過負荷時の Web アプリケーションの性能劣化を改善する Page-level Queue Scheduling

松沼 正浩　　千葉 滋　　佐藤 芳樹　　光来 健一

{matsunuma, chiba, yoshiki, kourai}@csg.is.titech.ac.jp

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

本稿では、過負荷時における Web アプリケーションサーバの性能劣化を改善する手法として、リクエスト処理を最適にスケジューリングする Page-level Queue Scheduling を提案する。本手法は、動的に Web ページを生成するプログラムの実行をアプリケーションレベルでスケジューリングするものである。従来の Web アプリケーションサーバでは、過負荷時にクライアントやページ間でリソースの競合が起きてしまう。本手法ではリソース競合を検知するためにページ毎に用意されたスケジューラがそれぞれのページの生成処理の性能を測定する。個々のスケジューラはそのページの測定結果だけを参照し競合の判定を行い、競合が解消されるような最適な並列度に制限することで、性能劣化を改善している。制限された並列度を超えたリクエストは、ページ毎に用意されたキューに格納される。その際に、優先させたいリクエストをキューの先頭に格納し、そのクライアントに対する Web ページの表示順序を早めることも可能である。

1　はじめに

従来 Web アプリケーションサーバは、ページ生成処理を並列化することで余剰リソースを効率的に使用し、性能を向上させてきた。ここでいうページとは、静的なページ (HTML) と Servlet・JSP などのサーバサイドプログラムで動的に生成されるページを指しており、同一のプログラムによって生成されるものを全て同一のページとして扱っている。しかし、近年大量のリソースを消費して動的にページを生成する大規模な Web アプリケーションが増加しており、それらを並列に処理した場合、リソース競合が発生して性能低下を引き起こすことがある。さらに、並列化すべきもの・すべきで無いものが複数同時に存在することが、問題を複雑にしている。

そこで我々は、リソース競合が起きずに効率よくリクエストを並列処理できる数をページ毎に動的に算出することで、性能劣化を改善する Page-level Queue Scheduling を提案する。リソース競合の発生を判定するため、本手法ではページ毎に用意されたスケジューラがそれぞれのページ生成処理のスループットを測定する。個々のスケジューラは他のページの測定結果は見ずに、そのページの結果だけを参照して並列処理数を算出する。

本手法ではキューを使って並列処理数を制限するので、クライアントの優先度を設定することもできる。これによって、クライアント間の処理順序自体をスケジューリングし、全体性能を向上させることも可能である。また、我々は本手法を Servlet 用 Java API として実装し、その有効性を示すために既存の Servlet と比較する実験を行った。

以下では、2 章で従来の Web アプリケーションサーバのリクエスト処理の問題点を述べ、3 章で本手法の説明を述べ、4 章で実装を述べる。5 章で実験を行い、6 章で関連研究を紹介し、7 章で本稿をまとめる。

2　Web アプリケーションサーバのリクエスト処理

Web アプリケーションサーバは、処理性能を向上させるためにマルチスレッドやマルチプロセスを用いて、クライアントからのリクエストを並列に処理している。提供する Web ページのほとんどが静的なものや、ごく簡単な CGI スクリプトによるもの

だった頃、軽量なリクエスト処理の並列化はサーバの性能向上に直結していた。しかし、J2EE 技術などにより大量のリソースを消費する Web ページが増えた現在では、単純に多くのリクエストをできるだけ並列にさばいても、必ずしもサーバの処理性能が向上するとは限らない。

従来は、同時に処理するリクエスト数を増やすことが Web アプリケーションサーバの処理性能を向上させると考えられていた。軽量な Web ページでは、リクエストを並列に処理することで CPU やメモリなどの計算機リソースを効率的に利用し、サーバの処理性能を向上させる事ができる。例えば、Web アプリケーションサーバのリクエスト処理は典型的に I/O bound であるため、CPU における I/O 処理待ちのアイドル時間を別のスレッドを実行するために利用することで効率的に並列処理を行うことができる。図 1 は、リソース消費の少ないページの例として、リクエスト毎に動的にフィボナッチを計算して結果を表示するページに、最大 50 個のクライアントが同時にアクセスした時の、リクエストの総処理時間の推移を計測したものである[1]。並列に処理することで、逐次的に処理を行うものより大幅に処理性能が向上していることが図より読み取ることができる。

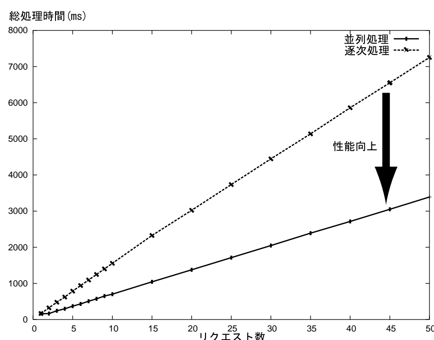

図 1: 並列化による性能向上

ところが、ページを表示するために大量のリソースを消費するような、近年広く普及している Web アプリケーションでは、その並列処理がリソースの競合を引き起こし、処理性能を低下させる場合がある。例えば、メモリ資源が競合すると、利用可能メモリを増やすために GC やスワップイン・スワップアウトが多発するため、全体の処理性能が低下してしまう。図 2 は、34KB 程度の XML データをパースしオブジェクトを大量に生成して探索を行うページに、最大 50 個のクライアントが同時にアクセスしたときのリクエストの総処理時間を計測したものである [1]。並列に処理した場合、逐次的に処理を行うものより性能が低下してしまっていることを図より読み取ることができる。

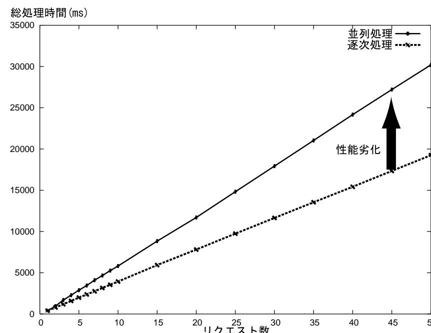

図 2: 並列化による性能劣化

このように、並列度を正しく決めないと、サーバ計算機のリソースを有効活用できないばかりか、リソース競合によって処理性能を低下させる場合もある。しかしながら、Tomcat [3] や JBoss [2] のような最大並列度を設定ファイルで静的に与えるような従来の Web アプリケーションサーバでは、並列度を正しく決められない。例えば、もし並列処理の効果が期待できる軽いページに併せて並列度を高くすれば、重いページが頻繁にアクセスされたときにリソース競合が全体の処理性能を落としかねない。逆に、並列処理がリソース競合を引き起こす重いページに併せて並列度を低くすると、リソースを有効に利用し処理性能を向上させられなくなってしまう。

3 Page-level Queue Scheduling

我々は、Web アプリケーションサーバの処理性能を上げるために最適な並列度を動的に設定する Page-level Queue Scheduling を提案する。本方式では、リソースの競合を起こさずに効率良くリクエストを並列処理できる数を動的に算出し、実行中の Web アプリーションサーバへ適用する。また、最適

[1]実験環境
サーバ CPU:UltraSPARC 750MHz × 2 メモリ:1024MB OS:Solaris8 NIC:100BaseTX
クライアント CPU:Pentium 733MHz メモリ:512MB OS:Linux2.4.19-Ovl11(VineLinux) NIC:100BaseTX 15 台

な並列度はページを表示するプログラム毎に異なるので、並列度はページ単位で決定される。

3.1 Page-level Queue Scheduling の概観

Page-level Queue Scheduling において、Web アプリケーションサーバに到達したリクエストは、まずページ毎に用意されたキューに格納される。ページ毎にキューを用意することで、各ページのリソース消費量を考慮したスケジューリングを行うことができる。リクエストは到達順にキューへ格納されるが、クライアントに優先権を設定することで、格納順序を変更することも可能である。それにより、特定のクライアントのリクエストを優先的に処理させ、過負荷時にも一定のレスポンス性能を維持できる。

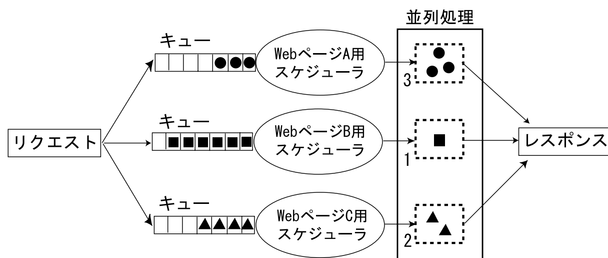

図 3: Page-level Queue Scheduling におけるリクエスト処理

それぞれのキューに格納されたリクエストは、ページ毎に用意したスケジューラにより設定された並列度で処理される。また、ページ毎の並列度は、サーバ計算機の有効リソースを監視しながら動的に調整される。

3.2 並列度の動的な決定

Page-level Queue Scheduling はサーバ全体ではなく、それぞれのページ生成処理のスループット（進捗状況）の推移に着目し、ページ毎に並列度を Progress-based regulation [1] に基づいて動的に決定する。Progress-based regulation とは、プロセスの進捗状況に応じて、リソース割り当てを動的に変更するスケジューリング手法である。ページ毎のスループットは、サーバ計算機の有効リソース量、すなわち稼動中のアプリケーション (Web アプリケーション以外のものを含む) 数やその消費リソース量に依存する。本手法ではページの生成処理の進捗状況を測定し、ページ生成の進捗が悪化した場合には、並列度が高すぎるため何らかのリソース競合が発生していると考える。そして、並列度を下げるようにスケジューラを調整する。同一のページでも消費リソースは毎回多少異なるため、スループットからページ生成のリソース消費量を完全に推定することはできない。しかし、大多数の Web アプリケーションは、平均的にページ毎に一定のリソースを消費すると仮定し、このような方法を採用した。

上記の仮定は長い時間では平均的に成り立つが、短い時間で見ると必ずしも成立しない。この影響を最小限にするために、単純なスループットの測定・比較だけでなく、過去のスループットの変遷履歴を考慮した統計処理を行う。

また、本手法ではより最適な並列度を求めるために、スケジューラが定期的に並列度を変化させて、スループットの変動を監視し、よりよい並列度を見つけようとする。ところが、スループットを測定する際に測定誤差が混入し、最適な並列度を正しく導き出せない場合がある。そのため、スケジューラは、最も高いスループットを出した時の並列度を記録し、その値を大幅に下回る状況に陥ったら、その並列度に戻す制御を行う。

3.3 Page-level Queue Scheduling による効果

Page-level Queue Scheduling を使用すると、Web アプリケーションサーバの性能に関して、以下にあげる 4 つの効果を得られると我々は考えている。

リソースの有効的な活用

並列に処理するとスループットが上昇するページを選んで、その並列度を積極的に上げる。したがって、サーバ計算機の余剰リソースを無駄なく有効に活用し、全体の処理性能を向上させられる。

競合が発生するページのスループット改善

並列に処理すると、リソースの競合を発生するようなページを検出して、その並列度を抑制する。そのようなページの並列度を競合が解消されるように抑えることで、従来ならスループッ

トを低下させていたページの処理性能を改善する。

ページ間の干渉を減少
スケジューラとキューをページ毎に用意することで、単位時間あたりに処理されるリクエストの数をページ毎に保障し、ページ間の干渉による性能低下を減少させることが出来る。例えば、処理の重いページへのリクエストが殺到したとしても、そのページの消費リソース量を制限することで、その他のページの処理速度を最低限保つことが出来る。

クライアントの優先度の設定
優先度に応じてリクエストのキューへの格納順を変更することが可能である。それにより、関連する複数のページ (セッション) において、既にセッション処理を開始したクライアントを優先的に処理させられる。例えば、オンラインショッピングサイトなどでは、既にカートに商品を入れているセッション処理中のクライアントは、優先的にそのサイトを巡回するように設定できる。

4 実装

我々は、以上に述べてきた Page-level Queue Scheduling を Java Servlet 用 の Java API (Application Programming Interface) として実装した。実装した API は、クライアントからのリクエストをインターセプトし、その実行を制御するための ControlServlet クラスを提供する。リクエストが Servlet サーバに到達すると、はじめに ControlServlet が呼ばれ、その内部でアプリケーション servlet が呼び出される。ControlServlet クラスには、スケジューラが実装されていて、各 servlet プログラムは、ControlServlet クラスを継承することで実行を制御される。

ControlServlet クラスは、それぞれのページで並列して動けるスレッド数を保持する変数 (maxThread) を持つ。実行中のスレッド数が maxThread に達した後のリクエストは、スケジューラが wait メソッドを用いて一旦停止させる。停止させたリクエストの処理順序を保つために、スケジューラはスレッドを停止する時に該当するスレッド名を到達順に保存する。

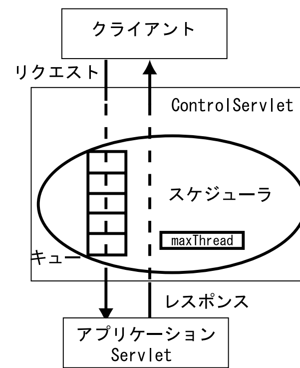

図 4: ControlServlet の概要図

優先処理するクライアントの識別には、Servlet のセッション機能を利用している。スケジューラは、優先すべきクライアントに対してユニークな Session オブジェクトを作成して ControlServlet クラスに保存する。保存した優先クライアントの情報を利用することで、複数のページの間で一貫した優先度を保つことができる。

また、ControlServlet クラスは、並列度の決定処理をカスタマイズするためのメソッドを提供している。アプリケーション servlet からそれらのメソッドを利用することで、手動で並列度の最大数や最小数、さらに並列度を変更するための条件を変えることができる。この機能によって、実測値から自動的に導き出される並列度に関わらず、重要度の高いページの並列度を上げたり、重要度の低いページの並列度を下げたりすることができる。

4.1 maxThread の決定

maxThread は、並列度を変更した時のスループットの変動を計測して決定している。次に具体的な maxThread の決定アルゴリズムを示す。基本的には、ある時点における maxThread の増加・減少の効果がスループットの計測により判断できる場合、その変更を行い続けるという戦略をとる。ここでは、ある時点 T で実行中の maxThread の値を Max(T) とし、それよりも一つ前に実行した時の maxThread の値を Max(T-1) とし、それぞれで計測したスループットの値 (処理リクエスト数/総処理時間) を Throughput(T)、Throughput(T-1) とする。今回計測したスループットとその時の maxThread を以下の条件式に当てはめることで次

回の maxThread が決定される。

- 条件 1： $\frac{\texttt{Throughput}(T)-\texttt{Throughput}(T-1)}{\texttt{Max}(T)-\texttt{Max}(T-1)} \geq 0$

  $$\Rightarrow \quad \texttt{Max}(T+1) = \texttt{Max}(T) + \Delta k$$

  Max(T)≥Max(T-1) かつ
  Throughput(T)≥Throughput(T-1)、あるいは
  Max(T)<Max(T-1) かつ
  Throughput(T)<Throughput(T-1)
  のとき上記条件を満たす。
  つまり、maxThread の大きい方のスループットが良いと判断されるので、設定された増減幅 Δk だけ増やす。

- 条件 2： $\frac{\texttt{Throughput}(T)-\texttt{Throughput}(T-1)}{\texttt{Max}(T)-\texttt{Max}(T-1)} < 0$

  $$\Rightarrow \quad \texttt{Max}(T+1) = \texttt{Max}(T) - \Delta k$$

  Max(T)≥Max(T-1) かつ
  Throughput(T)<Throughput(T-1)、あるいは
  Max(T)<Max(T-1) かつ
  Throughput(T)≥Throughput(T-1)
  のときに上記条件を満たす。
  つまり、maxThread の小さい方のスループットが良いと判断されるので、設定された増減幅 Δk だけ減らす。

また、ControlServlet クラスが提供しているメソッドを利用し条件式における左辺の分子部分 (`Throughput`(T) - `Throughput`(T-1)) を変更することが出来る。つまり、プログラマが maxThread の変更条件を変えることができる。例えば、(`Throughput`(T) - `Throughput`(T-1)∗k) のようにすることで (k>1)、maxThread の増加条件をスループットの単純な比較だけではなく、差分がある程度以上ある時に変更することができる。これにより、リソース競合が起きていなくても、リソースの消費を抑制したいページがあった場合、プログラマが増加条件を厳しくし、そのページの並列実行数を抑えることができる。

5 実験

3 章で述べた Page-level Queue Scheduling の効果を示すためにいくつかの予備実験を行った。はじめに、本手法におけるスケジューラがどのように maxThread を制御するかを示す。そのあとに、リソースを大量に消費する servlet プログラムを用いて生成される重いページと、リソースをあまり消費しないもので生成される軽いページの 2 種類を用意し、本手法を使用した場合と使用しない場合について性能を比較する実験を行った。ここでいう、重いページとは、34KB 程度の XML データをパースしてデータ検索を行い、結果を表示するページであり、1 リクエスト時における処理時間は 450ms 程度のものである。また、軽いページは、ほとんど計算を行わずに HTML を生成するだけのページであり、1 リクエスト時における処理時間は 70ms 程度のものである。

実験に使用した計算機は、サーバ計算機として CPU:UltraSPARC 750MHz × 2 メモリ:1024MB OS:Solaris8 NIC:100BaseTX、クライアント計算機として CPU:Pentium 733MHz メモリ:512MB OS:Linux2.4.19-0vl11(VineLinux) NIC:100BaseTX を 15 台を用いた。

5.1 本手法のスケジューラの動き

スケジューラが実際にどのように maxThread を変化させているかを示すために、本手法を用いた際の maxThread の遷移を示す実験を行った。実験に用いたページは 2 章で挙げた、並列化によって競合が発生し性能が劣化するページ (Xml パース) と 並列化によって性能が向上するページ (fib 計算) であり、それらの maxThread の遷移を図 5 に示す。

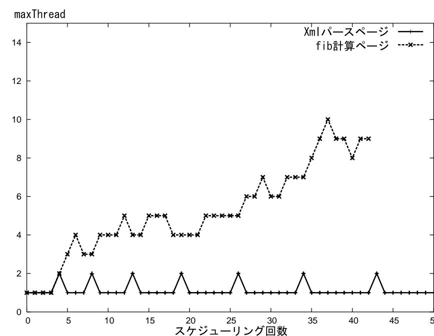

図 5: スケジューラによる maxThread の遷移

重いページに対しては、スケジューラが並列実行数を抑えることで、競合を防ぐように動作していることが分かる。一方、並列化することで性能の向上が見込まれるページに対しては、並列実行数を増加させて余剰リソースを効率的に使用することで、性

能向上を図っている。

5.2　競合が発生するページのスループット改善

重いページへ最大50個のクライアントが同時にアクセスした時の、リクエスト数に対するスループットの遷移について、本手法の使用時と未使用時のそれぞれで実験を行った。実験結果を図6に示す。

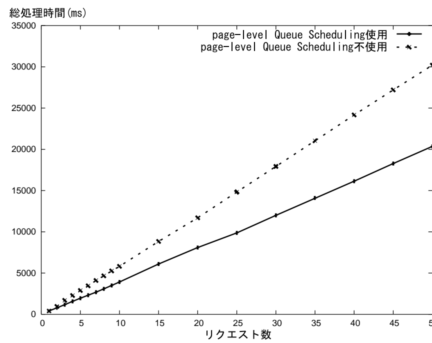

図6: Page-level Queue Scheduling 使用による競合発生時のスループット改善

実験結果から、本手法を用いることで制御を行わない時 (最大並列度200) に比べ、最大33%程度の性能向上が見られた。このページは、3並列程度でリソース競合が発生し、実験ではスケジューラによって最大並列度が1〜2程度に抑えられた。したがって、本手法によりリクエスト処理間で発生するリソース競合が十分解消されたと考えられる。

5.3　ページ間の干渉減少を実現

本手法が、ページ間の干渉による性能低下を減少させられる事を示すための実験を行った。実験は、常時軽いページに30クライアントがアクセスしている状況下で、重いページへ同時に80リクエスト与えた時の、軽いページからの時間当たりのレスポンス数の変化を測定した。本手法を用いない時 (軽いページの最適な並列度) と、静的に最大接続数を1(重いページの最適な並列度) に固定した時、また本手法使用時の場合における結果をそれぞれ図7、図8、図9に示す。

図9から、本手法により軽いページのレスポンス速度は重いページの処理に圧迫されることなく、ほ

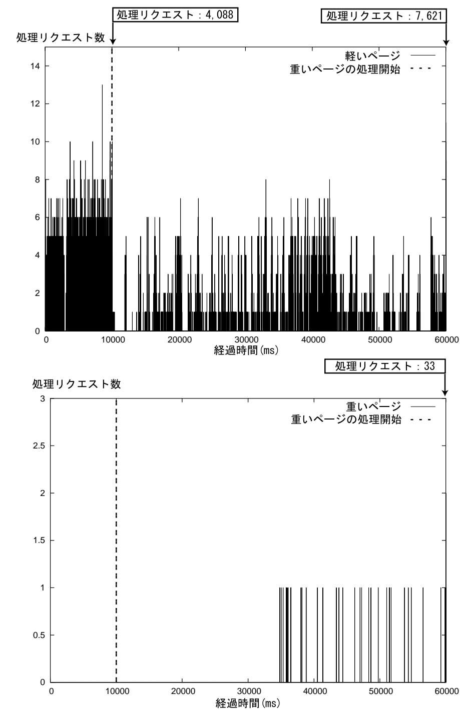

図7: Page-level Queue Scheduling 未使用時の、軽いページ (上段)・重いページ (下段) からの時間当たりのレスポンス数

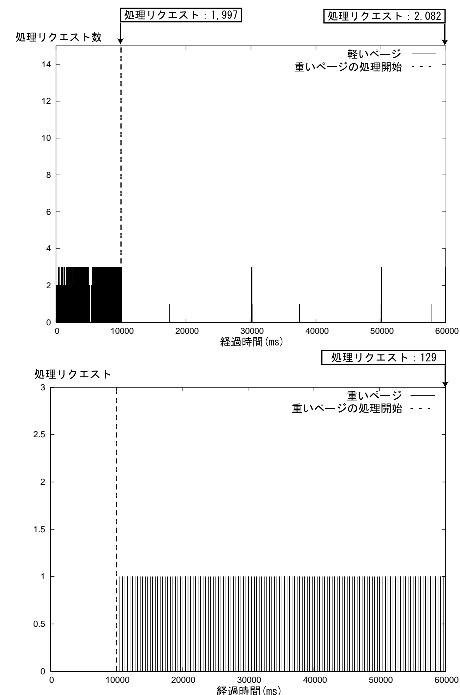

図8: 並列度を1に固定した場合の、軽いページ (上段)・重いページ (下段) からの時間当たりのレスポンス数

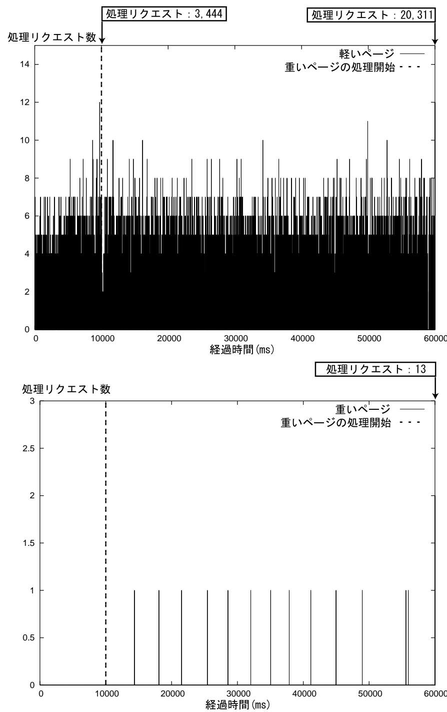

図 9: Page-level Queue Scheduling 使用時の、軽いページ (上段)・重いページ (下段) からの時間当たりのレスポンス数

ぼ一定に保たれていることが分かる。一方、本手法を用いない場合、図 7 より、重いページの処理がリソースを食いつぶしてしまい、軽いページのレスポンス速度を大幅に減少させていることが分かる。また図 8 から最大接続数を競合が発生しないような値に静的に決めてしまうと、並列処理を行った方が処理効率の向上するようなページの性能が低下してしまうことが分かる。

5.4 クライアントの優先度を実現

クライアントに応じて処理を優先させるスケジューリングの効果を実証するための実験を行った。実験は、重いページへ常時 50 クライアントがアクセスしている状況下で、優先権を持ったクライアントのレスポンス時間を計測した。実験結果を表 1 に示す。

表 1 から、クライアントに応じてリクエスト処理を優先的に行うことによって、過負荷時でも優先させたいクライアントのレスポンス時間をある程度保障することが可能であると言える。この実験での

	Page-level Queue Scheduling 無し	優先権なし	優先権あり
最低	41,260(ms)	21,820(ms)	1,390(ms)
最高	26,370(ms)	21,290(ms)	550(ms)
平均	37,500(ms)	21,550(ms)	850(ms)

表 1: レスポンス時間の比較

重いページの最大並列度は 1〜2 程度で変動しているため、大部分のリクエストは一旦キューに格納され、処理を停止させられる。しかし、優先権を持ったクライアントのリクエストはキューの先頭に格納されるので、他の処理を待たずに処理される。

6 関連研究

Web アプリケーションサーバにおける並列処理による性能低下を防ぐための研究として SEDA [4] がある。SEDA では、サーバにおけるリクエストの受付からレスポンスの返信までの一貫した処理を、ソケットの読み出しやキャッシュの操作などのように、ステージと呼ばれるモジュールに細かく分離して処理を行う。SEDA は、それぞれのステージでスループットを維持するべく最適なリソース管理を行う事で性能低下を防いでいる。細かいステージ毎にリソース管理を行うことで、処理低下のボトルネックとなる箇所の特定がし易く、その対処も比較的容易に行うことができる。しかし、SEDA はサーバ全体の性能劣化の改善を目標としており、本研究で目標としているようなページ単位での性能を保証することは出来ない。

7 まとめ

本稿では、過負荷時における Web アプリケーションの性能劣化を改善する新たな手法として Page-level Queue Scheduling を提案した。負荷分散のためにクラスターなどで各マシン毎に並列度を制限する方法は良く知られている。本研究では、資源の競合を回避して性能劣化を改善するために、同一のマシン上でページ毎に並列度を制限する方法を提案した。そして予備的な実験の範囲内においては本手法の効果のあることを示した。また本手法では、他のページの処理状況を測定せずに、そのページ単体の

測定結果だけから、資源の競合を判定している。実験の結果から、このような比較的実装しやすい方式でも一定の性能改善を達成できることを確認した。

今回は提案している手法の有効性を検証するために単純化されたワークロードで実験をおこない、その有効性を確かめた。今後の課題としては、より現実的なケースに対して本手法を適用し、その有効性を確かめる実験を行うことが挙げられる。また現在認識している問題点として、ページ毎のスケジューリングでは、それぞれのスケジューラが自身のページ性能を上げようとして、ページ間で余剰リソースを取り合ってしまう。そのような状況下で複数のページへリクエストが殺到すると、余剰リソースを使い果たしてしまい、本手法では性能劣化を防ぐことができない。今後は、そのような場合については他のスケジューラの挙動を考慮したスケジューリングをおこなうようにする必要があると考えている。

参考文献

[1] John R. Douceur and William J. Bolosky. Progress-based regulation of low-importance processes. In *Symposium on Operating Systems Principles*, pages 247–260, 1999.

[2] JBoss Group. Jboss. http://www.Jboss.org/.

[3] The Apache Jakarta Project. Tomcat. http://jakarta.apache.org/.

[4] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.