Web サーバにおける Linux2.4 のスケーラビリティ
－ ボトルネック解析と改善 －

平井 聡，久門 耕一
(株)富士通研究所
E-Mail: {ahirai, kumon}@flab.fujitsu.co.jp

要旨

本論文では， Web サーバアプリケーション使用時における Linux2.4 システムのスケーラビリティについて述べる．評価システムには，8way の PC サーバ機を使用し，Web サーバへの負荷生成には WebBench ベンチマークテストを使用した．性能分析にあたっては，カーネルにプロファイル機能を付加し，OS 関数レベルのボトルネック解析を行った．

その結果，カーネル 2.4 は少なくとも 8CPU までスケーラブルであること，カーネル 2.4 はピーク性能でカーネル 2.2 の 2 倍以上の性能が出ること，カーネル 2.4 でも高負荷時（多プロセス動作時）にスケジューラのオーバヘッドにより性能劣化が発生すること，そしてスケーラビリティ改善プロジェクト（Linux Scalability Effort）の修正パッチ適用により高負荷時でも性能劣化を抑えられることが分かった．

１ はじめに

2001年1月に公開されたLinux2.4カーネルは，エンタープライズ向けのさまざまな機能の追加，改善が行われた．64GBメモリサポート（Intel系Pentium Pro以降でPAE対応プロセッサ），パフォーマンスの改善，ファイルサイズ制限の引き上げ，最大プロセス数および最大ユーザ･グループ数の拡大，rawデバイスのサポート，LVM（論理ボリュームマネージャ）のサポートなど，企業用途向けサーバに必要なさまざまな機能が強化された．これら強化項目の中でも，エンタープライズ分野における最大の特徴は，パフォーマンスの改善，特にマルチプロセッサ機における性能向上である．

Linuxカーネルは，バージョン2.2においても複数プロセッサに対応していたが，商用アプリケーションにおいてプロセッサを増やした際の性能向上率（スケーラビリティ）が悪いということが指摘されていた．特に1999年4月および6月にMindcraft社により公表されたベンチマークレポート[1][2]で，Webサーバおよびファイルサーバにおける性能測定においてWindowsNTとの性能差，特にマルチプロセッサ構成での性能差[1] が指摘され，カーネル2.2におけるスケーラビリティの問題[2] が明らかとなった [3] .

今回我々は，カーネル2.4におけるスケーラビリティを検証するため，8wayのPCサーバ機において，Webサーバアプリケーション（Apache）を使用した場合に焦点をあて，以下の分析を行っている．

- カーネル2.2および2.4は8wayサーバにおいてどの程度スケーラビリティがあるか．
- カーネル2.4は2.2に比べてどの程度パフォーマンスが向上したか．
- カーネル2.4は高負荷時（多プロセス動作時）にどのような問題が発生するか．
- スケーラビリティ改善プロジェクト（Linux Scalability Effort）[4]の修正パッチはどの程度効果があるか．

Webサーバへの負荷生成にはWebBench 3.0 [5]を使用した．また分析にあたっては，カーネルにプロファイル機能を付加し，OS関数レベルのボトルネック解析を行った．

本論文では，2章で評価環境を説明し，3章でカーネル2.2および2.4のベンチマーク性能結果と各々のカーネルにおけるボトルネック分析結果について述べる．4章では2.4カーネルにおける高負荷時の性能結果およびその際のボトルネックを考察し，5章ではスケーラビリティ改善プロジェクト（Linux Scalability Effort）の修正パッチによる改善を検証する．

---

[1] 4SMP で WindowsNT の 1/2〜1/3 の性能．

[2] 1CPU→4CPU で 1.2〜1.4 倍しか性能が向上しなかった．

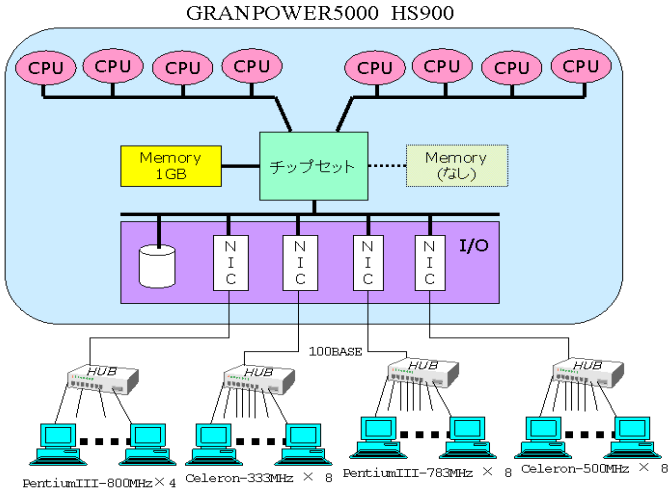

図1： 環境構成図

2 評価環境

2.1 ハードウェア構成

今回の評価に用いたハードウェア構成は図1に示す通りであり, サーバにはFujitsu GRANPOWER 5000(HS900)を使用した. 表1にサーバの構成を示す.

クライアントにはPC-AT互換機を28台使用し, 32台分の負荷をシミュレートした. ネットワーク構成は, クライアント28台を4系統に分け, 100BaseT Hub経由でサーバに接続した. なお, 本評価におけるクライアント1台あたりのネットワーク負荷は10Mbps程度であり, ネットワークによるボトルネックは発生しない.

表1：サーバ機－GRANPOWER5000(HS900)

CPU	Pentium-III Xeon 550MHz(1MB cache)×8
Memory	1GB
HDD	内蔵 SCSI 9GB×1
LAN	Intel 社 EtherExpress Pro/100×4

2.2 ソフトウェア構成

表2にサーバのソフトウェア構成を示す. カーネルは2.2系として2.2.19を, 2.4系として2.4.1を使用し, WebサーバはApache 1.3.9を使用した.

クライアントOSには, WindowsNT Workstation4.0 (SP5)を使用した.

表2：サーバソフトウェア

カーネル	2.2.19 / 2.4.1
ディストリビューション	RedHat Linux 6.1 US版
Web サーバ	Apache 1.3.9 (SMP 使用時の性能劣化を回避する SINGLE_LISTEN_UNSERIALIZED_ACCEPT オプション付で作成)

2.3 ベンチマークテスト

Webサーバへの負荷生成にはWebBench 3.0 [5]を使用した. WebBenchはWebサーバアプリケーションの性能評価を行うベンチマークである. 本評価では"static get"測定を実施し, 様々なサイズの静的なWebページ読出しをどれだけ処理できたか（1秒間あたりにサーバが処理したリクエスト数）の測定を行っている. 測定におけるコンフィグレーション等はMindcraft社のベンチマークレポート[2]の設定に準拠したが, ディスク容量の都合上アクセス・ログファイルの採取は行っていない. なお, Mindcraft社のベンチマークレポートではWebBench 2.0を使用しており, またハードウェア構成等も異なるため, 性能値の比較は行えないことに留意されたい.

図2は, カーネル2.2および2.4における4CPUでのWebBench（static get測定）実行時のUser/OS比率である. 本ベンチマークの特徴は, OSの実行時間比率がおよそ70%～90%であり, 性能の律速部分はWebサーバでなくドライバを含むOS部が主であるということである.

また, サーバメモリ1GBでの測定で, ベンチマ

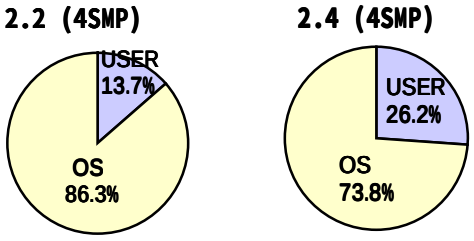

図2： WebBench User/OS実行時間比率

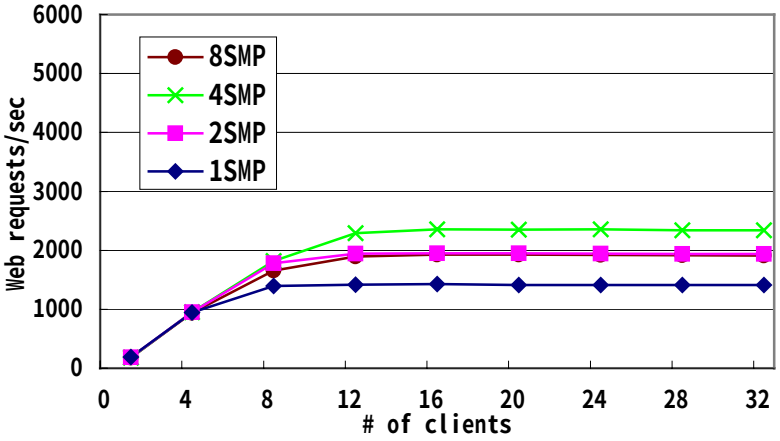

図3： WebBench性能（カーネル2.2）

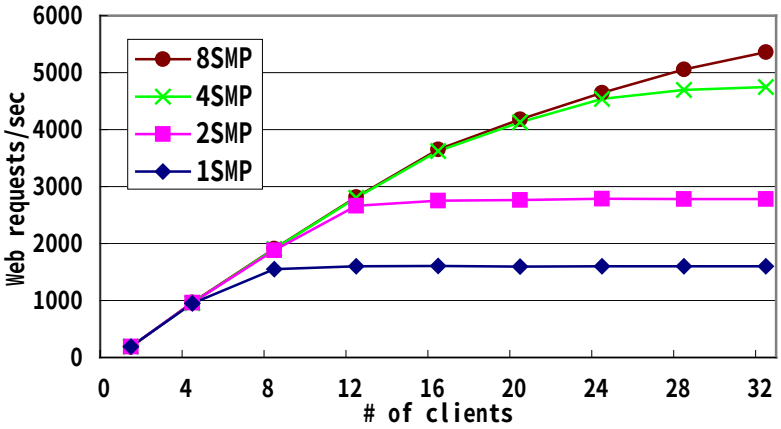

図4： WebBench性能（カーネル2.4）

表3：WebBench性能（性能比較表）

	ピーク性能 (Req/sec)	1SMP 比	カーネル 2.2 比
2.2 (1SMP)	1425	ー	ー
(2SMP)	1950	1.4	ー
(4SMP)	2358	1.7	ー
(8SMP)	1929	1.4	ー
2.4 (1SMP)	1605	ー	1.1
(2SMP)	2784	1.7	1.4
(4SMP)	4745	3.0	2.0
(8SMP)	5382	3.4	2.8

ーク値計測期間には読出しを行うWebページのファイルは全てファイル･キャッシュに載っており，ディスクアクセスはほとんどなく，ネットワーク処理およびプロセス･スケジューリングがテストの主である．

なお，本評価では1CPU構成においてもSMP（Symmetric Multi-Processor）カーネルを使用しており，以降1/2/4/8それぞれのCPU構成を1SMP，2SMP，4SMP，8SMPと記述する．

3 カーネル 2.2 と 2.4 の比較

3.1 スケーラビリティおよび性能比較

測定は，1/2/4/8SMPそれぞれについて，Webサーバへの負荷（リクエストを生成するクライアント・プロセス数）を増加させ，1秒あたりに処理したリクエスト数の比較を行った．1クライアント・プロセスあたり2スレッドでサーバへのリクエストを行う設定とし，最大でクライアント28台（32クライアント・プロセス）から64リクエストを同時発行する．

カーネル2.2の性能測定結果を図3に，カーネル2.4の結果を図4に，ピーク性能比較を表3に示す．図の横軸はクライアント・プロセス数，縦軸は1秒あたりにサーバが処理したWebリクエスト数である．

カーネル2.2は，スケーラビリティが非常に悪く，最も性能が高い4SMPで1SMPのピーク性能の1.7倍である．8SMPでは逆に性能が低下し，2SMPと同程度の性能しか出ない．

一方カーネル2.4は，絶対性能，スケーラビリティ共にカーネル2.2に比べて大幅に向上している．プロセッサ数が増加するに従いピーク性能は向上し，8SMPで1SMPのピーク性能の3.4倍である．（8SMPでは32クライアント・プロセスでも性能が頭打ちになっておらず，負荷不足でピーク性能には達していない．）

また，カーネル2.2と2.4のピーク性能の差は，1SMPでは1.1倍と大差はないが，8SMPでは2.8倍（以上）である．

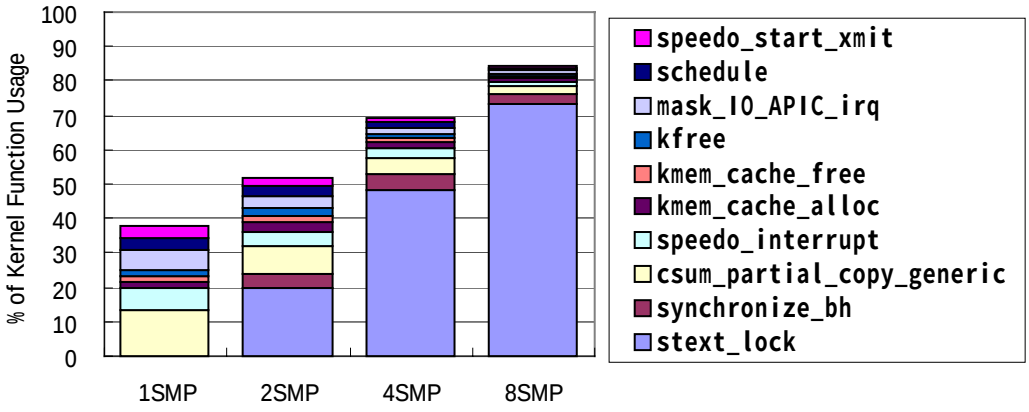

図5： OS関数プロファイル（カーネル2.2）

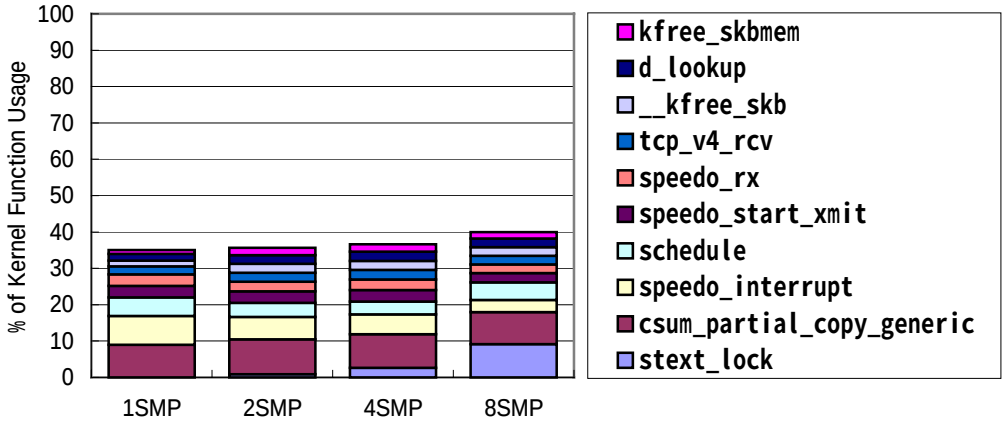

図6： OS関数プロファイル（カーネル2.4）

3.2 プロファイルによるボトルネック分析

カーネル2.2の問題点および2.4での改善状況を分析するために, カーネル内に一定時間ごとに実行命令アドレス等を記録するプロファイル機能を追加し，タイムベースサンプリングを行った．このプロファイル結果をもとに, カーネル内のどの部分がボトルネックとなっているか, またどの部分が改善されたかの分析を行った．

3.2.1 カーネル 2.2 のボトルネック分析

カーネル2.2でのWebBench実行時において，処理時間の多い10関数を抽出したものを図5に示す．

カーネル2.2では，プロセッサ数の増加と共に「stext_lock」のOS実行時間に占める割合が非常に高まっている．「stext_lock」は，カーネル内でのスピンロック競合時におけるロック待ち処理部である．1SMPでは競合が発生しないため0%であるが，2SMPで20%，4SMPで48%，8SMPでは73%とOS実行時間の大半がロック待ちで何も処理をしていないことがわかる．

なお，「stext_lock」は正確には関数でなく，カーネル内のスピンロック処理「spin_lock」[3] の一部である．図7に「spin_lock」の概念図を示す．

「spin_lock」は実際にロック獲得を行う部分と，ロック待ちを行う部分の2つに別れ，それぞれ異なるコード・セクションに置かれている．ロック獲得部はカーネル内で実際にロック獲得を行っている部分（「spin_lock」を記述している関数内）に展開され，ロック待ち処理部は「stext_lock」という特別なセクションにロック獲得部に1対1で対応した処理が固められて配置されている [4]．

本論文ではロック待ち処理に関しては個別の処理とみなし，「stext_lock」全体を1つの処理手続き（関数）として分類している．

[3] カーネル 2.2 ではマクロ，カーネル 2.4 ではインライン関数で実装されている．

[4] Intel系 32bit アーキテクチャ最適化手法のひとつで，通常のコードパス（ロック獲得成功時のルート）を直線的に走行させる（条件分岐をフォール・スルーさせる）よう配置することで静的分岐予測ミスを削減し，かつ余分な命令コードのプリフェッチを防いでいる．

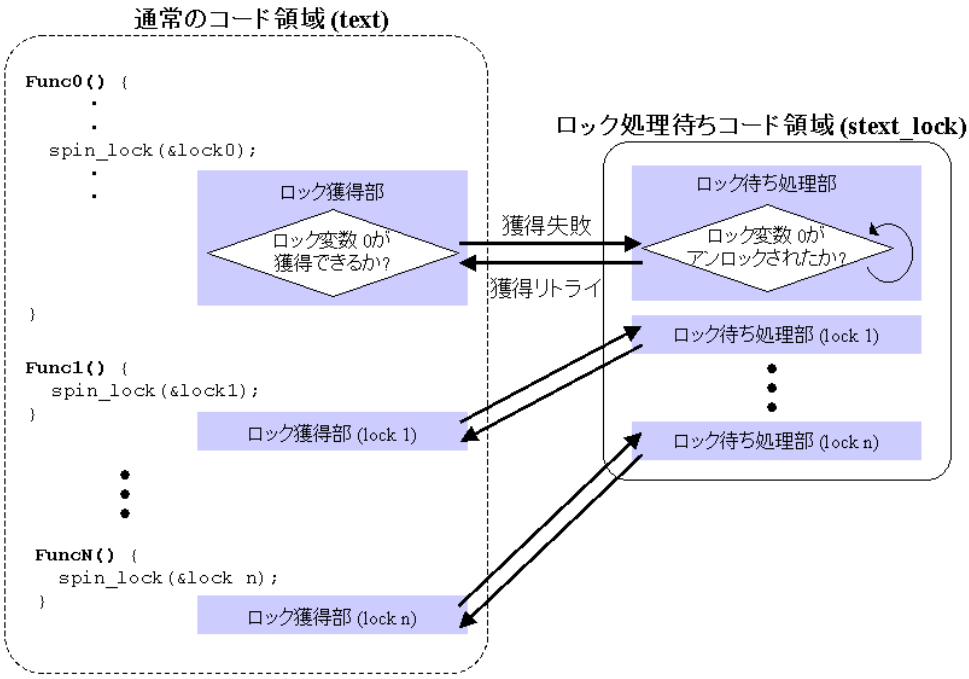

図7：「spin_lock」概念図

表4は，この「stext_lock」でのロック待ち処理時に, 待ちの対象となったロック変数の内訳である．ロック待ち時間の長い5変数について，全ロック変数のロック待ち時間総計に対する割合を示している．

表4：競合ロック変数（カーネル2.2）

ロック変数名	2SMP	4SMP	8SMP
kernel_flag	97.7%	99.0%	99.6%
irq_controller_lock	1.1%	0.4%	0.1%
runqueue_lock	0.7%	0.3%	0.1%
struct speedo_private.lock	0.3%	0.1%	0.1%
struct kmem_cache_t.c_spinlock	0.2%	0.1%	0.1%
その他	0.1%	0.1%	0.0%

この表から, カーネルのグローバル・ロック変数である「kernel_flag」が97%以上であることがわかる．また，図5のプロファイル結果と合わせて，8SMPでは実にOS実行時間の7割以上が「kernel_flag」をロックするための待ち処理（ビジー・ウェイト）であることがわかる．

カーネルのロック機構に関して，カーネル2.0では全てのカーネル処理（システムコール，割込み処理等）は1つのプロセッサ上でしか実行しないように，唯一のロック変数「kernel_flag」で排他制御を行っていた．これに対しカーネル2.2では，OS内部の処理を一部並列に行えるように，割込みコントローラへのアクセス(表4での「irq_controller_lock」)や，タスク・スケジューリングでの実行待ちキュー操作（表4での「runqueue_lock」）などカーネル処理全体を排他制御する必要がないリソースへのアクセスに対し，新たなロック変数を設けた．

しかし, カーネル2.2のロック粒度ではWebBenchにおいてはOS内での並列動作が十分行えず，またWebBenchではOSの実行時間が7割以上を占めることから, 非常にスケーラビリティが悪い結果となった．

3.2.2 カーネル 2.4 のボトルネック分析

カーネル2.4でのWebBench実行時において，処理時間の多い10関数を抽出したものを図6に示す．

カーネル2.4では，プロセッサ数の増加に伴い急激に増加するようなボトルネック関数はない．「stext_lock」に関しては，プロセッサ数に伴い増加しているものの, 2SMPでOS実行時間の0.9%, 4SMPで2.7%，8SMPでは9.1%であり，カーネル2.2と比べると大きく改善されている．

カーネル2.4のロック機構は，カーネル2.2で一部行った共有リソース毎にロック変数を分けるという手法をさらに進め, より粒度の細かいロック機構を実現している．この修正により，カーネル2.4ではロック待ち処理が8SMP時でもOS実行時間の1割以下となり，複数プロセッサによるOS内での並列動作が可能となったことで，WebBenchにおけるスケーラビリティが大幅に向上した．

なお，カーネル2.4でも一部グローバル・ロック変数「kernel_flag」を使用した排他制御を行っているが，本ベンチマーク測定における「kernel_flag」の競合はロック待ち処理時間全体の0.1%以下であり，全く問題にならないレベルとなった．

4 カーネル2.4 高負荷時の分析

次に，カーネル2.4でサーバ上で多プロセスが動作した場合に性能が変化するかの検証を行った．Webサーバにあてはめると，多数台のクライアントからCPU処理能力以上の要求が来た場合にどのように性能が変化するかといった評価である．理想的には，プロセス数が増加しても性能劣化することなく，ピーク性能（Webリクエスト処理数）を維持することが望ましい．

4.1 高負荷時の性能

今回のWebBench測定では，クライアント28台（32クライアント・プロセス）固定で，各クライアント・プロセスから要求するリクエストを4倍（2thread/client→8thread/client）にした場合の性能を測定した．Apacheは各クライアント（スレッド）からの要求に対応したプロセスを1対1で生成するため，32クライアント稼動時にサーバ上で最大64→256の4倍のプロセスが同時動作するようになる．なお，1/2/4CPU構成の場合，2thread/client時にすでにCPU負荷が100%であるため（8CPU構成では3%のIdleあり），8thread/clientにしても性能の向上はない．

図8に2thread/clientを1としたときの8thread/clientの性能を示す．

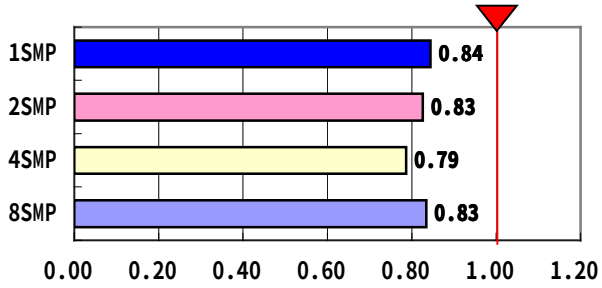

図8： 高負荷時の性能劣化（カーネル2.4）

全てのCPU構成において，8thread/client時の性能は2thread/client時に比べて20%前後劣化する結果となった．なお，プロセッサ数の増加に伴い性能劣化の比率は高まる傾向にあるが，8SMPに関しては2thread/client時に3%のIdleがあったため劣化の比率が低く見えている．

4.2 高負荷時のプロファイル分析

性能劣化の分析を行うために，前章と同様にプロファイルコードを用いたタイムベースサンプリングを行った．これを基に，OS関数のうち処理時間の増加した関数を分析したところ，「stext_lock」および「schedule」の2つが突出していることが判明した．図9は1Webリクエストあたりの処理時間を上記2関数を中心として分類したものである．

2thread/client 時と比較して8thread/clientでは処理時間全体に占める「stext_lock」および「schedule」の割合が非常に高くなっている．特に「stext_lock」に関しては，プロセッサ数の増加と共に処理時間が増加し，8SMPではUSER，OSを含めた処理時間全体の40%を占めている．

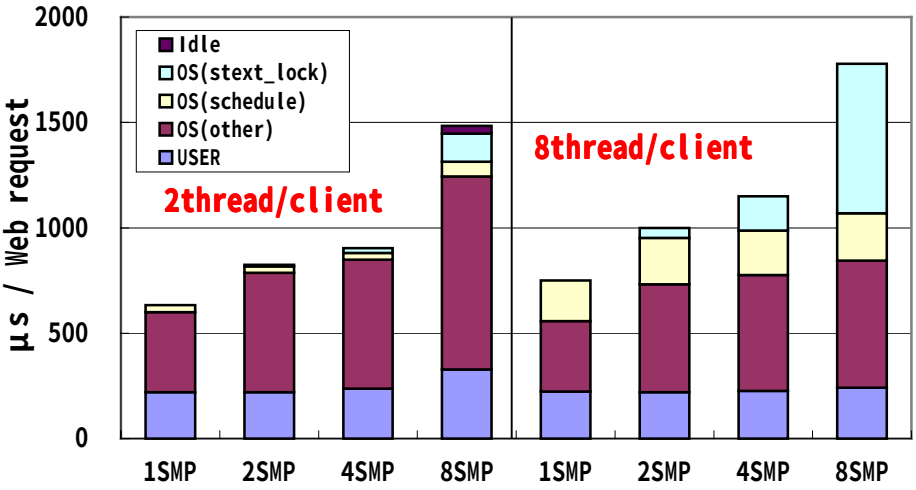

図9：「stext_lock」「schedule」処理時間

表5は，8SMPでの8thread/client時の競合ロック変数内訳である．ロック待ち時間の長い5変数について，全ロック変数のロック待ち時間総計に対する割合を示している．

表5：競合ロック変数（8SMP 8thread/client）

ロック変数名	比率
runqueue_lock	83.4%
struct sock.lock.slock	12.7%
dcache_lock	2.5%
files_lock	0.4%
inode_lock	0.3%
その他	0.7%

この表から，「runqueue_lock」変数が8割以上を占めていることがわかる．この変数は，スケジューラにおいて，実行しているあるいは実行可能状態であるプロセスの待ち行列（RUNキュー）を操作する際に獲得するロック変数である．このことから，カーネル2.4のサーバ上で多プロセスが動作した場合，スケジューラがネックとなって性能が低下する，特に「RUNキュー」の操作時のロック確保処理がネックとなっていることが分かる．

4.3 スケジューラの問題点

Linuxのスケジューラは，実行可能プロセス（Linuxのスケジューラではプロセスとスレッドを全く同じに扱うため単にプロセスと記述する）を単一キューで管理するという特徴を持っている．以下にその機能概要と問題点について述べる．

Linuxではプロセス毎の固有データを「task_struct」という構造体で管理している．この構造体には，プロセスID（pid）や状態（state），プロセス毎のメモリ管理構造体へのポインタ，あるいはスケジューリングにおける優先度やCPUタイムスライスなどプロセス管理における様々なデータが格納されている．Linuxのスケジューラは，図10に示すように実行しているプロセスおよび実行可能となったプロセスを「task_struct」への双方向リスト（正確にはtask_struct内のrun_listメンバへの双方向リスト）で管理している．このリストは単一のキュー構造（線形リスト構造）で「RUNキュー」を構成する．

この単一キュー構造は，スケジューラをシンプルに実現でき，実装も容易である．しかし，大規模構成システムにおいては，以下の2点の問題を持つ．

- プロセス数の増加に伴いキューが長くなり，リスト走査に時間がかかる（「schedule」の処理時間が増加する）．この結果として，キューに対する排他アクセスのためのロック保持時間が長くなる．
- プロセッサ数の増加に伴い，複数プロセッサによるロック競合の確率が高まる．

2thread/client 時と比較して8thread/client時に性能が劣化したのは，上記2つの問題が発生したためである．つまり，「RUNキュー」に連結される実行可能プロセスが増加することにより，1回のリスト走査（プロセス・スケジューリング）にかかる時間が増加し「schedule」の処理時間が増加したことに加え，複数プロセッサ構成では「runqueue_lock」に対するロック競合が発生し，「stext_lock」の処理時間が増加した．

今後Linuxの大規模システムへの適用においてプロセッサ数およびプロセス数が増加した場合に，この「RUNキュー」の構造が第一のボトルネックになることが予想される．

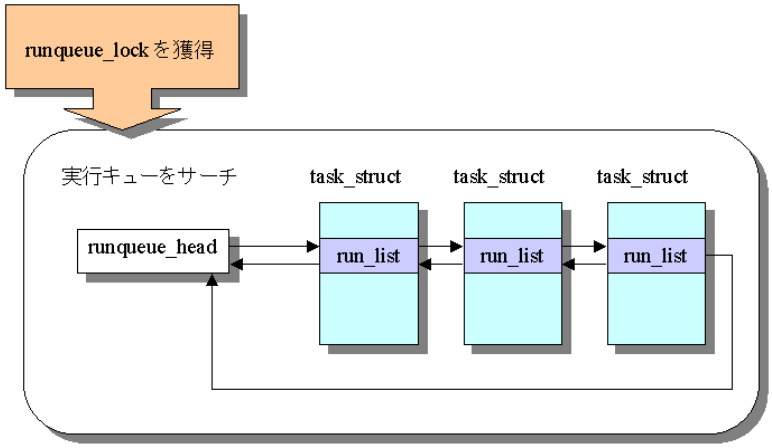

図10：「RUNキュー」の構造

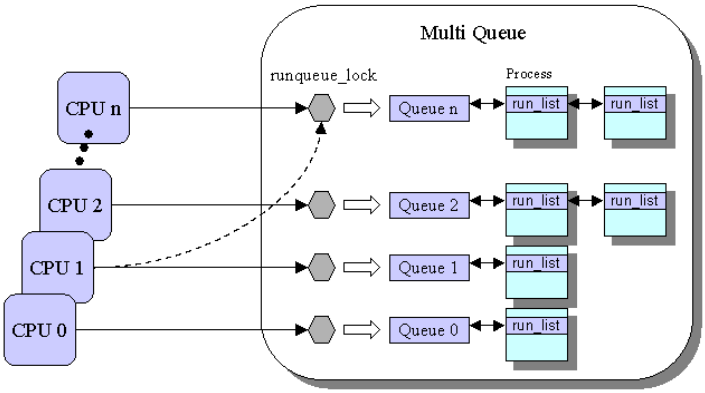

図11：multi-queue schedulerの構造

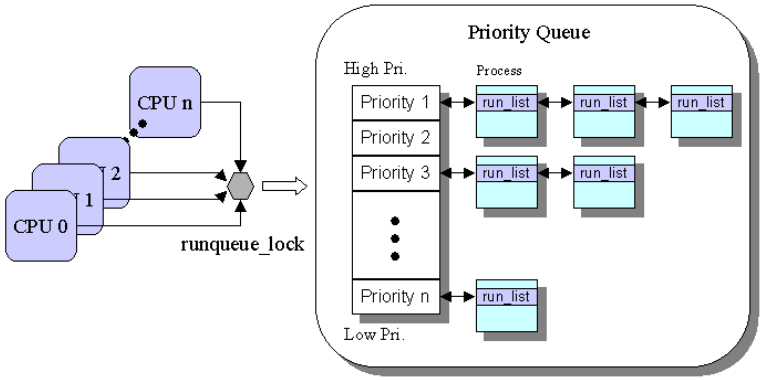

図12：priority-queue schedulerの構造

5 高負荷時ボトルネックの改善

スケジューラを含めたスケーラビリティの改善を目的とし, IBM社を中心としてオープンプロジェクト「Linux Scalability Effort」[4]が発足し, メーリングリストを中心とした活動が行われている. このプロジェクトで公開されているパッチを用い, 高負荷時のボトルネックがどの程度解消できるかの検証を行った.

5.1 スケジューラ修正パッチ

今回の検証では, プロジェクトで公開されているスケジューラ修正パッチのうち, 以下のものを適用し性能評価を行った.

① multi-queue scheduler パッチ
② priority-queue scheduler パッチ

双方のスケジューラ共にオリジナル・スケジューラのセマンティックスを維持し, 極力少ない修正量でスケーラビリティを向上させることを意図している [6] [7] .

multi-queue schedulerは, 図11に示すように「RUNキュー」を各プロセッサ毎に分割して管理することを特徴としている[5]. プロセッサ間のプロセス移動（プロセス・マイグレーション）のためにキュー毎に設けたロック変数による排他制御は必要となるが, 基本的にはロック競合がおこらず並列動作が可能である[6]. また, 同一プロセス数に対してはプロセッサ数が多いほど複数キューに分散され短くなるという特徴がある.

priority-queue schedulerは, 図12に示すように「RUNキュー」をプライオリティ毎（32段階）に分割して管理することを特徴としている. スケジューリング時のキュー走査は, 実行待ちプロセスの存在する最も高いプライオリティキューのみ行われる. ロック変数に関しては1つのみでありオリジナル・スケジューラと変わらないが, プライオリティ（nice値と残りタイムスライスにより決定）により連結されるキューが分散されるため, キューの走査時間を短く抑えることができ, 多プロセス動作時のロック保持時間の増大を抑えることができる.

[5] realtime タスクに対しては multi-queue scheduler, priority-queue scheduler 共に別キューを用意している.
[6] キュー毎に未スケジュール・プロセスの最高プライオリティ値を保持しており, その値の比較において他キューの優先度が高く, かつ該当キューがロックされていない場合のみマイグレーション処理が行われる.

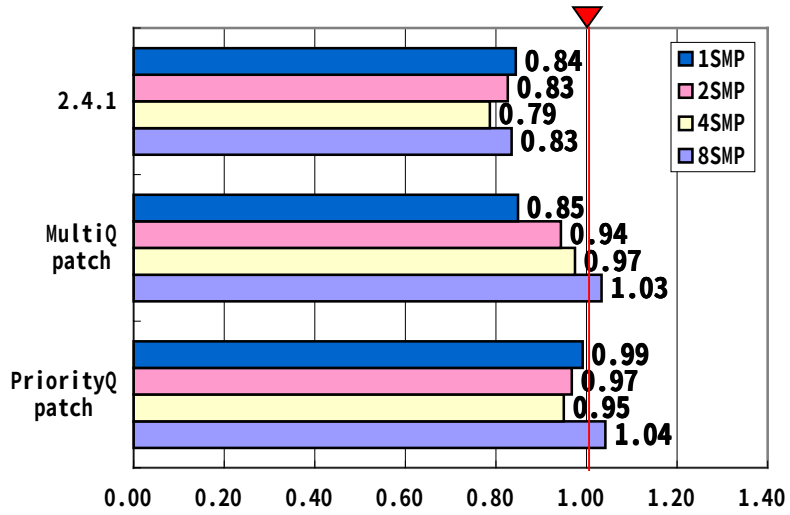

図13：パッチ適用時の性能劣化

5.2 スケジューラ修正パッチの性能

「4.1 高負荷時の性能」と同様の条件でそれぞれのパッチを適用して性能を測定した．図13は，各カーネルでの2thread/client を1としたときの8thread/client における性能である．なお，2thread/client時においてはスケジューラの違いによる性能差はほとんどない（0~+2%）．

オリジナルのスケジューラに比べて，双方のスケジューラ共に性能劣化を抑えられていることが分かる．multi-queue schedulerでは，プロセッサ数の増加に伴いキューの長さが短くなるので，プロセッサ数が多いほど効果が大きい．一方priority-queue schedulerではプロセッサ数に関係なく性能劣化を抑えている．なお，8SMPでは2thread/client 時に3～5% の Idle があるため，8thread/clientでは若干性能が向上している．

5.3 スケジューラ修正パッチのプロファイル分析

プロファイルによる分析では，8thread/client時にオリジナル･カーネルで特に処理時間が増加した「stext_lock」および「schedule」を抽出し，それぞれのスケジューラによる効果を調べた．図14に「stext_lock」の処理時間の推移を，図15に「schedule」の処理時間の推移を示す．グラフ凡例のカッコ内の2，8はそれぞれ2thread/client ，8thread/clientを表す．

- 「stext_lock」**について**

オリジナルのスケジューラに比べて双方のスケジューラ共に大きく処理時間が減少しており，「RUNキュー」の分割による効果が大きいことがわかる．さらに「runqueue_lock」の競合を分析するため，各スケジューラでの2/4/8CPU構成における1リクエストあたりの「runqueue_lock」ロックにおける待ち処理時間（8thread/client時）を比較したものを表6に示す．なお，multi-queue schedulerについては，プロセッサ毎に分割されたロック変数における待ち処理時間の総計である．

表6：runqueue_lock待ち処理時間（us/Request）

	2SMP	4SMP	8SMP
original scheduler	41.0	145.1	591.6
multi-queue scheduler	0.8	0.5	1.1
priority-queue scheduler	3.7	14.0	30.4

この表から，multi-queue scheduler は「runqueue_lock」の競合が非常に少なく，プロセッサ数の増加によるロック待ち処理時間の増加もほとんどないことがわかる．また，priority-queue schedulerに関しては，プロセッサ数の増加によりロック待ち処理時間が増加しているものの，オリジナル・スケジューラの1/10以下に抑えられていることがわかる．

- 「schedule」**について**

multi-queue schedulerはプロセッサ数の増加に伴い処理時間が減少しているが，これは，プロセッサ数と同数の複数キューに分割され，一つのキューの長さが短くなったためである．一方priority-queue schedulerに関してはプロセッサ数に応じたキューの長さの変化はないため，プロセッサ数とは無関係に一定の処理時間がかかっている（オリジナルのスケジューラも同様）．

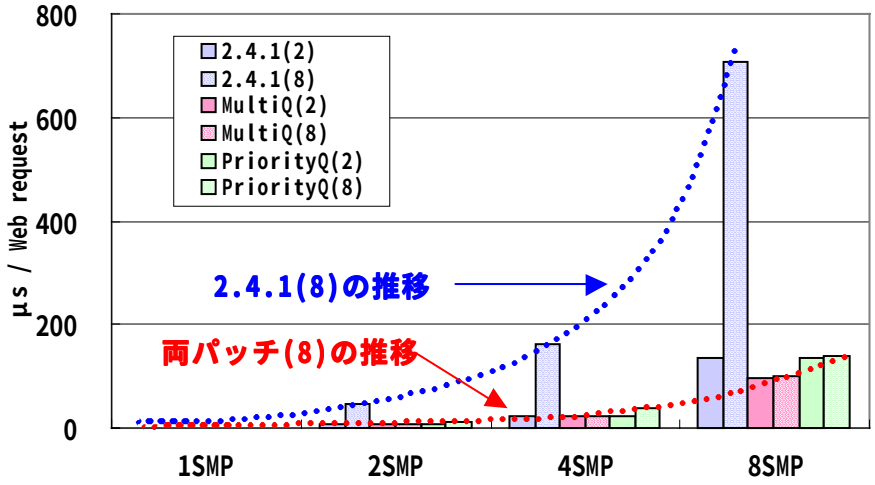

図14：パッチ適用時の「stext_lock」処理時間

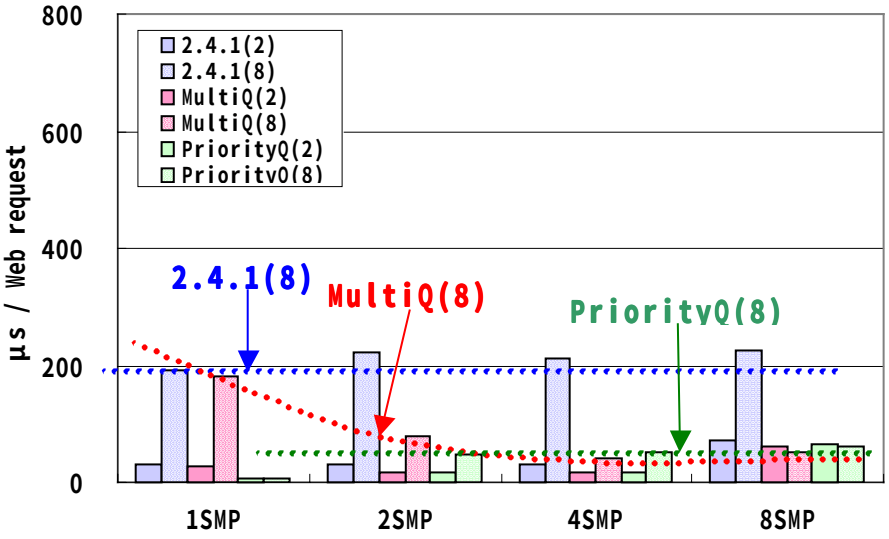

図15：パッチ適用時の「schedule」処理時間

これらの結果から，multi-queue schedulerおよびpriority-queue schedulerは，双方共にWebサーバでの高負荷時（多プロセスが動作時）の場合に非常に有効であり，特にプロセッサ数が少ない場合はpriority-queue schedulerが，プロセッサ数が多い場合はmulti-queue schedulerが有効であるといえる．

5.4 スケジューラ以外のボトルネック分析

図14の「stext_lock」の処理時間の推移を見ると8SMPでは全てのスケジューラ（カーネル）において処理時間，つまりロックの競合が増加している．表7は，8SMP 8thread/client時にmulti-queue schedulerおよびpriority-queue schedulerが要した1リクエストあたりのロック待ち処理時間，および4SMPからの増加率を示している．

この表から，multi-queue schedulerおよびpriority-queue schedulerを使用したカーネルにおいては「runqueue_lock」の競合よりも，「sock.lock.slock」（sock構造体内のロック変数），および「dcache_lock」の競合が多く，また増加率も高いことがわかる．

表7：競合ロック変数（8SMP-8thread）

ロック変数名	MultiQ		PriorityQ	
	us/Req	4SMP比	us/Req	4SMP比
struct sock.lock.slock	43.8	5.2	52.8	6.5
dcache_lock	40.4	4.4	39.7	4.4
runqueue_lock	1.1	2.2	30.4	2.2
files_lock	5.6	4.2	5.5	4.6
inode_lock	3.7	4.5	3.5	4.0
その他	8.7	3.9	7.8	3.5

「sock.lock.slock」は, ソケット管理構造体(sock構造体）に関する排他制御用ロック変数である．本ロック競合の増加は，4系統のネットワーク全てに対して単一のソケット（http:80）で待受けを行ったことによる．なお,「sock.lock.slock」の競合元は,「tcp_v4_rcv」関数および「tcp_accept」関数であり，それぞれ受信処理の入口とaccept処理を行う関数である．これら2つの関数で競合の約70%を占めていた．

もう一方の「dcache_lock」は，ファイルパス名等の情報を格納したディレクトリ･エントリへのアクセスに対する排他制御用ロック変数である．本ロック競合の増加は，ディレクトリ・エントリに対する排他制御機能（ロック変数）がシステム全体で一つしかなく, 多数のWebリクエストによるファイル・アクセス要求（ディレクトリ・エントリ検索)によってこの部分がボトルネックとなり発生した．なお,「dcache_lock」変数の競合元は，ディレクトリ・エントリの検索を行う「d_lookup」関数が大半であり，競合の約60%を占めていた．

これらのロック競合に関しては, さらに大規模構成になった場合に問題になってくると思われ, ロック変数の細粒度化などカーネル内部のさらなる最適化の検討が必要である．

6 おわりに

本論文では，カーネル2.2と2.4の性能およびスケーラビリティ比較，カーネル2.4での高負荷時の性能劣化, スケジューラパッチによる改善効果について, 実測とカーネルプロファイルにもとづく分析を行った．

その結果，2.2カーネルに比べて2.4カーネルは絶対性能, スケーラビリティ共に向上していること, 高負荷時にスケジューラのオーバヘッドにより性能劣化が起こること, そしてスケーラビリティ改善プロジェクトの修正パッチ適用によりその劣化を抑えられることが明らかとなった．

今後は，ファイルサーバやOLTP等，より広範なアプリケーションを用いた性能検証, ボトルネック分析を行うとともに, 分析データおよび改善手法のコミュニティへの公開を行っていく予定である．

参考文献

[1] Web and File Server Comparison: Microsoft Windows NT Server 4.0 and Red Hat Linux 5.2 Updated, http://www.mindcraft.com/whitepapers/first-nts4rhlinux.html

[2] Open Benchmark: Windows NT Server 4.0 and Linux, http://www.mindcraft.com/whitepapers/openbench1.html

[3] On Mindcraft's April 1999 Benchmark, http://www.kegel.com/mindcraft_redux.html

[4] Linux Scalability Effort, http://sourceforge.net/projects/lse

[5] WebBench, http://www.zdnet.com/etestinglabs/stories/benchmarks/0,8829,2326243,00.html

[6] Implementation of a multi-queue scheduler, http://lse.sourceforge.net/scheduling/mq1.html

[7] Implementation of a priority queue scheduler, http://lse.sourceforge.net/scheduling/PrioScheduler.html