平成16年度 修士論文

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

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

学籍番号 03M-37280

松沼 正浩

指導教員

千葉 滋 助教授

平成17年2月7日

概要

本稿では、アクセスが集中して過負荷状態に陥った Web アプリケーションサーバの性能低下を防ぐための Session-level Queue Scheduling について説明する。商用サイトで用いられる Web アプリケーションサーバには常に高いセッション処理性能が要求される。しかし、現状の Web アプリケーションサーバでは、リクエストが殺到し過負荷状態になると、サーバ計算機上の計算リソースが競合しサーバ全体の処理性能が低下する。その結果として、セッション処理性能が著しく低下してしまう問題があった。そこで、我々はそのような過負荷状態においても高いセッション処理性能を維持するスケジューリング手法 Session-level Queue Scheduling を提案する。Session-level Queue Scheduling は実行中の Web アプリケーションの進捗を監視する事でリソース競合を検出し、競合解消のためにリクエスト処理に動的な制限をかけるアドミッションコントロールを行う。既存の多くのアドミッションコントロールはページまたはサーバ全体を単位として制御を行うが、セッションを利用する現実のワークロードには適していない。実際のワークロードを重視したスケジューリングを行うために、Session-level Queue Scheduling ではページ単位のスケジューラとセッション単位のスケジューラを組み合わせた制御を行う。我々は本手法の有効性を示すために、Session-level Queue Scheduling を実現したプロトタイプシステム ContorlServlet を Tomcat 上に実装し、いくつかの性能比較実験をおこなった。実験内容は、商用サイトで多用されている商品購入アプリケーションに見立てたプログラムをセッションを用いる形で実装し、ページ間の遷移時間や一定時間経過した場合にリクエストを破棄するといった実際のクライアントの挙動を考慮した自作ベンチマークソフトによるシミュレーション上での性能測定とした。なお性能比較対象は、ページやセッションを単位とした既存のアドミッションコントロールとした。その結果、リクエスト殺到時の過負荷状態においては、既存手法のどれもがセッション処理性能を低下させる中、ControlServlet を用いた場合のみが高い処理性能を維持することが確認され、本手法が一定の有効性を持つことを示した。

謝辞

本研究を支えていただいた多くの方々への感謝の意をここでは表したいと思います。東京工業大学 千葉滋助教授には、私が千葉研究室に所属した 3 年間の間、さまざまな面でご指導いただきました。本研究においては、研究の方針や進め方、論文の書き方や研究発表に関する指導など、貴重な意見を数多くいただいきました。深く感謝いたします。また、東京工業大学 光来健一助手には、本研究を進めるにあたり最後までお世話になりました。特に実験に関しまして様々なご指導いただきましたし、更には関連研究についても数多くの助言をいただきました。深く感謝いたします。また、東京工業大学 佐藤芳樹氏には、研究の基礎的なことから方向性、実装に関しての大部分に関与していただきました。さらに、論文の添削といった基本的なことまでをお手伝いいただき深く感謝いたします。また、貴重な助言をくださった東京工業大学 西沢無我氏、柳澤佳里氏、日比野秀章氏に、そして、本研究に関して貴重な意見をくださった千葉研究室の方々に心から感謝いたします。

目 次

第 1 章　はじめに　8

第 2 章　背景　10
2.1　セッション処理性能を向上させるための要件 10
2.2　アドミッションコントロール 11
2.2.1　サーバ単位のアドミッションコントロール 11
2.2.2　ページ単位アドミッションコントロール 13
2.2.3　セッション単位アドミッションコントロール 20
2.3　背景のまとめ 22

第 3 章　Session-level Queue Scheduling　24
3.1　Session-level Queue Schedulin の概要 24
3.2　ページスケジューラ 25
3.2.1　Progress-based Regulation [4] 26
3.3　セッションスケジューラ 27
3.4　プロトタイプ実装 28
3.4.1　maxThread の決定 28
3.4.2　sessionInterval の決定 29
3.4.3　ControlServlet 提供機能 30
3.4.4　ControlServlet の利用方法 31
3.5　提案のまとめ 37

第 4 章　性能評価実験　38
4.1　ページスケジューラによる性能改善 39
4.2　セッションスケジューラによる性能改善 40
4.2.1　レスポンスタイム 42
4.2.2　セッション処理性能差の原因 43
4.2.3　スケジューラの挙動 44
4.2.4　クライアントの挙動による影響 46
4.2.5　タイムアウト時間の影響 46
4.2.6　シンクタイム時間の影響 48
4.3　サーバー処理性能によるセッション処理性能に与える影響　50

4.3.1 低性能サーバ 50
4.3.2 高性能サーバ 52
4.4 実験のまとめ 53

第 5 章 まとめと議論 **54**
5.1 まとめ 54
5.2 議論 55

図 目 次

2.1 並列化による性能向上 (傾斜が少ないほど高性能) 14
2.2 並列化による性能劣化 (傾斜が少ないほど高性能) 15
2.3 レスポンス時間に対するサービスの質 16
2.4 Application-aware Admission Control スケジューラ概要図 17
2.5 ゲートキーパ概要図 18
2.6 Page-level Queue scheduling スケジューラ概要図 19
2.7 ページ処理性能 (左図) とセッション処理性能 (右図) 20

3.1 Session-level Queue Scheduling の概要 25
3.2 ページスケジューラによるリクエスト処理の制御 26
3.3 セッションスケジューラによるリクエスト処理の制御 . . . 27
3.4 通常の Servlet 処理 (左図) と ControlServlet 条件下の Servlet 処理 (右図) 28

4.1 総処理時間 (タイムアウト無し、シンクタイム 3 秒) 39
4.2 セッション処理性能 (タイムアウト無し、シンクタイム 3 秒) 40
4.3 総処理時間 (タイムアウト 60 秒) 41
4.4 セッション処理性能 (タイムアウト 60 秒) 41
4.5 レスポンスタイム (125 クライアント時) 42
4.6 レスポンスタイム (400 クライアント時) 43
4.7 セッション失敗数 (タイムアウト 60 秒) 44
4.8 sessionInterval とページキューの変動 45
4.9 各ページの maxThread の変動 45
4.10 総処理時間 (タイムアウト 30 秒) 46
4.11 セッション処理性能 (タイムアウト 30 秒) 47
4.12 途中失敗数 (タイムアウト 30 秒) 47
4.13 総処理時間 (シンクタイム 1 秒) 48
4.14 セッション処理性能 (シンクタイム 1 秒) 49
4.15 途中失敗数 (シンクタイム 1 秒) 49
4.16 総処理時間 (タイムアウト 60 秒) 51
4.17 セッション処理性能 (タイムアウト 60 秒) 51

4.18 総処理時間 (タイムアウト 60 秒) 52
4.19 セッション処理性能 (タイムアウト 60 秒) 53

表 目 次

3.1 Session-level Queue Scheduling 使用に用いるメソッド . . . 32
3.2 Session-level Queue Scheduling 使用時の補助メソッド . . . 34
3.3 ページスケジューラ使用時の補助メソッド 35
3.4 ページ単位の静的な制御法使用に用いるメソッド 36
3.5 セッション単位の静的な制御法使用に用いるメソッド . . . 36

4.1 セッション内の各ページでの失敗数の内訳 (125 クライアント/400 クライアント) 44

第1章　はじめに

商用サイトでは複雑な処理を行うためにWebアプリケーションサーバが用いられるようになってきている。Webアプリケーションサーバでは処理の単位としてセッションという単位が用いられることが多い。セッションとは、ユーザを特定して、複数のページにまたがる処理を一貫して行えるようにするものである。セッションを使えば、例えば、個人認証、商品検索、商品決定 (カート利用)、個人情報入力、という処理をそれぞれのページに分けて行うことができる。これらの一連のページはセッションにより特定のユーザに関連づけられる。セッションは個人認証から個人情報入力までの全てのページを完了して初めて利益となるため、Webアプリケーションサーバの性能を向上させるには、ページ単位の処理性能だけでなくセッションを単位とした処理性能も向上させる必要がある。

しかしながら、従来のWebアプリケーションサーバでは大量のクライアントからのアクセスが集中するとセッション処理性能が大幅に低下してしまう。その原因の一つは、過負荷時には各ページの生成処理でリソース競合が発生する可能性があることである。静的なHTMLページのようにリソースをあまり消費しないページの生成処理は、並列化することで余剰リソースを効率よく使用し、処理性能を向上させることができる。しかし、Servlet [11] などにより実装されるような大量のリソースを消費するページ生成処理を単純に並列処理した場合、リソース競合が発生して処理性能の低下を引き起こす。このようなリソース競合を解消するためには、大量にリソースを消費するページの処理を制御する必要がある。

もう一つの原因は、各ページの処理性能が低下した結果、セッション処理が中断されてしまう可能性があることである。セッション処理はセッションを構成する複数のページを全て処理してはじめて成功となる。しかし、過負荷時に各ページの処理に時間がかかりすぎると、ブラウザやサーバのタイムアウトが発生したり、ユーザが待ちきれずに読み込みを中止したりして、セッションが途中で中断される場合がある。このようにセッションの途中のページで処理が失敗した場合、セッションの最初から処理をやり直さなければならない場合も多く、それまでに行ってきた処理が無駄になる。このような無駄な処理を減らすためには、セッションの中断を防ぐ必要がある。

そこで本研究は、セッション処理性能を向上させるために、ページ単位のスケジューラとセッション単位のスケジューラを組み合わせた Session-level Queue Scheduling を提案する。ページスケジューラは個々のページ毎に用意され、リソース競合を起こさずに効率よくページ処理を行えるようにアドミッションコントロールを行う。ページスケジューラは個々のページ生成処理のスループットだけを監視することでリソース競合を検出し、リクエストの最適な並列処理数を動的に決定する。一方、セッションスケジューラはセッションの種類毎に用意され、セッション処理の途中失敗を防ぐためのアドミッションコントロールを行う。セッションスケジューラはそれぞれのページスケジューラで待たされているリクエスト数を監視することにより、同時に処理されるセッション数を最適に保つ。

本研究では Session-level Queue Scheduling の有効性を示すために、これらのスケジューラを Tomcat 上に実装した。本研究の実装では、専用のクラスを継承するように既存の Servlet を書き換えるだけで、Session-level Queue Scheduling が適用されるようにすることができる。この実装を用いて既存の Servlet と性能を比較する実験を行い、セッションのタイムアウトやセッション中の思考時間などを考慮した場合でも、Session-level Queue Scheduling を用いることで過負荷時のセッション処理性能が改善することを確認した。

以下、2 章で従来手法とその問題点について述べ、3 章で本手法とそのプロトタイプ実装について述べる。4 章では本手法とその他の制御法を用いて行った実験について述べ、5 章で本稿をまとめる。

第2章　背景

この章では、過負荷時におけるセッション処理性能向上のための要件と、既存の Web アプリケーションサーバの処理性能向上に関する幾つかの手法について述べる。既存の性能向上手法を述べるにあたっては、手法の説明だけでなく、はじめに述べたセッション処理性能向上の要件と照らし合わせ、それらを考慮した際の問題点についても同時に述べることとする。

2.1　セッション処理性能を向上させるための要件

商用サイトで用いられる Web アプリケーションサーバには常に高いセッション処理性能が要求される。セッション処理性能は単位時間あたりに処理できるセッション数であり、セッションを処理して初めて利益が得られる商用サイトには特に重要なサーバ性能の指標である。しかしながら、過負荷時においても高いセッション処理性能を保つのは容易ではない。過負荷時に Web アプリケーションサーバが高いセッション処理性能を保てるようにするには、以下に示す要件を満たす必要があると考えられる。

セッション処理性能向上の要件

各ページ処理の高性能化

過負荷時にはリソースを大量に消費するページの生成処理でリソース競合が発生し、処理が滞る場合がある。セッション処理はセッションに含まれる一連のページの生成処理から成るため、過負荷時における個々のページ処理性能を向上させることが、セッション処理性能の向上につながる。

セッション処理の保護

セッション処理はセッションに含まれる一連のページを処理して初めて完了したことになる。セッション処理の成功率は各ページ処理の成功率の積となるため、セッションの長さに比例して失敗率も上がる [3, 1]。そのため、セッション処理性能を向上させるには、過負荷時でもセッション処理が途中で失敗しないようにする必要がある。

次に Web アプリケーションサーバの処理性能を向上させるための既存技術であるアドミッションコントロールを紹介し、その詳細を説明すると共に、セッション処理性能向上の要件を考慮した場合の問題点について述べる。

2.2　アドミッションコントロール

ここで、Web アプリケーションサーバの処理性能を向上させるために広く知られた手法であるアドミッションコントロールについて述べる。アドミッションコントロールとは、特定のリクエストを識別し、サーバの状態や設定に応じてリクエスト処理の制御を行う手法である。リクエスト処理の制御には、対象となるリクエスト自体を破棄する場合と、実行が許可されるまで待機状態とする場合、優先度を下げて実行する場合など様々な処理内容を設定することが可能である。既存のアドミッションコントロールには例えば、並列処理を許可されるリクエストの数をサーバにあらかじめ設定するアドミッションコントロールが知られている。ここでは、このような手法の中で、サーバ単位、ページ単位 [5, 15]、セッション単位 [3, 2] のアドミッションコントロールについての説明を行い、セッション処理性能向上に関する要件を満たす上での問題点を述べる。

2.2.1　サーバ単位のアドミッションコントロール

最も単純で基本的なアドミッションコントロールとして、Web アプリケーションサーバ自体の設定により、サーバ単位でアドミッションコントロールを行う手法がある。例えば、代表的な Web アプリケーションサーバとして Apache [6] や Tomcat [7] などでは、それぞれ設定ファイル (Apache では httpd.conf、Tomcat では Server.xml) に最大幾つまでのリクエストを同時処理するかを設定することが出来る。設定された同時処理リクエスト数を超える数のリクエストがサーバに到達した場合には、それらのリクエストはその場で破棄されるか、処理可能枠に空きができるまで待機状態となる。このように同時処理リクエスト数を設定することで、サーバの処理スレッド (プロセス) 数の上限値を設定することが可能である。そのため、並列処理実行時のコンテキストスイッチの多発によるオーバーヘッド増加によるサーバ処理性能の低下も比較的簡単に防ぐ事が可能となる。

Server.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Server>
    <ContextManager>
    <!-- ================ Connectors ================ -->
    <Http10Connector   port="8080"
                       secure="false"
                       maxThreads="100"  #←最大処理数
                       maxSpareThreads="50"
                       minSpareThreads="10" />
    </ContextManager>
</Server>
```

httpd.conf

```
#
# Based upon the NCSA server configuration files
# originally by Rob McCool.
#
<IfModule worker.c>
StartServers         2
MaxClients         150  #←最大処理数
MinSpareThreads     25
MaxSpareThreads     75
ThreadsPerChild     25
MaxRequestsPerChild  0
</IfModule>
```

しかし、このような設定ファイルを用いて起動時に処理上限数を設定する手法では、上限値を静的に決める必要があるため、ページ処理によるリソース競合を解消しページ処理性能を向上させることは難しい。なぜなら、サーバ計算機上の余剰リソース量は、その時に同時動作しているアプリケーション (サーバアプリケーション、サーバ以外のアプリケーションを含む) の数や、そのアプリケーションの消費リソース量に大きく影響され、静的に決定することができないためである。また、Web アプリケーションの消費リソース量は、その処理内容に応じて一定ではない。そのため、全ての Web アプリケーションに対してサーバ全体で一つの上限値を持ってリクエスト処理の実行を制限する手法では効果的なリソース使用を実現できず、処理性能が低下してしまう問題もある。例えば、消費リソース量の少ない軽い Web アプリケーションに対するリクエストは、高

い並列度で動作させたとしてもリソース競合を引き起こす可能性が低いため、並列度を高く保つべきであり、逆に消費リソース量の多い Web アプリケーションはリソース競合を引き起こしやすいため、並列度を低くすべきである。

以上に挙げた理由より、静的にサーバ処理リクエスト数の上限値を設定する方法では、ページ処理性能を向上させることは難しい。またこのような単純な手法を用いた場合、リクエストの実行はサーバに到達した順に許可されるため、セッション処理途中のクライアントからのリクエストであっても、途中のページでリクエストが破棄される、または待機状態とされてしまう。そのためセッション処理性能を向上させるための要件の一つである、セッションの保護についても実現することが出来ない。このようなことから、サーバ起動時の初期設定によるアドミッションコントロールを用いることで、セッション処理性能を向上させることは困難であるといえる。

2.2.2　ページ単位アドミッションコントロール

次に、ページ毎に最大処理リクエスト数を設定する、ページ単位アドミッションコントロールについて述べる。ページ毎に最大並列度を制限するアドミッションコントロールによって、ページ処理性能を向上させることができ、ページ処理の集合であるセッション処理に関しても性能向上を見込むことができる。ページ単位のアドミッションコントロールでは、ページ毎にリソース競合を発生させず、かつ余剰リソースの効率的な使用を実現させる最適な処理数を決め、それを越えるリクエスト処理を破棄したり待機させたりすることができる。

たとえば、計算リソースをそれほど必要としない軽いページは、典型的な I/O バウンドな処理であるため、CPU における I/O 処理待ちのアイドル時間を別スレッドの実行のために利用させるように高い並列度で処理することで 処理性能を向上させることができる。図 2.1 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 のクライアントが同時にアクセスした時の、リクエスト総処理時間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出来る。

[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 NIC:100BaseTX × 15 台

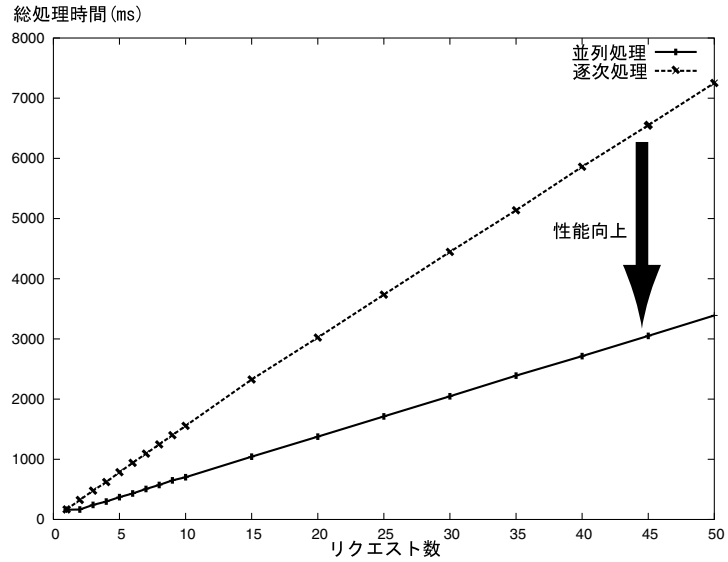

図 2.1: 並列化による性能向上 (傾斜が少ないほど高性能)

逆に、リソースを大量に消費する重いページでは、高い並列度は計算リソースの不足・競合を発生して処理性能が大きく低下することがある。図 2.2 は、リソースを大量に消費するページの例として、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、その中からキーワード探索を行い結果を表示する重いページに、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。このようなリソースを大量に消費する重いページの処理を並列に行った場合、逐次的に処理を行うものよりも性能が大きく低下してしまっていることが分かる。

このような問題点を解決するために、軽いページの処理並列度を上げ、重いページの処理並列度を下げるページ単位アドミッションコントロールを用いることで、リソース競合を回避しつつ計算リソースを効率よく利用することを可能にしサーバ全体の処理性能が向上する。並列度の決定方法には、アプリケーションプログラマによる予測 [2, 1, 8, 12] や、実行時に消費リソースを測定して動的に行うもの [5, 15, 9, 10] などがある。ここでそれぞれの手法について説明を行い、それぞれの問題点について述べ、最後にページ単位アドミッションコントロールでのセッションを考慮した場合の問題点を述べる。

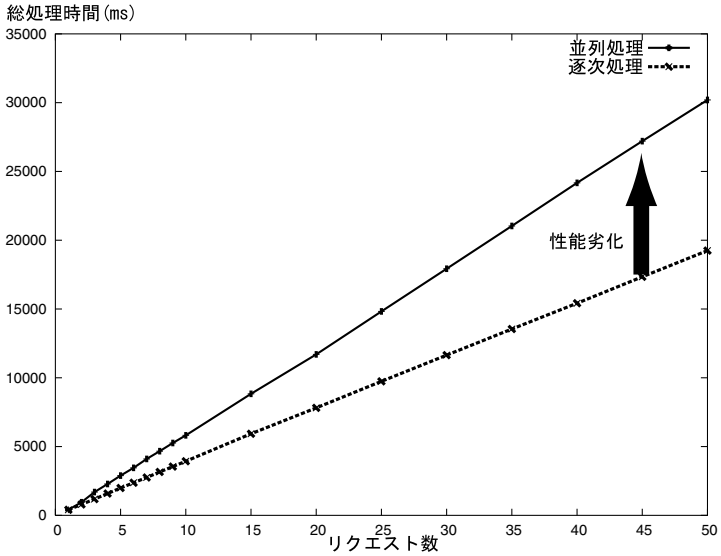

図 2.2: 並列化による性能劣化 (傾斜が少ないほど高性能)

A measurement-based admission-controlled web server [8]

計算リソースとして特にネットワークに注目し、その使用量をクライアント間で公平にすることを目的としたページ単位アドミッションコントロール。それぞれのアプリケーションのネットワーク消費量を予め測定しておき、ページ生成処理部分でクライアントのリクエスト処理がネットワーク消費量を公平にするように制御を行う。現在の Web アプリケーションサーバの多くでは、リクエストが到着した順に処理されてしまい、リクエストを多くなげたクライアントが優先的に処理されてしまうという不公平さを解消することを目的としており、サーバ処理性能を向上させるというよりもクライアント間の公平さを目標とした研究。処理性能自体を向上させるのでなく、またクライアントの公平さ自体はセッション保護とは逆の立場となるためセッション処理性能を向上させることは難しい。

Integrated Resource Management for Cluster-based Internet Services [10]

クラスタリングされたサーバシステムにおけるスケジューリング手法として提案しているアドミッションコントロール。ページ毎に関数を定義し、その関数から得られる数値の和を最大にするようにページ単位でリクエストのアドミッションコントロールを行う。ページ毎に定義する関数には、図 2.3 のようにレスポンスタイムと、そのレスポンスタイムに対して

得られるサービスの質を用いる。

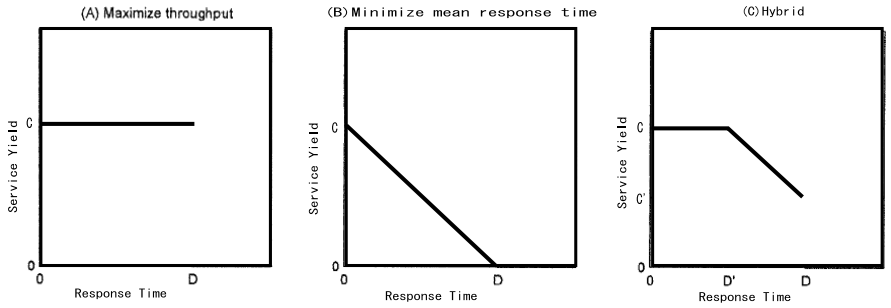

図 2.3: レスポンス時間に対するサービスの質

しかし、ページ毎の関数をアプリケーションプログラマが作成する必要があり、初期導入コストが高いという問題がある。また、セッションのような全てのページを処理して初めて意味が発生するようなアプリケーションの場合、同一のアプリケーションに含まれるページは全て同様の関数になりがちと考えられる。よってセッション処理の場合、この手法を用いても各ページ毎の制御が行われず、処理性能を向上することは難しい。

Application-aware Admission Control and Scheduling in Web-Servers [2]

Web アプリケーションサーバが提供している全アプリケーションをページを単位として機能 (消費するリソースに依存) ごとにステージと呼ばれるモジュールに分割し、そのモジュール毎にリクエストを分割して行うページ単位アドミッションコントロール。次にどのステージに対するリクエストを処理するかは、GPS(Generalized Processor Sharing) と呼ばれるコンピュータにおけるタイムシェアリングを一般化したものを利用している。具体的には、ステージ毎に重みを設定してやり、その重みに応じて処理時間を分配してやるという方法をとる。しかし、GPS による制御自体が特定のリソースの帯域をスケジューリングするための方式であり、複数のリソースを状況に応じて異なる使用量で用いる Web アプリケーションに対応させるためには、複数のリソースをリアルタイムに監視する必要がある。そのため監視するリソースの数に応じてオーバーヘッドが増大し、処理性能を低下させてしまう問題がある。

また、Application-aware Admission Control and Shceduling in Web-Servers では、セッションを単位としたアドミッションコントロールを始

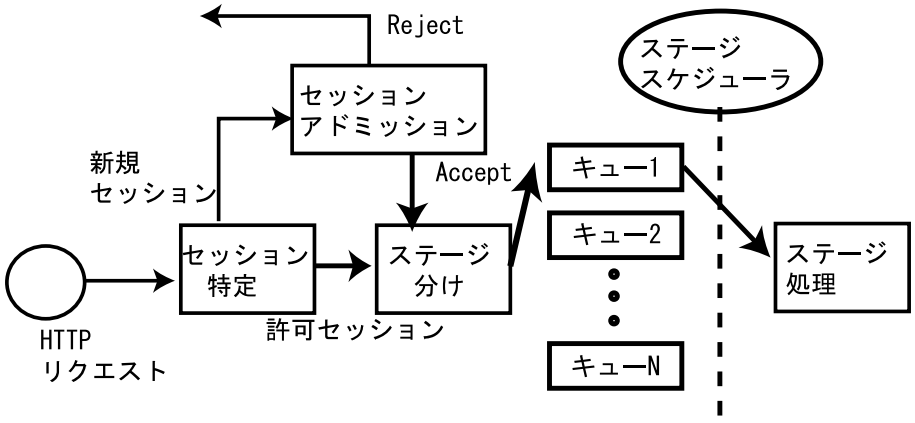

図 2.4: Application-aware Admission Control スケジューラ概要図

めにおこなっているが (図 2.4)、ページ単位アドミッションコントローラとの連携が行われておらず、静的にセッション処理の許可数を設定するだけである。そのため、実際のセッション処理を考慮した場合には対応することは困難であると考えられる。例えば、セッション単位アドミッションコントローラがセッション許可数を多く設定していた場合には、ステージ毎に用意されたキューで待機させられるリクエストの数が多くなり、セッションの保護が実現できず処理失敗を引き起こす可能性が高まる。逆にセッション許可数を少なく設定した場合には、リソースを効率的に使用させるだけの並列処理数を維持できなくなり処理性能を低下させる可能性が残る。また、この手法ではアプリケーションのステージ分割やステージ毎の重み計算といった処理は半自動化されているため、初期導入に必要なコストをある程度抑えることができる。しかし、Web アプリケーションサーバで提供するアプリケーションを増加またはサービス内容を変更した場合には、それらの計算を再度やり直す必要性が発生し、同時にサーバ自体を再起動させる必要があるなど問題が残っている。

A Method for Transparent AdmissionControl and Request Scheduling in EccomerceWebsite [5]

近年の商用サイトでは、Web サーバ、アプリケーションサーバ、データベースサーバを別々の計算機上で動作させる 3 点方式を採っている事が多いという前提にもとづいたページ単位アドミッションコントロール。3 つのサーバに関しては特にデータベースにおける処理が処理性能低下のボトルネックとなることが多く、そのデータアクセスのボトルネックを解消す

ることで処理性能を向上させる。ゲートキーパーと呼ばれるプロキシサーバをアプリケーションサーバとデータベースサーバの間に設置し、データベースアクセスを行う際の通信をトラップする (図 2.5)。通信をデータベースで行う処理内容に応じて分類し、対象となる通信の負荷を実行時に測定する。測定結果にしたがい、負荷が大きいと判断された通信には次回以降ペナルティをかし、軽い処理ほど実行されやすい状況を作りだす。

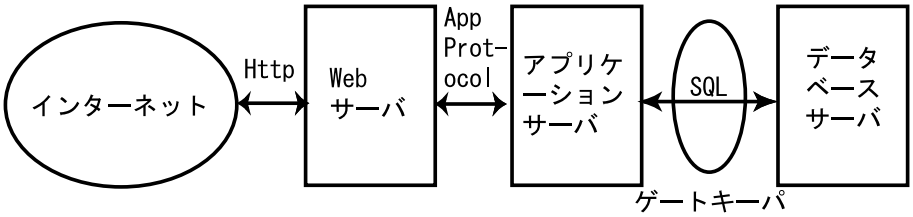

図 2.5: ゲートキーパ概要図

しかし、この手法ではデータベースに負荷を掛けないような軽いデータアクセスに関しては処理性能が向上するが、重い処理を行うデータベースアクセスは処理性能が逆に低下する問題が残る。通常、重い処理内容を伴うアプリケーションほどサーバに採って重要な処理を行うケースが多い事を考えた場合、そのような処理が実行されにくくなるという問題も軽視できない。また、現在提供されている実装ではデータベース使用という前提があるため、データベースを利用していないアプリケーションへの適用が出来ないという問題もある。

Page-level Queue Shceduling [15]

Web アプリケーションが消費するリソースの種類が非常に多く、全てのリソースを監視して、リソース競合の発生に備えるという方法ではオーバーヘッドが大きすぎるという前提を考慮したアドミッションコントロール。Web ページごとにリクエストの格納するためのキューを持つスケジューラを用意し、スケジューラは自身の管理するページ生成処理の実行時間 (進捗) を常時監視することで、リソース競合の発生をリアルタイムに検知する (図 3.4)。ページ生成処理の同時処理リクエスト数を動的に変更させながら進捗の変動を監視する。進捗状況が悪化した場合には、リソース競合が発生したと判定を行う。リソース競合が検知された場合、それぞれのスケジューラがページ処理の同時処理リクエスト数を抑えるように動作することでリソース競合の解消を行う。競合が発生したリソースの特定等は行わず、競合を発生させないということにのみ注目している。

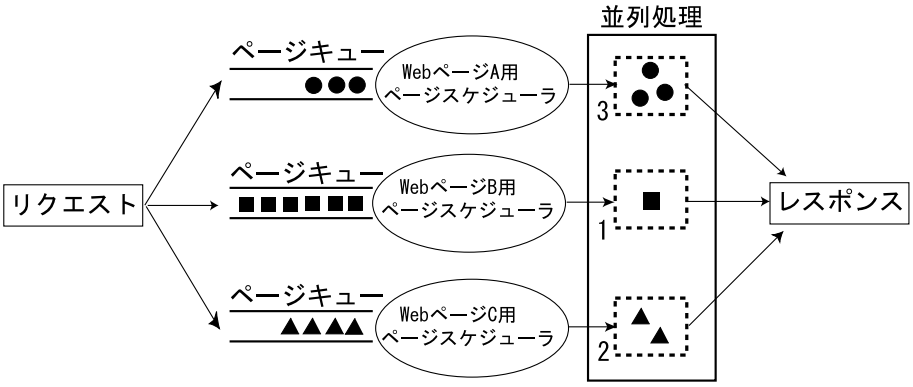

図 2.6: Page-level Queue scheduling スケジューラ概要図

しかし、特にリソースを大量に消費するようなページの同時処理リクエスト数は低く保たれるため、そのような Web ページに用意されたキューにリクエストが過度に溜まりすぎてしまう問題がある。セッションを考慮した場合には、後からキューに格納されたリクエストは長く実行を待たされるため、そのようなページで処理が途中失敗されやすくなり、セッション処理性能が低下してしまう。

ページ単位アドミッションコントロールのセッション処理を考慮した場合の問題点

以上に挙げたページ単位アドミッションコントロールでは、セッション処理性能向上の要件を考慮した場合にセッション保護が実現されていないため過負荷時にはセッション処理性能が低下してしまう。たとえば、並列度を低く設定された重いページに対するリクエストは、破棄されたり、すぐに処理されずに一旦待機状態となる。待機させられたリクエストは、場合によって Web サーバのタイムアウト [2]、ブラウザのタイムアウト [3]、さらにユーザが待ちきれずに読み込みを中止する事によって破棄されてしまう。セッション処理が中断されるとそれまでのページ処理が全て無駄になる。さらに、ユーザやブラウザによってリクエストが破棄された場合には、既存の Web アプリケーションサーバの多くはそれを検知できず、破棄されたリクエストの処理も行ってしまう。このような無駄な処理の分だけセッション処理性能が低下することになる。

[2] Apache 1.3 の初期値は 5 分
[3] Safari 1.2.3 以前の初期値は 60 秒
Mozilla Firefox では 120 秒
Mozilla Firebird では 120 秒

2.2.3　セッション単位アドミッションコントロール

続いて、同時処理セッション数を設定するセッション単位アドミッションコントロールについて説明する。同時に処理されるセッション数を制限するアドミッションコントロールは、セッション処理が途中で中断されないように保護することができる。例えば、セッション処理のボトルネックになっている最も重いページにあわせて、処理するセッションの数を制限すれば、セッションを構成するどのページでも並列処理による競合は起きなくなる。

このような方法では、リクエストはセッションの先頭ページで制御され、決められた並列度を越えるリクエストは先頭ページで破棄されるか待機状態となる。待機状態になった後、リクエストがタイムアウト等で破棄されたとしても、それによる損失はたかだか先頭ページでリクエストが破棄された後でも無駄に行なわれてしまう処理だけである。次に実際にセッションを考慮したアドミッションコントロールについて紹介し、セッション単位アドミッションコントロールで、実際にセッションを考慮した場合の問題点について述べる。

Web server support for tiered services [1]

Web サーバの提供するサービスの QoS(Quality of Service) を向上させるためにはネットワークのパフォーマンスを向上させる点に注目が集まりがちだが、Web サーバの処理性能自体も重要な要素であるとして Web サーバの処理性能を改善することで QoS の向上を目指している研究。

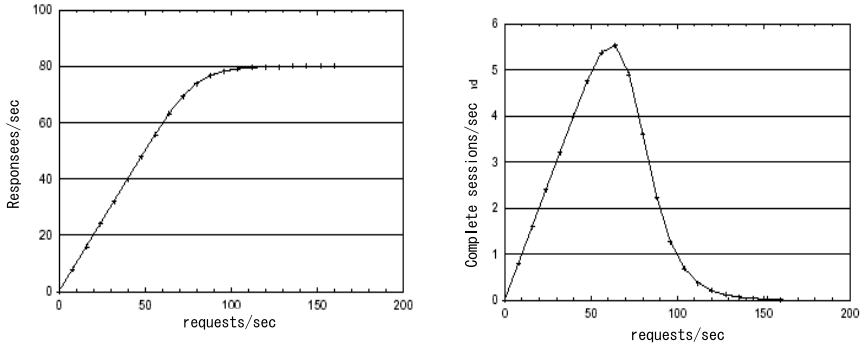

図 2.7: ページ処理性能 (左図) とセッション処理性能 (右図)

Web アプリケーションサーバの処理性能は低下しないようなケースでも、多くのクライアントが同時にアクセスして来ている状況下 (過負荷時)

では、セッション処理性能は大きく悪化するという点を解決する (図 2.7)。過負荷時にセッション処理性能が低下している主な原因は、サーバ自体の処理性能が低下せずとも途中で頭うちになるため、一定数以上のリクエストが投げられた場合にレスポンスが帰ってくるのが遅れる点にある。レスポンスが帰ってくるのが遅れだすと、タイムアウトやユーザーが待ちきれずにリクエストが途中破棄されてしまい、セッション処理が完了することなく失敗する。この問題点を解決するために特定のクライアントを優先的に処理されるプレミアムクライアントと設定し、そのクライアントの処理を優先的に行うことでレスポンスを向上させ、セッションの途中失敗を防止させる。

しかし、この手法では特定のクライアントを優先処理するため、それ以外のクライアントは若干処理性能が低下するという問題点を抱えている。また、プレミアムクライアントに設定されたクライアントがセッションを完了せずに、かつ適切なログアウト処理を行わずにセッションを中断した場合には、セッションタイムアウトまで新規プレミアムクライアントの設定ができないといった問題もある。たしかに過負荷時においても優先されたクライアントが処理されるため、セッション処理性能がゼロになることは避けられるが、サーバ全体の処理性能を向上させる事がなされていないため、全体的な処理性能向上はかなり限定されてしまうと考えられる。

session-based Admission Control [3]

アドミッションコントロールにセッションの概念を取り入れて、より実際のワークロードに対応させるように考えられたリクエスト処理モデルである。重要なセッション処理ほど長い傾向があるにも関わらず [4]、その長さに応じて途中失敗する可能性も高まってしまうことを問題点にあげている。特定のセッション処理に必要な計算リソース量を予め計算しておき、特定の計算リソースを常時監視し、そのリソースに 1 クライアント分のセッションを処理するのに必要な量の空きが出来た際にのみ、1 クライアントに処理の許可を与えるという仕組みである。この仕組みを用いることで、商用サイトなどでは重要な処理ほど長いセッションを利用している現実のワークロードにも対応させることができ、セッションの長さに関わらず処理を終わらせることができるようになる。

しかし、実際の Web アプリケーションでは複数の計算リソースを使用するため、特定のリソース監視だけでは不十分であり、逆に全ての計算リソースを監視するようにしてしまうと、オーバーヘッドが増大してしまうという問題点がある。また、処理性能の向上について測定を行っている

[4] 商用サイトでは、物の売買にからむセッションの長さが、その他の処理を行うセッションの長さの 2.5 倍程度と記述

実験では、シミュレーションモデルを用意しての実験に限定されているため、現実的なロードに基づいた実験とは言いがたい。

セッション単位アドミッションコントロールのセッション処理を考慮した場合の問題点

セッション単位アドミッションコントロールでは、現実のクライアントの挙動を考慮した場合にセッション処理性能が低下することがある。保護されたセッション処理が特定のクライアントによる計算リソースの占有を引き起こすため、サーバ全体の性能が悪化するという問題がある。セッション処理を保護するためには、保護されたクライアントが次のページにリクエストを出すまでの思考時間 (シンクタイム) でさえ他のクライアントのセッション処理を受け付けずにリソースを占有し続ける必要がある。もしシンクタイム中に他のクライアントの処理を開始してしまうと、保護されたクライアントを優先的に処理するのが難しくなる。

また、クライアントがセッションの終了手続き (ログアウト処理) を行なわずにセッションを放置することも多い。そのため、単純なセッション単位のアドミッションコントロールでは、リソースが長時間開放されないことになる。なぜなら、サーバ側ではセッションの終了をログアウトなしで判断するのは難しく、セッションのタイムアウト時間 [5]が経過するまで、セッション処理を保護し続けなければならないからである。

2.3　背景のまとめ

本章では、過負荷時におけるセッション処理性能向上のための要件と、既存の Web サーバ処理性能向上に関する幾つかの手法について述べてきた。セッション処理性能を向上させるためには、セッションに含まれる各ページの処理性能を向上させると共に、セッションを途中で失敗させないためにセッションを保護する必要がある。しかし、既存の Web アプリケーションサーバの処理性能を向上させる手法を用いてもその二つの要件を満たし、セッション処理性能を向上させることは難しい。例えば、リソース競合を解消しページの処理性能を向上させるページ単位アドミッションコントロールでは、セッション保護というセッション処理性能向上のための要件を満たすことができなかった。一方、セッション保護を実現しセッションの途中失敗を防ぐことを目的としたセッション単位のアドミッションコントロールでは、特定のクライアントが計算リソースの占有してしまい、効率的なリソース使用の妨げとなるような問題があった。以上の点か

[5]Tomcat の初期値は 30 分

ら、セッション処理性能を向上させるためには、そのような点を考慮した新しいアドミッションコントロールが必要であるということが分かる。

第3章　Session-level Queue Scheduling

過負荷時における Web アプリケーションサーバのセッション処理性能低下を改善するために、本研究ではページ単位とセッション単位のアドミッションコントロールを組み合わせた Session-level Queue Scheduling を提案する。本手法は、動的にページ処理を制御するページスケジューラと、ページスケジューラを監視しながらセッション処理を保護するセッションスケジューラによって実現される。ページ毎に用意したスケジューラが、各ページに最適な並列度を与える事で、計算リソースの競合を防ぎ Web アプリケーションサーバのページ処理性能を全体的に向上させる。一方、セッション毎に用意したスケジューラは、ページ用スケジューラが待機状態とするリクエスト数の調整をおこなう。これにより、リクエストがタイムアウトやクライアントによって破棄され、セッション処理を途中失敗させにくくできる。この二つのスケジューラを組み合わせ、二つを連携させて動作させることで過負荷時におけるセッション処理性能を向上・維持を実現する。

3.1　Session-level Queue Schedulin の概要

図 3.1 に示すように、Web アプリケーションサーバに到達したリクエストは、最初にセッションスケジューラによるアドミッションコントロールを受ける。セッションスケジューラが許可したリクエストは、続いて対象ページのページスケジューラによるアドミッションコントロールを受ける。セッションとページの二つのスケジューラの許可を得て、実際にページ生成処理が行われ、レスポンスをクライアントに返す。いずれの場合もリクエストの処理が許可されない場合は、そのリクエストは各スケジューラの保持するキューで待機状態となる。またセッションスケジューラによる制御は同一のセッションを通してはじめの一回に限るという特徴がページスケジューラによる制御との大きな違いである。セッションスケジューラによりセッション処理を一度許可されたクライアントからのリクエストは、同一セッションへのリクエストに限り、セッションスケジューラで待

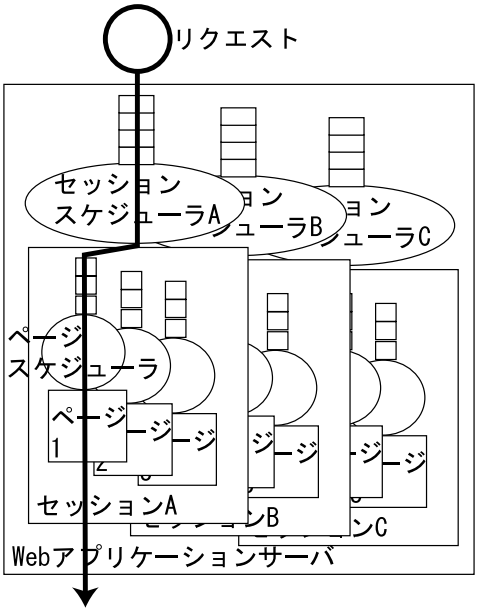

図 3.1: Session-level Queue Scheduling の概要

機状態となることはない。

3.2　ページスケジューラ

Session-level Queue Scheduling では、計算リソースの競合を解消し、ページ処理性能を向上させるために、ページ単位のアドミッションコントロールを行うためのページスケジューラを持つ。ページスケジューラは、計算リソースの競合を解消しページ処理性能を向上させる働きをする。ページスケジューラに到着したリクエストは、まずどのページに対するリクエストかの識別が行われ、つづいてページ毎に用意されたキューに格納される (図 3.4)。それぞれのキューに格納されたリクエストは、ページ毎に設定された並列度で処理が行なわれる。ページ毎の並列処理数は、ページ生成処理のスループットから 、Progress-based Regulation [4] の手法に基づいて動的に決定される。Progress-based Regulation の手法を利用することで、さまざまなリソースにおける競合の発生を検知することができ、特定のリソースを監視する必要もなくなる。ページ毎のスループットは、他のアプリケーションによって使用されていないサーバ計算機の有効リソース量に依存して変動する。ページスケジューラは、ページ生成処理のスループットを測定し、進捗が悪化した時には並列度が高すぎるために、何らかのリソース競合が発生していると判断し、並列度を下げる

調整を行う。逆に、進捗が悪化しなければ、さらに並列度を上げる調整を行う。

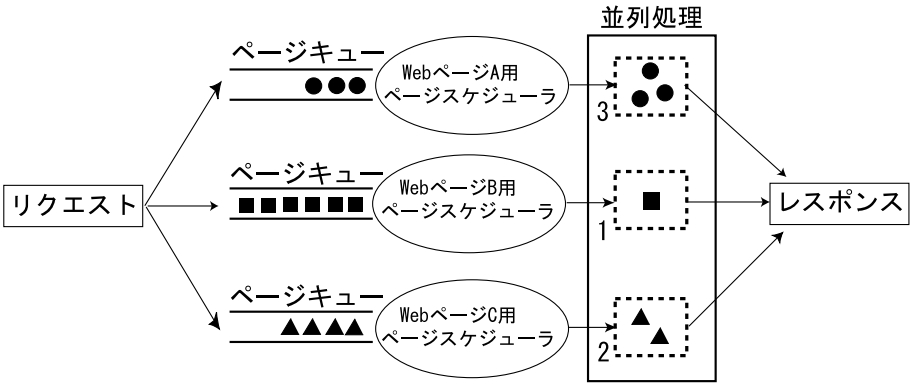

図 3.2: ページスケジューラによるリクエスト処理の制御

また、ページスケジューラは過去のスループットの変遷履歴を統計処理して並列度を決定する。Web アプリケーションのスループットは、リクエスト時に与えられるパラメータによる処理内容の違いやサーバ計算機上で動作する他のプロセスの影響によって変動することがある。この影響にページスケジューラが即座に反応してしまわないように、並列度の決定には単純なスループットの測定・比較ではなく過去数回に渡っての測定データを利用する。

3.2.1　Progress-based Regulation [4]

ここで、ページスケジューラのスケジューリングに応用した Progress-based Regulation についての説明をおこなう。progress-based Regulation とは OS のリソーススケジューリングの一手法であり、処理の進捗状況に応じてリソースの割り当てを変更する。重要性の低い処理の進捗を調べ、遅れていればリソースの競合が生じていると判断し、重要性の低い処理を一時停止させる。これにより重要性の低い処理とのリソースの競合を解消し、重要性の高い処理をすばやく行うことが可能である。2 つの処理の間でリソースの競合が生じる場合、互いの仕事の進み具合い、すなわち進捗が遅れてしまう。この進捗に注目することでリソースの競合が生じていることを探知することが可能である。特定のリソースの使用率に注目することはないので、すべてのリソース競合に対処することが可能である。

3.3　セッションスケジューラ

セッションスケジューラは同時セッション数を最大化する一方で、セッション処理の途中失敗を防ぐ。セッションスケジューラは、セッションに含まれる全てのページのキューを監視することで、セッションの並列度を決定する。例えば、ページスケジューラにおける待機リクエスト数が増加してページキューが長くなると、セッション途中において失敗がおきやすくなるので同時セッション数を減らす。逆にページキューが短い場合は、最適な並列度に達していない状態を意味するため、同時セッション数を増やす。

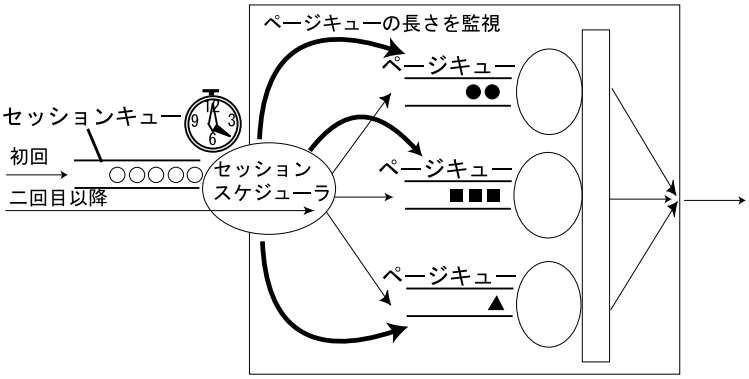

図 3.3: セッションスケジューラによるリクエスト処理の制御

セッションスケジューラは同時セッション数を変更するために、セッション処理の開始間隔を調節する。セッションスケジューラに到着したリクエストはまずセッションキューに入れられる。そして、セッション処理の許可に適度な遅延を持たせ、その遅延時間を調節することで、同時セッション数をコントロールする。遅延時間を大きくしていくと、同時セッション数を減少させられ、その結果、ページキューを短くできる。逆に遅延時間を小さくすると、同時セッション数を増加させられ、その結果、ページキューを長くできる。

また、セッションキューには受け付けられるリクエスト数の上限を設定できる。もし、セッションスケジューラがリクエストを無制限に受け付け、それを待機させると、実際の処理が開始される前にクライアントのタイムアウト等によりリクエストが破棄されてしまうかもしれない。すると、サーバがそれを検知できずに既に破棄されたリクエストがセッションキューに残ったままになってしまう。一定時間以内に処理が行われないと

判断できるリクエストを待機させずすぐに破棄[1]することで、リクエストの過度な蓄積を回避する。

3.4　プロトタイプ実装

Session-level Queue Scheduling の有効性を実証するために Tomcat [7] 上にこれらのスケジューラの実装を行った。ページスケジューラとセッションスケジューラが実装された ControlServlet クラスを提供しており、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 はクライアントからのリクエストをインターセプトしており、リクエストが Web アプリケーション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 はページの処理並列度 (maxThread) と、セッション処理の開始間隔 (sessionInterval) を保持し、その値は実行時に統計処理で決定される。

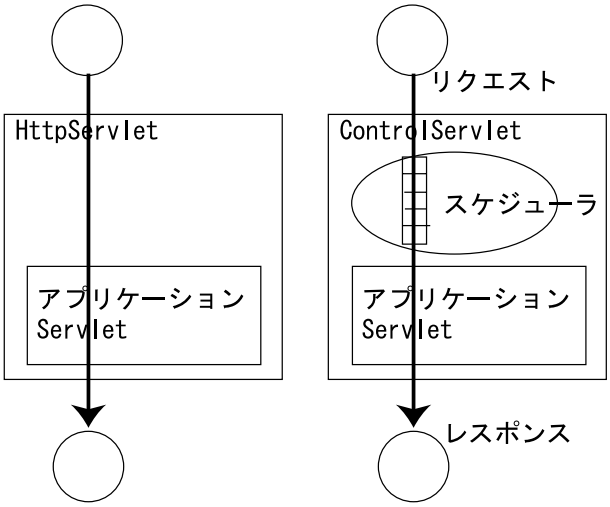

図 3.4: 通常の Servlet 処理 (左図) と ControlServlet 条件下の Servlet 処理 (右図)

3.4.1　maxThread の決定

maxThread は、ページ処理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される。ある時点において maxThread の値を増加・減

[1]エラーページを返信するなど

少させた時にその効果がスループットの計測により確認できる場合、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 MaxThread(T) とし、それよりも一つ前に実行した時の maxThread の値を MaxThread(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 が以下の関係を満たす時、

$$\frac{\texttt{Throughput}(T) - \texttt{Throughput}(T-1)}{\texttt{MaxThread}(T) - \texttt{MaxThread}(T-1)} \geq 0$$

MaxThread(T)≥MaxThread(T-1)・Throughput(T)≥Throughput(T-1)
あるいは
MaxTrhead(T)<MaxThread(T-1)・Throughput(T)<Throughput(T-1)
のとき上記条件を満たす。つまり maxThread が大きい方がスループットが良いと判断された時、

$$\texttt{MaxThread}(T+1) = \texttt{MaxThread}(T) + \Delta k$$

となる。一方、以下の関係を満たす時、

$$\frac{\texttt{Throughput}(T) - \texttt{Throughput}(T-1)}{\texttt{MaxThread}(T) - \texttt{MaxThread}(T-1)} < 0$$

MaxThread(T)≥MaxThread(T-1)・Throughput(T)<Throughput(T-1)
あるいは
MaxThread(T)<MaxThread(T-1)・Throughput(T)≥Throughput(T-1)
のときに上記条件を満たす。つまり maxThread の小さい方がスループットが良いと判断された時、

$$\texttt{MaxThread}(T+1) = \texttt{MaxThread}(T) - \Delta k$$

となる。
なお Δk は、アプリケーション Servlet から任意に設定が可能である。

3.4.2　sessionInterval の決定

sessionInterval は、ページ毎に用意したキューの長さを測定して決定している。次に具体的な sessionInterval の決定アルゴリズムを示す。ある時点におけるページキューの長さの平均が一定値 (maxQueue) 以上であ

る場合には、sessionInterval の値を増加させる。逆にページキューの長さの平均が一定値 (minQueue) 以下である場合には sessionInterval の値を減少させる。増分、maxQueue、minQueue の値はセッションの種類毎に任意に設定可能である。

$$QueueLength \geq maxQueue \Rightarrow sessionInterval + \Delta T$$

キューの長さの合計が長すぎると判断されるため、sessionInterval を設定された増減幅 ΔT だけ増やす。

$$QueueLength < minQueue \Rightarrow sessionInterval - \Delta T$$

キューの長さの合計が短すぎると判断されるため、sessionInterval を設定された増減幅 ΔT だけ減らす。
なお ΔT は、アプリケーション Servlet から任意に設定が可能である。

3.4.3　ControlServlet 提供機能

ここで、ControlServlet クラスの提供している機能について説明をおこなう。ControlServlet クラスは Session-level Queue Scheduling の機能を一括して使用するだけでなく、それぞれのスケジューラを個別に使用する仕組みを用意している。例えば、ページスケジューラを個別に利用することが可能であり、本手法で用いている Progress-based Regulation を利用したページ生成処理の進捗を利用したスケジューリングを基にしたアドミッションコントロールだけでなく、ページ毎にアプリケーションプログラマーが並列度を個別に静的設定することも可能である。また、ページスケジューラ以外にもセッション単位でのアドミッションコントロールについても同時セッション数を静的に決定して行うアドミッションコントロールなども同時に提供している。

使用可能機能

- Session-level Queue Scheduling:
  本手法によるアドミッションコントロール
- 動的ページアドミッションコントロール:
  Session-level Queue Scheduling のページスケジューラのみを利用したページ単位アドミッションコントロール
- 静的ページ単位アドミッションコントロール:
  静的に最大並列処理数を設定するページ単位アドミッションコントロール
- 静的セッション単位アドミッションコントロール:
  静的に最大クライアント数を設定するセッション単位アドミッションコントロール

以上の機能を Web アプリケーションサーバが提供しているアプリケーションに応じて使いわけることで、ページ単位やセッション単位で細かいリソース使用を実現することができる。例えば、特に重要な処理を行っており絶対に処理を失敗させたくないようなアプリケーションを提供している場合には、機能の使い分けによりその失敗する確率をさらに低くすることができる。例えば、アプリケーションが単一ページから成り立つ場合には静的な最大並列度を低く設定した静的ページ単位アドミッションコントロールを用い、複数のページからなるセッションの場合には、静的な最大クライアント数を低く設定した静的セッション単位アドミッションコントロールを用いる事で実現することが出来る。

3.4.4　ControlServlet の利用方法

次に ControlServlet によるスケジューラの実際の利用方法について説明を行う。まずはじめに、基本的な Servlet プログラムを示し、そのリクエスト実行をスケジューラによって制御させる手法について説明を行う。ベースとなる Web アプリケーションは以下のようなものとする。HttpServlet クラスを継承しており、Http リクエストの Get 命令に対応するための doGet() メソッドと、ポスト命令に対応するための doPost() メソッドを実装したものである。

```
1:  public class AppServlet extends HttpServlet{
2:      public void doGet(HttpServletRequest request,
3:                        HttpServletResponse response){
```

```
4:      //メインロジック開始
5:      ·····
6:      //
7:      }
8:      public void  doPost(HttpServletRequest request,
9:                          HttpServletResponse response){
10:     ·····
11:     }
12: }
```

Session-level Queue Scheduling による制御

はじめに本稿で提案している Session-level Queue Scheduling によるスケジューリングを利用するための方法について説明を行う。最も単純な利用方法は ControlServlet クラスを継承し、Servlet プログラムの init() メソッド内で、以下の 2 つのメソッドを呼び出す方法である。

表 3.1: Session-level Queue Scheduling 使用に用いるメソッド

戻り値	メソッド名	説明
void	setSessionName(String sessionName)	呼び出し元の Servlet クラスを sessionName というセッションとして登録する
void	setSessionLast(String sessionName)	呼び出し元の Servlet クラスを sessionName というセッションの最終ページとして登録する

セッションスケジューラに制御を行うセッションとして登録するために HttpServlet クラスの代わりに ControlServlet クラスを継承し、init() メソッド内で setSessioinName(String sessionName) メソッドを呼び出す。あるクライアントが setSessioinName() メソッドを呼び出しているアプリケーション Servlet が実行された場合、その時のクライアントのセッション情報をセッションスケジューラが記憶し、次回以降のアクセス時には同一名 (sessionName) で登録されているアプリケーション Servlet 実行時にはセッションスケジューラで制御されることなく実行が許可される。

```
1:  public class AppServlet extends ControlServlet{
2:      public void init(){
```

```
3:        setSessionName("Application1");
4:     }
5:     public void controlGet(HttpServletRequest request,
6:                            HttpServletResponse response){
7:        //メインロジック開始
8:         ·····
9:        //
10:    }
11:    public void  doPost(HttpServletRequest request,
12:                        HttpServletResponse response){
13:        ·····
14:    }
15: }
```

また、セッションスケジューラにセッションの最終ページとして登録するためには init() メソッド内で、setSessionLast(String sessionName) を呼び出す。なお、setSessionName() と setSessionLast() メソッドの引数の String は等しくする必要がある。setSessionLast() メソッドを呼び出しているアプリケーション Servlet が実行されると、それまで保存していたセッション情報が破棄され、セッションスケジューラを無条件通過できなくなる。

```
1:  public class AppServlet extends ControlServlet{
2:     public void init(){
3:        setSessionLast("Application1");
4:     }
5:     public void controlGet(HttpServletRequest request,
6:                            HttpServletResponse response){
7:        //メインロジック開始
8:         ·····
9:        //
10:    }
11:    public void  doPost(HttpServletRequest request,
12:                        HttpServletResponse response){
13:        ·····
14:    }
15: }
```

また Session-level Queue Scheduling の挙動をある程度制限するためのメソッドをいくつか用意しており、ここでその中から代表的なものを幾つ

か紹介する。

表 3.2: Session-level Queue Scheduling 使用時の補助メソッド

戻り値	メソッド名	説明
void	setSessionInterval(long time)	sessionInterval の長さを固定値 time に変更
void	setSessionIntervalMin(long time)	sessionInterval の長さの固定値 time を決定
void	setSessionIntervalMax(long time)	sessionInterval の長さの最大値 time を決定
void	setCheckInterval(long time)	sessioninterval を変更する時間間隔を固定値 time に変更
void	setSessionQueueLength(int queueLength)	セッションスケジューラのキューの長さを固定値 queueLength に変更

ページスケジューラ

次に Progress-based Regulation に基づいたページスケジューラを個別利用するための方法について述べる。ページスケジューラのみを個別に利用するためには以下のように ControlServlet クラスを継承しつつ、対象としたい Http リクエストに対応したメソッド (doGet() メソッドや doPost() メソッドなど) を control**() に置き換えればよい。現在提供しているメソッドは controlGet() メソッドと controlPost() の二つであるが、この種類を増やすことは簡単である。

```
1:  public class AppServlet extends ControlServlet{
2:     public void controlGet(HttpServletRequest request,
3:                              HttpServletResponse response){
4:       //メインロジック開始
5:        ·····
6:       //
7:     }
8:     public void  controlPost(HttpServletRequest request,
9:                                HttpServletResponse response){
10:       ·····
11:    }
12:  }
```

control**() と書き換えられたメソッドに対応する Http リクエストは、Progress-based Regulation を利用したページ単位スケジューラによりリソース競合を発生させないように制御される。また、doGet() のみを controlGet() にし、doPost() はそのままにするといったメソッド毎に制御を振り分けるようなことも可能である。以下に、ページスケジューラの挙動をある程度制限するためのメソッドをいくつか用意しており、ここでその中から代表的なものを幾つか紹介する。

表 3.3: ページスケジューラ使用時の補助メソッド

戻り値	メソッド名	説明
void	setMaxThread(int maxThread)	各ページの並列処理リクエスト数を固定値 maxThread に変更
void	setMaxThreadMin(int maxThread)	各ページの並列処理リクエスト数の最小値 maxThread を決定
void	setMaxThreadMax(int maxThread)	各ページの並列処理リクエスト数の最大値 maxThread を決定
void	setCheckThroughput(int checkNumber)	スループットの統計処理に用いるデータの試行回数 checkNumber を決定

ページ単位の静的な制御法の利用

次にページ単位で静的な maxThread 設定する方法について述べる。以下のように ControlServlet クラスを継承しつつ、対象としたい Http リクエストに対応したメソッド (doGet() メソッドや doPost() メソッドなど) を control**() に置き変え、init() メソッド内で以下のメソッドを呼び出せばよい。このようにすることで、setStaticControl() メソッド内の int 型引数の値を maxThread とした静的なページ単位アドミッションコントロールが行われる。

表 3.4: ページ単位の静的な制御法使用に用いるメソッド

戻り値	メソッド名	説明
void	setStaticControl(int maxThread)	呼び出し元の Servlet クラスの並列処理数 maxThread を設定する

```
1:  public class AppServlet extends ControlServlet{
2:    public void init(){
3:      setStaticControl(5);
4:    }
5:    public void controlGet(HttpServletRequest request,
6:                           HttpServletResponse response){
7:
8:    }
9: }
```

セッション単位の静的な制御法の利用

次にセッション単位で静的な並列処理クライアント数を設定する方法について述べる。以下のように ControlServlet クラスを継承しつつ、対象としたい Http リクエストに対応したメソッド (doGet() メソッドや doPost() メソッドなど) を control**() に置き変え、init() メソッド内で以下のメソッドを呼び出せばよい。このようにすることで、setSessionStaticControl() メソッド内の int 型引数の値を最大処理クライアント数とした静的なセッション単位アドミッションコントロールが行われる。

表 3.5: セッション単位の静的な制御法使用に用いるメソッド

戻り値	メソッド名	説明
void	setSessionStaticControl(int maxThread)	呼び出し元の Servlet クラスを sessionName というセッションとして登録する

```
1:  public class AppServlet extends ControlServlet{
2:     public void init(){
3:        setSessionStaticControl(5);
```

```
4:     }
5:     public void controlGet(HttpServletRequest request,
6:                            HttpServletResponse response){
7:     }
8:  }
```

3.5　提案のまとめ

本章では、過負荷時における Web アプリケーションサーバのセッション処理性能低下を改善するために Session-level Queue Scheduling を提案し、その詳細とプロトタイプ実装 ControlServlet について述べてきた。Session-level Queue Scheduling は、ページ単位アドミッションコントロールを行うページスケジューラと、セッション単位アドミッションコントロールを行うセッションスケジューラの両方を持つ。ページスケジューラは、各ページに最適な並列度を与える事で、計算リソースの競合を防ぎ Web アプリケーションサーバのページ処理性能を全体的に向上させ、セッションスケジューラは、ページ用スケジューラが待機状態とするリクエスト数の調整をおこなう。それぞれのスケジューラは個別に独立して動作するものではなく、それぞれが連携して動作することで、既存のアドミッションコントロールでは実現できなかったセッション処理性能の向上に関する二つの要件を実現した。

第4章　性能評価実験

Session-level Queue Scheduling による性能改善を示すためにいくつかの現実的なシナリオに基づいた Web アプリケーションを用い、幾つかの既存手法との性能比較実験を行った。実験に使用したアプリケーションは、商用サイトの商品購入アプリケーションに見立てたもので以下のような Servlet 群から構成される。

- 1 ページ目: ログイン (セッションの開始)
- 2〜4 ページ目: 商品の閲覧
- 5 ページ目: 商品検索・表示
  (約 650KB の XML ファイルをパースして探索)
- 6 ページ目: 商品を購入
- 7 ページ目: ログアウト (セッションの終了)

1 リクエストに対する処理時間は 5 ページ目のみ 400ms で、それ以外は 30ms 程度である。セッション処理性能を比較には、

(a) 制御無し

(b) ページ単位アドミッションコントロール [15]

(c) セッション単位アドミッションコントロール (セッション並列度:30)

(d) 重いページにあわせたセッション単位アドミッションコントロール
(セッション並列度:1)

(e) Session-level Queue Scheduling

の 5 つの既存手法を用いた。

実験には、Intel Xeon 2.40GHz × 2 の CPU、2GB のメモリ、ギガビットイーサネットを持つサーバ計算機を用い、OS には Linux 2.4.2、Web アプリケーションサーバには Tomcat 3.3.1 [7] を動作させた。また、Intel

Xeon 3.06GHz × 2 の CPU、2GB のメモリ、ギガビットイーサネットを持つクライアント計算機を用い、OS には Linux 2.6.8 を動作させた。

4.1　ページスケジューラによる性能改善

はじめに、ページ処理並列度の増加が引き起こすリソース競合とページスケジューラによるその競合解消を確認するための実験を行なった。実験は、同時にアクセスするクライアント数を 1 から 400 まで変えて同時にリクエストを送りはじめ、全クライアントのセッションが完了するまでの総処理時間を測定した。クライアントがセッション内のページを遷移する間に要するシンクタイムは 3 秒とした。また、ページスケジューラの効果だけを確認するために、クライアントは無制限にレスポンスを待つものとした。それによりセッションは失敗せずに必ず保護される。

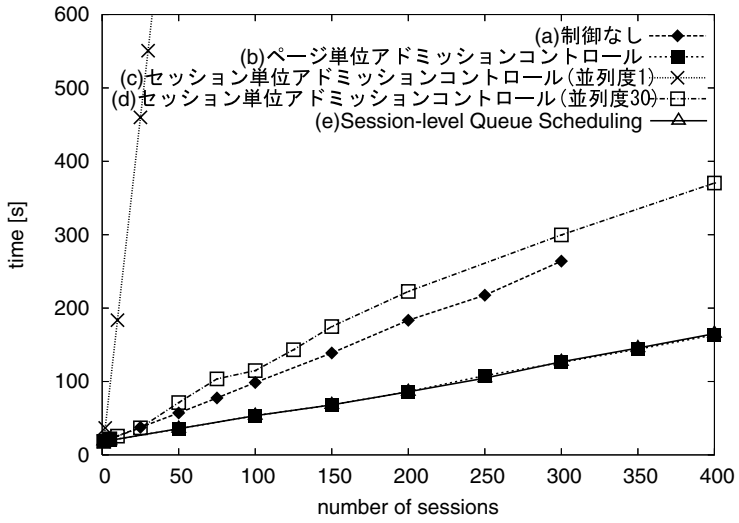

図 4.1: 総処理時間 (タイムアウト無し、シンクタイム 3 秒)

図 4.2 に総処理時間から算出した、同時セッション数に対するセッション処理性能を示す [1]。(a) の制御無しと (d) のセッション単位アドミッションコントロール (並列度 30) を用いた場合、リソース競合が発生してセッション処理性能が低下した。一方、(b) のページ単位アドミッションコントロールと (e) の Session-level Queue Scheduling は、各ページの並列度を制御してリソース競合を解消するため、最もセッション処理性能が高

[1]途中でデータが途切れているものは、それ以上のセッション数では 500 秒以上にわたりレスポンスが得られなかったことを示している。以下の実験でも同様。

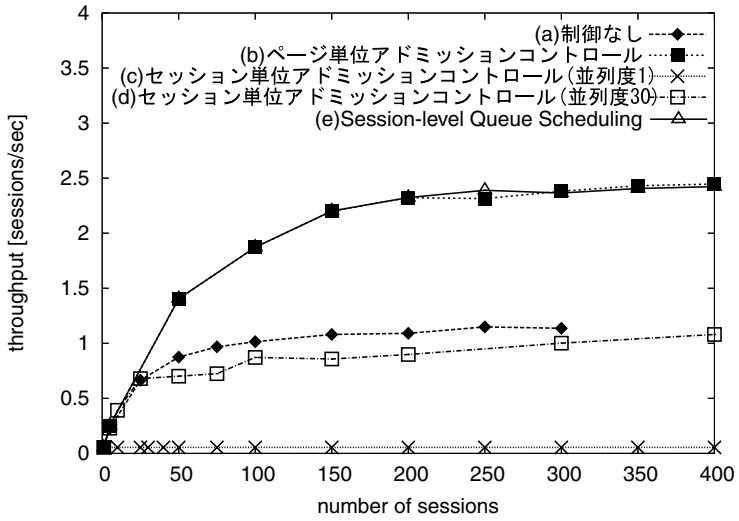

図 4.2: セッション処理性能 (タイムアウト無し、シンクタイム 3 秒)

くなった。セッションが失敗しない場合、その二つの手法は同じ挙動を示すため同程度の性能を得た。この二つからセッションスケジューラのオーバーヘッドは十分小さいことが分かる。なお、(c) のセッション単位アドミッションコントロール (並列度 1) で著しく処理性能が低下したのは、シンクタイムを含んだセッション処理が逐次的に行なわれたためである。

4.2　セッションスケジューラによる性能改善

セッションスケジューラによる性能改善を示すために、4.1 節と同様の実験をブラウザのタイムアウトを発生させて行なった。タイムアウトの時間は 60 秒とし、タイムアウトしたクライアントはそのページへのリクエストを破棄し、セッションの先頭ページから再試行するものとした。

図 4.4 より、ブラウザのタイムアウトを考慮しても、(e) の Session-level Queue Scheduling だけが高いセッション処理性能を維持できた。(a) の制御無しの場合、約 50 クライアントで処理性能が頭打ちになる。また、(b) のページ単位アドミッションコントロールでは、同時クライアント数が 150 を越えたぐらいからセッション処理性能を維持できなくなった。一方、(c) のセッション単位アドミッションコントロール (並列度 1)、(d) のセッション単位アドミッションコントロール (並列度 30) 共に、セッションの途中失敗数がそれぞれの最大クライアント数に達した時点で、その失敗したセッションをセッションタイムアウトまで保護するため、他のリク

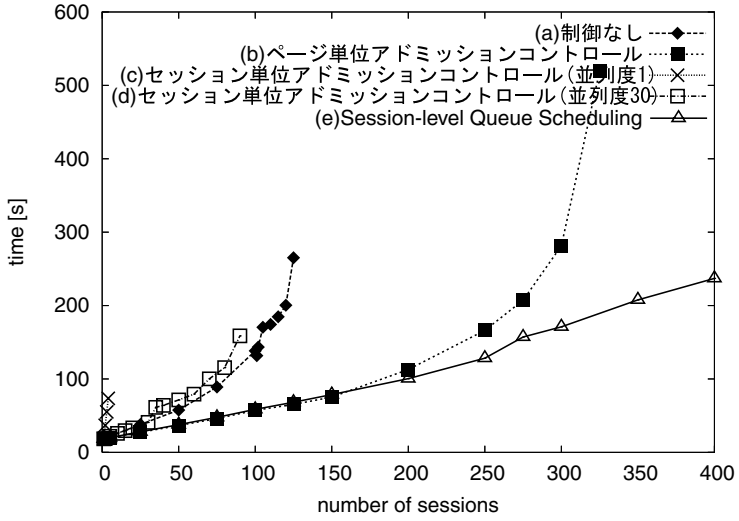

図 4.3: 総処理時間 (タイムアウト 60 秒)

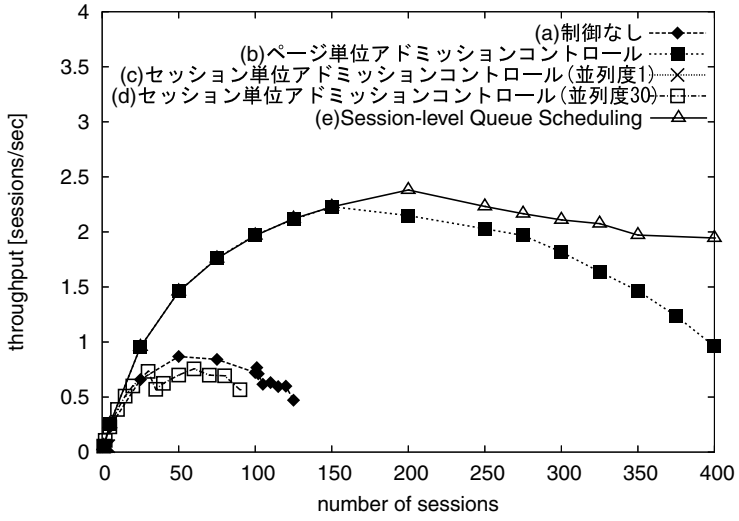

図 4.4: セッション処理性能 (タイムアウト 60 秒)

エストが全く処理されなくなる。

4.2.1　レスポンスタイム

セッション処理性能を測定するにあたり、スループットだけでなくレスポンスタイムの測定実験も行った。レスポンスタイムの測定はセッション数 125 の時と 400 の時で行い、各セッションのレスポンスタイムを 10 秒ごとに分け、そのレスポンスタイムに対応するセッション数を分類した。結果を図 4.5、図 4.6 に示す。

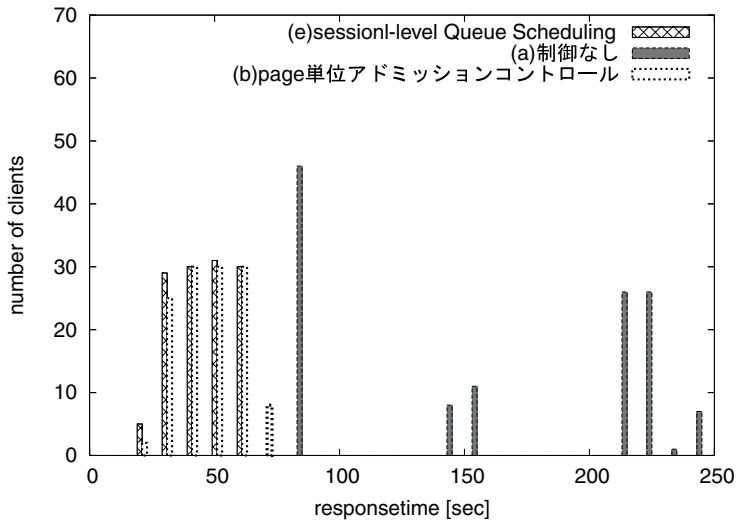

図 4.5: レスポンスタイム (125 クライアント時)

セッション数 125 の時には、(b) のページ単位アドミッションコントロール、(e) の Session-level Queue Scheduling を用いた場合にレスポンス性能が (a) の制御なしの時に比べて、大きく向上していることが分かる。これは、リソース競合が解消されてレスポンス性能自体が向上したことと、重いページの並列処理数が制限された結果、セッション処理が徐々に行われるようになったためだと考えられる。また、セッション数 400 の時には、(e) の Session-level Queue Scheduling が (b) のページ単位アドミッションコントロールと比較しても高いレスポンスタイム性能を出すことができた。これは、セッションスケジューラがセッションの途中失敗を防ぎ、無駄な処理を減少させたことに原因があると考えられる。

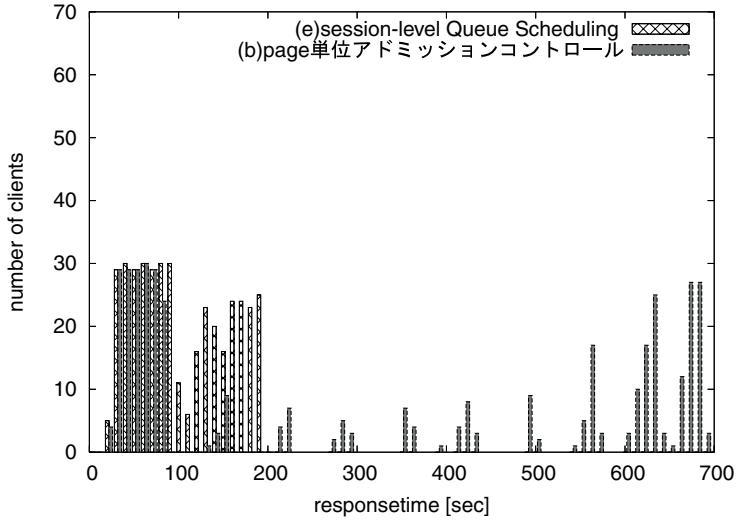

図 4.6: レスポンスタイム (400 クライアント時)

4.2.2　セッション処理性能差の原因

セッション処理性能の差異を生み出す原因を分析するために、それぞれの手法における途中失敗数を図 4.7 に示す。この結果より、途中失敗数の増加がセッション処理性能の低下を引き起こしている事が分かり、(e) の Session-level Queue Scheduling が最も途中失敗数を抑えることに成功しているため、過負荷時においても高いセッション処理性能を維持できていると考えられる。

さらに、(a) の制御無し、(b) のページ単位アドミッションコントロール、(e) の Session-level Queue Scheduling の三手法における、セッションの途中失敗数の内訳を表 4.1 に示す。内訳は、(a) の制御無しで測定できた最大クライアント数 (125)、および全体を通しての最大クライアント数 (400) を抽出した。(e) の Session-level Queue Scheduling 以外では、リソースを多く消費するページ (5 ページ目) でコネクションタイムアウトによる途中失敗が多く発生していた。一方、本手法では、途中失敗が低く抑えられると共に、ほとんどのコネクションタイムアウトが先頭ページで起こっているため、セッションを途中まで処理してから失敗することがほとんど無い。つまり、セッションスケジューラによるセッション処理の保護が高いセッション処理性能の維持につながっていることが分かる。

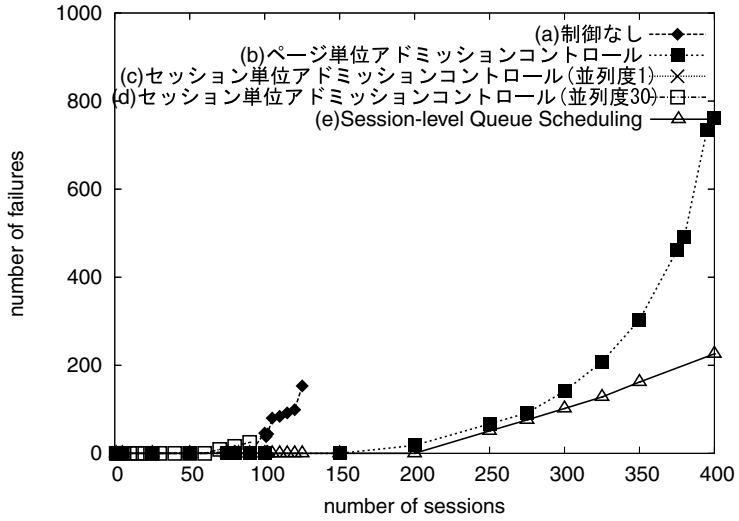

図 4.7: セッション失敗数 (タイムアウト 60 秒)

表 4.1: セッション内の各ページでの失敗数の内訳 (125 クライアント/400 クライアント)

ページ番号	1	2	3	4	5	6	7
(a) 制御無し	0/-	0/-	0/-	3/-	150/-	0/-	0/-
(b) ページ単位アドミッションコントロール	0/0	0/0	0/0	0/0	0/735	0/0	0/0
(e) Session-level Queue Scheduling	0/226	0/0	0/0	0/0	0/0	0/0	0/0

4.2.3　スケジューラの挙動

セッションスケジューラとページスケジューラの挙動を示すために、sessionInterval の調節とそれにより変動するページキューの平均長の推移を図 4.8 に示し、各ページ maxThread の調節を図 4.9 に示す (ページ 2〜4 とページ 6、7 はページ 1 の結果とほぼ同様の結果であり、図を見やすくするために省略)。測定データはクライアント数が 400 の時のものである。グラフから、ページキューが長すぎて待機リクエスト数が多いときは、セッションスケジューラが sessionInterval を増加させることが分かる。逆に、ページキューが短すぎる場合には、sessionInterval が減少していく。また、各ページの maxThread は、セッション中の最も重いページにあわせて変動し、一定の時間を経過して安定している。

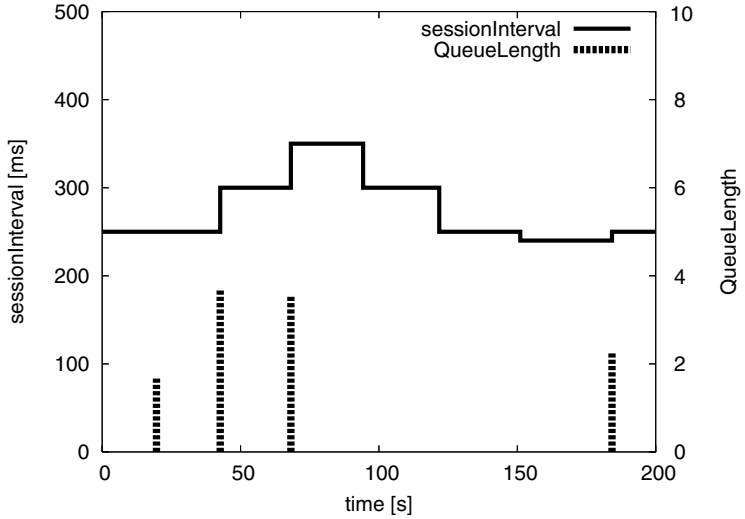

図 4.8: sessionInterval とページキューの変動

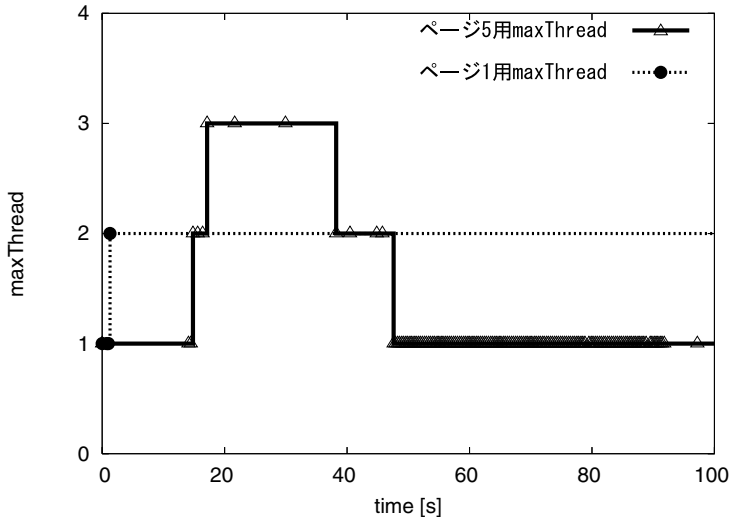

図 4.9: 各ページの maxThread の変動

4.2.4　クライアントの挙動による影響

クライアントの挙動がセッション処理性能に与える影響を調べるために、タイムアウトとシンクタイムをそれぞれ変更して同様の実験を行った。

4.2.5　タイムアウト時間の影響

タイムアウト時間がセッション処理性能へ与える影響を調べるためにシンクタイムは 3 秒のままで、タイムアウト時間を 30 秒へ変更して同様の実験を行なった。図 4.11 より、(e) の Session-level Queue Scheduling のセッション処理性能は、タイムアウト時間が 60 秒の時の結果 (図 4.4) とさほど変わらない。これは、セッションスケジューラによりタイムアウトが発生しないようにセッションが保護されているためである。一方、その他の手法はタイムアウト時間を短くすることで、より少ないクライアント数でもセッション処理性能が低下するようになった。これは、セッションタイムアウトに達するまでの同時セッション数が少なくなった (図 4.12) からである。

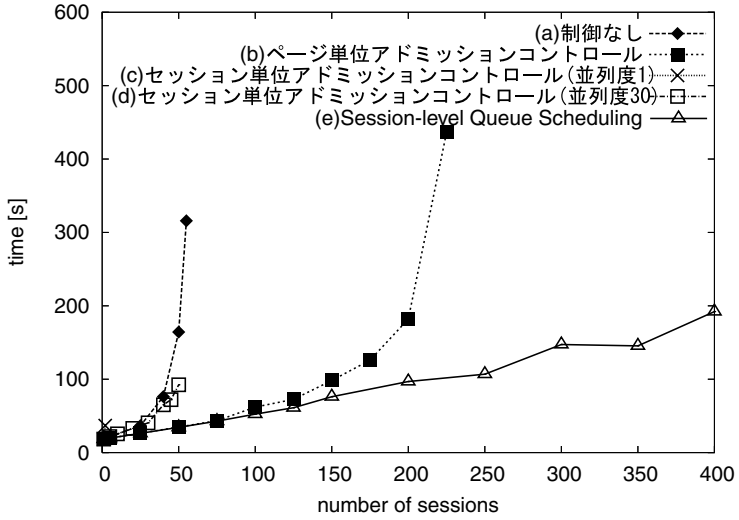

図 4.10: 総処理時間 (タイムアウト 30 秒)

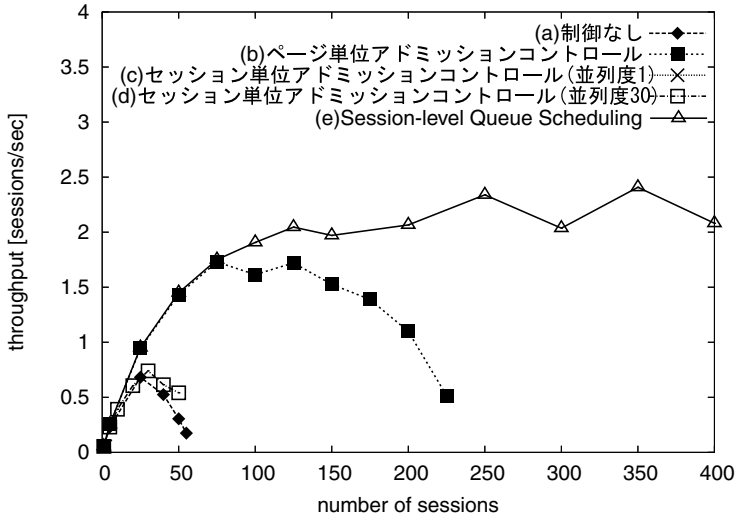

図 4.11: セッション処理性能 (タイムアウト 30 秒)

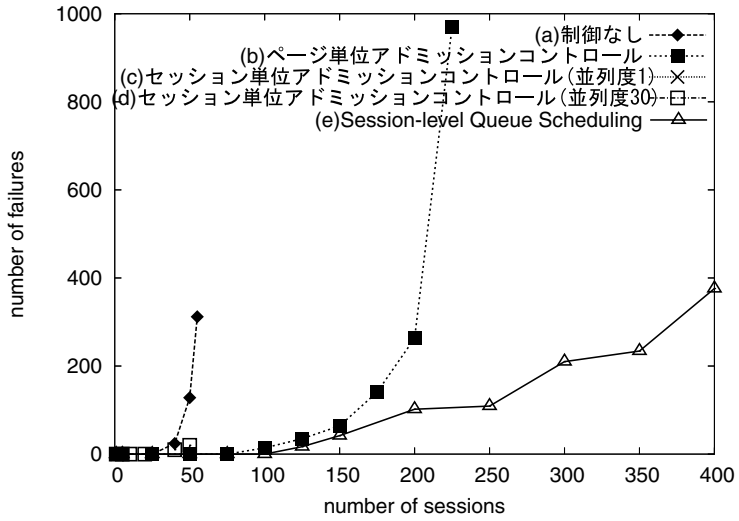

図 4.12: 途中失敗数 (タイムアウト 30 秒)

4.2.6　シンクタイム時間の影響

シンクタイム時間がセッション処理性能へ与える影響を測定するためにタイムアウトは 60 秒のままで、シンクタイム時間を 3 秒から 1 秒へ変更して同様の実験を行なった。セッション処理性能は、シンクタイム 3 秒の場合 (図 4.4) とほとんど変わらなかった (図 4.14)。シンクタイムの長短は次にリクエストを投げるまでの時間が変わるだけで、タイムアウトによる失敗にはそれほど影響しないため (図 4.15)、セッション処理性能自体にはほとんど差が生じなかった。

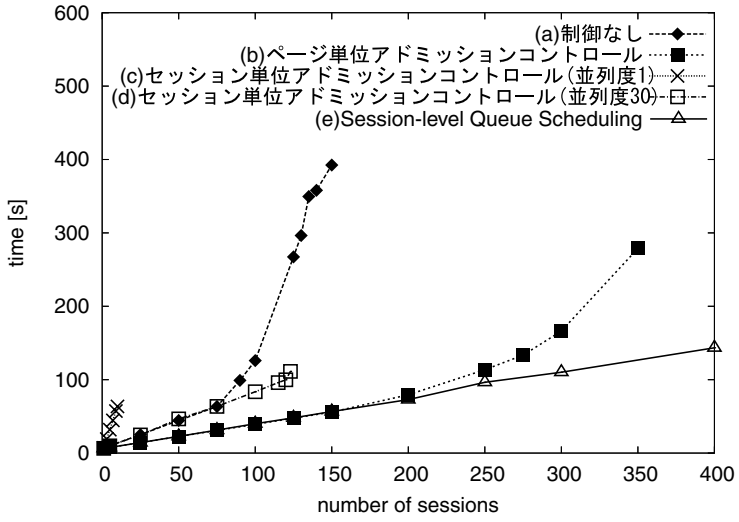

図 4.13: 総処理時間 (シンクタイム 1 秒)

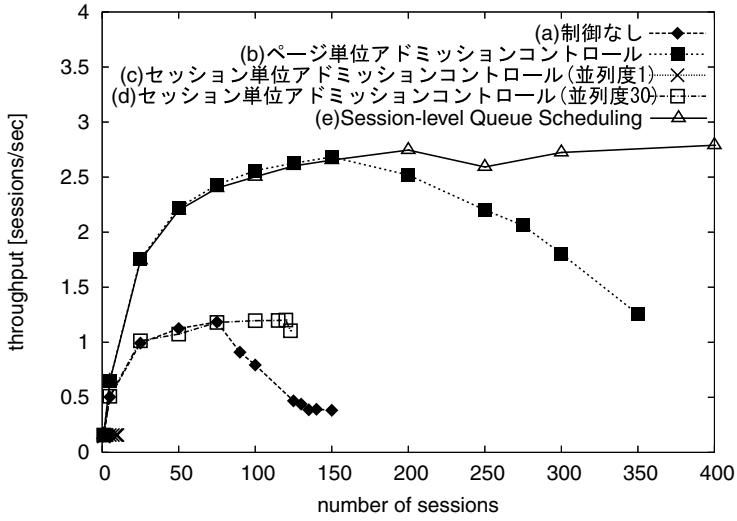

図 4.14: セッション処理性能 (シンクタイム 1 秒)

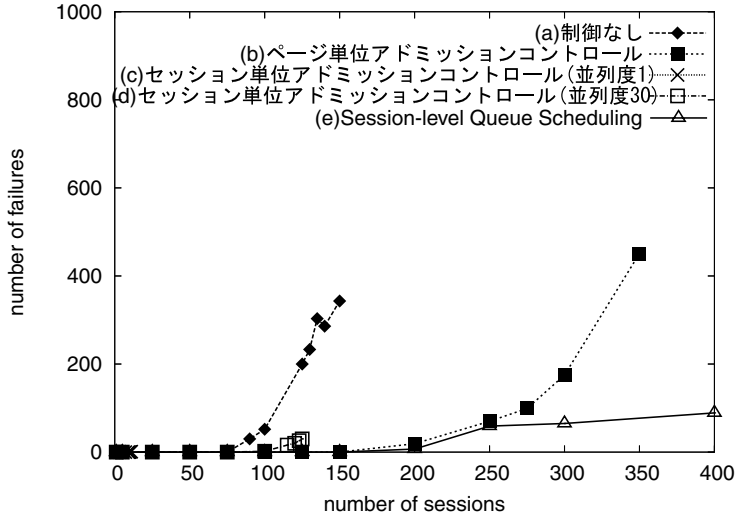

図 4.15: 途中失敗数 (シンクタイム 1 秒)

4.3　サーバー処理性能によるセッション処理性能に与える影響

サーバ処理性能がセッション処理性能に与える影響を調べるために、サーバ計算機の基本処理性能を変更して同様の実験を行った。使用したサーバ計算機は以下の二つ。

- 低性能サーバ
  UltraSPARC　750MHz × 2 の CPU、1024MB のメモリ、100BaseTX の NIC を持ち、OS には Solaris8、Web アプリケーションサーバには Tomcat 3.3.1 [7] を動作させた。

- 高性能サーバ
  Intel Xeon 3.06GHz × 2 の CPU、2GB のメモリ、ギガビットイーサネットを持ち、OS には Linux 2.4.2、Web アプリケーションサーバには Tomcat 3.3.1 [7] を動作させた。

なお、実験対象アプリケーションはこれまでと同様のものを用い、クライアントの挙動としてはシンクタイムは 3 秒、タイムアウトは 1 分で実験を行った。

4.3.1　低性能サーバ

サーバ計算機の処理性能を低下させた場合に、セッション処理性能に与える影響を調べるために、Web アプリケーションサーバを動作させるサーバ計算機のマシン性能を低下させ, 4.2 節と同様の実験を行った。

図 4.17 より、サーバ計算機のマシン性能を低下させたことにより、(e)Session-based Queue Scheduling 以外の 4 手法では、より少ないクライアント数でセッション処理性能が低下するようになった。これは、計算機の処理性能が低下したことにより、リソース競合が発生しやすくなり、より少ないクライアント数でタイムアウトが発生するようになったためであると考えられる。一方、(e)Session-based Queue Scheduling を利用した場合には、計算機の性能を低下させた場合にでも、一定のセッション処理性能を維持することができ、本手法の有効性を示すことができた。

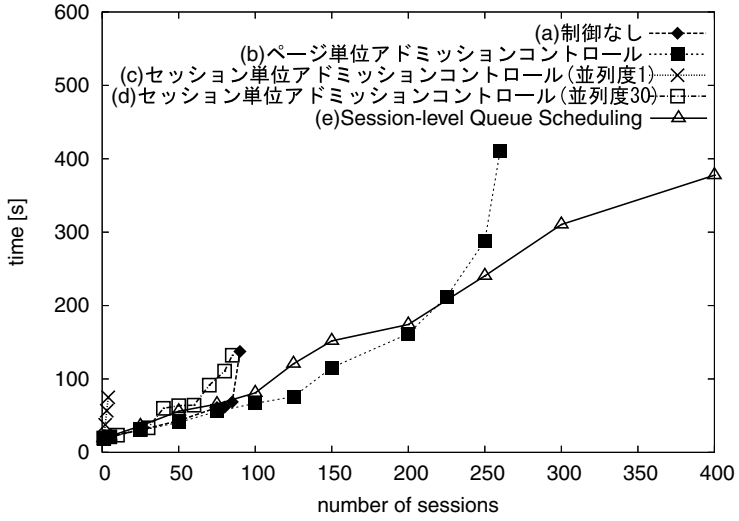

図 4.16: 総処理時間 (タイムアウト 60 秒)

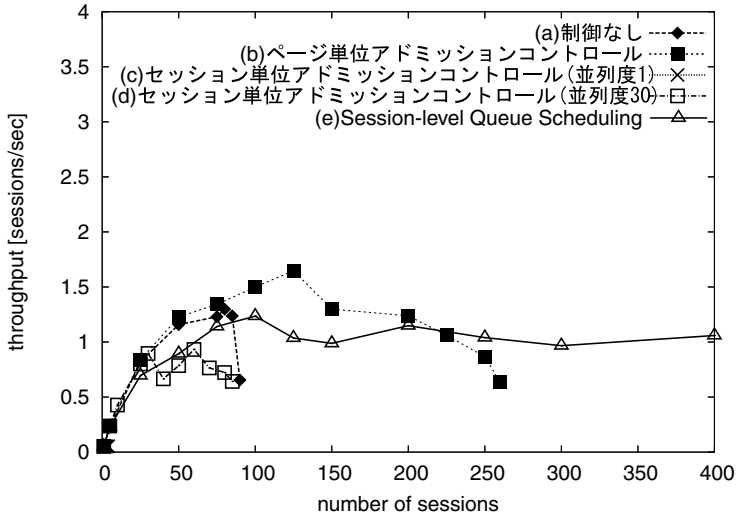

図 4.17: セッション処理性能 (タイムアウト 60 秒)

4.3.2　高性能サーバ

次は、逆にサーバ計算機の処理性能を向上させた場合に、セッション処理性能に与える影響を調べるために、Web アプリケーションサーバを動作させるサーバ計算機のマシン性能を向上させ, 4.2 節と同様の実験を行った。

図 4.19 より、サーバ計算機のマシン性能を向上させることで、全ての手法でリソース競合が発生しにくくなり、セッション処理性能が向上したことが見て取れる。しかし、(e) の Session-level Queue Scheduling 以外の手法では、相変わらず一定のクライアント数を超えた時点でセッション保護が実現できなくなるために、セッション処理性能の低下が緩やかながら測定された。以上の結果より、サーバ計算機のマシン性能を向上させたとしても、本手法以外の制御を行った場合には、セッションが途中失敗する可能性は残り、それによりセッション処理性能が低下するということが分かる。

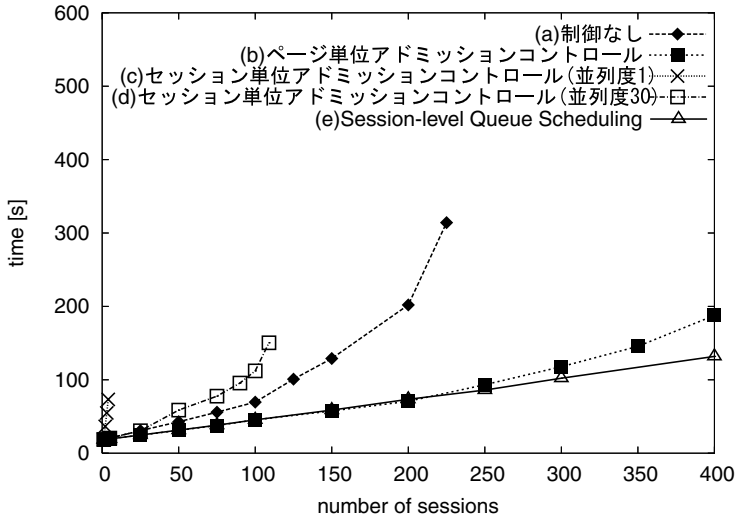

図 4.18: 総処理時間 (タイムアウト 60 秒)

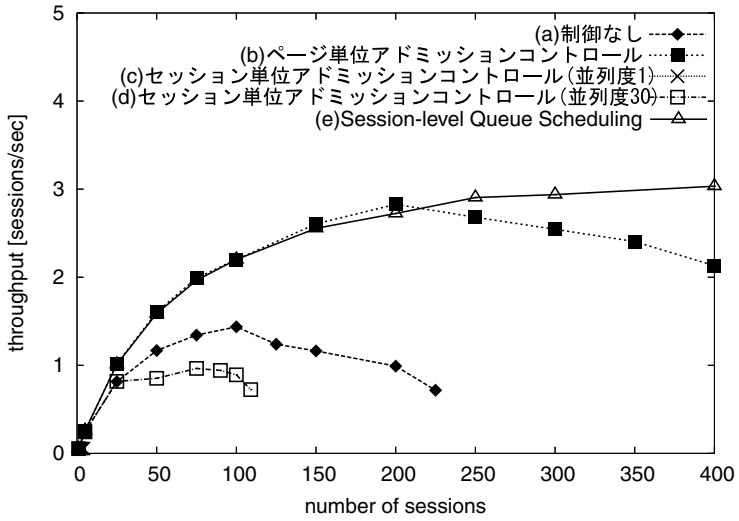

図 4.19: セッション処理性能 (タイムアウト 60 秒)

4.4　実験のまとめ

本章では、Session-level Queue Scheduling の有効性を確かめるために、セッション処理性能に関して、いくつかの手法との比較実験を行った。結果として、クライアントがレスポンスを無制限に待つという特殊な条件下を除いて、行った全てのケースでの本手法によるセッション処理性能が最も高く維持できるということが分かった。また、レスポンスを無制限に待つ場合に関しても、処理性能が低いというわけではなく、最も高い処理性能を示したページ単位のアドミッションコントロールと同程度の処理性能に落ち着いた。以上のことから、今回行った実験に限っては Session-level Queue Scheduling を用いることで、高いセッション処理性能を維持することができ、有効性を示すことができたと言える。

第5章　まとめと議論

5.1　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を改善する Session-level Queue Scheduling について述べた。リソース競合の解消を行いページの処理性能向上を行うページスケジューラと、セッションを処理する最大クライアント数を制限するセッションスケジューラを同時に用いることで過負荷時における処理性能向上・維持を行う。Session-level Queue Scheduling を実現するプロトタイプシステム ContorlServlet を用いた実験を行った結果、本手法を用いることで過負荷時においても高いセッション処理性能を維持することができ、一定の有効性を示すことができた。

ページスケジューラは、Progress-based Regulation を応用しページ処理の進捗結果をもとにリソース競合の検知を行う。既存手法の多くは特定リソースの利用率を実際に監視してリソース競合の検知を行っており、より正確にリソース競合を検出することができる。しかし、そのような手法では実際の Web アプリケーションが非常に多岐に渡るリソースを消費することを考慮にいれた場合、リソース監視によるオーバヘッドが多くなり処理性能を低下させてしまう恐れがある。そういった問題を解決するためにも、本手法では進捗状況をもとにしたリソース検知を行い、実験からも十分にリソース競合の検知が可能であることを示した。

一方、セッションスケジューラは既存手法のようにセッション処理数に上限を設定するような手法ではなく、各クライアントに許可を与える際に時間的遅延を掛ける新しいセッション単位のアドミッションコントロールを開発・利用した。セッション処理は複数のページから構成されており、ページ間の移動は全てクライアントの思考に依存している。例えば、非常に短い時間で移動するものから、長い時間を掛けて移動するものまで多様である。また、セッションを終了させる際にもログアウト処理を行わずとも途中のページで抜けることも可能であり、そのようにして途中終了させられてしまうと、サーバ側からそれを知るすべが現在のところ無い。こういったクライアント依存の挙動を考慮した場合には、本手法のような若干あいまいな制御を行った方が処理性能を向上できると判断し、このような手法を用いた。

実験では、商用サイトの商品購入アプリケーションに見立てたアプリケーションを用いて既存手法との性能比較を行った。ページの移動に掛かる時間 (シンクタイム) やレスポンスの最大待ち時間 (タイムアウト) を設定した現実的なクライアントの挙動を用いた場合には、本手法のみが過負荷時にも高いセッション処理性能を維持することができた。

5.2　議論

ここでは、議論として本手法と共に用いることでさらなるセッション処理性能向上を見込まれる手法について幾つか紹介をおこなう。それぞれページの処理性能を向上させるための仕組み、セッション保護を助けるための仕組み、異なる OS 間の性能差を解決する手法である。

ページ処理性能の向上手法

過負荷時の性能を改善するために、SEDA アーキテクチャ [14] を用いたアドミッションコントロールの研究も行われている [13]。SEDA はサーバの処理性能を向上させるための手法で、リクエスト処理を細かいステージに分割してリソース管理をより細かく行なう。各ステージでアドミッションコントロールを行うことで、過負荷時にも各ステージの性能を保つことができる。SEDA アーキテクチャを用いるためには Web アプリケーションの大幅な変更が必要となるが、本研究の手法と組み合わせることも可能であると考えられる。

セッション保護の実現手法

セッション処理性能の向上を妨げる原因の一つであるコネクションタイムアウトを発生させにくくするために、ハートビートと呼ばれる機構がサーバに実装されている場合がある。この手法は、タイムアウトする可能性のあるページへのリクエストに対して簡単なページを返しておき、そのページに埋め込まれたメタタグにより一定時間毎にブラウザにページをリロードさせる。これにより、ブラウザのタイムアウトを防止することができる。しかし、ハートビートがサーバに実装されていたとしても、サーバのタイムアウトやクライアントによる読み込みの中止によりセッションが途中終了する場合があるため、本手法も必要になる。

OS 間の処理性能の影響差の減少手法

Web アプリケーションは異なる OS でサービスを提供した場合には、各ページの処理性能が大きく変動することがある。例えば、OS によってはリソース消費量の多い重いページの処理を優先したり、軽いページの処理を優先したりといったことが起きる。そのような OS 間の処理性能差は、サービスをどのサーバ計算機上でも等しく提供したいと考えた場合に大きな問題を引き起こす。このような、OS 間の性能差を減少させることを目的とした研究 [16] がなされている。各ページの処理途中で一定時間の Sleep 処理を埋め込むことで、OS のスケジューラの影響を減少させることで実現している。このような手法を、本手法と同時に用いることで異なる OS 上でも、均一なセッション処理性能を提供できると考えられる。

参考文献

[1] N. Bhatti and R. Friedrich. Web server support for tiered services. In *IEEE Network, 13(5):64–71*, 1999.

[2] J. Carlstrom and R. Rom. Application-aware admission control and scheduling in web servers. In *Proceedings of IEEE Infocom*, 2002.

[3] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. In *Proceedings of the International Workshop on Quality of Service*, 1998.

[4] J. Douceur and W. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of Symposium on Operating Systems Principles*, pp. 247–260, 1999.

[5] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web*, pp. 276–286, 2004.

[6] Jakarta. Apache. http://apache.org.

[7] Jakarta. Tomcat. http://jakarta.apahe.org.

[8] K. Li and S. Jamin. A measurement-based admission-controlled web server. In *Proceedings of INFOCOM*, pp. 651–659, 2000.

[9] D. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proceedings of 20th International Conference on Data Engineering*, 2004.

[10] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the*

5th USENIX Symposium on Operating Systems Design and Implementation, 2002.

[11] Sun MicroSystems. Java Servlet Technology. http://java.sun.com/products/servlet/.

[12] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of 2001 USENIX Annual Technical Conference*, pp. 189–202, 2001.

[13] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems*, 2003.

[14] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of Symposium on Operating Systems Principles*, pp. 230–243, 2001.

[15] 松沼正浩 千葉滋 佐藤芳樹光来健一. 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 Page-level Queue Scheduling. 第 7 回 プログラミングおよび応用のシステムに関するワークショップ (SPA 2004).

[16] 日比野秀章 松沼正浩 光来健一千葉滋. アクセス集中時の web サーバの性能に対する OS の影響. 日本ソフトウェア科学会第 21 回大会, 2004.