン名
7523	08122b67	LWLockRelease
5581	0812294b	LWLockAcquire
4795	08122967	LWLockAcquire
4750	08122b71	LWLockRelease
1992	0812293e	LWLockAcquire
1429	0816ac8e	SearchCatCache
1423	0811a2fc	ReadBufferInternal
1281	080768e4	heapgettup
1271	081803ba	HeapTupleSatisfiesSnapshot
1235	08175051	hash_search

Linux kernel 2.4.18 で上記のベンチマークを 1 回づつ実行しそれぞれのキャッシュミスを計測した結果が下記である。カーネルパッチとして abyss ([9]) を利用した。

メモリトラフィック (L1 および L2 キャッシュミス) を計測した。L1 キャッシュミスのトップ 10 および L2 キャッシュミスのトップ 10 である。(表 5、表 6)

この実験では、サンプリング間隔を 10000 と設定した。すなわちイベントが 10000 回発生するたびに PC の情報を収集した。その結果、L1 キャッシュミスは 89709 個、L2 キャッシュミスは 11475 個の PC をサンプリングした。

L1 キャッシュミスをおこしている個所のキャッシュミス回数を頻度順に並べて累積度数をグラフ化 (図 2) した。頻度 Top 10 %で約 87 %のキャッシュミスを発生している。

3.2 分　　析

PEBS によって下記のような PEBS レコードを取得した。PEBS レコードは、イベントが発生した時の各レジスタ値 (eflags, linear_ip, eax, ebx, ecx, edx, esi, edi, ebp, esp) を持つ。(PEBS レコードは通常のファイルにプレーンテキストとして出力されるので通常の Unix のコマンドで簡単に処理ができる)

これをイベントが発生した時のリニアアドレス (linear_ip) で

表 6　L2 キャッシュミスの頻度と場所、PostgreSQL 7.3 2.4.18 kernel

頻度	アドレス	ルーチン名
739	080956c1	OpernameGetCandidates
634	080e3f16	DLMoveToFront
499	080e3f0f	DLMoveToFront
476	0816ac8e	SearchCatCache
363	0816a683	AtEOXact_CatCache
335	08175051	hash_search
329	08180260	HeapTupleSatisfiesSnapshot
307	081804a0	HeapTupleSatisfiesSnapshot
251	08095700	OpernameGetCandidates
227	080e3f23	DLMoveToFront

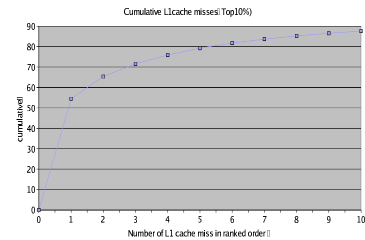

図 2　L1 cache miss の累積度数 (Top10 %)

ソートし、その回数を数えたのが、(表 5、表 6) である。リニアアドレスとそれを含むルーチン名は、オブジェクト・ファイルを逆アセンブル (objdump) することによって入手できる。また、当該部分のソースコードはコンパイル時 (gcc) に-g オプションを付加しビルドし、objdump コマンドに-S オプションを付加することによってえられる。

```
# Abyss Event-Based Sample File
#
# Date:    Mon Jan  6 00:18:08 2003
# Command: /usr/bin/time --output=./emon_data08/job.ld_miss_2L_nop.pebs_eip1/\
job.ld_miss_2L_nop.pebs_eip1.time.txt -f "percent_cpu=%P, os_seconds=%S, \
user_seconds=%U, elapsed_seconds=%e, major_page_faults=%F, \
minor_page_faults=%R, swaps=%W, exit_status=%x" \
/usr/local/pgsql/bin/pgbench -c 10 -t 1000 pgbench >/dev/null
#
# EBS Configuration:
#    EBS type:            precise
#    CCCR MSR address:    370
#    Counter MSR address: 310
#    events per sample:   10000
#    samples to buffer:   3000
#    sample type:         ALL
#    max samples:         200000
#    samples collected:   11475
#    dropped samples:     0
#    sample filename:     ./emon_data08/job.ld_miss_2L_nop.pebs_eip1/\
job.ld_miss_2L_nop.pebs_eip1.ebs_samples.txt
#
# Each sample is: {eflags, linear_ip, eax, ebx, ecx, edx, esi, edi, ebp, esp}
#
# eflags linear_ip     eax      ebx      ecx      edx      esi      edi      ebp      esp
#
00000202 40008eff 000075f0 40014dc0 400349ea 0000075f 40015920 4002a7ec bffff164 bffff08c
00000246 40008eb1 40014f68 40014dc0 400fd86b 40015180 40015180 400e82fc bffff264 bffff18c
00000246 40008eae 40015384 40014dc0 00000001 40015438 40015438 400e82fc bffff514 bffff43c
00200246 0807f0c0 00001ff0 4035291c 40352930 00000024 082850ec 00000002 bfffe900 bfffe8b8
00200206 0809f11c 00000001 0000007b bfffee68 bfffeb50 0000007b 0000007b bfffeff0 bfffea98
00200283 0816a81b 00000000 40469460 0826ad38 000000b6 00000004 4044d8c0 bfffee40 bfffee18
00200282 c012455d e11dc3bc f4a995c4 f4a995c4 c02a86b4 f47f77d4 00000000 f4a995c4 dd239ea0
00200202 40008f09 00004ca0 40014dc0 00000826 40055db8 40055db8 400dce9c bffff008 bfffef30
00200202 4000907c 4005596c 40014dc0 0805ba5e 40055920 40055930 0805ba5d bfffef78 bfffeea0
00200246 08173a45 00000000 08292258 4030c4a4 0820a120 4033c918 08292280 bfffec00 bfffebd4
00200246 40008eba 4002d220 40014dc0 0805ba5e 40015980 40015980 401fe088 bfffec48 bfffeb70
00200283 08174fb2 bfffeae8 4038d18c bfffeae8 08249990 bfffeae8 00000001 bfffe9c8 bfffe9a8
00200216 080e3f0f 404677d0 00000050 404677d0 40465680 4046f020 00000014 bfffecd0 bfffeccc
00200206 0807f2d9 08255b88 4037691c 00000001 00000081 00000036 00000003 bfffea88 bfffea40
00200206 080956c1 0825a8c0 4046d7bc 00000051 4046d762 4046d750 00000000 bfffee50 bfffee18
00200202 08080100 08255b88 0000002e 0000001a 4033e91c 08290750 0000001a bfffeac8 bfffea90
00200206 0810ac90 08295c60 bfffee98 08296378 0810ac78 08295778 bfffee98 bfffee10 bfffedf8
00200206 0809f11c 00000001 0000026c bfffee62 bfffeb44 0000026c 0000026c bfffeff0 bfffea98
00200206 080956c1 0825a850 40467b8c 00000016 40467b62 40467b20 00000000 bfffee50 bfffee18
00200282 080aa106 000006c1 0000027a bfffee6c 00001182 bfffeb5c 082902e8 bfffeff0 bfffea98
00200202 080aabf1 00000004 081af54c 081af5fc 0000002c 082901eb 081fca40 bfffea70 bfffea48
...
```

以下略

例えば、L1 キャッシュミスが多発しているルーチン (LWLock-Release) の当該命令 (アドレス：08122b67) の付近は下記である。コンパイル時 (gcc) に -g オプションをつけビルドするとデバッグ用のシンボルを作成するので、逆アセンブル (objdump -S) すると対応するソースコード読めるので便利である。

キャッシュミスが多発しているルーチンから順に最適化できないかをソースコードを見ながら検討する。L1 キャッシュミスが多発している LWLockRelease および LWLockAcquire をターゲットに最適化を考える。

```
objdump -S による出力
/* Acquire mutex.  Time spent holding mutex should be short! */
SpinLockAcquire_NoHoldoff(&lock->mutex);
 8122b40: 84 c0             test %al,%al
 8122b42: 74 1c             je   8122b60 <LWLockRelease+0x94>
 8122b44: 83 c4 fc          add  $0xfffffffc,%esp
 8122b47: 68 b1 01 00 00    push $0x1b1
 8122b4c: 68 fc eb 1d 08    push $0x81debfc
 8122b51: 56                push %esi
 8122b52: e8 55 01 00 00    call 8122cac <s_lock>
 8122b57: 83 c4 10          add  $0x10,%esp
 8122b5a: 8d b6 00 00 00 00lea  0x0(%esi),%esi

/* Release my hold on lock */
if (lock->exclusive > 0)
 8122b60: 8a 46 02          mov  0x2(%esi),%al
 8122b63: 84 c0             test %al,%al
 8122b65: 7e 0a             jle  8122b71 <LWLockRelease+0xa5>
  lock->exclusive--;
 8122b67: 8a 46 02          mov  0x2(%esi),%al
 8122b6a: fe c8             dec  %al
 8122b6c: 88 46 02          mov  %al,0x2(%esi)
 8122b6f: eb 07             jmp  8122b78 <LWLockRelease+0xac>
else
{
  Assert(lock->shared > 0);
  lock->shared--;
 8122b71: 8b 46 04          mov  0x4(%esi),%eax
 8122b74: 48                dec  %eax
 8122b75: 89 46 04          mov  %eax,0x4(%esi)
}
以下略
```

LWLock は Light Weight Locks で通常は共有メモリ上のデータ構造をロックするために利用される。spinlock を利用して実装されている。spinlock は下記のような実装になっている。TAS (test and set) はアトミックにロック変数へ値をセットするマクロである。インテルプラットフォームでは lock xchg 命令で実装している。

ここで TAS() は、アトミック性を保証するためにバスを lock(占有) し、かつメインメモリにアクセスしているために非常にコストが高いことに注意をしないといけない。ロックを保有しているプロセスがロックを解放するまで spin loop するのだがそのループは毎回バスを占有しチェックをするので、(つまり LWLock によって「メモリバス」が占有される)、マルチプロセッサ環境においてスケーラビリティ上の問題がある。

```
s_lock(volatile slock_t *lock, const char *file, int line)
{
...
        while (TAS(lock))
        {
            if (++spins > SPINS_PER_DELAY)
            {
...
                (void) select(0, NULL, NULL, NULL, &delay);
                spins = 0;
            }
        }
}
```

そこで、while(TAS(lock)) ループの内側にもうひとつループ (whicle(*lock)) をもうけて最適化をはかる。

```
s_lock(volatile slock_t *lock, const char *file, int line)
{
...
        while (TAS(lock)) /* TAS はバスを占有しコストが高い */
        {                 /* 下記の while() を追加した */
            while (*lock) /* キャッシュへのアクセス */
            {
                if (++spins > SPINS_PER_DELAY)
                {
...
                }
            }
        }
}
```

この実装では、ロックの監視をコストのかかる TAS で毎回おこなうのではなく、キャッシュへのアクセスへと変更している。キャッシュの監視はアトミックな test and set ではないので、ロックがリリースされた後 (内側のループを抜けた後)、再度、外側のループで test and set をアトミックにおこないロックの確保を試みる。内側のスピンループ中はメインメモリにアクセスしないので、メモリコンテンションが下るのでスケーラビリティの向上がみこまれる。

この非常にシンプルな改良で L1 cache miss にどの程度の影響があったか確認してみた。これは `pgbench -c 10 -t 1000` というコマンドを 4 回実行した時のキャッシュミスの回数である。先の実験と同様、10000 回ごとにサンプリングし、280759(オリジナル版)、265927 個 (改良版) のデータを収集した。(表 7 および表 8)

なお先の実験のとき (表 5) と今回の L1 cache miss(表 7) の傾向が若干異なるが、これは、先の実験では、コマンドの実行回数が 1 回だけで、試行の回数が異なるためである。

表 7　L1 キャッシュミスの頻度と場所 (オリジナル)、PostgreSQL 7.3 2.4.18 kernel

頻度	アドレス	ルーチン名
15551	08122b67	LWLockRelease
15429	08122967	LWLockAcquire
14558	0812294b	LWLockAcquire
12635	08122b71	LWLockRelease
5683	081803ba	HeapTupleSatisfiesSnapshot
5556	0816ac8e	SearchCatCache
4917	080768e4	heapgettup
4653	0807ee60	_bt_isequal
3776	08175051	hash_search
3736	0811a752	write_buffer

表 8　L1 キャッシュミスの頻度と場所 (改良版)、PostgreSQL 7.3 2.4.18 kernel

頻度	アドレス	ルーチン名
13771	0812294b	LWLockAcquire
13198	08122b67	LWLockRelease
13098	08122967	LWLockAcquire
10473	08122b71	LWLockRelease
5695	081803ca	HeapTupleSatisfiesSnapshot
5417	0816ac9e	SearchCatCache
5060	080768e4	heapgettup
4210	0807ee60	_bt_isequal
3914	08175061	hash_search
3821	08095672	OpernameGetCandidates

これによれば、該当ルーチン (LWLockAcquire および LWLockRelease) において、約 15 %ほど L1 cache miss が減少 (改善) していることを確認できた。全体でみると L1 cache miss の減少は約 5.3 %、TPS (Transaction Per Second) で約 2.7 %の向上が見られた。L1 cache miss を減少させることが実行性能 (TPS) を向上させることにつながった。

さらなる性能向上には、より詳細な検討が必要になる。今回の測定はシングルプロセッサ環境でおこなったので、ロックの確保と解放は実際にはロックを獲得しているプロセスへのコンテキストスイッチが発生したタイミングでおこなわれるためメモリコンテンションは問題とならない。

postgres の実装では、ロックをスピンループで待っていた場合、あらかじめ設定していたループ回数をまわった時点で、select システムコールによってボランタリにコンテキストスイッチを発生させて CPU を解放している。今回の改良ではスピンループをより速くまわるので結果として積極的にコンテキストスイッチを発生させることによってスループットを向上させている。

上記の改良を行った結果、SMP 環境では、スピンで待機しているプロセスは内側のループでキャッシュを監視するため、バスを確保したり、メインメモリにアクセスしないので、他の CPU のメモリアクセスの妨げにならないようになった。これはスケーラビリティを向上させることになる。(バスのコンテンションが減るため)

今回はデータキャッシュについて測定し、命令キャッシュについては議論しなかったが、命令キャッシュの動的特性に関しても、同様に計測できる。そのような情報を入手できるので命令キャッシュミスがおきにくいようなプログラムの再配置等についても検証できる。

4. 考　　察

ここでは、Hennessy および Petterson([1]) にならって、キャッシュミスの原因を次の 3 つに分類する。

- 初期ミス – 最初にアクセスする時に発生するキャッシュミス
- キャパシティミス – キャッシュサイズに比べてワーキングセットが大きい時に発生する
- コンフリクトミス – あるキャッシュラインにアクセスが集中することによって発生する

メモリプロファイリングによって、プログラムのどの個所でキャッシュミス等が発生しているか容易に同定できる。

そして、メモリイベントが発生した時のレジスタの値、メモリアドレスの情報を入手しているので、それをもとに、上記のどれが原因でキャッシュミス等が発生しているかを特定できる。

Intel Xeon の場合、L1 キャッシュは 8KB、4 ウェイセットアソシアティブ、L2 キャッシュは 512KB、8 ウェイセットアソシアティブなので、それぞれ 4 つないし 8 より多くのアドレスが 2KB(=8KB/4) ないし 64KB(=512KB/8) のモジュロに等しい場合、コンフリクトミスが発生する。コンフリクトミスは他のキャッシュラインが空いていても発生するので注意が必要である。

例えばアドレスを 2KB で割った余りが等しい場合、それらは同じキャッシュラインの一つにアサインされる。この時、アドレスの下位 6 bits (64 bytes 分) は、どのキャッシュラインにのるかを決定しない。結局下位 6 bits を無視したアドレスを 2KB で割った余りが等しい場合、それらは同じキャッシュラインの一つにのる。すなわち下記の A の部分が等しいアドレスの場合、同じキャッシュラインの一つにアサインされる。(x は don't care)

```
xxxx xxxx xxxx xxxx xxxx xAAA AAxx xxxx
```

その時、4 ウェイセットアソシアティブキャッシュには同時に 4 つのキャッシュラインしか保持できないので、それ以上のデータをキャッシュしようとすると、同じキャッシュラインのどれかと交換しなければならなくなる。すなわちアドレスの bit 位置 10 から bit 位置 6 までが等しい場合にコンフリクトミスが発生する可能性が高い。他のキャッシュラインが空いていてもアクセスするアドレスによって、このコンフリクトが発生する。

例えばキャッシュコンフリクトの場合、カラーリングとして知られる手法によって、コンフリクトを減らすことができる。Linux Kernel 2.4 の例でスケジューラがコンフリクトミスを多発していることを発見した山村ら ([13]) は、キャッシュのカラーリング手法を利用してスケジューラの L2 キャッシュミス率 (= (L2 キャッシュミス) ／ (L2 キャッシュアクセス)) を 85 %～90 %から 3 %～14 %ほどに削減した。

キャパシティミスか、初期ミスかは、ソースコードを確認することによって判断できる。同じアドレスがよくキャッシュミスを発生している場合はキャパシティミスをうたがってみる。

キャパシティミスに関しては、ブロッキングとして知られている方法によって、ワーキングセットを減らせる場合がある。

初期ミスに関してはプリフェッチをおこなうことによってレイテンシを下げることができる場合がある。あるいはデータ構造を工夫して、データサイズを減らし、一回のアクセスでいくつかの関連するデータを一度にキャッシュラインにのせることなどで対処できる。

いづれにせよメモリプロファイリングによって選られた情報とソースコードを分析することによってキャッシュミスを削減することが可能である。

5. 関連する研究

5.1 パフォーマンスモニタリングファシリティ

インテルの IA-32 のパフォーマンスモニタリングファシリティを利用できるようにしたものとして、表 2 がある。perfctr [8] は P6 および Pentium 4/Intel Xeon に対応している。しかし PEBS には未対応である。abyss [9] は Pentium 4/Intel Xeon には対応しており PEBS にも対応しているが、P6 は対応していない。また Linux Kernel 2.4 系のみに対応している。rabbit [10] は、P6 だけに対応しており、PEBS や Pentium 4 には対応していない。また最近メンテナンスされていないようである。PAPI [11] はマルチプラットフォームなパフォーマンスライブラリである。パフォーマンスモニタリングファシリティの機能については、

perfctr [8] を利用しているが、Pentium 4 および PEBS には対応していない。以上のツールは全てオープンソースで、インターネット上で入手できる。PEBS に対応しているのは、現状では abyss [9] だけである。

われわれは、abyss [9] および perfctr [8] をベースに、PEBS 対応、Linux Kernel 2.4/2.5 対応のドライバおよび設定を容易にする GUI ツールなどを開発中である。

インテルから VTune という製品が出荷されているが、オープンソースでないため、機能のカスタマイズなどができない。

5.2 DBMS および OLTP workload とメモリ階層

従来キャッシュの動的特性については、SPEC ベンチマークに代表される科学技術計算分野では、多くの研究成果がある。([16], [17])

一方 DBMS 関連の商用ワークロード (OLTP など) については、近年研究成果が報告されつつある。([4] など) しかしながら、その成果はワークロード全体でのメモリトラフィックの傾向 (キャッシュミス率や CPI (Cycle per Instruction)) 等の報告がほとんどである。

個々の実装の、どこの部分での、どのくらいのメモリトラフィックがあるか、キャッシュミスのホットスポットがどこにあるか等のミクロな動的特性まで議論したものは少ない。

今回提案した方法は、IA-32 のパフォーマンスモニタリングファシリティを利用した実装なので、特別なハードウェアは一切必要なく、しかもプロセッサが提供する機能を利用するため、シミュレーションによる方式と違って、実時間での計測が可能である。またアプリケーションの変更を必要としない。

キャッシュミスなどの動的特性の分析は、SPEC ベンチマークに代表される科学技術計算に対するものが多かったが、IA-32 の機能を利用することによって、データベース管理システムやカーネルに代表される動的特性がまだ十分知られていない問題に対しても分析が可能となった。

また、Pentium 4/Intel Xeon で新たに実装された PEBS の機能を利用すれば、容易にメモリ特性のボトルネックを同定できる。そしてそのメモリボトルネックがなぜ発生しているかの特定も可能である (レジスタ値を分析することによって、詳細な検討が可能になった)。そしてそのような要因が特定できれば、対応する解決策についても、従来知られている方法で対処することができるようになった。

6. 今後の方向と課題

Pentium 4/Intel Xeon プロセッサが持つパフォーマンスモニタリングファシリティを利用したメモリプロファイリングツールについて報告した。プロジェクトの今後の方向、課題などを述べる。

6.1 実装そのものの評価

メモリプロファイリングツールを実行させることによる、命令キャッシュ(トレースキャッシュ)、およびデータキャッシュの影響の評価が必要である。

PEBS によって精密なメモリトレースが取得できたが、サンプリング用ドライバがデバッグストア内にある PEBS レコードをユーザ空間にコピーする時にも当然、メモリアクセスが発生する。それによって L1/L2 キャッシュを利用するので、影響が生じる。ドライバが実行されることによる命令キャッシュ(トレースキャッシュ) の影響の評価も必要である。

一つの方法として非常に大きな空間をカーネルのデバッグストアに割りあてておき、サンプリング中は DS 領域がオーバーフローしないようにしておくことによりドライバを起動しないようにするというのが考えられる。この方式をとれば命令キャッシュへの影響はさけられる。

あるいは、サンプリング間隔を十分長くとることによって、サンプリングによる影響を相対的に低くすることが考えられる。例えば今回は 1 万回イベントが発生するたびにサンプリングし、数万レコード採取した。ベンチマークの実行時間を長くできるのであれば、サンプリング間隔を長くすることができる。

また、ツールの実行時のオーバヘッドも検証しないといけない。これはツールを動かした時と、動かさない時の実行時間の差を測定する。PEBS による詳細情報のサンプリング、イベント回数だけの測定、ツールを一切動作させない場合、それぞれについて、タイムスタンプカウンタにより、クロック数を計測し評価する。クロック数の差によって、ツールのオーバヘッドを推定する。

予備的な実験では、ツール起動による顕著なオーバヘッドは確認できなかった。

6.2 キャッシュコンシャスなアルゴリズムの適用

今回はメモリプロファイリングツールの評価のみを行なったが、いろいろなキャッシュコンシャスなアルゴリズムを実装し、評価する必要がある。

プログラムの機械的な改良、あるいは逐次的な改良ではコンスタントオーダーの改良しか期待できないが、アルゴリズムの改良は計算量のオーダーが変更する場合がある。

また従来のアルゴリズムの評価基準は主に計算量であった。しかし今まで議論したように、キャッシュにヒットするかしないかで最大数百サイクルの差がでてくる。計算量だけではなくて、メモリ階層を考慮したアルゴリズムの評価が重要になってくる。例えば、従来よく知られているアルゴリズムなどをキャッシュを意識した実装とそうでない実装などの比較、評価が必要となってくる。

6.3 各種ベンチマークの実施

今回 pgbench (TPC-B like なベンチマーク) を実施し、チューニングをおこなったが、今後はより一般的な OLTP (TPC-C) ないし DSS (TPC-H) 等のベンチマークを実施し、ツールが実際のアプリケーション環境において、チューニングに必要な十分な情報を提供しているかを確認する。

さらに、原因の特定などを行ない実験的なチューニングを実施し、再度ベンチマークを行ない、チューニング前後の差異を容易に認識できることを確認する。

6.4 チューニングのパターン化

PEBS によってイベントが発生した時の詳細な情報が入手可能となった。それによってキャッシュミス時の要因もある程度特定できるようになった。そこで、各要因に関しての対処方法

などをまとめれば一つのチューニング方法論になる。

High Performance Computing(HPC) の分野で知られている高速化のノウハウをパターン化し、メモリプロファイリングツールで発見される現象と関連付ける。

例えばメモリアドレスが、2KB のモジュロで等しければコンフリクトミスが発生するので、その場合は、既知の方式の何々でチューニングするとかいうノウハウをまとめることが必要である。

しかし RDBMS は、HPC のような配列に対するループによるアクセスがほとんどなくて、線型リストをポインタで手繰るような処理が多いので、単純には適用できないものが多いと推定される。

7. まとめ

CPU とメモリ処理速度の向上率の差から、今後益々メモリ構成を意識したプログラミングが重要になってくる。しかしながら従来はメモリアクセスのコストの差を簡単に指摘するツールがなかったためメモリアクセスに優しいプログラムを書くことが難しかった。

今回開発したツールを利用すれば、簡単にユーザーにメモリ関連の動的特性に関する情報を提供できる。

7.1 謝辞

本プロジェクトは平成 14 年度未踏ソフトウェア創造事業 (プロジェクトマネージャー喜連川優東京大学教授)「OLTP 性能向上を目的としたメモリプロファイリングツール」として支援を受けている。

また、今回開発したスクリプト等は下記の URL で公開している。またメーリングリストなどもあるのでご活用いただければ幸いである。http://sourceforge.jp/projects/hardmeter/

文献

[1] Hennessy, J. L., and Patterson, D. A., Computer Architecture: A Quantitative Approach, 3rd Edtion, Morgan Kaufmann Publishers, 2002

[2] Intel, The IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide, Order Number 245472, 2002

[3] Hinton, G. et. al, "The Microarchitecture of the Pentium 4 Processor", Intel Technology Journal, Q1 2001 Issue, February 2001

[4] Ailamaki, A. et. al, "DBMSs On A Modern Processor: Where Does Time Go?", Proceedings of the 25th VLDB Conference, 1999

[5] Dean, J. et. al, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-Of-Order Processors", Proceedings of Micro-30, December, 1997

[6] Sprunt, B., "Pentium 4 Performance Monitoring Features", IEEE Micro, July-August, 2002,

[7] 吉岡弘隆, "Intel 系 (IA-32) プロセッサのパフォーマンスモニタリングファシリティを利用したメモリプロファイリングツール", 第 44 回プログラミングシンポジウム、箱根、2003 年

[8] http://www.csd.uu.se/mikpe/linux/~perfctr/

[9] http://www.eg.bucknell.edu/~bsprunt/

[10] http://www.scl.ameslab.gov/Projects/Rabbit/

[11] http://icl.cs.utk.edu/projects/papi/

[12] Ingo Molnar, ultra-scalable O(1) SMP and UP scheduler http://www.uwsg.iu.edu/hypermail/linux/kernel/0201.0/0810.html

[13] 山村周史他, "エンタープライズ向けプロセススケジューラの評価および改良", Linux Conference 2002 年

[14] Intel, Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, Order Number 248966, 2002

[15] Sears, C. B., "The Elements of Cache Programming Style", Proceedings of the 4th Annual Linux Showcase and Conference, 2000

[16] Cantin, J. F., et al, " Cache Performance for SPEC CPU2000 Benchmarks", http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/

[17] Gee, J. D., et al, "Cache Performance of the SPEC92 Benchmark Suite", IEEE Micro (August) 17-27, 1993

付録

1. IA-32 のパフォーマンスモニタリングファシリティ

Intel 社の 32 ビットマイクロプロセッサ (IA-32 と記す) はモデル固有のハードウェア・パフォーマンス・モニタリング機能を持つ。ここでは簡単にその機能について紹介する。([2])

パフォーマンスカウンタ (PMC) は Pentium から実装され、さまざまなハードウェアイベントの計測を可能としている。計測できるイベントはアーキテクチャモデルによってことなる。最新の Pentium 4 および Intel Xeon プロセッサでは、18 個の 40 ビットのパフォーマンスカウンタを持ち、多くのイベントを同時に計測することができるようになった。(計測できるイベント例：分岐命令数、分岐予測失敗数、バストランザクション数、キャッシュミス数、TLB ミス数、実行命令数等々多数)

タイムスタンプカウンタ (TSC) は、ハードウェアリセット時に 0 から開始し、プロセッサのクロックサイクル毎に増加する 64 ビットのレジスタである。RDTSC 命令によって、カウンタの値を読む。(非特権命令)。ユーザーモードからも読めるので、簡単に実行時のクロック数を計測するのに利用できる。例えば、あるルーチンの実行コストは、入口と出口で TSC を読み、その差が実行コスト (サイクル数) である。

```
/* rdtsc 命令の利用例*/
#define rdtscll(val) \
     __asm__ __volatile__("rdtsc" : "=A" (val))

unsigned long long before;
unsigned long long after;

rdtscll(before);
/* 計測する部分*/
rdtscll(after);

diff=after-before; /* 実行クロック数 */
```

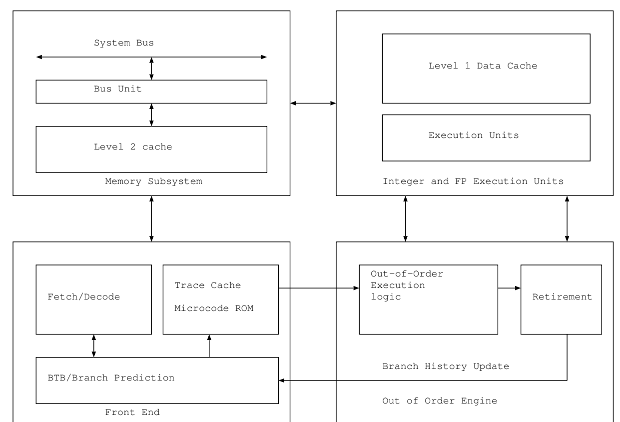

図 A· 1 Pentium 4 ブロックダイアグラム

2. Pentium 4/Intel Xeon 系 (NetBurst アーキテクチャ)

パフォーマンスカウンタは Pentium から実装されたが、以下に示すようにいくつかの限界があった。

- 同時に計測できるイベントが少ない (Pentium III で 2 つ)
- キャンセルされた命令のイベントも計測する
- イベントサンプリングの精度 (粒度) が荒い

Pentium III(P6 アーキテクチャと呼ばれている) 系プロセッサは投機的な実行をおこなう。したがって分岐予測がはずれた場合、命令をキャンセルするが、キャンセルされた命令が引き起こしたイベントもカウントする。予測がはずれた側の命令が引き起こしたイベント (例えば L2 キャッシュミス) の回数も数えてしまう。この場合、当該イベントが多数発生するのは、分岐予測がはずれたのが主な原因なのか、それとも、L2 キャッシュミスを多発させるようなプログラムなのかは、この情報だけでは判断できない。プログラムを改良する時、分岐予測がはずれないように改良するべきか、それとも L2 キャッシュミスを減らすような改良をするべきか、どちらにプライオリティを置くかなどが判断できないのである。

リタイヤ (コミット) した命令によって発生したイベントとキャンセルした命令によって発生したイベントを明確に分離できなければならない。

最近のプロセッサは、(1) 深いパイプラインを持つ、(2) 投機的な実行をする、(3) スーパースカラで同時に複数命令を実行する、(4) アウトオブオーダー実行する、などの特徴を持つ。このため、イベントサンプリングをする場合、ある回数毎の例外処理ルーチンが起動され実際のプログラムカウンタ (PC) や各種レジスタの情報を収集する時点では、例外処理ルーチンのレイテンシおよび上記のプロセッサの特性により、取得した PC の値は実際のイベントを発生させた PC よりかなり先を行っている。 Dean ([5]) らの報告によると、PentiumPro で、取得した PC と実際の PC は 25 命令以上隔たって分布した。Sprunt ([6]) の報告によれば Pentium 4 では 65 命令以上実際の命令と隔たってサンプルされた。プロセッサがより深いパイプラインを持つようになればなるほど、この隔たりは広くなると予想される。

このようにイベントの発生の場所を正確に特定できないため、イベントが発生している時点の正確なコンテキストを入手できない。イベントが発生していることはわかっても、正確なコンテキストがわからないので、性能上何が問題か特定することはこれらの情報だけでは困難である。

イベントの発生の正確な特定と、その時点での精密なコンテキストの入手が求められている。

上記の問題を解決するために、Pentium 4/Intel Xeon プロセッサでは下記のような拡張を行っている。

2.1 多くのパフォーマンスカウンタ

次のようなパフォーマンスモニタリングファシリティを持つ。

- ESCR (Event Selection Control) 45 個
- PMC (Performance Monitoring Counter) 18 個
- CCCR (Cunter Configuration Control) 18 個
- DS (Debug Store)

P6 系プロセッサに比べ、同時に計測できる PMC の数が、2 個から 18 個に増えている。ESCR に、計測すべきイベント (例; L1 cache miss/TLB miss など) を、CCCR に計測方法 (イベントのフィルタリング、割り込みの制御方法など) と計測すべきイベントを設定した ESCR を指定し計測を開始するとイベント数が PMC に格納される。

2.2 At-Retirement 計測

At-Retirement 計測というのは、実際にコミットされた命令にまつわるイベント (non-bogus ないし retire と呼ぶ) と、投機的に実行されコミットされなかった命令 (最終的にキャンセルされた命令) に関するイベント (bogus と呼ぶ) にタグをつけ、命令の retirement 時に区別する機能を指す。(投機的に実行した命令を確定することを retirement するという)

パフォーマンスカウンタは bogus ないし non-bogus を区別して計測できる。

従来は、実際にコミット (リタイヤ) した命令のみならず、キャンセルした命令によって引きおこされたイベントもカウントしてしまっていた。

より粒度の細かい計測を可能にするために、Pentium 4/Intel Xeon プロセッサでは、イベントにタグを付け、コミットした命令だけ (あるいはキャンセルされた命令だけ) を計測する機能を提供した。この機能のことを At-Retirement 計測と呼ぶ。

2.3 Pentium 4/Intel Xeon における精密なイベントサンプリング

最近のプロセッサの特徴のため、サンプリングしたプログラムカウンタ (PC) と実際にイベントを発生させた PC が一致しないという問題があった。([5]、[6])

この問題を解決するために Pentium 4 系プロセッサで精密なイベントサンプリング (PEBS – Precise Event Based Sampling) という機能が実装された。

カーネル空間にデバッグストア (DS – Debug Store) 領域というのを確保できる。パフォマンスカウンタが PEBS 用に設定されている場合、カウンタのオーバフローが発生する都度、プロセッサは汎用レジスタ、EFLAGS レジスタ、インストラクションポインタを DS 領域にある PEBS バッファの PEBS レコードに自動的に格納する。これはマイクロコードによって行なわれる (ハードウェアが自動的に実行する) ので、プロセッサの精密な状態を保存が可能となっている。そしてプロセッサはパフォーマンスカウンタの値をリセットし、カウンタをリスタートする。DS 領域に閾値を設定しておくと、それを越えるレコードが格納された時、PMI 割り込みが発生するので、DS 領域のデータをユーザー空間に退避するようなデバイスドライバを用意しておく。

PMC は 40 ビットなので、2^{40} 回イベントを計測するとオーバーフローする。これを利用して、PMC にあらかじめ 2 の補数をセットしておけば、その回数毎にイベントをサンプリングできるのである。例えば、-100 を PMC にセットすれば、100 回目にオーバフローが発生するので、PEBS の機能 (ハードウェアの機能) によって、イベントが発生したプログラムカウンタ (PC)、各種レジスタの値などを保存される。これをパフォーマンスイベントサンプリングと呼ぶ。

PEBS は At-Retirement イベントのサブセットだけをサポートしている。