情報処理学会論文誌　プログラミング　Vol. 1　No. 3　11–18 (Oct. 2008)

マルチコア・システムにおける SAP Java アプリケーション・サーバのスケーラビリティ性能比較

上田陽平[†1]　小松秀昭[†1]　中谷登志男[†1]

Java アプリケーション・サーバのスケーラビリティを改善するために，単一マシン上で複数の Java 仮想マシン（JVM）を利用する手法が広く用いられている．本論文では SAP の Java アプリケーション・サーバ上で動作する SAP EP-ESS ベンチマークのスケーラビリティ性能を，2 種類のマルチコア・システムを用いて評価した．8 コアの Sun Niagara を 1 個搭載したシステムと，4 コアの Intel Clovertown を 2 個搭載したシステムの 2 種類の 8 コア・システムを使用し，それぞれのシステムで 1，2 および 4 個の JVM を用いた場合のスケーラビリティを比較した．単一 JVM の構成では，どちらのシステムでもロックコンテンションが発生し，6 コア以上では性能向上が見られなかった．複数 JVM 構成の場合は，Niagara において 2 JVM 構成が最高性能を達成したのに対し，Clovertown では 4 JVM 構成が最高性能を達成した．両者のマイクロ・アーキテクチャ性能を解析した結果，Niagara での性能劣化は，Niagara のキャッシュメモリと TLB が Clovertown と比較して小さいために生じたことが判明した．Niagara でより高いスケーラビリティを達成するためには，単一 JVM がよりスケーラブルである必要があり，我々はロックコンテンションを削減することによって，Niagara 上での単一 JVM の性能を 15%改善し，ほぼリニアなスケーラビリティを実現した．

Scalability Comparison of SAP Java Application Server on Two Multicore Systems

Yohei Ueda,[†1] Hideaki Komatsu[†1]
and Toshio Nakatani[†1]

To improve scalability of Java application servers, multiple JVM instances are often used on a single machine in production environments. In this paper, we studied scalability of a Java application server, SAP EP-ESS benchmark, on the two multicore systems, Sun's single-socket 8-core Niagara system and Intel's dual-socket quad-core Clovertown system, with three configurations of 1, 2, and 4 JVMs. For a single JVM, lock contention was observed on more than 6 cores in both systems. For multiple JVMs, 2 JVMs achieved the highest throughput on Niagara, while 4 JVMs achieved the highest throughput on Clovertown. We investigated the micro-architectural performance data, and we concluded that this is because Niagara comes with smaller caches and TLB, whereas Clovertown comes with larger caches and TLB. To achieve better scalability on Niagara, the application must be more scalable with a single JVM. We confirmed this by reducing lock contention using a Java concurrent library. As a result, we obtained a 15% performance gain on Niagara, and the throughput scaled almost linearly.

1. はじめに

将来のサーバ・システムにはマルチコア・プロセッサが不可欠となっている．2001 年に IBM が最初のデュアルコア・プロセッサを開発して以降[1]，Intel と AMD が 4 コアのプロセッサを投入し[2),3)]，最近では Sun Microsystems が 8 コアのプロセッサを製造している[4),5)]．Chip Multi-Processing（CMP）および Simultaneous Multi-Threading（SMT）の技術をさらに拡大し，大規模なマルチコア・プロセッサの開発を推し進めている企業もある．

マルチコアの流れは，低消費電力の観点から避けられないものであるが，マルチコア・プロセッサ上で既存のアプリケーションを実行すると，さまざまなボトルネックが発生し，十分な性能を発揮できないことが多い．そのため，マルチコア・プロセッサに適したスケーラブルなアプリケーションを構築するには，マルチコア・プロセッサの特性を理解し，それをアプリケーションの設計に反映させる必要がある．

マルチコア・プロセッサのコアの設計には simple core と complex core という 2 つの選択肢がある[6)]．典型的な simple core は比較的低い動作周波数で稼働し，in-order のパイプラインを持ち，小さなキャッシュメモリを持つものである．たとえば，Sun の UltraSPARC T1（Niagara）および UltraSPARC T2（Niagara 2）は simple core のマルチコア・プロセッサである．一方，典型的な complex core は，高い動作周波数で稼働し，out-of-order のパイプラインを持ち，大きなキャッシュメモリを持つものである．たとえば，IBM の POWER5 お

†1 日本 IBM 東京基礎研究所
IBM Tokyo Research Laboratory

よび POWER6，Intel Xeon 5300 Series（Clovertown）および Xeon 5400 Series（Harpertown）や AMD Opteron 8300 Series（Barcelona）は complex core のマルチコア・プロセッサである．これらのコアの性質の違いはアプリケーションのスケーラビリティに大きな影響を与えることが多々ある．

Java Application Server のスケーラビリティを向上させる手法として，1 台のマルチプロセッサ・システム上でも複数の Java 仮想マシン（JVM）を実行するという方法がある．これは，複数 JVM を用いると，ロックコンテンションなどの多くのボトルネックを簡単に削減することが可能なためであり，実際の商用環境でも広く用いられている手法である．

これまでに，マルチコア・プロセッサ上での Java ワークロードのスケーラビリティ性能解析はいくつか行われているが[7),8)]，それらは主に単一 JVM での性能に注目していた．一方，本研究は広く利用されている複数 JVM の構成でマルチコア上のアプリケーションの性能評価を主眼としているものである．我々は測定対象のワークロードとして，SAP の Java Application Server 上で稼働する SAP EP-ESS ベンチマークを選択し，我々はこれを Sun Niagara および Intel Clovertown 上でスケーラビリティ性能の評価を行った．このベンチマークは，実際の商用環境でも用いられているアプリケーションを基にした現実的な Java ワークロードであり，マルチコア・プロセッサの性能評価に適していると考えられる．

単一 JVM を用いた構成での測定では，Niagara と Clovertown の両方で激しいロックコンテンションが発生した．そこで，我々は複数 JVM を用いてロックコンテンションの削減を試みたところ，Niagara では JVM が 2 個のときにスループットが最高となったのに対し，Clovertown では 4 個のときが最高であった．解析の結果，Niagara で 4 JVM 構成の性能が劣化した原因は，増加したキャッシュミスが原因であることが判明した．1 JVM 構成と比べて 4 JVM 構成はメモリ・フットプリントが増加しており，それが Niagara の小さなキャッシュメモリはフィットしないため，キャッシュミスが増加していたのである．

Java アプリケーションのスケーラビリティを向上させるために複数 JVM 構成を用いることはキャッシュメモリが小さい simple core のシステムには適していない．むしろ，単一 JVM 構成でも Java アプリケーションがスケールするように設計する必要がある．

EP-ESS ベンチマークの場合では，ハッシュ表へのアクセスがロックコンテンションを引き起こしていたため，このハッシュテーブルへのアクセスを分散させ，コンテンションを削減したところ，コア数に対するスループットがほぼリニアにスケールするようになった．

2. ベンチマークの概要

SAP Enterprise Portal Employee Self-Service（EP-ESS）は SAP のアプリケーション・サーバ（SAP NetWeaver[9)]）上で稼働するアプリケーション・パッケージの 1 つであり，勤務時間記録や個人情報の登録・参照機能を従業員に提供する企業向け Web アプリケーションである．EP-ESS ベンチマーク[10)] は EP-ESS の実行性能を測定するものであり，SAP が提供するベンチマークのうち，唯一 Java コンポーネントのみを用いる．他のベンチマークは，SAP の独自言語である ABAP（Advanced Business Application Programming）ケーション・サーバへの Web リクエストは，別マシン上で稼働する負荷生成プログラム（Driver）により生成される．Driver は複数ユーザの Web トランザクションをシミュレートし，その応答時間を記録する．平均応答時間 2 秒以下で処理できる最大ユーザ数およびリクエストのスループットがベンチマークのスコアとなる．

Driver が生成する Web リクエストの遷移は次のとおりである．(1) ログオン，(2) 勤務時間の記録，(3) 休暇申請，(4) 休暇申請一覧，(5) 個人情報表示，(6) 住所表示，(7) 銀行口座表示，(8) 給与支払い状況，(9) ログオフ．(2) から (8) までの 7 個の画面については一定の回数ループ処理さる．各リクエスト間の待ち時間（Think Time）は 10 秒固定である．したがって，1 回のループに最低でも 70 秒必要である．ベンチマークの実施は，Ramp Up，Warm Up，Steady State，Ramp Down の 4 つのフェーズからなる．まず，Ramp Up フェーズでユーザ数を徐々に増やされ，Warm Up フェーズに入るとユーザ数は固定となる．そのまま一定時間経過するのを待ち，Steady State に入ると測定が開始され，スループットおよび応答時間が記録される．測定が終了すると Ramp Down フェーズに入り，ユーザ数が徐々に減らされてゆき，すべてのユーザのループが完了した時点で，ベンチマークも

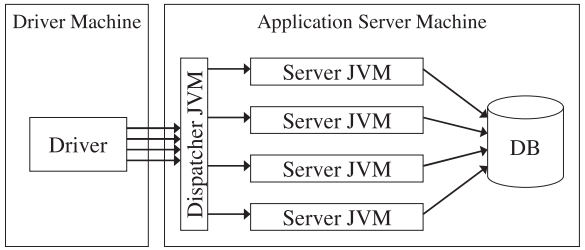

図 1　EP-ESS ベンチマークの構成
Fig. 1　Configuration of the EP-ESS benchmark.

終了となる．

図 1 は EP-ESS ベンチマークの構成を示している．Server JVM がアプリケーション・サーバの本体であり，EP-ESS が稼働している．Driver からの Web リクエストは Dispatcher JVM により複数の Server JVM に振り分けられる．Server JVM がアクセスするデータベースも同一マシン上に配置されている．

3. システム構成

3.1 ハードウェア

本研究では simple core と complex core の 2 種類のマルチコア・プロセッサ上で EP-ESS ベンチマークを動かし，さまざまな性能比較を行った．Simple core のプロセッサとして Sun UltraSPARC T1（Niagara）を使用し，complex core のプロセッサとして Intel Xeon E5345（Clovertown）を使用した．**表 1** は我々が使用した Niagara および Clovertown のキャッシュサイズなどの詳細である．

Niagara システムは，8 コアの Sun UltraSPARC T1 1.2 GHz を 1 個搭載した Sun Fire T2000 である．Niagara のコアは 4 個のハードウェアスレッドをサポートしており，システム全体では 32 個のハードウェアスレッドを持つ．1 つのコアの 4 つのスレッドが 16 KB の L1 命令キャッシュと 8 KB の L1 データキャッシュを共有しており，プロセッサ内の 8 個のコアは 3 MB の L2 統合キャッシュを共有している．このシステムは 32 GB の DDR2-533 SDRAM，10,000 RPM SAS ハードディスク，および Gigabit Ethernet ポートを搭載している．

一方，Clovertown システムは，4 コアの Intel Xeon E5345 2.33 GHz を 2 個搭載した IBM BladeCenter HS21 XM である．Clovertown は SMT をサポートしないため，システム全体で 8 個のハードウェアスレッドを持つ．1 つのコアは 32 KB の L1 命令キャッシュと 32 KB の L1 データキャッシュを持つ．プロセッサ内には 2 個のダイが含まれ，それぞれのダイが 2 個のコアを持っている．1 個のダイが 4 MB の L2 統合キャッシュを持っているため，ソケットあたりでは 8 MB，システムあたりでは 16 MB の L2 キャッシュを持つ．このシステムは 16 GB の DDR2-667 SDRAM，10,000 RPM SAS ハードディスク，および Gigabit Ethernet ポートを搭載している．

表 1　マルチコア・プロセッサの詳細
Table 1　Processor specifications.

	Niagara	Clovertown
CPU	UltraSPARC T1	Xeon E5345
動作周波数	1.2 GHz	2.33 GHz
ソケット数（システムあたり）	1	2
コア数（1 ソケットあたり）	8	4
SMT 数（1 コアあたり）	4	1
L1 キャッシュ（1 コアあたり）	命令：16 KB，データ：8 KB	命令：32 KB，データ：32 KB
L2 キャッシュ（1 コアあたり）	統合：3 MB	統合：8 MB
ITLB エントリ数（1 コアあたり）	64	128
DTLB エントリ数（1 コアあたり）	64	L0：16，L1：256

3.2 ソフトウェア

Java アプリケーション・サーバには SAP NetWeaver 7.0（2004s）を使用し，データベースには IBM DB2 Universal Database V8.2 を使用した．我々の実験ではアプリケーション・サーバとデータベースを同一マシンに配置する構成を採用したが，これは，EP-ESS ベンチマークではデータベースの CPU 使用率は小さく，構成も容易になることから，同一マシンに配置することが推奨されているためである．

SAP NetWeaver は，Dispatcher JVM がクライアントからの Web リクエストを Server JVM に割り振ることにより，複数の JVM プロセスをアプリケーション・サーバとして使用することができる．この Dispatcher JVM は，Server JVM が 1 つしかない構成でも必須となっている．

Niagara システムでは，OS に Solaris 10 を，JVM に Sun HotSpot Java VM 1.4.2（64 bit 版）を使用した．最新ではないこの JVM を選択したのは，SAP NetWeaver がこのバージョンを要求しているためである．8 GB の Java ヒープを用い，ゴミ集め（GC）手法は世代別 GC を使用した．世代別 GC の young 領域には全体の 25%を割り当て，残りを old 領域に割り当てた．また，Java ヒープには 4 MB のラージページを使用している．複数 JVM 構成の場合は，合計の Java ヒープが 8 GB になるように分割した．たとえば，2 JVM 構成の場合は Java ヒープを 4 GB に，4 JVM 構成の場合は Java ヒープを 2 GB に設定した．各種の性能測定には OS 付属のツールである vmstat，busstat，nicstat，cpustat などを用いた[11]．

Clovertown システムでは，OS として Red Hat Enterprise Linux 5（Linux 2.6.18）を，JVM として IBM J9 Java VM 1.4.2（64 bit 版）を使用した．Niagara とは異なるベンダの JVM を利用している理由は，Sun がこのバージョンの 64 bit 版 JVM を Linux 向けに提供していないためである．Java ヒープおよび GC のオプションの指定は，Niagara システムと同様である．ただし，このバージョンの J9 JVM は Linux ラージページをサポート

していないため，Java ヒープに標準の 4 KB ページを使用した．各種性能測定には，OS 付属の oprofile および nmon [12] を使用した．

4. 性能評価

まず，使用するコアの数を増やしながら，スループット性能のスケーラビリティを測定した．Niagara では，コアのハードウェアスレッドを 4 個すべて有効にして測定した．次に，Niagara システムにおいて，8 個のコアすべてを用いながら，ハードウェアスレッドの数を増やしてゆき，スループット性能のスケーラビリティを測定した．SMT をサポートしない Clovertown ではスレッド・スケーラビリティは測定していない．

4.1 コア・スケーラビリティ

図 2 は Niagara および Clovertown システムにおいて，EP-ESS ベンチマークを 2，4，6 および 8 コアで測定した結果である．Niagara ではコアあたり 4 個のハードウェアスレッドをすべて用いている．Clovertown では 2 および 4 コア測定時には 1 ソケットを，6 および 8 コア測定時には 2 ソケットを用いている．1 JVM 構成においては，Niagara で 6 コアから，Clovertown で 4 コアから性能向上の限界にほぼ達している．**図 3** は 1 JVM 構成時の両システムの CPU 使用率である．どちらのシステムでもコア数が増えると，ユーザ時間の割合が減少しており，Niagara ではシステム時間が，Clovertown ではアイドル時間が増加している．これが 1 JVM がスケールしない原因だと考えられる．

Java スレッドのスタックダンプと実行プロファイルを取得し，解析した結果，あるハッシュ表へのアクセスがロックコンテンションを引き起こしていることが判明した．原因のハッシュ表は java.util.Hashtable クラスのインスタンスであり，このオブジェクトへのアクセスには同期が必要であるが，この同期処理が 1 JVM 構成でのボトルネックとなっていた．これが原因でシステム時間やアイドル時間が増加していたのである．

図 2 に示されているとおり，複数 JVM 構成では 1 JVM と比較してスケーラビリティが向上しており，ハッシュ表のロックコンテンションが軽減されていることが分かる．複数 JVM 構成の性能を 8 コアでの結果で比較すると，Niagara では 2 JVM 構成が最高の性能を達成しているのに対し，Clovertown では 4 JVM が最高の性能を達成している．

Niagara が 4 JVM 構成で性能低下している原因を特定するために，我々はマイクロアーキテクチャ性能（**図 4**）を解析した．このグラフのキャッシュミス率は，1 インストラクションあたりではなく，1 リクエストあたりの数値を示している．これは，ロックコンテンションが存在する場合に，1 インストラクションあたりのミス率が減少してしまい，他の結果と

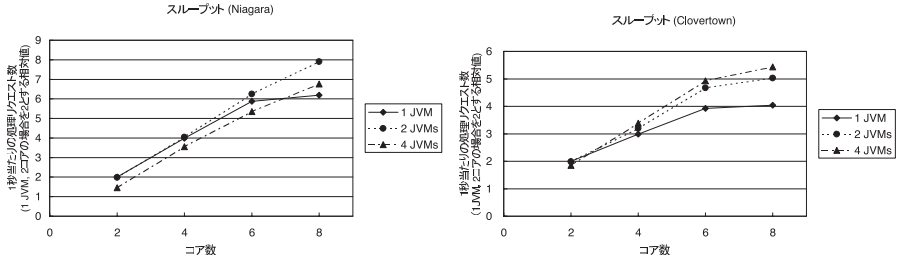

図 2　コア・スケーラビリティ
Fig. 2　Core scalability.

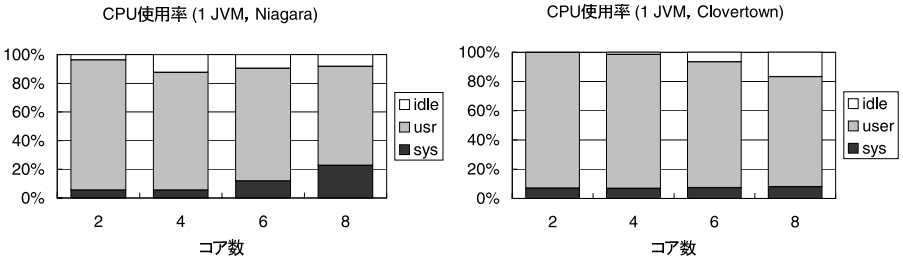

図 3　CPU 使用率
Fig. 3　CPU utilization.

の比較が難しいためである．この現象はロックコンテンション時にスピンロックが行われると，見かけ上，データおよび命令の局所性が向上するが，実際には有益な仕事は行っていないために発生する．

1 JVM 構成の Niagara の場合，コアが増えるごとに 1 リクエストあたりのサイクル数および命令数が増えているが，これはロックコンテンションが発生しているためである．2 JVM と 4 JVM を 8 コアで比較すると，1 リクエストあたりの命令数はほとんど同じであるにもかかわらず，サイクル数は 4 JVM の方が悪くなっている．これは，4 JVM 構成により増大したキャッシュプレッシャによるものと考えられる．このことは L2 命令キャッシュミス，L2 データキャッシュミス，D-TLB ミスのグラフからも明確に読み取れる．Niagara のキャッシュと TLB は小さいため，多くの JVM を使用することによるロックコンテンションの削減効果よりも，メモリ・フットプリントの増加の影響が大きいと性能が低下するのである．

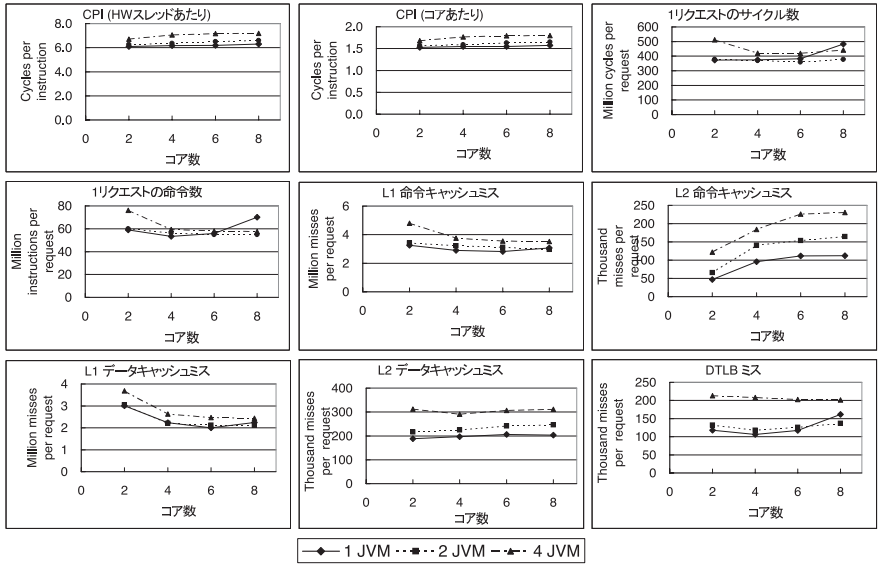

図 4　Niagara のマイクロアーキテクチャ性能
Fig. 4　Micro-architectural performance on Niagara.

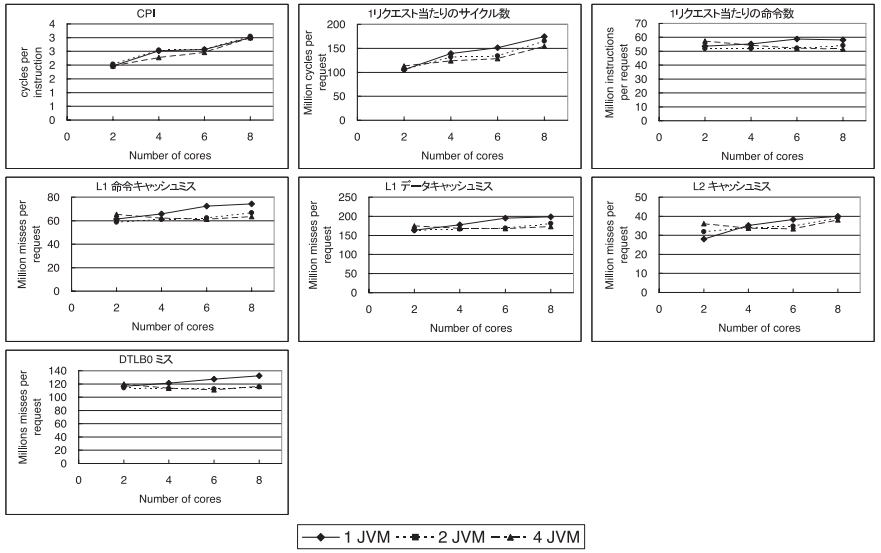

図 5　Clovertown のマイクロアーキテクチャ性能
Fig. 5　Micro-architectural performance on Clovertown.

4 JVM での L2 命令キャッシュミスは 1 JVM の 2 倍であり，L2 データキャッシュミスは 1.5 倍になっている．4 MB のラージページを使用しているにもかかわらず，4 JVM の D-TLB ミスは 2 JVM の 1.5 倍となっている．JVM プロセス間ではコードやデータが共有されていないために，このようなキャッシュプレッシャの増加が発生するのである．

一方，Clovertown では，スループットが最大になったのは 4 JVM 構成の場合である[⋆1]．**図 5** が Clovertown のマイクロアーキテクチャ性能であるが，Niagara で見られた JVM 数の増加によるキャッシュプレッシャの増加は見られない．JVM 数を 1 個から 4 個に増やした際に，1 リクエストあたりの命令数およびサイクル数が減少しているが，これは主にロックコンテンションの削減の影響によるものである．コア数が増えるにつれて L2 キャッシュミスが増加しているが，これは L2 キャッシュをコア間で共有しているためだと考えられる．

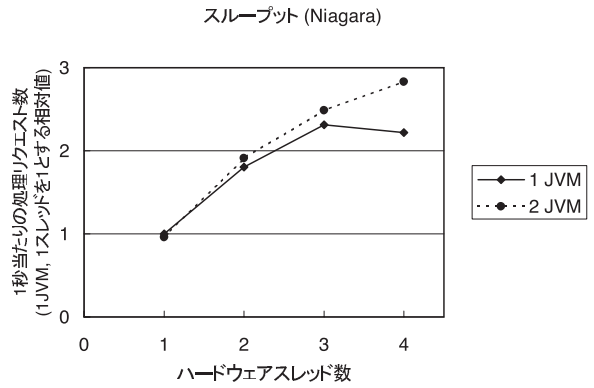

図 6　スレッド・スケーラビリティ
Fig. 6　Thread scalability.

⋆1 8 コアの Clovertown での追加実験によると，8 JVM 構成よりも 4 JVM 構成の方が高いスループットを達成した．

4.2　スレッド・スケーラビリティ

図 6 は，Niagara システムにおいて 8 個のコアすべてを使用しつつ，ハードウェアスレッドの数を増やしてゆき，スループットを測定した結果である．1 JVM 構成の場合では，3 スレッド目までは性能向上しているが，4 スレッド目を加えると逆に性能劣化が起きてしまっている．これは，すでに述べたとおり，ハッシュ表に対するロックコンテンションが原因である．3 スレッドでのスループットは 1 スレッドの場合の 2.2 倍であった．2 JVM 構成では，4 スレッド目を加えても，さらに性能向上が見られる．これはロックコンテンションが緩和され，本来の SMT の性能を発揮できるようなったためである．4 スレッドでのピーク性能は，1 スレッドのときの 3.0 倍であり，Niagara の SMT は EP-ESS ベンチマークに対してとても効果的であるといえる．

5.　スケーラビリティの向上

これまで述べてきたとおり，Java アプリケーションのスケーラビリティ向上のために単純に複数 JVM を用いる手法は simple core には適していない．この問題を解決し，simple core でスケーラビリティ向上を実現するためには，2 つの手法が考えられる．1 つは，アプリケーションのロックコンテンションを削減し，単一 JVM 構成のスケーラビリティを向上させる方法である．もう 1 つは複数 JVM 構成のメモリプレッシャを減らして，性能向上させる方法である．

5.1　単一 JVM のスケーラビリティ

一般に，アプリケーションのロックを削減すれば，スケーラビリティを向上させることができる．我々は，SAP EP-ESS ベンチマークの内部で使用されているハッシュ表へのアクセスがロックコンテンションを起こしていることを発見した．そこで，Javassist [13] を利用してバイトコード書き換えることにより，コンテンションを起こしていた Hashtable オブジェクトを Java 並列コレクションライブラリ [14] の ConcurrentHashMap に置き換えた．

図 7 は 1 JVM 構成における，オリジナルの性能と ConcurrentHashMap に置き換えた場合の性能を比較したグラフである．バイトコード書き換え前は，スループットが 6 コアで上限に達していたのに対し，書き換え後は 8 コアまでスループットの向上が見られ，ほぼリニアなスケーラビリティを実現している．書き換え前と比べてスループットは 15%改善されている．

このように，少ないコア数のシステムでは問題とならなかったアプリケーション・コードが，マルチコア・システムでボトルネックとして顕在化することはよくあることである．

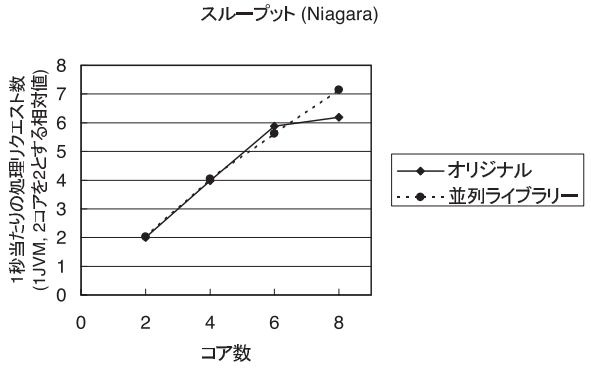

図 7　並列ライブラリによるロックコンテンション削減の効果
Fig. 7　Comparison between results without and with concurrent library.

Complex core においては，単純に複数 JVM 構成で解決できるが，simple core においては，ロックコンテンションを起こさない，正しいコーディングによって解決することが望ましい．

5.2　プロセッサアフィニティ

プロセスやソフトウェアスレッドを特定のハードウェアスレッドに割り当てるプロセッサアフィニティを用いることにより，メモリの局所性を高め，スケーラビリティを向上させることができる．我々は，Solaris 標準のコマンドである psrset コマンドを使用して各 JVM を一定のハードウェアスレッドの集合に割り当てる実験を行った．

図 8 は 4 JVM 構成のアプリケーション・サーバにプロセッサアフィニティを適用した際のハードウェアスレッド使用率である．Psrset コマンドの制限により，プロセスやスレッドをあるハードウェアスレッドに割り当てる場合，そのハードウェアスレッドはそのプロセスやスレッドに排他的に割り当てられてしまう．したがって，32 個あるハードウェアスレッドを 8 個ずつ 4 つの Server JVM に割り当ててしまうと，Dispatcher JVM やデータベースや OS のデーモンなど他のプロセスの実行ができなくなってしまう．そこで，4 個の Server JVM には 7 個のハードウェアスレッドを割り当て，残りの 4 スレッドで他のプロセスを動作させた．図 8 の 0，8，16 および 24 が，その他のプロセスのためのスレッドである．つまり，各 Server JVM には 1+3/4 個のコアが割り当てられ，残った 1/4 コアを 4 つ合わせた部分でその他のプロセスが実行される．この方法により，8 コア 4 JVM 構成のスループットをアフィニティなしの場合と比較して 7%向上させることができた．さまざまな

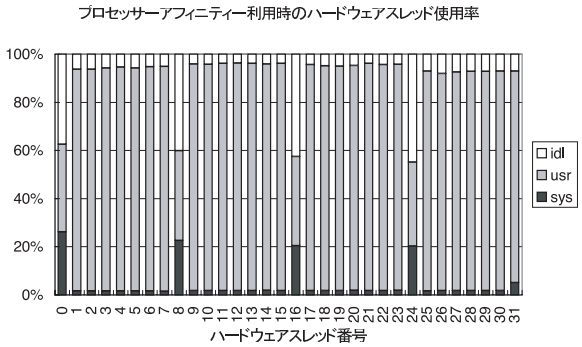

図 8　プロセッサアフィニティを利用した 4JVM 構成時のハードウェアスレッドの使用率
Fig. 8　Hardware thread utilization with processor affinity (4 JVMs).

構成を試した結果，これが最も性能が向上した構成である．しかし，アフィニティなしの 2 JVM 構成と比較すると，8%低いスループットしか出ていない．Psrset コマンドの排他的な割当てしかできないという制限のため，どうしても割当てに偏りが生じてしまい，図 8 の 0，8，16 および 24 番のスレッドに見られるようにアイドル時間が生じてしまうのである．Psrset コマンドが排他的でない割当てが可能であれば，偏りなくハードウェアスレッドへ割当てが可能となり，さらなる性能向上が可能であると考えられる．

6. 議　　論

複数 JVM 構成のメモリ・フットプリントを削減する手法として，JVM 間でデータの共有を行うことが考えられる．JVM は Java ヒープ以外にもクラス情報などのさまざまなメタデータを持っており，複数 JVM 構成の場合，これらのメタデータは JVM 間で重複しているものも多い．最新の IBM J9 JVM は JVM 間でクラスファイルを共有することが可能であるが，この機能は L2 データキャッシュミスの削減には効果があると考えられる．

命令キャッシュミスを削減するには，JVM 間で Just-In-Time コンパイルされたコードを共有する必要がある．IBM J9 JVM には Ahead-of-time コンパイルされたコードを共有する機能はあるが，Just-in-time コンパイルされたコードを共有する機能はなく，よりチャレンジングな問題と考えられる．

7. ま と め

本論文は Sun Niagara と Intel Clovertown という典型的な 2 種類のマルチコア・システムを用いて，SAP EP-ESS ベンチマークの性能分析を行った．単一 JVM 構成では，どちらのシステムでも 4 コア以上でロックコンテンションが発生した．複数 JVM 構成をとった場合，Niagara では 2 JVM 構成が最高性能を達成したのに対し，Clovertown では 4 JVM 構成が最高性能を達成した．この差異は，Niagara が小さなキャッシュや TLB を持つのに対し，Clovertown はより大きなキャッシュや TLB 持つために生じた．

Niagara でより高いスケーラビリティを実現するためには，単一 JVM 構成でもアプリケーションがスケールする必要がある．我々は Java 並列コレクションライブラリを用いて EP-ESS のロックコンテンションを削減することにより，単一 JVM 構成でも Niagara 上でスケールすることを確認した．

参 考 文 献

1) Tendler, S.F.H.L.J., Dodson, S. and Sinharoy, B.: POWER4 System Microarchitecture, IBM Technical White Paper (Oct. 2001).
2) Intel Corporation: Quad-core for servers. http://www.intel.com/quadcoreserver/
3) AMD Inc: AMD Multi-core Processors. http://multicore.amd.com/
4) Kongetira, P., Aingaran, K. and Olukotun, K.: Niagara: A 32-Way Multithreaded SPARC Processor, *IEEE Micro*, Vol.25, No.2, pp.1–29 (Mar./Apr. 2005).
5) Nawathe, U., Hassan, M., Warriner, L., Yen, K., Upputuri, B., Greenhill, D., Kumar, A. and Park, H.: An 8-core, 64-thread, 64-bit, power efficient SPARC SoC, *2007 IEEE International Solid-State Circuits Conference* (*ISSCC*) (Feb. 2007).
6) Olukotun, L. Hammond and Laudon, J.: *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*, Morgan & Claypool Publishers (2007).
7) Kaseridis, D. and John, L.K.: CMP/CMT Scaling of SPECjbb2005 on UltraSPARC T1, *10th Workshop on Computer Architecture Evaluation using Commercial Workloads* (Feb. 2007).
8) Tseng, J.H., Yu, H., Nagar, S., Dubey, N., Franke, H., Pattnaik, P., Inoue, H. and Nakatani, T.: Performance Studies of Commercial Workloads on a Multi-core System, *2007 IEEE International Symposium on Workload Characterization* (*IISWC*) (Sep. 2007).
9) SAP AG: SAP NetWeaver. http://www.sap.com/platform/netweaver/
10) SAP AG: SAP EP-ESS benchmark. http://www.sap.com/solutions/benchmark/ep-ess.epx

11) McDougall, R., Mauro, J. and Gregg, B.: *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*, Prentice Hall PTR (2006).
12) IBM: nmon performance: A free tool to analyze AIX and Linux performance. http://www.ibm.com/developerworks/aix/library/au-analyze_aix/
13) Chiba, S.: Javassist — A Reflection-based Programming Wizard for Java, *Proc. ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (Oct. 1998).
14) Lea, D.: Overview of package util.concurrent Release 1.3.4. http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

(平成 20 年 4 月 22 日受付)
(平成 20 年 7 月 7 日採録)

上田　陽平

1977 年生．2000 年東京大学理学部情報科学科卒業．2002 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同年日本アイ・ビー・エム（株）に入社．現在同社東京基礎研究所に勤務．並列・分散プログラミングの研究に従事．

小松　秀昭（正会員）

1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM 東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．

中谷登志男（正会員）

1975 年早稲田大学理工学部数学科卒業．同年日本アイ・ビー・エム（株）入社．1985 年米国プリンストン大学大学院コンピュータ・サイエンス学科より M.S.E. および M.A.，1987 年 Ph.D. を各取得．同年より同社東京基礎研究所においてプログラム最適化技術の設計・実装・評価の研究に従事．2000 年より IBM ディスティングイッシュト・エンジニア．現在，同研究所システムズ担当．基礎研究部門におけるコンパイラおよびエンド・トゥ・エンド・パーフォーマンスのストラテジストを兼任．2007 年より ACM ディスティングイッシュト・エンジニア．IEEE 会員．