ウェブアプリケーションサーバの Degradation Scheme の制御に向けて

光来 健一　　日比野 秀章　　松沼 正浩　　千葉 滋

東京工業大学大学院 情報理工学研究科 数理・計算科学専攻

要 旨

graceful degradation はウェブアプリケーションサーバが高負荷時でもサービスを提供し続けるために重要である。複数のサービスを提供するアプリケーションサーバの場合、それぞれのサービスは要求される品質に応じて異なる度合いで性能を低下させなければならない。本稿では、我々が *degradation scheme* と呼んでいる高負荷時の各サービスの性能低下の度合いは、ミドルウェアで制御が行われなければ OS 間で異なることを報告する。我々の実験結果によると、いくつかの OS では高負荷時に特定のサービスの性能が著しく低下することが分かった。さらに、ミドルウェアで degradation scheme を制御できるようにするために、OS 間での degradation scheme の違いの主な要因についても調べた。

1　はじめに

ウェブアプリケーションサーバは商用ウェブサイトにおいて複雑なウェブアプリケーションを構築するのに使われる。このようなサーバでは Java サーブレットなどによって動的なページが生成されるため、サーブレットの実行性能が重要になる。特に、サーバが高負荷になった時には、スループットを維持するためにリクエストへの応答時間を長くするといった graceful degradation を行うことが望ましい [8]。graceful degradation を行なうことにより、高負荷時でもサービスそのものが利用できなくならないようにすることができる。

アプリケーションサーバが複数のサービスを提供している場合、高負荷時の各サービスの性能低下のさせ方には様々な方法が考えられる。全てのサービスの性能を一律に低下させたい場合もあれば、特定のサービスの性能を優先させたい場合もある。高負荷時におけるこのような複数のサービスの性能低下のさせ方を我々は *degradation scheme* と呼んでいる。degradation scheme は実行環境に依存しており、アプリケーション開発者が彼らの実行環境で各サービスの性能のバランスをチューニングしたとしても、ユーザの実行環境では最適なチューニングにならない可能性がある。このような問題を解決するために、実行環境による degradation scheme の違いはミドルウェアで吸収できるようにするべきである。

本稿では、ミドルウェアで制御を行わなければ、異なる OS を用いた場合に異なる degradation scheme を示すことを報告する。我々は対象 OS として、Solaris、Linux、FreeBSD、Windows Server を用い、Linux については 3 つのバージョンで実験を行った。これらの OS について Tomcat 上でサーブレットとして実装された様々な重さのサービスを同時に動かし、重いサービスによる負荷を高めながら各サービスのスループットを測定した。その結果、軽いサービスのスループットは Solaris ではゆっくり低下したのに対し、他の OS では急激に低下するといった違いが見られた。また、Linux についてはわずかなバージョンの違いであっても degradation scheme が大きく異なる場合があることが分かった。

ミドルウェアで degradation scheme を制御できるようにするために、我々はカーネルレベルの詳細な実験を行い、Solaris と Linux の間での degradation scheme の相違の要因を調べた。我々の実験結果によると、最も大きな要因は Linux の方が Solaris よりロック待ち時間が長いことであった。その理由は、(1) Linux スレッドは実装上、ロックを獲得する時にブロックしやすいこと、および、(2) Linux のスレッドスケジューリングのポリシーは我々の用いたサービスに関して不適切であったことである。

以下、2 章では degradation scheme について説明する。3 章では OS 間の degradation scheme の違いを明らかにする。4 章では Solaris と Linux 間の degradation scheme の違いの要因について述べる。5 章では我々の研究の今後の展望について述べる。6 章では関連研究について述べ、7 章で本稿をまとめる。

2　Degradation Scheme

多くの商用ウェブサイトはウェブサーバ、アプリケーションサーバ、データベースサーバの 3 層で構成され

る。ウェブサーバはクライアントからのリクエストを受けつけ、計算を伴なわない静的なページのみを処理する。アプリケーションサーバはデータベースサーバにアクセスしながら、複雑な計算を伴なう動的なページの処理を行う。近年、ウェブアプリケーションの高度化に伴ない、アプリケーションサーバでの計算量は増加する傾向にある。

例えば、トラック運送の帰り便を有効に使えるようにする B to B システムである帰り荷 Web [9] では、アプリケーションサーバで荷物の送り手と帰り便のマッチングを行う。データベースへのクエリでもマッチングの条件をある程度しぼることができるが、このウェブアプリケーションでは帰り便の空き状況だけでなく、トラックの経路、荷物の引き取り時刻、到着時刻、料金など様々な条件を考慮しなければならない。そのため、効率よくマッチングを行うためにはアプリケーションサーバでかなりの計算量を必要とする。一方、このアプリケーションサーバは荷物のマッチングという重い処理を行うだけではなく、ログインしてきた顧客に応じて動的にカスタマイズしたページを提供するなどといった比較的軽い処理も行う。

このように、様々な種類のサービスを提供している場合、アプリケーションサーバが高負荷になった時の挙動には様々なものが考えられる。例えば、時間のかかる荷物のマッチング処理を優先させる場合、マッチング処理を要求した顧客の待ち時間は減らすことができるが、ログインしてきた他の顧客への応答時間が極端に長くなる可能性がある。一方、公平に各サービスの処理を行う場合、どの顧客の処理もある程度ずつ進めることができるが、重い処理へのリクエストがタイムアウトを起こしてしまう可能性がある。セッションを用いるウェブアプリケーションでは、重い処理に時間がかかるとセッションの途中でタイムアウトしやすくなってしまい [10]、セッションを最初からやり直さなければならなくなる場合も多い。

ユーザの満足度とアプリケーションサーバの性能をともに最大化するには、状況に応じて適切な degradation scheme を選択できるようにすることが重要である。degradation scheme とは、各サービスに要求される品質を満たすために、高負荷時にそれらの性能をどの程度ずつ低下させるかを表わす。それぞれのサービスはシステムリソースを共有しているので、高負荷時にリソースが競合するとそれぞれに割り当てられるリソース量が減り、それに応じて性能も低下する。つまり、degradation scheme は限りあるシステムリソースを各サービスにどのように配分し、各サービスの性能のバランスをどのように取るかを決定する。負荷が高まった時にどのサービスの性能をどのように低下させたいかは、サービスの内容や実行コンテキストによって異なる。

3 OS 間の相違

アプリケーションサーバの degradation scheme はミドルウェアで制御するべきである。OS レベルでの制御も考えられるが、OS も含めた実行環境の変化に対応できるようにするには特定の OS に依存しない方が望ましい。ミドルウェアの制御がないと、OS は不適切な degradation scheme を使ってしまう場合がある。我々の実験によると、今日広く使われている OS はミドルウェアの制御がないと異なる挙動を示した。いくつかの OS の挙動は 2 章で述べたシナリオでは不適切であった。OS 間の挙動の違いは高負荷時のウェブアプリケーションの挙動が OS に大きく依存することを意味する。

この章では OS 間の degradation scheme の違いを明らかにするために行った実験について述べる。

3.1 実験内容

我々は Tomcat アプリケーションサーバ [7] の上で動くいくつかのサービスを作成して、degradation scheme を調べる実験を行った。Tomcat は Java VM (JVM) 上で動作し、サーブレットをインストールすることで様々なサービスを提供することができる。我々は degradation scheme を調べやすくするために、ウェブサーバやデータベースサーバを利用しない単純化した構成を用いた。この実験では、重いサービス、軽いサービス、中程度の重さのサービスの 3 種類のサービスを用意した。重いサービスは荷物のマッチング処理に対応するように、XML ファイルから Document Object Model (DOM) ツリーをメモリ内に作り、ツリーの全ノードの探索を 100 回繰り返した。重いサービスは大量のメモリリソースと CPU リソースを利用し、GC を頻繁に引き起こす。他方、軽いサービスとしては、CPU リソースを少しだけ利用する 25 番目のフィボナッチ数の計算を行った。また、中程度の重さのサービスとしては、CPU リソースを比較的利用する 35 番目のフィボナッチ数の計算を行った。

我々は degradation scheme を測定するために、Tomcat に対してクライアントから様々なワークロードを生成させた。実験 1 では、軽いサービスへの並行リクエスト数は 30 に固定し、重いサービスへの並行リクエスト数を 0 から 40 まで変化させたワークロードを用い

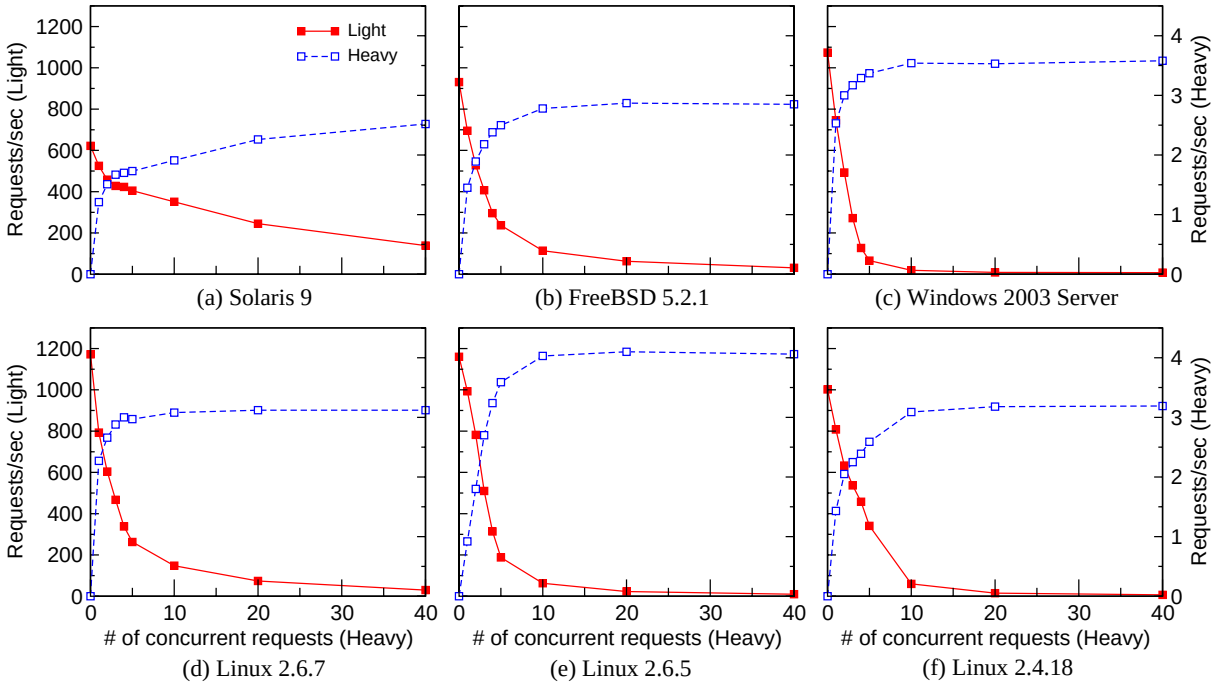


図 1: OS 間での degradation scheme の違い (実験 1)

た。実験 2 では、軽いサービスと中程度のサービスへの並行リクエスト数は 20 に固定し、重いサービスへの並行リクエスト数を 0 から 40 まで変化させたワークロードを用いた。クライアントは各サービスへの並行リクエスト数が指定した値になるまでリクエストを送り、Tomcat からレスポンスを受け取ったらすぐに新しいリクエストを送るようにした。

このようなワークロードを用いて、Tomcat の各サービスのスループットを Solaris 9、Linux 2.6.7、2.6.5、2.4.18、FreeBSD 5.2.1、Windows 2003 Server Enterprise Edition について測定した。サーバホストには Xeon 3.06GHz の CPU を 2 つ、2GB のメモリ、1Gbps の NIC を備えた Sun Fire V60x を用いた。使用した Tomcat のバージョンは 5.0.25、JDK のバージョンは 1.4.2 であった。クライアントホストには Pentium 733MHz の CPU、512MB のメモリ、100Mbps の NIC を備えた計算機を 8 台用いた。クライアントホストの OS は Linux 2.4.19 であった。これらのホストは 1Gbps のスイッチで接続されていた。

3.2 実験結果

実験 1 図 1 の (a)〜(d) は異なる OS について、重いサービスと軽いサービスからなるワークロードを用いて実験を行った場合の各サービスのスループットを示している。横軸は重いサービスの並行リクエスト数、縦軸は各サービスのスループットを表わす。どの OS でも似た傾向を示しており、サーバの負荷が高くなるにつれて軽いサービスのスループットは低下している。しかし、性能低下の度合いは異なる。図 2 はスループットをサービス別にプロットしたものである。このグラフから、OS の違いによって性能低下の度合いも異なることが分かる。Solaris の軽いサービスのスループットはゆっくりと低下しているのに対し、その他の OS では急激に低下した。重いサービスの並行リクエスト数を 0 から 40 まで増加させた時、軽いサービスのスループットは Solaris では 22%に低下しているが、その他の OS では 1%〜3%にまで低下している。さらに、Linux と FreeBSD の軽いサービスのスループットはほぼ同じ曲線を描いていることも分かる。逆に、重いサービス

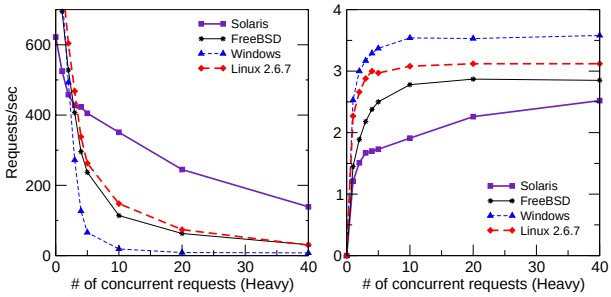
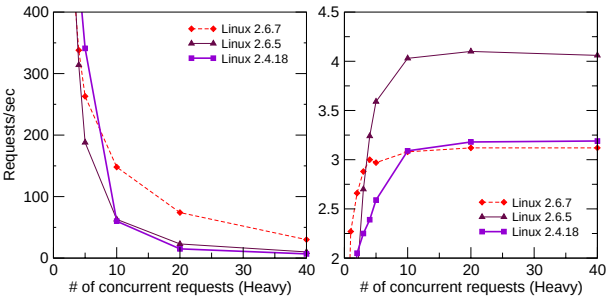


図 2: 異なる OS での各サービスの性能低下 (実験 1)



(a) Light-weight service (b) Heavy-weight service

図 3: 異なるバージョンでの各サービスの性能低下 (実験 1)

については Solaris の性能が一番低くなっていた。このことから、各サービスの性能はお互いに強く影響を与えあっていることが分かる。

同じ実験を Linux の異なるバージョンについて行った結果が図 1 の (d)〜(f) である。また、図 3 はスループットをサービス毎にプロットしたものである。このグラフから、バージョンが 2.6.5 と 2.6.7 というわずかな違いであっても大きく degradation scheme が異なることが分かる。高負荷時において、軽いサービスについては 2.6.5 と 2.4.18 が似た性能を示したが、重いサービスについては 2.6.7 と 2.4.18 が似た性能を示した。

実験 2　図 4 は 3 種類のサービスからなるワークロードを用いて実験を行った場合のスループットを各サービス毎にプロットしたグラフである。軽いサービスに関しては Solaris だけが比較的よい性能を保っているが、他の OS ではほとんど処理されていない。中程度のサービスでは各 OS 間の差はあまり大きくなかったが、重いサービスでは Solaris が最も悪い性能を示した。サービス数が増えると degradation scheme は複雑になるが、全てのサービスについてよい性能を示す OS はないことが分かる。

4　OS 間の相違の要因分析

3 章では OS 間の degradation scheme の相違を示した。degradation scheme を制御できるようにするには、この相違の主な要因が分からなければならない。この要因を探るために、我々は Solaris 9 と Linux 2.6.7 において軽いサービスを実行するスレッドの挙動を比較した。我々は 3 章の実験 1 と同じ実験環境を用い、軽いサービスに対する並行リクエスト数を 30、重いサービスに対しては 20 に固定した時の実験結果に対して分析を行った。

表 1: リクエスト毎のスレッド処理時間の内訳 (ms)

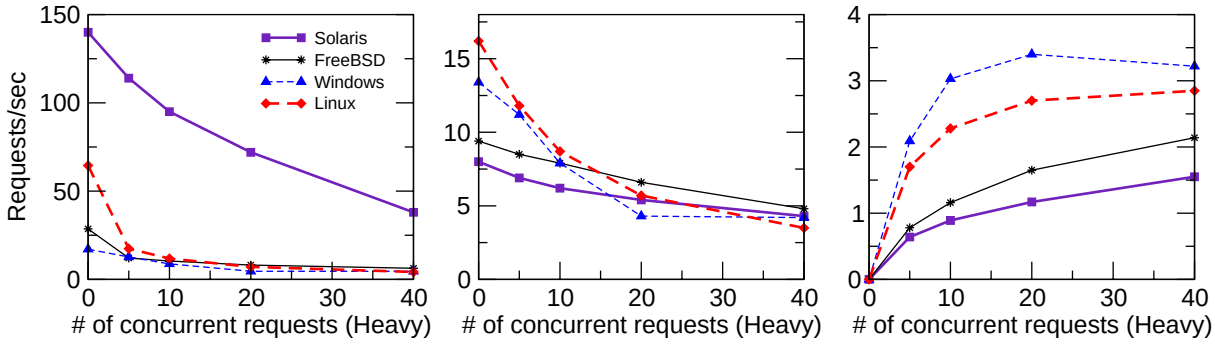
	Solaris	Linux
CPU 利用時間	3.71	3.91
待ち時間	137	375
(accept)	1.19	2.44
(poll)	0.41	19.4
(ロック)	136	348

4.1　リクエスト毎のスレッド処理時間

軽いサービスへのリクエスト処理に要するスレッド処理時間の内訳を調べるために、Tomcat の全てのスレッドについて、CPU スケジューリングとシステムコールに関するイベントを記録した。Tomcat が使う Java スレッドは Solaris 9 と Linux では特定のカーネルスレッドに結びつけられているので、カーネルレベルで Java スレッドを区別することができる。これらのイベントを記録するために、Solaris では `prex(1)` コマンドを利用した。`prex` はカーネル内で発生したイベントを選択的に記録することができる。Linux については *kev* と呼ぶ同様のツールを開発した。

これらのツールを用いて記録されたイベントログから、スレッドが 1 つのリクエストを処理する間に発生するイベントを取り出した。Tomcat はスレッドプールを使ってリクエスト処理のためのスレッドを再利用している。この実験では、合計で 51 スレッドが作られ、その内、50 スレッドは並行してリクエストを処理し、1 スレッドは次のリクエストを待っていた。そこで、各スレッドについてイベント列を `accept` システムコールの終了から次の `accept` システムコールの終了までの区間に分割し、1 つのリクエストを処理するのにスレッドが要した時間とした。

表 1 に Solaris と Linux における、軽いサービスへのリクエスト毎のスレッド処理時間の内訳を示す。CPU 利用時間は Solaris と Linux でほぼ同じであり、主にフィボナッチ計算に使われたと考えられる。一方、待ち時間の内訳を見るとロック待ちがスレッド処理時間のほとんどを占めていることが分かる。Solaris ではロック待ちのために `mutex_lock` システムコールと `cond_wait` システムコールが呼ばれており、Linux では `futex` システムコールが呼ばれている。Linux のロック待ち時間は Solaris の 2.6 倍長く、この違いが degradation scheme の相違の原因と考えられる。



(a) Light-weight service (b) Middle-weight service (c) Heavy-weight service

図 4: 3 種類のサービスでの各サービスの性能低下 (実験 2)

表 2: リクエスト処理中のロック待ち時間 (ms)

	Solaris	Linux
システムコールにかかる時間	64.6	65.2
システムコールの頻度	1.3	2.9
待ち時間の合計	82.0	189

表 3: スレッドプールでの待ち時間 (ms)

	Solaris	Linux
各スレッドの処理時間	4.34	11.7
待ちスレッドの平均数	12.7	13.0
待ち時間	52.6	154

4.2 ロック待ち時間の分析

我々はさらに、スレッド処理時間をリクエスト処理中とスレッドプール内とに分けてロック待ち時間の分析を行なった。

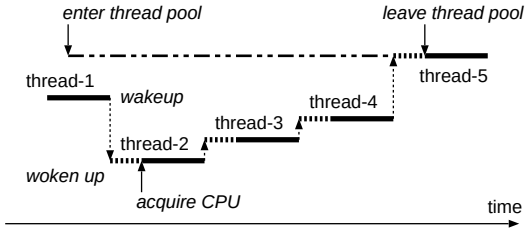


図 5: スレッドスケジューリング

4.2.1 リクエスト処理中の要因

我々はリクエスト処理中に発行されるロック待ちシステムコールにかかる時間を測定した。リクエスト処理時間はクライアントからの接続を受けつけてから、その接続を閉じてスレッドプールに入るまでの時間である。表 2 は Solaris と Linux におけるリクエスト処理中のロック待ち時間である。合計のロック待ち時間は Linux の方が Solaris より 2.3 倍長い。しかし、ロック待ちシステムコール 1 回あたりにかかる時間はほぼ同じであった。一方、システムコールが発行される回数は Linux が Solaris の 2.2 倍であり、この頻度の差がロック待ち時間の差となって表われている。

この 1 つの理由は、Solaris の JVM 1.4.2 はロック獲得時になるべくシステムコールを発行しないように実装されていることである。JVM はシステムコールを発行する `mutex_lock` 関数を呼ぶ前に `mutex_trylock` 関数を呼んでロック獲得を試みる。この関数によりユーザランドでロックが獲得できれば、ロック待ちシステムコールを発行する必要がない。2 つ目の理由は、Solaris のスレッドライブラリがスピンロックと `mutex_lock` システムコールの両方を使うアダプティブロックを実装していることである。スピンしている間にユーザランドでロックを獲得できれば、システムコールを発行する必要がない。3 つ目の理由は、Linux が Solaris とは違い `cond_wait` システムコールを提供していないことである。そのため、`pthread_cond_wait` 関数は 4 つのロック獲得操作を使って実装されており、システムコールを発行する頻度が増える。

4.2.2 スレッドプールでの要因

スレッドはリクエスト処理を終えた後スレッドプールに入り、スレッドプールから出た他のスレッドに起こされる順番が回ってくるまでロック待ちを行う。スレッドプールでの待ち時間は表 3 に示すように Linux の方が 2.9 倍長い。この待ち時間は図 5 に示すように、

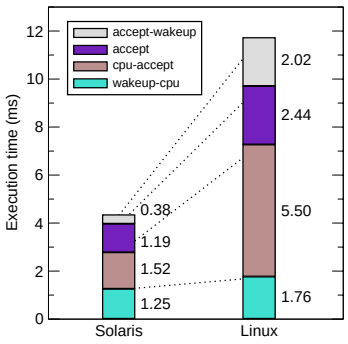
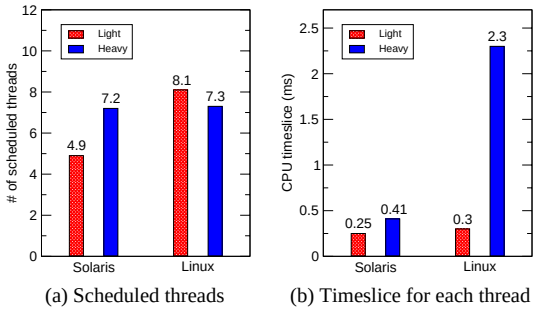


図 6: 各スレッドの実行にかかる時間の内訳



(a) Scheduled threads (b) Timeslice for each thread

図 7: 各スレッドの実行中に切り替わるスレッド数とそのタイムスライス

スレッドプール内の各スレッドが起こされてから、次のスレッドを起こすまでの時間の合計である。我々の実験によると、スレッドプールに入る時に待っているスレッド数の平均はほぼ同じであった。一方、各スレッドが次のスレッドを起こすまでの処理に要する時間は Linux が Solaris の 2.7 倍かかっていた。

スレッドプール内の各スレッドは他のスレッドによって起こされた後、以下のように処理を行う。まず、CPU の獲得を待つために CPU の実行キューの 1 つに入る (wakeup–cpu)。CPU スケジューラによって CPU が割り当てられたら実行を開始し、`accept` システムコールを発行する (cpu–accept)。`accept` システムコールの中で、スレッドはクライアントからの新しい接続を待つ (accept)。そして、クライアントからの接続処理を完了したら、スレッドプールで待っている次のスレッドを起こす (accept–wakeup)。

図 6 は各スレッドの処理にかかる時間の内訳を示したものである。このグラフから、スレッドプールでの待ち時間の最大の要因は、スレッドが CPU を獲得してから次のスレッドを起こすまでの accept 以外の処理にあることが分かる。そこで我々は、`accept` システムコールの発行中を除いて、スレッドが CPU を獲得してから次のスレッドを起こすまでの間の CPU スケジューリングを調べた。この区間でスケジュールされた延べスレッド数を図 7 (a) に示す。この区間に要する時間は Linux の方が 4.0 倍長いにも関わらず、重いサービスを実行する延べスレッド数はほぼ同じであった。この現象を説明する鍵は CPU スケジューラのタイムスライスの長さである。図 7 (b) に示されるように、重いサービスを実行するスレッドのタイムスライスは Linux の方が 5.6 倍長い。この長いタイムスライスがスレッドプールでの待ち時間の長い原因である。

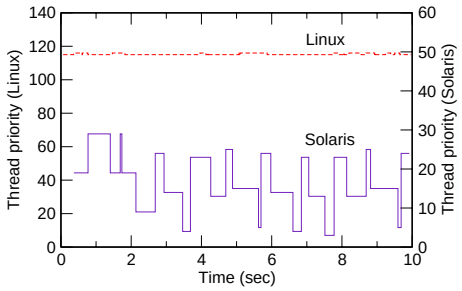


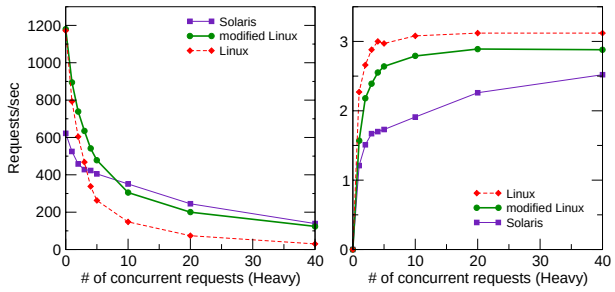
図 8: スレッド優先度の変化

Linux での長いタイムスライスの原因は CPU スケジューラのスレッド優先度の管理である。図 8 に示されるように、Solaris のスレッド優先度は頻繁に変動し、我々の重いサービスのような主に CPU を利用するスレッドは頻繁にプリエンプトされていた。一方、Linux のスレッド優先度はほぼ一定であり、CPU はほとんどプリエンプトされていなかった。そのため、Linux のタイムスライスは Solaris より長くなっていたと考えられる。

4.3 タイムスライスの変更による制御

我々の目的はミドルウェアで degradation scheme を制御することであるが、まず、タイムスライスを変更することで Linux の degradation scheme を変更することができるかどうかを確かめてみた。タイムスライスを短かくするために、Linux 2.6.7 のカーネルソースに修正を加え、最大タイムスライスを 200ms から 2ms に、最小タイムスライスを 10ms から 1ms に短くした。

この Linux を用いて、2 種類のサービスを用いる 3 章の実験 1 と同様の実験を行った結果を図 9 に示す。サーバが高負荷になった時の軽いサービスのスループットは高くなり、Solaris に近いスループットを示した。一方、重いサービスのスループットは低下してはいるものの、元の Linux に近かった。リクエスト処理にかかった時間の内訳を見ると、タイムスライスを短かくした Linux のリクエスト毎の待ち時間は 375ms から 161ms



(a) Light-weight service (b) Heavy-weight service

図 9: タイムスライスを短くした Linux の degradation scheme (実験 1)

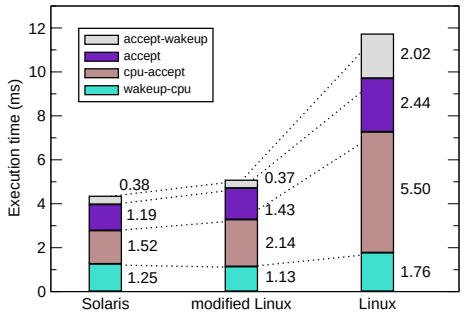


図 10: タイムスライスを短くした Linux で各スレッドの実行にかかる時間の内訳

に減っていた。その内、スレッドプールでの待ち時間は 154ms から 66.9ms に減っていた。スレッドプールでの各スレッドの待ち時間の内訳は図 10 のようになり、どの区間でも大幅に減っていた。一方、リクエスト処理中の待ち時間も 189ms から 53.0ms に減っていた。これは 1 つのシステムコールでの待ち時間が 65.2ms から 19.6ms に減ったためである。

次に、3 種類のサービスを用いる 3 章の実験 2 と同様の実験を行った結果を図 11 に示す。軽いサービスのスループットは元の Linux より高くなり、元の Linux と Solaris のほぼ中間の性能を示した。逆に、重いサービスについては元の Linux よりスループットが低下した。一方、中程度の重さのサービスについては、元の Linux や Solaris よりも高いスループットを示した。

これらの結果から、タイムスライスを変更することである程度 degradation scheme を変更でき、各サービスの性能のバランスを変更できることが分かった。

5 今後の展望

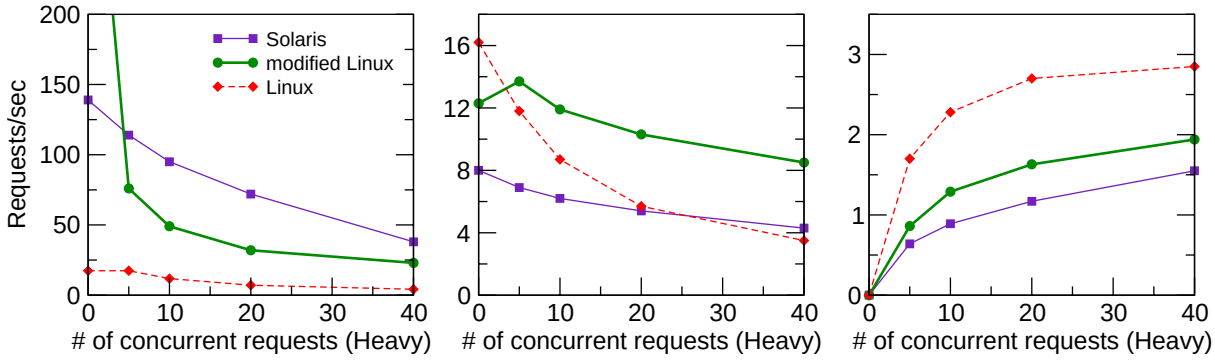
我々は今後、degradation scheme を制御するミドルウェアを開発する予定である。本稿で示したように、degradation scheme に関するアプリケーションサーバの挙動は OS に強く依存している。開発者は OS の挙動の違いを意識せずに最適な degradation scheme を指定でき、指定された degradation scheme は OS が何であっても自動的にアプリケーションサーバに適用されるようにすることが目標である。このようなミドルウェアは、例えば、開発時と運用時とで OS が変更された場合に役立つ。開発に時間がかかる場合、開発時と運用時とで OS のバージョンが変わることはよくある。新しいバージョンの方が性能がよくなっていることが多いので、運用時には最新のバージョンの OS を使えた方がよい。ミドルウェアで degradation scheme を制御できれば、OS のバージョンを変更する度に各サービスの性能のバランスをチューニングし直す必要がなくなる。

4.3 節の結果に基づき、我々はアプリケーションサーバやサーブレットにスケジューリングのためのコードを挿入することにより、degradation scheme の制御を行うことを考えている。例えば、重い処理を行うサーブレットに CPU を譲るコードを挿入することで、軽い処理を行うサーブレットが高負荷時でもより多く動けるようになる。サーブレットにコードを挿入することにより、外部から制御するのが難しい Java スレッドを制御できる。我々はこのようなコード挿入をアスペクト指向プログラミングを用いて汎用的に行えるようにする予定である。

6 関連研究

静的なウェブページからなるワークロードについては、高負荷時のウェブサーバの挙動がすでに報告されている。Almeida らは HTML ファイル、画像、音声、動画からなるワークロードについて高負荷時の Apache ウェブサーバの挙動を調べている [1]。彼らによると、リクエスト処理の 90%を費やすカーネル内 I/O 処理がボトルネックであった。Pradhan らは静的なウェブページへのリクエストについて、異なるワークロードでは異なるボトルネックがあることを指摘している [5]。パーシステント・コネクションが使われた時のボトルネックは accept キューであるが、SSL 暗号化が使われた時は CPU の実行キューであった。

McWherter らはデータベースにアクセスするサーブレットのボトルネックはデータベースによって異なると報告している。あるデータベースではボトルネック



(a) Light-weight service (b) Middle-weight service (c) Heavy-weight service

図 11: タイムスライスを短くした Linux の degradation scheme (実験 2)

はデータベースオブジェクトに対するロックであるが、別のデータベースではトランザクション処理のための I/O の同期であった [4]。さらに、サーブレットを変更した時に異なる結果が得られたと報告している。

高負荷時にサービス毎に性能低下の度合いを変えるために、優先度の基づくスケジューリング手法が多く提案されている。軽い処理の応答時間を短くするために、Crovella らは shortest-connection-first スケジューリングを提案している [2]。Elnikety らは優先度に基づくスケジューリングをウェブアプリケーションに適用している [3]。Neptune は各サービスについて利益を計算する関数を定義させ、その和が最大になるようにウェブアプリケーションのスケジューリングを行う [6]。しかし、これらの手法では OS に依存せず、自由に degradation scheme を制御するのは難しい。

7 まとめ

本稿では、高負荷時のウェブアプリケーションサーバの性能低下の挙動は OS に強く依存することを報告した。我々は Solaris、Linux、FreeBSD、Windows Server について、軽いサービスと重いサービスを提供する Tomcat のスループットを測定した。その結果、軽いサービスのスループットは Solaris ではゆっくり低下したのに対し、その他の OS では急激に低下することが分かった。さらに Solaris と Linux について実験データを分析した結果、この相違の主な要因はロック待ち時間であることが分かった。Linux での長いロック待ちの要因は、ロック待ちシステムコールが多く発行されていること、および、CPU スケジューラが重いサービスを実行するスレッドに高い優先度を与えていることであった。

参考文献

[1] Almeida, J., Almeida, V. and Yates, D.: Measuring the Behavior of a World-Wide Web Server, Technical Report 1996–025 (1996).

[2] Crovella, M., Frangioso, R. and Harchol-Balter, M.: Connection Scheduling in Web Servers, in *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (1999).

[3] Elnikety, S., Nahum, E., Tracey, J. and Zwaenepoel, W.: A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites, in *Proceedings of the 13th International World Wide Web Conference* (2004).

[4] McWherter, D., Schroeder, B., Ailamaki, A. and Harchol-Balter, M.: Priority Mechanisms for OLTP and Transactional Web Applications, in *Proceedings of the 20th International Conference on Data Engineering* (2004).

[5] Pradhan, P., Tewari, R., Sahu, S., Chandra, C. and Shenoy, P.: An Observation-based Approach Towards Self-managing Web Servers, in *Proceedings of the International Workshop on Quality of Service* (2002).

[6] Shen, K., Tang, H., Yang, T. and Chu, L.: Integrated Resource Management for Cluster-based Internet Services, in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (2002).

[7] The Apache Software Foundation, : Apache Jakarta Tomcat, `http://jakarta.apache.org/tomcat/`.

[8] Welsh, M., Culler, D. and Brewer, E.: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services, in *Proceedings of the 18th Symposium on Operating Systemsm Principles* (2001).

[9] デジタルサーブ株式会社：帰り荷 Web, `http://demo.nimo.jp/info/top.html`.

[10] 松沼正浩, 日比野秀章, 佐藤芳樹, 光来健一, 千葉滋：過負荷時の Web アプリケーションの性能劣化を改善する Session-level Queue Scheduling, 第 2 回ディペンダブルソフトウェアワークショップ (2005).