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

日比野秀章 光来健一 千葉滋

本稿では一般の J2EE サーバ上で動く既存のアプリケーションに対して、きめ細かい QoS 制御を追加する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、アプリケーションのコード書き換えを避けるために、アスペクトを用いてアプリケーションにスケジューラを結合する。また、アプリケーションレベルできめ細かいスケジューリングを実現するために、自動的にアプリケーション内部から適したメソッド呼び出し位置を選択し、定期的にスケジューラを呼び出させる。我々は JBoss 上で開発された河川水位監視システムに対して GluonJ を利用して QoS 制御を追加し、既存のアドミッションコントロールとの比較を行った。

1 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、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。

そこで、我々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

Hideaki Hibino, Kenichi Kourai, Sigeru Chiba, 東京工業大学, Tokyo Institute of Technology

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。

我々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 [2] 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、我々の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 章では提案手法について述べる。4 章で提案手法を適用した河川水位監視システムと制御内容について述べる。5 章では実験について述べ、6 章では関連研究を紹介し、7 章で本稿をまとめる。

2 従来の過負荷制御

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、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。しかしその場合に、アプリケーションのコードの変更は避けたい。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。

アプリケーションのコードを書き換えずに QoS を追加する従来手法としては、OS レベルのスケジューリングポリシの変更 [7] とプロキシサーバの追加 [4] という手法があげられる。capriccio [7] はユーザレベルのスレッドパッケージで、アプリケーションを処理しながら収集した情報を基に、アプリケーションに応じたスレッドスケジューリングを行うことができる。アプリケーションのコードに手を加えることなく、資源が有効活用されるようにスケジューリングが行われるが、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区別して制御をするのは困難である。gatekeeper [4] は、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の機能を備えたプロキシである。多階層モデルのシステムで各階層の間に置くだけでリクエスト処理を制御することができる。しかし、各階層の間でしか制御できないため、時間のかかるリクエスト処理の場合には細かい制御を行うことができない。

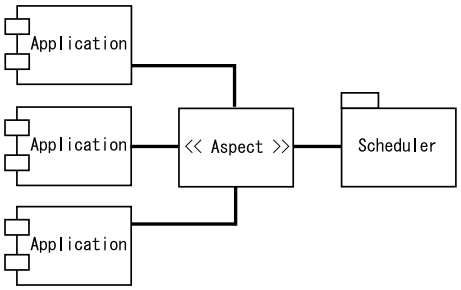

図 1 アプリケーションレベルスケジューリングの構成

3 AOP を用いたアプリケーションレベルスケジューリング

我々は 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。

3.1 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。

3.2 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 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。

```
<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 2 GluonJ によるアスペクトの例

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。

3.3 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。

3.3.1 記述例

以下では、アスペクト指向システムとして我々が用いた 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 2 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。

3.3.2 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、我々は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 Case study

4.1 河川水位監視システム

この章では我々が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。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 3 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。

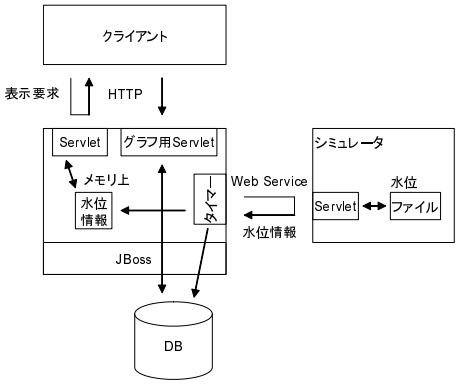

図 3 河川水位監視システムの仕組み

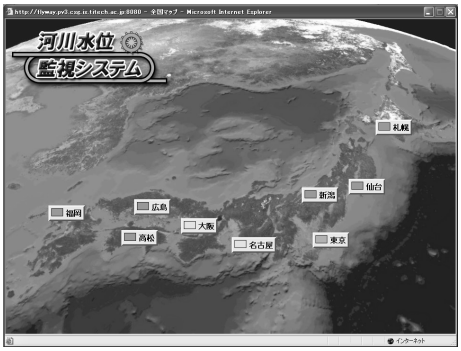

図 4 水位情報の閲覧ページ

一方、J2EE サーバは JBoss [5] 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、メモリ上に置かれた水位情報から図 4 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。

4.2 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、我々は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。我々が作成したアスペクトを図 5 と図 6 に、スケジューラクラスを図 7 に示す。

図 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 6 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 7 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。

5 実験

4.2 のスケジューラクラスとアスペクトを河川水位監視システムに適用し、その効果を確かめる実験を行った。このシステムで使われる 2 つのサーバホストには 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 を用いた。使用した OS は

```
<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)))
    ORAND
      :
  </pointcut>
</pointcut-decl>
```

図 5 アスペクト (pointcut-decl 部)

```
<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 6 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 7 スケジューラクラス (抜粋)

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 [1] を利用した。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 アプリケーションレベルスケジューリングの

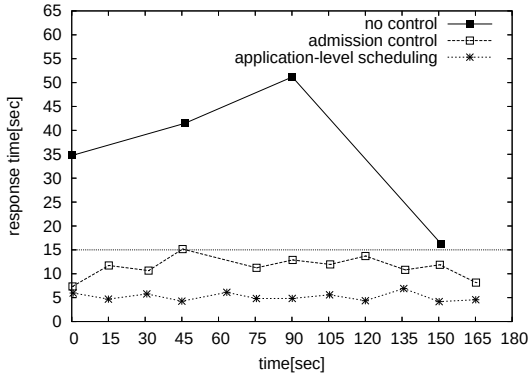

図 8　水位収集の処理時間

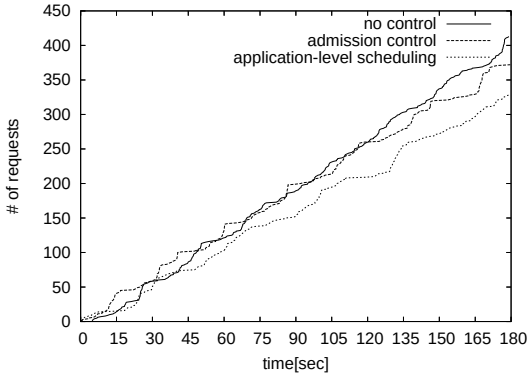

図 9　グラフ生成のリクエスト処理数

効果

我々の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リクエストを常に 40 送っている過負荷な状況で、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 8 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 9 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 10%、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べた。図 10 と図 11 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 10) 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 11) 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

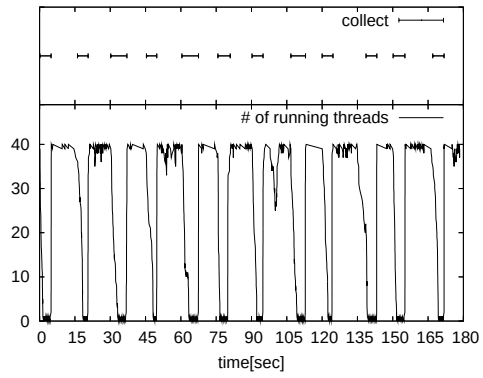

図 10　動作しているスレッド数の変化 (アプリケーションレベルスケジューリング)

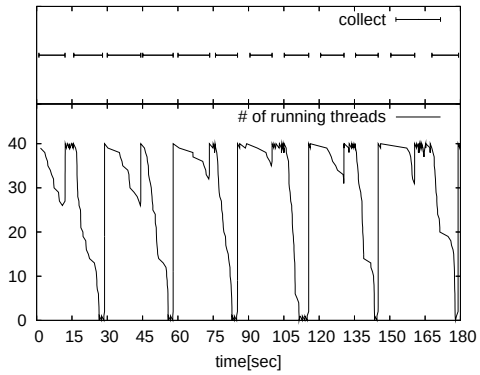

図 11 動作しているスレッド数の変化 (アドミッションコントロール)

表 2 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	20	3.0	1.1
12 時間	20	4.3	1.0
12 時間	40	8.7	2.0

いることが分かる。

5.2 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 20 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、3.3.2 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 1 と表 2 に示す。表 1 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 1 と表 2 を比べてみると、水位収集にかかる時間の差はほぼ並列処理数を 1 にするのにかかる時間の差に等しいことが分かる。

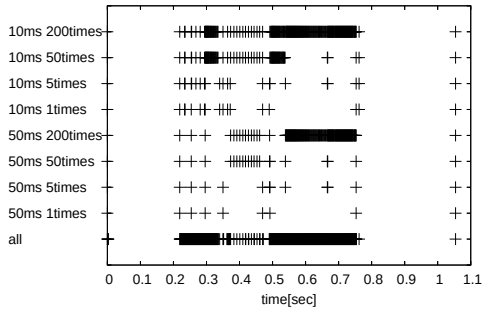

図 12 選択されたメソッド呼び出しの時刻

5.3 選択アルゴリズムのパラメータの影響

3.3.2 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 50 ms に変えた場合、候補に入れる出現頻度を 1、5、50、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時刻をプロットしたものが図 12 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 50 ms、出現頻度を 5 回までとした場合には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。

6 関連研究

Re-QoS [6] 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ができる。アプリケーションの QoS 要求に応じて、QoS アスペ

表 1　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	20	38.4	6.9	5.1
12 時間	20	22.9	7.0	4.7
12 時間	40	35.9	11.4	5.2

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 管理ポリシを変換することができる。アスペクトを用いて QoS ポリシを変更するという点では我々の手法と似ているが、リクエスト処理中にきめ細かいスケジューリングを行う機構は備えていない。

MS Manners [3] 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、重要度の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。我々の手法ではアプリケーションのコードを書き換えることなく、同様の制御を行うことができる。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。このような外部からスレッドを制御する方法は Java では推奨されないため、我々は AOP を利用してアプリケーション内部から制御を行っている。

SEDA [8] では、処理をステージと呼ばれる細かい単位に分割して、ステージ毎にアドミッションコントロールを行うことができる。アプリケーションの実行の途中でスケジューリングが可能になるが、ステージは一連の処理のかたまりであるため、我々の手法とは異なり、アプリケーションの任意の位置でスケジューリングを行うことはできない。また、開発者がステージを意識してアプリケーションを開発しなければならないため、大幅な変更が必要になる。

7　まとめ

本稿では、AOP を用いてアプリケーションに QoS 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した。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するので、アプリケーションのコードに手を加える必要がなく、スケジューラの再利用も可能である。また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させることで、きめ細かい制御が可能となる。我々は、現実的な J2EE アプリケーションである河川水位監視システムに提案手法を適用し、過負荷時に QoS 要求が満たされることを確認した。今後の課題としては、他のアプリケーションにも適用し有効性を確かめる。

参考文献

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

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

[3] 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, 1999, pp. 247–260.

[4] 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, 2004, pp. 276–286.

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

[6] 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, 2005, pp. 39–50.

[7] 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, 2003, pp. 268–281.

[8] 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, 2001, pp. 230–243.