平成17年度 修士論文

J2EE アプリケーション
における
アプリケーションレベル
スケジューリング

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 04M-37254

日比野 秀章

指導教員

千葉 滋 助教授

平成18年1月20日

概要

現在、J2EE アプリケーションは様々なシステムで使用されているが、リクエストの増大や規模の拡大によりシステムが過負荷になり、重要な処理が滞ってしまう場合がある。本来的にはハードウェアの増強や、軽量化、クラスタ化といったソフトウェアの抜本的改修によって対処するべきだが、期間や予算の制約で、必ずしも可能とは限らない。安価な対策として、QoS 制御を後から追加するという方法が有効である。

本論文ではまず、何を制御することが QoS の改善に最も有効かを調べるために、複数の OS での挙動の違いを詳細に比較し、過負荷時に J2EE アプリケーションの挙動に影響を及ぼす主な要因を分析する。結果として、OS のスケジューリングポリシーとスレッドライブラリの実装がリクエストの処理に大きな影響を与えることを示す。

この分析結果をうけ、本論文では J2EE アプリケーションにきめ細かいリクエストスケジューリングを追加し QoS を改善するアプリケーションレベルスケジューリングを提案する。アプリケーションレベルスケジューリングでは、アスペクト指向プログラミング (AOP) を用いて、スケジューラとアプリケーションの結合を行う。そのため、アプリケーションのコードに手を加える必要がなく、ロジックを壊してしまう危険性も少ない。アスペクトを変更することでスケジューラを再利用することもできる。アプリケーションレベルスケジューリングでは、AOP を用いてアプリケーション内部からスケジューラを頻繁に呼び出しことにより、スレッドの操作が困難な Java で擬似的なスレッドスケジューリングを実現する。またアプリケーションレベルでスケジューリングするので、OS やミドルウェアを変更する手法に比べシステムへの影響が少なく、検証時間が短くて済む。

本手法の有効性を示すため、河川水位監視システムに本手法を用いて QoS 制御を追加し、既存のリクエスト単位のアドミッションコントロールと比較する実験を行った。この監視システムはリクエストの増加に伴い重要性の高い水位収集処理が滞るという問題があったが、本手法を用いるとアドミッションコントロールよりも短時間で水位収集処理を完了させることができた。

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、様々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きました。深く感謝いたします。

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の最後までお世話になりました。研究の基本的なことから方向性、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きました。

また、貴重な助言をくださった三菱総合研究所 佐藤芳樹氏、東京工業大学 西沢無我氏、柳澤佳里氏に、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

図 目 次

表 目 次

第1章　はじめに

J2EE サーバでは静的なページを扱う Web サーバと比べて複雑な処理を行うため、サーバの負荷が高くなりやすい。サーバが過負荷状態になると、サーバ上で行われる処理が全体的に遅延してしまう。時間的制約のある重要な処理の場合、遅延が発生してデッドラインを超えるとシステムへの影響が大きい。

J2EE サーバが過負荷状態になってしまう原因としてはいくつかあげられる。まず、サーバへのリクエストが増大して、当初の予想よりも負荷が高くなってしまう場合が考えられる。オンラインショッピングサイトのように、どの程度の利用者が見込めるか判断が難しいシステムの場合、利用者の増加によりサーバへのリクエストが当初の予想を超えてしまうことがある。また、システムの規模を拡大した場合にも、当初の予想よりも高い負荷がかかり、サーバが過負荷状態になってしまうことがある。例えば、2005 年 7 月 23 日に発生した地震では、東京都の震度計ネットワークシステムが正常に動作せず、気象庁にデータを送信するのに約 30 分の遅れが出た。このシステムを構築した当時は都内に 14 箇所しか震度計は設置されていなかったが、その後震度計が増設されていった結果、サーバの計算処理能力が追いつかず問題が発生してしまった。

過負荷状態を改善する一般的な方法としては、ハードウェアの増強やアプリケーションの改修があげられる。しかし、これらの方法にはかなりの時間と費用がかかる場合が多いため、すぐに対処しなければならない場合や費用に見合う利益が得られない場合には現実的ではない。そこで安価な対処法として、ボトルネックを分析して QoS 制御を後から追加することにより、過負荷状態でも重要な処理が行われるようにすることが考えられる。OS やミドルウェアを置き換えたり、アプリケーションのコードを書き変えたりすると動作の検証に時間がかかるため、OS やミドルウェアには変更を加えず、アプリケーションのコードも書き換えることなく QoS を追加できることが望ましい。

また、Web システムの構築に広く利用されている J2EE アプリケーションが、リアルタイム性の高いシステムの構築に向いていない 1 つの要因として、リアルタイム性の強いシステムの構築に必要な CPU やメモリといった資源の管理能力が乏しいということがあげられる。さらに、Java

ではスレッドを外部から操作することが推奨されていないため、リクエスト処理を外部からこまめにスケジューリングするのが困難だという問題点がある。

本稿ではまず、過負荷状態のアプリケーションサーバの QoS 制御の手法を開発するにあたり、過負荷状態でリクエストの処理に影響を及ぼす主な要因について分析を行なう。その結果、OS のスケジューリングポリシーやスレッドライブラリの実装が大きく影響を及ぼしていることを示す。この分析から、過負荷状態での QoS 制御において、リクエストの処理途中における細かな制御が重要であることが判明する。

そこで本稿では、アスペクト指向プログラミング (AOP) を用いて、アプリケーションに QoS (Quality of Service) 制御を追加するアプリケーションレベルスケジューリングを提案する。アスペクトを用いてスケジューラとアプリケーションを結合するため、アプリケーションのコードを手で書き換える必要がなく、アプリケーションのロジックを壊してしまう危険性が少ない。また、アスペクトを介してアプリケーション内部の様々な箇所からスケジューラを呼び出せるため、きめ細かい制御を行うことができる。

アプリケーションレベルスケジューリングの有効性を示すために、JBoss 上で開発された河川水位監視システムに対して、アスペクト指向システムの GluonJ [5] を利用して QoS 制御を追加した。時間的制約のある重要度の高い処理を行っている間は重要度の低い処理を一時的に停止させるというスケジューリングポリシを適用した結果、過負荷時でも重要度の高い処理が遅延しないようにすることができた。さらに、従来のアドミッションコントロールとの比較も行い、提案手法はより効果的な優先度制御ができることを確認した。

以下、2 章では従来手法とその問題点について述べ、3 章ではアプリケーションサーバの挙動の分析について述べる。4 章では提案手法と河川水位監視システムに適用した制御内容について述べる。5 章では実験について述べ、6 章で本稿をまとめる。

第2章　従来の過負荷制御

この章では、従来の過負荷制御に関する対処方法とその問題点について述べる。

過負荷状態を改善する一般的な方法としては、(1) ハードウェアの増強、(2) アプリケーションの抜本的な改修などが挙げられる。(1) については、より高速な計算機に置き換えたり、クラスタ構成を取っている場合には台数を増やすなどして、システムを高性能にする方法が取られる。(2) については、アプリケーションを軽量化したり、クラスタ化・多階層化することにより負荷を分散させる、といった方法が取られる。

ただし上記の方法では、過負荷状態を改善するのに長い時間と膨大な費用がかかってしまう。ハードウェアの選定や追加の開発、検証が必要であり、特にアプリケーションの改修を行う場合や受注生産のハードウェアを購入する場合には、長い時間がかかることが多い。商用のシステムのように、システムの正常な動作と収益が密接に関わっている場合には、すぐに対処しなければ損失が大きくなってしまう。また、変更にかかる費用も企業によっては重大な問題である。特にソフトウェアの改修には膨大な費用が必要となるため、それに見あう利益が得られない場合には費用を出すことはできないだろう。

そこで過負荷状態に対するより安価な対処法として、ボトルネックを分析し、QoS (Quality of Service) 制御を後から追加することが考えられる。QoS 制御を追加することで過負荷時にすべてのサービスの実行が滞る事態を避けることができる。

2.1　サーバのボトルネックの解析

ウェブサーバ及びウェブアプリケーションサーバにおけるボトルネックの分析に関する既存研究について、その一部を述べる。

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 [2]。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 [13]。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 [12]。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。

2.2　QoS 制御の追加

ウェブサービスの分野で過負荷の管理に関連した QoS 制御の手法が多数提案されている。この節では、アドミッションコントロール、制御理論、service degradation に分けて既存研究について述べる。

2.2.1　アドミッションコントロール

アドミッションコントロールは、指定されたポリシーにしたがってシステムが受け入れるリクエストを制限する手法である。ウェブサービスの分野で、アドミッションコントロールを利用した QoS 制御の手法が多数提案されている。

Cherkasova と Phaal [4] は、個々のリクエストや接続ではなく、セッションを単位とするセッションベースのアドミッションコントロールを提案している。多くのインターネットサービスで、いったんセッションを確立したユーザからのリクエストを拒否するのは望ましくない。提案手法では、CPU の利用率に基づき、CPU の利用率が指定地を越えた場合、新しいセッションは拒否される。彼らは、シミュレーションを行い、パラメータの設定によるトレードオフがあることを示している。

Voigt [17] らは、過負荷状態のウェブサーバでアドミッションコントロールと service differentiation を行なうためのカーネル空間の機構を示している。彼らの手法では、落とされる SYN パケットに応じて接続数を制限し、カーネル空間で HTTP リクエストの識別を行い、リクエストの URL やクライアントの IP アドレスによってソケットのリッスンキュー内での位置を決める。リクエストのタイプや IP アドレスにしたがってソケットのリッスンキュー内での順番を決めるので、ウェブサーバは特定のリクエ

ストに対し、優先度を与えることができる。彼らは、AIX で提案手法を評価し、良い性能が得られることを確認している。

Web2K [14] では、リクエストのタイプと、IP アドレスやクッキーによるユーザクラスをもとにウェブサーバのフロントエンドでアドミッションコントロールを行なう。Web2K では、SYN パケットを落とさないように接続をすぐに受け付け、リクエストは優先度に基づいて内部のキューに格納され、ウェブサーバはこのキューからリクエストを取り出す。アドミッションコントロールは各優先度のキューの長さを制限するために利用されている。Web2K はウェブサーバと TCP/IP スタックの間で機能するミドルウェアで、ウェブサーバに追加されるダイナミックライブラリとして実現されており、OS やウェブサーバの変更をする必要はない。Apache を用いて提案手法を評価し、優先度の高いリクエストが過負荷状態で安定したレスポンスタイムを維持することを示している。

Kanodia と Knightly [10] は、複数のサービスクラスのレスポンスタイムを制御するために、従来のアドミッションコントロールに加えて service envelope を利用している。service envelope は共有リンク上のトラフィックを特徴付けるために用いられる技術で、リクエストの割合や容量を扱うために利用している。彼らは、シミュレーションを用いて提案手法を評価している。

Welsh らは、並列性の高いインターネットサービスの設計方法として、staged event-driven architecture(SEDA [19]) を提案している。SEDA では、アプリケーションを複数のステージとそれをつなぐキューによって構成し、負荷に応じてステージ毎にアドミッションコントロールやリクエストスケジューリングを行うことで、効率のよい資源利用を実現している。

2.2.2　制御理論

制御理論は、動的なシステムやフィードバック制御の振る舞いを論理づけて考えるために用いられる形式的な枠組みである。ウェブサービスの分野で QoS 制御に制御理論を用いた手法が多数研究されている。

Abdelzaher と Lu [1] は、CPU の利用目標を $\ln 2$ に維持する制御理論に基づいたアドミッションコントロールを提案している。$\ln 2$ というのはリアルタイムスケジューリング理論に基づくもので、標準的な OS に適応することを想定しているわけではない。彼らは、サーバの性能にリクエストの到着率とネットワークのバンド幅に基づく単純な線形モデルを用いた proportional-integral(PI) コントローラを設計している。このモデルは静的なウェブページのアクセスに限られている。

Diao ら [6] は、資源の利用制約を満たすように、制御理論を用いて Apache ウェブサーバの KeepAlive と MaxClient パラメータをチューニ

ングしている。彼らは PI コントローラを設計し、CPU とメモリの使用率が指定されたレベルを満たすように、サーバのプロセス数と接続毎のタイムアウト時間を調整している。

2.2.3　service degradation

過負荷の対処方法として、一部の研究者はアドミッションコントロールやクライアントを拒否を行なうのではなく、service degradation を用いた研究を行なっている。service degradation とは、解像度の低い画像など、より小さいコンテンツを提供するという形で、クライアントに提供されるサービスを減らす方法である。

service degradation に関連する例としては、個々の HTTP リクエストによる負荷を減らすために、全てのウェブページを置き換えるという手段がとられたことがあった。これは、2001 年 9 月 11 日に、世界貿易センターとペンタゴンへのテロリストの攻撃の後、サーバの負荷が高まったため CNN.com で実際にとられた方法です。CNN ではイーサネットパケットを 1 つだけ含む単純な HTMP ページにフロントページを置き換えた [11]。

2.3　コード変更を必要としない QoS 制御の追加手法

2.2 で述べたように、多数の QoS 制御が提案されている。これらの既存手法は、OS やウェブサーバ、アプリケーションを変更する必要がある。しかし、期間や予算の制約を考慮に入れた場合、次の用件を満たす必要がある。1 つはアプリケーションのコードの変更を避けることである。なぜなら、コードに手を加えることにより、時間をかけて検証したアプリケーションのロジックを壊してしまう危険性があるからである。もしアプリケーションに変更を加えてしまうと、検証に時間がかかるため、費用も増加してしまう。また、OS やミドルウェアに対する大幅な変更も検証に時間がかかるため避けた方がよい。リアルタイム OS やリアルタイム Java など、リアルタイム機能を持つ OS やミドルウェアに置き換える方法は、細かい QoS 制御ができる反面、システム全体の検証に時間がかかってしまうからである。

そのため、QoS 制御を追加する際、コード変更を必要としない手法が望ましいといえる。以下、QoS 制御を追加する際、コード変更を必要としない方法について既存研究を述べる。

2.3.1 Capriccio

Capriccio [18] はユーザレベルのスレッドパッケージで、並列性の高いサーバを実現するために設計された。Capriccio では、アプリケーションを実行しながら、プログラムの各箇所で消費されるリソースの量を計測し、blocking graph と呼ばれるグラフにその情報をまとめる。このリソース消費の情報に基づいて、効率よくリクエストが処理されるようにスレッドのスケジューリングを行う。アプリケーションのコードに手を加えることなく、リソースが有効活用されるようにスケジューリングが行われる。

しかし、スレッドプールを利用する J2EE アプリケーションの場合、J2EE サーバ上で提供されている多数のサービスを同一のスレッドが処理することになるが、処理しているサービスの内容を区別して制御をするのは困難である。

2.3.2 Gatekeeper

Elnikety らは、多階層の商用ウェブサイトでアドミッションコントロールとリクエストスケジューリングを実現する方法を提案している。彼らは gatekeeper [8] と呼ばれるプロキシに、アドミッションコントロールやリクエストスケジューリングの機能をつけ、各階層の間にこのプロキシを置くことでリクエスト処理の制御を実現している。プロキシでは、リクエストの種類を区別しながら、その処理によるコストを見積もり、過負荷防止やリクエストスケジューリングを行う。そのため、ホスト OS、ウェブサーバ、アプリケーションサーバ、データベースへの変更を必要としない。

しかし gatekeeper では、階層の間でしか制御を適用することができないため、各階層の内部で処理の途中に制御を適用することはできない。そのため、時間のかかるリクエスト処理に対し、その処理途中ですぐに制御を適用するなどの細かい制御を行うことができない。

2.3.3 MS Manners

MS Manners [7] は、アプリケーションに手動でコードを埋め込むことで、重要度の低い処理による資源の競合を解消する機構を提供している。MS Manners では、利用者が手動で埋め込んだコードにより、低いプロセスの進捗度を監視し、進捗の悪化を資源の競合と判断して、重要度の低い処理を停止させる。また、アプリケーションを修正することなく外部から MS Manners を適用できる BeNice というモニタも実装されている。

MS Manners では、低いプロセスの進捗度を監視するために手動でアプリケーション内にコードを埋め込まなければならいので、煩雑である。

BeNice を利用するとそのような煩雑さはなくなるが、Java では外部からスレッドを制御することは推奨されていないため、同様の方法を J2EE アプリケーションに適用することは難しい。

2.3.4　Re-QoS

Re-QoS [15] では、コンポーネントを利用したリアルタイム組み込みシステムにおいて、QoS 制御を追加するための手法を提案している。Re-QoS では、アスペクトと QoS コンポーネントからなる QoS アスペクトパッケージを定義し、QoS 制御のない既存のシステムに追加することで、QoS 制御を適用するすることができる。また、アプリケーションの QoS 要求に応じて、QoS アスペクトパッケージ内のアスペクトを交換することで、容易に QoS 管理ポリシーを変換することができる。アスペクトを通じて適用する QoS 管理ポリシーを変更を行うため、アプリケーションのコードに手を加える必要はない。

Re-QoS では、主に組み込みのデータベースを対象にしている。そのため、本研究とは対象とするアプリケーションが異なっている。また、J2EE アプリケーションのようにスレッドの操作がある程度制限されている場合に、どのように細かい制御を実現するべきかということには触れられていない。

2.4　QoS 制御の追加のまとめ

過負荷時のウェブサーバに対する QoS 制御について、アドミッションコントロールや制御理論、service degradation を用いた手法が多数提案されている。しかし、多くの手法は OS やサーバ、アプリケーションのコードを変更に対する配慮がなされていなかった。システムが過負荷になってしまった際には、過負荷が改善されるまでにかかる期間や予算の制約を考慮する必要ある。QoS 制御を追加して対処する場合には、コード変更による検証時間の増大を避けるため、コード変更はなるべく避ける必要がある。

一部の研究者は、コード変更を避けることを重要視した QoS 制御を提案している。しかし、現在システムの構築によく利用されている J2EE アプリケーションを対象とした場合、それらの手法は必ずしも有効ではない。MSManners や Capriccio は、J2EE アプリケーションの構成や仕組み上、そのまま利用することは困難である。また、Gatekeeper では、制御できる箇所が制限されており、Re-QoS では対象としているアプリケーションが異なっている。

第3章　アプリケーションサーバの振る舞いに影響する要因の分析

この章では、過負荷状態のアプリケーションサーバで、アプリケーションサーバ上で動作する複数のサービスの処理性能に最も影響を及ぼしている要因について追求を行う。まず、複数の OS 上でアプリケーションサーバを動作させ、その振る舞いを比較する。その違いを生む要因こそ、アプリケーションサーバで各サービスの処理性能に影響を及ぼす要因であると考え、OS 間での相違の分析を行う。

3.1　OS 間の相違

アプリケーションサーバの振る舞いに与える OS の影響についてみるために、複数の OS 上で以下の実験を行った。

3.1.1　実験内容

Tomcat アプリケーションサーバ [16] の上で動くいくつかのサービスを作成して、複数の OS 上でその振る舞い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。OS が与える影響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、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。

OS 間での振る舞いの違いをみるために、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

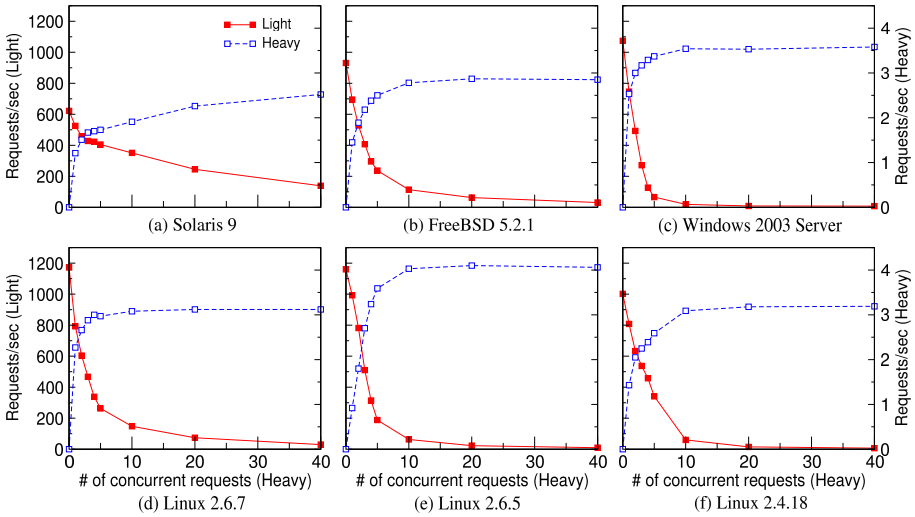


図 3.1: OS 間での各サービスの処理性能の違い (実験 1)

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、1 Gbps 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 1 Gbps のスイッチで接続されていた。

3.1.2　実験結果

実験 1 図 3.1 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

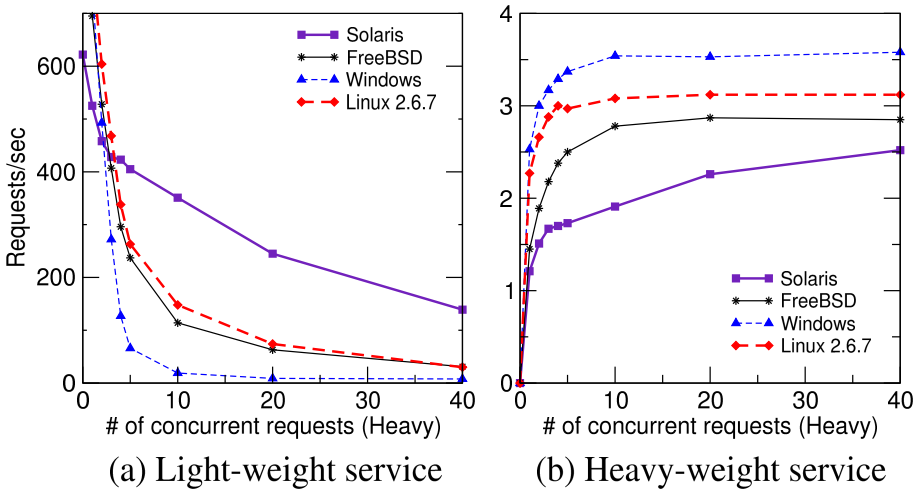


図 3.2: 異なる OS の各サービスの処理性能 (実験 1)

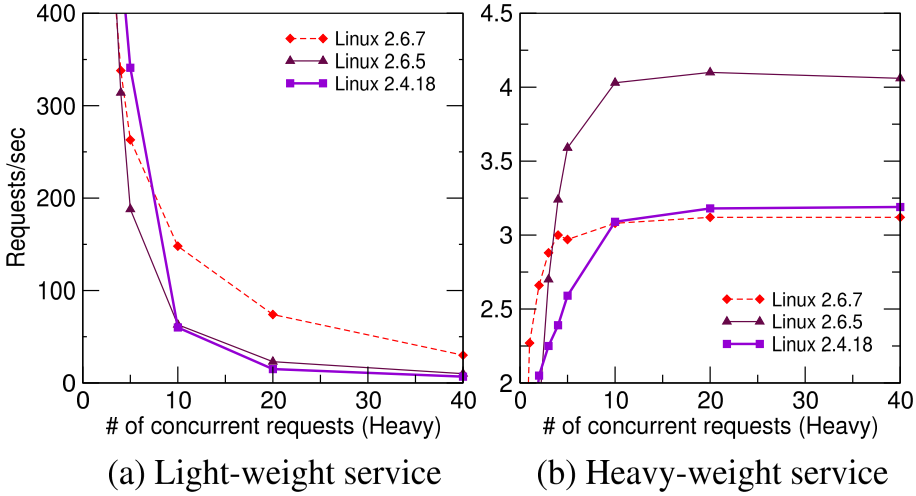
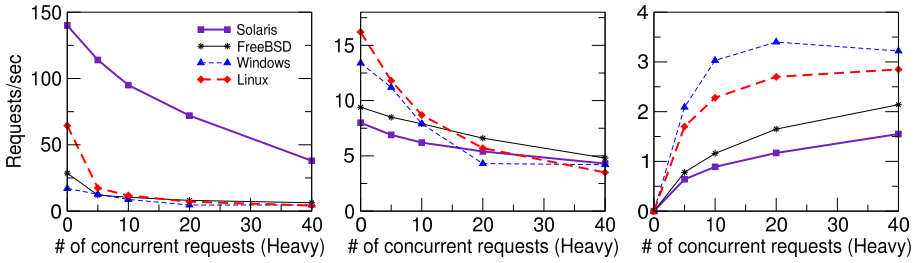


図 3.3: 異なるバージョンでの各サービスの処理性能 (実験 1)



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

図 3.4: 3 種類のサービスでの各サービスの処理性能 (実験 2)

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 3.2 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 の違いによって各ワークロードでの処理性能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 3.1 の (d)〜(f) である。また、図 3.3 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各サービスの処理性能が異なることが分かる。過負荷時において、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 3.4 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各サービス間での処理性能の関係は複雑になるが、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

表 3.1: netstat による統計情報

	Solaris		Linux		FreeBSD		Windows	
サービス	軽い	重い	軽い	重い	軽い	重い	軽い	重い
接続数	28171	669	1977	834	5945	778	3570	958
接続失敗回数	0	0	0	0	0	0	0	0
受信セグメント数	140996	3613	10011	4372	29317	4111	17602	5049
送信セグメント数	141196	3723	10080	4485	29981	4218	18145	5208
再送回数	0	7	0	2	2	4	9	14

分かる。

3.2　OS 間の相違の要因分析

3.1 では、複数のサービスを提供するアプリケーションサーバを複数の OS 上で動作させることで、OS が与える各サービスの処理性能への影響を確認した。アプリケーションサーバ上で動作するサービスの処理性能に影響を与える要素を追求するためには、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、Solaris 9、Linux 2.6.6、FreeBSD 5.2.1、Windows 2003 Server Enterprise Edition において、ネットワーク性能と JVM のガベージコレクションの影響について調べた。

その後、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。この分析は、 3.1 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行った。

3.2.1　実行環境によるネットワーク処理への影響

まず、性能差の要因が各 OS でのネットワーク処理の違いによるか否かを検証するための実験をおこなった。軽いサービスには常時 30 のリクエストを、重いサービスには常時 40 のリクエストを 3 分間投げ続ける。その後、クライアントマシンの側で netstat を用いてパケットの送受信の統計情報を調べた。

得られた統計情報を表 3.1 に示す。どの OS の場合も再送はほとんど起きておらず、コネクションの失敗も起きていない。したがって、OS によるネットワーク処理の違いが、性能差の原因であるとは考えにくい。

表 3.2: 軽いサービスと重いサービスにおける GC の挙動

	軽いサービス			
	GC の回数	減少ヒープ量の和 (GB)	停止時間 (秒)	処理数
Solaris	1333	2.73	7.3	37906
Linux	353	0.51	1.3	70045
FreeBSD	236	0.36	2.2	52633
Windows	936	1.65	5.0	52072

	重いサービス			
	GC の回数	減少ヒープ量の和 (GB)	停止時間 (秒)	処理数
Solaris	5044	19.5	33.7	742
Linux	5946	19.8	27.3	765
FreeBSD	4915	15.7	37.8	603
Windows	10244	21.3	35.3	771

3.2.2　ガベージコレクション

JVM の実装は OS によって異なるので、GC (ガベージコレクション) の実装も OS ごとに大きく異なる可能性がある。そこで GC の性能が全体の性能差に与える影響を調べるための実験をおこなった。実験では、軽いサービスにのみ 80 クライアントから 1 分間常時リクエストを投げつづけた場合と、重いサービスにのみ 80 クライアントから 4 分間常時リクエストを投げた場合について、java の logcc オプションを用いて GC の情報を集め、解析を行った。

解析結果は表 3.2 のようになった。GC の頻度とリクエストの処理数の関係は、OS 毎にかなり違いがあり、一貫性はみられない。重いサービスへのリクエストの増加により軽いサービスへのリクエストの処理数が急激に減少してしまう Linux, FreeBSD,Windows で共通した傾向が見られないので、GC の実装の違いが、OS 間の性能差に大きな影響を与えているとは考えにくい。

3.2.3　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

表 3.3: 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

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 3.3 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、この違いがサービスの処理性能の相違の原因と考えられる。

3.2.4　ロック待ち時間の分析

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。

表 3.4: 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

リクエスト処理中の要因

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 3.4 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。

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 3.5 に示すように Linux の方が 2.9 倍長い。この待ち時間は図 3.5 に示すように、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。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

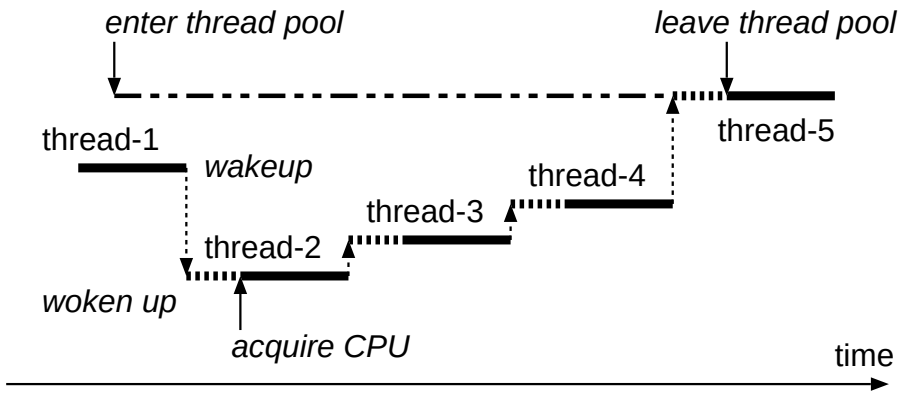


図 3.5: スレッドスケジューリング

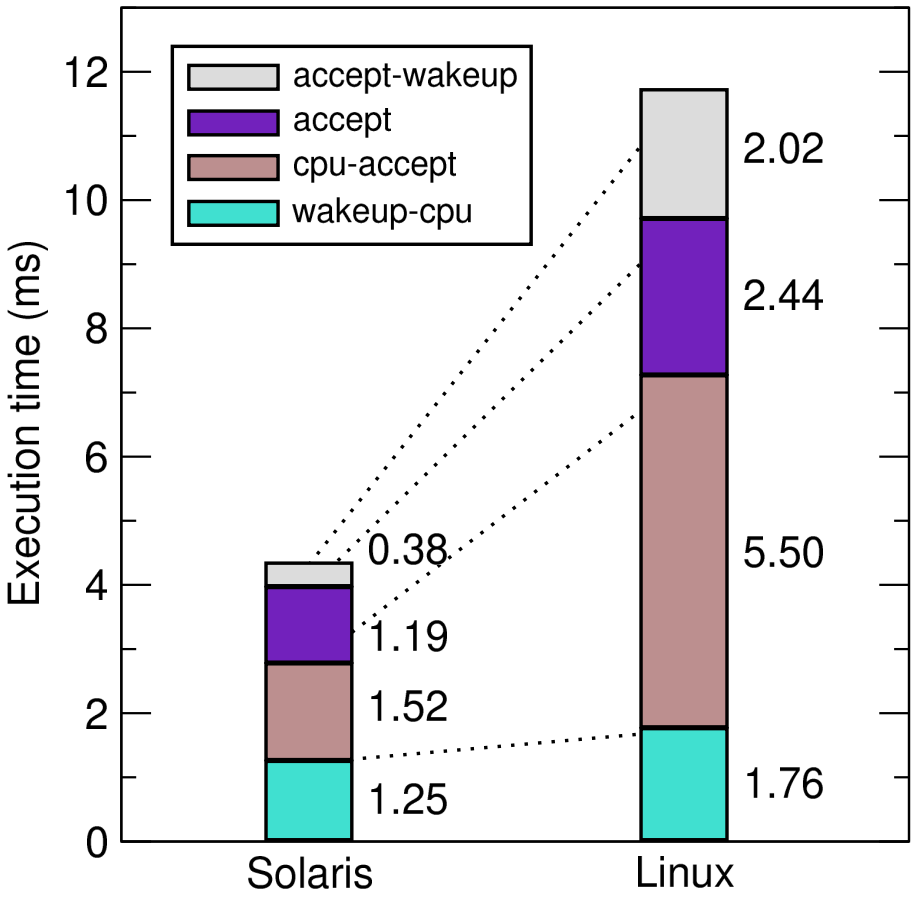
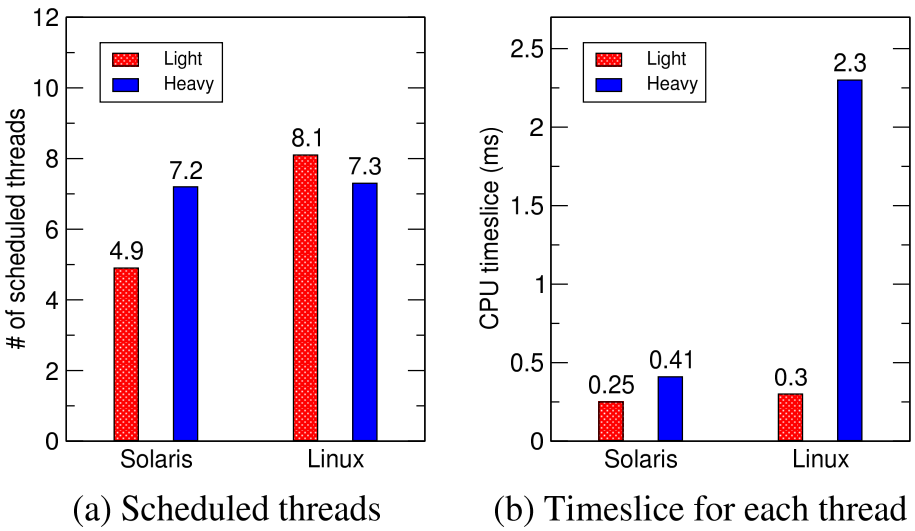


図 3.6: 各スレッドの実行にかかる時間の内訳

表 3.5: 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



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

図 3.7: 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

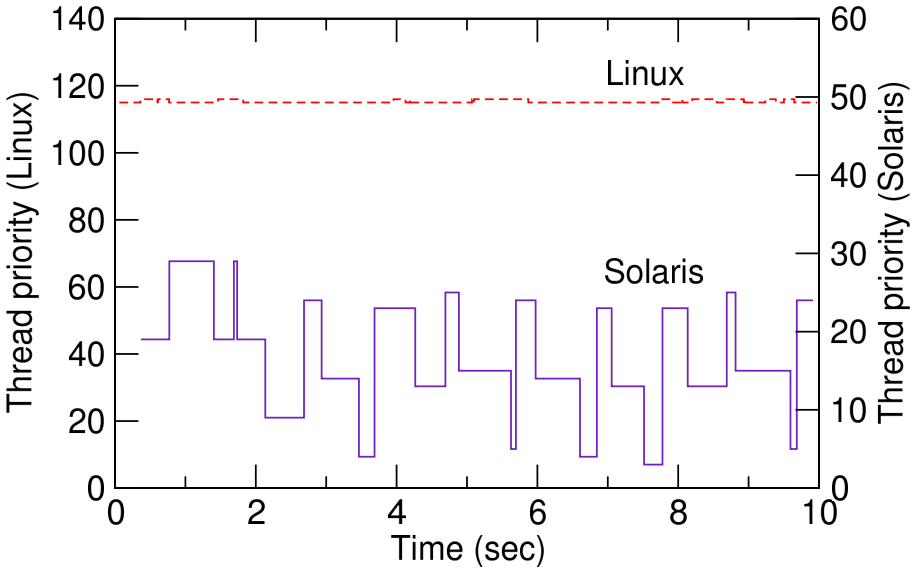


図 3.8: スレッド優先度の変化

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 3.6 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、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 3.7 (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 3.7 (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。

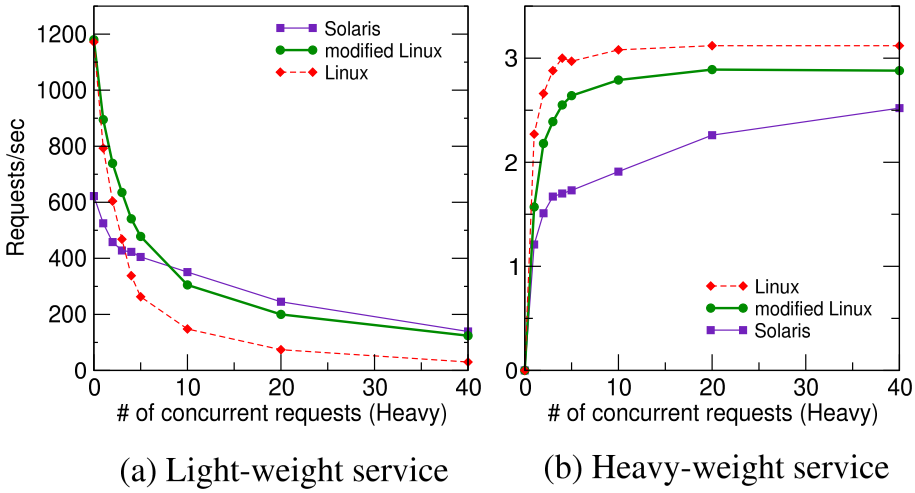


図 3.9: タイムスライスを短くした Linux における各サービスの処理性能 (実験 1)

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 3.8 に示されるように、Solaris のスレッド優先度は頻繁に変動し、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。

3.2.5　タイムスライスの変更による影響

タイムスライスを変更することで Linux において各サービスの処理性能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 3.1 の実験 1 と同様の実験を行った結果を図 3.9 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 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

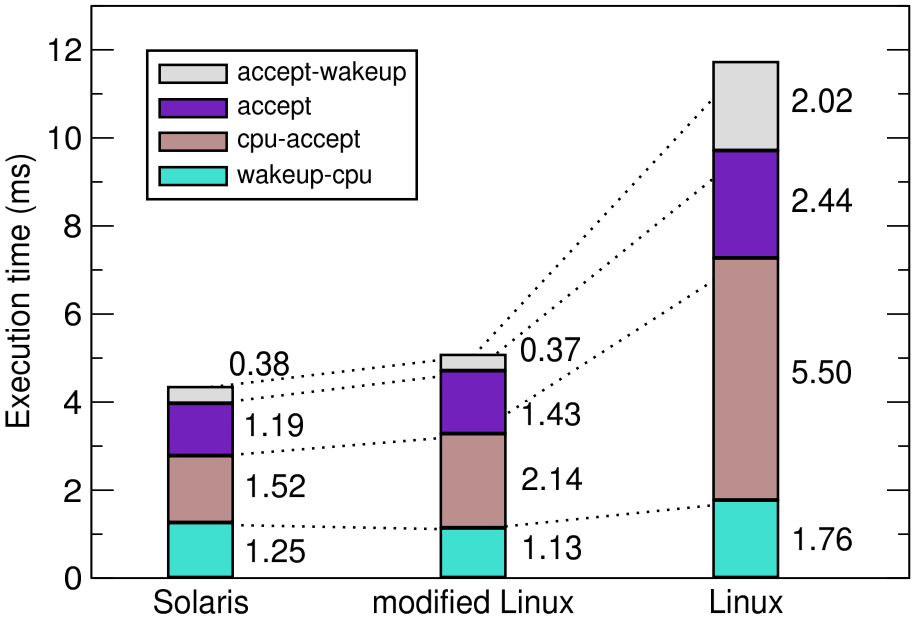


図 3.10: タイムスライスを短くした Linux で各スレッドの実行にかかる時間の内訳

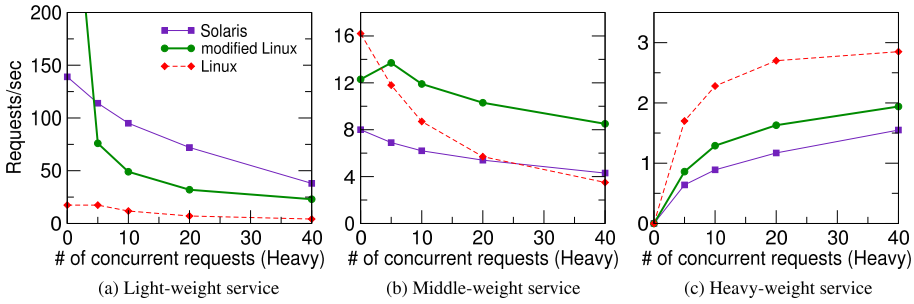


図 3.11: タイムスライスを短くした Linux における各サービスの処理性能 (実験 2)

訳は図 3.10 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 3.1 の実験 2 と同様の実験を行った結果を図 3.11 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、各サービスの処理性能を変更でき、処理性能に影響を与える主な要因であることが分かった。

3.3　要因分析のまとめ

ウェブアプリケーションサーバ上で提供されるサービスの処理性能に影響を与える要因について調べた。OS 以外は同じハードウェア、アプリケーション、ワークロードを用いて、Solaris と Linux でアプリケーションサーバの挙動を比較分析した。分析の結果、OS のスケジューリングポリシー及びスレッドライブラリの実装の違いが主に影響していることが分かった。

第4章　AOP を用いたアプリケーションレベルスケジューリング

この章では AOP を用いることにより、アプリケーション自体がスケジューリングを行うように QoS 制御を追加できるアプリケーションレベルスケジューリングを提案する。

4.1　アプリケーションレベルスケジューリング

アプリケーションレベルスケジューリングでは、アスペクトを用いて J2EE アプリケーションにスケジューラを結合し、アプリケーション側からスケジューラクラスを呼び出させるようにすることでスケジューリングを実現する。スケジューラクラスには、適用するスケジューリングポリシーを記述する。例えば、時間的制約のある処理を優先的に処理するポリシーを記述することにより、過負荷時でも影響を受けずに処理を行うことができる。このスケジューラクラスは、J2EE アプリケーションが J2EE サーバにロードされる際にアスペクト指向システムを用いて J2EE アプリケーションと結合される。

アプリケーションレベルスケジューリングはアプリケーション以外には変更を加えず、アプリケーションのコードを手で書き換える必要もない。そのため、アプリケーションのロジックを壊す危険性はない。また、アスペクトを用いることでアプリケーションからスケジューラに処理内容とスレッドの対応づけを通知することができるため、OS での制御とは異なり、サービスを区別して制御を行うことができる。さらに、アスペクトを介してアプリケーション内部の様々な位置からスケジューラを呼び出させるため、リクエスト処理の開始時・終了時だけでなく、よりきめ細かい制御を行うことができる。

さらに、スケジューラを再利用することができるという利点もある。アプリケーションレベルスケジューリングではアスペクトによってスケジューラとアプリケーションの依存関係が弱められているため、別のアプリケー

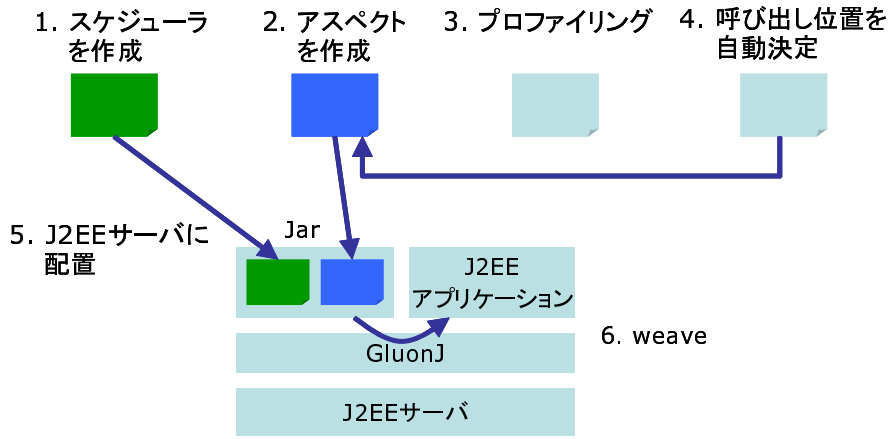


図 4.1: アプリケーションレベルスケジューリングの構成

ションに対しても、アスペクトだけを作成すれば同じスケジューラを使うことが可能である。

4.2　スケジューラ

スケジューラでは、スケジューリング対象であるアプリケーションのスレッドを管理し、アスペクトを通してアプリケーションから呼び出されることでスケジューリングを実現する。一般的なスケジューラでは以下のような処理を記述する必要があるが、これ以外の処理を記述することも可能である。

- アプリケーションスレッドの登録・削除
- スレッド実行の制御
- スケジューリングの変更要求

アプリケーションスレッドの登録・削除を行うことで、スレッドプールが使われていている場合でもスケジューラが処理の開始、終了を知ることができる。スケジューラはアプリケーションの各スレッドに対してスケジューリングに用いられる実行可能フラグを管理する。

スレッド実行の制御については、アプリケーションのスレッドが自身の実行を制御するために、スケジューラを呼び出す形で実現する。スケジューラを呼び出した際、スレッドの実行可能フラグを調べ、実行可能フラグが立っていなければ、その地点で一時停止する。このような自発的なスケジューリングを行うのは、Java がスレッドを別のスレッドから

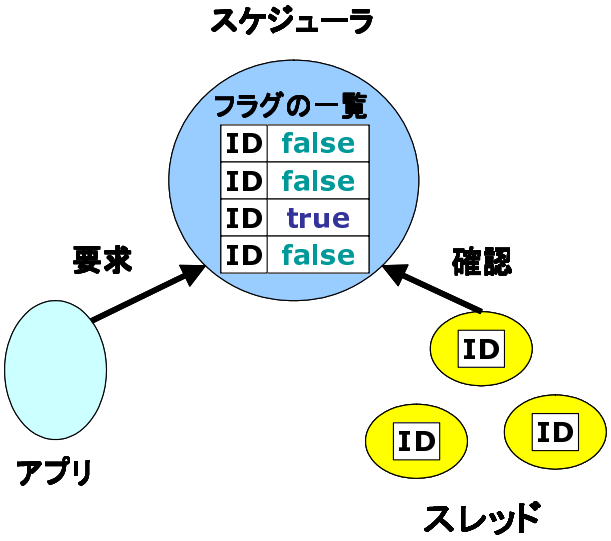


図 4.2: スケジューラ

suspend したり resume したりすることを推奨していないためである。もし、suspend されたスレッドがシステムクラスのモニタをロックしていた場合、デッドロックが発生する危険性がある。アプリケーションレベルスケジューリングではシステムクラス内からはスケジューラを呼ばせないようにすることでこの問題を回避している。

スケジューリングの変更要求には、適用したいスケジューリングアルゴリズムを記述する。アルゴリズムに従って、一時的に停止させたいスレッドの実行可能フラグをクリアすれば、そのスレッドがスケジューラを呼び出した時点で自発的に停止する。スレッド実行を再開させる時には実行可能フラグを立ててから起こしてやればよい。

4.3　アスペクト

アスペクトにはスケジューラを呼び出す位置と、そこで呼び出されるスケジューラクラスのメソッドを記述する。

4.3.1　記述例

以下では、アスペクト指向システムとして用いた GluonJ におけるアスペクトの記述例を示す。GluonJ は AspectJ のポイントカット・アドバイスモデルを利用した Java 用の AOP 言語であり、glue と呼ばれるアスペ

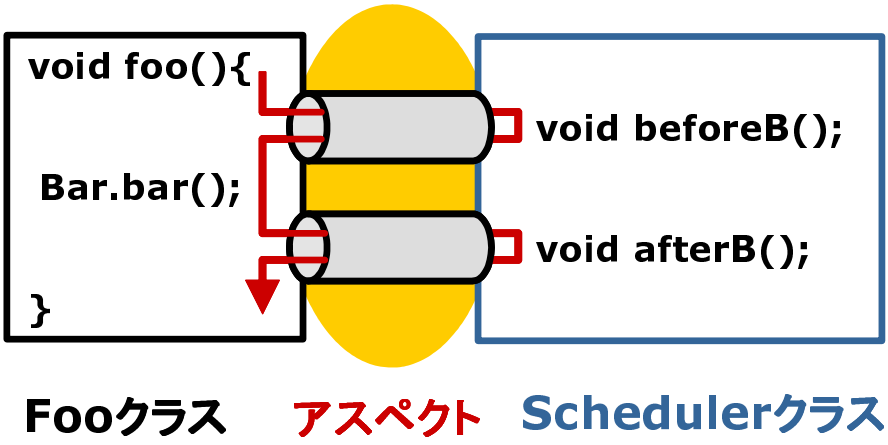


図 4.3: アスペクトによる結合

クトを用いてコンポーネント間の結びつきを記述する。J2EE サーバにデプロイされているアプリケーションのクラスが実際にロードされる際に、アスペクトが適用 (weave) される。

図 4.4 の XML 記述が glue コード、つまりアスペクトである。pointcut-decl タグの中ではコード中の位置を指定してその位置に名前をつけることができる。指定方法としては withincode で呼び出し元のメソッドを、call で呼び出し先のメソッドを指定し、ANDAND で論理積を取る。この例では、Foo.foo() の中から Bar.bar() を呼び出している位置を指定している。また、behavior タグの中では指定した位置でどのようなコードを実行するかを指定することができる。pointcut タグで位置を指定し、around タグでその位置で実行するコードを指定する。around タグの中では$proceed() を使って指定した位置にあるコードを実行できる。$$で引数を、$_で返り値を表す。この例では、まず、Scheduler.beforeB() を呼び出し、指定した位置のコードである Bar.bar() を実行した後で、Scheduler.afterB() を呼び出している。これにより、Foo.foo() から Bar.bar() を呼び出す前後に Scheduler クラスの beforeB() と afterB() を呼び出すことができる。

4.3.2　定期的に呼び出す位置の自動生成

アプリケーションレベルスケジューリングでは短い間隔でスケジューラの実行制御メソッドを呼び出させることで、アプリケーションの処理の途中できめ細かく制御することができる。しかし、無数にある呼び出し位置の候補から人手で選び出すのは現実的ではない。そこで、プロファイリング情報から実行制御メソッドを呼び出す位置を自動的に決定できるように

```
<glue>
  <pointcut-decl>
    <name> point </name>
    <pointcut>
      withincode(void Foo.foo())
        ANDAND call(void Bar.bar())
    </pointcut>
  </pointcut-decl>

  <behavior>
    <pointcut> point </pointcut>
    <around>
      Scheduler.beforeB();
      $_ = $proceed($$);
      Scheduler.afterB();
    </around>
  </behavior>
</glue>
```

図 4.4: GluonJ によるアスペクトの例

した。呼び出す位置の候補は制御対象となる処理の実行中に呼び出されるすべてのメソッドで、そのメソッドの呼び出し位置は withincode と call を用いて呼び出し元メソッドと呼び出し先メソッドの組で特定する。

まず、制御対象となる処理の途中で呼び出されるすべてのメソッドで、呼び出し時刻のログを取る。ログを取るためのコードはアスペクトを利用して候補となる箇所に自動で埋め込む。ログを取る際には、制御対象の処理に対するリクエストを 1 つだけ処理するようにした。複数のリクエストが処理されている状態だと、資源の競合やスレッドの切り替わりなどの外乱の影響が大きくなるため、実行制御メソッドの呼び出し位置を等間隔になるように選んでも、ばらつきがひどくなるためである。このログに基づいて等間隔で実行制御メソッドが呼び出されるように選んだとしても、状況によっては定期的に呼び出せるとは限らないが、ある程度一定の間隔で実行制御メソッドが呼ばれることを期待できる。

以下、このログを基に、T (ms) 間隔で実行制御メソッドを呼び出す場合の選択方法を説明する。メソッド呼び出しが複数回出現する場合は、1 つのメソッド呼び出し位置の指定で複数の候補が選択されてしまうため扱いにくい。そこで、出現頻度の低いメソッド呼び出しを優先的に選ぶことにした。

1. まず、出現頻度が 1 回のメソッド呼び出しの中から選択する。選択さ

れるのは、ログの処理開始時刻を 0 とした場合に、時刻 $(m-1/2)T$ 〜$(m+1/2)T$ の中で $m \times T(m = 1, 2, \ldots)$ に最も近いメソッド呼び出しとする。

2. 出現頻度が 1 回のメソッド呼び出しが存在しなかった区間 $(k-1/2)T$ 〜$(k+1/2)T$ で選択を行う。実行制御メソッドの呼び出しが定期的でかつ少なくなるような箇所を選択する。方法としては、以下の加点を行い、一番点数の高いものを選択する。対象メソッドが出現する区間にすでに選択されているメソッド呼び出しが存在しなければ、その数だけ加点する。さらに、出現する区間すべてが現在対象となっているメソッド呼び出しの出現区間に包含されているものがあれば、その出現回数だけ加点する。このようにして、実行制御メソッドの呼び出し回数をどれだけ減らすことができるかを評価していく。そして一番点数が高いものを選んだ後で、すでに選ばれているものの内、出現区間がすべて包含されていたものは、候補から外す。

3. すべての区間 $(k-1/2)T$ 〜 $(k+1/2)T$ で同様の選択を行い、さらに出現頻度が N 回のメソッド呼び出しまで同様にして選択していく。

4.4　Case study 1

この節では、アプリケーションレベルスケジューリングの対象とする、単純な J2EE アプリケーションについて説明する。

4.4.1　アプリケーション

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が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 [20] 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が提供されているが、ここでは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、重いサービス、軽いサービスの 2 種類を用意する。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 200 回繰り返す。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 35 番目のフィボナッチ数の計算を行う。

この 2 種類のサービスを J2EE アプリケーションとして用意する。

4.4.2　追加した QoS 制御

様々な種類のサービスを提供している場合、適用する QoS 制御も多数考えられる。ここでは、荷物のマッチング処理に対応する重いサービスの影響で、ログインなどの軽いサービスに時間がかかってしまうことを問題とする。この問題に対処するため、アプリケーションレベルスケジューリングを用いて、ログイン処理である軽いサービスを優先するという QoS 制御を追加する。このような QoS 制御を実現するために、軽いサービスを行なっている間は重いサービスの並列処理数を下げるというスケジューリングポリシーを適用する。頻繁に並列処理数の変更要求が発生するのを防ぐため、軽いサービスの処理を終えてから 2 秒間経過するまでに軽いサービスにリクエストがこなければ、並列処理数を制限を解除することにする。作成したアスペクトを図 4.5 と図 4.6 に、スケジューラクラスを図 4.7 に示す。

図 4.5 はアスペクトからスケジューラクラスを呼び出す位置の記述を抜き出したものであり、図 4.11 はアスペクトからどのようにスケジューラクラスを呼び出すかという記述を抜き出したものである。PriorityPolicy クラスでは実際のスケジューリングアルゴリズムを記述した ThreadController クラスを呼び出す。

4.5　Case study 2

この節では、アプリケーションレベルスケジューリングの対象とする J2EE アプリケーションとして用いた河川水位監視システムについて説明する。

```
<pointcut-decl>
  <name> lowImportant </name>
  <pointcut>
   execution(void DomSearch.search())
  </pointcut>
</pointcut-decl>
<pointcut-decl>
  <name> highImportant </name>
  <pointcut>
   execution(void FibonacciBean.fibonacci())
  </pointcut>
</pointcut-decl>
<pointcut-decl>
  <name> checkpoint </name>
  <pointcut>
    (withincode(void DomSearch.search())
      ANDAND call(void DomSearch.search(org.w3c.dom.Node)))
  </pointcut>
</pointcut-decl>
```

図 4.5: アスペクト (pointcut-decl 部)

4.5.1　河川水位監視システム

このシステムは現実に運用されているシステムに近づけるために、アプリケーションレベルスケジューリングの詳細を隠して SMG 社に開発してもらったものである。このシステムは全国の主な河川に設置された水位計からの情報を集めるサーバとその情報をユーザに提供する J2EE サーバからなる。前者のサーバは全国各地に置かれ、水位情報をウェブサービスとして提供する。現在のところ、図 4.8 のように 1 台のサーバでシミュレータを使って実現している。

一方、J2EE サーバは JBoss [9] を用いて構築されている。このサーバは各地点に置かれたサーバからウェブサービスを使って定期的に水位情報を取得する。取得した水位情報はデータベースに格納され、最新の水位情報はメモリ上にも置かれる。水位情報は刻一刻と変化するするため、取得を開始してから一定時間以内に全ての地点の水位情報を取得できなかった場合、取得できなかった地点のデータは欠測として扱われる。欠測が発生した場合、全ての地点の水位情報を取得し終わってから次の定期的な取得を行う。

このサーバはユーザが水位情報を閲覧できるように 2 種類の Servlet を提供している。1 つは現在の水位情報を閲覧するための Servlet であり、

```
<behavior>
  <pointcut> lowInportant </pointcut>
  <around>
    SimplePriorityPolicy.beforeLowPriority();
    $_ = proceed($$);
    SimplePriorityPolicy.afterLowPriority();
  </around>
</behavior>
<behavior>
  <pointcut> highImportant </pointcut>
  <around>
    SimplePrioirtyPolicy.beforeHighPriority();
    $_ = proceed($$);
    SimplePriorityPolicy.afterHighPriority();
  </around>
</behavior>
<behavior>
  <pointcut> checkpoint </pointcut>
  <before> SimplePrioirtyPolicy.checkpoint();
   </before>
</behavior>
```

図 4.6: アスペクト (behavior 部)

```
public class SimplePriorityPolicy {
 public static void beforeLowPriority() {
  controller.enter(Thread.currentThread());
 }
 public static void afterLowPriority() {
  controller.exit(Thread.currentThread());
 }
 public static void beforeHighPriority() {
  controller.schedule(1);
 }
 public static void afterHighPriority() {
  controller.schedule(80);
 }
 public static void checkpoint() {
  controller.checkpoint(
                  Thread.currentThread());
 }
}
```

図 4.7: スケジューラクラス (抜粋)

メモリ上に置かれた水位情報から図 4.9 のように各地点の水位を表示する。もう 1 つは過去の水位情報をグラフ化して表示する Servlet であり、データベースにアクセスして指定された時間分の過去の水位情報を取得し、グラフの生成処理を行って、作成したグラフ画像を表示する。

この河川水位監視システムでは QoS 制御を何も行っていなかったが、クライアント数が少ない時には欠測が起こらず仕様通りに動作していた。しかし、クライアント数を増やすと欠測が起こるようになった。これはグラフ表示用 Servlet がグラフ生成処理を行う際にサーバの CPU 資源をかなり長時間使い、定期的に水位情報を取得する処理が滞ったためであった。欠測が発生すると、その時間帯の水位情報を提供できないことになるため、QoS 制御の追加が必要とされた。

4.5.2　追加した QoS 制御

過負荷時でも水位情報の欠測が発生しないようにするために、アプリケーションレベルスケジューリングを用いて、定期的な水位情報の取得処理を優先する QoS 制御を追加した。このような QoS 制御を実現するために、重要度の高い処理を行っている間は重要度の低い処理の並列処理数を下げるというスケジューリングポリシを適用した。河川水位監視シス

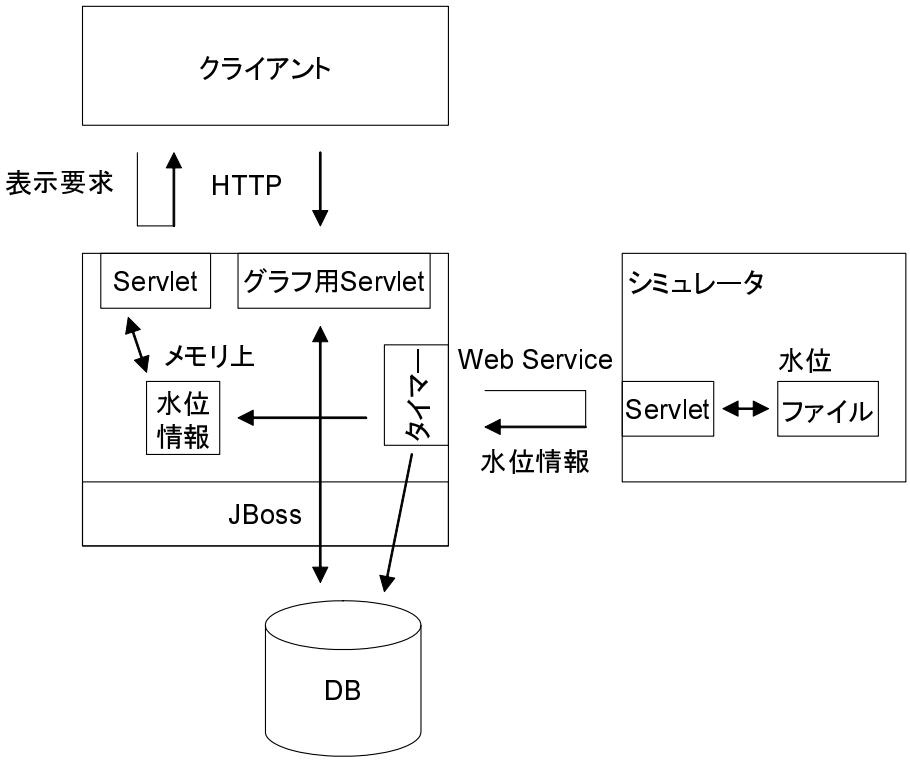


図 4.8: 河川水位監視システムの仕組み

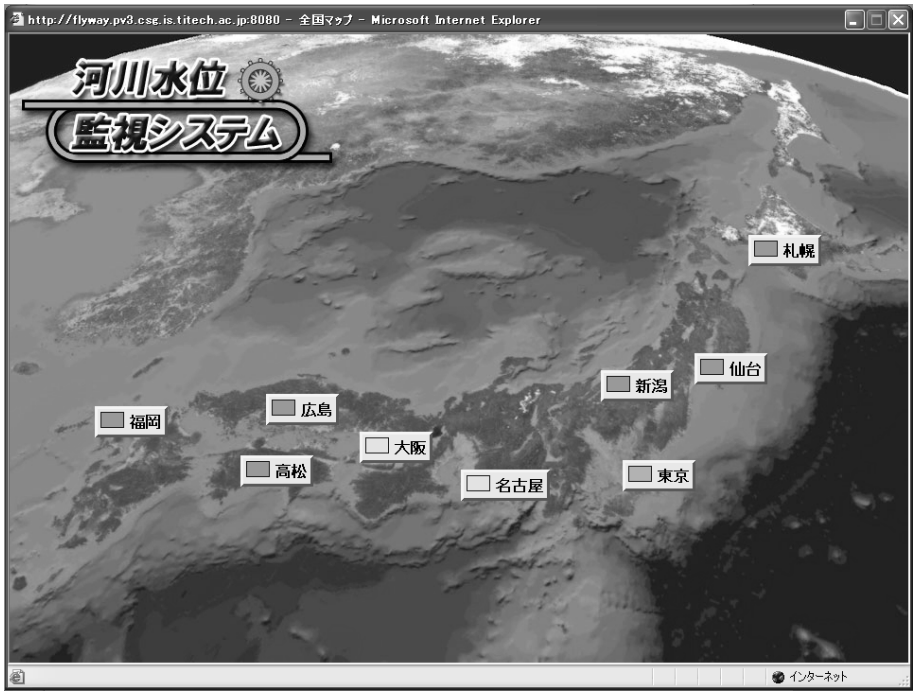


図 4.9: 水位情報の閲覧ページ

テムでは、定期的な水位情報の取得処理が重要度の高い処理になり、グラフ生成処理が重要度の低い処理となる。作成したアスペクトを図 4.10 と図 4.11 に、スケジューラクラスを図 4.12 に示す。

図 4.10 はアスペクトからスケジューラクラスを呼び出す位置の記述を抜き出したものである。lowImportant、highImportant は execution でメソッドが実行される位置を指定することで、それぞれ重要度の低い処理を開始するメソッドと重要度の高い処理を開始するメソッドの位置を指定している。また、checkpoint は 3.3.2 のアルゴリズムを使って重要度の低い処理の中で定期的に呼び出される位置を OROR で論理和を取りながら列挙して指定している。

図 4.11 はアスペクトからどのようにスケジューラクラスを呼び出すかという記述を抜き出したものである。例えば、lowImportant で指定された位置では、その位置のコードを実行する前後で beforeLowPriority メソッドおよび afterLowPriority メソッドを実行する。また、checkpoint で指定された位置では、before タグを使うことで、その位置のコードを実行する前に checkpoint メソッドを実行する。アスペクトから呼び出されるこれらのメソッドは図 4.12 のスケジューラクラスに記述されている。

PriorityPolicy クラスでは実際のスケジューリングアルゴリズムを記述し

```
<pointcut-decl>
  <name> lowImportant </name>
  <pointcut>
   execution(void PlaceChartCreatePseudActionImpl.execute(..))
  </pointcut>
</pointcut-decl>
<pointcut-decl>
  <name> highImportant </name>
  <pointcut>
   execution(void CollectorImpl.doObtain())
  </pointcut>
</pointcut-decl>
<pointcut-decl>
  <name> checkpoint </name>
  <pointcut>
    (withincode(Range CategoryPlot.getDataRange(ValueAxis))
      ANDAND call(Range Range.combine(Range,Range)))
    OROR
      :
  </pointcut>
</pointcut-decl>
```

図 4.10: アスペクト (pointcut-decl 部)

た ThreadController クラスを呼び出す。このクラスの enter メソッドはスレッドの登録を行い、実行中のスレッド数が最大値を超えていなければスレッドに割り当てた実行可能フラグを true に設定する。exit メソッドはスレッドの登録削除を行う。schedule メソッドはスケジューリングを実行し、実行中のスレッド数が指定された値になるように、各スレッドの実行可能フラグを調整する。このメソッドで重要度の低い処理の並列処理数を制御している。checkpoint メソッドはスレッドの実行可能フラグをチェックして、false ならば wait メソッドを読んで一時停止する。一時停止したスレッドは実行可能になった時に notify メソッドによって再開される。このメソッドは重要度の低い処理から定期的に呼ばれるため、きめ細かいスケジューリングが可能となる。

```
<behavior>
  <pointcut> lowInportant </pointcut>
  <around>
    PriorityPolicy.beforeLowPriority();
    $_ = proceed($$);
    PriorityPolicy.afterLowPriority();
  </around>
</behavior>
<behavior>
  <pointcut> highImportant </pointcut>
  <around>
    PrioirtyPolicy.beforeHighPriority();
    $_ = proceed($$);
    PriorityPolicy.afterHighPriority();
  </around>
</behavior>
<behavior>
  <pointcut> checkpoint </pointcut>
  <before> PrioirtyPolicy.checkpoint();
   </before>
</behavior>
```

図 4.11: アスペクト (behavior 部)

```
public class PriorityPolicy {
 public static void beforeLowPriority() {
  controller.enter(Thread.currentThread());
 }
 public static void afterLowPriority() {
  controller.exit(Thread.currentThread());
 }
 public static void beforeHighPriority() {
  controller.schedule(1);
 }
 public static void afterHighPriority() {
  controller.schedule(40);
 }
 public static void checkpoint() {
  controller.checkpoint(
                  Thread.currentThread());
 }
}
```

図 4.12: スケジューラクラス (抜粋)

第5章　実験

この章では、 4.4.2 及び 4.5.2 のスケジューラクラスとアスペクトそれぞれの J2EE アプリケーションに適用し、その効果を確かめる実験を行った。サーバホストには、Xeon 3.06 GHz の CPU を 2 つ、2 GB のメモリ、1 Gbps の NIC を備えた Sun Fire V60x を、4.4 では 1 台、 4.5 では 2 台用いた。OS は Linux 2.6.8 であり、使用した JBoss のバージョンは 4.0.2、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、AthlonXP-M 1.53GHz の CPU、1 GB のメモリ、1 Gbps の NIC を備えた Sun Fire B100x を、4.4 では 2 台、4.5 では 1 台用いた。使用した OS は Solaris 9 であった。これらのホストを 1 Gbps のスイッチで接続した。またワークロードの生成には、パフォーマンス測定・負荷テストツールの JMeter [3] を利用した。 4.5 の河川水位監視システムの定期的な水位収集は 15 秒間隔で行われる。比較のため、制御をしない場合およびリクエスト処理の開始時のみで制御する従来のアドミッションコントロールを適用した場合でも実験を行った。

5.1　Case study 1

4.4 について、4.4.2 のスケジューラクラスとアスペクトを適用し、その効果を調べた。

5.1.1　アプリケーションレベルスケジューリングの効果

スケジューラの効果をみるために、重いサービスに対して、常に 80 のリクエストを送っている状況で、軽いサービスに対し 100 リクエストを一斉に送り、その総処理を計測した。重いサービスの並列処理数は、軽いサービスが行なわれている間は 1 に、そうでない場合は 80 に制限した。測定結果は図 5.1 に示す。縦軸は軽いサービスに対するリクエストがすべて終わるまでにかかった時間である。アドミッションコントロール (AC) を用いることにより、何も制御しない場合 (NC) に比べ、短時間で処理できるているが、アプリケーションレベルスケジューリング (ALS) を適用することにより、さらに短時間で処理されている。

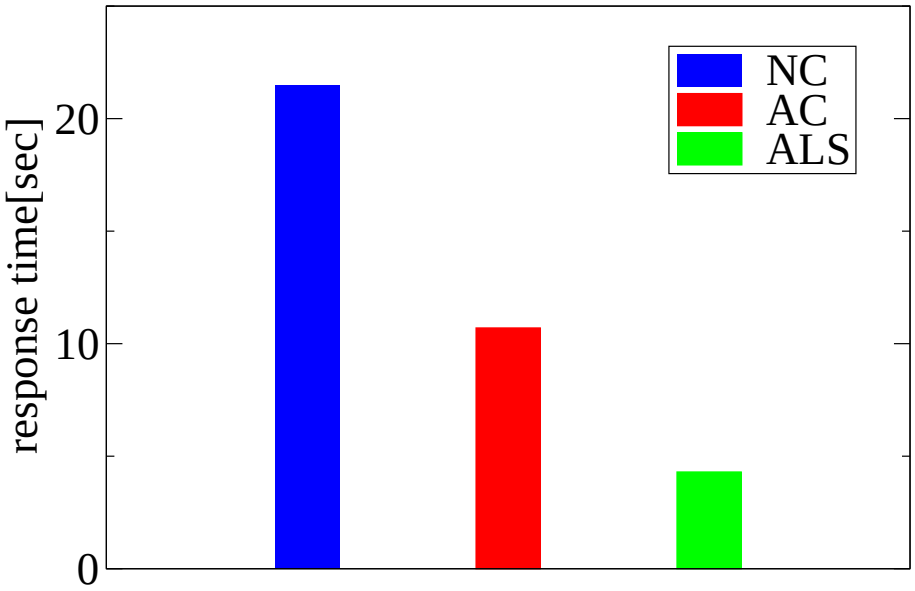


図 5.1: 軽いサービスの総処理時間

並列処理数の制限がどのように行われたかを確かめるために、重いサービスを行うスレッドのうち、実際に動作しているスレッドの数を調べた。図 5.2 と図 5.3 はその実験結果であり、動作しているスレッド数の変化を示している。アドミッションコントロールを用いた場合 (図 5.3)、軽いサービスが処理し終わる間に実際に動作しているスレッドの数が 1 に抑えられていないが、アプリケーションレベルスケジューリングを用いた場合 (図 5.2) は、短時間で 1 に抑えられている。その結果、アプリケーションレベルスケジューリングでは、短時間で軽いサービスの処理ができている。

5.2　Case study 2

4.5 について、4.5.2 のスケジューラクラスとアスペクトを適用し、その効果を調べた。

5.2.1　アプリケーションレベルスケジューリングの効果

スケジューラの効果をみるために、グラフ生成ページに対して、過去 12 時間分の水位情報のグラフ生成要求を常に 40 送っている過負荷な状況で、

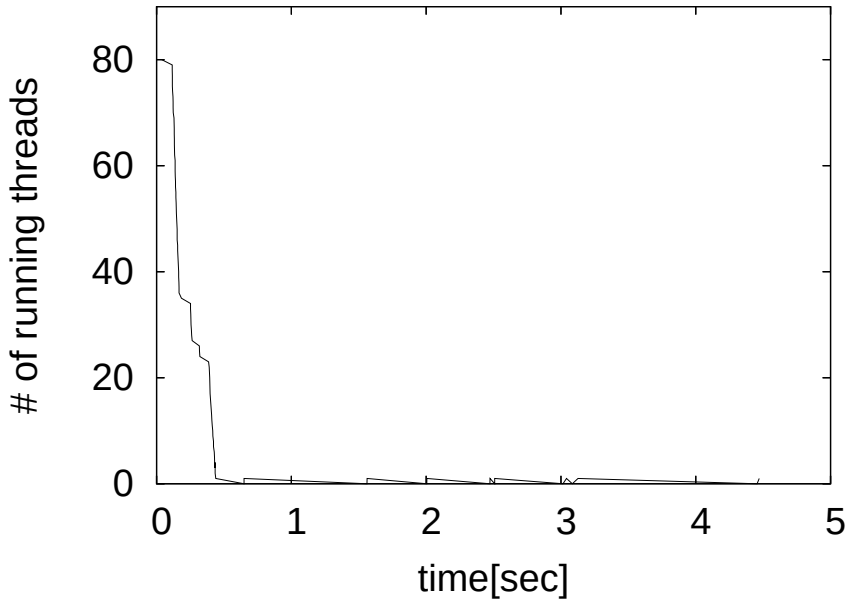


図 5.2: 動作しているスレッド数の変化 (アプリケーションレベルスケジューリング)

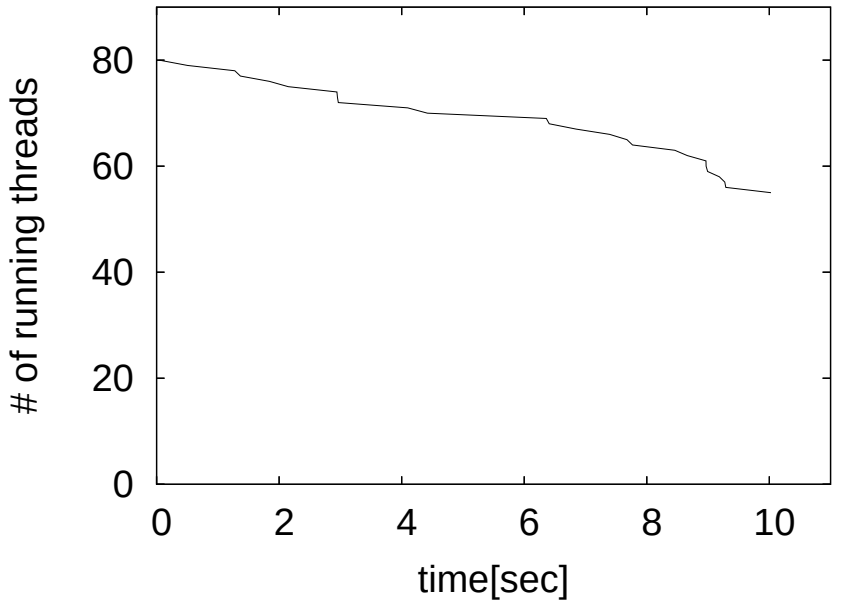


図 5.3: 動作しているスレッド数の変化 (アドミッションコントロール)

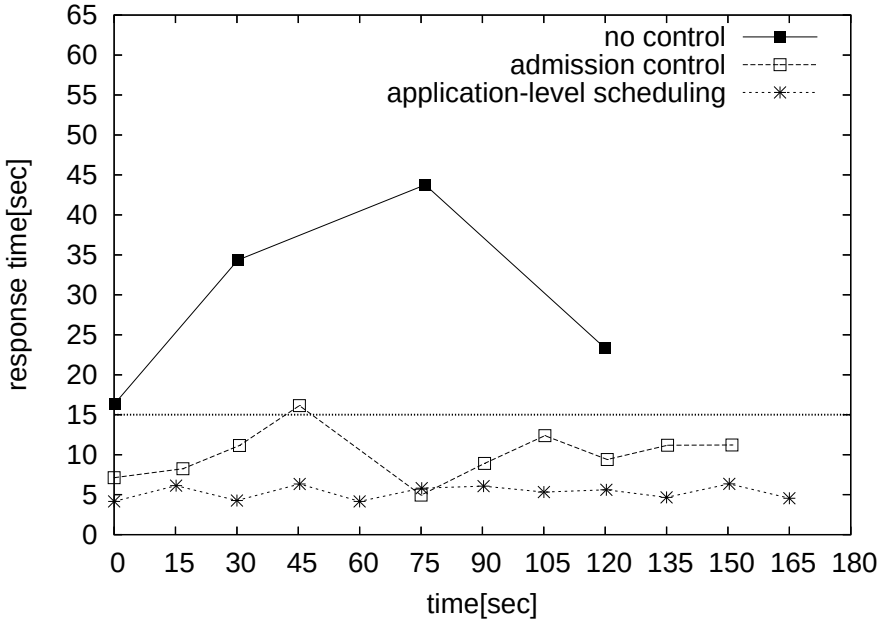


図 5.4: 水位収集の処理時間

定期的な水位収集にかかる時間を計測した。グラフ生成の並列処理数は、水位収集が行われている間は 1 に、そうでない場合は 40 に制限した。

測定結果は図 5.4 に示す。横軸は経過時間、縦軸が水位収集にかかった時間である。何も制御しない場合は、水位収集にかかる時間が 15 秒を大きく超えており、常に欠測が発生している。水位収集が完了するまでは次の水位収集を行わないため、収集できたのは 3 分間でわずか 4 区間であった。一方、提案方式で制御した場合は常に 5 秒程度で水位収集が完了しており、欠測は全く発生しなかった。対して、グラフ生成の開始時だけで並列処理数の制限を行なった場合は、提案方式に比べ処理時間が長く、ばらつきも大きくなっている。さらに 45 秒のところで欠測も発生した。

この時に処理されたグラフ生成ページへのリクエストの総数を時系列に描くと、図 5.5 のようになった。制御しない場合に比べ、従来のアドミッションコントロールでは約 5%、提案方式では約 20% 少なくなっている。これはサーバの資源が水位収集処理に優先的に割り当てられたためである。提案方式での処理数がアドミッションコントロールに比べて少ないのは、並列処理数の制限がより厳密に行われた結果、グラフ生成の処理が抑えられたためだと考えられる。

そこで、並列処理数の制限がどのように行われたかを確かめるために、グラフ生成を行うスレッドのうち、実際に動作しているスレッドの数を調

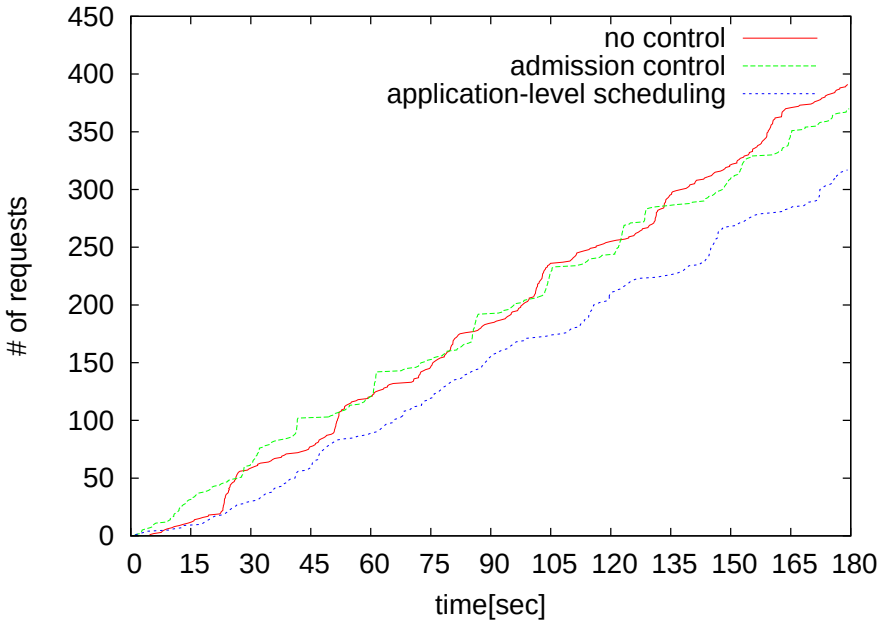


図 5.5: グラフ生成のリクエスト処理数

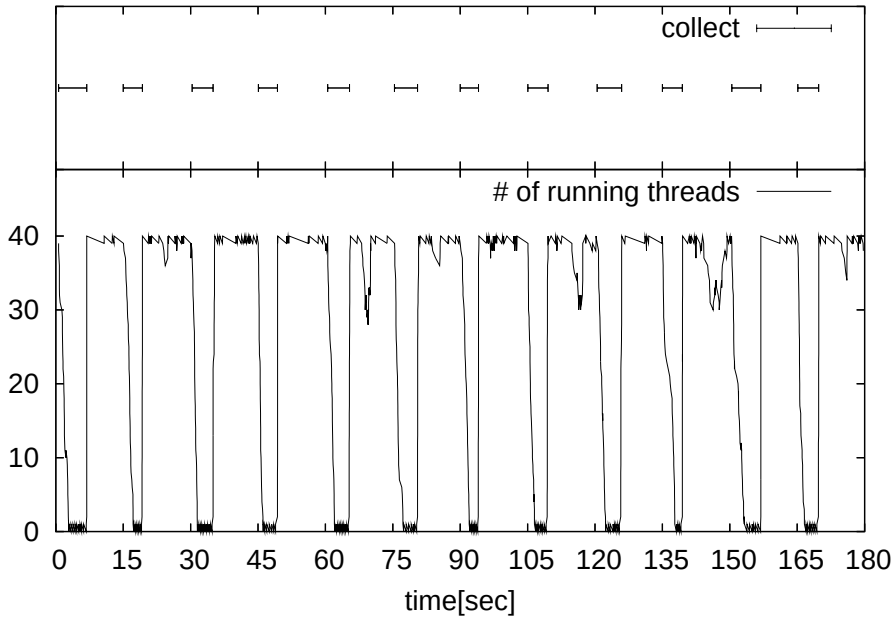


図 5.6: 動作しているスレッド数の変化 (アプリケーションレベルスケジューリング)

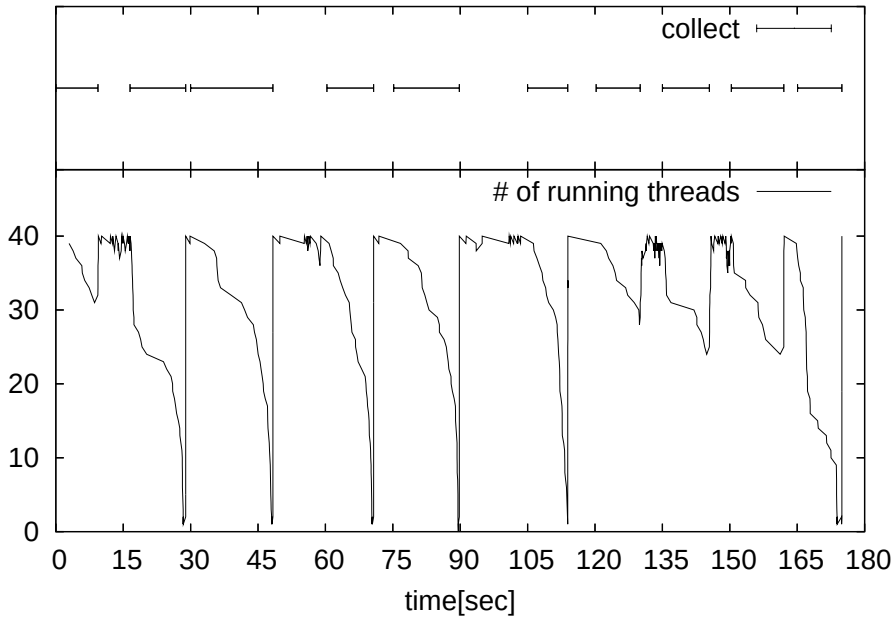


図 5.7: 動作しているスレッド数の変化 (アドミッションコントロール)

表 5.1: グラフ生成に対するリクエストの処理数

制御なし	AC	提案手法
944	905	890

べた。図 5.6 と図 5.7 はその実験結果であり、上のグラフは水位収集が行われていた区間、下のグラフは動作しているスレッド数の変化を示している。提案方式 (図 5.6) では、15 秒間隔で水位収集処理が始まるたびに、動作しているスレッド数が短時間で 1 に制限されていることがわかる。それに対し、従来のアドミッションコントロール (図 5.7) では、グラフ生成の開始時でしか処理を一時停止させられないので、水位収集が終わるまでに動作中のスレッド数が 1 になっていないことが多い。また、動作中のスレッド数が 1 になっている場合でも時間がかかっている。このことが水位収集処理を遅らせる原因となっている。この結果より、提案方式では動作中のスレッドを迅速に制御することによって、水位収集を優先して処理することができていることが分かる。

また、提案方式によるオーバーヘッドをみるために、水位収集を行わないように設定して、同様の負荷をかけた。表 5.1 はその実験結果であり、4 分間の間で処理されたグラフ生成処理の数を示している。何も制御しな

表 5.2: 水位収集にかかる時間 (秒)

グラフ区間	要求数	制御なし	AC	提案手法
6 時間	40	50.3	12.3	5.2
12 時間	20	22.5	6.8	4.6
12 時間	40	29.5	10.1	5.3

表 5.3: 並列処理数を抑えるのにかかる時間 (秒)

グラフ時間	要求数	AC	提案手法
6 時間	40	11.0	1.5
12 時間	20	5.1	1.0
12 時間	40	12.0	1.9

い場合に比べて、提案方式を適用した場合の処理数は 6% 程低下しているが、従来のアドミッションコントロールを適用した場合とほとんど違いはない。

5.2.2　負荷の違いによる制御への影響

サーバにかける負荷の違いによる制御への影響をみるために、複数のパターンの負荷を用いた。サーバにかけた負荷は 3 種類で、過去 6 時間分の水位情報のグラフ生成要求を常に 40 送った場合と 12 時間分のグラフ生成要求を常に 20 または 40 送った場合である。提案手法に関しては、6 時間分、12 時間分のそれぞれに対して、4.3.2 に基づいて自動的に選択された位置でスケジューリング制御を行った。これらの異なる 3 種類の負荷を用いてスケジューリングの効果とスケジューラの挙動を調べた。スケジューリングの効果としては、水位収集にかかる時間を計測した。スケジューラの挙動としては、グラフ生成を行うスレッドの中の動作中のものの数が 1 まで抑えられるのにかかる時間を計測した。

この実験を 3 分間行った時の平均値を表 5.2 と表 5.3 に示す。表 5.2 から提案手法はサーバへの負荷が変化しても安定して水位収集処理を終えられていることが分かる。一方、アドミッションコントロール (AC) はリクエストが常に 20 の時は比較的短い時間で水位収集を終えられているが、40 になるとかなり時間がかかるようになっている。また、表 5.3 から、常に投げられるリクエストが増加した際に、アドミッションコントロール (AC) では、並列処理数が 1 に抑えられるまでの時間が大幅に増加しているが、提案方式ではかかる時間の増加が少なく済むことが分かる。

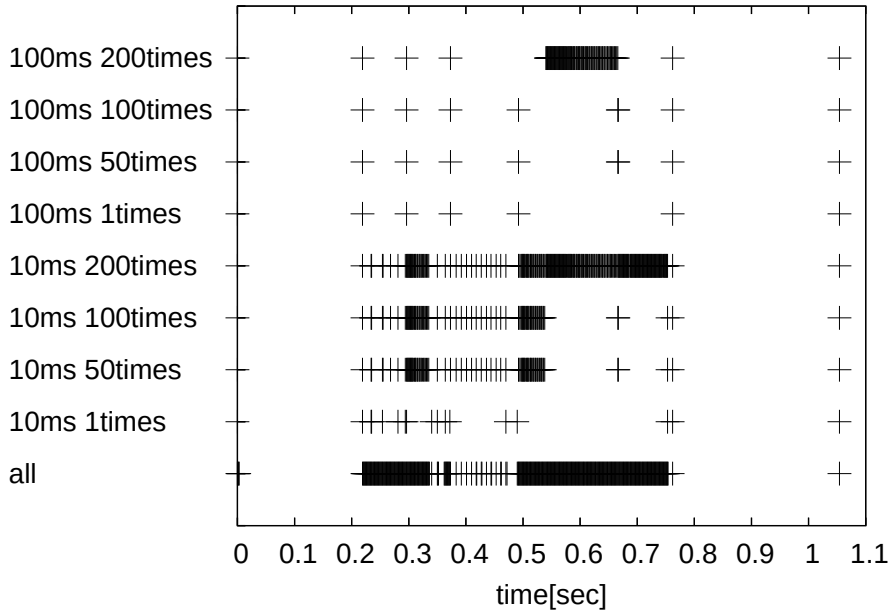


図 5.8: 選択されたメソッド呼び出しの時刻 (6 時間分のグラフ)

5.2.3　選択アルゴリズムのパラメータの影響

4.3.2 で説明したアルゴリズムは 2 つのパラメータを持っている。1 つはスケジューラを呼び出す位置を選択する時間間隔、もう 1 つは出現頻度が何回までのメソッド呼び出しを候補に入れるかである。選択する時間間隔を 10 ms または 100 ms に変えた場合、候補に入れる出現頻度を 1、50、100、200 回までに変えた場合に、選択されたメソッド呼び出しが行われるプロファイリング情報上の時刻を、過去 6 時間分の水位情報のグラフ生成要求と 12 時間分のグラフ生成要求についてプロットしたものが図 5.8 と図 5.9 である。参考のためにすべてのメソッド呼び出しの時刻もプロットしてある。

この図からパラメータによってスケジューラの呼び出され方が大きく変わることが分かる。この中で、時間間隔を 100 ms、出現頻度を 50 回までとした場合には、6 時間分及び 12 時間分の水位情報のグラフ生成要求をした場合の両方で、最初と最後を除いては比較的等間隔に選択されていることが分かる。最初と最後で制御できていないのは、データベースアクセスおよび制御できない javax 内の処理が行われているためである。javax 内の処理で制御ができないのは、コードをアプリケーションに組み込むために利用しているアスペクト指向システムの GluonJ が、javax を含め一部のパッケージへのコード挿入を禁止しているためである。また中間付近

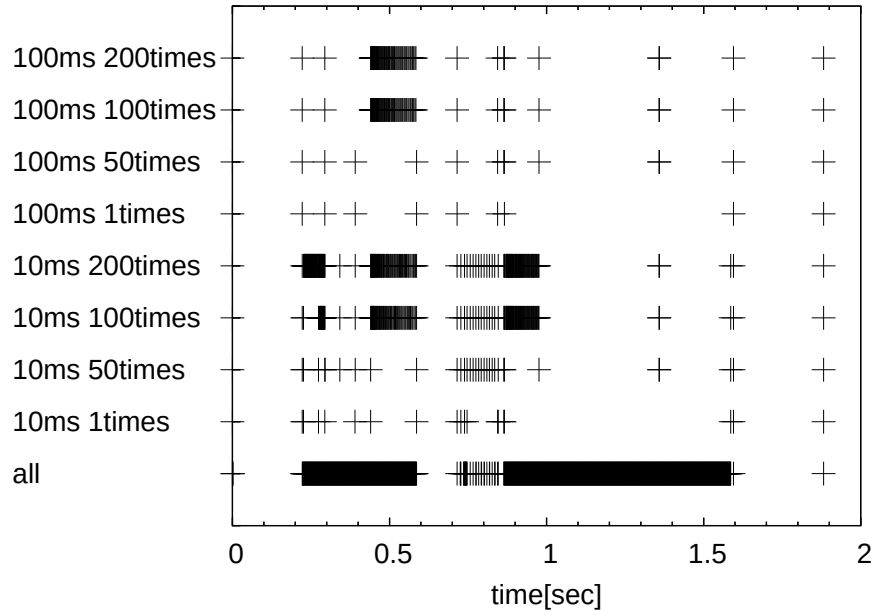
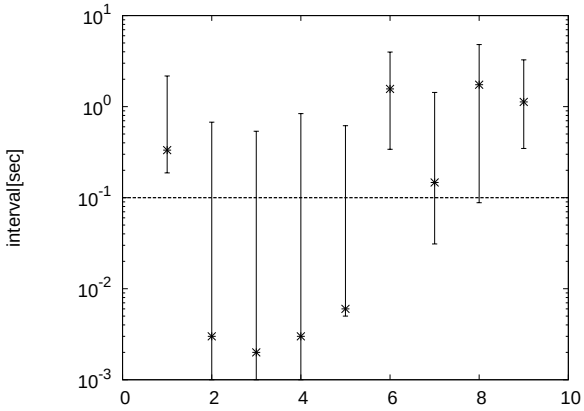


図 5.9: 選択されたメソッド呼び出しの時刻 (12 時間分のグラフ)

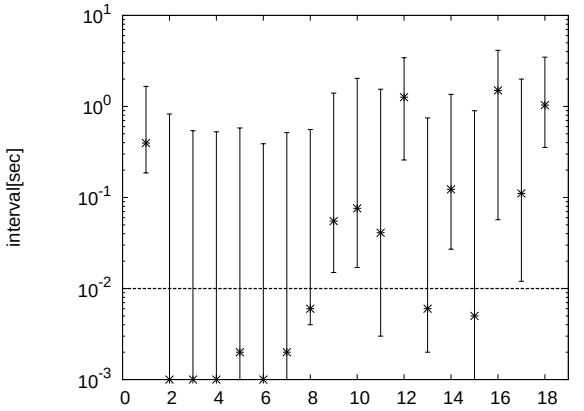
で、メソッド呼び出しが選択されていない区間がある。これは出現頻度が高いメソッドがその区間を占めているためである。

次に、選択したメソッド呼び出しが実際にどの程度の間隔で呼ばれているのかを調べた。選択する時間間隔を 10ms、候補に入れるメソッド呼び出しを出現頻度が 1、50 回までにした場合、及び選択する時間間隔を 100ms、候補に入れる出現頻度を 1 回にした場合ついて調べた。サーバにかけた負荷は、過去 12 時間分の水位情報のグラフ生成要求を常に 20 または 40 送った場合である。3 分間計測を行い、スケジューラが呼び出された時刻を調べ、呼び出しの間隔を、リクエストの処理中に呼び出された順でならべた。それぞれについて最小値、最大値、中央値をプロットしたものを図 5.10 と図 5.11 にまとめた。4.3.2 で説明したアルゴリズムは、単一のリクエストを処理した際の処理経過時刻をもとに行っているので、常に送るリクエストの数が増加すると、その分目標とする時間間隔よりも呼び出し間隔が長くなっていることが伺える。

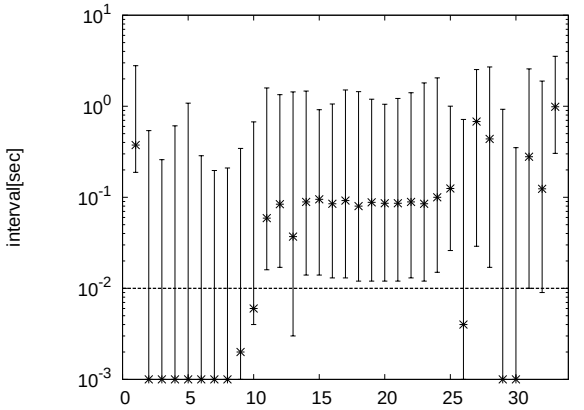
また 4.3.2 で説明したアルゴリズムで、2 つのパラメータを変更した場合にスケジューリングにもたらされる影響をみるため、水位収集にかかる時間、及びグラフ生成を行うスレッドの中の動作中のものの数が 1 まで抑えられるのにかかる時間を複数のバラメータ設定で比較した。実験は、選択する時間間隔を 10ms、候補に入れるメソッド呼び出しを出現頻度が



(a) 間隔 100ms・出現頻度 1 回

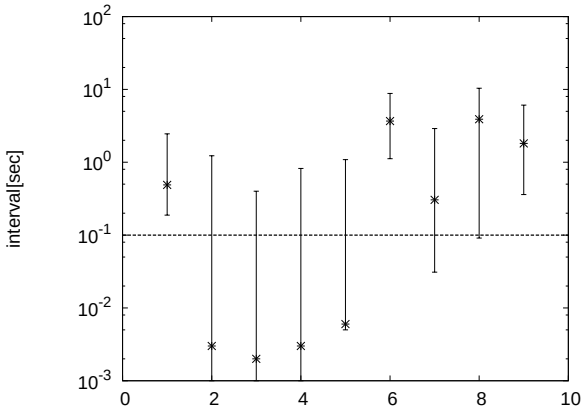


(b) 間隔 10ms・出現頻度 1 回

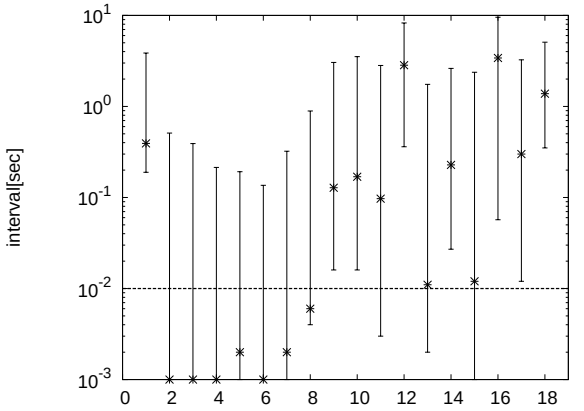


(c) 間隔 10ms・出現頻度 50 回

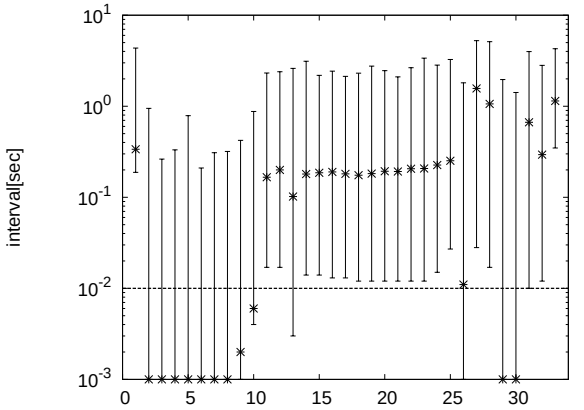
図 5.10: 選択されたメソッド呼び出しの間隔 (常時 20)



(a) 間隔 100ms・出現頻度 1 回



(b) 間隔 10ms・出現頻度 1 回



(c) 間隔 10ms・出現頻度 50 回

図 5.11: 選択されたメソッド呼び出しの間隔 (常時 40)

表 5.4: アプリケーションレベルスケジューリングの効果 (秒)(常時 20)

時間間隔	出現頻度	水位収集にかかる時間	抑えるのにかかる時間
10ms	1 回	4.6	1.0
10ms	50 回	4.9	0.7
100ms	1 回	4.6	1.2
すべて		4.7	0.5

表 5.5: アプリケーションレベルスケジューリングの効果 (秒)(常時 40)

時間間隔	出現頻度	水位収集にかかる時間	抑えるのにかかる時間
10ms	1 回	5.3	1.9
10ms	50 回	5.5	1.7
100ms	1 回	5.7	2.9
すべて		5.0	0.8

1、50 回までにした場合、及び選択する時間間隔を 100ms、候補に入れる出現頻度を 1 回にした場合、及びすべてのメソッド呼び出しを選んだ場合ついて行った。サーバにかけた負荷は、過去 12 時間分の水位情報のグラフ生成要求を常に 20 または 40 送った場合で、実験結果は表 5.4 及び表 5.5 になる。全体的な傾向として、選択する時間間隔を短くし、候補に入れるメソッドの出現頻度を大きくするほど、グラフ生成を行うスレッドで動作中のものの数が 1 に抑えられるまでにかかる時間が短くなり、水位収集にかかる時間も短くなっている。

また、4.3.2 で説明したアルゴリズムで 2 つのパラメータを変更した場合に、オーバーヘッドの変化を見るために、水位収集を行わないように設

表 5.6: グラフ生成処理の処理数 (常時 20)

時間間隔	出現頻度	処理数
制御なし		906
AC		917
100ms	1 回	892
10ms	1 回	883
10ms	50 回	922
すべて		842

表 5.7: グラフ生成処理の処理数 (常時 40)

時間間隔	出現頻度	処理数
制御なし		944
AC		905
100ms	1 回	831
10ms	1 回	890
10ms	50 回	876
すべて		817

定した状態で、複数のパラメータ設定について、グラフ生成処理の処理数を調べた。選択する時間間隔を 10ms、候補に入れる出現頻度を 1、50 回までに変えた場合、及び選択する時間間隔を 100ms、候補に入れる出現頻度を 1 回にした場合、及びすべてを選択した場合ついて調べた。サーバにかけた負荷は、過去 12 時間分の水位情報のグラフ生成要求を常に 20 または 40 送った場合とする。表 5.6 及び表 5.7 に、4 分間での処理数をのせる。パラメータの設定の違いによる処理数への影響は、それほど大きなものはみとめられないが、すべてのメソッド呼び出しなど極端に多くの呼び出しを選択した場合には、処理数が少なくなることが確認された。

したがって、4.3.2 のパラメータは、時間間隔を短く、候補に入れる出現頻度を大きくするほど、短時間で並列処理数を制限することができるが、呼び出し位置が多くなりすぎると、オーバヘッドが増加してしまうことが分かった。

5.3　実験のまとめ

本章では、アプリケーションレベルスケジューリングの有効性を確かめるために、単純な 2 種類のサービスを提供する J2EE アプリケーションと河川水位監視システムに対して、従来のアドミッションコントロールを適用した場合と比較実験を行なった。複数のワークロードを用いて J2EE サーバを過負荷状態にし実験を行なった結果、すべての場合で並列処理数がアドミッションコントロールに比べ短時間で抑えられ、重要な処理が短時間で処理された。また、重要な処理である水位収集処理で欠測が起きないことを確認した。また、スケジューラの呼び出し位置の自動決定アルゴリズム (4.3.2) について、頻繁に呼び出されるようにパラメータを選択することで、短時間で並列処理数を抑えらることができるが、呼び出しが極端に多くなってしまうと、そのオーバーヘッドが高くなってしまうことが分かった。

第6章　まとめ

本稿では、過負荷状態の J2EE アプリケーションサーバにおける QoS 制御の実現方法について述べた。まず、過負荷時のウェブアプリケーションサーバの挙動を詳しく分析し、リクエストの処理に影響を及ぼす主な要因について分析を行なった。その結果、OS のスケジューリングポリシーとスレッドライブラリの実装が大きく影響を及ぼしていることが分かった。そこで、OS やミドルウェアを大幅に変更せずにリクエストの細かなスケジューリング及び QoS 制御を可能にするアプリケーションレベルスケジューリングを提案した。提案方式では、アスペクトを用いてアプリケーションとスケジューラを結合させるため、アプリケーションのコードを修正することなく、スケジューリング機能を追加することができる。提案方式の有効性を示すため、河川水位監視システムという現実的な J2EE アプリケーションに対して、重要度の高い水位収集処理を優先する QoS 制御を提案方式を用いて追加した。その結果、従来のアドミッションコントロールに比べ、より厳密な QoS 制御が行なえることを確認した。

過負荷状態のアプリケーションサーバの挙動の分析では、OS に Solaris と Linux を用い、それ以外はすべて同一の構成、ワークロードで挙動の違いを分析した。OS のスケジューリングポリシー、特にタイムスライスの違いがリクエスト処理に大きな影響を及ぼしていることが分かった。

アプリケーションレベルスケジューリングでは、アプリケーションのレベルで擬似的なスレッドスケジューリングを実現した。これは、OS やミドルウェアの大幅な変更を避け、かつアプリケーションサーバの挙動に大きな影響を与えるスケジューリングを制御するためであった。Java では、外部からのスレッドの操作が推奨されていないため、AOP を用いて様々な箇所からスケジューラを呼び出させることで、擬似的なスレッドスケジューリングを実現した。また、スケジューラを呼び出しを指定する煩雑な作業を軽減するため、スケジューラ呼び出し位置をプロファイリング情報を用いて実行決定できるようにした。

実験では、河川水位監視システムという J2EE アプリケーションに対して QoS 制御の追加を行なった。水位監視システムでは、重要度の低いグラフ生成処理にリクエストが殺到した場合、定期的に行なわれる重要度の高い水位収集処理が滞るという問題があった。そこで、水位収集処理を

優先させるため、水位収集処理を実行している間は、グラフ生成処理の並列処理数を下げるという QoS 制御を追加した。従来のアドミッションコントロールでは、実行中のリクエスト処理を停止させることができないため、並列処理数を即座に制限することができなかったが、提案方式では AOP を用いてスケジューリングコードを組み込んだ様々な箇所でリクエストの処理を停止させられるため、より厳密な並列処理数の制限を行なうことができた。

参考文献

[1] Abdelzaher, T. and Lu, C.: Modeling and Performance Control of Internet Servers (2000).

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

[3] Apache Jakarta Project: Apache JMeter, `http://jakarta.apache.org/jmeter/`.

[4] Cherkasova, L. and Phaal, P.: Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites, *IEEE Trans. Comput.*, Vol. 51, No. 6, pp. 669–685 (2002).

[5] Chiba, S. and Ishikawa, R.: Aspect-Oriented Programming beyond Dependency Injection, *ECOOP 2005*, pp. 121–143 (2005).

[6] Diao, Y., Gandhi, N., Hellerstein, J. L., Parekh, S. and Tilbury, D. M.: Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server, *Proceedings of the Network Operations and Management Symposium*, Florence, Italy (2002).

[7] Douceur, J. R. and Bolosky, W. J.: Progress-based regulation of low-importance processes, *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, New York, NY, USA, ACM Press, pp. 247–260 (1999).

[8] Elnikety, S., Nahum, E., Tracey, J. and Zwaenepoel, W.: A method for transparent admission control and request scheduling in e-commerce web sites, *WWW '04: Proceedings of the 13th international conference on World Wide Web*, New York, NY, USA, ACM Press, pp. 276–286 (2004).

[9] JBoss Group: JBoss Application Server, `http://www.jboss.com/`.

[10] Kanodia, V. and Knightly, E. W.: Ensuring Latency Targets in Multiclass Web Servers, *IEEE Trans. Parallel Distrib. Syst.*, Vol. 14, No. 1, pp. 84–93 (2003).

[11] LeFebvre, W.: CNN.com: Facing a world crisis. Invited talk at USENIX LISA'01 (2001).

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

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

[14] Srinivas, P. B.: Web2K: Bringing QoS to Web Servers (2000).

[15] Tesanovic, A., Amirijoo, M., Amirijoo, M. and Hansson, J.: Empowering configurable QoS management in real-time systems, *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, New York, NY, USA, ACM Press, pp. 39–50 (2005).

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

[17] Voigt, T., Tewari, R., Freimuth, D. and Mehra, A.: Kernel Mechanisms for Service Differentiation in Overloaded Web Servers, *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, pp. 189–202 (2001).

[18] von Behren, R., Condit, J., Zhou, F., Necula, G. C. and Brewer, E.: Capriccio: scalable threads for internet services, *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, ACM Press, pp. 268–281 (2003).

[19] Welsh, M., Culler, D. and Brewer, E.: SEDA: an architecture for well-conditioned, scalable internet services, *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, New York, NY, USA, ACM Press, pp. 230–243 (2001).

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