DBMSにおけるスケーラビリティーボトルネックの分析

堀川　隆
NEC 共通基盤ソフトウェア研究所

CPU資源が強化されたマルチコアのマシンでは、排他制御のためのロック（論理資源）がボトルネックとなることがある。このような状況では、ボトルネック特定が困難な場合が多く、チューニングなどの対応を効率的に行うための体系的立てた方法論が求められている。本論文では、イベント・トレース手法の応用により論理資源のボトルネックを特定する手法を提案する。データベース処理のベンチマークプログラムに対して提案手法を適用することで、マルチコア・マシンの処理時間に大きく影響していたロックを特定できた。更に、最大のボトルネックとなっていたロックについて、プログラム構造の見直しによるチューニングを行うことで、スループット性能の向上を実現でき、提案手法の有用性が裏付けられた。

An Analysis method for Scalability Bottlenecks and Case Study on Database Management Systems

Takashi Horikawa
Common Platform Software Research Laboratories, NEC

Identifying performance bottleneck of a system is a crucial first step for efficient bottleneck solution. Recent multi-core CPU based IT systems, however, often have a bottleneck that is difficult to identify, such as those resulting from contention about such logical resource as a lock used for mutual exclusion. In order to address this situation, this paper proposes a well defined methodology for identifying bottlenecks based on event tracing methodology. The proposed method has been applied for an IT system using multi-core CPUs in executing database-benchmark program and has succeed to identify its bottleneck due to lock contentions. Furthermore, throughput performance of the system has succeedingly improved by tuning database management program to relieve the biggest bottleneck identified by the proposed method. These results clearly proves that the proposed method is useful in addressing bottlenecks derives from logical resources.

©2009 Information Processing Society of Japan

1．はじめに

近年、マルチコア・プロセッサの普及によってITシステムのCPU資源は強化されており、性能問題は自然に解消されるとの期待もある。しかしながら、マルチコアを活用してシステム性能を向上させるには、複数プロセッサによる効率的な並列処理の実現が必須であり、並列処理の効率を阻害する要因（システム性能上のボトルネック[1]）の解消は避けて通れない課題と考えられる。

ボトルネック解消の第一歩は、ボトルネックの特定である。ITシステムの場合、ボトルネックの可能性は多岐に渡っていることに加え、アプリケーションの処理内容やシステムとの相性等によって発現の様子は異なってくることから、ボトルネック特定は困難を極めることが多い。特に、マルチプロセッサによる並列処理特有のボトルネックである『並列化できない順実行部分』については、物理資源（CPU, disk, network, memoryなど）の使用率測定や実行プロファイル採取といった従来手法では判明しないため、ボトルネック特定を効率的に実施できる手法が必要とされている。

『並列化できない順実行部分』は、主にdoループやサブルーチンを単位とする粒度の細かい並列性を利用する科学技術計算の分野で意識されることが多いが、プロセスやスレッドを単位とする粒度の粗い並列性を利用するITシステムの領域にも存在する。典型例は、排他制御などによりアトミック性を確保して行う必要のある処理（クリティカルセクション）である。

一般には、並列処理の粒度が粗くなるほど処理間の干渉が少なくなり、効率的な並列実行が可能となる傾向にある。しかし、複数のプロセスやスレッドが多数のデータを共有して行う処理の場合、それらのデータへのアクセスを制御するためのクリティカルセクションの影響が大きくなり、効率的な並列実行が阻害されることがある。ITシステムの重要な土台であるシデータベース管理システム（DBMS）は、このボトルネックが発現する典型例といえる。

本稿では、マルチコア・システムにおいて効率的に稼動するソフトウェアを構築するための方法論として、まず、実働システムの測定・分析からボトルネックを特定する手法を提案する。次に、これをDBMSのベンチマーク・プログラム実行時のシステム動作に適用し、ボトルネックを特定する。更に、ボトルネックを軽減する改造を行い、効果を確認することで、方法論としての有用性を確認する。

2．マルチプロセッサ特有のボトルネック

2．1　排他制御を実現するメカニズム

排他制御を実現するメカニズムとして多用されているのはロックである。その基本操作は獲得と解放であり、各々、競合の有無に応じて異なる振る舞いを示す。すなわち、ロック獲得操作では競合の有無により、また、ロック解放操作ではロック待ち処理の有無により動作は異なる。

性能に大きく影響するのは、ロックを獲得できなかったスレッドがロック解放を待ち合わせる方法である。待ち合わせの方法には表１に示す２種類があること、各々の方式には一長一短あること、および、両者を組合せたadaptive lock（最初はbusy waitし、その間にロックが解放されなければsleep）も用いられていること、は広く知られている。Adaptive lockでは、busy waitする時間も性能にとって重要なパラメータとなるが、現状では、この設定はheuristicによって行うようになっていることが多い。

2．2　DBMSにおける排他制御

DBMSにおける排他制御は、大別すると、トランザクションのACID[2]性を実現するためのロック[3]とDBMS内部リソースを保護するためのラッチの２種類がある。両者の違いとしては、1)ロックはアプリケーションからDBMSへの処理要求（SQL）に関係しているため、この見直しによってある程度はチュ

表1　ロック待ち方法の比較

	概要	Pros	Cons
Busy wait	CPUを使ってロック解放を監視する	ロック解放に対する反応が早い	ロック待ち時間に伴ってCPU消費が大きくなる
Block	Sleepしてロック解放を待つ	CPU消費は、ロック待ち時間に依存しない	ロック獲得スレッドにロック待ちを通知する必要がある ロック解放時はwake up操作が入るため時間がかかる

[1] 本論文では、「システム性能上のボトルネック」を、単に「ボトルネック」と表記する。

[2] Atomicity, Consistency, Isolation, Durability

[3] 本節のみロックをDBMS用語として使用する。

©2009 Information Processing Society of Japan

ーニング可能、2)ラッチはDBMSの内部処理に関係しているため、これを直接チューニングすることは困難、という点が挙げられる。

しかしながら、いずれも、排他制御のために獲得と解放を行うこと、競合した場合はロック（ラッチ）待ち処理とロック（ラッチ）解放を通知する処理が行われる、という点では共通している。本稿では、特定のベンチマーク・プログラム（DBT1）を実行する際のロックおよびラッチ競合を扱うことから、以降の節では簡単のため、ロックとラッチを区別することなく、単に「ロック」と表記する。

排他制御のバリエーションとして、保護対象データの読み込み操作については同時実行を可能とするための Read-Write Lock（以降、rw_lock と表記）や、並行実行する処理の数を制限するためのもの（以降、conc_lock と表記）がある。前者は、Readers and Writer Lock とも呼ばれており、書き込み操作のみが他の書き込み操作や読み込み操作との排他実行を必要とする、というものである。後者は、主に性能上の観点から、排他制御区間を実行する処理（スレッド）数を制限する目的で用いられる排他制御である。

２．３　ベンチマーク・プログラム（DBT-1）実行におけるボトルネック

DBT-1[1]は、OSDL（Open Source Development Labs.）が開発した Database Test Suite の１つで、オンライン書店を想定し、Web アプリケーションのトランザクションをシミュレートするテストを行うベンチマーク・プログラムである。これは、web e-Commerce をモデルとした TPC-W[2]に準じて実装されているが、その仕様を完全に満たすものではない。また、オープンソース DBMS の性能を、DBT-1 のようなオープンなベンチマーク・プログラムによって測定した結果や考察が、情報処理推進機構（IPA）からオープンソース情報データベース（OSS iPedia）として公開されている。

ここでは、OSS iPedia の中で、DBMS を mysql（ストレージエンジンは InnoDB）、ベンチマークを DBT-1 として、コア数が４と８の両ケースについて測定および考察した結果[3]に着目する。このレポートには、1) mysql 5.0.24 ではコア数４が性能ピークであり、コア数を８に増加すると性能（スループット）が劣化する、2) mysql 5.0.32 では、ロックの実装が改良されており、1)の性能問題は解消している、という結果が示されている。しかしながら、mysql 5.0.32 の場合でも、8 コアの性能は 4 コアの場合と同程度であり、コア数に見合った性能向上が達成されているわけではない。

上記の測定結果は、ロックがボトルネックになっている可能性を示していると考えられることから、本論文では、DBT-1 を mysql で実行させたときのロック待ち状況を取り上げ、提案手法による測定・分析を実施した。

３．測定・分析方法および結果

稼働システムの測定・分析により、ボトルネックを特定する方法として、イベント・トレースのフレームワーク[4]を基礎とする手法を提案する。具体的には、物理リソース（CPU, disk, network）に関する振る舞いを検出するためのカーネル・プローブと、ロック競合検出や実行している DBMS 処理を検出するためのアプリケーション・プローブを併用する測定によって得られたトレース・データを分析し、ボトルネックを特定する方法である。

３．１　測定対象イベントとデータ分析

フレームワークでは、リソースに対する要求、利用開始、利用終了、待ち開始を採取対象イベントとして規定しているが、ロックに関しては、採取対象を「待ち開始」イベント（以降、ロック待ち開始イベント）と待ち状態解消による「利用開始」イベント（以降、待ち終了イベント）に限定することにした。競合がない状況でのロック操作（獲得・解放）まで測定対象に含めると測定オーバーヘッドが大きくなり、稼働状況を反映した結果が得られなくなると考えたためである。

また、実行中の DBMS 処理を特定するために、処理開始と処理終了イベントも測定対象イベントに含めた。これは、測定対象の処理を一種のリソースと考えた場合の利用開始および利用終了を意味するイベントと考えることができる。

トレース・データ分析では、ロック待ち開始イベントおよび対になるロック待ち終了イベントから、ロック待ちの経過時間、その間の CPU 使用時間、および、ロック待ち回数を求め、これらをロックの種類別に分類した。更に、実行中の DBMS 処理を特定するためのイベントを併用することで、各処理を実行中に発生した待ちに限定した集計も実施した

©2009 Information Processing Society of Japan

３．２　測定点設置

測定対象である mysql 内部に、3.1 に示したイベントを採取するための測定点（アプリケーション・プローブ）を設置した。これらのプローブが検出するイベントと設置対象関数を以下に示す。

(1) ロック競合検出のためのプローブ

測定対象 DBMS である mysql で用いられているロックは、mutex, rw_lock, conc_lock の３種類である。各々、ロック競合が発生した際におけるロック獲得までの待ちを検出するため、下記関数内の待ち開始および終了に対応する位置にアプリケーション・プローブを設置した。

mutex: mutex_enter_func()
rw_lock: rw_lock_s_lock_func(),
rw_lock_x_lock_func()
conc_lock: innodb_srv_conc_enter_innodb()

(2) DBMS 処理特定のためのプローブ

mysql のデータベースエンジンからストレージエンジン（InnoDB）への処理要求を把握するため、下記関数の入口と出口にアプリケーション・プローブを設置した。

ha_innobase::write_row()
ha_innobase::update_row()
ha_innobase::delete_row()
ha_innobase::index_read()
ha_innobase::general_fetch()
innobase_commit()

３．３　ベンチマーク実行

mysql や DBT-1（mysql への接続は ODBC）の構築とベンチマーク実行、および、mysql の実行パラメータ（my.cnf に記述）の設定は、OSS iPedia にある資料[5]に沿って行った。なお、ThnikTime については、iPedia 掲載の測定結果[3]と合わせるため、3.6 秒とした。

mysql の実行パラメータの内、並行実行処理の上限値（innodb_thread_concurrency）は、ロック待ち状況に大きく影響するため、6, 8, 12, 20, 30, 50, 無制限と変化させて測定を実施した。また、性能上の観点から、予め立ち上げておくスレッド数（thread_cache_size）を 200 とした。

ハードウエア構成については、使用 CPU 数が 8 と 16 の２種類について測定を実施した。この設定は、OS（linux）の boot パラメータ（maxcpus）によるものである。

負荷量については、EU（Emulate User）の値を 400, 800, 1600, 3200 と変化させた。

３．４　測定結果

(1)　スループットと CPU 使用率

CPU 使用率（x 軸）とスループット（y 軸）の関係を、innodb_thread_concurrency の値別にプロットしたグラフを図１（8CPU）および図２（16CPU）に示す。CPU 使用率は、1CPU分を 100%としているので、8CPU では最大 800%, 16CPU では最大 1600%となる。スループット値は、DBT-1 の結果集計プログラムが出力する BT/s (BogoTransation/秒) である。

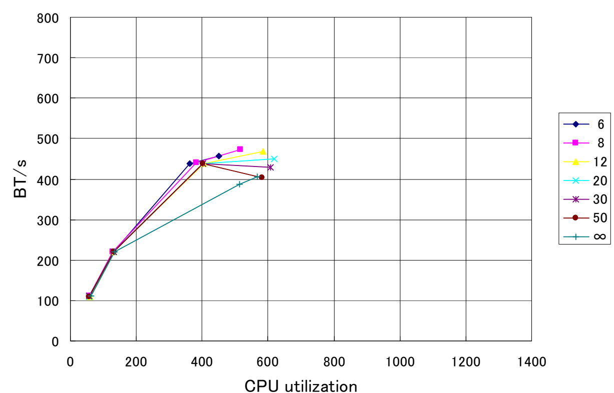

図１　CPU 使用率とスループット（8CPUs）
各系列とも、innodb_thread_concurrency を一定、EU を 800～3200 まで変化させたときの結果。図２も同様。

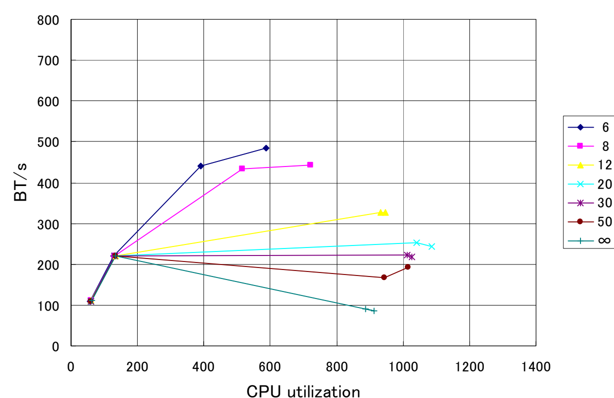

図２　CPU 使用率とスループット（16CPUs）

結果より明らかな通り、8CPU では 450BT/s 程度まで CPU 使用率の上昇とともスループットも向上しているが、16CPU では、並行実行処理の上限値を

©2009 Information Processing Society of Japan

6や8に制限した場合を除き、CPU使用がスループット向上に役立っていないことが分かる。

(2) ロック待ち時間

ロック待ち時間については、スペースの関係上、並行実行処理数が無制限のケース、および各CPU数の中で最も高いスループットを示したケースの2種類（計4種類）の結果を示す。なお、EUは3200、CPU数は、8および16である。また、集計対は、2.5秒間の動作である（以下同様）。

図3に8CPUの結果、図4に16CPUの結果を示す。並行実行処理数の上限値が有限の場合はconc_lockの待ち時間が大半となるが、本質的なボトルネックは、conc_lockではなく、それによって保護された部分に隠されているものと考えられる。

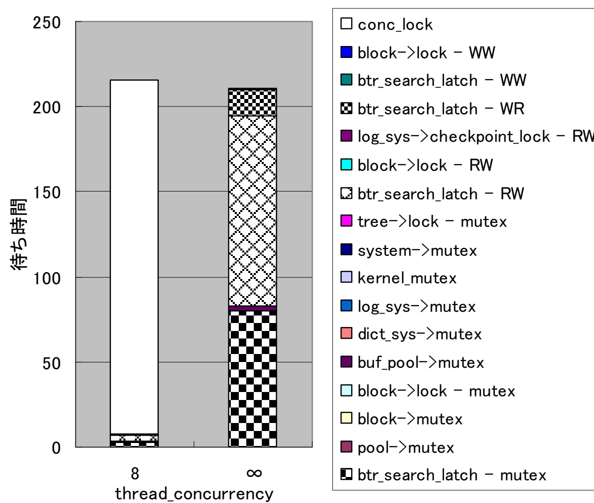

図3　ロック待ち時間の内訳（8CPUs）

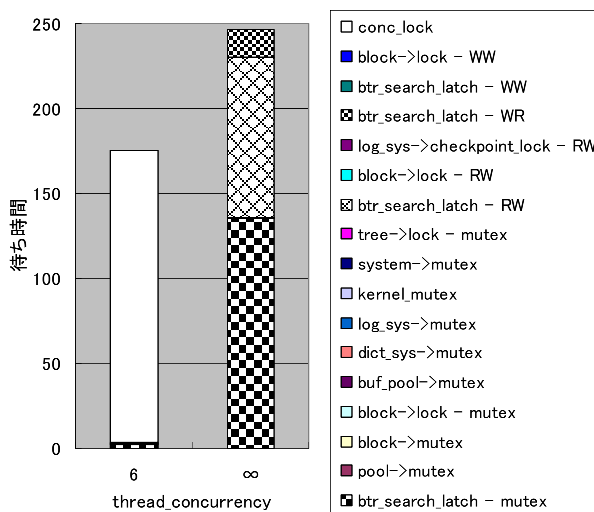

図4　ロック待ち時間の内訳（16CPUs）

(3) 処理内容とロック待ちの関係

ここでは、InnoDB内部のロック待ちと処理の関係を分析するため、innodb_thread_conncurrencyを∞としたケースについて、実行回数、CPU（使用）時間、経過時間、ロック待ち時間を処理別に集計した。表2、表3に示す。これより、1) InnoDBで実行される処理の大半はindex_read()とgeneral_fetch()で費やされていること、2) 最もlock待ちの影響を受けている処理はread_index()であることが分かった。

表2　InnoDB処理毎の集計（8CPUs）

時間の単位は秒。処理名は一部省略して表示した。表2も同様。

	実行回数	CPU時間	経過時間	lock待時間
write_row()	26	0.0097009	0.2555106	0.231957
update_row()	41	0.0024525	0.0056321	0.003107
index_read()	91399	7.9406988	206.23293	202.5892
general_fetch()	453644	1.1134265	8.4792632	7.51478
commit()	465	0.0098226	0.380624	0.003876

表3 InnoDB処理毎の集計（16CPUs）

	実行回数	CPU時間	経過時間	lock待時間
write_row()	18	0.0024597	0.0024832	0.000043
update_row()	12	0.0048349	0.0570324	0.055933
index_read()	31699	15.257769	229.24042	227.6535
general_fetch()	129006	1.523098	19.125124	18.72200
commit()	164	0.004496	0.3334836	0.000008

4．チューニング実施および効果の検証

ロック待ち時間の分析により、その大半はrw_lock内にあるmutexによるものであることが分かった。本節では、このmutexについてチューニングを実施し、その効果を検証する。

4．1　チューニング内容

問題となっているmutexは、rw_lockの状態を管理する変数への排他アクセスを実現するためのものである。幸いにも、その変数は、1)アクセス方法が単純、2)アクセスしている箇所は少ない、3) データのサイズが小さい、という特徴があったので、Compare And Swap（CAS）命令を用いたlock-freeな方式への改造は容易と予想された。そこで、下記に示す改造を行った。

まず、mutexによる排他制御を行ってアクセスす

©2009 Information Processing Society of Japan

るデータは、rw_lock 構造体の内、図５に示すメンバであることが判明したので、これらのメンバをCAS 可能な 64 ビット（図６）に集約した。この際、writer_thread は 64bit データであるポインタ である pthread_self()を用いていたが、これを 32bit データである gettid()を使用するように変更した。

```
struct rw_lock_struct {

    ulint   reader_count;   /* Number of readers who have locked this
                            lock in the shared mode */
    ulint   writer;         /* This field is set to RW_LOCK_EX if there
                            is a writer owning the lock (in exclusive
                            mode), RW_LOCK_WAIT_EX if a writer is
                            queueing for the lock, and
                            RW_LOCK_NOT_LOCKED, otherwise. */
    os_thread_id_t  writer_thread;
                            /* Thread id of a possible writer thread */
    ulint   writer_count;   /* Number of times the same thread has
                            recursively locked the lock in the exclusive
                            mode */

    ulint   pass;           /* Default value 0. This is set to some
                            value != 0 given by the caller of an x-lock
                            operation, if the x-lock is to be passed to
                            another thread to unlock (which happens in
                            asynchronous i/o). */
    ulint   waiters;        /* This ulint is set to 1 if there are
                            waiters (readers or writers) in the global
                            wait array, waiting for this rw_lock.
                            Otherwise, == 0. */
    ibool   writer_is_wait_ex;
                            /* This is TRUE if the writer field is
                            RW_LOCK_WAIT_EX; this field is located far
                            from the memory update hotspot fields which
                            are at the start of this struct, thus we can
                            peek this field without causing much memory
                            bus traffic */

}
```

図５ mutex で保護している
ロック変数のメンバ

```
union rw_status {
 atomic64_t a;
 long i;
 struct {
   os_thread_id_t     writer_thread;
   unsigned dummy: 1;
   unsigned waiters: 1;
   unsigned writer_is_wait_ex: 1;
   unsigned writer: 2;
   unsigned pass: 1;
   unsigned writer_count: 13;
   unsigned reader_count: 13;
 } b;
};
```

図６ 32bit に集約した rw_lock 変数のメンバ

```
mutex_enter(&(lock->mutex));
  lockのrw_status部分を更新;
mutex_exit(&(lock->mutex));
```

(a) 元のコード

```
while (1) {
  new = old = lockのrw_status部分;
  newを更新
  if( CAS( &lock, old, new ) が成功 )
}
```

(b) CASによりlock-freeとした

図７ lock-free な rw_status の更新

プログラムにおいて、rw_lock 変数（lock）をアクセスする部分は、図７に示す書き換えを行った。すなわち、mutex を確保して行っていた lock の更新操作をローカル変数に対して実施するように変更し、lock に対する更新操作は CAS により行うように変更した。これにより、rw_lock に関する mutex を不要とした。

４．２ 効果

rw_lock の mutex を不要とした mysql について、前節と同様の測定を実施した。CPU 使用率とスループットの結果を図８と図９に示す。これらの結果より、8CPU, 16CPU とも、負荷が高い領域でのスループットが改善されていることが分かる。これにより、3.4(2)で行った測定・分析は、ボトルネックを正しく特定できていたといえる。

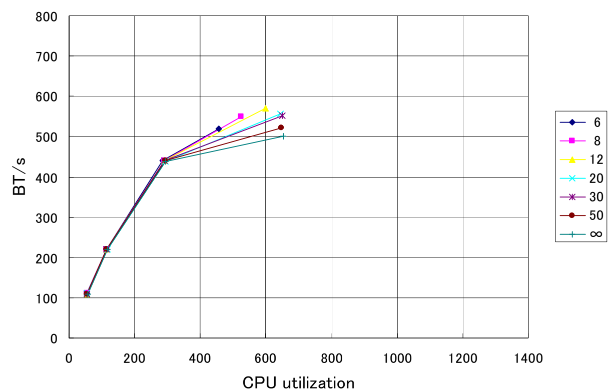

図８ チューニング後の CPU 使用率と
スループット（8CPUs）

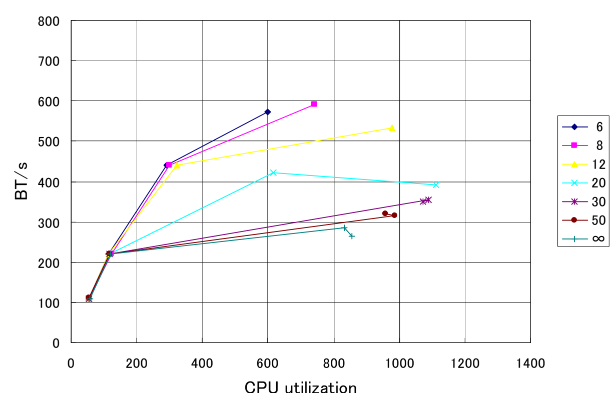

図９ チューニング後の CPU 使用率と
スループット（16CPUs）

ロック待ち時間については、前節と同様、最も高いスループットを示した並行実行処理数上限値のケ

©2009 Information Processing Society of Japan

ースと無制限のケースについての結果を図１０、図１１に示す。これらの結果より、改造によってボトルネックはbuf_poolのロックに移動したことがわかる。

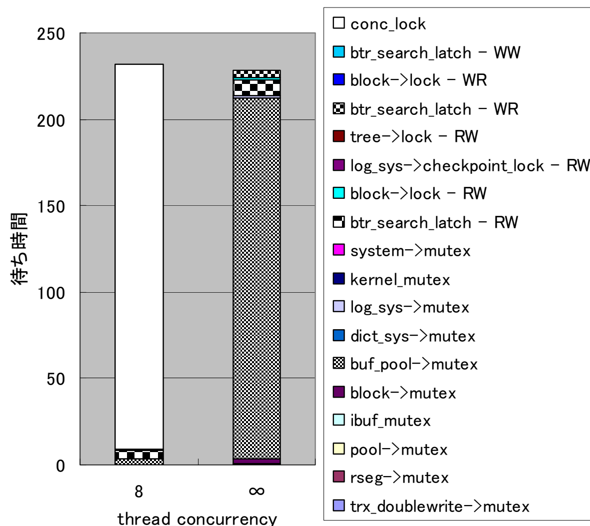

図１０　チューニング実施後の
ロック待ち時間の内訳（8CPUs）

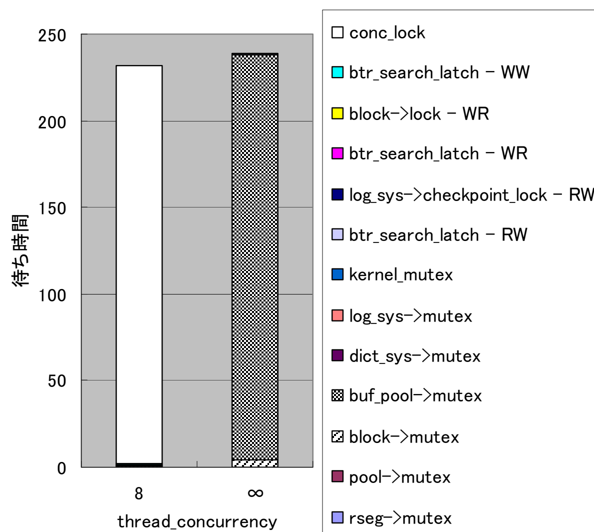

図１１　チューニング実施後の
ロック待ち時間の内訳（16CPUs）

５．関連研究

５．１ ロック待ち時間の測定

マルチコア・マシンの性能を考える上でロック競合は重要な要素であることから、測定ツールも充実している。代表例としては、Solaris DTraceのプロバイダであるlockstat, plockstat、linuxのlockmeterが挙げられる。

本研究で用いた手法は、これらのツールと基本概念レベルでは共通している。但し、lockstat と lockmeter が対象とするのはカーネル内で使用されるロックの競合、plockstatが対象とするのは、ライブラリが提供するpthread_mutex_tやmutex_tに関する競合である。このため、mysqlのようにプログラム内部で独自の排他制御を実装しているケースには対応できない。

また、3.4(3)で示したような、処理ごとのロック待ち状況分析も、既存ツールでは困難と考えられる。plockstatはDTraceのプロバイダであるので、処理内容を判別するためのプローブを別途用意し、両者を利用してロック待ち時間を集計するスクリプトを作成すれば同様の結果は得られることになるが、1)トレーサ内で集計を行うため測定オーバーヘッドが大きくなる、2)異なる観点からの集計処理が必要となった場合は再測定が必要となる、という欠点が生じる。測定で収集したトレース・データを、後からオフラインで分析する提案手法であれば、集計プログラムを用意するだけで異なる観点からの集計処理が可能となる、という利点がある。

５．２ ロックのチューニング

(1) ロック操作の効率化

マルチコア・システムの性能にとってロックは重要な要因との認識から、ロック操作について様々な効率化の提案がなされており、adaptive lockにおけるbusy waitのチューニング[6]やlock-free synchronization[7][8]が代表的なアプローチである。4.で実施したチューニングは、rw_lockの状態を管理する変数に対するアクセスをlock-freeで行うものであり、上記lock-free synchronizationよりも適用範囲は狭くなるが、処理は軽くて済む、という違いがある。

いずれにしても、チューニングを実施する際は、まずチューニング対象（ボトルネック）を特定する必要がある。本稿で提案した測定・分析手法は、一連のチューニングにおいて最初に実施するボトルネック特定のための技術と位置づけられる。

(2) mysqlのチューニング

マルチコア・システムを意識した性能チューニングの結果として、2009年4月21日にリリースされたmysql 5.4のInnoDBストレージエンジンは16-way x86 servers や64-way CMT serversに対応

©2009 Information Processing Society of Japan

するまでのスケーラビリティーを実現したと発表されている[9]。チューニングの一つは、rw_lock の状態を管理する変数に対するアクセスを CAS 等のアトミック操作で行うというもので、基本的な考え方は 4.で行ったチューニングと同様であった。

本研究の意義は、個別のチューニング内容にあるのではなく、まずボトルネックを特定し、その後にチューニングを行うという方法論を実現したことにあると考えている。4.で実施した rw_lock のチューニングは、方法論の有用性を示すための一実証例である。

なお、mysql 5.4(beta)について、提案手法により DBT-1 実行時のロック待ち状況を調べたところ、4.2 と同様、buf_pool の mutex による待ち時間が最も多いという結果が得られた。これより、DBT-1 に関する限りではあるが、次のチューニング対象は buf_pool の mutex であるといえる。

６．おわりに

本論文では、マルチコア・システムの性能にとって重要な要因と考えられるロックへの対応として、稼動システムの測定からボトルネックを特定する手法を提案し、これを DBMS に適用した。また、判明したボトルネック（ロック）をチューニングすることで性能が向上することを確認した。これにより、ロックがボトルネックとなる状況に対処する方法論の有用性を実証できた。

今後は、OS 別ミドルウエア別といったセグメント毎に断片的に提供されている既存ツールの共通化やボトルネックに対処する方法の体系化といった、簡便に使える技術としての完成を目指す方向に発展していくものと考えている。

謝辞

本研究における評価実験を行うにあたり、実験用機材の借用を快諾して頂きました F5 ネットワークスジャパン株式会社様に感謝します。

参考文献

[1] Database Test Suite, http://sourceforge.net/projects/osdldbt/
[2] TPC-W, http://www.tpc.org/tpcw/
[3] MySQL に対応した評価ツール DBT-1 を利用したハードリソース変更によるパフォーマンスへの影響の考察, http://ossipedia.ipa.go.jp/capacity/EV0612260303/
[4] T. Horikawa, "A Framework for Performance Evaluation Based on Event Tracing," IPSJ Journal, Vol. 42, No. 1, pp. 68-78, 2001.
[5] OSS 技術開発・評価コンソーシアム,「OSS 性能・信頼性評価 / 障害解析ツール開発」, DB 層～OSDL DBT-1/3 による DBMS 評価編～ http://www.ipa.go.jp/software/open/forum/development/download/051115/db-dbt.pdf
[6] 小崎資広, 「スケジューラの挙動は三巨頭会談で決まるのだ？」, Linux Kernel Watch, 2009 年１月版、 http://www.atmarkit.co.jp/flinux/rensai/watch2009/watch01b.html
[7] Simon Doherty, et al., "Bringing Practical Lock-Free Synchronization to 64-bit Applications," in Proc. PODC '04, pp. 31-38, 2004.
[8] Keir Fraser and Tim Harriss, "Concurrent Programming Without Locks," ACM Transactions on Computer Systems, Vol. 25, No. 2, pp. 2-61, 2007.
[9] MySQL :: Sun Announces MySQL 5.4: Up To 90% Faster Response Times, and Scalability Up to 16-way x86 Servers and 64-way CMT Servers, http://www.mysql.com/news-and-events/generate-article.php?id=1602

©2009 Information Processing Society of Japan