情報処理学会論文誌　Vol. 51　No. 7　1382–1393 (July 2010)

データベースにおけるマルチプロセッサスケーラビリティボトルネックの分析手法

堀　川　　隆[†1]

CPU 資源が強化されたマルチコアプロセッサ搭載マシンでは，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場合がある．このようなボトルネックは特定が困難なことから，チューニングなどの対応を効率的に行うための体系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．マルチコアプロセッサ搭載のシステムによるデータベース処理ベンチマーク・プログラム実行に対して提案手法を適用し，各ロックが処理時間に与える影響を分析することができた．さらに，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が実現できた．これらにより，提案手法の有用性を裏付けることができた．

An Analysis method for Bottlenecks of Multi-processor Scalability and Case Study on Database Management Systems

Takashi Horikawa[†1]

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 processors 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.

1. はじめに

近年，マルチコアプロセッサの普及にともなって IT システムの CPU 資源が強化されつつあり，性能問題は自然に解消されていくという期待もある．しかしながら，マルチコアプロセッサを活用してシステム性能を向上させるには，複数 CPU 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）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，ボトルネックとなりうる要因は多岐にわた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などにも依存することから，その特定は困難をきわめることが多い．特に，複数 CPU による並列処理特有のボトルネックである『並列に実行できない処理』については，物理資源（CPU，disk，network，memory など）の使用率測定[1] や実行プロファイル採取[2] 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．

『並列に実行できない処理』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項目を共有して行う処理の場合は，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データベース管理システム（DBMS）は，このようなボトルネックが発現する典型例といえる．DBMS のスケーラビリティに関する評価としては，たとえば，BerkeleyDB

†1 日本電気株式会社
NEC Corporation

では 4，MySQL では 8，PostgreSQL では 16 が同時動作するスレッド数の限界，という報告が紹介されている[3]．

本論文では，多数の CPU を搭載する IT システム上で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．さらに，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

クリティカルセクションがスケーラビリティに関するボトルネックとなる原因は，1) その区間を実行できるスレッド数が制限される，2) 競合発生時は，調停操作がオーバヘッドとして加わる，の 2 点である．1) は，クリティカルセクション実行時間の比率，すなわち，アプリケーションの構造に依存しており，スケーラビリティに影響するメカニズムは自明である．一方，2) は，ライブラリ（アプリケーション内部の場合もある）や OS で実装されている排他制御方式といった複数の要因が影響するため，スケーラビリティへの影響は複雑となる．

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，状況に応じて異なる振舞いを示す．具体的には，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解放を待つ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．待合せの方法については，1) **表 1** に示す 2 種類があること，2) 各々の方式には一長一短あること，3) 両者を組み合わせた 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 する）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 性を実現するためのロックと 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，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，という点があげられる．

表 1　ロック待ち方法の比較
Table 1　Pros and cons for the way of enforcing critical sections.

	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	ロック待ち時間が長くなると，それに伴って 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 ロック解放への反応は，wake up 操作が入るため，時間がかかる

しかしながら，いずれも排他制御のためにロック（ラッチ）の獲得操作と解放操作を行うこと，さらには，競合が発生した場合はロックを待つ処理やロック解放を通知する処理が行われる，という点で共通している．本論文では，特定のベンチマーク・プログラム（DBT-1）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[4] 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た web e-Commerce システムにおけるトランザクション（TPC-W[5] に準じているが，その仕様を完全に満たすものではない）を発生させて DBMS の性能をテストするベンチマーク・プログラムである．また，このようなオープンなベンチマーク・プログラムによってオープンソース DBMS の性能を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[6] 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 に増加させると性能（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，直接的には，CPU 以外の要素がボトルネックとなっていることを示している．さらに，他の物理資源（disk や network など）の使用率も飽和していないことから，ボトルネックはロックといった論理的な資源である可能性が高いと考えられる．そこで，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・分析手法

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[7] 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 コンセプト

性能チューニングを目的としてボトルネックを特定する際に重要なのは，性能への影響度を明確にすることである．IT システムでは，ボトルネックとなる可能性のある要因は多岐にわたっているため，性能に最も影響する要因を特定することが性能チューニングの第一歩となる．性能にほとんど影響していない要因について対策を施してもその効果は少ないため，結果としてチューニング作業が無駄になってしまうからである．

このため，提案手法では，システムの稼働状況を分析してボトルネックを特定する際には，処理に要する時間（レスポンス）に対してボトルネック要因が占める割合を調べることを基本とした．

3.2 採取対象イベントとデータ分析

提案手法では，イベント・トレースに関するフレームワーク[7] に基づく測定・分析により，ボトルネック特定に必要な結果を得ることになる．

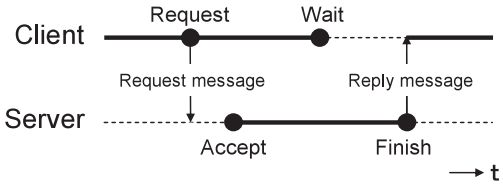

図 1　能動資源への処理要求・応答に関するイベント
Fig. 1　Abstract events for active resource.

(1) フレームワークの概要

フレームワークでは，IT システム内の要素をオブジェクトと考え，オブジェクト間の相互作用を抽象的なイベントとして規定している．オブジェクトは，処理に関係するすべての要素であり，プロセスやスレッドといったプログラムの実行主体，CPU や disk といった処理の実行を担う物理資源，処理を進めるために必要となる権限（排他制御のための lock など）といった論理資源，を総称している．

抽象イベントは，あるオブジェクトから別のオブジェクトに対する要求と応答に関する状態遷移，すなわち，利用要求（Request），利用開始（Accept），利用終了（Finish），待ち開始（Wait）を意味している．ここで，要求を出す側の Client オブジェクトは，主体的に処理を進めるプロセスといった能動オブジェクトであるが，要求を受ける側の Server オブジェクトは，能動資源の場合と，処理に必要な権限を表す受動資源の場合がある．

Server オブジェクトが能動資源の場合における処理要求と応答に関するイベントのタイミング関係を**図 1** に示す．ここでの Wait は，Client オブジェクトが Server オブジェクトからの処理完了を待つことに対応する．一方，Server オブジェクトが受動資源の場合におけるタイミング関係を**図 2** (a)，(b) に示す．ここでの Wait は，Server オブジェクトが利用可能状態になるのを Client オブジェクトが待つことに対応する．能動資源の場合との大きな違いは，1) Client オブジェクトが処理を実行するので，イベントは Client オブジェクトにおいて発生すること，2) Client オブジェクトの Request は Wait を兼ねていること，3) Request & Wait のタイミングで Server オブジェクトが使用可能（競合なし）であれば，Accept も同時に発生したものと解釈すること，である．

フレームワークでは，これらの抽象イベントの時系列（イベント・トレース）から，IT システムの振舞いを表す各種指標を求める方法も規定している．以上，フレームワークに基づく測定・分析は，1) 抽象イベントに対応する実イベントの規定，2) 実イベントの時系列

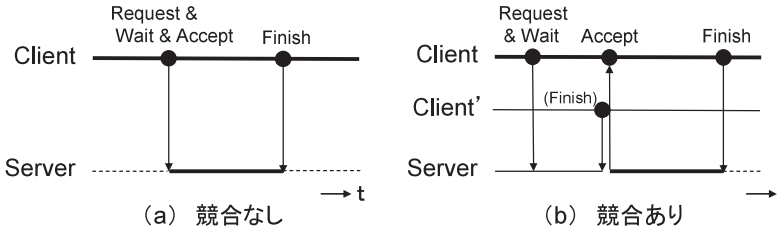

図 2　受動資源への使用要求に関するイベント
Fig. 2　Abstract events for passive resource.

採取，3) トレース・データの分析，という手順で実施することになる．

(2)　採取対象イベントの設定

具体的な採取対象イベントは，測定対象ごとに抽象イベントの意味を解釈して設定する必要がある．ここでは，処理時間，物理資源，論理資源を測定対象として採取対象イベントを設定した．

処理時間の測定に必要となるのは，処理の開始と終了イベントである．これらは，処理を担うスレッドを一種のリソースと考えた場合の使用開始および使用終了イベントと考えることができる．ここでは，DBMS のストレージエンジンに対する各種の処理要求を測定対象とした．採取するイベントは，処理要求に応じて起動されるストレージエンジン処理の開始および終了イベントである．

物理資源（CPU，disk，network）の使用状況測定で必要となるイベントは文献 7) で規定されている．通常，これらのイベントはオペレーティング・システム（以降，OS）内部で検出する必要がある．ここでは，Linux カーネル内部に挿入した測定点（プローブ）によってこれらのイベントを採取した．

論理資源（ロックなど）についても 4 種類の抽象イベントを規定できるが，実働状況を反映した結果を得るという観点から，ここでは，採取対象を競合発生時における「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「使用開始」イベント（以降，ロック待ち終了イベント）に限定した．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ヘッドが大きくなり，稼働状況を正確に反映した結果が得られなくなると考えたためである．

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．さらに，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．

3.3　測定点設置

3.2 節で示した採取対象イベントのうち，処理時間およびロック待ちに関係するイベントを検出するための測定点（アプリケーション・プローブ）は，MySQL 内部に設置した．これらのプローブを設置した MySQL 内部の関数を以下に示す．

(1)　処理時間測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のアプリケーション・プローブは，下記インタフェース関数の入口と出口に設置した．

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

(2)　ロック待ち検出のためのプローブ

InnoDB 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，ロック競合時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．InnoDB の mutex と rw_lock に関しては，同じ関数が異なる mutex や rw_lock で利用されているので，プローブではロックの種類を区別するための情報として，mutex および rw_lock 構造体内に含まれている構造体作成処理の位置情報（source ファイル名と行番号）を利用した．これにより，ロック対象リソースの種類を区別することができる．conc_lock については，ロック対象リソースは 1 種類であったため，種類を区別するための情報は不要であった．

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

4.　ベンチマーク実行時におけるボトルネックの測定・分析

ここでは，MySQL で DBT-1 ベンチマーク・プログラムを実行させたときの挙動を測

定・分析し，マルチプロセッサのスケーラビリティを阻害するボトルネックを特定する．測定対象マシンは，Intel の Xeon E7310（1.6 GHz，4cores）を 4 プロセッサ（計 16cores）搭載するサーバ，OS は Linux（CentOS 5.2，カーネルは 2.6.18-92.1.22.el5），DBMS は，MySQL-5.0.75 である．イベント・トレース採取は，OS や MySQL にソフトウェア・プローブを挿入して実施した．

4.1　DBT-1 の実行

(1)　実行環境

DBT-1 ベンチマークの実施方法，具体的には，MySQL や DBT-1 環境の構築，ベンチマーク・プログラムの実行方法，および，my.cnf に記述する MySQL の実行パラメータは，おおむね OSS iPedia より公開されている資料[8] の記載に従った．DBMS への負荷量設定に関しては，ThinkTime を 3.6 秒（固定値）とし，EU（Emulate User）数を変化させた．なお，ThinkTime の設定では，iPedia 掲載の測定結果[6] との比較の容易さを重視した．

その他の実行パラメータについては，次のように取り扱った．ベンチマーク測定によって実行パラメータと性能の関係を調べるため，性能に影響する可能性が高いと考えられるパラメータについて，下記に示す組合せをベンチマーク対象ケースとした．

- 並行実行処理の上限値（innodb_thread_concurrency）

このパラメータは，ロック待ち状況に大きく影響すると考えられる．ベンチマークでは，このパラメータが 6，8，12，20，30，50，無制限の場合について測定を実施した．

- 使用するコア数

マルチコア・システムのスケーラビリティを評価するため，使用するコア数が 8 と 16 の場合について測定を実施した．なお，使用コア数は，OS（Linux）の boot パラメータ（maxcpus）により設定した．

(2)　実行結果

DBT-1 ベンチマークが出力する実行結果は，1 秒間に実行したトランザクション数（スループット）とトランザクション種別ごとの平均応答時間である．また，OS が標準的に提供している測定機能により CPU や I/O の使用率も測定するようになっている．

ここでは，実行パラメータとベンチマーク性能の関係を俯瞰するため，スループットと CPU 使用率の関係を調べた．結果を**図 3**（8CPU）および**図 4**（16CPU）に示す．両グラフとも，innodb_thread_concurrency が 6 から無制限（∞）の各ケースについて EU を 400，800，1,600，3,200 と変化させてベンチマークを実行し，各 EU 値における CPU 使用率（x 軸）とスループット（y 軸）のプロットを線で結んだ．CPU 使用率は，1CPU 分を 100%としたので，8CPU では最大 800%，16CPU では最大 1,600%となる．スループットの単位は，BT/s（BogoTransation/秒）である．

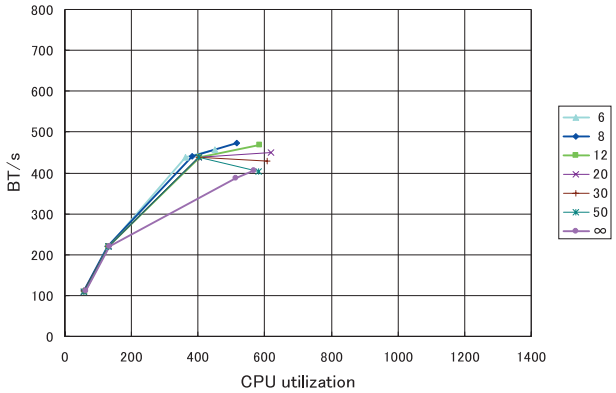

図 3　CPU 使用率とスループット（8CPUs）
Fig. 3　CPU utilization and throughput for 8-CPU case.

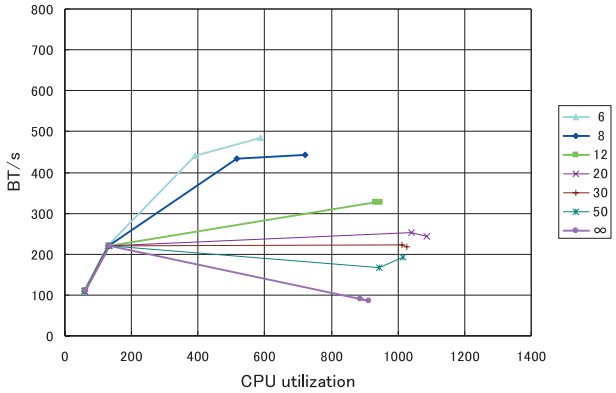

図 4　CPU 使用率とスループット（16CPUs）
Fig. 4　CPU utilization and throughput for 16-CPU case.

8CPU では 450 BT/s 程度まで CPU の使用量増加（使用率上昇）とともにスループットが向上しているが，16CPU では，並行実行処理の上限値を 6 や 8 に制限した場合を除き，CPU の使用量増加がスループット向上に役立っておらず，場合によっては性能が低下する，

という結果となった．並行実行処理の上限値が 12 以上で負荷（EU 値）の高い領域において，16CPU の性能が 8CPU のものより明らかに低くなる，という現象である．この原因は解明されているわけではないが，1 つの可能性として，クリティカルセクションによるスケーラビリティの限界（たとえば，MySQL では有効に同時動作するスレッド数の限界は 8[3]）の発現が考えられる．この限界を超えるスレッドが同時動作するとクリティカルセクションに関する競合が激しくなり，調停操作のオーバヘッドが CPU 使用量の増加とスループット低下を招く，という因果関係である．IPA より公開されている資料に，DBT-1 を対象とした oprofile によるプロファイリング結果として，MySQL 内部で実装されている mutex 関連の関数や pthread_mutex 関連の関数が実行時間の上位となっているデータ[9] が示されていることは，この予想を支持する一例と考えられる．

4.2　イベント・トレース手法による測定・分析

DBT-1 ベンチマーク・プログラムを実行しているマシンからイベント・トレースを採取し，ボトルネックを特定するための分析（集計処理）を行った．ベンチマークの実行条件によって単位時間あたりのトレース・データ量は異なるため，イベント・トレースに記録できた動作時間は異なっているが，ここでは簡単のため，分析対象時間を全ケースで同一（2.5 秒間）とした．

(1)　ボトルネックの特定

4.1 節の DBT-1 実行時におけるボトルネックは，ロックなどの論理資源と推測できるので，提案手法による測定・分析を行い，これを検証する．

CPU 数が 8 および 16，並行実行処理数を無制限としたケースを対象に，InnoDB への処理要求の種別ごとに実行回数，実行時間（経過時間），CPU 使用時間，ロック待ち時間を集計した結果を**表 2**，**表 3** に示す．表において，時間の単位は秒，また，CPU を使用して Lock を待つ場合もあるので，CPU 時間と Lock 待ち時間の合計は実行時間より大きくなることもある．

この結果より，ストレージエンジンに対する処理要求の大部分については処理時間の大半が lock 待ち時間に費やされており，CPU 時間の割合は少ないことが判明した．さらに，最も lock 待ちの影響を受けている index_read() が InnoDB 処理時間の大部分を占めているが，実行回数では general_fetch() が最も多いことも分かった．

(2)　ロック待ち時間の分析

ここでは，ロック待ちの影響が最も大きかった index_read() について，実行 1 回あたりのロック待ち時間の内訳を求める．内訳とは，ロック対象リソースの種類（3.3 節 (2) 参照）別に集計したロック待ち時間である．最も負荷をかけたとき（EU = 3,200）の結果を**図 5** と**図 6** に示す．各図で示した並行実行処理数は，無制限，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 種類である．なお，図の凡例には，待ち時間がゼロでなかった lock をすべて列挙したが，大半の lock については待ち時間がゼロに近い値であったため，棒グラフでは存在していないように見える．

並行実行処理数の上限値が有限値の場合はいずれも conc_lock の待ち時間が大部分を占めているが，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．このボトルネックは，並行実行処理数を無制限とした場合に発現することから，複数 CPU 処理のスケーラビリティに影響しているボトルネックは，このケースの結果を調べることで判断できると考えられる．

並行実行処理数の上限値を無制限とした場合の分析結果より，実行時間の大半を占める

表 2　InnoDB 処理ごとの集計（8CPUs）
Table 2　Processing time details for InnoDB APIs (8CPU case).

	実行時間	CPU 使用時間	Lock 待ち時間	実行回数
write_row()	0.255511	0.009701	0.231957	26
update_row()	0.005632	0.002453	0.003107	41
index_read()	206.232933	7.940699	202.589160	91399
general_fetch()	8.479263	1.113426	7.514778	453644
commit()	0.380624	0.009823	0.003876	465

表 3　InnoDB 処理ごとの集計（16CPUs）
Table 3　Processing time details for InnoDB APIs (16CPU case).

	実行時間	CPU 使用時間	Lock 待ち時間	実行回数
write_row()	0.002483	0.002460	0.000043	18
update_row()	0.057032	0.004835	0.055933	12
index_read()	229.240424	15.257769	227.653505	31699
general_fetch()	19.125124	1.523098	18.722005	129006
commit()	0.333484	0.004496	0.000008	164

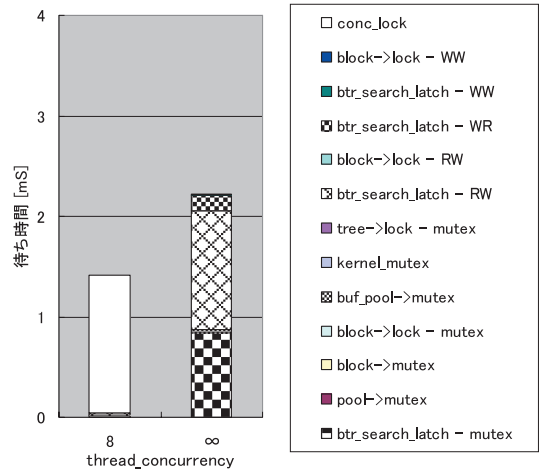

図 5　ロック待ち時間の内訳（8CPUs）
Fig. 5　Lock waiting time detail (8CPU case).

ロック待ち時間の大部分は btr_search_latch に関する待ちであることが分かった．ここで着目すべき点は，CPU 数が 8 から 16 に増加すると，btr_search_latch 内の mutex に関する待ち時間の比率が増大することである．CPU 数が増えるにつれて性能への影響が大きくなる要素は，CPU 数に関するスケーラビリティ・ボトルネックと考えられる．

また，図 5 と図 6 に示したロック待ち時間の逆数が，おおむねスループット性能と相関関係にある点も注目に値する．これは，今回の DBT-1 実行におけるスループット性能を決める最大の要因がロック待ち時間であること意味しているので，この要素が性能チューニングにおいて最初に考慮すべき要因であることを明瞭に示しているといえる．

(3)　測定オーバヘッド

イベント・トレース手法による性能測定・分析の懸念は測定オーバヘッドである．本節で実施した測定において，イベント・トレースを採取した場合，しない場合における性能指標値の比較を**表 4** に示す．負荷の高い領域（EU = 3,200）では，イベント・トレースを採取することで，スループットが 20〜35%低下している．いずれも，オーバヘッドによる CPU 使用量増加がスループット低下に直結する動作状況ではないため，この値は測定オーバヘッドそのものではないが，目安にはなると考えられる．カーネル・プローブのみで測定を行った場合のオーバヘッドはおおむね数%以内という結果が得られている[10] ことから，このオーバヘッドは，ロック待ちの開始・終了という高頻度で発生するイベントを採取対象に含めたことによるものと考えられる．

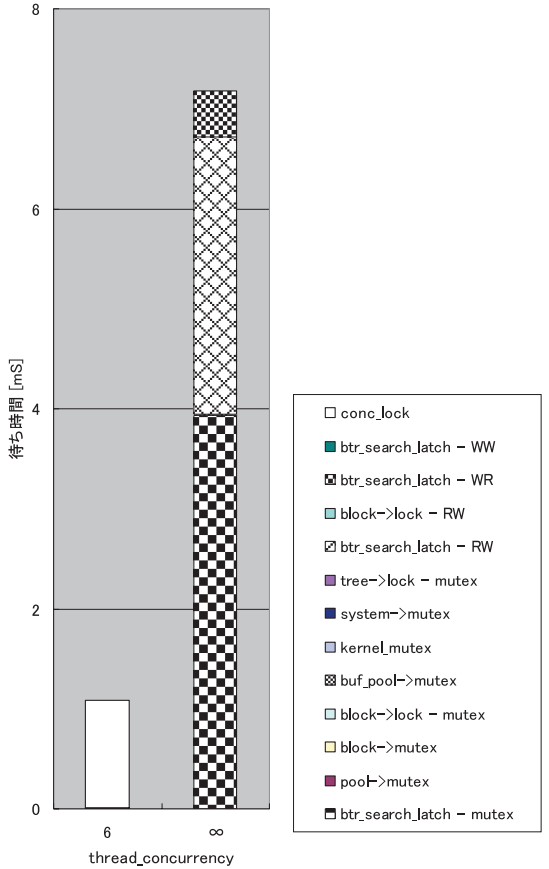

図 6　ロック待ち時間の内訳（16CPUs）
Fig. 6　Lock waiting time detail (16CPU case).

一般論としては，測定オーバヘッドは測定時の動作を実運用時（非測定時）と乖離させる要因になるので，小さい方が望ましい．この乖離により，実運用時の動作を反映した結果が得られなくなると，測定・分析の有用性が薄れてしまうからである．しかしながら，測定・

表 4　測定オーバヘッドによる性能低下
Table 4　Performance degradation due to the measurement overhead.

実行条件 EU=3200		イベント・トレース 採取なし		イベント・トレース 採取あり	
CPU 数	thread concurrency	スループット [BT/s]	CPU 使用率[%]	スループット [BT/s]	CPU 使用率[%]
8	8	472.9	517.04	378.0	524.24
8	∞	405.9	568.56	266.2	473.84
16	6	484.5	587.20	379.0	584.16
16	∞	90.9	887.84	62.5	861.76

分析の有用性はオーバヘッドの大小だけで決まるのではない点は言及しておく必要があると考える．たとえば，本節で示したようなボトルネック特定を目的とした測定・分析の場合に問題となるのは，オーバヘッドのために実運用時とは異なる要素が測定実施時のボトルネックとなり，その測定データ（トレース・データ）の分析で間違った要素をボトルネックとして特定してしまう，といった状況である．このような問題が起きていないかどうか，すなわち，本節で行った測定・分析の妥当性，いいかえると，特定したボトルネックの検証は，5 章の性能チューニングで実施する．

また，システムの実運用時にイベント・トレース採取といった詳細な性能分析のための測定を行うことはほとんどないため，システムの品質に測定オーバヘッドが悪影響を及ぼすことはない点も補足しておく．

5. ボトルネック分析に基づく性能チューニング

本章では，4.2 節で行った測定・分析によって判明したスケーラビリティ・ボトルネック（rw_lock 内の mutex）のチューニングを実施する．チューニングにより性能が向上すれば，イベント・トレースを利用した提案手法は正しくボトルネックを特定できたと考えられる．

5.1 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，以下に示す

```
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 7　mutex で保護している rw_lock 構造体のメンバ変数
Fig. 7　The member variables in rw_lock_struct protected with the mutex.

改造を行った．

まず，排他制御によって保護するデータは，rw_lock 構造体のうち，**図 7** 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 8**）に集約した．次に，プログラムにおいて，rw_lock 変数（lock）をアクセスする部分に対して**図 9** 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．

メンバ集約のために実施した各変数の bit 数削減によって大きな影響を受けたのは，writer_thread，reader_count，writer_count である．この他のメンバは，TRUE，FALSE もしくは状態を表す id を格納する目的で使用されていたので，id に対応する定数値の変更

```
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 8　32 bit に集約した rw_lock 構造体のメンバ変数（rw_status）
Fig. 8　The member variable shrunk into rw_status of 32 bits.

```
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 ) break;
}
```

(b) CASによりlock-freeとしたコード

図 9　lock-free な rw_status メンバの更新
Fig. 9　Atomic modification of rw_status.

といった軽微な影響で収まった．

write lock を獲得しているスレッドの id を示す writer_thread については，格納する値を，64 bit データ（ポインタ）である pthread_self() の値から 32 bit データである gettid() の値に変更したが，プログラムの意味は，変更前と同じに保つことができた．

一方，read lock を獲得しているスレッド数を示す reader_count と，同じスレッドが write lock を再帰的に取得した数を示す writer_count は，bit 数削減により，表現可能な値の上限が $2^{13} - 1 = 8{,}191$ に減少する，という影響を受けた．このため，bit 数削減によって問題が生じる可能性についての検証が必要である．

reader_count については，Linux で使用可能なスレッド数（Linux ではプロセス数と同じ）の上限である 2^{15} より小さいため，問題を起こす可能性はゼロではない．しかしながら，下記の理由により，実用的な問題は発生しないと考えている．

- 今回の DBT-1 ベンチマーク実行では，資料[6] に従い，MySQL への最大接続数をクライアント側は 100，MySQL 側は 120 に設定しているので，各接続に対応して動作するスレッドの数は 100 程度である．
- 実際のシステムでも，数千のスレッドを使用することはほとんどない．
- Linux や MySQL に装備されている，スレッド数の上限値を設定する機能により，問題の生じないスレッド数の範囲でシステムを動作させることが可能である．

writer_count の上限値はプログラムの動作次第であり，MySQL の source code や OS の仕様から検討を加えることは困難であった．そこで，writer_count の上限値を別途調査したところ，DBT-1 実行中の最大値は 7 であったため，少なくとも，DBT-1 実行に関しては bit 数削減の問題は発生することはないと考えてよい．そこで，DBT-1 実行におけるチューニング効果確認を目的とする今回の実験では，単純に bit 数を削減する方法を採用した．なお，writer_count に関しては，改造や動作確認の手間は増えるが，以下の方法で，プログラムの意味を保ったままでの改造が可能である．着目するのは，writer_count を 2 以上に設定する操作は，すでに write lock を獲得した状況で実施するため，排他的なアクセスは不用であるという点である．具体的には，まず，writer_count を，1) write lock されているかどうかを示す 1 bit のフラグ，2) 再帰的に write lock を獲得した回数，の 2 つに分けたうえで，1) のフラグのみを CAS によるアクセス対象に含め，2) には bit 数削減前のデータ型（ulint）を使う，という改造になる．

5.2　効果の検証と考察

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 10**，**図 11** 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．また，チューニング実施後も index_read() の実行時間が大部分を占める状況に変化はなかったので，4.2 節 (2) と同様，index_read() 実行 1 回あたりのロック待ち時間内訳を調べた．結果を**図 12**，**図 13** 示す．これより，チューニングの効果として，1) ロック待ち時間が短縮されていること，2) ボトルネックは buf_pool を保護する mutex に移動したこと，が分かった．また，btr_search_latch に関するすべての待ちが軽減されたことにも注目する必要があろう．それらの待ち時間には，内部の mutex が直接的，間接的に関わっていた可能性が考えられる．

チューニングの効果として着目すべき点は，最大のスループットを発揮した並行実行処理数上限値が，8CPU の場合は 8 から 12，16CPU の場合は 6 から 8 に増加した点である．これは，チューニングによってスケーラビリティが改善されたことを意味している．したがって，4.2 節で実施した測定・分析は，スケーラビリティに関するボトルネックを正しく特定できていたと考えられる．

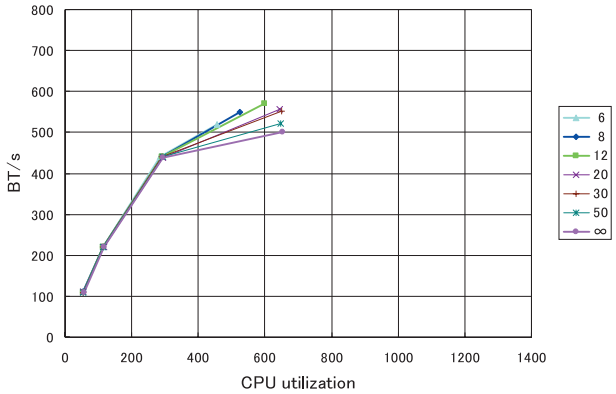

図 10　チューニング後の CPU 使用率とスループット（8CPUs）
Fig. 10　CPU utilization and throughput for tuned system (8CPU case).

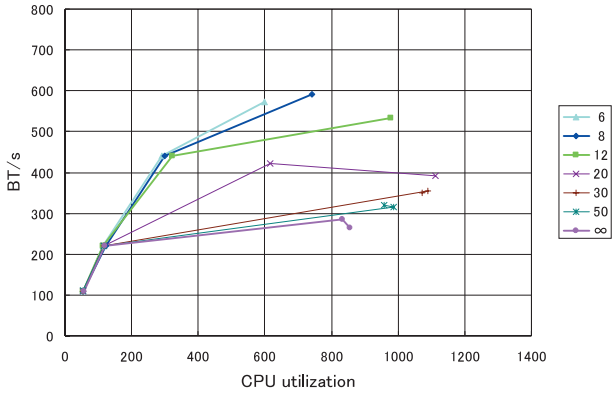

図 11　チューニング後の CPU 使用率とスループット（16CPUs）
Fig. 11　CPU utilization and throughput for tuned system (16CPU case).

なお，btr_search_latch のチューニング実施後も，16CPU で innodb_thread_concurrency の大きい領域で，8CPU よりスループットが顕著に悪い，という問題が残っている．これについては，1) チューニング後は buf_pool の mutex に関する待ち時間がロック待ち時間の最大要因，2) ロック待ち時間の逆数がスループット性能と大まかな相関関係にある状況（4.2 節 (2)）に変わりはない，の 2 点より，依然，ロック待ち時間に原因があると考える

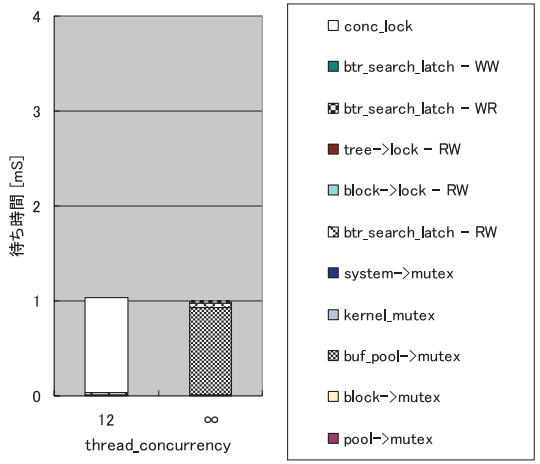

図 12　チューニング実施後のロック待ち時間の内訳（8CPUs）
Fig. 12　Lock waiting time detail for tuned system (8CPU case).

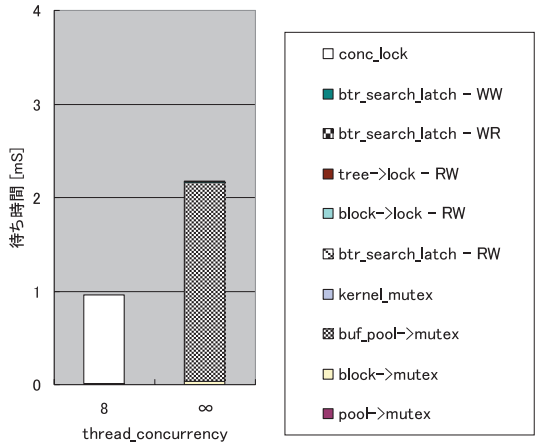

図 13　チューニング実施後のロック待ち時間の内訳（16CPUs）
Fig. 13　Lock waiting time detail for tuned system (16CPU case).

のが妥当である．したがって，buf_pool によるクリティカルセクションの処理をチューニングするなどにより，この待ち時間を低減，さらには解消することができれば，8CPU と 16CPU の差は解消されるものと期待できる．これは，当初のシステムに対して提案手法を適用し，btr_search_latch のチューニングを行ったのと同様の手順を適用することになる．

buf_pool チューニング後のボトルネックがどの要素になるのかは現時点では不明だが，別のロックに移動した場合でも，さらにそのロックについてチューニングする，という操作を繰り返すことにより，最終的には CPU などの物理資源がボトルネックになる状況に持ち込むことができると考えている．これは，物理資源を強化することにより性能向上が見込める状況，すなわち，スケーラビリティ上のボトルネックを解消できたことを意味している．

6. 関 連 研 究

6.1 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うえでロック競合は重要な要素と考えられており，測定ツールも整備されている．代表例としては，Solaris DTrace のプロバイダである lockstat，plockstat，Linux の lockmeter があげられる．

これらのツールは，ロックに関する振舞い（取得と競合）に特化しているため，これだけでは，ボトルネック特定で重要と考えられる処理時間に最も影響している要因の調査（4.2 節参照）は実施できない．ここで用いた性能測定・分析手法は，測定（トレース・データ収集）が完了した後に集計処理を行って各種の性能指標を調べる方式のため，測定時に必要なイベントを採取しておくことで，後から種々の観点からの集計処理を行って性能を分析することができる，という利点がある．

ロック待ち時間を求めるための基本処理，すなわち，各スレッドのロック待ち開始から終了までの時間を集計するというレベルでは，どのツールも共通している．ただし，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，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 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への対応は容易ではない，という点も問題となる可能性がある．

6.2 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[11] や lock-free synchronization[12),13)] が代表的なアプローチである．5 章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．本論文で提案した測定・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) 一般的な DBMS や MySQL のチューニング

文献 3) では，マルチコアプロセッサによる DBMS 処理では，従来よりも排他処理が性能やスケーラビリティに影響するとの認識を示し，種々の mutex の実装方式（文献では synchronization primitives）について性能比較を行っている．しかしながら，ボトルネックとなっている mutex の特定は行われていない．

マルチコア・システムを意識した MySQL の性能チューニングとしては，2009 年 4 月 21 日にリリースされた MySQL 5.4（beta）において，InnoDB ストレージエンジンを 16-way x86 servers や 64-way CMT servers に対応できるスケーラビリティを実現したとされている[14),15)]．そこで行われているチューニングの 1 つは，rw_lock の状態を管理する変数へのアクセスをプロセッサが提供するアトミック操作（CAS 命令など）を利用して行うもので，基本的な考え方は 5 章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．5 章で実施した 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，5.2 節と同様，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．この結果は，少なくとも MySQL による DBT-1 実行に関する限り，次のチューニング対象は buf_pool の mutex による排他処理であることを示している．

7. 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 別やミドルウェア別といったセグメントごと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　DBT-1 ベンチマークの実行手順や評価結果の公開に関わられました日本 OSS 推進フォーラム開発基盤ワーキンググループのメンバ各位に謹んで感謝の意を表します．また，本論文に対して有意義なコメントをいただきました国立大学法人九州大学大学院システム情報科学研究院福田晃教授に感謝します．

参 考 文 献

1) Loukides, M.（著），砂原秀樹（監訳）：UNIX システムチューニング，アスキー出版局 (1991).
2) OPROFILE. http://oprofile.sourceforge.net/
3) Johnson, R., Pandis, I. and Ailamaki, A.: Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines, *Proc. DaMon 2008*, pp.35–40 (2008).
4) Database Test Suite. http://sourceforge.net/projects/osdldbt/
5) TPC-W. http://www.tpc.org/tpcw/
6) 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/
7) Horikawa, T.: A Framework for Performance Evaluation Based on Event Tracing, *IPSJ Journal*, Vol.42, No.1, pp.68–78 (2001).
8) 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
9) OSS 技術開発・評価コンソーシアム，OProfile による PostgreSQL 8.1 と MySQL 5.0 の性能分析．http://ossipedia.ipa.go.jp/capacity/EV0603260042/index.php
10) Horikawa, T.: TinyTOPAZ: A Hybrid Event-Tracer for UNIX Servers, *Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pp.203–210 (1999).
11) 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
12) Doherty, S., et al.: Bringing Practical Lock-Free Synchronization to 64-bit Applications, *Proc. PODC '04*, pp.31–38 (2004).
13) Fraser, K. and Harriss, T.: Concurrent Programming Without Locks, *ACM Trans. Computer Systems*, Vol.25, No.2, pp.2–61 (2007).
14) 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
15) Handy, B. and Tolmer, J.: High Availability and Scalability Patches from Google. http://www.mysqlconf.com/mysql2009/public/schedule/detail/6903

(平成 21 年 11 月 18 日受付)
(平成 22 年 4 月 1 日採録)

堀川　隆（正会員）

1959 年生．1981 年同志社大学工学部電子工学科卒業．1983 年京都大学大学院工学研究科修士課程修了．同年日本電気（株）入社．以来，マイクロプロセッサやキャッシュ・メモリのアーキテクチャ，IT システムの性能評価技術の研究開発に従事．現在，同サービスプラットフォーム研究所サービスインテグレーションテクノロジグループ主幹研究員．2007 年度山下記念研究賞．